SR 03

Printemps 2007

M. Vayssade

ARCHITECTURES INTERNET

université de technologie de compiègne

SR 03

Printemps 2007

M. Vayssade

ARCHITECTURES INTERNET

université de technologie de compiègne

Université de Technologie de Compiègne

UV SR03 - 2006

Architectures Internet

Michel Vayssade

Ch01 - Architectures Internet - Introduction

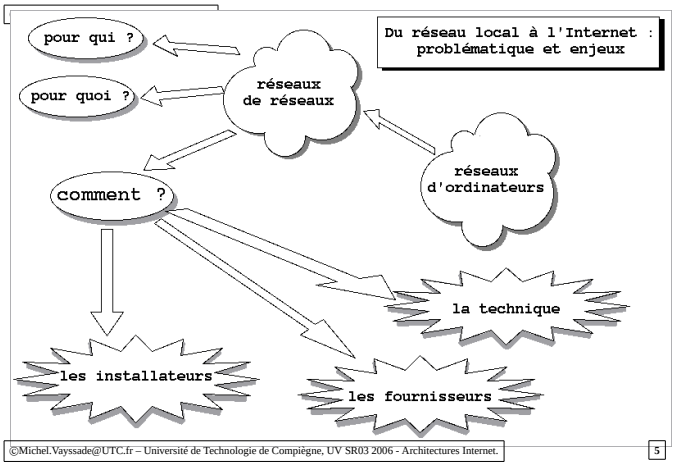
- **De l'ordinateur au réseau, puis au réseau de réseaux**
- **Les enjeux de l'Internet**
- **Les taux d'équipement**
- **Du réseau local à l'Internet : comment ?**
- **Le plan de l'UV**

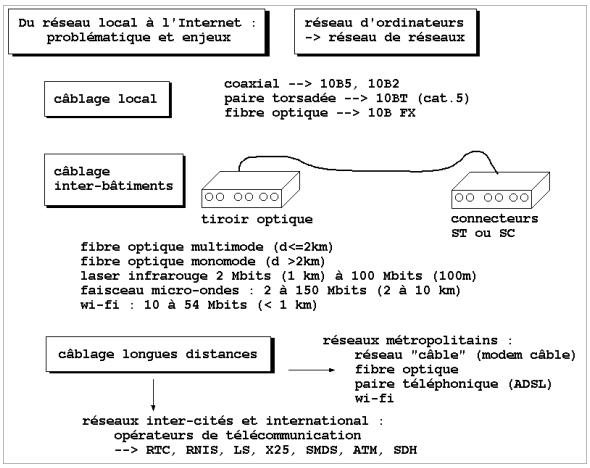
De l'ordinateur au réseau, puis au réseau de réseaux

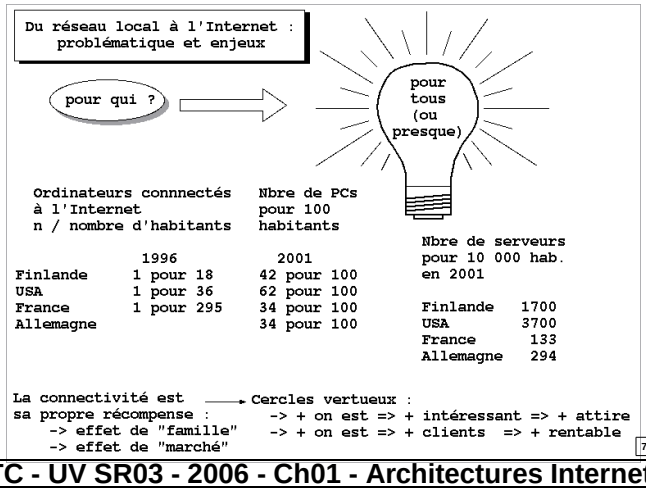
- **réseaux :** débuts au début des années 80
- **1968 :** Vint Cerf démo. liaison d'ordinateurs par réseau devant l'ARPA
- **1969 :** Le Dod américain crée ARPANET, avec 4 ordinateurs
- **1972 :** démo. reliant par ARPANET 40 machines (conf. inter)
- **1972 :** Vint Cerf préside l'InterNet Working Group (INWG) qui a pour mandat de définir un protocole
- **1979 :** USENET relie deux universités américaines.
- **1980 :** Passerelle créée entre le réseau ARPANET et CSNET.

De la diversité à l'unification

- **1981 :** Lancement de BITNET réseau coopératif subventionné par IBM. L'UTC y est reliée. Tous les grands constucteurs ont développé leur propre architecture de réseau (IBM avec SNA, Digital avec Decnet).
- **1982 :** adoption des protocoles TCP et IP. Le mot "Internet" désigne l'ensemble des réseaux utilisant les protocoles TCP/IP.
- **1984 :** introduction du DNS
- **1988 :** 60000 ordinateurs sur l'Internet ; Création du RIPE.
- **1991 :** création au CERN du www et de HTML
- **1993 :** création de Renater en février
- **1994 :** Netscape







Internet : taux d'équipement ; France

France	Taux d'équipement en PCs		Taux d'équipement en connexions Internet	
1er trim.	nbre de foyers en millions	taux %	nbre de foyers en millions	taux %
2001	7,890	32,3 %	4,074	16,7 %
2003	10,478	42,1 %	6,661	26,8 %
2004	11,201	44,8 %	7,303	29,2 %

(source http://www.journaldunet.com)

Internet : taux d'équipement ; Europe

Europe : nombre de micro-ordinateurs pour 100 habitants
(source ITU, décembre 2002)

Suède	56,1
Finlande	42,4
Allemagne	38,2
Royaume-Uni	36,6
France	33,7
Italie	19,5
Moyenne europe	18,3
Grèce	8,1

Internet : nombre d'internautes en Europe

Le nombre d'internautes en Europe en 2003 et 2007
(En pourcentage de la population totale)

Pays	en 2003	en 2007
France	40,9 %	63,8 %
Allemagne	45,9 %	62,4 %
Italie	40,4 %	63,5 %
Royaume-Uni	50,6 %	67,1 %

Aux États-Unis : Juin-août 2004
156,0 millions taux: 73,0 % de la population

(Source : eMarketer Mis à jour le 25/03/2004)

Internet : nombre d'internautes aux États-Unis

États-Unis : pénétration d'Internet à domicile
(Individus de 2 ans et plus ayant une connexion à domicile
en février 2004, en millions)

	accès	total	-> taux
Pop. de 2 ans et +	204,307	272,810	74,9 %

États-Unis : les villes les plus connectées

Villes	Taux de pénétration (foyers)
Seattle	80,6 %
Boston	79,1 %
San Francisco	78,3 %

Paris / Ile-de-France 66,0%

Internet : équipement des ménages en haut débit

Europe : Taux de pénétration du DSL par pays
(Proportion de lignes DSL sur le nombre de lignes téhéphoniques)
Fin sept 2004

Danemark	28 %
Belgique	22 %
France	16 %
Suède	14 %
Europe des 15	14 %
Allemagne	12 %
Royaume-Uni	12 %
Grèce	01 %

(Source : ECTA Mise à jour 31/01/2005)

Internet : Europe : prix du haut débit pour 1Mbit/s

France	28 euros
Pays-Bas	32 euros
Suède	33 euros
Royaume-Uni	40 euros
Allemagne	42 euros
Grèce	100 euros

(Source : Ministère finlandais des Télécommunications
Mise à jour le 04/01/2005)

Internet, Europe : taux de pénétration par pays du haut débit

(Au 1er janvier 2004, nombre de lignes pour 100 habitants, en %)

Danemark	12,7 %	Allemagne	5,6 %
Belgique	12,1 %	Espagne	5,5 %
Pays-Bas	11,5 %	Royaume-Uni	5,3 %
Suède	10,4 %	Portugal	4,9 %
Finlande	8,4 %	Italie	4,7 %
Autriche	8,4 %	Luxembourg	2,8 %
France	6,2 %	Irlande	0,9 %
Total Europe des 15 --> 6,0 %		Grèce	0,1 %

(Source : Commission Européenne, Mise à jour le 27/05/2004)

Internet : part des internautes ayant un accès haut débit

France	38%
Espagne	37%
Pays-Bas	36%
Suède	30%
Suisse	28%
Allemagne	27%
Royaume-Uni	22%
Italie	16%
Etats-Unis	35%
Hong-Kong	82%

(Source : Nielsen//NetRatings Mise à jour le 27/04/2003)

Internet : les accès haut débit en Europe (2005)
➤ forte progression.

Pays	N	P	T
Allemagne	6,9	97 %	18 %
France	6,8	93 %	27 %
Royaume-Uni	6,1	67 %	24 %
Italie	4,7	96 %	20 %
Espagne	3,3	79 %	27 %
Pays-Bas	3,1	58 %	43 %
Belgique	1,7	59 %	38 %
Suède	1,4	64 %	31 %
Suisse	1,3	62 %	39 %
Danemark	1,0	60 %	39 %
Portugal	0,9	44 %	22 %

N = Nombre de foyers connectés à l'Internet haut débit (en millions)

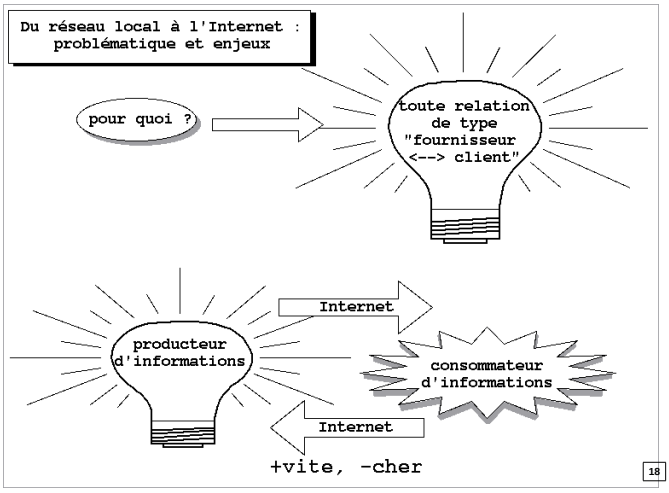
P = Part des accès à l'Internet haut débit réalisés avec l'ADSL

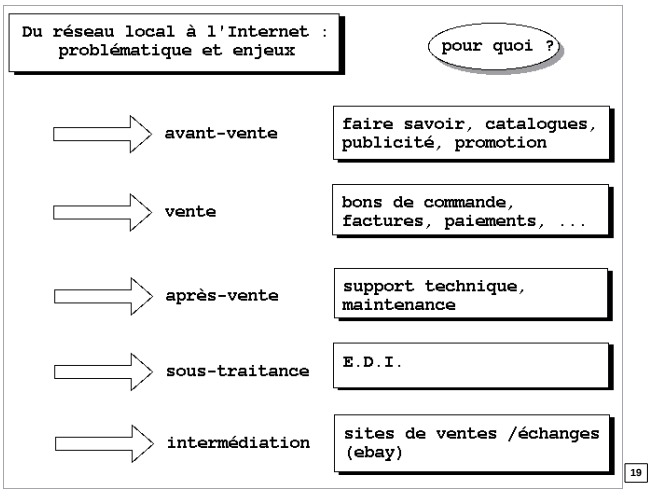
T = Taux de pénétration du haut débit par foyers

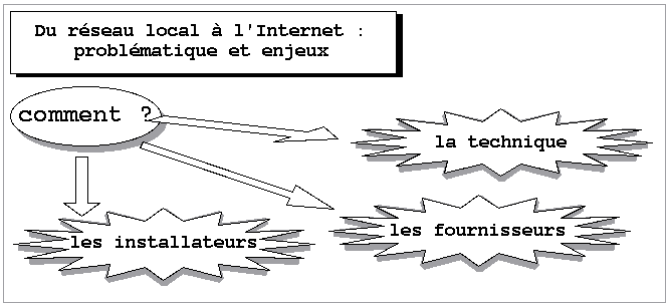
Autriche	0,8	50 %	24 %
Norvège	0,7	86 %	32 %
Finlande	0,7	85 %	30 %
Ireland	0,1	100 %	10 %
Grèce	0,0	-	1 %
Total	39,2	81 %	24 %

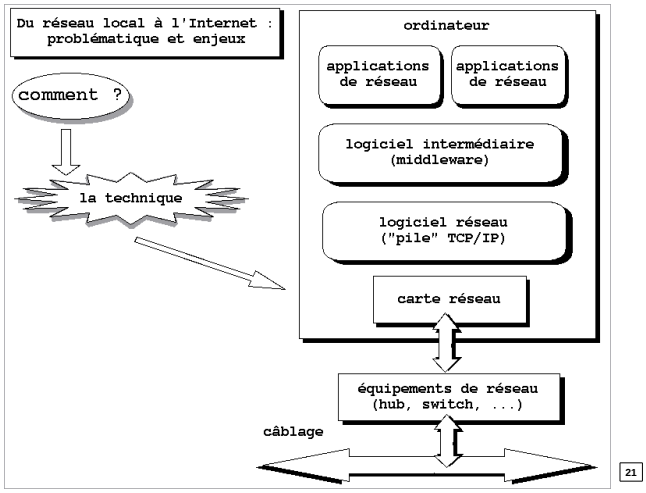
Le taux de pénétration du haut débit en France est passé de 15% fin 2003 à 27% fin 2004 et sans doute **40% fin 2005**. La prévision est de 85% fin 2010.

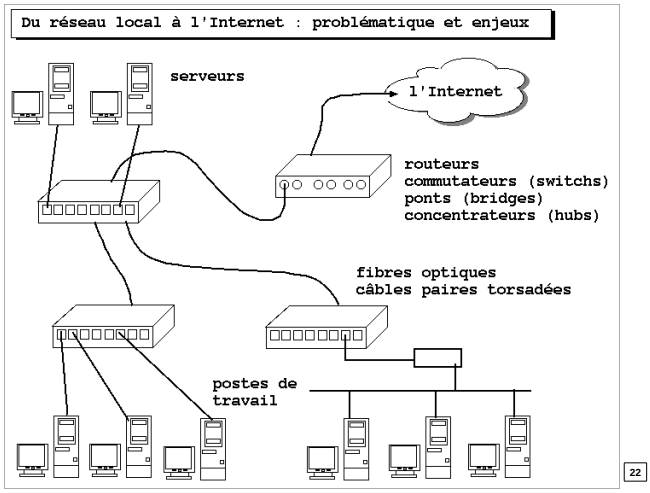
➤ Très supérieur aux États-Unis (en 2005 : "haut-débit" ≈15%, ADSL ≈6%). À noter : la définition du "haut-débit" aux USA = 200kbits, ailleurs 1 Mbits.

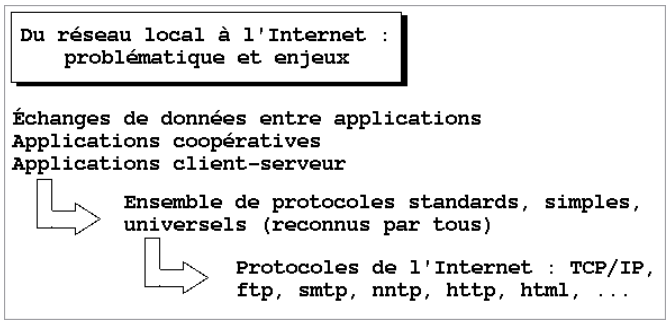


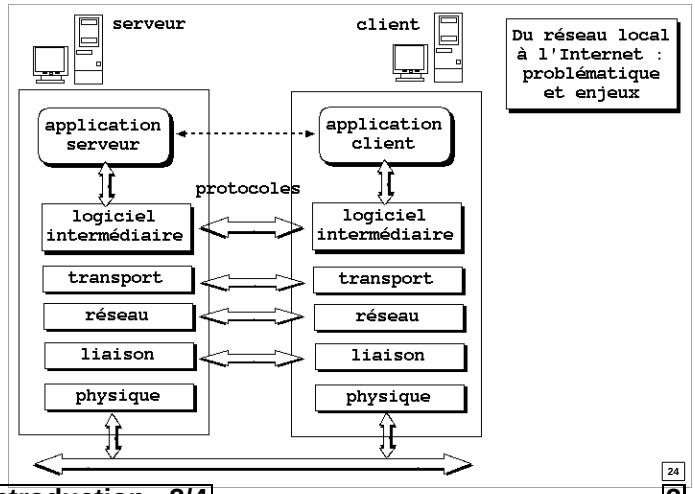


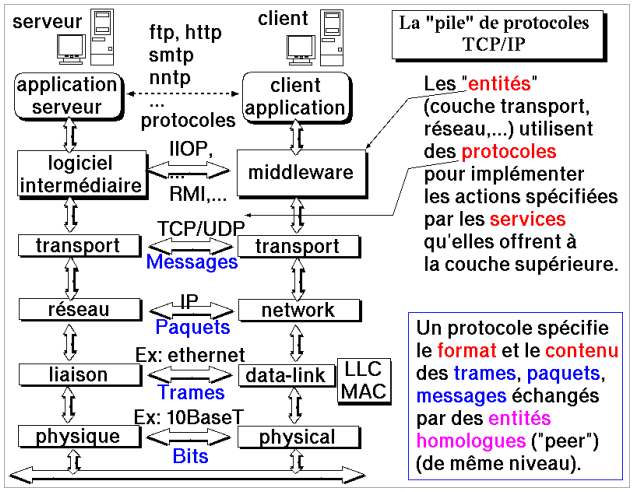












On en déduit le plan de l'UV

1. Enjeux de l'Internet (ce chapitre)
2. Rappels et compléments sur les échanges entre process
3. Échanges entre process sur des machines différentes, modèles mémoire
4. Notion de protocole de communication
5. Introduction au fonctionnement d'un ethernet
6. Concepts de base d'un Internet, introduction à TCP/IP
7. Les protocoles UDP et TCP
8. Communication par RPC

9. Le web et http
10. Les CSS
11. Interactivité sur le client : javascript
12. Interaction entre client et serveur web : CGI, PHP
13. Client-serveur et objets distribués : javaRMI
14. Java, le client-serveur et le web
15. Le système Java
16. Programmation Java
17. Servlets et JSP
18. Client-serveur et objets distribués : CORBA

19. CORBA : exemple horloge ; IDL ; héritage et délégation
20. CORBA : le POA
21. XML et l'Internet, XSLT
22. XMLRPC, services web
23. Les Serveurs d'application Internet
24. Serveurs d'application java : (Servlets, JSP, J2EE)
25. Autre exemple de serveur d'application : ZOPE
26. Le parallélisme et PVM
27. nouveautés du web : schémas, webservices, XUL
28. pki

Ch01 - Architectures Internet - Introduction.
Exemples de questions

➤ À quand peut-on dater le début des réseaux ?
Réponse : diapo 3.

➤ Année d'adption des protocoles TCP/IP et du mot "Internet" ?
Réponse : diapo 4.

➤ Quelles sont les trois grandes classes de câblage de réseaux ?
Réponse : diapo 6

➤ Nombre de personnes ayant une connexion Internet en France (en % de la population) ?
Réponse : diapo 10 et 11 : entre 40 et 70 % suivant les régions.

➤ Dessinez la pile des couches réseaux TCP/IP en indiquant pour chacune le ou les protocoles associés.
Réponse : diapo 24. Ce schéma **doit être su par coeur**. **Point**.

(diapo vide)

Ch01 - Architectures Internet - Introduction.

Fin du chapitre.

Ch02 - Communications Inter-Processus - Files de Messages.

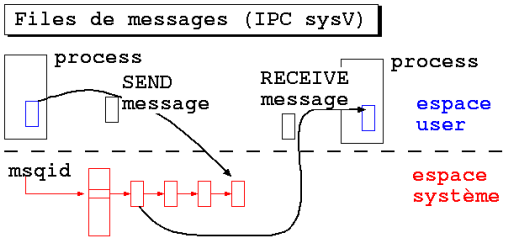
Objectif du chapitre : compléter les méthodes de **communications entre process dans le contexte d'une même machine** déjà vues dans le cours précédent (SR02 - Systèmes d'Exploitation - Chap.02).

➤ On présente ici les "Files de Messages" à cause de l'analogie sémantique entre cet "IPC System V" et les protocoles réseau d'échange de datagrammes.

Les files de messages

La "file de messages" (message queue) est implantée comme une structure dans l'espace système. Ainsi une fois un message déposé dans la file, il est conservé, même si le process émetteur se termine avant que le destinataire n'ai reçu le message.

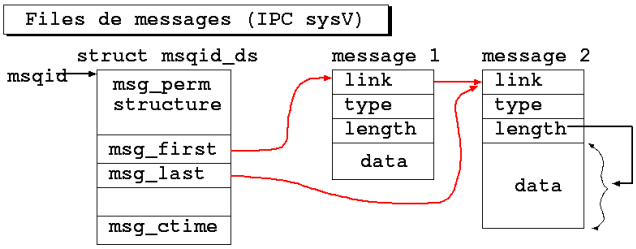
Les opérations d'envoi (send) et de réception (receive) sont toujours à l'initiative des process utilisateurs.




```
MSGOP(2)            Linux Programmer's Manual          MSGOP(2)
NAME msgop - message operations
SYNOPSIS
     #include <sys/types.h>
     #include <sys/ipc.h>
     #include <sys/msg.h>

     int msgsnd(int msqid, struct msgbuf *msgp,
                size_t msgsz, int msgflg);
     ssize_t msgrcv(int msqid, struct msgbuf *msgp,
                    size_t msgsz, long msgtyp, int msgflg);
DESCRIPTION
     To send or receive a message, the calling process
     allocates a structure of the following general form:
```



```
struct msgbuf {
     long mtype;    /* message type, must be > 0 */
     char mtext[1]; /* message data */
};
```



Caractéristiques des files de messages :

- tous les messages sont stockés dans le noyau,
- un int msqid identifie la file ,
- les process lisent ou écrivent dans n'importe quelle queue,
- il n'y a pas de synchronisation écrivain/lecteur nécessaire comme dans le cas des PIPES ou des FIFOs,
- un process peut écrire un ou plusieurs messages, un autre les lire plus tard, à son rythme.
- rappel : le champ "type" contenu dans un message est obligatoirement >0.


```
struct msgbuf {  long mtype;
                 char mtext[n];}
struct msgbuf buf;

id = msgget(MA_cle, PROT|IPC_CREAT);
 \_______
  |      |
  |     msgsnd (id, &buf, lng, flag);
  |_______________
  |               |
  |     msgrcv (id, &buf, lng, type, flag);
  |_____________
                |
        msgctl(id, IPC_RMID,0);
```



```
argument "type" dans msgrcv :
  type de message demandé
  si type=0 : demande le 1er message de la file
  si type>0 : demande le 1er message de ce type
  si type<0 : demande le 1er message dont le type est
              le plus petit qui soit aussi <= au type
              demandé

rappel : le type contenu dans un message est
         obligatoirement >0.
```


Ch02 - Communications Inter-Process - Files de Messages. Exemples de questions

➤ Que devient le message juste après le "send" ?
Réponse : diapo 2.

➤ Pourquoi l'utilisation des files de messages n'implique pas de synchronisation entre émetteur et receveur ? Réponse : diapo 2.

➤ À quoi peut servir le champ "type" contenu dans la structure "message" ? Réponse : diapo 8.

Ch02 - Communications Inter-Process - Files de Messages.

Fin du chapitre.

Ch03 - Architectures Internet - Les sockets

- **Origine, présentation et conception des sockets**
- **Les types de sockets**
- **Exemple de serveur et de client UDP**
- **Modèles mémoires différents et échanges de données**

Origine, présentation et conception des sockets

Les sockets sont apparus avec BSD 4.2 et se sont stabilisés avec BSD 4.3. Leur utilité a fait qu'ils sont maintenant disponible sur tous les systèmes unix et sur de nombreux autres systèmes.

Ce sont en effet, le support privilégié de la plupart des communications entre process sur l'Internet.

C'est une interface de communication générale permettant de créer des applications distribuées.

Elle crèe un canal unique bidirectionnel (full duplex), contrairement aux "pipes" unix qui ne fonctionnent que dans un seul sens.

Les sockets couvrent deux domaines et sont de deux types différents.

Domaines :

- domaine AF_UNIX : permet à des process de la même machine de communiquer entre eux ;
- domaine AF_INET : permet à des process situés sur deux machines différentes d'un même internet de communiquer. Ceci suppose que les deux machines communiquent par la suite de protocoles réseau TCP/IP.

Un domaine défini donc l'ensemble des autres sockets avec lesquels un process donné pourra communiquer.

Types :

le type du socket défini les propriétés de la communication. Les deux types principaux sont :

- le type datagramme, permettant l'échange de messages complets et structurés, sans négociation préalable (communication sans connexion) ;
- le type connecté, permettant l'échange de flux de données sur un circuit virtuel préalablement établi.

Objectifs de conception de l'interface socket

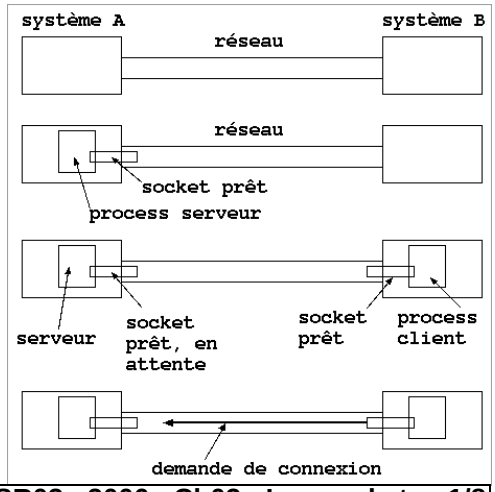
Les sockets ont été conçus pour permettre à des process de communiquer à travers un réseau tout en respectant :

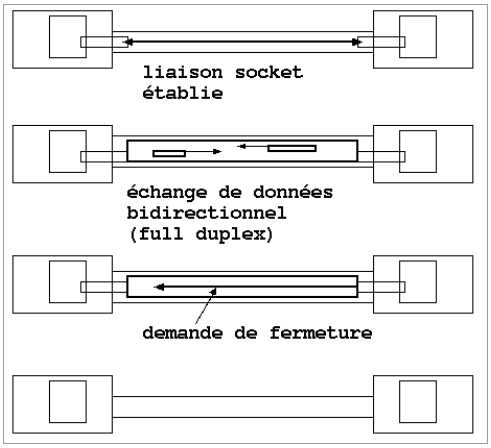
- **transparence :** que la communication soit identique si les process sont sur la même machine ou non ;
- **efficacité :** implantés comme bibliothèque système incluse dans le noyau, ils ont accès direct au driver de l'interface réseau évitant ainsi une double recopie du buffer de données et ils travaillent comme un appel système depuis le process utilisateur, dans le contexte de celui-ci, sans obliger à un changement de contexte comme cela aurait été le cas s'ils avaient été implantés sous forme de process indépendant ;
- **compatibilité :** sockets vus des process comme descripteurs de fichiers, => permet opérations habituelles de redirection E/S (dup()) ;

Mode client-serveur et établissement d'une communication

La caractéristique la plus fréquente des connexions en mode "client-serveur" est d'être asymétrique :

- **d'abord**, un process **serveur** se prépare ; il se mets en écoute sur un **port** de communication. Ce n'est encore qu'une "moitié" de la liaison. C'est le robinet sur lequel le tuyau n'a pas encore été branché.
- **ensuite**, un process **client** va envoyer à ce port serveur déja existant une demande de connexion.
- **alors**, la connexion est établie et les données peuvent circuler dans les deux sens.





Les types de sockets

Les deux types principaux de sockets correspondent aux protocoles **sans connexion** et aux protocoles **orientés connexion** :

- protocoles **sans connexion** : ("connectionless") il y a échange de messages autonomes, complets et structurés ; chaque envoi doit contenir le destinataire ; ceci peut se comparer à l'envoi de lettres par la poste.
- protocoles **orientés connexion** : ("connection oriented") après établissement de la connexion (circuit virtuel), tous les envois sont implicitement pour le même destinatire ; ceci peut se comparer au téléphone.

1. **sockets SOCK_DGRAM** : envoi de datagrammes (messages), mode non connecté ; protocole **UDP** au-dessus de **IP** :
   - messages de taille bornée,
   - préserve les frontières de message,
   - pas de garantie de remise ("best effort"),
   - pas de garantie de remise dans le même ordre que l'envoi ;
2. **sockets SOCK_STREAM** : flot de données, mode connecté, protocole **TCP** au-dessus de **IP** :
   - transfert fiab le (pas de perte ni d'altération de données),
   - données délivrées dans l'ordre d'envoi,
   - pas de duplication,
   - supporte la notion de messages urgents ("out-of-band") pouvant être lus avant les données "normales" ;

3. **sockets SOCK_RAW** : permet d'accéder au protocole de plus bas niveau (IP) pour construire d'autres protocoles. Réservé au mode superviseur.
4. **sockets SOCK_SEQPACKET** : paquets avec préservation des frontières, délivrés de façon fiab le et en séquence (dans l'ordre). Cité pour la complétude, était utilisé dans le protocole Decnet.

Caractéristiques d'un socket

Vu "de l'intérieur" du process utilisateur, un socket est un descripteur de fichiers. On peut ainsi rediriger vers un socket les E/S standard d'un programme développé dans un cadre local.

Une autre conséquence est que les sockets sont hérités par le fils lors d'un fork.

Un socket est créé par la primitive **sd = socket()**. La valeur de retour étant le descripteur sur lequel on va faire les opérations de lecture et d'écriture.

La différence avec un fichier est qu'il est ensuite possible (et nécessaire !) **d'attacher** au socket une adresse du domaine auquel il appartient. Ceci se fait avec la primitive **bind()**. Cette adresse est communément appelée **numéro de port**.

Création et attachement d'un socket

```
NAME
       socket - create an endpoint for communication
SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>

       int socket(int domain, int type, int protocol);
```

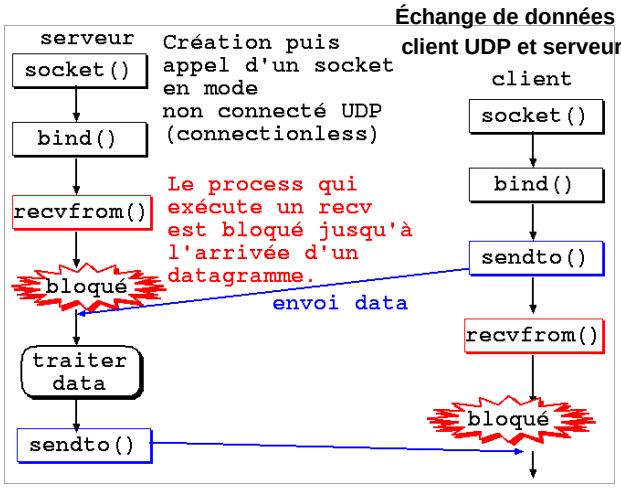
Le troisième paramètre est souvent donné à 0, car souvent il n'y a qu'un seul type de protocole disponible (par exemple pour un SOCK_DGRAM dans le domaine AF_INET, il n'y a que UDP, et pour SOCK_STREAM que TCP).


```
Exemple:   sd = socket (AF_INET, SOCK_DGRAM, 0);
     ou:   sd = socket (AF_INET, SOCK_STREAM, 0);
```

Attachement (bind)

```
$ man 2 bind
NAME
     bind - bind a name to a socket
SYNOPSIS
     #include <sys/types.h>
     #include <sys/socket.h>
     int bind(int sockfd, struct sockaddr *my_addr,
              socklen_t addrlen);
(addrlen est la longueur de la structure my_addr)
```


Échange de données entre client UDP et serveur UDP




```
$ man 2 send
$ man 2 recv

SEND(2)          Linux Programmer's Manual          SEND(2)
NAME
     send, sendto, sendmsg - send a message from a socket
SYNOPSIS
   #include <sys/types.h>
   #include <sys/socket.h>
   ssize_t send(int s, const void *msg, size_t len,
                int flags);
   ssize_t  sendto(int s, const void *msg, size_t len,
     int flags, const struct sockaddr *to, socklen_t tolen);
   ssize_t sendmsg(int s, const struct msghdr *msg,
                int flags);
```

```
DESCRIPTION
   Send, sendto, and sendmsg are used to transmit a message
   to another socket. Send may be used only when the socket
   is in a connected state, while sendto and sendmsg may be
   used at any time.

   The address of the target is given by to with tolen
   specifying  its size. The length of the message is given
   by len. If the message is too long to pass atomically
   through the underlying protocol, the  error EMSGSIZE is
   returned, and the message is not transmitted.
```

L'émetteur peut être soit le client, soit le serveur. La seule contrainte est qu'ils doivent être programmés pour qu'il y ait autant de recv() exécutés par l'un que de send() exécutés par l'autre.


```
Exemples :
        sendto(sa, bufa, nbufa, flag, &to, lento)
        recvfrom( sb, bufb, nbufb, flag, &from, lenfrom)
ou:
        send(sa, bufa, nbufa, flag)
        recv(sb, bufb, nbufb, flag)
```

Remarque : le "man" précide que "Send may be used only when the socket is in a connected state". Or, on est ici dans un contexte UDP, c'est-à-dire sans connexion. Qu'en est-il ?

En fait il s'agit d'une facilité de programmation : il est possible, afin d'éviter de remettre dans chaque appel "sendto()" l'adresse du destinataire, si c'est toujours le même, de stocker cette adresse dans la structure socket. On peut alors utiliser simplement send() vers ce

destinataire implicite. Il s'agit d'une **pseudo-**connexion dont l'autre entité n'a pas connaissance. C'est juste une facilité de programmation locale.

On peut ainsi faire un send() vers un process qui lui-même est uniquement programmé avec des recvfrom(), et inversement.

Flags :

- **MSG_OOB** send or recv Out-of-Band data
- **MSG_PEEK** permet de "regarder" les data à lire sans que le système ne les enlève du buffer d'arrivée.

Exemple de serveur et de client UDP

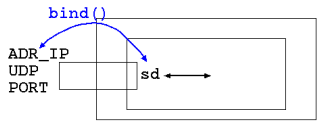
```
        Un serveur UDP  et  un client UDP
(1)  le serveur se met en attente sur un port UDP,
(2)  le client crèe son socket local,
(3)  le client envoie un message sur le host
     et le port UDP du serveur,
(4)  à réception du message, le serveur récupère
     l'adresse (host,port) de l'émetteur et traite
     le message, puis envoie une réponse.
```



```
/* udpser.c  */                          /* serveur UDP */
/* UTC UV SR03 - (c) Michel.Vayssade@utc.fr
linux:   gcc -o udpser udpser.c
solaris: gcc -o udpser udpser.c -lsocket -lnsl */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <errno.h>

#define PORT 4677 /* port du serveur */
int sd,i,n;
int *fromlen;
struct sockaddr_in mon_s;
struct sockaddr_in son_s;

typedef struct messages {
  char mtext[50];
} message;
message *mess;

void affiche_errno() { printf("\nerrno=%d",errno); }
```

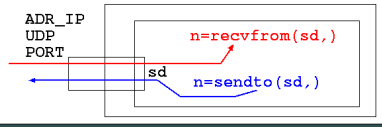
21

```
main(){                                  /* serveur UDP */

sd=socket(PF_INET,SOCK_DGRAM ,0);

if(sd==-1)
   perror("udpser:err socket"),affiche_errno(),exit(1);

bzero(&mon_s,sizeof(mon_s));

/* accept input from any interface */
mon_s.sin_addr.s_addr = INADDR_ANY;
mon_s.sin_family = PF_INET; /* sur IP */
mon_s.sin_port = htons(PORT); /* port du serveur */

n=bind(sd,(struct sockaddr*)&mon_s, sizeof(mon_s));

if(n==-1) perror("udpser:err bind"),affiche_errno(),exit(1);
mess=(message*)malloc(sizeof(message));
```



22

```
                                         /* serveur UDP */
fromlen=(int*)malloc(sizeof(int));
*fromlen=sizeof(son_s);

n=recvfrom(sd,mess,sizeof(message),0,
   (struct sockaddr*)(&son_s),fromlen);

if (n==-1)
    perror("udpser:err recvfrom"),affiche_errno(),exit(1);
printf("Message recu= %s\n",mess->mtext);

strcpy(mess->mtext,"serveur repond et envoie un message");
printf("serveur envoi message\n");

n=sendto(sd,mess,sizeof(message),0,
        (struct sockaddr*)(&son_s),sizeof(son_s));

if (n==-1)
    perror("udpser: err sendto "),affiche_errno(),exit(1);
}
```



23

```
/* udpcli.c   */                          /* client UDP */
/* UTC UV SR03 - (c) Michel.Vayssade@utc.fr
 linux:   gcc -o udpcli udpcli.c
 solaris: gcc -o udpcli udpcli.c -lsocket -lnsl */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <errno.h>

#define PORT 4677 /* port du serveur */
int sd,i,n;
int *fromlen;
struct sockaddr_in mon_s;
struct sockaddr_in son_s;
struct hostent *hs;

typedef struct messages {
  char mtext[50];
} message;
message *mess;

void affiche_errno() { printf("\nerrno=%d",errno); }
```

24

```
main(){                                          /* client UDP */
hs=gethostbyname("localhost");

sd=socket(PF_INET,SOCK_DGRAM ,0);

if(sd==-1)perror("udpcli:err socket"),affiche_errno(),exit(1);
bzero(&son_s,sizeof(son_s));    /* adr IP serveur */
bcopy(hs->h_addr,&son_s.sin_addr,hs->h_length);
son_s.sin_family = PF_INET;  /* sur IP */
son_s.sin_port = htons(PORT); /* port du serveur */
bzero(&mon_s,sizeof(mon_s));
mon_s.sin_addr.s_addr = INADDR_ANY;
mon_s.sin_family = PF_INET;  /* sur IP */
/* port client choisi par le système : */
mon_s.sin_port = htons(0);
/* si on impose le même port que pour le serveur,
   et que l'on est sur la même machine => déjà utilisé */

n=bind(sd,(struct sockaddr*)&mon_s,sizeof(mon_s));
if (n==-1) perror("udpcli:err bind"),affiche_errno(),exit(1);
```

client — bind() — sd — ADR_IP UDP PORT client — ADR_IP UDP PORT serveur — sd — serveur

25

```
mess=(message*)malloc(sizeof(message));      /* client UDP */
strcpy(mess->mtext,"j'envoie un message");

n=sendto(sd,mess,sizeof(message),0,
        (struct sockaddr*)&son_s,sizeof(mon_s));

if(n==-1)perror("udpcli:err sendto"),affiche_errno(),exit(1);
fromlen=(int*)malloc(sizeof(int));
*fromlen=sizeof(mon_s);
printf("lire message\n");

n=recvfrom(sd,mess,sizeof(message),0,
          (struct sockaddr*)&mon_s,fromlen);

if (n==-1)
    perror("udpcli: err recvfrom "),affiche_errno(),exit(1);
printf("Message recu= %s\n",mess->mtext);
}
```

client — n=sendto(sd,) — sd — n=recvfrom(sd,) — ADR_IP UDP PORT client — ADR_IP UDP PORT serveur — serveur — n=recvfrom — sd — n=sendto

26

client UDP — ADR_IP client — ADR_IP serveur — serveur UDP

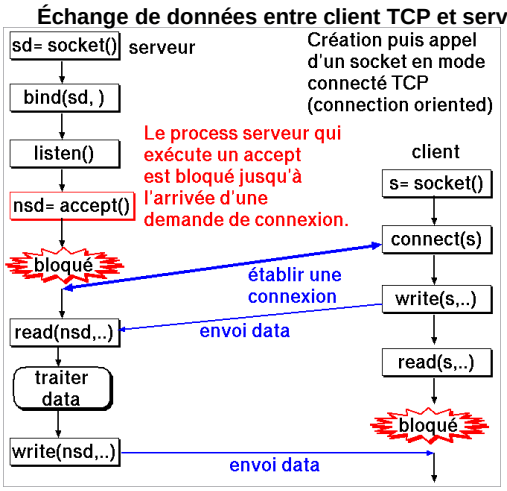
client — sd=socket() — sd — n=bind(sd,) — ADR_IP UDP PORT client — ADR_IP UDP PORT serveur — serveur — sd=socket() — sd — n=bind()

serveur — n=recvfrom — bloqué — sd

client — n=sendto(sd,) — sd — n=recvfrom(sd,) — ADR_IP UDP PORT client — serveur — n=recvfrom — sd — n=sendto

27

Échange de données entre client TCP et serveur TCP



28

Utilisation :

```
#include <sys/types.h>
#include <sys/socket.h>

sd = socket(AF_INET, SOCK_STREAM, 0);
|              |          |
descripteur    domaine    type
=-1 si erreur

#define PORT 0x1234
#include <netinet/in.h>   in: domaine internet
struct sockaddr_in mon_soc;
bzero(&mon_soc, sizeof(mon_soc)); /* init mon_soc */
mon_soc.sin_family = AF_INET;
```

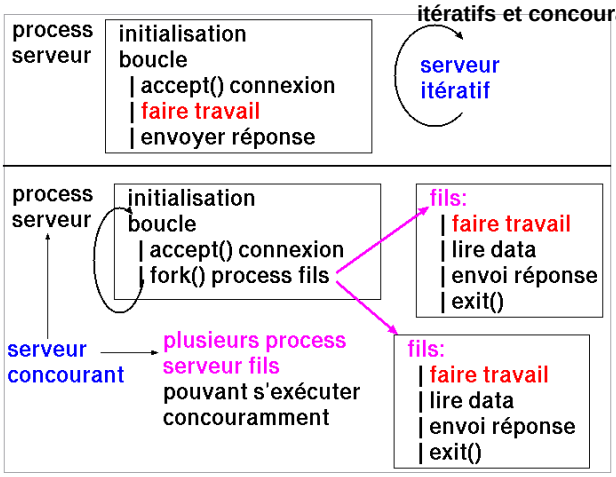


```
mon_soc.sin_port = htons(PORT);

bind(sd, &mon_soc, sizeof(mon_soc));
/* bind() donne un identifiant (le numéro de port
   dans le domaine AF_INET) au socket pour que les
   clients puissent lui envoyer une demande de
   connexion en utilisant cet identifiant */

listen(sd, 5);
/* listen indique au système que l'on va se mettre
   en attente de demandes de connexion */

newsd = accept(sd, 0, 0);
/* on se met en attente d'arrivées de demandes de
```



```
connexion de clients qui exécutent l'appel connect()
Cet appel est BLOQUANT. On en ressort seulement si
un client appelle le serveur.
accept() renvoie un nouveau descripteur sur lequel
se feront tous les échanges de données avec ce client.
Le descripteur initial est réservé à l'écoute des
demandes de connexion.  */
```


Différents types de serveurs TCP : itératifs et concourants



32

- **itératifs :** si le travail à faire est petit et prévisible, cette méthode est plus rapide et moins coûteuse.
- **concourants :** si le travail à chaque connexion est plus long ou non prévisible, il y a un risque de perte de demandes de connexion dans le mode itératif. On crée alors un process fils pour faire le travail et le serveur principal reboucle aussitôt sur le accept().

Exemple de serveur et de client TCP

```
        Un serveur TCP  et  un client TCP
(1)  le serveur se met en attente sur un port TCP
(2)  le client se connecte au serveur
(3)  le serveur accepte la connexion sur un nouveau socket newsd
(4)  client boucle sur envoi "n" messages de tailles 8, 16,...
(5)  le serveur boucle sur la lecture du socket,
     pour récupérer des messages complets, de 8,16,32, il doit
     implanter une boucle (while(attendu)) pour chaque message
     car TCP mets tous les messages à la file, sans frontière.
```



```
/*  tcpser.c  */                          /* serveur TCP */
/* UTC UV SR03 - (c) Michel.Vayssade@utc.fr
 linux:        gcc -o tcpser tcpser.c
- créer socket SOCK_STREAM
- init structure sockaddr
- bind socket to PORT
- listen socket: accepte demandes connexions
- accept une connexion demandee par un client
- boucle:  . lire données sur le socket
           . IF valeur[0]=-1 fin de boucle, FIN programme
           . afficher qques valeurs lues sur le socket    */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <signal.h>
#include <errno.h>
#define NBENREG  10  /* transferer NBENREG messages */
#define ENREGSIZE 8  /* de longueurs 8,16,24,32,40,..*/
#define END_OF_SEND -1
#define SLEEPTIME 0
#define PORT  0x2001
union type_uni { char txt[ENREGSIZE*NBENREG];
   double valeur[NBENREG];  } msg_src;
static int nbErreurTotal = 0 , recu , attendu , fois ;
void affiche_errno() { printf("\nerrno=%d",errno); }
```

35

```
main() {                                  /* serveur TCP */
        struct  sockaddr_in     sin;
        int     sd, i, j, newsd, count, nbInfo , d1=0 ;
        printf("\nser - init socket");
/*- socket SOCK_STREAM ----------------------------------*/
        if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
                perror("\nser: error init socket"),exit(1);
/*- init structure sockaddr ------------------------------*/
        bzero(&sin, sizeof(sin));
        sin.sin_family = AF_INET;
        sin.sin_port   = htons(PORT);
/*- bind socket to PORT ----------------------------------*/
        if(bind(sd, (struct sockaddr*)&sin,sizeof(sin))==-1)
                perror("ser: error bind"), exit(1);
/*- listen socket: accepte demandes connexions -----------*/
        if (listen(sd, 5) == -1)
                perror("ser: error listen"), exit(1);
        printf("\nAttachement reussi- Attente connexion.\n");
/*- accept une connexion demandee par un client -----*/
        if ((newsd = accept(sd, 0, 0)) == -1)
                perror("ser: error accept"), exit(1);
        printf("\nConnexion etablie\n");
/*- raz buffer lecture -----------------------------------*/
        for ( i=0 ; i<NBENREG ; i++)
                msg_src.valeur[i] = 0;
```

36

```
/*- BOUCLE - lecture messages -------- serveur TCP -------*/
        for ( nbInfo=1 ; nbInfo<= NBENREG+5 ; nbInfo++)
    {
        sleep(1);
        attendu = nbInfo*ENREGSIZE; recu = 0; fois = 0;
        while (attendu) {
        count = recv(newsd, &msg_src.txt[recu],attendu,0);
        if ((int)msg_src.valeur[0] == END_OF_SEND)
        {       printf("\nRecu END_OF_SEND.");
        printf("\n mess coupes= %6d. d1= %5d. Exit\n",
                nbErreurTotal,d1);
                 exit(0); }
        if (count==-1) perror("ser: error recv"),exit(1);
        recu = recu + count; attendu= attendu - count;
        fois = fois +1;
        }
            if (fois=1) d1= nbInfo; else nbErreurTotal++ ;
printf(\
"\nmsg # %4d,errno=%5d,recu= %4d,pre=%6d der=%6d en %3d fois",
nbInfo,errno,recu,
(int)msg_src.valeur[0],(int)msg_src.valeur[nbInfo-1],fois);
    }
printf("\n Fin bcle lecture. Mess coupes=%6d. d1=%5d. Exit\n",
  nbErreurTotal,d1);
}
```

37

```
/*  tcpcli.c  */                          /* client TCP */
.............mêmes déclarations que serveur
main() {
      struct  sockaddr_in     sin;
      struct  hostent         *hp;
      int     sd, i, j, nbInfo, result, count;

      errno = -1;
/*- lire adresse destinataire dans "/etc/hosts" ----------*/
      if ((hp = gethostbyname("localhost")) == 0)
          perror("cli: gethostbyname"), exit(1);
/*- init structure sockaddr ------------------------------*/
      bzero(&sin, sizeof(sin));
      bcopy(hp->h_addr, &sin.sin_addr, hp->h_length);
      sin.sin_family = hp->h_addrtype;
      sin.sin_port   = htons(PORT);
/*- ouvrir socket ----------------------------------------*/
      sd = socket(AF_INET, SOCK_STREAM, 0);
      affiche_errno();
      if (sd == -1) perror("cli: socket"), exit(1);
/*- connect sur PORT -------------------------------------*/
      result=connect(sd,(struct sockaddr*)&sin,sizeof(sin));
      affiche_errno();
      if (result == -1) perror("cli: connect"), exit(1);
      printf("\nConnexion etablie. Envoi messages.");
```

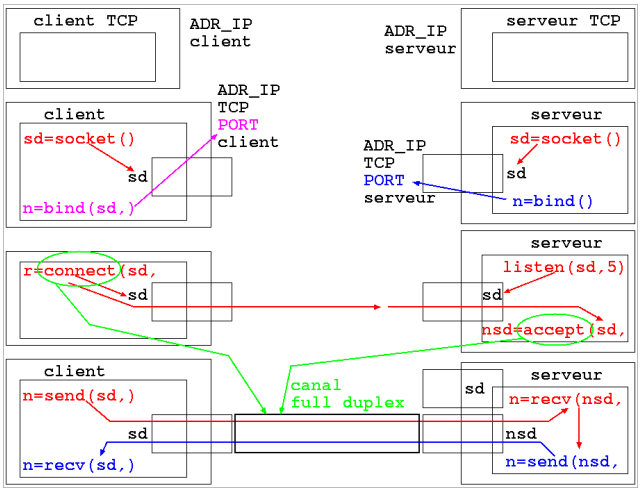
38

```
                                          /* client TCP */
/*- BOUCLE envoi successif d'une table de 8,16,.. octets -*/
     for ( nbInfo=1 ; nbInfo<=NBENREG ; nbInfo++)
     {
         init_table(nbInfo);
         count = send(sd, msg_src.txt,nbInfo*ENREGSIZE, 0);
         if (count == -1) perror("cli: send"), exit(1);
         printf("\nmsg data # %4d errno=%4d envoi %5d
         octets pre=%6d der=%6d",nbInfo,errno,
         nbInfo*ENREGSIZE,(int)msg_src.valeur[0],
         (int)msg_src.valeur[nbInfo-1]);
     }
/*- Envoi message terminaison ----------------------------*/
     nbInfo = 1;
     msg_src.valeur[0] = END_OF_SEND;
     for ( i=1 ; i<=3 ; i++)
     {  count = send(sd, msg_src.txt,nbInfo*ENREGSIZE,0);
        if (count==-1) perror("cli:send termin."),exit(1);
        printf("\nmsg fin # %4d errno=%4d envoi%5d
        octets pre=%6d der=%6d",i,errno,nbInfo*ENREGSIZE,
        (int)msg_src.valeur[0],(int)msg_src.valeur[nbInfo-1]);
     }
/*- Fermer connexion -------------------------------------*/
     close(sd); printf("\nExit.\n"); exit(0);
}
```

39

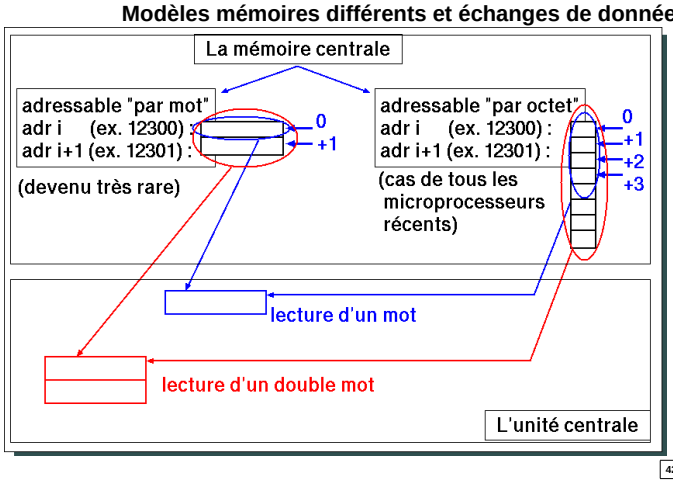
```
$ gcc -o tcpcli tcpcli.c
$ gcc -o tcpser tcpser.c

$ ./tcpser                          (1)
                                     |
ser - init socket                    |
errno=-1                             |
Attachmt reussi- Attente connexion   (2)
                                     |
errno=-1                             |
Connexion etablie                    |  (4)------\  (6) réception
                                     |   |        |   |
msg #    1, errno= -1, recu=   8,pre= |   | 1 der= |   | 1 en 1 fois
msg #    2, errno= -1, recu=  16,pre= |   | 2 der= |   | 1 en 1 fois
.............                        |   |        |   |
msg #   10, errno= -1, recu=  80,pre= |   |10 der= |   | 9 en 1 fois
Recu END_OF_SEND.                    |   |        |   |
 mess coupes=      0. d1=    10.Exit  |   |        |   |
-------------------------------------------------------------
Dans une autre fenêtre :             |   |        |   |
$ ./tcpcli                           |   |        |   |
                                     |   |        |   |
errno=0                              V   |        |   |
Connexion etablie, Envoi messages.   (3)-/        (5)-/ envoi

msg data #   1 errno=   0 envoi     8 octets pre=   1 der=   1
msg data #   2 errno=   0 envoi    16 octets pre=   2 der=   1
.............
msg data #  10 errno=   0 envoi    80 octets pre=  10 der=   9
msg fin #    1 errno=   0 envoi     8 octets pre=  -1 der=  -1
msg fin #    2 errno=   0 envoi     8 octets pre=  -1 der=  -1
msg fin #    3 errno=   0 envoi     8 octets pre=  -1 der=  -1
Exit.
```

40



41

Modèles mémoires différents et échanges de données



42

Ch03 - Les sockets

Les UC sont construites pour être capables de manipuler plusieurs **types de données** (entiers, flottants,caractères) qui sont de tailles différentes :

– certaines sont **csur 32 ou 64 bits** (int, long, float, double),

– d'autres sont des **collections d'octets**.

Quand on doit ranger (lire) une donnée de 32 bits dans une mémoire organisée en octets, il faut couper cette donnée en 4 morceaux et ranger (lire) chaque morceau (chaque octet) dans un octet de mémoire.

Mais par quel "bout" commencer ? Par le gros bout ou par le petit bout ?

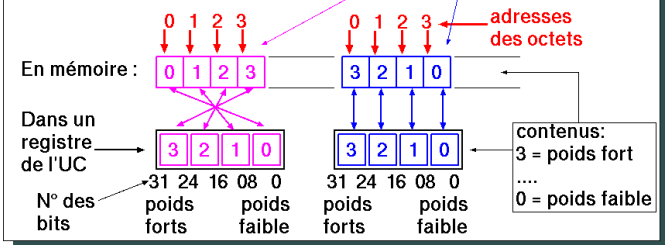
Il n'y a aucune nécessité technique qui favorise l'un plutôt que l'autre.

43

Schémas de rangement d'une suite de 4 octets formant un mot de 32 bits

Il y a deux possibilités: - les octets de poids forts d'abord
- les octets de poids faibles d'abord

Soit : le "gros bout" d'abord ("big endian" first),
ou le "petit bout" d'abord ("little endian" first).



44

Ch03 - Les sockets

Comme les protagonistes des Voyages de Gulliver certains préfèrent commencer par le gros bout ("big endian") d'autres par le petit bout ("little endian").

Et, comme il y avait **deux choix** possibles, la moitié des constructeurs d'ordinateurs ont fait l'un des choix, et l'autre moitié ... l'autre choix !

big endian : Sparc, MIPS (SGi), ...

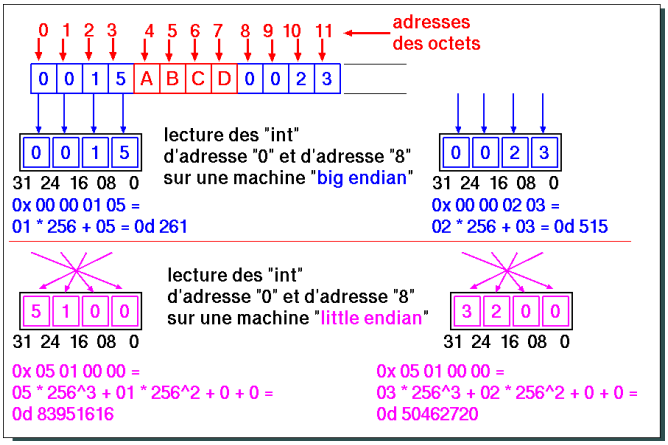
little endian : VAX, Motorola, Intel x86, ...

Remarque : certains processeurs modernes peuvent fonctionner selon les deux modes : Alpha, Power-PC, MIPS, ARM9, ...

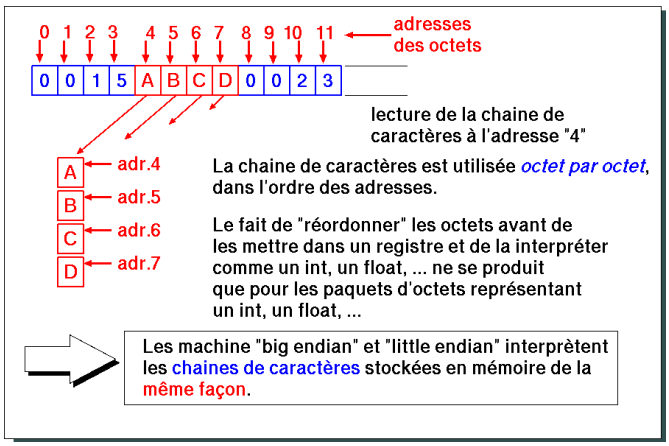
Mais en quoi ceci nous concerne-t-il ?

L'ordre des octets ("byte ordering") sur une machine concerne ***toutes*** les applications "réseau".

45



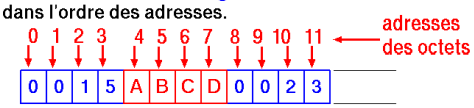
46



47

Si on veut avoir successivement en mémoire :
- un entier valant 261,
- une chaine de caractères contenant A,B,C,D,
- un entier valant 515,

alors, sur une machine big-endian, la mémoire devra contenir :
dans l'ordre des adresses.

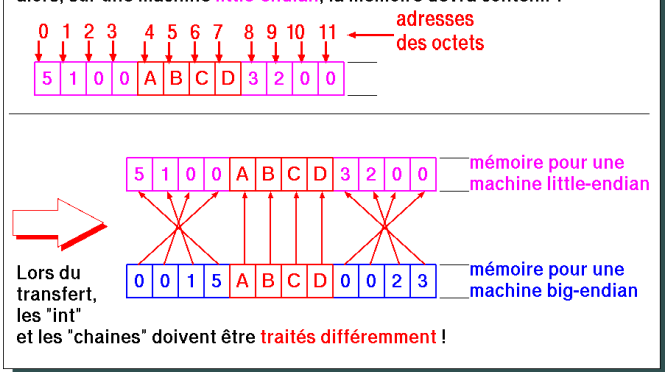


Si on transfère telle quelle cette portion de mémoire dans la mémoire d'une machine little-endian, alors, lors de l'interprétation de son contenu, on trouvera :
- un entier valant 83951616,
- une chaine de caractères contenant A,B,C,D,
- un entier valant 50462720.

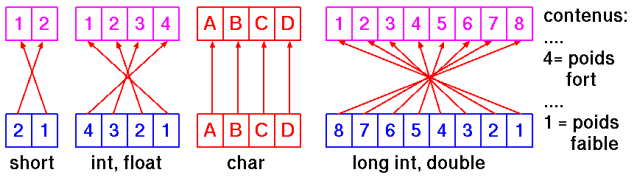
Après transfert, valeurs fausses pour les entiers, mais juste pour les chaines de caractères.

48

Si on veut avoir une interprétation correcte,
[entier 261, chaine A,B,C,D, entier 515]
alors, sur une machine little-endian, la mémoire devra contenir :



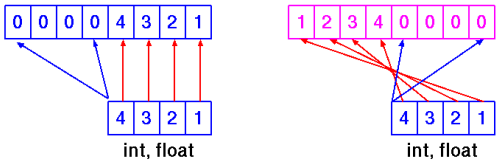
En fait, lors d'un transfert, c'est chaque type de donnée qui doit subir un traitement particulier, qui dépend de la nature de la machine de départ et de celle de la machine cible.



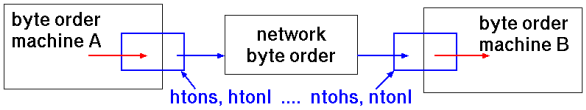
Le transfert est impossible si on ne connait pas le type de chaque donnée.
On ne peut pas prendre un bloc de mémoire et écrire une fonction de transfert générale.

Pour chaque paquet de données, il faut indiquer à la fonction de transfert, la suite des types à transférer.

Les choses se compliquent encore un peu quand on transfère des données d'une machine 32 bits vers une machine 64 bits.

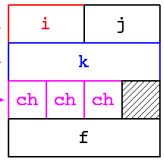


Pour l'utilisation des appels systèmes "réseaux" :



Pour les applications → Utilisation de la bibliothèque système "XDR"


```
/* wbl.c */
/* gcc -o wbl wbl.c */
#include <stdio.h>
/* #include <sys/file.h> define O_CREAT et O_RDWR */
#include <string.h> /* pour memcpy */
                       /* pour solaris */
#include <sys/fcntl.h> /* define O_CREAT et O_RDWR */

typedef struct {
   short i;
   short j;
   int k;
   char ch[3];
   int f;
} biglit;
extern int errno;
```




```
main()
{  biglit test;
   char *name = "wbl.dat";
   int fd,i;

   test.i = 10;
   test.j = 10*256+1;
   test.k = 10*256*256+11*256+12;
   test.ch[0]= 'A'; test.ch[1]= 'B'; test.ch[2]= 'C';
   test.f = 11*256*256+12*256+13;

   if ((fd = open(name,O_CREAT|O_RDWR,0600)) <=0)
   { printf("system err.in open, errno= %d\n",errno);
     exit(0); }

   i = write (fd,&test,sizeof(test));
   printf("écrit %d\n",i);
   if (i==-1)  {
     printf("system err.in write test, errno= %d\n",errno);
     perror("erreur="); }
   close (fd);
}
```



```
linux : Intel x86 32 bits, little endian
sunserv : Sparc 32 bits, big endian

[vayssade@linux endian]$ od -b wbl.dat
0000000 012 000 001 012 014 013 012 000 \
        101 102 103 277 015 014 013 000

lo33 sunserv:~/mv/endian> od -b wbl.dat
0000000 000 012 012 001 000 012 013 014 \
        101 102 103 004 000 013 014 015

1ère moitié  test.k = 10*256*256+11*256+12

linux    012 000 001 012 014 013 012 000   0*256*256*256

sunserv  000 012 012 001 000 012 013 014

2ème moitié  A B C

linux    101 102 103 277 015 014 013 000

sunserv  101 102 103 004 000 013 014 015
```


Ch03 - Les sockets. Exemples de questions

➤ En quelques mots, que sont les "sockets" et à quoi servent-ils ?
Réponse : diapo 2.

➤ Quels sont les deux domaines d'action des sockets ?
Réponse : diapo 3.

➤ Quels sont les deux types de sockets ?
Réponse : diapo 4.

➤ Décrire (un schéma serait bienvenu), le processus d'établissement d'une liaison par socket.
Réponse : diapos 6 à 8.

➤ Quels sont les caractéristiques des deux types principaux de sockets (leur méthodes d'échange de données) ?

Réponse : diapo 10.

➤ Quels sont les deux premiers appels systèmes dans l'établissement d'une liaison par socket ?
Réponse : diapos 12 à 14 : sd= socket() et bind(sd,..).

➤ Quels sont les appels systèmes d'échange de données sur un socket datagramme ?
Réponse : diapo 15 à 19 : sendto(), sendmsg(), recvfrom(), et, sous certaines conditions, send() et recv().

➤ Quel est le rôle exact de l'appel système bind() ?
Réponse : diapos 22, 25 et 27 : associer le descripteur "sd" renvoyé par sd=socket() au triplet (adr_ip, protocole, Port).

➤ Quel est le rôle de l'appel système listen() ? Dans quel cas doit-on l'utiliser ?

Réponse : diapos 28 à 30.

➤ Quel est l'appel système qui met un socket "connecté" en attente des demandes de connexion ? Et que renvoi cet appel ?

Réponse : diapos 30 et 31 : newsd= accept()

➤ Décrire le mode de fonctionnement des deux types de process serveurs de socket connecté ?

Réponse : diapo 32.

➤ Faite un schéma du process client et du process serveur connectés par un socket "stream", avec l'indication des directives d'échange de données et des descripteurs qu'elles utilisent.

Réponse : diapo 41.

➤ Soit le mot de 32 bits 0x01020304 (int ou long suivant les machines). Dessinez la position des *octets* composant cet entier dans

la mémoire d'une machine "big endian" et dans celle d'une machine "little endian".

Réponse : diapo 46 : Big= [01 | 02 | 03 | 04], Little= [04 | 03 | 02 | 01].

➤ Es-ce que c'est la machine (l'UC) ou bien la mémoire qui est "Big" ou "Little" endian ?

Réponse : diapo 49 : l'unité centrale.

➤ Quand on envoie un paquet de données d'une machine "Big endian" à une machine "Little endian" (ou l'inverse !), suffit-il d'inverser tous les octets 4 par 4 (avec htonl() par exemple) ?

Réponse : diapo 50 : Non ! !

➤ Si on a l'élément "char ch[3]" dans une structure de donnée, que font les compilateurs modernes ? Pourquoi ?

Réponse : diapo 52 : ils ajoutent un octet de remplissage (padding) parce que les accès à la mémoire "calés" sur des "frontières" de mot sont beaucoup plus performants (de 2 à 10 fois selon les machines).

Ch03 - Les sockets.

Fin du chapitre.

(diapo vide)

Ch04 : Cours Architectures Internet - Notion de protocole de communication

Objectif du chapitre : montrer que l'**impossibilité** de garantir la **fiabilité** d'une transmission oblige à créer des **protocoles** complexes.

Un protocole : un algorithme, mis en œuvre dans un logiciel de contrôle, et qui gère des échanges de données.

Primitives d'échange de données sur une liaison

On a vu que les échanges de messages entre processus à travers un "socket" pouvaient être soit fiables (socket "stream"), soit non fiables (socket "datagram").

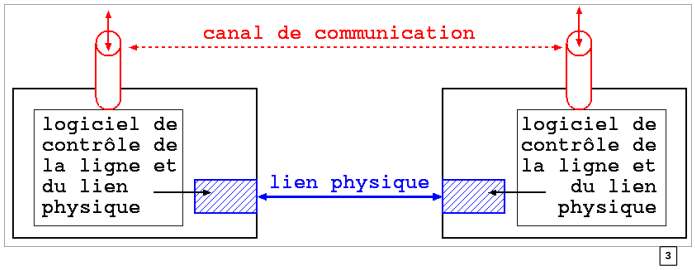
De plus, on a vu que :

- dans le cas non fiable (datagram), la communication était du type non connectée,
- dans le cas fiable (stream) la communication était du type connectée (connection oriented), c'est-à-dire qu'à l'ouverture du socket, le système établit un "circuit virtuel" par lequel les informations échangées transistent comme dans un "pipe" unix.

échange de données sur une liaison

Les deux types de communication reflètent l'état des liens qui sont utilisés pour véhiculer cette communication.

logiciels de contrôle : but est de "**cacher**" aux programmes utilisateurs le fait que, dans le cas général, les transmissions sur un réseau sont sujettes à des **erreurs**, des **pertes** de messages, des **altérations**, des **duplications** de messages, etc ...

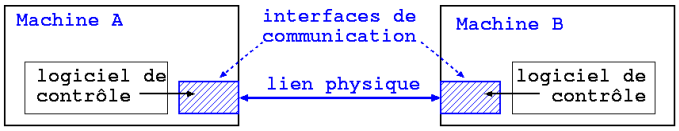


Protocoles et détection d'erreurs

- Le logiciel de réseau doit **tenir compte** de ce **manque de fiabilité** de la transmission et utiliser des **techniques de dialogue** entre les machines qui lui permettent de **transférer** tout de même des informations de **façon fiable**.
- Ces techniques de dialogues sont appelées des "**protocoles**". Elles sont fondées sur la **détection d'erreurs** de transmission, et la **ré-émission** des données altérées, après requête par le destinataire.
- La détection des erreurs est, elle, basée sur des **calculs de parité** plus ou moins complets.

Position du problème : échanges de données entre deux machines

Une machine A veut envoyer des données à une machine B.



On pourrait penser que l'on va faire comme dans le cas de deux process communiquant à travers un "pipe".

Mais, il y a ici plusieurs **différences majeures** avec le cas du pipe.

Différences "pipe"/connexion distante

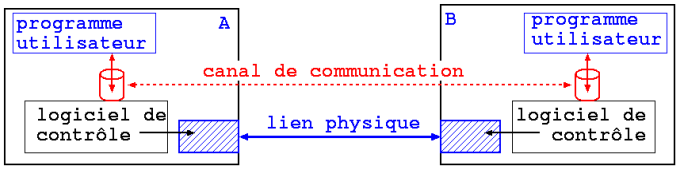
- A et B n'ont **pas de mémoire commune**, A ne peut donc pas savoir si le "canal" est plein ou non pour arrêter ou continuer à envoyer des données.
  - si A continue à envoyer des données alors que B n'est pas prête à les traiter, une partie des données seront perdues,
  - ==> B doitpouvoir avertir A pour lui dire "attends" ou "continue",

"pipe"/connexion distante (suite)

- **le temps de transit** des informations de A vers B ne peut plus être considéré comme négligeable (de même de B vers A).
  - Ainsi, par exemple, quand B veut dire à A "attends je n'ai plus de place", le temps que ce message, parvienne à A, ce dernier à encore envoyé d'autres données.
- les messages dans les deux sens **peuvent être altérés** par la transmission.
  - Ainsi, si un message "attends" de B est altéré, A ne va pas le comprendre et continuer à transmettre des données vers B.
  - Inversement si B envoie un message "continue" qui est altéré, A ne va pas reprendre ses émissions.

Contrôle de la connexion distante

Pour le cas simplifié de la connexion entre deux machines, nous allons construire petit à petit le **logiciel de contrôle** de la communication.



L'objectif du logiciel de contrôle est de présenter aux programmes utilisateurs un **canal de communication** qui puisse être utilisé aussi simplement qu'un "pipe" entre deux process unix.

Contrôle de la connexion distante : primitives de base

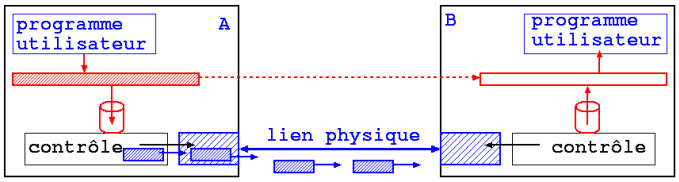
Nous allons tout d'abord décrire un certain nombre de primitives élémentaires, fournies par le système d'exploitation (avec une aide éventuelle du matériel).

Ensuite, nous compliquerons la situation pour voir comment on est alors obligé de faire évoluer les algorithmes de contrôle en les rendant plus complexes.

Contrôle de la connexion distante : découpage des données

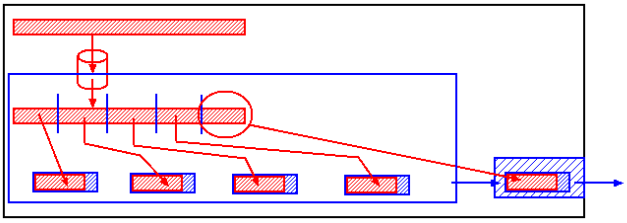
Tout d'abord, les messages échangés par les applications peuvent être de longueur quelconque (ou presque).

Ce n'est pas le cas des échanges qui sont effectués entre les matériels des processeurs de communication qui contrôlent la ligne reliant les deux machines.



Contrôle de la connexion distante : découpage des données

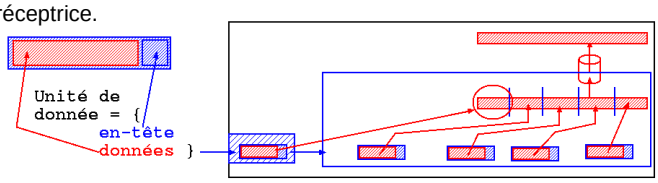
On procède donc à un **découpage** des données à transmettre en blocs plus petits, et on insère (on "emballe") ces blocs dans des "**paquets**" appelés (dans le langage des réseaux) des **unités de données**.



Contrôle de la connexion distante : découpage des données

Ces unités de données sont constituées comme un enregistrement: Unité_de_donnée: en_tête
données

L'en-tête de chaque unité permet de reconstituer le message originel sur la machine destinataire avant de le passer à l'application réceptrice.



Contrôle de la connexion : échange des unités de données

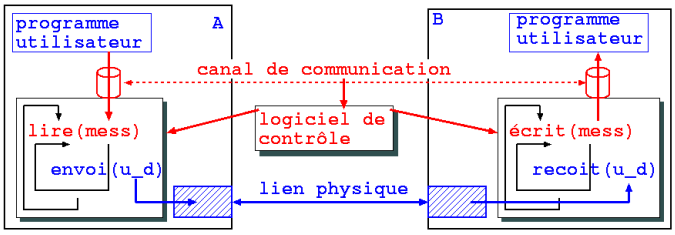
Les processeurs de communication vont exécuter un protocole pour échanger les unités de données.

Pour écrire l'algorithme du protocole on va utiliser les primitives suivantes :

envoi (u_d)	envoi d'une unité de données vers l'autre processeur de communication
recoit (u_d)	recevoir une unité de données de l'autre processeur de communication
lire (mess)	lire un message depuis l'application cliente de la communication
écrit (mess)	écrire un message à destination de l'application cliente

Contrôle de la connexion : échange des unités de données

L'objectif est d'écrire le logiciel qui contrôle les envois et réceptions de données depuis un programme utilisateur, à travers l'interface de communication, qui, elle, gère les échanges sur le lien physique.



Contrôle de la connexion : échange des unités de données

Hypothèses simplificatrices pour un premier algorithme

- une ligne de communication **fiable** : pour toutes les unités de données envoyées au destinataire :
  - parviennent sans erreur,
  - aucune n'est perdue,
  - aucune n'est dupliquée.
- les processeurs de communication ne mémorisent qu'une seule unité de donnée ; donc que l'application découpe ses envois en messages assez petits pour tenir dans une seule unité de données.

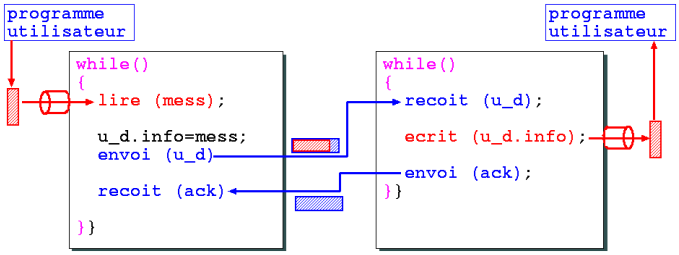
Contrôle de la connexion : échange des unités de données

```
protocole machine_A
/* procédure d'émission */
{ struct unité_de_données {
      int en-tête;
      char *info; }
  unité_de_données u_d, ack;
  char *mess;

while() /* boucle infinie */
{
  lire (mess);    /* lire le
          mess à transmettre */

  u_d.info=mess; /* le copier
      dans unité de données */
  envoi (u_d)     /* envoyer
                cette unité */

  recoit (ack)    /* attendre
        accusé de réception */
}}
```

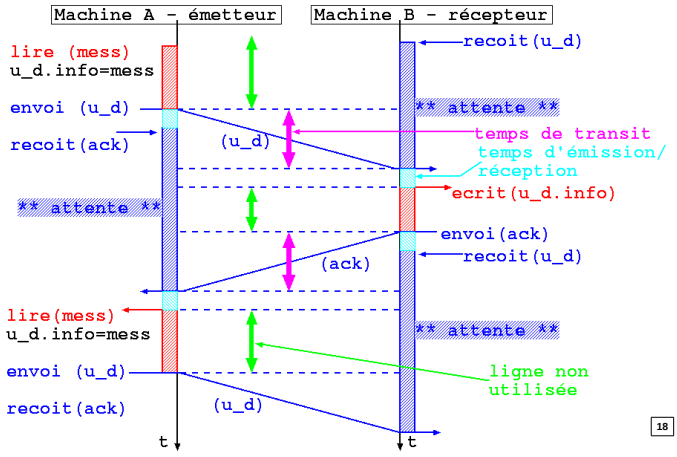
```
protocole machine_B
/* procédure de réception */
{ struct unité_de_données {
      int en-tête;
      char *info; }
  unité_de_données u_d, ack;
  char *mess;

while() /* boucle infinie */
{
  recoit (u_d);      /* lire
      une unité de données */

  ecrit (u_d.info); /*
                transmettre
  message au destinataire */

  envoi (ack);       /* envoi
        accusé de réception */
}}
```

Contrôle de la connexion : échange des unités de données

On a donc un échange en **alternance** d'unités de "**données utiles**" et d'unités de "**données de service**" destinées à réguler la communication ;



échange d'unités de données "à l'alternat" : diagramme de temps.



échange d'unités de données "à l'alternat" : diagramme de temps Commentaires.

- dans un diagramme de temps d'échange réseau, chaque délai doit être pris en compte. Ici :
  - temps d'échange avec l'application et temps de traitement,
  - temps d'émission et réception (vit.de transmission/quantité : 128 Ko = 1280 Kbits à 512 kbits/s => 2.5 s),
  - temps de transit sur la ligne (vit. sur le fil/distance) ;
- le diagramme n'est pas à l'échelle : le temps de transit sera plus faible que le temps d'émission sur une liaison à courte distance et bas débit ; ce sera l'inverse sur une liaison longue distance à haut débit.

échange d'unités de données "à l'alternat" : diagramme de temps Commentaires.

- => si on augmente le débit, ça ne modifie **pas** le temps de transit !
- dans un échange à l'alternat (simpliste), chaque extrêmité passe la plus grosse part de son temps à **atttendre**.
- dans un échange à l'alternat les temps de traitement de chaque extrêmité entrainent une **sous utilisation de la ligne**.

Primitives d'échange de données sur une liaison non fiable

Nous allons maintenant compliquer la situation en cessant de faire l'hypothèse que la ligne de transmission est fiable.

==> Certaines unités de données peuvent parvenir **modifiées** au destinataire.

On va supposer que l'on dispose d'une primitive permettant de **tester la validité** d'une unité de données :

```
valid (u_d)  = 1 si u_d correcte, = 0 sinon
```

Ce test de validité se fait grâce à un calcul de bits de parité ajoutés par l'émetteur à la suite du message. Dans la pratique, on utilise des codes de Hamming permettant de détecter les erreurs avec une bonne fiabilité.

échange d'unités de données sur une liaison non fiable.

Nous devons modifier notre algorithme pour traiter un cas supplémentaire : l'arrivée d'une **u_d** que l'on détecte comme **incorrecte**.

Le processeur de B doit alors **prévenir** le processeur de A de cette erreur afin que l'u_d erronée soit**réémise**.

Première tentative pour l'algorithme du récepteur B :

```
recoit(u_d);          /* lire une unité de données */
si valid(u_d) == 1    /* arrivée u_d correcte */
    ecrit(u_d.info)   /* transmettre message */
    ret = ack;        /* renvoi ack ok */
else                  /* arrivée u_d incorrecte */
    ret = err;        /* renvoi err: réémettre */
envoi (ret);          /* envoi ack */
```


échange d'unités de données sur une liaison non fiable.

Si A reçoit un accusé de réception "ok" il émet le message suivant, sinon il réémet la même unité de données que précedemment.

! MAIS ! Si la ligne n'est pas fiable, et peut altérer des u_d de l'émetteur, cette même ligne peut **aussi** altérer les u_d envoyées par le récepteur comme accusé de réception !

Ainsi si A reçoit un accusé de réception que la vérification déclare erroné, il **ne sait pas** si cet accusé de réception lui demandait d'émettre l'u_d suivante ou de réémettre la dernière !

échange d'unités de données sur une liaison non fiable.

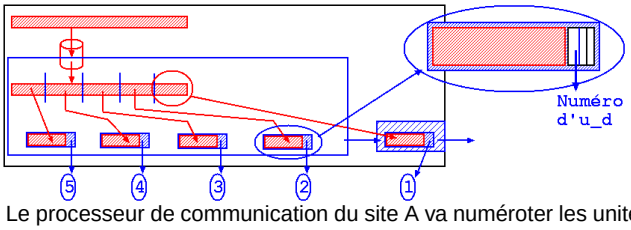
Si A décide d'émettre la suivante et que l'ack avait la valeur "réémettre", **une u_d est perdue** !

Si A décide de réémettre la dernière u_d envoyée et que la valeur de l'ack était "ok", B va recevoir **deux fois** cette u_d ! Il y aura donc **duplication** d'un message.

Il faut donc donner à B un moyen de faire la distinction entre les u_d reçues de A.

Ce moyen de distinguer les u_d va être un **numéro d'ordre** inséré dans l'en-tête de chaque u_d émise par A.

échange d'unités de données sur une liaison non fiable.



Le processeur de communication du site A va numéroter les unités de données envoyées vers B.

Il augmente le numéro de 1, pour chaque nouvelle u_d.

Bien sûr, le numéro reste inchangé lorsque A réémet une u_d, suite à l'arrivée d'un ack indiquant une erreur de transmission, OÙ à la réception d'un ack **lui-même** incorrect.

échange d'unités de données sur une liaison non fiable.

```
protocole machine_A      /* procédure d'émission */
                         /* avec numérotation des u_d */
{  struct unité_de_données { int en-tête;
                             int numéro;
                             char *info; }
   unité_de_données u_d , ack ;
   int numero_ud;  /* numéro de l'u_d envoyée */
   char *mess;
   numero_ud = 1 ; /* initialiser numéro d'u_d */
   lire (mess);    /* lire premier mess à transmettre */

while() /* boucle infinie */
{
   u_d.info = mess;          /* copier mess dans u_d */
   u_d.numero = numero_ud;   /* affecter un numéro à l'u_d */
   envoi (u_d)               /* envoyer cette unité */

   recoit (ack)            /* attendre accusé de réception */
   if( valid(ack)== 1 )    /* réception correcte de ack */
   { if(ack.info == "u_d bien reçue")  /* u_d bien arrivée */
       { numero_ud = numero_ud + 1 ;   /* u_d suivante */
         lire (mess);                  /* lire mess suivant */
       } /* si u_d mal arrivée: réémettre sans inc. numéro */
   } /* si ack incorrecte : réémettre sans inc. numéro */
} /* fin while */
} /* fin */
```


échange d'unités de données sur une liaison non fiable.

```
protocole machine_B      /* procédure réception */
                         /* avec numérotation des u_d */
{  struct unité_de_données { int en-tête;
                             int numéro;  char *info; }
   unité_de_données u_d , ack ;
   int numero_ud;  /* numéro de l'u_d attendue */
   char *mess;
   numero_ud = 1 ; /* initialiser numéro d'u_d attendue */
while() /* boucle infinie */
{
   recoit (u_d)              /* attendre u_d */
   if( valid (u_d) == 1 )    /* réception correcte de u_d ? */
   { if(numero_ud == u_d.numero)  /* reçu la bonne ? */
       { ecrit (u_d.info);     /* transmettre mess contenu */
         numero_ud = numero_ud + 1  /* attendre suivant */
         ack.info= "u_d bien reçue" /* annoncer bien reçue */
       }
       else                        /* pas reçu la bonne */
        ack.info= "u_d bien reçue" /* mais ack ok pour */
       endif            /* éviter qu'elle soit réémise.  */
   }                    /* i.e. on décide de "l'oublier" */
   else
     ack.info = "u_d mal reçue"
   endif
   envoi (ack);         /* envoi accusé de réception */
} /* fin while */  } /* fin */
```


échange d'unités de données sur une liaison non fiable.

! MAIS ! une ligne de communication non fiable, peut non seulement altérer des u_d, modifier leur ordre, **mais aussi perdre** des unités de données.

Or, dans le deuxième essai, ci-dessus, l'émetteur **A attend** une réponse de B (un ack), pour poursuivre ses émissions.

Si l'ack de B est perdu par la ligne, les deux protocoles **se bloquent indéfiniment**.

De même, si une u_d de A se perd, B attend indéfiniment.

Ils ne sont donc **pas assez robustes** pour fonctionner dans cette situation nouvelle, avec des **pertes** possibles d'u_d.

Il faut donc trouver un moyen de limiter cette attente éventuelle de A ou B.

échange d'unités de données sur une liaison non fiable.

Ce moyen va être un **compteur de temps** provoquant une interruption de l'attente à son expiration (time-out).

On va utiliser une primitive "reçoit" modifiée :

is= recoit (u_d,délai) recevoir une unité de données de l'autre processeur de communication avec un **délai de garde**

Si réception **avant fin de "délai"**, sortir de "recoit" avec **is=ok** et u_d contient unité reçue

Si **rien reçu** pendant "délai", sortir de "recoit" avec **is="hors délai"**.

Dans ce troisième essai, les lignes ajoutées par rapport au deuxième cas sont marquées par "++>".

échange d'unités de données sur une liaison non fiable.

```
protocole machine_A    /* procédure d'émission */
....................
++> int délai = VALEUR;/* valeur du délai */
    numero_ud = 1 ;    /* initialiser numéro d'u_d */
    lire (mess);       /* lire premier mess à transmettre */
while()  /* boucle infinie */
{
    u_d.info = mess;        /* copier mess dans u_d */
    u_d.numero= numero_ud;  /* affecter un numéro à l'u_d */
    envoi(u_d)              /* envoyer cette unité */

--> is= recoit(ack,délai) /* attendre accusé réception */
++> if(is=="hors délai")  /* u_d OU ack perdu: réémettre */
++> {}       /* rien: retourner à while et réémettre */
++> else     /* sinon reçu ack: cas précédent */

    if( valid(ack) == 1 )  /* réception correcte de ack */
    { if(ack.info == "u_d bien reçue") /* u_d bien arrivée */
      { numero_ud = numero_ud + 1;    /* u_d suivante */
        lire (mess);                  /* lire mess suivant */
      }   /* si u_d mal arrivée : réémettre */
    }     /* si ack incorrecte : réémettre  */

++> endif   /* fin if hors délai */
} /* fin while */  } /* fin */
```


échange d'unités de données sur une liaison non fiable.

```
protocole machine_B  /* procédure réception */
.....
while()  /* boucle infinie */
{
--> is=recoit(u_d,délai) /* attendre u_d */
++> if(is=="hors délai") /* u_d OU ack perdu: réémettre ack */
++>  {}      /* rien: retourner à envoi ack puis while */
++> else     /* sinon reçu ack: cas précédent */

     if( valid(u_d)== 1 ) /* réception correcte de u_d ? */
     { if(numero_ud == u_d.numero)  /* reçu la bonne ? */
       { ecrit (u_d.info); /* transmettre mess contenu */
         numero_ud = numero_ud + 1 /* attendre suivant */
         ack.info = "u_d bien reçue"
       }
       else                        /* pas reçu la bonne */
         ack.info= "u_d bien reçue" /* mais ack =ok pour */
       endif             /* éviter qu'elle soit réémise. */
     }                   /* i.e. on décide de "l'oublier" */
     else
       ack.info = "u_d mal reçue"
     endif
++> endif   /* fin if hors délai */

     envoi (ack);     /* envoi accusé de réception */
} /* fin while */  } /* fin */
```


échange d'unités de données sur une liaison non fiable.

Quelle doit être la valeur du délai de garde ?

A –> émet [u_d] en direction de B,

l'u_d "voyage" sur la ligne de communication à une certaine vitesse

==> il existe donc une borne supérieure au temps que va mettre une u_d pour aller de A à B.

Le processeur B va prendre un certain temps pour traiter l'u_d puis va renvoyer vers A une u_d servant d'accusé de réception.

échange d'unités de données sur une liaison non fiable.

Vu de A on donc :

```
t1= temps de transfert de u_d vers B
t2= temps de traitement de u_d sur B
t3= temps de transfert de u_d vers A
```

Après un temps t= t1+t2+t3, A s'attend à recevoir l'accusé de réception de B.

Si les temps t1, t2 et t3 sont des bornes supérieures choisies largement, on peut régler le délai de garde de A à la valeur t1+t2+t3.

Si après ce temps, A n'a pas reçu l'u_d en retour de B (contenant l'ack), il en conclut que l'une des u_d (soir la donnée, soit l'ack) s'est perdue.

échange d'unités de données sur une liaison non fiable. Remarque.

Les deux processeurs A et B effectuent une boucle infinie .

==> le numéro affecté aux unités de données va devenir très grand et ne plus pouvoir être codé sur le champ prévu au départ. Il faudra donc le remettre à zéro lorsque le débordement se produira. On va alors réutiliser les numéros.

Pour qu'il n'y ait pas de confusion possible d'u_d, il faut que, au moment du passage au maximum, **toutes les u_d, et tous les acks**, de numéros 1,2,3,... aient disparu du réseau.

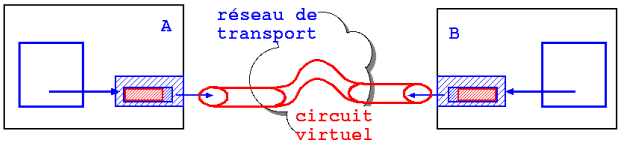
Ainsi, sur un réseau où le délai de garde doit être réglé à 4 secondes, si on peut envoyer 5000 u_d par seconde, comme 4*5000 > 2**16, un champ de 16 bits pour le numéro d'u_d sera insuffisant.

échange d'unités de données sur une liaison non fiable.

Dans les tentatives précédentes, on a pas envisagé le cas où des unités de données parviennent au destinataire dans un **ordre différent** de l'ordre d'émission.

Bien sûr, ceci ne peut pas se produire dans le cas d'une seule ligne de communication reliant deux machines.

Mais il peut se produire si cette ligne de communication est, en fait, un circuit virtuel dans un réseau.



échange d'unités de données sur une liaison non fiable.

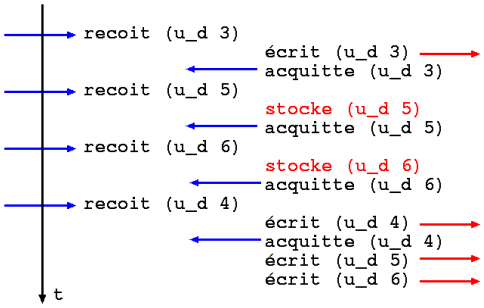
Le circuit virtuel, suivant le type de réseau utilisé, **ne garanti pas** nécéssairement le **séquencement** des unités de données (c'est le cas par exemple d'un datagramme dans le domaine internet).

Ainsi deux u_d, peuvent emprunter des **chemins physiques** différents, de durée différentes, et donc parvenir à destination dans un ordre différent de l'ordre d'émission.

Supposons que le processeur de communication de B, puisse gérer un certain nombre de **zones tampons** où il va pouvoir **stocker** les u_d qui lui arrivent dans le désordre.

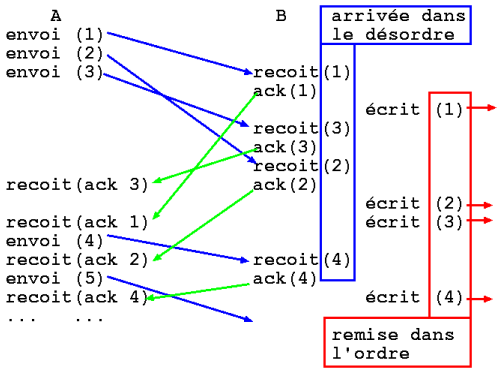
échange d'unités de données sur une liaison non fiable. Gestion des u_d hors séquence.

B émet, dès réception, les acquittements de 5 et 6 afin d'éviter que le délai de garde associé à 5 et 6 ne se déclenche et que ces deux u_d ne soient réémises inutilement.



échange d'unités de données sur une liaison non fiable. Gestion des u_d hors séquence.

Si B peut stocker quelques u_d, A peut en profiter pour émettre **à l'avance** des u_d vers B, **sans attendre** d'avoir reçu l'acquittement des précédentes.



échange d'unités de données, notion de fenêtre.

Le nombre d'u_d envoyée par A "en avance" s'apelle **la fenêtre**.

La borne inférieure de la fenêtre est le numéro d'u_d attendue par B, la borne supérieure est plus élevée d'un nombre qui dépend du nombre de tampons disponibles dans B.

Comme B **doit délivrer** les u_d **dans l'ordre**, il doit garder une place disponible pour recevoir l'u_d de numéro "attendu".

C'est seulement alors qu'il peut délivrer cette u_d et toutes les suivantes, en séquence, déja parvenues et stockées en attente.

échange d'unités de données, notion de fenêtre.

De son côté, A ne doit pas "submerger" B d'u_d que celui-ci ne pourrait **ni stocker** (plus de tampons) **ni délivrer** (car l'u_d "attendue" ne serait pas arrivée). C'est pourquoi, dans l'exemple ci-dessus, A ne "redémarre" que lorsqu'il reçoit l'ack de la première u_d envoyée : A sait alors que B a libéré au moins un buffer.

échange d'unités de données, notion de fenêtre.

Ci-dessus, A attends de recevoir l'ack 1 pour continuer.

Mais il aurait pu envoyer l'u_d suivante un peu plus tôt, en supposant qu'elle parviendrait sur B, **après** que celui-ci ait reçut, acquitté et délivré l'u_d 1. Ceci, en supposant que l'ack 1 est "en route". On parle alors pour A de **stratégie** d'émission **optimiste** par opposition à la stratégie pessimiste du schéma.

échange d'unités de données, notion de fenêtre.

Si la tratégie de A est **trop** optimiste et que B reçoit l'u_d suivante (no.4 ici) **avant** d'avoit pu libérer ses tampons, il peut soit :

- ignorer cette arrivée trop précoce, l'u_d 4 ne sera pas acquittée et au bout du délai de garde, A va la réémettre,

- envoyer à A un **acq spécial "4 trop tôt"** => A va réémettre 4, mais avant l'expiration du délai de garde. Ceci accélère la "reprise" du cours normal des émissions.

Ch04 - Notion de protocole de communication.
Exemples de questions

➤ Qu'est-ce qu'un protocole ?
Réponse : diapo 1.

➤ Pourquoi as-t-on besoin de protocoles ?
Réponse : diapo 1.

➤ Quel est le rôle des protocoles de communication mis en oeuvre par les logiciels réseaux ?
Réponse : diapos 3 et 4.

➤ Quelles sont les 3 différences fondamentales entre une communication par "pipe" entre deux process et une communication entre deux machines reliées par un lien physique ?
Réponse : diapos 6 et 7.

➤ Comment se fait l'échange de données sur une liaison ?
Réponse : diapos 10 à 12.

➤ Quel sont les algorithmes de principe des programmes qui contrôlent l'échange de données (échange simplifié "à l'alternat" sur ligne fiab le) ?
Réponse : diapo 17.

➤ Dessinez le diagramme de temps décrivant un échange de données à l'alternat.
Réponse : diapo 18.

➤ Dans un échange sur une ligne *non able* , pourquoi ne suffit-il pas de pouvoir renvoyer soit un ack "ok" soit un ack "erreur", mais faut-il en plus autre chose dans le protocole ?
Réponse : diapos 23 et 24.

➤ Dans le cas où une ligne non fiab le peut non seulement altérer des

données, mais aussi *perdre* des paquets, quel est le moyen habituel utilisé dans les protocoles pour tenir compte de ce défaut ?
Réponse : diapos 28 et 29.

➤ Qu'appelle-t-on protocole "à fenêtre" ?
Réponse : diapo 39.

Ch04 - Notion de protocole de communication.

Fin du chapitre.

(diapo vide)

Ch05 - Les réseaux locaux de type ethernet

ethernet : premiers travaux en 1976 (Xerox Park, Palo Alto)

1ère norme industrielle : Dec + Xerox + Intel : sept 82

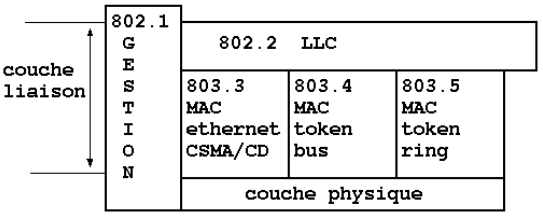
puis ISO 802.3 qui définit le format des trames et les paramètres du réseau.

Normes régissant ethernet

Réseaux locaux et modèle OSI

couche liaison (couche 2), découpée en 2 sous-couches :

MAC = Medium Access Control, Contrôle d'Accès au Canal
LLC = Logical Link Control, Contrôle de liaison logique



CSMA/CD = Carrier Sense Media Access / Collision Detect
Test de de porteuse et détection de collision

token = jeton (sur bus ou sur anneau)

Adressage

L'émission d'un message est faite sur une **liaison** à laquelle **tous** les abonnés sont **raccordés**.

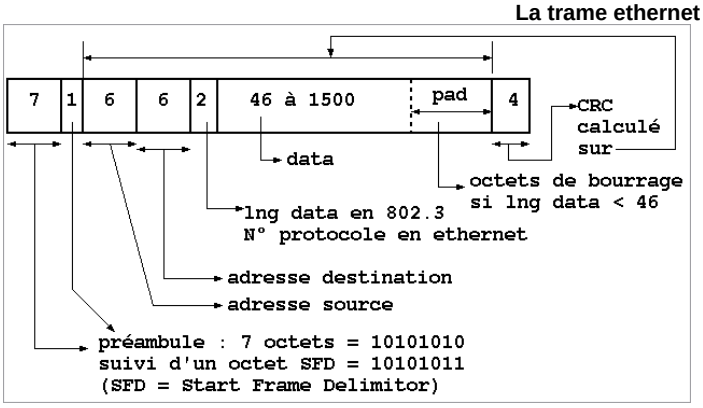
- le ou les **destinataires** doivent **reconnaître** les **trames** au moment ou elles passent "devant" eux ;
- chaque trame contient :
  - **l'adresse** du **destinataire**,
  - **l'adresse** de **l'émetteur** (source),
  - des informations de **contrôle**,
  - les **données** (contenu du message).

Chaque équipement branché sur un ethernet possède une adresse **unique** : l'adresse "MAC".

Le caractère unique de "l'adresse ethernet" est assuré par une norme internationale qui affecte un numéro unique à chaque frabriquant de matériel ethernet.

L'adresse associe ce numéro de fabriquant à un numéro de série, l'ensemble (sur 48 bits, soit 6 octets) est stocké dans une mémoire non volatile sur l'équipement.

La trame ethernet



ethernet et 802.3

```
Différences entre trames ethernet et 802.3 :
          <- 6 octets-> <- 6 octets-> <2 octets>
802.3    | destination |  adr source | lng LLC |
ether    | destination |  adr source | num pro |
```

Les 2 octets suivant l'adr scre contiennent :
- pour 802.3 : une longuer <=1500 (en octets)
- pour ether : un numéro de protocole >= 0x0800 (=2048 > 1500).

La trame 802.3 contient un champ longueur_data_LLC absent de la trame ethernet.

Cette dernière contient directement le paquet du protocole de niveau 3.

Donc la longueur des data utiles se trouve dans le paquet de niveau 3 : il faut cette lng pour enlever les bits de bourage : avec la trame ethernet, ne peut pas se faire au niveau de la couche 2, sans "regarder" dans le paquet de niveau 3 => il y a violation de la notion de couche

```
adresses MAC :    début= 3 octets fabriquant
                  fin= 3 octets numéro de série
exemples d'adr. de fabriquants
(certains peuvent en avoir plusieurs)
        00:00:0C   cisco        00:0D:60  IBM
        08:00:20   SUN          00:0D:9D  HP
        08:00:2B   DEC          00:0B:CD  HP
        08:00:51   IBM
```


Le protocole ethernet : CSMA/CD

CSMA/CD est un protocole "**à compétition**" : tous les équipements sont à l'écoute du média. Quand un équipement a besoin d'émettre :
- si le média est **occupé, il ajourne** l'émission,
- si le média est **libre, il émet**.

Mais, il est possible que deux équipements décident d'émettre au même instant : il y aura alors **collision**.

Les équipements sont construits de telle sorte qu'ils **écoutent le média en même temps qu'ils émettent**. Ainsi, s'ils sont seuls à émettre, le signal écouté doit être le même que le signal émis. Si les deux signaux sont différents, l'équipement en déduit qu'il y a collision.

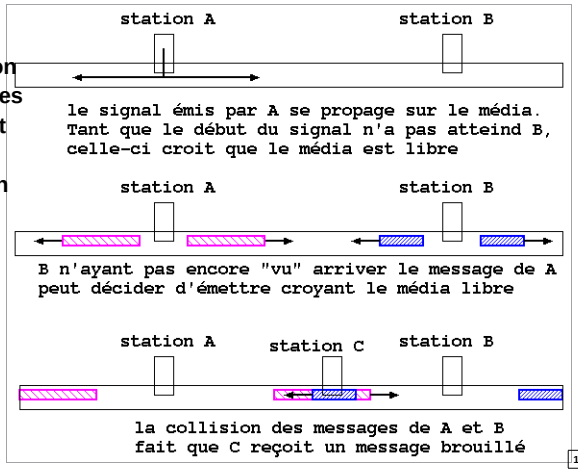
Le protocole ethernet : CSMA/CD

En cas de collision, l'équipement **cesse d'émettre** et entame la séquence de **résolution de conflit**.

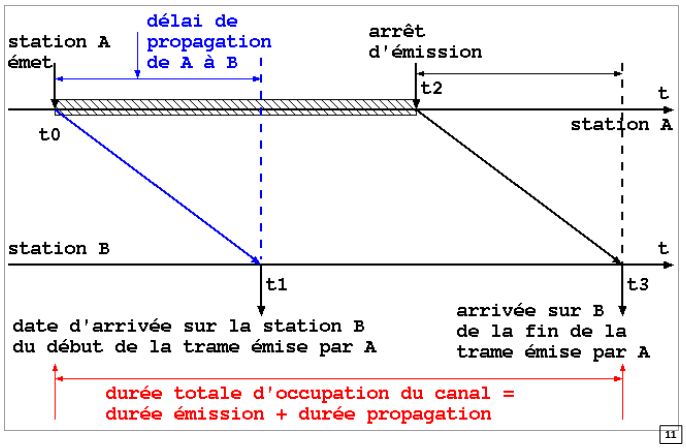
La **détection** de collision dépend du **délai de propagation** sur le câble et de la **distance** entre les deux équipements émetteurs.

En effet, l'émission est faite par l'envoi sur le média d'un signal électromagnétique dont la vitesse de propagation est **finie**. Il faut donc un certain **délai "dt"** pour que le signal émis par une station parvienne aux autres stations.

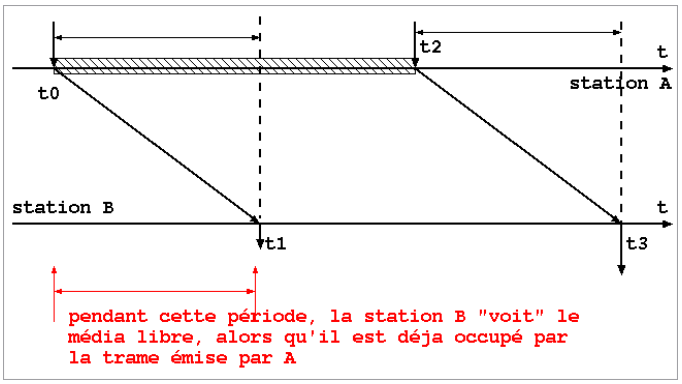
Émission de trames ethernet et collision



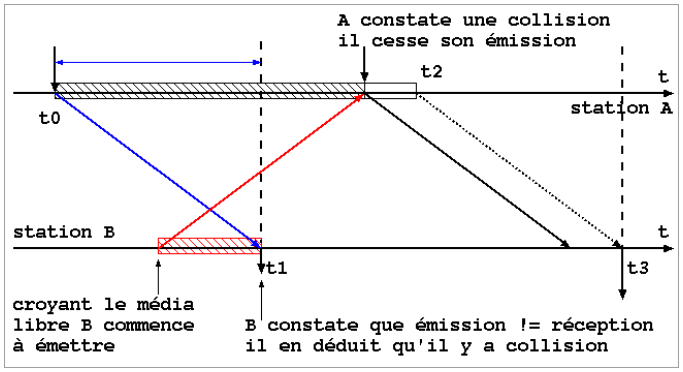
Émission de trames ethernet et occupation du canal 1/2



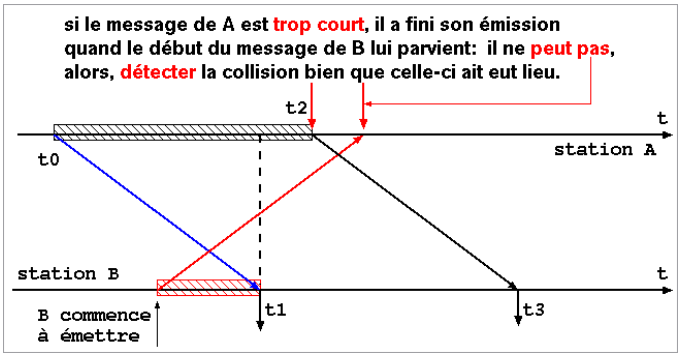
Émission de trames ethernet et occupation du canal 2/2



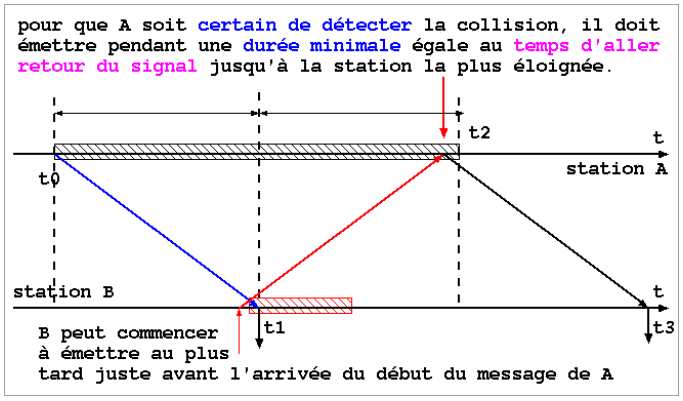
Trames ethernet et condition de collision



Trames ethernet, collision et condition de détection 1/2



Trames ethernet, collision et condition de détection 2/2



CSMA/CD : principe de fonctionnement

1. attente d'un silence minimum avant émission ("délai intertrame" égal à 9.6μs),
2. longueur minimale de trame (si data < 46 octets, utilisation de bits de bourrage (padding)), supérieure au temps du plus long aller-retour pour assurer qu'une collision soit détectée,
3. après détection d'une collision, **continuation** de l'émission de bits de brouillage (au moins 32) ; une trame ainsi tronquée (runt frame) aura au moins 96 bits (64 d'en-tête + 32 de brouillage) ;
4. à la réception, ignorer toute trame de longueur insuffisante (jabbers) ; normalement il n'y en a pas, mais toutes les implémen-

tations ne sont pas forcément correctes, par exemple quand on connecte des équipements 10BASE-T sur des prises RJ45 100BT ;

5. après une collision, exécuter l'**algorithme de ré-émission**.

Nota : la seule raison du délai inter-trames spécifié par la norme est de permettre aux stations de faire basculer ses circuits de connexion du mode émission au mode réception.

Il ne faut pas oublier que la spécification date des années 1970. Certains vendeurs trichent et fabriquent des cartes avec un délai plus court entre deux trames. Ces cartes vont plus vite que la concurence, par contre elles peuvent ne pas interopérer correctement avec celles d'autres fabriquants.

Algorithme de ré-émission

Après une collision, les deux stations émettrices des trames entrées en collision, vont devoir ré-émettre leur trame. Or il ne faudrait pas qu'elles fassent cette tentative après le même délai depuis la collision.

De plus ces deux stations **ne peuvent pas "se concerter"** pour décider qui va ré-émettre le premier, puisque pour se concerter elle devraient échanger des informations, donc émettre sur le média !

Il faut donc un algorithme **distribué**, qui exécuté indépendamment par les deux stations fournisse à chacune un délai différent avant de ré-émettre la trame entrée en collision.

L'idée de base d'ethernet est de **ré-émettre la trame après un délai aléatoire** avec les caractéristiques suivantes pour le délai :

- l'écart entre deux valeurs possibles du délai est supérieur au

temps de propagation sur le bus. Ainsi, en l'absence d'autre traffic on est certain que les deux stations ré-emettent sans collision.

- les délais sont de plus en plus longs à chaque nouvelle collision pour la même trame ;

Algorithme de ré-émission : description plus formelle

L'algorithme **BEB** (Binary Exponential Backoff) est défini par :

- après le Nième collision (pour la même trame),
  - si N=16, abandon et signaler erreur,
  - i = min(N,10) (le délai double chaque fois, jusqu'à N=10),
  - tirer un entier aléatoire "na" dans l'intervalle [0, $2^i - 1$],
  - délai = na * ie (ie est l'intervalle élémentaire, 51,2μs).

Algorithme de ré-émission : description

```
-----[xxxxxx]-----|-----|-----|-----|-----|-----|
        ^    \____ ____/|                      |
        |         V     |<----- 4 tranches ---->|
  tranche     2 tranches        essayées
  avec        essayées          au second tour
  collision   au 1er tour

après collision chaque station A et B impliquée tire {0,1}
aléatoirement pour ré-émettre dans une des 2 tranches
suivantes ==> 4 cas :  A    B
                       0    1     A réémet, puis B
                       1    0     B réémet, puis A
                       0    0     nouvelle collision
                       1    1     nouvelle collision
si nouvelle collision on passe à 4 tranches => 1 chance sur 4
seulement de faire une troisième collision
```


Algorithme de ré-émission : description

D'où vient ce délai de 51,2μs ?

On a vu que pour être certain de détecter une collision, une trame devait avoir une longueur minimale égale au double du temps de propagation entre les 2 stations les plus éloignées (worst case).

Pour un ethernet à 10mégabits/sec, 51,2μs correspond à l'émission de 512 bits (on dit parfois 512 temps bits). Ce qui donne 64 octets (14 de préambule, 46 de data et 4 de CRC).

De plus, il ne fallait pas que ce délai soit trop long. Le choix du délai était donc aussi le choix de la distance maximum d'un réseau ethernet (environ 2500 m). Cette durée d'aller-retour maximum est

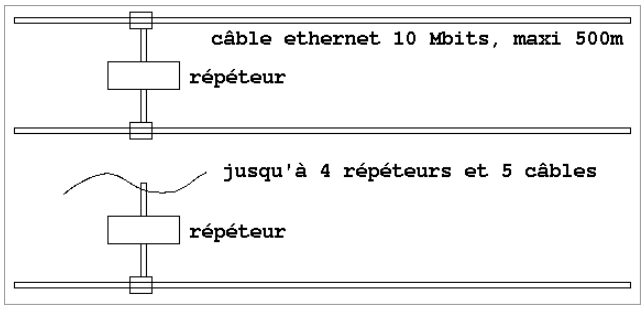
appelée "tranche canal".

La relation entre la longueur du réseau et la durée d'une trame est bien sûr une conséquence de la vitesse de propagation du signal dans le câble de cuivre : environ 2/3 de la vitesse de la lumière.

- vitesse de la lumière dans le vide = 300 000 km/s,
- vitesse du signal électrique dans le câble = 200 000 km/s,
- longueur occupée par un bit dans un ethernet à 10 Mbits/s = (200,000,000 m/s) / (10,000,000 bits / s) = 20 mètres,
- temps d'aller-retour sur un câble de 500 m = 2 * 500 / 200,000,000 = 5 10^{-6} s = 5 μs, soit le temps d'émettre 10 000 000 * 5 10^{-6} = 50 bits.

On a calculé 50 bits, et on a dit que la norme a choisit 512 bits ! Pourquoi ?

Si la longueur maximum d'un câble est de 500m, la norme précise aussi qu'il est possible d'étendre cette distance jusqu'à 2500m en utilisant 4 répéteurs.



Il faut donc tenir compte, en plus de la propagation sur les câbles, de la durée de traversée des répéteurs.

De plus, les concepteurs d'ethernet, ont choisi des spécifications en général deux fois plus strictes que nécessaire pour pallier aux marges d'erreurs dans la construction des équipements à l'époque de la création de la norme.

Ainsi quand on construit un ethernet, on peut se permettre des "petites" violations de la norme (un câble un peu trop long ici, un répéteur de trop ailleurs, etc ...). Il fonctionnera quand même correctement.

Nombre maximum de trames sur un ethernet

La durée d'émission d'une trame étant directement liée à sa taille, il faut faire le calcul pour les plus petites trames possibles et pour les plus grosses :

- **trame mini** : 8+14+46+4 = 72 octets = 576 bits.
- **trame maxi** : 8+14+1500+4 = 1526 octets = 12 208 bits.

à chacune il faut ajouter le délai inter-trames de 96 bits, soit :

- **trame mini** : 576 + 96 = 672 bits.
- **trame maxi** : 12 208 + 96 = 12304 bits.

d'où le nombre de tramesmaxi :

- **plus petite trame** : $10^6/672$ = 14 880 trames/sec.
- **plus grosse trame** : $10^6/12304$ = 812 trames/sec.

On peut alors calculer la quantité de données LLC transportées en supposant qu'il n'y ait pas de collision :

- **plus petite trame** : 14 880 * 46 = 684 480 octets/sec.
- **plus grosse trame** : 812 * 1500 = 1 218 000 octets/sec.

Soit, avec une occupation du média à 60% de **0.5 à 0.7 Mo/s**.

Remarque : pour des données utiles de 512 octets, on trouve environ 2200 trames par seconde.

Avantages et défauts d'ethernet

Avantages

- logique simple,
- réalisation décentralisée, symétrique,
- ne conduit pas à des blocages,
- il existe de nombreux fournisseurs indépendants,
- les délais de transmission sont courts sur un bus peu utilisé.

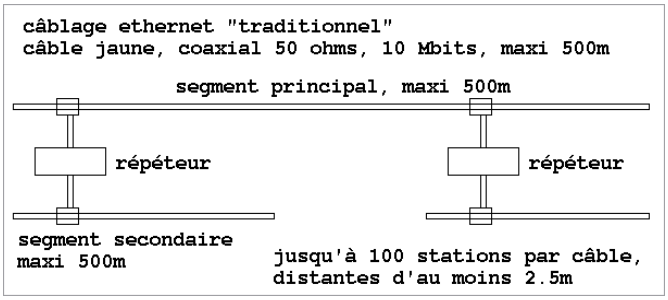
Défauts

- ne permets pas de garantir qu'une station pourra émettre une donnée dans un délai déterminé,
- le calcul du délai aléatoire repose sur la connaissance d'un majorant du délai maximal de propagation qui donne aussi le calcul de la longueur minimale de la trame :
  - si calculé trop large, l'efficacité est réduite,
  - aussi il est fixé par la norme, ce qui limite la dimension maximale du réseau ;

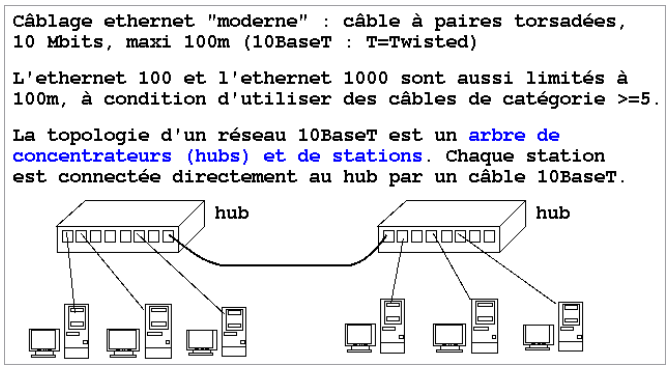
Depuis la publication de la norme, l'apparition de nouveaux équipements (ponts ("bridge") et routeurs, fibres optiques, ...) a permis d'étendre les réseaux ethernet pratiquement autant que souhaité.

Configuration de réseaux ethernet ou 802.3

L'ethernet d'origine : câbles et répéteurs



L'ethernet aujourd'hui : 10BaseT, hubs et switchs



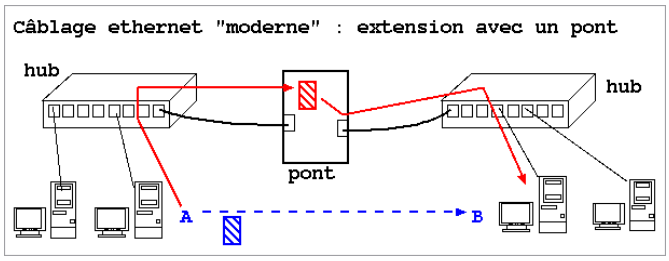
Un "hub" est un répéteur : quand un message arrive sur l'un de ses ports (matétialisé par une prise RJ45), ce message est répété sur tous les ports du hub. Le hub ce comporte exactement comme un câble unique. Toutes les stations peuvent se parler sans intermédiaire. Le hub constitue aussi un "domaine de collision".

Si on relie entre eux deux hubs, on étends le domaine de collision.

Extension d'un ethernet par un pont ("bridge")

Un pont est un équipement plus sophistiqué qu'un simple répéteur. Il possède une mémoire, une unité centrale et (au moins) deux interfaces ethernet.

Extension d'un ethernet par un pont ("bridge")



Fonctionnement : quand une station A veut envoyer un message à une station B, située "de l'autre côté" du pont, c'est le pont qui va récupérer **complètement** le message, puis le **ré-émettre** sur son autre interface, en faisant croire à B que le message à été émis directement par A.

Les stations A et B n'ont aucun moyen de savoir qu'il y a entre elles un pont qui a lu le message et l'a ré-émis. Ceci permet d'insérer un pont dans un réseau, sans rien modifier sur les stations déja connectées.

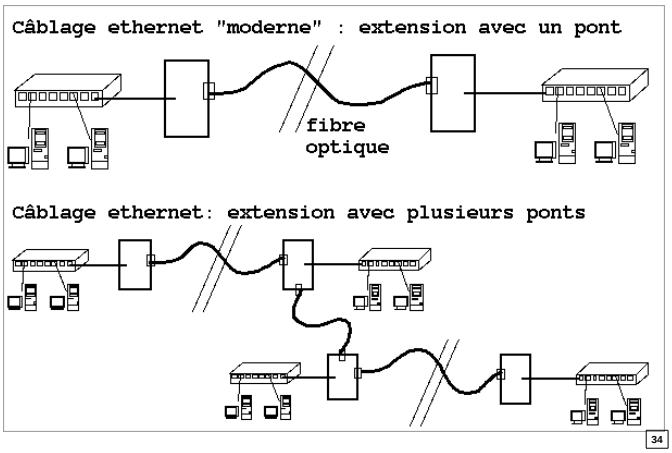
Le pont est suffisamment "intelligent" pour ne pas retransmettre les collisions, séparant ainsi le réseau en deux domaines de collision. De plus, comme le message en transit est intégralement copié dans la mémoire du pont, le calcul du délai de 51,2μs "repart à zéro"

de l'autre côté du pont, permettant ainsi une grande extension du réseau. En particulier par l'utilisation de plusieurs ponts connectés entre eux par des fibres optiques.

Comment le pont "sait-il" de quel côté se situe une machine ? En fait il le sait par auto-apprentissage : comme toute trame émise contient l'adresse ethernet de la source, il peut très vite remplir une table contenant toutes les adresses sources ayant émis des trames et les associer à l'interface sur laquelle il a "vu" cette source.

Le pont se met dans un mode spécial dit "promiscuous" : il écoute **toutes** les trames qui circulent sur chacun des réseaux et ne retransmet de l'autre côté que celles qui y sont destinées.

Extension d'un ethernet par des ponts



Utilisation de commutateurs ethernet ("switch")

Un commutateurs a le même aspect qu'un hub : une boîte comportant des prises RJ45 (de 4 à 48 suivant les modéles). Il existe de plus gros commutateurs pouvant contenir plusieurs centaines de prises : ils sont constitués de chassis dans lesquels on insère des cartes de 24 ou 48 prises.

La différence entre un switch et un hub est bien sûr à l'intérieur : le commutateur est une sorte de pont généralisé : chacun des ports du commutateur se comporte comme un port de pont. Les paquets qui arrivent sur un port sont stockés dans la mémoire du commutateur et mis dans la file d'attente du port de destination en sortie.

Passage à l'ethernet 100

À 10 Mbps durée de trame mini est 51,2μs. À 100 Mbps il faudrait 640 octets pour remplir cette durée, or la plupart des trames sont petites, donc il faudrait beaucoup de bourrage entrainant une sous-utilisation du réseau.

Ceci implique de réduire la durée maxi de propagation pour compenser, donc la distance maxi du réseau.

Le choix de la norme ethernet-100 est de passer la tranche canal à 5,12μs et de garder une taille de trames identique à ether-10 et de diviser les distances par 10.

En pratique, l'ethernet 100 n'existe que sur des câbles à paires torsadées de catégorie supérieure ou égale à 5 sur lesquelles la

distance utilisable est de 100m.

Un autre avantage de ce choix de diminuer la tranche canal en gardant la même taille de trame et que les équipements peuvent traiter sans modification des trames émises à 10 et des trames "100". Ainsi il est possible de construire des commutateurs ou des hubs qui gèrent un mélange d'équipements "10" et d'équipements "100".

Différence entre débit nominal et débit utile, différence entre LAN et WAN

- **LAN** : temps propagation courts : on peut utiliser ce délai court dans la définition des protocoles ;
- **WAN** : temps propagation longs => en général, plusieurs paquets sont émis avant de recevoir une information en retour => protocoles différents.

Soit "a = P / T" le rapport du temps de propagation (temps mis par le premier bit pour atteindre la destination) sur le temps de transmission (temps mis par la source pour envoyer le paquet).

Le média est occupé depuis le premier bit envoyé (posé sur le média) jusqu'à la réception du dernier bit par le destinataire, soit : **occupation = transmission + propagation**.

Le pourcentage d'utilisation du média est : U = T / (T+P) (rapport de la partie utile, la transmission, sur l'occupation).

Soit U = 1 / (1 + P/T) = 1 / (1+a)

Les schémas ci-dessous illustrent la formule dans les deux cas où le temps de propagation est >> temps de transmission (cas le plus fréquent dans les WAN haut débit), et celui où le temps de propagation est << au temps de transmission (cas le plus fréquent dans les LAN).

```
        début de transmission          début de transmission
        xxx]->                         xxx]->
t0      |--------------------|  t0      |--------------------|
        []          LAN     []         []            WAN     []
```



```
        début de réception             fin de transmission
        xxxxxxxxxxxxxxxxxxxx]->        [xxxxxxx]->
t0+P    |--------------------|  t0+T   |--------------------|
        []          LAN     []         []            WAN     []

        fin de transmission            début de réception
        [xxxxxxxxxxxxxxxxxxxxx->                        [xxxxxx]->
t0+T    |--------------------|  t0+P   |--------------------|
        []                  []         []                  []

        fin de réception               fin de réception
                            [x->                           [x->
t0+T+P  |--------------------|  t0+T+P|--------------------|
        []                  []         []                  []
```

Le média est libéré quand le dernier bit du message est retiré par le destinataire. Ce dernier bit entre sur le réseau à t0+T et en ressort "P" secondes plus tard.

Application numérique

débit Mbps	taille trame (bits)	long réseau (km)	a	U
1	100	1	0.05	0.95
1	1000	10	0.05	0.95
1	100	10	0.5	0.67
10	1000	1	0.05	0.95
10	100	1	0.5	0.67

100	10000	1	0.05	0.95
100	1000	1	0.5	0.67
100	100	1	5.	0.17
100	10000	10	0.5	0.67
100	1000	10	5.	0.17
100	1000	50	25.	0.04

Exemple :
débit 10 Mbits, trame de 100 bits =>
T=100*10*10^-6=10^-5.
P=D/V= 1 / 200 000 = 0.5 10^-5
a = P/T = 0.5

Ainsi, on voit que lorsque le débit et la distance augmentent, il devient très difficile d'utiliser correctement le réseau.

Sur un WAN il faut utiliser des protocoles qui permettent d'envoyer plusieurs trames en même temps sur le réseau :

```
     [xxx]->          [xxxxx]->  [xxxxxx]->          [xxx]->
t0   |-------------------------------------------------|
```

Application numérique

En 1 nanoseconde, la lumière parcours $300 * 10^6/10^9 = 0,3m$ ... *dans le vide*.

Dans une fibre optique d'indice 1,45, la vitesse est de 210 000 km/s. La distance parcourue en une nanoseconde est de 0,21m.

Pour parcourir une fibre optique de 6000 km, il faut à la lumière 30 ms, soit 60ms d'aller-retour.

Si dans une fibre optique gérée à 1 Gigabits/sec on envoie des paquets de 576 octets, soit 4608 bits, en 30ms, on a le temps de déposer 6510 paquets, soit plus de 3 Mo utiles !

Si, à la réception du premier paquet qui ressort de la fibre , le destinataire envoie un ordre "stop", cet ordre d'arrêt ne parvient à l'émetteur qu'une fois que celui-ci à déja envoyé 6 mégaoctets !

Donc, on ne peut pas "remplir" une liaison de ce type avec une seule connexion gérée par un protocole qui implique des échanges entre émetteur et récepteur.

Pour "remplir" les liaisons à très haut débit, on va **multiplexer** un très grand nombre de liaisons à bas débit afin d'avoir toujours quelque chose d'utile à transférer. La gestion de ce type de liaison est le **métier des opérateurs** de télécom, assez différent de celui de la gestion de réseaux locaux.

Par exemple, pour "remplir" les fibres utilisées aujourd'hui au débit de 1 térabit/s, on multiplexe par équipement WDM (Wave Division Multiplexing) 40 circuits à 2.4 gigabits, eux-mêmes constitués du multiplexage de liaisons à plus bas débit.

Vous pourriez faire remarquer, que la distance étant toujours présente, la liaison bas débit tout en bas de la chaine n'est pas non plus "remplie". D'abord, du fait qu'elle est à plus bas débit, la perte est moindre, mais surtout, l'opérateur se moque qu'elle soit sous-utilisée car il a vendu la totalité du débit !

Quand on achète 512kbits à un opérateur, on paye la même chose que l'on s'en serve ou pas !

Ch05 - Réseau local ethernet. Exemples de questions

➤ Comment s'effectue l'échange de données sur un réseau ethernet ?
Réponse : diapos 3 et 4.

➤ Dessiner une trame ethernet.
Réponse : diapo 5.

➤ Selon quel protocole fonctionne un réseau ethernet et quel est le principe de ce protocole ?
Réponse : diapos 8 à 10.

➤ Montrer, à l'aide d'un diagramme, pourquoi les trames ethernet doivent avoir une longueur minimale.
Réponse : diapos 14 et 15.

➤ Quel est l'algorithme de ré-émission d'un réseau ethernet ?
Réponse : diapos 18 et 20.

➤ Pourquoi l'utilisation d'un pont pour étendre un réseau ethernet permet-elle de s'affranchir du délai intertrame ?
Réponse : diapos 31 et 33.

➤ Quel est l'élément clé qui différencie un "LAN" d'un "WAN" ?
Réponse : diapos 38 et 39 : la longueur (distance entre les stations).

Ch05 - Réseau local ethernet.

Fin du chapitre.

Ch06 - Introduction au concept d'internet

On a vu que :

communication à distance entre deux programmes **==> complexité**.

Les concepteurs disposent de deux approches pour masquer cette complexité :

- utiliser les programmes de niveau "application",
- cacher les détails dans le système d'exploitation.

La première approche conduit à des communications peu efficaces (on a vu qu'il n'est pas si simple d'utiliser au mieux la bande passante d'un réseau à haut débit). De plus, l'ajout de fonctions nouvelles conduit à modifier les programmes sur chaque noeud du réseau.

Si on ne dispose pas de service de **transport de type "réseau"**, les programmes applicatifs doivent se charger de la **communication de proche en proche**. Or, l'expérience a montré que cette méthode fonctionne très mal sur de grands réseaux (elle n'est **pas "scalable"**, ne peut pas "passer à l'échelle").

L'alternative est de baser les communications entre programmes sur un **système d'interconnexion qui opère au niveau du réseau**.

C'est un mécanisme, inclus dans le S.E. qui assure en temps réel le contrôle de l'acheminement des paquets de données entre l'application émettrice et l'application destinataire.

Ce mécanisme va commuter de "petits" paquets de données plutôt que des messages "longs" (petits = 100 à 1000 octets).

Ceci présente plusieurs avantages :

- une correspondance directe avec les réseaux physiques sous-jacents (ethernet, FDDI, X25, ATM, ...),
- séparer l'activité de communication des programmes applicatifs, ce qui permet aux machines intermédiaires de faire leur travail sans avoir besoin de comprendre les informations de niveau applicatif,
- conserver de la souplesse au système (modifier les applis ou ajouter de nouveaux protocoles).

Le concept de base d'un internet est celui d'un système de communication abstrait dissociant la notion de communication des détails techniques des réseaux et masquant ces détails de bas niveau aux programmes d'application. Ce système abstrait repose sur la technique d'interconnexion de réseaux (internetworking).

Cette notion d'interconnexion résulte de deux constats :

- un seul type de réseau ne peut satisfaire tous les besoins (différences entre LAN et WAN),
- on souhaite pouvoir interconnecter tout type de réseau.

Architecture d'un internet

Les utilisateurs ou les programmes d'application ne doivent pas avoir à connaitre les détails des connexions physiques pour pourvoir utiliser un internet.

On doit pouvoir connecter un nouveau réseau à un internet existant sans avoir besoin de le connecter à un ordinateur central.

Tous les ordinateurs de l'internet doivent partager un ensemble d'identificateurs universels.

Comment des réseaux différents sont-ils connectés pour former un internet ?

Un internet viable nécessite des équipements spécialisés capables de faire passer des paquets de données d'un réseau à un autre.
==> Ces équipements sont appelés "passerelles IP" ou "routeurs IP" (Internet gateway ou Internet router).

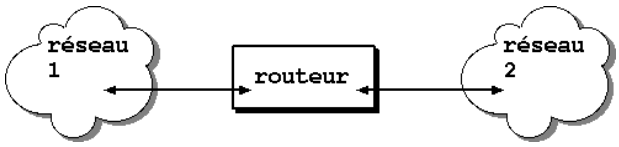


FIG. 1: Réseaux interconnectés par un routeur

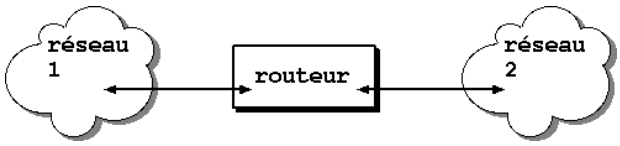


FIG. 2: Réseaux interconnectés par un routeur

Ici le routeur [R] prends des blocs de données sur le réseau 1 qui sont destinés à des machines du réseau 2, et les transmet sur le réseau 2 (en faisant si nécessaire les conversions de trames si les réseaux ne sont pas physiquement de même type) (et symétriquement dans l'autre sens).

Dans un internet plus complexe que celui de la fig. précédente, chaque routeur doit connaître des données relatives à la topologie générale du réseau bien au-delà des réseaux qui lui sont immédiatement raccordés.

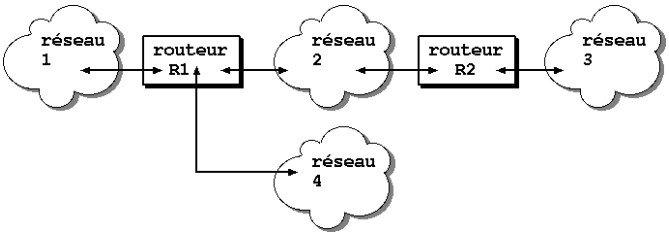


FIG. 3: Réseaux interconnectés par des routeurs

Dans ce deuxième exemple, R1 doit savoir que les données à destination du réseau "3" doivent être envoyées vers le réseau 2 ; de même, R2 doit savoir que les données qu'il reçoit pour les réseaux 1 et 4 doivent être envoyées sur le réseau 2.

Quand l'internet grandit, la tâche du routeur qui doit choisir la route que vont prendre les paquets qui transitent par lui devient plus complexe.

Un internet est un ensemble de réseaux interconnectés par des équipements spécialisés appelés **routeurs** qui assurent le service d'acheminement des données d'une source quelconque vers une destination quelconque de façon transparente pour la source et la destination.

On pourrait craindre que dans un internet contenant des millions de machines chaque routeur doive connaître la lccalisation de l'ensemble des machines.

En fait non, l'adressage hiérarchique de IP permet à chaque routeur de n'avoir à connaître que des numéros de réseaux (et même dans les versions récentes des ensembles de réseaux que l'on a rendus "contigus" au sens de la topologie du réseau). Ainsi les routeurs peuvent être réalisés avec de petits ordinateurs.

La façon dont les routeurs apprennent et mettent à jour leurs connaissances des routes vers les autres routeurs de l'internet (protocoles RIP, BGP, EGP) fait partie des cours sur les réseaux.

Les protocoles TCP/IP ont été conçus pour que les utilisateurs puissent se représenter l'internet comme un simple moyen de communication entre toutes les machines participantes, quel que soit le sous-réseau sur lequel elles sont effectivement branchées.

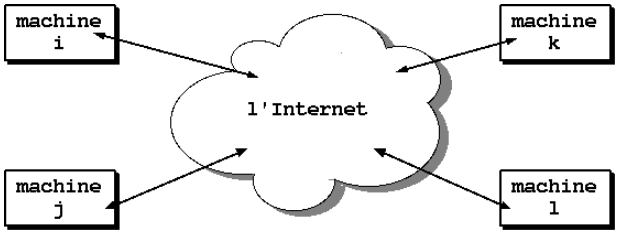


FIG. 4: L'Internet permet d'interconnecter des machines

Il est possible, en utilisant correctement les protocoles TCP/IP, d'écrire des programmes d'application qui fonctionneront entre l'ordinateur [j] et l'ordinateur [k], sans avoir à connaître le détail des sous-réseaux (éventuellement nombreux) qui les séparent.

Il est même possible de modifier la structure du réseau sous-jacent et de continuer à utiliser les programmes d'application sans avoir besoin de les recompiler, ni même de les arrêter.

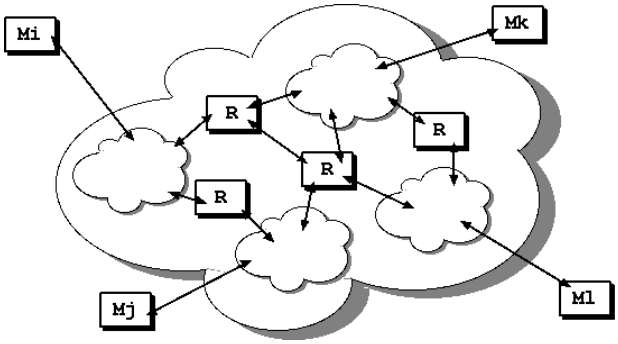


FIG. 5: L'Internet est un réseau de réseaux

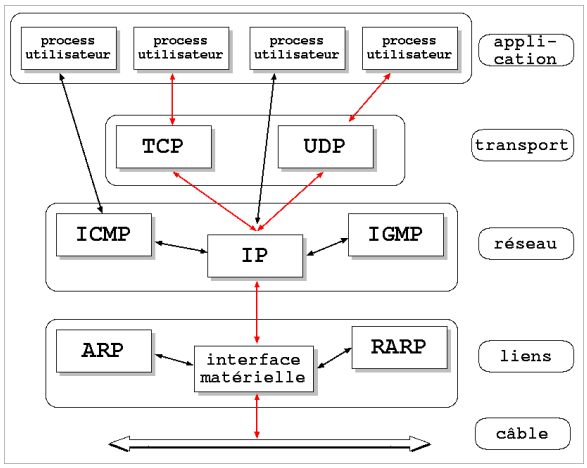
Les routeurs d'un internet n'assurent pas une connectivité directe d'un réseau quelconque vers un autre réseau quelconque. Le traffic entre deux réseaux peut emprunter un ou plusieurs réseaux intermédiaires.

Les réseaux qui participent à un internet doivent accepter du traffic "de transit" en plus de leur traffic propre. En contrepartie il peut écouler son propre traffic en utilisant les autres réseaux de l'internet.

Nota : en pratique l'Internet est constitué de réseaux de transport gérés par des opérateurs qui passent entre eux des accords de traffic croisé (accord de "peering") tenant compte des quantités de traffics échangés et donnant lieu éventuellement à des compensations financières (les plus gros font payer les petits, comme d'habitude).

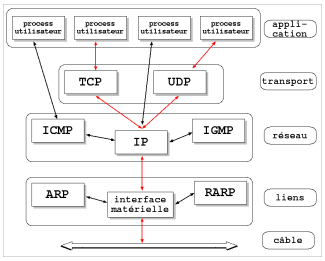
Les réseaux des fournisseurs d'accès (ISP : Internet Service Provider, FAI en français) sont connectés en bordure de ces réseaux de transport (moyennant finances évidemment).

Stratification en couches de TCP/IP



Stratification en couches de TCP/IP

ARP Address Resolution Protocol
RARP Reverse Address Resolution Pro.
ICMP Internet Control Message Pro.
IGMP Internet Group Management Pro. (multicast)
IP Internet Protocol
UDP User Datagram Protocol
TDP Transmission Control Protocol



- **Couche lien :** carte réseau + driver gèrent interface avec le média (LLC+MAC).
- **Couche réseau : IP :** gère les échanges de paquets, envoie les paquets directement au destinataire ou à une passerelle (gateway) en fonction des informations de routage (netstat -r) dont elle dispose,
- **Couche transport : TCP :** gère le flux de données
  – cette couche envoie à IP des paquets de taille ad-hoc,
  – fait la gestion des acquittements, établit des temporisations pour s'assurer que tous les paquets sont acquittés (garantie de transmission fiab le),
  – UDP, elle, fournit un service plus simple, non fiab le, sans ACK.

Ces 3 couches fonctionnent en **mode système** et gèrent les détails

de la communication.

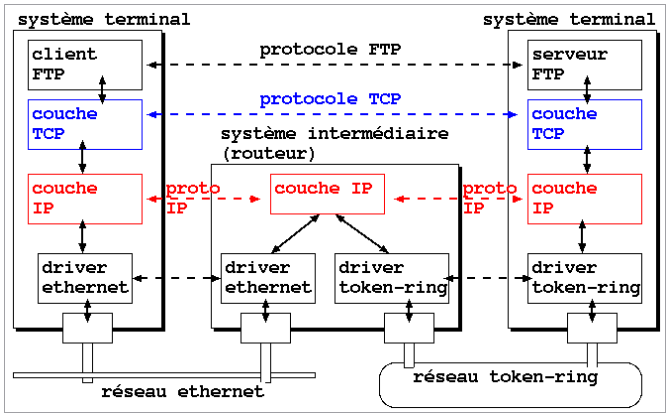
Couche applicative : elle fonctionne en mode utilisateur. Exemples : Telnet, rlogin, FTP, SMTP (mail), SNMP, HTTP, ...

Beaucoup d'applications sont non-symétriques : le côté client et le côté serveur sont différents.

Couches liens, IP et routage

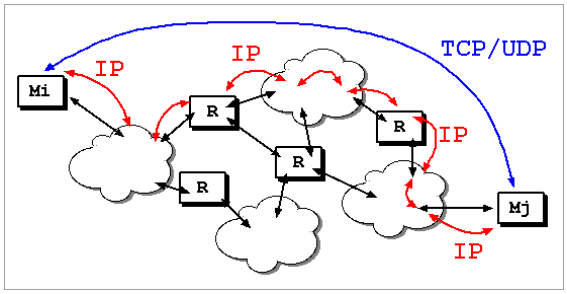
La couche IP et la couche liens ne font pas seulement la liaison entre deux machines, mais aussi entre deux réseaux en utilisant un équipement équipé de deux cartes d'interface (de type éventuellement différent : par exemple une carte ethernet et une carte token-ring). Cet équipement est appelé un **routeur**.

Un routeur fonctionne sur la couche IP



Couches IP et routage

Le protocole IP est dit protocole de saut en saut (hop by hop), alors que les protocles TCP (ou UDP) sont dits "protocoles de bout en bout" (end to end).



Les adresses IP

Dans un internet les adresses de toutes les machines participantes doivent être **uniques**.

==> ceci implique une cohérence dans les attributions.

==> autorité globale : Inter-NIC (Network Information Center).

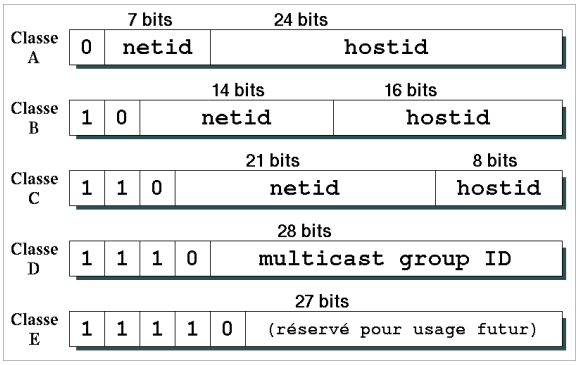
cette autorité **délègue** la gestion de zones géographiques à des autorité locales, en leur confiant la gestion d'un**groupe d'adresses**.

Les adresses IP sont de trois types :

- unicast : concerne **une** machine,
- broadcast : concerne **toutes** les machines **d'un** réseau,
- multicast : concerne **un ensemble** de machines (les membres d'un groupe multicast).

Les adresses internet (IP)

A	0.0.0.0	127.255.255.255		D	224.0.0.0	239.255.255.255
B	128.0.0.0	191.255.255.255		E	240.0.0.0	247.255.255.255
C	192.0.0.0	223.255.255.255				



Les noms de domaines et de machines : le DNS

Toute machine a donc une adresse unique. Toutefois, les machines sont utilisées par des humains qui manipulent plus facilement des noms que des numéros.

Dans le monde TCP/IP, le **service de noms de domaines** (**DNS**) est une application qui gère une base de donnée **distribuée** qui associe **un nom unique** à chaque adresse.

Des fonctions du système d'exploitation (**gethostbyname**) permettent de passer des noms aux adresses et inversement.

L'espace de noms du DNS est une **structure arborescente**. Chaque noeud de l'arbre possède une étiquette (un nom), la liste des étiquettes depuis la racine désigne un sous-domaine.

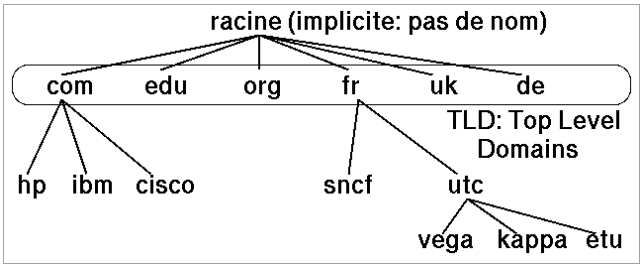


FIG. 6: L'arbre du domaine de noms

À chaque noeud de l'arbre est associé un ensemble d'informations constituant le "**Ressource Record**" (RR). Ce peut être une adresse IP (pour une machine) ou la désignation d'un serveur de courrier (exemple du noeud "etu" dans le schéma ci-dessus).

Le DNS est maintenu par un ensemble de **serveurs de noms** ("name servers") : chacun d'eux se voit déléguer une **autorité** pour des parties de l'espace des noms.

Une requête de type "**SOA**" (Source Of Authority) donne le serveur qui a cette délégation pour un domaine :

```
$ dig -tSOA kappa.utc.fr
...
;; AUTHORITY SECTION:
utc.fr.                 299     IN      SOA     orion.utc.fr.
```


Quand un programme a besoin d'une information DNS (traduire un nom en adresse ou l'inverse), il interroge un programmme résolvant (le "resolver") qui lui-même interroge un serveur DNS.

Le résolver trouve l'adresse du serveur à interroger dans le fichier **/etc/resolv.conf** (voir man resolv.conf).

Si ce serveur DNS ne connait pas la réponse, il interroge le serveur DNS de niveau supérieur qui se chargera de transmettre la question.

Soit la réponse sera trouvée dans un cache, soit la question atteindra un serveur DNS ayant autorité pour le domaine concerné par la question.

Les programmes gérant le DNS (serveurs et résolveurs) utilisent les ports **TCP 53** et **UDP 53**. Il faut donc "laisser passer" ces 2 ports dans la configur ation d'un routeur pare-feu.

Le DNS est en général implémenté par le programme **BIND**, issu de la distribution unix de Berkeley. Ce programme BIND s'incarne dans le démon **named**.

Les étiquettes (noms des noeuds) sont définies sur au plus 63 caractères, dont le premier désigne la longueur (donc 62 car. utiles). **Mais** la longueur totale d'un nom de domaine (par exemple kappa.utc.fr) ne doit pas dépasser 255 caractères. D'où la tendance des administrateurs à choisir des noms courts, sinon on limite implicitement la profondeur de l'arbre.

Majuscules et miniscules sont **interchangeables** dans un nom DNS : KAPPA.utc.fr est la même chose que Kappa.UTC.fr.

Exemple :

```
$ host kappa.utc.fr
kappa.utc.fr has address 193.55.117.100

$ host -a kappa.utc.fr
Trying "kappa.utc.fr"
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 42370
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 5, ADDITIONAL: 4

;; QUESTION SECTION:
;kappa.utc.fr.                  IN      ANY

;; ANSWER SECTION:
kappa.utc.fr.           7194    IN      A       193.55.117.100
```



```
;; AUTHORITY SECTION:
utc.fr.                 7194    IN      NS      ns0.pasteur.fr.
utc.fr.                 7194    IN      NS      ns1.pasteur.fr.
utc.fr.                 7194    IN      NS      orion.utc.fr.
utc.fr.                 7194    IN      NS      epsilon.utc.fr.
utc.fr.                 7194    IN      NS      ns.crihan.fr.

;; ADDITIONAL SECTION:
ns.crihan.fr.           86015   IN      A       195.221.20.10
ns.crihan.fr.           78556   IN      AAAA    2001:660:7401:201::10
orion.utc.fr.           6815    IN      A       195.83.155.16
orion.utc.fr.           7194    IN      AAAA    2001:660:4201:1::2
```



```
Received 244 bytes from 193.252.19.3#53 in 121 ms
```

Fonction inverse

Pour retrouver un nom à partir d'une adresse, il a été créé un pseudo-domaine **IN-ADDR.ARPA** dans lequel les noeuds sont étiquettés par leur adresse IP, lue à l'envers (pour avoir le numéro de réseau près de la racine). Ainsi, kappa.utc.fr est repésentée par :

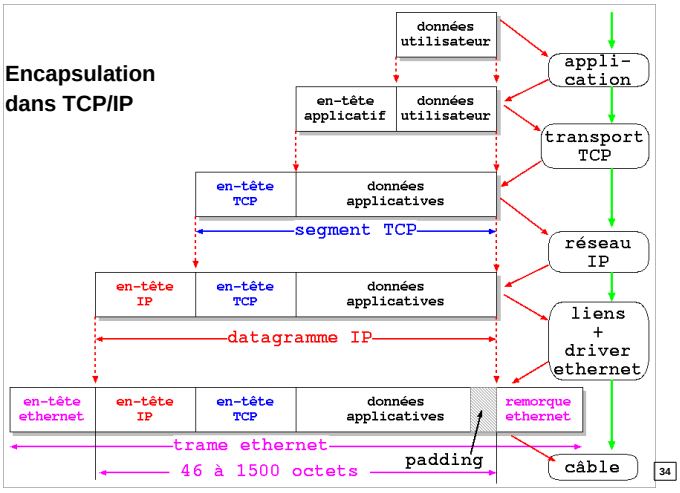
```
$ host 193.55.117.100
100.117.55.193.in-addr.arpa domain name pointer kappa.utc.fr.
```

Encapsulation dans TCP/IP

Chaque couche de la pile TCP/IP recoit des "**données utiles**" de la couche supérieure. Elle **encapsule** ces données utiles dans une structure de données dans laquelle elle **ajoute** les informations qui seront utiles à son **homologue** (i.e. la couche de même niveau qu'elle) dans la machine de destination.

Cette structure de données devient elle-même les "**données utiles**" de la couche inférieure.

Encapsulation dans TCP/IP



Au niveau de la couche liens, IP transmet à l'interface réseau un **paquet** qui peut être :

- soit un datagramme complet (cas le plus fréquent),
- soit un **fragment** de datagramme (si le **MTU** du réseau est trop petit) (MTU = Maximum Transfert Unit).

TCP, UDP, ICMP et IGMP envoient tous des données à IP.

==> à l'arrivée il faudra rendre à chaque service les données qui sont les siennes.

==> pour cela, IP distingue l'origine de chaque paquet par le champ **proctocole** de l'en-tête IP : ICMP=1, IGMP=2, TCP=6 et UDP=17.

De même, TCP (ou UDP) peuvent recevoir des données de plusieurs applications **==>** ils utilisent pour distinguer chacune des applications la notion de **numéro de port** (entier sur 16 bits).

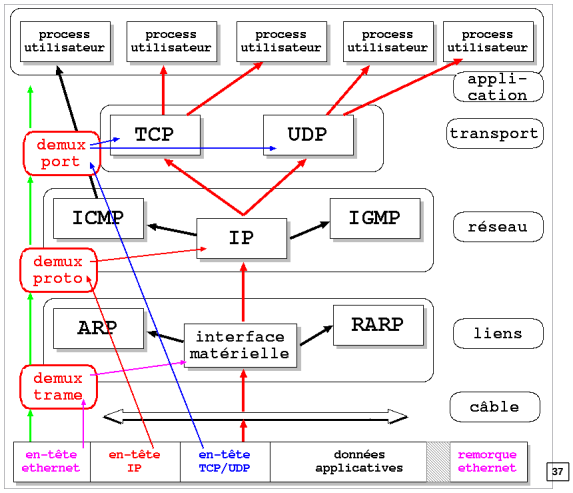
Un certain nombre de ports sont préaffectés à certaines applications : ce sont les **ports bien connus** (well known ports) (liste dans **/etc/services**), par exemple :

- FTP serveur = TCP / 21
- Telnet serveur= TCP / 23
- TFTP serveur = UDP / 69

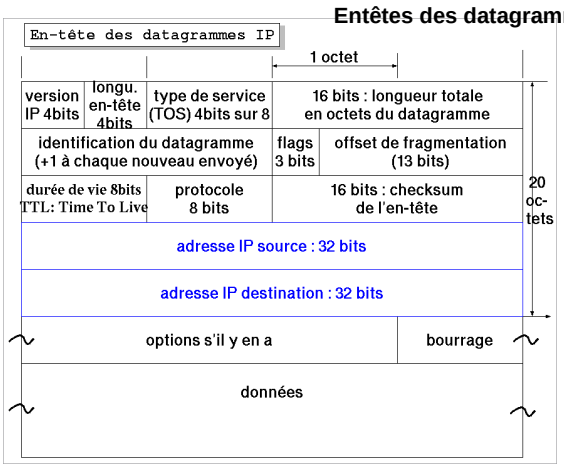
Les ports bien connus sont en général entre 1 et 1023. Ils sont gérés par l'**IANA** (Internet Assigned Numbers Authority).

Sur unix, les ports 1 à 1023 sont réservés aux process systèmes (en "su").

Démultiplexage des trames dans TCP/IP



Entêtes des datagrammes IP



flags :

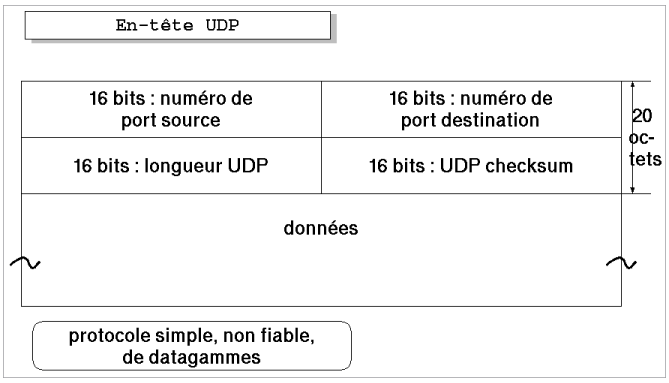
- bit "more fragments" = 1 si il y une suite =0 si dernier fragment,
- bit "don't fragment" : empêcher couper si MTU < taille.

fragment offset : permet de remettre fragments à leur place dans le datagramme complet s'ils arrivent dans le désordre.

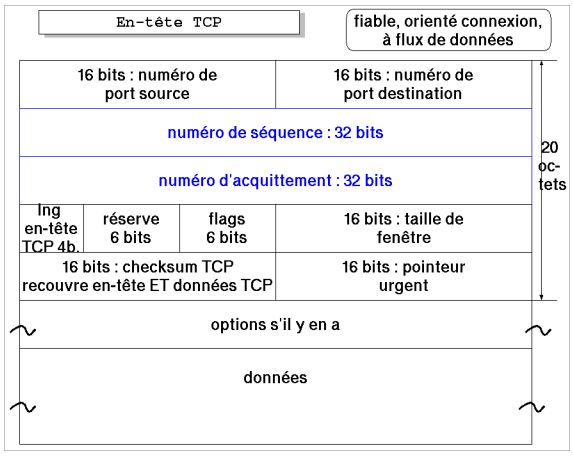
longueur totale : obligatoire car la couche liens peut faire une trame *plus longue* que le datagramme (ex. lng mini ethernet de 46 octets).

Durée de vie : -1 à chaque saut, datagramme détruit silencieusement quand arrive à zéro. Pour éviter que des datagrammes bouclent indéfiniment dans le réseau, suite à des pannes de routeur.

Entêtes UDP



Entêtes TCP



41

flags du paquet TCP :

- bit URG : le pointeur est valide
- bit ACK : le numéro d'acquittement est valide
- bit PSH : (push) donnée à passer à l'application "asap"
- bit RST : (reset) réinitialise la connexion
- bit SYN : synchroniser les numéros de séquence pour initialiser une connexion
- bit FIN : l'émetteur a fini d'envoyer des données

Remarque : les données sont envoyées sur le réseau octet par octet dans le "network byte order" qui est "big endian".

42

Encapsulation ethernet

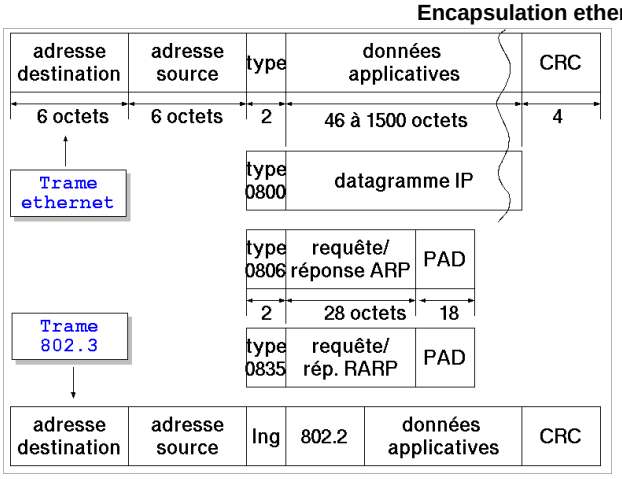
Rappel : sur un réseau ethernet peuvent cohabiter (et interopérer) 2 types de trames :

- ethernet - RFC 894,
- 802.3 - RFC 1042.

```
[root@localhost /root]# arp -a
brx (192.168.130.1) at 40:40:40:40:30:C6 [ether] on eth0
```

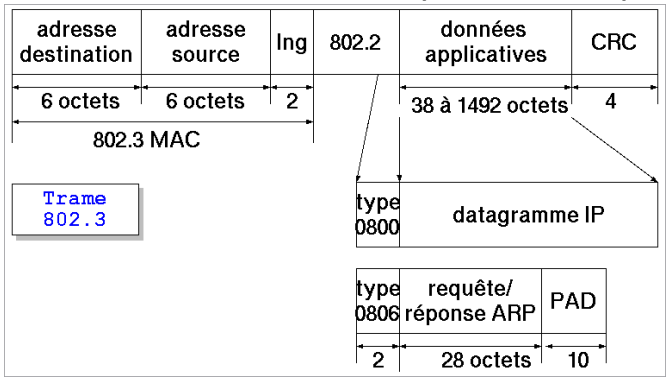

43

Encapsulation ethernet



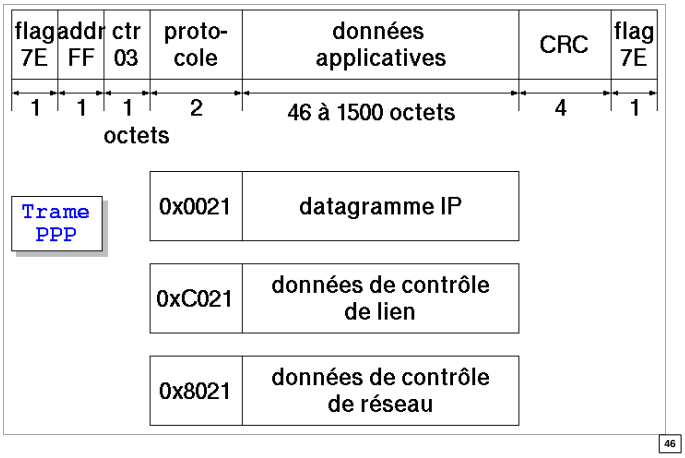
44

Encapsulation ethernet (802.3)



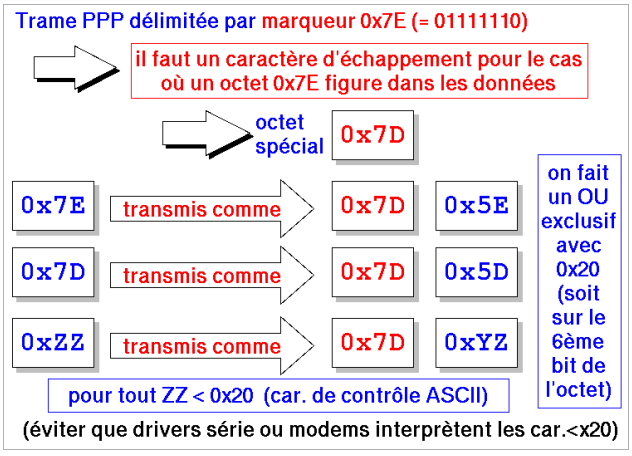
45

Trame du Protocole PPP



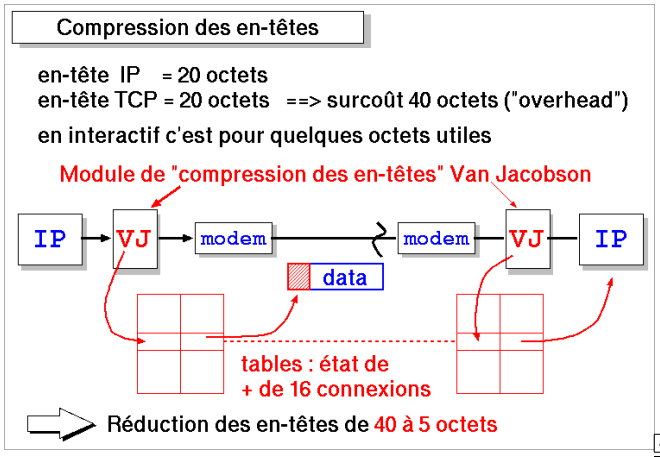
46

Trame du Protocole PPP



47

Compression des en-têtes sur liaison PPP



48

MTU et vitesse de transfert 1/2

Ligne 9600 bits/s. avec 1 bit start et 1 bit stop => 960 octets/s.

Paquet de 1024 octets = 1024/960 = 1066 ms (plus de 1 sec.)

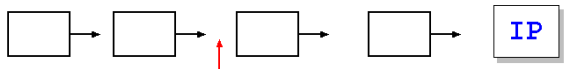
En interactif seul : $\frac{41}{960}$ x 2 = 85 ms pour l'aller-retour

avec la compression VJ: 2 x 6/960 = 12.5 ms

⇨ "temps de réponse" ~= 13 ms : bon

MTU et vitesse de transfert 2/2

interactif + transfert de données simultanés



paquets de données

insertion d'un paquet interactif

même inséré en début de file, il devra attendre en moyenne 1/2 durée de transmission

Si MTU = 1024 => 1066/2 ~= 533 ms → inacceptable !

Si MTU = 256 => 256/(960x2) ~= 133 ms → presque acceptable

⇨ Il faut faire un compromis entre le temps de réponse et l'efficacité (% d'utilisation)

IP : le Protocole Internet

IP est le protocole de base de la pile "TCP/IP". Les données de TCP, d'UDP, d'ICMP et d'IGMP sont transmises comme des datagrammes IP.

IP délivre un service **non fiable, sans connexion**.

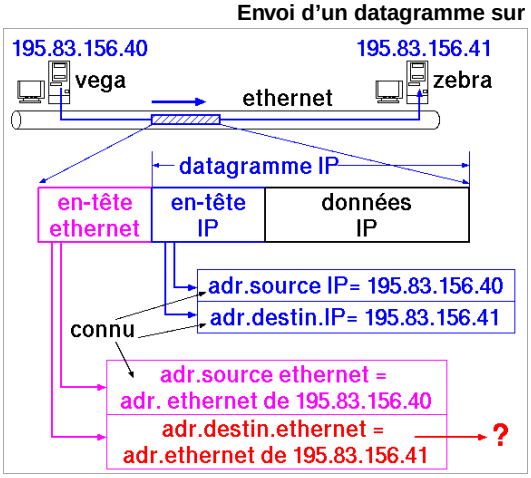
- **non fiable :** pas de garantie qu'un datagramme IP atteigne sa destination avec succès. IP fournit un service "de moindre effort". Toute la fiabilité requise doit être assurée par les couches supérieures (e.g. TCP).
- **sans connexion :** IP ne maintient aucune information d'état concernant les datagrammes successifs. Chacun est géré indépendamment des autres. => ils peuvent être délivrés en désordre (ils peuvent suivre des chemins différents dans le réseau).

Le **routage IP** se fait sur une base de **saut à saut** : IP ne connaît la route complète (de bout en bout) d'aucune destination.

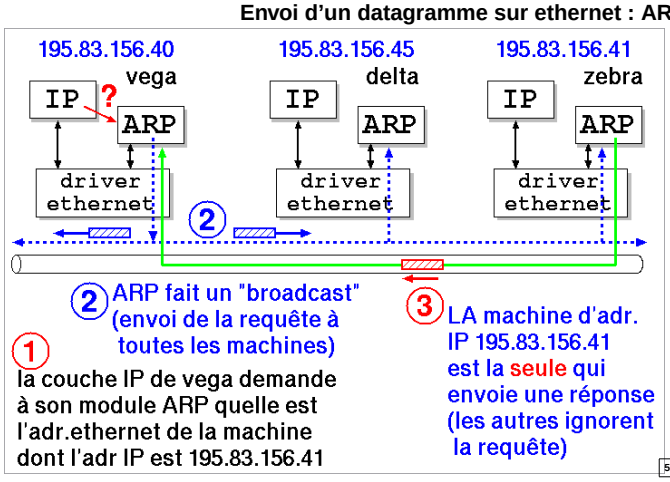
Chaque machine connaît seulement les destinations qui lui sont directement connectées.

La table de routage d'une machine fournit seulement l'adresse IP du routeur de saut suivant, vers lequel le datagramme doit être envoyé.

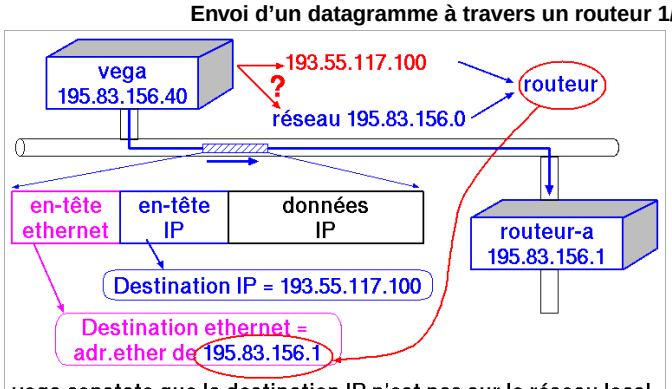
Envoi d'un datagramme sur ethernet



Envoi d'un datagramme sur ethernet : ARP

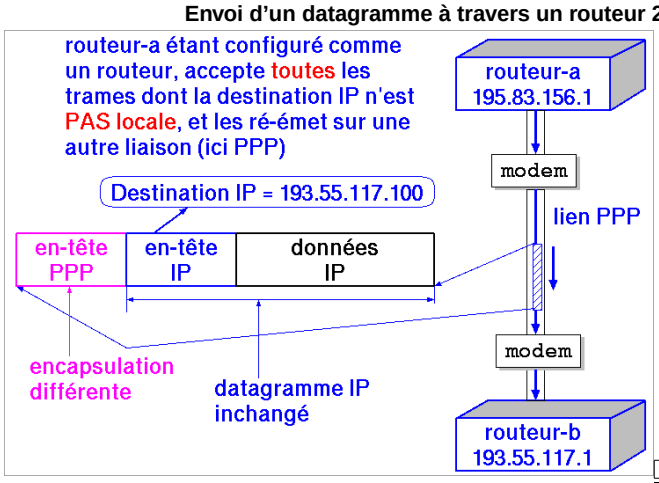


Envoi d'un datagramme à travers un routeur 1/3

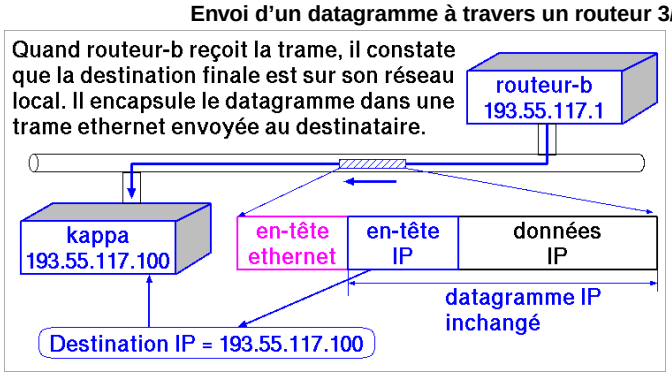


vega constate que la destination IP n'est pas sur le réseau local, alors elle cherche l'adr. IP d'un routeur dans sa table de routage (il y a au moins la "route par défaut", et y envoie le paquet IP.

Envoi d'un datagramme à travers un routeur 2/3



Envoi d'un datagramme à travers un routeur 3/3



Deux commandes unix permettent d'obtenir des informations sur l'état des interfaces réseau (**ifconfig**) et l'état des liens IP (TCP ou UDP) (**netstat**).

```
mv@localhost mv> /sbin/ifconfig
eth0  Link encap:Ethernet  HWaddr 00:00:B4:95:40:CF
      inet addr:192.168.130.5  Bcast:192.168.130.255  Mask:255.255.255
      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
      RX packets:15806 errors:0 dropped:0 overruns:0 frame:1
      TX packets:16511 errors:0 dropped:0 overruns:0 carrier:0
      collisions:2 txqueuelen:100
      Interrupt:3 Base address:0x300

lo    Link encap:Local Loopback
      inet addr:127.0.0.1  Mask:255.0.0.0
```



```
      UP LOOPBACK RUNNING  MTU:3924  Metric:1
      RX packets:62 errors:0 dropped:0 overruns:0 frame:0
      TX packets:62 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:0

277 lo33 b116serv:~> ifconfig -a          (sur Solaris)
lo0: flags=849<UP,LOOPBACK,RUNNING,MULTICAST> mtu 8232
        inet 127.0.0.1 netmask ff000000
hme0: flags=863<UP,BROADCAST,NOTRAILERS,RUNNING,MULTICAST>
        mtu 1500
        inet 172.19.2.90 netmask ffff0000 broadcast 172.19.255.255
```



```
273 lo33 b116serv:~> netstat -r     (sur solaris)
Routing Table:
  Destination           Gateway       Flags Ref  Use  Interface
-------------------- -------------- ----- --- ------ --------
r4.utc.fr             b116serv        U     4  30337  hme0
r4b.utc               b116serv        U     4 147828  hme0
BASE-ADDRESS.MCAST.net b116serv       U     4      0  hme0
default               c6k-msfc-bf.utc UG    0 225581
localhost             localhost       UH    0  36687  lo0

root@localhost /root# netstat -r     (sur linux)
Kernel IP routing table
Destination     Gateway  Genmask       Flags  MSS Win irtt Iface
192.168.130.0   *        255.255.255.0 U       0 0      0 eth0
127.0.0.0       *        255.0.0.0     U       0 0      0 lo
default         brx      0.0.0.0       UG      0 0      0 eth0
```



```
root@localhost /root# netstat -t -p -a    (sur linux)
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (servers and established)
Proto Rcv-Q Snd-Q Local Addr  Foreign Addre State       PID/Prog.name
tcp        0      0 tn5:1238   sirius:telnet ESTABLISHED 26084/telnet
tcp        0      0 tn5:1221   brx:telnet    ESTABLISHED 25736/telnet
tcp        0      0 *:X        *:*           LISTEN      25413/X
tcp        0      0 *:printer  *:*           LISTEN      585/
tcp        0      0 *:ssh      *:*           LISTEN      564/sshd
tcp        0      0 *:login    *:*           LISTEN      491/
tcp        0      0 *:shell    *:*           LISTEN      491/
tcp        0      0 *:telnet   *:*           LISTEN      491/
tcp        0      0 *:auth     *:*           LISTEN      431/identd
tcp        0      0 *:1024     *:*           LISTEN      362/
tcp        0      0 *:sunrpc   *:*           LISTEN      335/
```



```
274 lo33 b116serv:~> netstat -P tcp   (sur solaris)

TCP                                        Send     Recv
Local Address    Remote Address      Swind -Q Rwind -Q  State
-------------- -------------------- ----- -- ----- -- -------
b116serv.32784   b116serv.32780      32768  0 32768  0 ESTABLISHED
b116serv.32780   b116serv.32784      32768  0 32768  0 ESTABLISHED
b116serv.32778   lima.34301           8760  0  8760  0 ESTABLISHED
b116serv.55128   sigma.utc.fr.ftp    32736  0  8760  0 CLOSE_WAIT
b116serv.55148   sigma.utc.fr.ftp    32736  0  8760  0 CLOSE_WAIT
localhost.31418  localhost.37370     32768  0 32768  0 CLOSE_WAIT
b116serv.32778   lima.37423           8760  0  8760  0 ESTABLISHED
b116serv.telnet  satanas.gi.utc.57607 24820  0  8760  0 ESTABLISHED
b116serv.telnet  ncd10.utc.fr.3673     3911  0  9112  0 ESTABLISHED
b116serv.40228   xsme15.utc.6000      4096  0  9216  0 ESTABLISHED
```

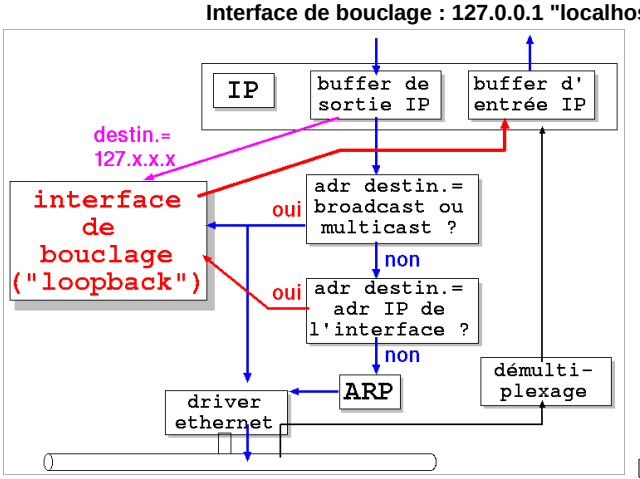

Les commandes **ping** et **traceroute** permettent de tester la connectivité : si ping et traceroute indiquent que la connectivité est acquise et que les applications (telnet, mozilla) ne fonctionnent pas, il faut chercher la raison du côté du DNS (vérifier resolv.conf).

```
1001 vayssade kappa:~> ping vega
PING vega.utc.fr (195.83.156.40): 56 data bytes
64 bytes from 195.83.156.40: icmp_seq=0 ttl=63 time=1 ms
64 bytes from 195.83.156.40: icmp_seq=1 ttl=63 time=0 ms
64 bytes from 195.83.156.40: icmp_seq=2 ttl=63 time=0 ms
^C
----vega.utc.fr PING Statistics----
3 packets transmitted, 3 packets received, 0% packet loss
round-trip (ms)  min/avg/max = 0/0/1 ms
```



```
1006 vayssade kappa:~> traceroute vega
traceroute to vega.utc.fr (195.83.156.40), 30 hops max, 40 byte packet
 1  c6k-msfc-si (195.83.155.1)  1 ms  0 ms  1 ms
 2  vega (195.83.156.40)  1 ms  1 ms  0 ms

[root@localhost /root]# traceroute vega
traceroute to vega (195.83.156.40), 30 hops max, 38 byte packets
 1  brx (192.168.130.1)  3.515 ms  3.154 ms  2.997 ms
 2  brxutc (195.83.155.35)  45.499 ms  48.789 ms  46.050 ms
 3  195.83.155.1 (195.83.155.1)  51.317 ms  47.926 ms  50.609 ms
 4  vega (195.83.156.40)  49.084 ms  49.604 ms  49.719 ms
```

Interface de bouclage : 127.0.0.1 "localhost"



Ch06 - Bases d'un internet. Exemples de questions

➤ Quel est l'objectif initial de la création des réseaux IP ?
Réponse : diapos 2 et 3.

➤ Quel est la raison du choix de l'interconnexion de réseaux pour faire un internet ?
Réponse : diapo 5.

➤ Quel sont les éléments clés constitutifs d'un internet ?
Réponse : diapo 7.

➤ Définition d'un internet.
Réponse : diapo 10.

➤ Dessiner la stratification en couches de TCP/IP.
Réponse : diapos 17 et 18.

➤ Montrer sur un schéma sur quelle "couche" de TCP/IP intervient un routeur IP.
Réponse : diapo 21.

➤ Pourquoi dit-on que IP est un protocole de "saut en saut" ? Que dit-on alors de TCP et UDP ?
Réponse : diapo 22.

➤ Qu'est-ce qu'une "classe d'adresses" IP ? Combien y en a-t-il ?
Réponse : diapo 24. Conseil : retenez par coeur 128, 192 et 224.

➤ Qu'est-ce que le DNS et à quoi sert-il ?
Réponse : diapo 25.

➤ Quelle est la structure du DNS ?
Réponse : diapo 25.

➤ Dans le cadre du DNS, qu'est-ce qu'une "SOA" et quel est son rôle ?
Réponse : diapo 27.

➤ Sur une machine unix, ou indique-t-on le serveur DNS à interroger, quel est le programme qui fait ça et sous quel "démon" apparait-il ?
Réponse : diapos 28 et 29 : /etc/resilv.conf, BIND, named.

➤ Comment fait le DNS pour retrouver un nom à partir d'une adresse ?
Réponse : diapo 32 : utilise le pseudo-domaine IN-ADDR.ARPA

➤ Montrer la structure d'encapsulation des données dans la pile TCP/IP.
Réponse : diapo 34.

➤ Qu'est-ce que le MTU ? Que se passe-t-il quand il est petit ?

Réponse : diapo 35.

➤ Qu'est-ce que les ports bien connus ?
Réponse : diapo 36.

➤ Montrer sur un schéma le processus de démultiplexage d'un paquet TCP contenu arrivant sur une interface ethernet.
Réponse : diapo 37.

➤ Donner les éléments principaux contenus dans l'en-tête IP.
Réponse : diapo 38 : donner au moins "adr.srce", "adr destin", longueur, TTL, flags et offset fragment.

➤ Donner les éléments principaux d'une en-tête UDP.
Réponse : diapo 40 : ports srce et destin.

➤ Donner les éléments principaux d'une en-tête TCP.

Réponse : diapo 41 : les 4 à donner au minimum : ports srce et destin., numéro de séquence et numéro d'acquittement.

➤ Faites le schéma d'une trame ethernet (mais pas 802.3) contenant un datagramme IP qui contient lui-même 4 octets de données.
Réponse : diapo 44. ! ! Piège : il faut 2 octets de padding.

➤ Dans une trame PPP circulant entre deux modems, pourquoi faut-il "déguiser" les octets ayant la valeur 0x7E s'il y en a dans les données transportées ? Et comment fait-on ?
Réponse : diapo 47.

➤ Qu'est-ce que la compression Van Jacobson ?
Réponse : diapo 48.

➤ Montrer qu'avec un MTU trop gros sur une ligne a trop faible débit, il est impossible de faire en même temps de l'interactif et du transfert

de données "en masse".
Réponse : diapos 49 et 50.

➤ Que fait une machine TCP/IP pour envoyer un paquet IP a une autre machine IP située sur le même réseau IP et sur le même réseau physique ethernet ?
Réponse : diapos 53 et 54.

➤ Une machine A envoie un datagramme IP à une machine B. A et B sont sur des réseaux IP différents et sur deux éthernet différents, reliés par une liaison PPP. Donner les contenus des trames successives qui transportent les données de A à B.
Réponse : diapos 55 à 57.

➤ Quelles sont les deux commandes unix permettant de voir (et modifier) l'état des interfaces réseau ?

Réponse : diapo 58.

➤ Quelles sont les deux commandes unix permettant de tester la connectivité entre deux machines ?
Réponse : diapo 67.

Ch06 - Bases d'un internet.

Fin du chapitre.

Ch07 - Les protocoles UDP et TCP

- **Le protocole UDP**
  - en-tête, pseudo-en-tête, fragmentation
  - serveur UDP
- **Le protocole TCP**
  - en-tête, connexion, segments, fiabilité,
  - flux d'octets, fenêtre, données OOB,
  - gestion d'une connexion
  - audit des serveurs TCP sur une machine
  - TCP : flux interactif et en masse
  - Résumé sur TCP

Le protocole UDP

UDP est un protocole de transfert de messages (appelés datagrammes), sans garantie de fiabilité, ni de remise dans l'ordre d'envoi.

Un datagramme UDP est généré pour chaque opération (sendto) faite par un processus applicatif.

```
Encapsulation :
        |<---------- datagramme IP ------------>|
                    |<---- datagramme UDP ----->|
        +-----------+-------+-------------------+
        |  en-tête  |en-tête|      données      |
        |    IP     |  UDP  |        UDP        |
        +-----------+-------+-------------------+
          20 octets  8 octets
```


L'application doit se préoccuper de la taille des datagrammes qu'elle émet : si cette taille dépasse le MTU du réseau, le datagramme IP fait l'objet d'une fragmentation (vrai pour chaque réseau traversé => MTU du chemin).

En-tête UDP :

```
|<--- 16 bits--->|<--- 16 bits--->|
+----------------+----------------+
| num. port srce | num. port dest |
+----------------+----------------+
| long. UDP      | checksum UDP   |
+----------------+----------------+
|       données UDP               |
|                                 |
+----------------+----------------+
```


Les numéros de ports identifient le processus émetteur et le processus destinataire.

"long UDP" = longueur en-tête + long. data

Somme de contrôle UDP (checksum)

Rem : l'en-tête UDP ne contient pas l'adr. IP des machines srces et dest. Cette info. doit être récupérée dans le datagramme IP englobant.

Pour calculer la somme de contrôle (somme des compléments à 1 de mots de 16 bits), UDP (mais aussi TCP) ajoutent au datagramme un **pseudo-en-tête** de 12 octets.


```
|<--- 16 bits--->|<--- 16 bits--->|   pseudo-en-tête
+----------------+----------------+   12 octets
|   adresse IP source (32 bits)   |
+----------------+----------------+
|   adresse IP destination        |
+-------+--------+----------------+
|       | proto. | long. UDP      |
+-------+--------+----------------+
| num. port srce | num. port dest |   datagramme IP
+----------------+----------------+
| long. UDP      | checksum UDP   |
+----------------+----------------+
|       données UDP               |
|                                 |
+       +--------+----------------+
|       |bourage |
+----------------+
```


Cet en-tête ne fait pas partie du voyage : il est généré au départ et regénéré à l'arrivée et ne sert qu'au calcul de la somme de contrôle.

Le **pseudo en-tête** contient entre autre, l'adr.IP de la machine émettrice et l'adr.IP de la machine destinataire. En vérifiant la somme de contrôle, on vérifie ainsi que le datagramme est parvenu sur la bonne machine.

Si une erreur est détectée dans le calcul de la somme de contrôle, le datagramme est détruit en silence, sans émission de message d'erreur.

Fragmentation IP

Si un datagramme a une longueur plus grande que le MTU de l'un des réseaux traversés, la couche IP fragmente le datagramme avant de l'émettre.

Quand un datagramme est fragmenté, il n'est réassemblé **que** sur la machine de destination finale. C'est la couche IP qui réassemble. Ainsi la fragmentation reste invisible à UDP ou à TCP.

L'en-tête contient un champ **identification** qui contient une valeur unique pour chaque datagramme IP envoyé par l'émetteur. Ce champ est **copié** dans chaque fragment.

Les fragments, sauf le dernier, ont dans le champ **flags** le bit **more fragments** mis à 1.

La façon dont les fragments sont construits fait que si **un** fragment est perdu, tout le datagramme doit être réémis.


```
Fragmentation :
        |<---------- datagramme IP ------------>|
                    |<---- datagramme UDP ----->|
        +-----------+-------+-------------------+
        |  en-tête  |en-tête|      données      |
        |    IP     |  UDP  |        UDP        |
        +-----------+-------+-------------------+
          20 octets  8 octets       /    \
                                 /--/      \----\
  20 octets  8 octets          /      20 octets \
+-----------+-------+----------+    +-----------+-----------+
|  en-tête  |en-tête| données  |    |  en-tête  | données   |
|    IP     |  UDP  |   UDP    |    |    IP     | UDP suite |
+-----------+-------+----------+    +-----------+-----------+
```

Conception d'un serveur UDP

La conception et l'implémentation des clients est généralement plus facile que celle des serveurs : typiquement, un client démarre, envoie un datagramme à un serveur, lit la réponse et se termine.

Un serveur, lui, démarre, se met en sommeil en attendant une requête d'un client, répondent à **ce** client, puis se remettent en sommeil.

Le serveur doit demander au système d'exploitation l'origine du message (adr.IP source et numéro de port source).

Souvent les serveurs UDP sont des serveurs **itératifs**. Il n'y a donc qu'un seul process serveur pour supporter les requêtes de tous les clients qui arrivent, toutes sur le même port UDP (celui du serveur).

Pendant le traitement d'une requête, s'il en arrive une autre, elle doit être **mise en file d'attente**.

Il y a une limite à la longueur de cette file d'attente.

➤ Si la file déborde, les datagrammes arrivants sont rejetés par la couche UDP de la machine du serveur. Les clients ne sont pas prévenus de ce rejet. Le serveur n'est pas prévenu que sa file d'attente est surchargée.

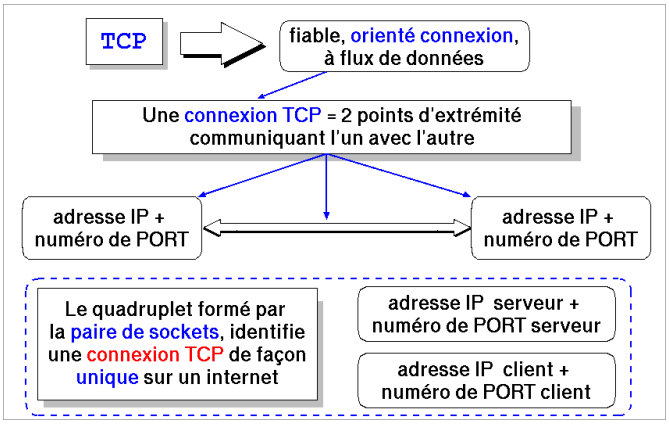
Le protocole TCP

Services de TCP et en-tête TCP

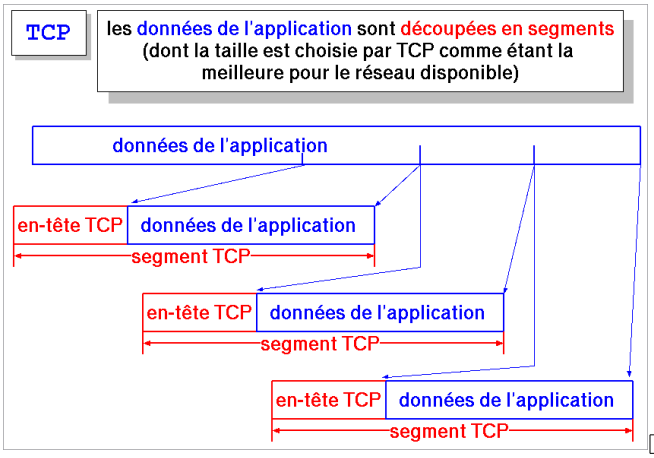
TCP est un protocole de transfert de flus d'octets, **avec** garantie de fiabilité, et de remise des octets dans l'ordre d'envoi, sans duplication.

```
Encapsulation :
        |<---------- datagramme IP -------------->|
                   |<-------- segment TCP ------>|
        +-----------+---------+-------------------+
        |  en-tête  | en-tête |      données      |
        |    IP     |   TCP   |       TCP         |
        +-----------+---------+-------------------+
          20 octets  20 octets
```

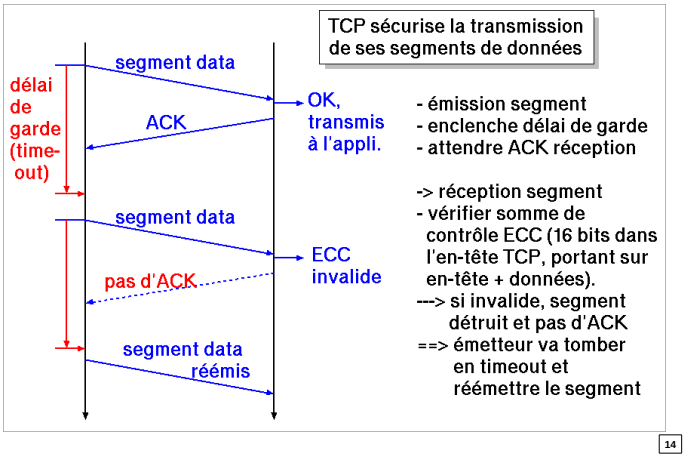

Connexion TCP



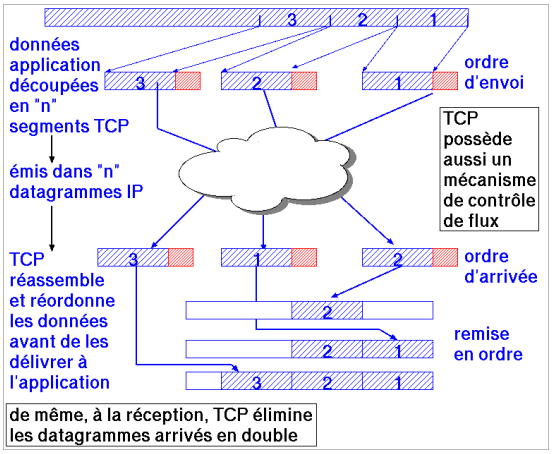
Fonctionnement de TCP



Fonctionnement de TCP



Fonctionnement de TCP : flux d'octets



TCP : fonctionnement, fiabilité

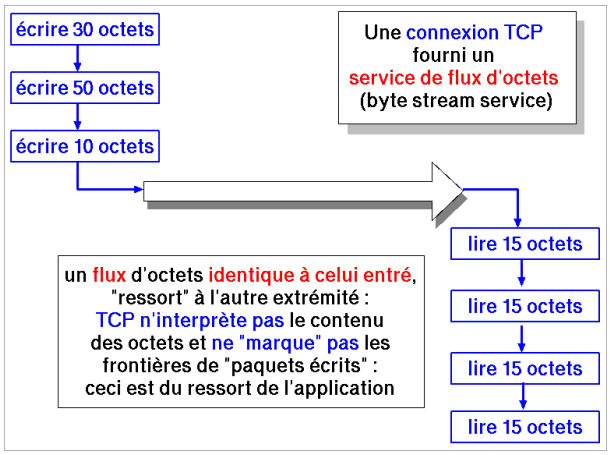
TCP fournit un **service orienté connexion** et apporte la fiabilité par les actions suivantes :

- les données application sont fragmentées en **segments** dont la longueur est celle jugée la meilleure par TCP ;
- à chaque segment émis, TCP maintient un délai de garde ("timer") en attendant l'arrivée d'un acquittement de la part du destinataire du segment ;
- à la réception de données, un acquittement est émis ;
- à la réception TCP réordonne les données si nécessaire (elles ont voyagé dans des datagrammes IP) ;
- TCP maintient dans chaque segment une somme de contrôle d'extrémité à extrémité ;

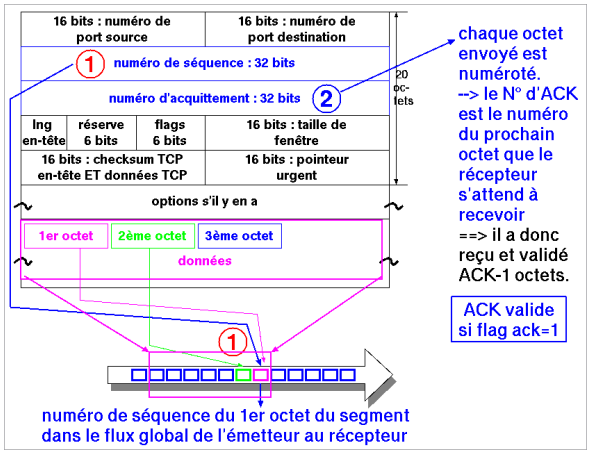
- TCP rejette les données dupliquées ;
- TCP fournit un contrôle de flux pour ne pas surcharger un destinataire.

TCP fournit un **flux d'octets** (byte stream service) indifférenciés sans marqueurs de fin d'enregistrement (c'est éventuellement à l'application de les mettre).

Fonctionnement de TCP : flux d'octets



Fonctionnement de TCP : flux d'octets



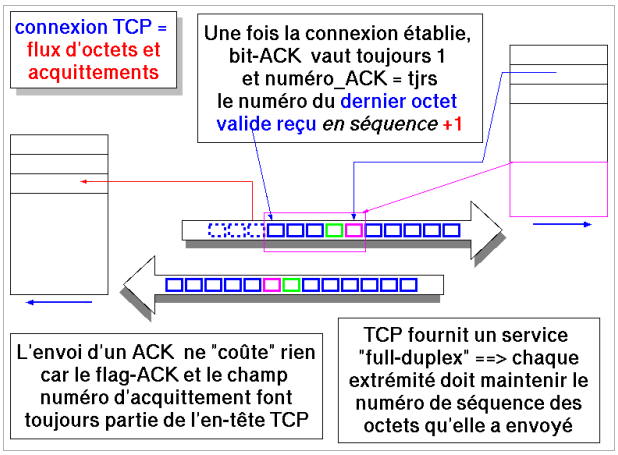
Le numéro de séquence dur 32 bits est un entier non signé. Il retource à 0 après $2^{32} - 1$.

C'est un **numéro d'octet** et non de mesage puisque les frontières des paquets écrits ne sont pas préservés : il n'y a pas de notion de message dans TCP (mais l'application peut l'ajouter avec ses propres conventions).

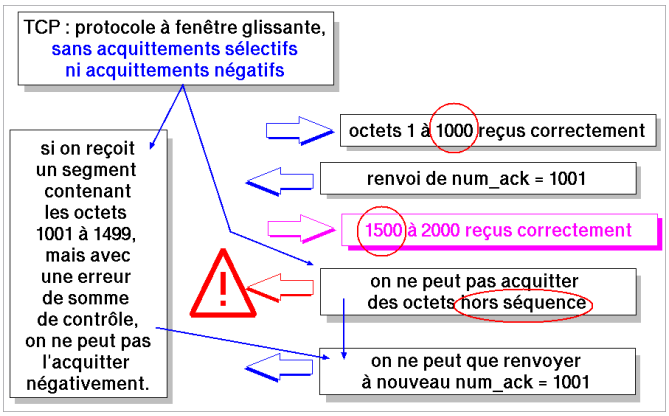
Le numéro de séquence des octets début à ISN+1.

ISN (Initial Sequence Number) est la valeur du champ numéro de séquence dans le segment d'établissement de la connexion (segment avec SYN=1 et pas d'ack envoyé par le *demandeur* d'ouverture de connexion).

Fonctionnement de TCP : flux d'octets



Fonctionnement de TCP : flux d'octets



Choix du NSI : (ISN Inital Sequence Number) utilisés au démarrage de chaque connexion.

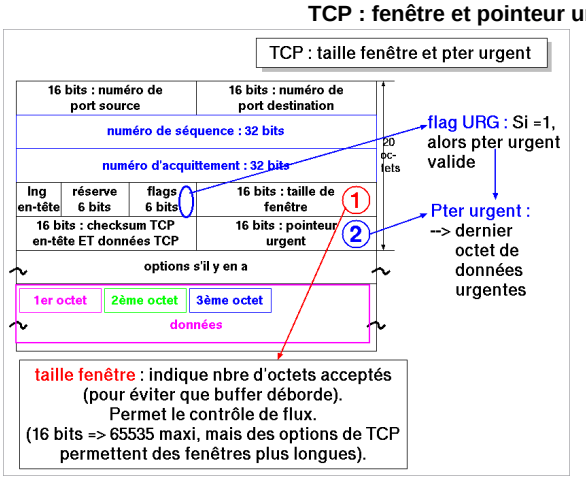
Quand une extrémité émet un segment "SYN" pour établir une connexion, elle choisit un ISN. La RFC 793 spécifie que cet ISN doit être la valeur d'un compteur de 32 bits, maintenu par chaque machine. Ce compteur est initialisé à 1 au démarrage de la machine puis incrémenté de 1 toutes les 4 microsecondes.

En pratique, la plupart des implémentations suivent celle de BSD4.4 qui incrémente toutes les 8 microsec., soit un incrément de **64000 par demi-seconde** soit un passage par zéro toutes les 9,5 heures.

Le but de ce mécanisme est d'éviter que des paquets ayant pris du **retard** dans le réseau soient délivrés plus tard et interprétés par erreur comme faisant partie d'une nouvelle connexion qui ne serait pas celle qui les a créés.

Ceci serait possible si l'ISN était pris à 1 à chaque connexion.

TCP : fenêtre et pointeur urgent



TCP messages urgents (bit URG)

Le pter urgent est un entier positif qui, ajouté au numéro de séquence de l'en-tête TCP, désigne le **dernier octet** de **données urgentes**.

*** Il n'y a aucun moyen de désigner le premier octet de données urgentes : c'est à l'application d'avoir son propre protocole pour reconnaitre le type de donnée urgente en fonction de la valeur de ce dernier octet.

TCP informe le processus récepteur (celui qui a fait le read sur le socket) qu'un pointeur a été reçu.

L'application doit pouvoir retrouver les données urgentes dans le flux.

Si un nouveau pointeur urgent arrive alors que les données du précédent n'ont pas encore été traitées, le pointeur est avancé et sa valeur précédente perdue. ==> si les données urgentes sont importantes pour l'appli., celle-ci doit les "marquer" spécialement.

Par exemple, telnet qui utilise les données urgentes, **marque** tous les octets de commande en les préfixant par l'octet 255.

Envoi de données urgentes : option MSG_OOB de send(2)

envoi par : **send (descript_sock, buf, 1, MSG_OOB) ;** (OOB = Out_Of_Band).

Attention : OOB = Out_Of_Band, mais cette appellation "hors bande" de l'API des sockets est incorrecte : de vraies données hors bande devraient passer par un canal logique séparé (le plus simple serait

une seconde connexion TCP).

Si le message envoyé contient plusieurs caractères, seul le dernier est désigné par le pointeur urgent.

Le récepteur doit demander la notification par :

ioctl (descript_sock, SIOCSPGRP, &pid) ;

il recevra alors un signal SIGURG.

On peut, soit demander à placer les données urgentes dans le flux, soit demander à les traiter à part, dans ce cas il faut faire une lecture spéciale.


```
man 7 tcp
man 7 socket
 SO_OOBINLINE
  If this option is enabled, out-of-band data is directly
  placed into the receive data stream. Otherwise  out-of-
  band data is only passed  when the MSG_OOB  flag is set
  during receiving.

man 2 recv
 MSG_OOB
  This flag requests receipt of out-of-band data that
  would not be received in the normal data stream.
```


Si les données OOB sont dans le flux,le caractère urgent est inséré dans le buffer de lecture du récepteur mais il n'est pas accessible (par défaut) par la lecture normale :

read (descript_sock, buf, 1024) ;

va **butter** sur le caractère urgent (s'il y a 500 car. avant le car.urg, ce read lit 500 car ;)

Il faut lire le car.urg par :

read (descript_sock, buf, 1, MSG_OOB) ;

On peut **tester** si la position de lecture est parvenue au car. urgent par (voir man 2 ioctl et man 2 ioctl_list) :

ioctl (descript_sock, SIOCATMARK, &at) ;

L'entier "at" est mis à une valeur positive si des données OOB sont

atteintes. ("The GNU C Library Reference Manual").

Nota : l'appel système **select()** permet aussi de surveiller l'arrivée de données OOB.

Attention : il y a eu des interprétation différentes de la première RFC sur le sujet. Normalement, cela a été précisé dans une autre RFC (Host requirements) qui indique que le pointeur doit désigner le **dernier octet urgent**, MAIS, certaines implémentations héritées de BSD font pointer le pointeur sur l'octet **QUI SUIT** le dernier octet urgent.

Une implémentation correcte pourrait ne pas communiquer correctement avec ces implantations erronées mais répandues.

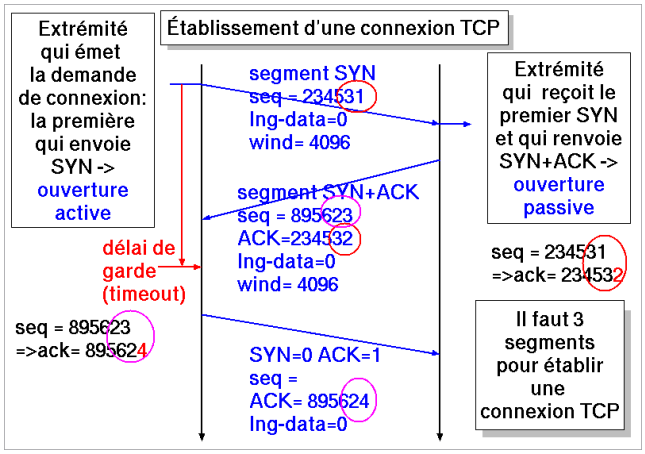
Socket et signaux

1. **SIGIO** données à lire sur un descripteur
2. **SIGURG** car. urgent arrivé
3. **SIGPIPE** on essaie d'écrire sur un descripteur fermé en écriture ou n'ayant plus de lecteur

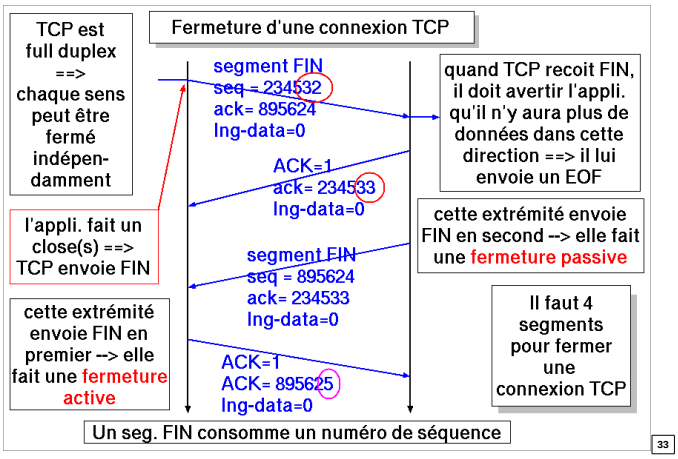
Les primitives d'E/S bloquantes sont interrompues par l'arrivée d'un signal. Après traitement du signal, elles ressortent de l'appel système avec le **code statut -1 et errno=EINTR**.

Ceci concerne : read, write, accept, open, fcntl, wait, connect, semop, msgsnd, msgrcv, sendto, recvfrom, send, recv, ...

Établissement d'une connexion TCP

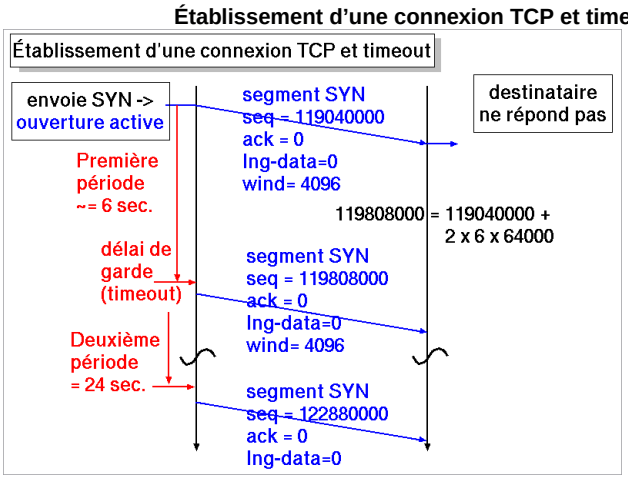


Fermeture d'une connexion TCP



33

Établissement d'une connexion TCP et timeout



34

Le premier numéro de séquence du dessin précédent est 119040000 : soit : (15x60+30) x 2 x 64000 = 119040000 (+64000 ttes les 1/2 sec.) ==> la machine de l'émetteur (le client du socket) a démarré il y a 15'30".

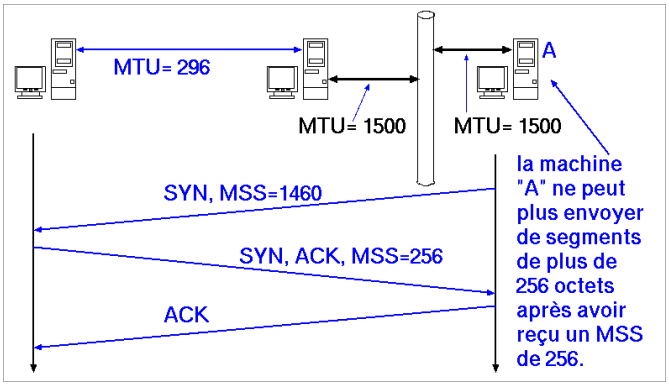
La première période, en fait, expire entre 5,5 et 6 sec. car les implantations BSD de TCP utilisent un timer de 500 millisec.

TCP compte donc les 6 sec. comme se terminant à la 12ème expiration du timer après la pose du timeout.

Comme la pose du timeout a lieu "quelque part" au milieu d'un intervalle, la précision du premier timeout est de 0,5s.

Le 2ème timeout de 24 sec. étant relancé au moment où le timer vient d'expirer, est lui, très proche de 24 sec.

TCP : MSS et MTU



36

TCP : MSS et MTU

MTU = Maximum Transfert Unit ; MSS = Masimum Segment Size

Le MSS est le plus grand morceau de données que TCP enverra vers l'autre extrémité.

Un datagramme = MSS + 20 en-tête TCP + 20 en-tête IP Si pas de MSS annonçée dans les segments SYN => val. par défaut qui est 536 soit un datagramme de 576 octets.

Le MSS permet a une machine d'éviter la fragmentation sur les connexions qui empruntent des liaisons avec un petit MTU.

Lors de l'établissement d'une connexion (envoi du segment SYN), TCP envoie une valeur de MSS <= au MTU de l'interface sortante.

Sur un ethernet : MTU= 1500 => MSS= 1460. C'est ce que font certaines implémentations (ex ; SunOS 4, Solaris 2.2).

Certaines implémentations annoncent MSS=1024 pour que les données soient un multiple de 512 (ex. BSD, SVR4).

Une machine ne peut choisir un MSS inférieur à la valeur par défaut que si son interface est connectée à une liaison ayant un MTU de moins de 576.

Si 2 machines sont connectées sur un ethernet et que leur connexion *traverse* une liaison dont le MTU est de 296, la fragmentation se produira, **sauf** à utiliser le mécanisme de découverte de MTU de chemin : on envoie un paquet avec le bit DF=1 (Don't Fragment). Quand ce paquet arrive "à l'entrée" d'une liaison dont le MTU est inférieur au paquet, il revient à l'envoyeur un paquet d'erreur ICMP avec le statut "can't fragment".

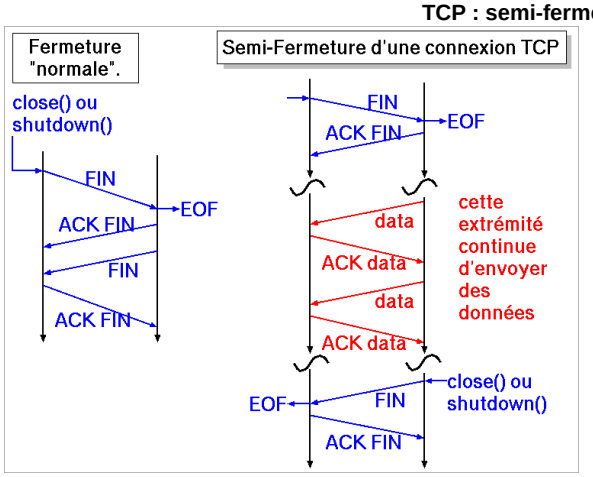
Semi-fermeture de TCP

TCP permet à une extrémité de fermer sa sortie (envoyer le segment FIN) tout en continuant à recevoir des données sur son entrées (jusqu'à l'arrivée du segment FIN).

L'API des sockets BSD supporte cette fonction par l"appel système **shutdown** avec **1** en deuxième argument (alors que l'appel de **close** ferme les deux sens de la connexion).

Cette possibilité est appelée **semi-fermeture** de TCP.

TCP : semi-fermeture



40

À quoi sert la Semi-fermeture de TCP ?

Exemple d'une commande **rsh** sur unix :

sun> **rsh vega sort < fichier_data**

la commande sort est exécutée sur vega avec comme données d'entrée le contenu du fichier "fichier_data"**situé sur sun**.

rsh crée une connexion TCP entre lui-même et le programme exécuté sur l'autre machine :

```
                        machine sun     machine vega
fichier_                  +-----+          +------+
data --> stdin de rsh --> |--   | connexion |      |
                          |  \->|---------->| sort |
                          |  /<-|<----------|      |
        stdout de rsh <-- |<-   |   TCP     |      |
                          +-----+          +------+
```


ensuite rsh copie l'entrée standard sur la connexion, et la sortie de la connexion sur sa sortie standard.

Le programme **sort** ne peut générer aucune sortie tant qu'il n'a pas reçu **toutes** les données à trier ! Comment sait-il qu'il a reçu toutes les données ? Seulement quand il reçoit une fin de fichier (eof) !

Donc rsh doit utiliser la semi-fermeture pour faire parvenir cet **eof** à sort qui peut alors commencer à trier et ensuite envoyer le résultat sur sa sortie standard qui peut encore accepter des données car elle n'est pas fermée (semi-fermeture).

Sans la semi-fermeture il aurait fallut utiliser une autre technique pour faire parvenir l'indication de fin de fichier .

État d'attente 2MSL

Après la fermeture de la connexion, TCP pase dans l'état **TIME_WAIT** aussi appelé état d'attente 2MSL (2MSL timeout).

MSL = Maximum Segment Lifetime : durée de vie maximum d'un segment

Les segments TCP sont transmis dans des datagrammes IP qui possèdent un champ TTL (Time To Live) après lequel ils sont silencieusement détruits.

Ceci pour éviter que des datagrammes tournent à l'infini dans le réseau suite à des problèmes de routage.

La RFC 793 spécifie MSL à 2 minutes mais les implémentations les plus courantes sont de 30", 1 minute ou 2 minutes selon les cas.

Le but de cette attente de 2 x MSL est de permettre à TCP de renvoyer l'ACK de FIN au cas ou celui-ci se serait perdu et que l'autre extrémité ne l'ayant pas reçu et étant tombée en timeout ait renvoyé son segment FIN.

******* un effet de bord de cette attente est que la paire de socket (IP serv, PORT serv, IP client, PORT cli) ne peut pas être réutilisée immédiatement.

C'est l'extrémité qui fait la fermeture active qui passe dans l'état TIME_WAIT. Le plus souvent c'est le client. Si on relance aussitôt le client, il ne peut pas réutiliser le même port. Ce n'est pas grave puisque le client utilise le plus souvent un port éphémère attribué dynamiquement par le système.

Par contre si on arrête brutalement le serveur, en provoquant donc une fermeture active, et si on relance aussitôt le serveur, on obtient l'erreur "**address already in use**".

Il n'y a pas d'autre solution alors que d'attendre la fin du TIME_WAIT.

"Audit" des serveurs TCP sur une machine

La plupart des serveurs TCP sont **"concurrents"** :
nouvelle connexion d'un client ==> création d'un process qui gère le nouveau client sur un nouveau numéro de socket :

nsd = accept(soc, ...).

Comment sont gérés les numéros de ports quand un serveur accepte une nouvelle connexion ?

netstat (sur Solaris)

- **-a** affiche tous les points d'extrémité
- **-n** affiche les adr.IP et les N° de ports en décimal
- **-f inet** affiche seulement les points TCP et UDP.

netstat (sur Linux)

- **-a** affiche tous les points d'extrémité
- **-n** affiche les adr.IP et les N° de ports en décimal
- **-t** affiche les points d'extrémité TCP
- **-u** affiche les points d'extrémité UDP
- **-p** ajoute le PID/programme à l'extrémité du socket
- **-Ainet** affiche tous les points "internet".

Exemple (sur Linux)

```
netstat -a -Ainet
Active Internet connections (servers and established)
Proto R-Q S Local Address       Foreign Address     State
tcp    0  0 *:nfs               *:*                 LISTEN
tcp    0  0 *:rsync             *:*                 LISTEN
tcp    0  0 *:telnet            *:*                 LISTEN
tcp    0  0 simnav12.utc:34017  sunserv.utc:telnet  ESTABLISHED
tcp    0  0 simnav12.utc:33712  tuxa.sme.utc:telnet ESTABLISHED
tcp    0  0 simnav12.utc:telnet tuxa.sme.utc:38998  ESTABLISHED
udp    0  0 *:nfs               *:*
udp    0  0 *:netbios-ns        *:*
udp    0  0 *:sunrpc            *:*
```

TCP : Fonctionnement d'un serveur concourant

1 Sur la machine locale (nav12), un serveur telnet a ouvert un socket de connexion et est en attente. Il accepte les connexions depuis n'importe quelle interface.

```
nav12> netstat -a -Ainet
Pro R S  Local Address     Foreign Address   State

tcp 0 0  *:telnet          *:*               LISTEN
```

2 on fait un telnet depuis nav12 (local) vers tuxa (distant)

3 on fait un telnet depuis tuxa (distant) vers nav12 (local)

```
nav12> netstat -a -Ainet
Pro R S  Local Address     Foreign Address   State

tcp 0 0  *:telnet          *:*               LISTEN

tcp 0 0  nav12.utc:33712   tuxa.utc:telnet   ESTABLISHED

tcp 0 0  nav12.utc:telnet  tuxa.utc:38998    ESTABLISHED
```


"Audit" des serveurs TCP sur une machine

Comment sont gérés les numéros de ports quand un serveur accepte une nouvelle connexion ?

Considérons les deux machines "nav12" et "tuxa". Elles ont toutes les deux un process **serveur telnet** qui écoute sur le port telnet (23).

Il faut distinguer deux cas :

- **process sortant** : Pour sortir, comme il y déjà un process (le serveur telnet) qui utilise le port 23, la commande "telnet" qui sort de la machine locale, ne peut utiliser ce port : le process qui exécute le telnet sortant utilise un **port alloué dynamiquement** par le système (dit port éphémère). Sur une même machine **un seul process** à la fois peut utiliser un port.

- **process entrant** : À l'entrée la connexion telnet arrive sur le port du serveur. Le problème est de pouvoir **démultiplexer correctement** les segments TCP qui arrivent. Peut-on le faire si on laisse toutes les connexion telnet entrantes associées au même port (le port local telnet) ? **Oui**, car toutes les connexions entrantes sont **distinguables** par leurs paires **{host_client, port_client}** qui sont, par construction, toutes différentes.

C'est donc bien le **quadruplet {host-srce, port-srce, host-destin, port-destin}** qui identifie de façonunique chaque connexion.

On peut le vérifier sur le schéma suivant en regardant**sur chaque machine**, comment sont enregistrées **deux** connexions entre "nav12" et "tuxa".

Il y a alors plusieurs points d'extrémité actifs sur le port 23 du serveur :

- le point d'extrémité LISTEN reçoit les segments SYN (demandes de connexion) et ne reçoit aucun segment de données,
- les points d'extrémité ESTABLISHED recoivent les segments de données. Ils ne peuvent recevoir de segments SYN.

TCP : serveur concourant

```
nav12> netstat -a -Ainet | grep tuxa
Pro R S  Local Address     Foreign Address   State

tcp 0 0  *:telnet          *:*               LISTEN
tcp 0 0 nav12.utc:telnet tuxa.utc:59980      ESTABLISHED
tcp 0 0 nav12.utc:telnet tuxa.utc:38998      ESTABLISHED
tcp 0 0 nav12.utc:34019  tuxa.utc:telnet     ESTABLISHED
tcp 0 0 nav12.utc:34018  tuxa.utc:telnet     ESTABLISHED
```

1 sessions de tuxa vers nav12

```
tuxa> netstat -a -Ainet | grep nav12
Pro R S  Local Address     Foreign Address   State

tcp 0 0  *:telnet          *:*               LISTEN
tcp 0 0 tuxa.utc:38998   nav12.utc:telnet    ESTABLISHED
tcp 0 0 tuxa.utc:59980   nav12.utc:telnet    ESTABLISHED
tcp 0 0 tuxa.utc:telnet  nav12.utc:34018     ESTABLISHED
tcp 0 0 tuxa.utc:telnet  nav12.utc:34019     ESTABLISHED
tcp 0 0 tuxa.utc:telnet  nav12.utc:34020     ESTABLISHED
```

2 sessions de nav12 vers tuxa

Session supplémentaire vers tuxa pour aller y faire le "netstat ..."

Les transferts de données avec TCP

Des mesures statistiques sur des réseaux en fonctionnement ont permis de constater que :

- ≈**50%** des segments TCP contenaient des données "**en masse**",
- ≈**50%** des segments TCP contenaient des données "**interactives**".

Mais, si on compte les octets :

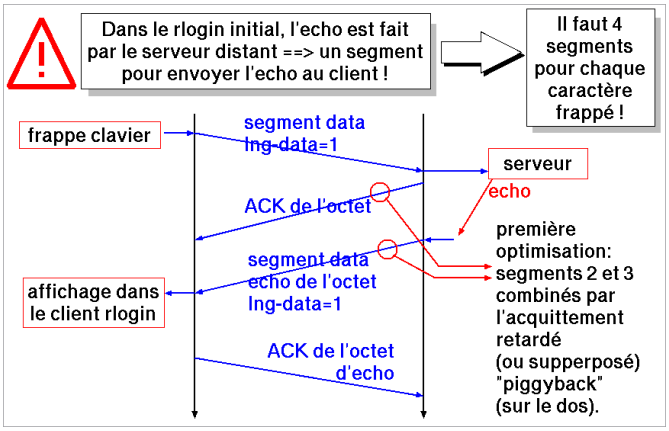
- ≈**90%** des octets sont transportés par des segments TCP contenant des données "**en masse**",
- ≈**10%** des octets sont transportés par des segments TCP contenant des données "**interactives**",

Par exemple, 90% des segments telnet contenaient moins de 10 octets de données.

On a donc introduit des modifications à TCP pour améliorer deux types de communications :

- **les flux de données interactifs** : telnet, rlogin ... Pour cela a été introduit **l'algorithme de Nagle (RFC 896)**,
- **les flux de données en masse** : ftp. Fenêtre glissante. L'algorithme de Jacobson utilise la valeur de la fenêtre pour ralentir ou accélérer le transfert.

TCP : flux interactif (telnet et rlogin)



Algorithme de Nagle (RFC 896)

➤ dans rlogin, 1 octet à la fois du client vers le serveur. ===> Très nombreux "tinygrammes" génants surtout sur les réseaux longue distance. De plus mauvaise utilisation : 1 octet utile plus 20 d'en-tête TCP + 20 d'en-tête IP.

Algorithme de Nagle : une connexion TCP ne peut avoir qu'un petit segment qui n'a pas encore été acquitté. => pas de petit segment envoyé tant que l'ACK du petit segment précedent n'est pas arrivé. Au lieu de cela, TCP collecte les données provenant de l'aplication pour les envoyer en un seul segment.

L'algo. est élégant car **auto-régulé** : si les ACK reviennent vite, les données sont envoyées vite par petits paquets. Si les ACK tardent

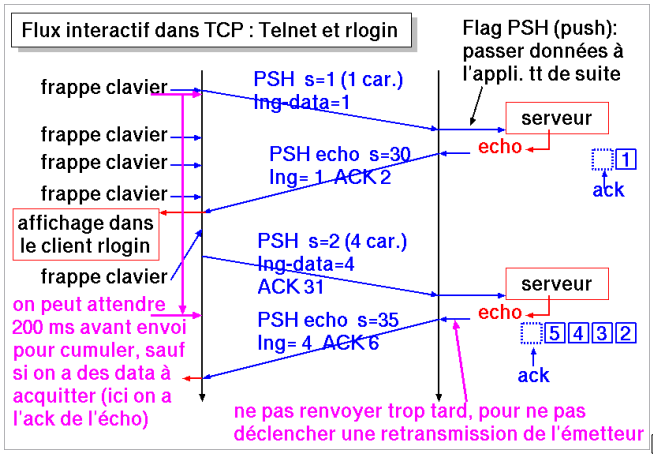
(congestion ou réseau grande distance), les données sont envoyées par paquets plus gros (meilleure efficacité).

Sur un ethernet, avec rlogin le temps moyen de retour des "echo+ACK" est de moins de 20 millisec. ==> il faudrait frapper au clavier à plus de 50 caractères par sec. pour déclencher l'algo. de Nagle.

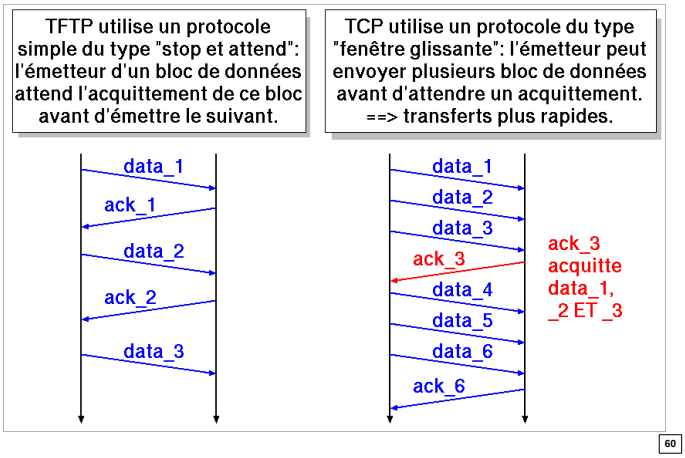
Sur un WAN par contre, le temps de retour est bien plus long.

Nota : l'option **TCP_NODELAY** permet de dévalider l'algorithme de Nagle dans certaines sessions interactives (ex ; X11). Par exemple ceci peut perturber la gestion de touches de fonction pour lesquelles le serveur renvoi en echo une séquence "escape".

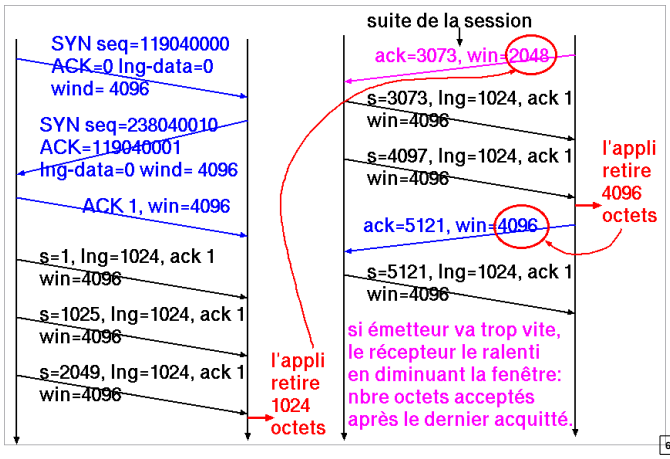
TCP : fluxinteractif



TCP : fluxde données "en masse"



TCP : fluxde données "en masse"



TCP : orienté connexion - résumé des propriétés

=> échange message pour établir la connexion (handshaking)

assure la fiabilté par :

- **données** applicatives **découpées en fragments** (segments TCP)
- taille des segments choisie par TCP
- **à chaque segment** émit, déclenche un **délai de garde** (time-out) et attends acquittement (ACK)
- si **pas d'ACK** dans le délai => **retransmission**
- émission d'un ACK, soit à chq réception, soit pour plusieurs réception (ack groupés),
- maintient une **somme de contrôle** couvrant l'en-tête et les données : c'est une somme "**de bout en bout**" ou "d'extrémité à extrémité" ; si la somme est erronée, l'ACK n'est pas émis et le segment sera émis après time-out

TCP **utilise les services de la couche IP** en lui **ajoutant** la fiabilité :

- les segments TCP voyagent dans des datagrammes IP, ils peuvent donc arriver dans le désordre ; TCP est chargé de les **remettre dans l'ordre** avant de les passer à l'application,
- TCP **élimine** les éventuels segments **dupliqués**,
- TCP fournit un **contrôle de flux** en permettant à l'émetteur d'envoyer autant de données que ses buffers de réception peuvent en contenir mais pas plus.

L'ensemble permet de présenter aux deux applications d'extrémité un service d'échange d'un **fluxd'octets** bi-directionnel.

Les "frontières" des writes sont perdues ==> l'interprétation du flux est de la responsabilité des applications.

Ch07 - Les protocoles UDP et TCP. Exemples de questions

➤ Que fait chaque opération sur un socket UDP ?
Réponse : diapo 2.

➤ Comment la couche UDP s'assure qu'un datagramme UDP est bien arrivé (avant de l'utiliser) ?
Réponse : diapos 4 à 6.

➤ Que fait la couche UDP si un datagramme UDP est mal arrivé ?
Réponse : diapo 6. Elle l'ignore *silencieusement*.

➤ À quoi sert le "pseudo-en-tête" UDP ?
Réponse : diapos 4 à 6.

➤ Dans quel cas y-a-t-il fragmentation d'un datagramme ?
Réponse : diapo 7.

➤ Comment se fait la fragmentation d'un datagramme ?
Réponse : diapos 7 et 8. Attention ! C'est le datagramme **IP** qui est fragmenté, pas le datagramme UDP.

➤ Que se passe-t-il si un serveur UDP itératif est surchargé ?
Réponse : diapos 9 et 10

➤ Qu'est-ce qui défini une "connexion TCP" ?
Réponse : diapo 12.

➤ Comment fonctionne TCP (i.e. comment assure-t-il le transport fiab le des données) ?
Réponse : diapos 13 à 17

➤ Comment TCP gère-t-il la suite d'octets que lui envoie l'application ?
Réponse : diapos 18 à 20

➤ Comment TCP gère-t-il les acquittments des données reçues ?
Réponse : diapos 21 et 22

➤ Qu'est-ce que l'ISN ? Comment est-il géré ?
Réponse : diapo 23.

➤ Qu'est-ce que les "messages urgents" de TCP ?
Réponse : diapos 25 et 27

➤ Quand les données urgentes de TCP sont "dans le flux", comment l'application destinataire peut-elle les gérer ?
Réponse : diapos 27, 29 et 30.

➤ Comment s'établit une connexion TCP ?
Réponse : diapo 32.

➤ Comment se fait la fermeture complète d'une connexion TCP ?

Réponse : diapo 33.

➤ Comment fonctionne la semi-fermeture d'une connexion TCP ?
Réponse : diapo 40.

➤ Donner un exemple d'utilisation de la semi-fermeture de TCP.
Réponse : diapos 41 et 42

➤ Qu'est-ce que l'état d'attente "2MSL", et à quoi sert-il ?
Réponse : diapos 43 et 44.

➤ Dans quel cas l'état d'attente "2MSL" as-t-il un effet de bord désagréable ?
Réponse : diapo 45.

➤ Quelle commande unix (version Linux) permet d'observer les connexions TCP ?

Réponse : diapo 48.

➤ Sur la sortie ci-dessous, identifiez les connexions serveur, entrantes et sortantes (expliquez !).

```
nav12> netstat -a -Ainet | grep tuxa
tcp 0 0 *:telnet          *:*               LISTEN
tcp 0 0 nav12.utc:telnet tuxa.utc:59980   ESTABLISHED
tcp 0 0 nav12.utc:telnet tuxa.utc:38998   ESTABLISHED
tcp 0 0 nav12.utc:34019  tuxa.utc:telnet  ESTABLISHED
tcp 0 0 nav12.utc:34018  tuxa.utc:telnet  ESTABLISHED
```

Réponse : diapos 50 à 53.

➤ Bonus : Pour des raisons de simplification, il y a une petite incohérence dans la sortie donnée à la question précédente. Laquelle ?
Réponse : Quoi ! ? Retournez en SR02 !

➤ Quelles sont les fonctions ajoutées à TCP pour améliorer (au sens de la charge du réseau) les connexions interactives ?
Réponse : diapos 56 à 59.

➤ Montrer que la manipulation de la fenêtre de TCP permet de ralentir ou accélérer le flux émis.
Réponse : diapo 61.

Ch07 - Les protocoles UDP et TCP.

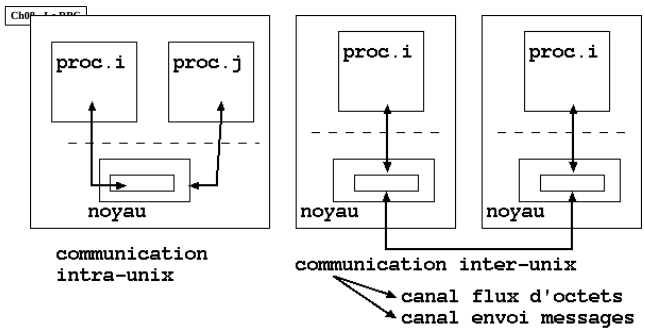
Fin du chapitre.

(diapo vide)

Ch08 - Le RPC

- les contextes de communications entre process
- Le principe du RPC
- Le RPC de SUN (ONC RPC) : notion d'adaptateur
- Mécanisme du "service RPC" : le portmapper
- Premier exemple : programmation sur l'interface de haut niveau
- Deuxième exemple : programmation avec rpcgen et rpcl
- Troisième exemple : reprogrammation avec rpcgen et rpcl du premier exemple
- Autres possibilités des RPCs

Communication de base entre process répartis



On a vu l'outil de base des comunications entre process situés sur des machines distantes : **le socket**, qui crèe un **canal de communication bidirectionnel**.

Communication de base entre process répartis

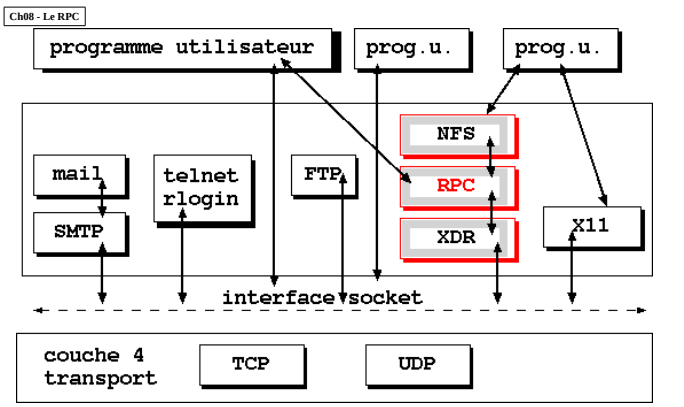
Sur cet outil de base (le socket), on a construit un certain nombre d'applications, par exemple :

- le transfert de fichiers (FTP),
- le courier (mail SMTP),
- la connexion à distance (telnet, rlogin, ssh),
- le service de nommage DNS,
- le graphique à distance (X11, VNC,...),
- ..

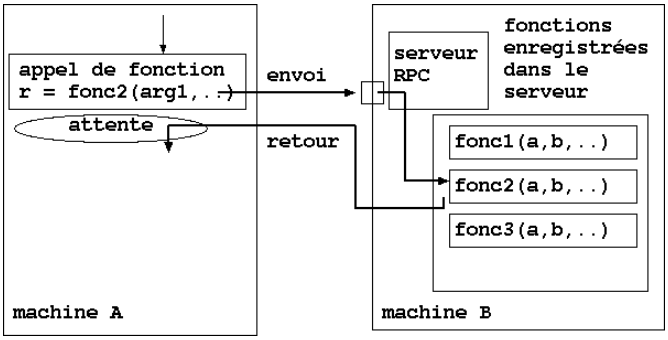
Le principe du RPC

- applications plus sophistiquées ==> besoin pour un "service réseau" de **plus haut niveau**.
- conception de NFS par SUN –> service de **RPC** ("Remote Procedure Call"), appel de procédure à distance.
- mode de communication **s'ajoute** à ceux déja disponibles, "**au-dessus**" des couches TCP et UDP, car il est d'un niveau conceptuel supérieur au simple échange de données.

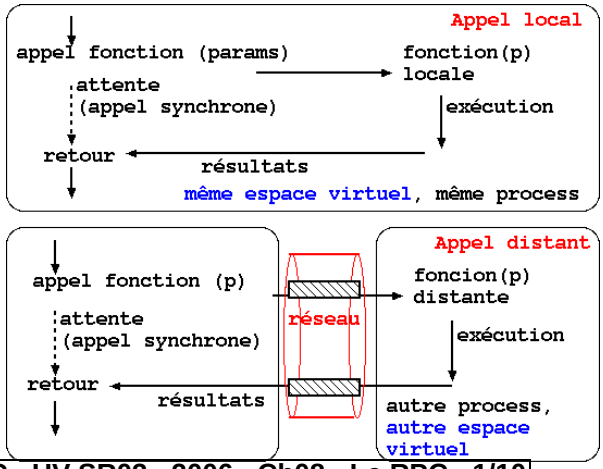
Positionnement du RPC



Le principe du RPC



Appel de procédure local versus appel distant



Fonctionnement du RPC

- Ainsi, le RPC réalise un appel de sous-programme à travers le réseau.
- Le RPC est construit au-dessus des sockets. C'est une interface de plus haut niveau.
- Les arguments sont encapsulés (empaquetés : "argument marshalling"), puis envoyés au process qui "propose" les services des fonctions qui sont enregistrées.
- Le serveur RPC, décode le contenu du paquet et appelle la bonne fonction avec les arguments envoyés par l'appelant.
- La fonction s'exécute et renvoie un ou des résultats. Ces résultats sont eux aussi empaquetés ("result marshalling") et envoyés sur le réseau au process appelant.

Le service RPC est indépendant du protocole sous jacent utilisé pour transporter les messages, car il n'a besoin d'aucune information sur la façon dont le message est transmis.

Si on utilise le RPC au-dessus de UDP, c'est l'application qui doit se préoucper de la fiabilité des échanges et mettre en place ses propres mécanismes de chiens de garde (time-out) et de retransmission.

Ainsi, si l'application, utilisant le RPC au-dessus de UDP, reçoit une réponse après avoir fait une réémission après un time-out, elle peut seulement conclure que la procédure a été excutée **au moins une fois**.

Si l'application, utilisant le RPC au-dessus de TCP, reçoit une réponse, elle peut conclure que la procédure a été excutée **exactement une fois**.

ONC RPC est disponible à la fois sur TCP et sur UDP. Le choix de UDP comme transport, plus léger, ne doit être fait que si l'application a certaines caractéristiques :

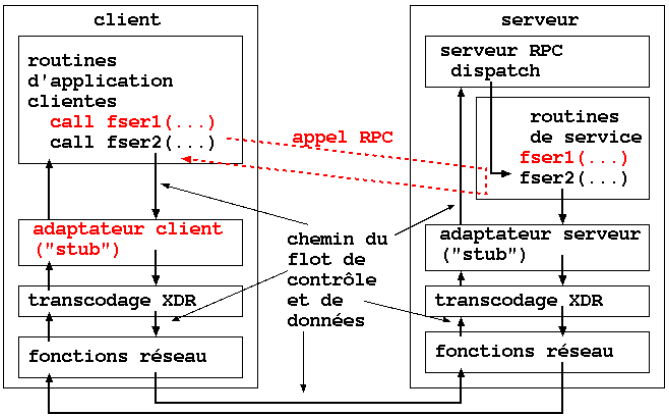
- les procédures sont idempotentes (la procédure peut être exécutée plusieurs fois sans effet de bord) ; par exemple, lire un bloc de données est idempotent, céer un fichier ne l'est pas ;
- la taille des arguments aussi bien que des résultats est inférieure à la taille maximum d'un paquet UDP (8 Koctets) ;
- le serveur doit gérer plusieurs centaines de clients ;

un serveur UDP peut faire cela facilement car il n'a pas à garder d'information d'état pour chacun de ses clients ;
Par contre un serveur TCP qui garde une information d'état pour chaque client connecté va consommer beaucoup de ressources (mémoire, descripteurs, ports socket) ;

Les éléments ci-dessus expliquent, que NFS étant un protocole sans état (par exemple il n'y a pas de fonction "open" dans NFS), puisse sans problème utiliser RPC sur UDP.

Le mécanisme de RPC a été spécifié par SUN et est connu sous le nom de SUN/RPC ou ONC/RPC (défini dans la RFC 1831) (ONC = Open-Network Computing).

Le RPC de SUN (ONC RPC) : notion d'adaptateur



Le RPC de SUN (ONC RPC) : notion d'adaptateur

L'adaptateur "simule" la procédure appelée. En effet, le client et le serveur sont deux programmes différents qui ont tous deux été compilés et linkés. Si le client fait appel à une fonction "fser1", le linker exige de trouver une fonction de ce nom lors de la création de l'exécutable.

Donc, une fonction ayant le nom "fser1" est fournie par le "stub". Mais, au lieu de faire le travail de la vraie fonction "fser1", cette pseudo fonction va servir de mandataire ("proxy") et déléguer le travail à la vraie fonction, située dans le serveur RPC.

La fonction de l'adaptateur va utiliser les fonctions de la bibliothèque XDR pour encapsuler les paramètres et les données d'appel de "fser1", puis fabriquer un paquet qu'elle va transmettre sur le réseau, en utilisant un socket UDP ou un socket TCP.

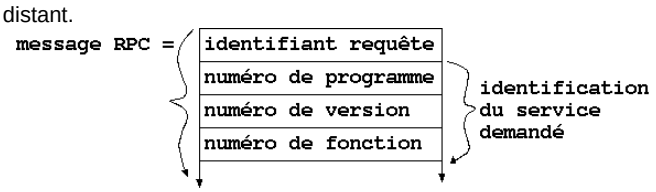
Ce paquet est envoyé au serveur RPC qui, en fonction du contenu du paquet, va envoyer ("dispatch") cet appel à la bonne fonction de service. Pour cela il utilise aussi les fonctions XDR pour "déballer" les données et les passer à la fonction à travers ses arguments.

Celle-ci s'exécute normalement, et fabrique une structure de données contenant les résultats. Cette structure sera traduite par des fonctions XDR avant d'être incluse dans un paquet "réseau" à destination du client.

Le "stub" côté client va réceptionner ce paquet, déballer la réponse avec les fonctions XDR, écrire les différentes parties de la réponse dans les arguments de la fonction appelée.

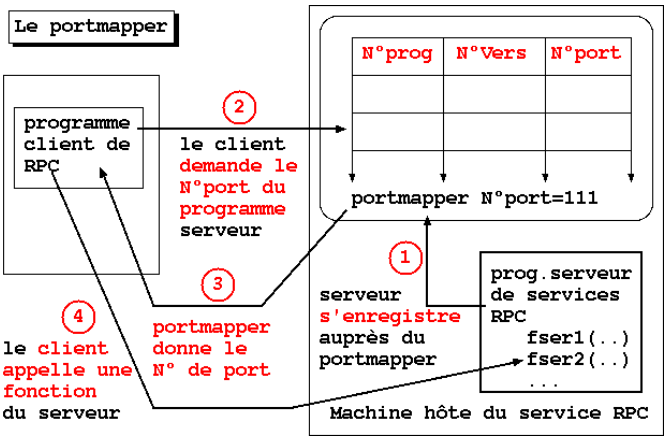
La fonction appelante du client "croit" faire un appel de procédure ordinaire. La complexité est masquée par le "stub".

Les stubs client et serveur utilisent un protocole définissant le format des messages qu'ils échangent entre eux pour réaliser un appel distant.



Mécanisme du "service RPC" : le portmapper

- Un "service RPC" est une liste de fonctions appelables à distance, avec une définition des arguments d'appel et de retour.
- Un "service RPC" est repéré par un **numéro de programme** plus un **numéro de version**.
- Un service RPC est **localisé** par un appel à un programme particulier, le **portmapper** (démon portmap) qui est en écoute sur un **port "bien connu"** (rappel : les numéros de port bien connus sont attribués par l'IANA).



Le port bien connu du portmapper est le **port 111**, aussi bien en UDP qu'en TCP :

```
[root@linux]# rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
...........
1 sr03 sunserv:~> rpcinfo -p
   program vers proto   port  service
    100000    4   tcp    111  rpcbind
    100000    4   udp    111  rpcbind
```

Il existe un certain nombre de programmes systèmes qui utilisent le RPC et qui sont déclarés auprès du portmapper avec des numéros conventionnels. Ces services sont listés dans le fichier "/etc/rpc".


```
2 sr03 sunserv:~> more /etc/rpc
#ident  "@(#)rpc        1.11    95/07/14 SMI"  /* SVr4.0 1.2 */
#       rpc
rpcbind         100000  portmap sunrpc rpcbind
rstatd          100001  rstat rup perfmeter
rusersd         100002  rusers
nfs             100003  nfsprog
.....
```

La fonction système **getrpcbyname** permet de récupérer le numéro d'un service en donnant son nom (défini dans /etc/rpc).

Identification des procédures.

Comme vu sur le schéma, les procédures sont regroupées en **programmes** ou services RPC. Chaque **procédure est identifiée**, parmi celles d'un programme, **par un entier** défini symboliquement.

Un programme est identifié par un entier long de 32 bits. Par exemple NFS a comme identifiant le numéro de programme 100003 (nombre décimal).

Les valeurs possibles sont découpées en plages. Une plage est réservée aux programmes "connus" (par exemple NFS). Les numéros y sont attribués par SUN.

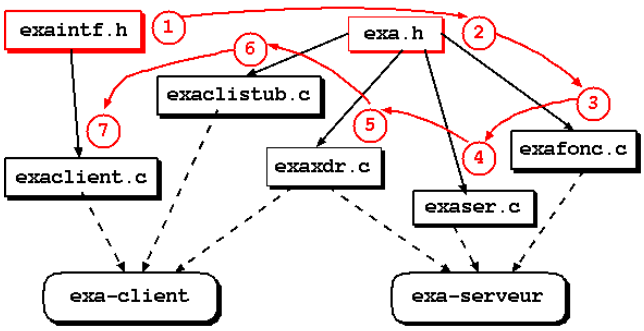
Les utilisateurs peuvent choisir un numéro dans la plage 0x20000000 : 0x3fffffff.

Programmation d'un service RPC

Il y a trois façons d'écrire un service RPC :

- **L'interface "de haut niveau".** Elle est plus facile à programmer que la suivante, moyennant quelques limites : elle impose UDP, et impose aussi le nombre de tentatives et la valeur du time-out en cas de non-réponse.
- **L'interface "de la couche basse".** Elle lève toutes les limitations précédentes, mais est beaucoup moins simple à utiliser.
- **Le langage RPCL et le compilateur RPCGEN.** À partir de la spécification de l'interface,**rpcgen** va générer les fichiers sources des stubs, des fonctions interfaces et des fonctions XDR. Le programmeur n'a plus qu'à écrire le serveur, le client et l'implémentation des fonctions de service.

Premier exemple : programmation sur l'interface de haut niveau



La figure montre les fichiers sources et les deux exécutables produits : le serveur qui contient les fonctions de service RPC et le client qui les appelle :

1. **exaintf.h** définition de l'interface des fonctions distantes,
2. **exa.h** définitions symboliques et définitions de structures,
3. **exafonc.c** implémentation des fonctions de service RPC,
4. **exaser.c** programme principal du serveur qui enregistre le service et attends les requêtes,
5. **exaxdr.c** fonctions de traduction big/little endian,
6. **exaclistub.c** l'adaptateur client ("stub"),
7. **exaclient.c** le code du client qui appelle les fonctions distantes.

Les ordres de compilation et link sont les suivants :

```
$ gcc -c exaxdr.c
$ gcc -c exafonc.c
$ gcc -o exa-serveur exaser.c exafonc.o  exaxdr.o

$ gcc -c exaclistub.c
$ gcc -o exa-client exaclient.c exaclistub.o exaxdr.o
```

```
/* exaintf.h   */ (1)
/* définition d'interface des fonctions de service RPC
 *
   fichier à inclure dans les programmes clients
 */
/*            machine  nom fi  place  buf  taille */
int distread  (char *, char *, long, char *, int);
int distwrite (char *, char *, long, char *, int);
```

fonctions appelables, avec leurs arguments:
le client pourra faire :

```
rep = distread(host, file, 8, buf, 12);
```



```
/* exa.h : fichier de définition de structures */ (2)

int distwrite (char *, char *, long, char *, int);
```

Pour chaque liste d'arguments on fabrique une structure qui la représente :

```
typedef struct {// structure requête write
  char* fichier;// nom fichier
  long  place;  // place (déplacmnt depuis début)
  char* buf;    // data
  int   nbre;   // nbre de car. à écrire
} ecriture;
```

cette structure permet de manipuler la liste d'arguments avec un seul paramètre:
l'adresse de la structure;
--> déclarée dans exa.h

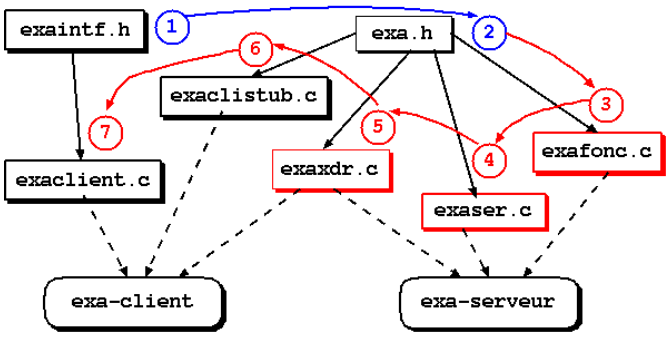

```
/* exa.h : fichier de définition de structures */ (2)
```

Complication pour les arguments en sortie '"out")

```
int distread  (char *, char *, long, char *, int);

typedef struct { // structure requête read
  char* fichier; // nom fichier
  long  place;   // place (deplacmnt depuis début)
  int   nbre;    // nbre de car. à lire
} lecture;

typedef struct { // structure réponse read
  int   nbrelu;  // nbre de car.effectivement lus
  char* buf;     // data
} replecture;
```

--> déclarées dans exa.h


```
/* exa.h : définition de structures */ (2)
#include <stdio.h>,<stdlib.h>, <rpc/types.h>,<rpc/xdr.h>

/*0x20000001= 536870913 ds plage 0x20000000:0x3fffffff */
/* numéros de prog. ou de service RPC et version */
#define EXA_PROG 0x20000001
#define EXA_V1 1        /* version 1 */
#define EXA_READ  1     /* procedure read */
#define EXA_WRITE 2     /* procedure write */
#define TAILLEMAX 1024  /* taille max read ou write */

typedef struct { ... } ecriture;
typedef struct { ... } lecture;
typedef struct { ... } replecture;

/* déclaration des filtres XDR définis dans exaxdr.c*/
bool_t xdr_ecriture(XDR *, ecriture *);
bool_t xdr_lecture(XDR *, lecture *);
bool_t xdr_replecture(XDR *,replecture *);

void *exawrite(ecriture *); // protos fonctions serveur
void *exaread (lecture *);  // renvoient pter non typé
```


```
/* exafonc.c : fonctions de service RPC */
.........
/* les fonctions de service RPC DOIVENT renvoyer
   un pointeur void * sur une zone de mémoire
   obligatoirement allouée en static */

static int repwrite; // renvoyer nbre car.écrits

void *exawrite (ecriture *ecrire) {
  int id;
  id = open(ecrire->fichier, O_WRONLY);
  if(id==-1) {
     repwrite=-1; return(void *)&repwrite;}
  if(lseek(id, ecrire->place, SEEK_SET)==-1) {
     repwrite=-2; return(void *)&repwrite;}
  repwrite = write(id, ecrire->buf, ecrire->nbre);
  close(id);
  return(void *)&repwrite;
}
```



```
/* exafonc.c : fonctions de service RPC */
.........
static replecture repread; // structure
static int bufalloc=0; // allouer buffer une fois

void *exaread (lecture *lire) {
  int id;
  if (bufalloc==0) { bufalloc=1;
      repread.buf = (char *)malloc(TAILLEMAX); }
  id = open(lire->fichier, O_RDONLY);
  if(id==-1) {
     repread.nbrelu=-1; return(void *)&repread;}
  if(lseek(id,lire->place, SEEK_SET)==-1) {
     repread.nbrelu=-2; return (void *)&repread;}
  repread.nbrelu = read(id, repread.buf, lire->nbre);
  close(id);
  return(void *)&repread;
}
```



```
/* exaser.c : le serveur RPC */
#include "exa.h"

main() {
  // commencer par le désenregistrer par sécurité */
  pmap_unset(EXA_PROG, EXA_V1);

  // enregistrer chacune des fonctions de service

  if (registerrpc(EXA_PROG, EXA_V1, EXA_READ,
                  exaread, xdr_lecture,
                  xdr_replecture) ==-1) {
            fprintf(stderr,"Echec register EXA_READ\n");
            exit(2);}

  if (registerrpc(EXA_PROG, EXA_V1, EXA_WRITE,
                  exawrite,xdr_ecriture, xdr_int)==-1){
            fprintf(stderr,"Échec register EXA_WRITE\n");
            exit(2);}
  // le serveur se met en attente des requêtes
  svc_run();  // jusqu'à arrêt du serveur; never return
  fprintf(stderr,"Échec de svc_run\n");
}
```

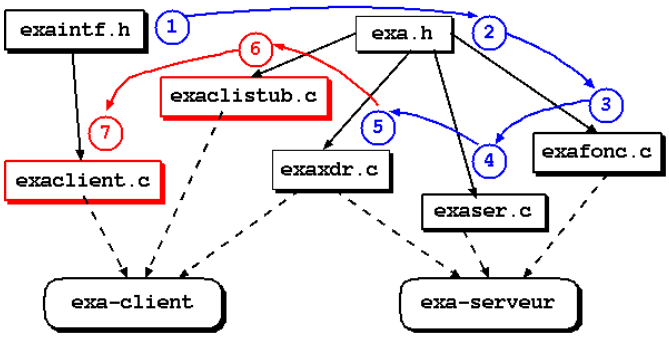
```
/* exaxdr.c : fichier de définition fonctions XDR */
#include "exa.h"
/* voir "man xdr" pour les fonctions XDR dispo. */

bool_t xdr_ecriture(XDR *xdr, ecriture *ecrire) {
return (    xdr_string(xdr, &ecrire->fichier, 64)
         && xdr_long   (xdr, &ecrire->place)
         && xdr_bytes  (xdr, &ecrire->buf,
                             &ecrire->nbre,TAILLEMAX)
         && xdr_int    (xdr, &ecrire->nbre)        );
}
bool_t xdr_lecture(XDR *xdr, lecture *lire) {
return (    xdr_string(xdr, &lire->fichier, 64)
         && xdr_long   (xdr, &lire->place)
         && xdr_int    (xdr, &lire->nbre) );
}
bool_t xdr_replecture(XDR *xdr, replecture *lire) {
int n;
n= lire->nbrelu; if(n<0) n=0; /* si<0, statut err */
return (  xdr_int    (xdr, &lire->nbrelu)
       && xdr_bytes  (xdr, &lire->buf,&n,TAILLEMAX));
}
```

33

```
bool_t xdr_ecriture(XDR *xdr, ecriture *ecrire) {
return (    xdr_string(xdr, &ecrire->fichier, 64)
         && xdr_long   (xdr, &ecrire->place)
         && xdr_bytes  (xdr, &ecrire->buf,
                             &ecrire->nbre,TAILLEMAX)
         && xdr_int    (xdr, &ecrire->nbre)        ); }
```

```
man xdr_long

  xdr_long(xdrs, lp)
  XDR *xdrs;                un stream xdr
  long *lp;

A filter primitive that translates between C long
integers and their external representations. This
routine returns one if it succeeds, zero otherwise.
```

34



35

```
/* exaclistub.c : fonctions "talon" des services RPC
 *  ces fonctions présentent aux programmes clients
 *  l'interface des fonctions distantes et effectuent
 *  l'emballage/déballage des arguments et l'appel
 *  distant effectif vers le serveur
 */
#include "exa.h" , <rpc/rpc.h> , <rpc/clnt.h>

int distread  ( char *host, char *file,
                long place, char *buffer, int size){
  // variables locales pour emballage/déballage
  replecture repread; int rep; lecture lire;
  repread.buf = buffer;
  lire.fichier = file;
  lire.place = place;
  lire.nbre = size;          service   entrées   sorties

  rep = callrpc(host, EXA_PROG, EXA_V1, EXA_READ,
              (xdrproc_t)xdr_lecture,   (char *)&lire,
          (xdrproc_t)xdr_replecture,(char *)&repread);

  if (rep!=RPC_SUCCESS) {clnt_perrno(rep); return -1;}
  return repread.nbrelu;
}
```

36

```
/* exaclient.c : client des services RPC */
 *
 * appelle les fonctions présentées aux programmes
 * clients dans l'interface exaintf.h
 */
#include "exaintf.h"
char buf[32];  int rep;
char buffer[] = "1234567890--";
char file[]= "exatest.txt"; char host[]= "localhost";

main() {
  printf("lire portion de %s sur %s\n",file, host);
  rep = distread(host, file, 8, buf, 12);
  printf("lu %d car. = %s \n",rep,buf);

  printf("ecrire portion de %s sur %s\n",file, host);
  rep = distwrite(host, file, 8, buffer, 12);
  printf("ecrit %d car. \n",rep);

  printf("relire même portion de %s sur %s\n",
         file, host);
  rep = distread(host, file, 8, buf, 12);
  printf("lu %d car. = %s \n",rep,buf);
}
```

37

Ch08 - Le RPC

L'exécution donne :

```
./exa-serveur        ! lancer le serveur
                     ! dans une autre fenêtre :
[root@linux]# rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
..........
 536870913    1   udp  32805
[root@linux]# rpcinfo -u localhost 536870913 1
program 536870913 version 1 ready and waiting

$ more exatest.dat
-abc-def-hij-klm-nop-qrs-tuv-wxy-z01
```

©Michel.Vayssade@UTC.fr – Université de Technologie de Compiègne,UV SR03 2006 - Architectures Internet.

38

Ch08 - Le RPC

```
$ cp exatest.dat exatest.txt

$ ./exa-client
lire portion de exatest.txt sur localhost
lu 12 car. = -hij-klm-nop
ecrire portion de exatest.txt sur localhost
ecrit 12 car.
relire même portion de exatest.txt sur localhost
lu 12 car. = 1234567890--

ctrl-C pour arrêter le serveur =>
[root@linux]# rpcinfo -u localhost 536870913 1
rpcinfo: RPC: Timed out
program 536870913 version 1 is not available
```

©Michel.Vayssade@UTC.fr – Université de Technologie de Compiègne,UV SR03 2006 - Architectures Internet.

39

Ch08 - Le RPC

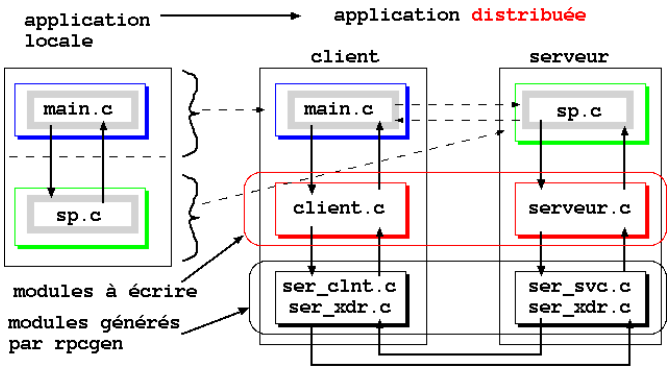
Deuxième exemple : programmation avec rpcgen et rpcl

Prenons l'exemple d'une application locale dans laquelle un programme principal "main.c" appelle des fonctions situées dans un fichier source "sp.c".

On va transformer cette application en deux parties : une application "client" qui contiendra "main.c" et une application serveur qui contiendra "sp.c" et qui rendra les fonctions de "sp.c" appelables à distance par le mécanisme de RPC.

©Michel.Vayssade@UTC.fr – Université de Technologie de Compiègne,UV SR03 2006 - Architectures Internet.

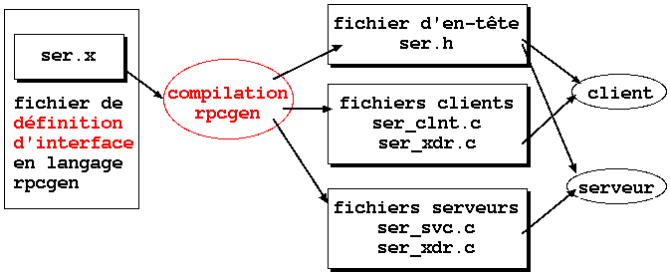
40



Pour chaque fonction de "sp.c" appelée par "main.c", on écrit dans les fichiers de service "client.c" et "serveur.c" desfonctions d'encapsulation qui vont masquer aux yeux du client d'une part, à ceux des fonctions de "sp.c" d'autre part, la complexité des appels à distance.

Pour réaliser les fonctions auxiliaires d'emballage et déballage des arguments d'appel et de réponse par les fonctions de la bibliothèque XDR, on va utiliser l'utilitaire **rpcgen**.

C'est un compilateur qui prend en entrée un fichier de**description d'interface** et qui génère des fichiers **sources** de type **.h** et **.c** qui seront inclus dans les commandes de compilation et édition de liens des programmes serveur et client.



Exemple de fichier rpcgen

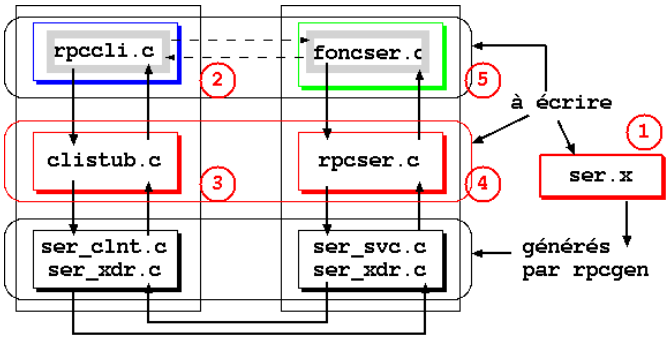
Pour une fonction dont le prototype serait :

```
int proc (int i1, int i2);.
```

On passe obligatoirement par l'intermédiaire d'une **structure**. C'est cette structure qui sera convertie par les fonctions XDR.

```
struct argproc {
      int x1;
      int x2; };
```

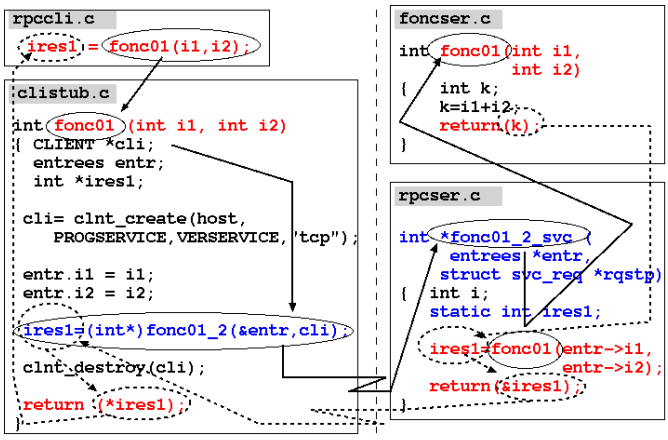
Le compilateur rpcgen va construire les filtres XDR qui vont permettre de transférer cette structure. Il va aussi construire les fonctions d'appel RPC qui vont prendre ces structures en argument et en valeur de retour pour passer les données dans les deux sens.



Compilation :

```
rpcgen ser.x

gcc -o cli rpccli.c clistub.c ser_clnt.c ser_xdr.c

gcc -o ser rpcser.c foncser.c ser_svc.c ser_xdr.c
```

Le fonctionnement des quatre fichiers principaux est donné sur la figure suivante qui montre la trace des appels et retours de résultats.




```
/* ser.x: fichier rpcgen */

struct entrees{ int i1;
                int i2;};
typedef struct entrees entfonc;

struct entrees2{ double i3;
                 double i4;};
typedef struct entrees2 entfonc2;

struct sortie2{ double dd;};

program PROGSERVICE
{  version VERSERVICE
   {
     int FONC01(entfonc) =1;       /* fonction numéro 1*/
     sortie2 FONC02(entfonc2) =2;  /* fonction numéro 2*/

   }=2;     /*version numéro 2 de PROGSERVICE */
} = 839909900;  /* 0x32100005; /* numéro de programme */
                /* =839909381 en décimal */
```

```
/* rpccli.c  - exemple rpc - client */
......
int fonc01(int , int);   /* fonctions distantes */
double fonc02(double, double);
/* fonctions équivalentes à fonc01,. mais locales */
int loc01 (int i1, int i2) {....}
double loc02 (double xx, double xy) {....}

main(int argn, char **argv) { ......
i1 = atoi (argv[1]); i2 = atoi (argv[2]);
i3 = atof (argv[3]); i4 = atof (argv[4]);
t0= clock(); printf("Temps 1=%d\n",t0);

/*Appel à distance, comme si procédures étaient locales*/
  for(i=0;i<50;i++) {
      ires1 = fonc01(i1,i2);
      z = fonc02(i3,i4);  }
t1 = clock(); printf("Temps 2=%d %d\n",t1,t1-t0);
/* Appel aux procedures locales*/
  for(i=0;i<50000;i++) {        // pour comparer appel
      k = loc01 (i1,i2);        // local et à distance
      z = loc02 (i3,i4); }
t2 = clock(); printf("Temps 3=%d %d \n",t2,t2-t1);
}
```

49

```
/* clistub.c - exemple rpc - talon d'appel du client */
#include "ser.h" , <rpc/rpc.h>
#define SERVEUR "localhost"
.......
int fonc01 (int i1, int i2)
{   CLIENT *cli;  /* handle client*/
    entrees entr; /* paramètres d'appel */
    int *ires1;   /* résultat*/
    char *host;   /* nom du serveur distant*/
host=SERVEUR;   /*connexion au serveur*/
cli= clnt_create(host,PROGSERVICE,VERSERVICE,"tcp");
if (cli==NULL) {clnt_pcreateerror(host);return (-1);}

entr.i1 = i1; /* prépare structure argument d'appel */
entr.i2 = i2;
/* faire l'appel rpc à fonc01_1 car dans ser.x on
   déclare un appel à FONC01 version =1 */
    ires1=(int*) fonc01_1(&entr,cli);

if (ires1 ==NULL) {
    clnt_perror(cli,"err appel rpc"); return(-1);}
    clnt_destroy(cli); /* libérer handle connexion */
    return (*ires1);  /* paramètre de retour */
}
```

50

```
/* rpcser.c - exemple rpc - serveur.c talon d'appel */
#include "ser.h" , <stdio.h>
FILE *iout;
int fonc01(int , int);
double fonc02(double, double);
/*appel a fonc01_1 car declare version=1 dans ser.x*/
int *fonc01_1_svc (entrees *entr, struct svc_req *rqstp)
{  int i;
   static int ires1;
   ires1=fonc01(entr->i1,entr->i2); /* appel effectif */
   return(&ires1);
}
/*appel a fonc02_2 car declare version=2 dans ser.x*/
sortie2 *fonc02_2_svc(entrees2 *entr,struct svc_req *rqstp)
{  int i;
   static sortie2 dble;
   dble.dd = fonc02(entr->i3,entr->i4);

   iout = fopen ("log.log","a+b");
   i = fprintf (iout,"fonc02 i3 i4= %f %f dd=%f \n",
                entr->i3,entr->i4,dble.dd);
   fflush(iout); fclose(iout);
   return(&dble); }
```

51

```
/* foncser.c - exemple rpc  */
/* fonc01.c et fonc02.c fonctions de service
/* appelées par rpc */

int fonc01(int i1, int i2) {
    int k;
    k=i1+i2;
    return(k);
}

double fonc02(double i3, double i4)
{   double k;
    k=i3*i4;
    return k;
}
```

Rien ne les distingue d'une fonction ordinaire : tout le travail de transfert d'appel est fait dans les stubs client et serveur

52

Exécution de l'exemple précédent :

```
! dans une première fenêtre
$ ./ser

! dans une deuxième fenêtre
$ ./cli 3 4 1.2 2.
Appel au serveur avec i1=  3, i2=  4, i3=1.200000,i4=2.000000
Temps 1=0
clistub 2.400000
........
clistub 2.400000
resultat fonc01=       7
resultat fonc02= 2.400000
Temps 2=20000 20000
```



```
resultat loc01=       7
resultat loc02= 2.400000
Temps 3=30000 10000
```

Variante de l'exemple précédent :

On modifie le paramètre de retour de la fonction "fonc2" de façon à lui faire renvoyer un réel "double", à la place de la structure contenant un double.

Ce qui est à noter ici ce sont les différences dans la manière de traiter correctement ce paramètre de retour. Les diférences entre les codes sources des deux exemples sont données ci-dessous :


```
/* ser.x: fichier rpcgen */
/* version de ser.x avec FONC02 renvoyant un double
   et simplification déclaration param. fonctions   */

struct entrees{ int i1; int i2;};
struct entrees2{ double i3;
                 double i4;};

program PROGSERVICE
{  version VERSERVICE
   { int FONC01(entrees) =1;      /* fonction numéro 1*/

    double FONC02(entrees2) =2;  /* fonction numéro 2*/

   }=2;     /*version numéro 2 de PROGSERVICE */
} = 839909901; /* 0x32100005; /* numéro de programme */
```

à la place de

```
struct sortie2{ double dd;};

sortie2 FONC02(entfonc2) =2;
```

55

les sources rpccli.c et foncser.c ne sont pas modifiées

```
[vayssade@linux ex2]$ diff rpccli.c ../ex1/rpccli.c

[vayssade@linux ex2]$ diff foncser.c ../ex1/foncser.c

[vayssade@linux ex2]$ diff clistub.c ../ex1/clistub.c
38c38
<     double *ires2; /* resultat */
---
>     sortie2 *ires2; /* resultat */
51c51
<     printf("clistub %f\n",*ires2);
---
>     printf("clistub %f\n",ires2->dd);
57c57
<     return (*ires2);  /* parametre de retour */
---
>     return (ires2->dd);  /* parametre de retour */
```

56

```
[vayssade@linux ex2]$ diff rpcser.c ../ex1/rpcser.c
21c21
< double *fonc02_2_svc(entrees2 *entr,struct svc_req *rqstp)
---
> sortie2 *fonc02_2_svc(entrees2 *entr,struct svc_req *rqstp)
23c23
<    static double dble;
---
>    static sortie2 dble;
25c25
<    dble = fonc02(entr->i3,entr->i4);
---
>    dble.dd = fonc02(entr->i3,entr->i4);
29c29
<              entr->i3,entr->i4,dble);
---
>              entr->i3,entr->i4,dble.dd);
```

seuls sont modifiés les stubs client et serveur

Programme auxiliaire : unreg.c :

Le petit programme suivant permet de désenregistrer un serveur, pour le cas où on a besoin de l'arrêter et de le relancer souvent.

On peut aussi, comme dans l'exemple 1 faire débuter le serveur par un essai de désenregistrement (voir code source de exaser.c).

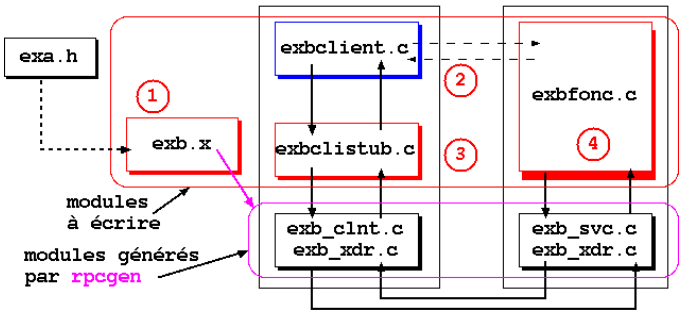

```
/* unreg.c : unregister a program or function
             from portmapper */

#include "ser.h"
#include <rpc/rpc.h>

main ()
{
printf("unregister prog=%d service=%d\n",
       PROGSERVICE, VERSERVICE);

svc_unregister(PROGSERVICE, VERSERVICE);
}
```

rpcgen ser.x

```
pmap_unset(EXA_PROG, EXA_V1);
```


Troisième exemple : reprogrammation avec rpcgen du premier exemple

reprogrammer l'exemple "exa" avec rpcgen



On commence par construire un fichier descriptif dans lequel sont définies les structures d'entrée et de sortie et les foncions appelables par RPC.


```
/* exb.x  définition d'interface distante pour rpcgen */
#define TAILLEMAX 1024 /* taille maxi d'un read ou write */

/* définition strutures échangées, recopiées de exa.h */
/* les commentaires par // pas acceptés par rpcgen    */
struct ecriture {      /* structure requête write */
  string fichier<64>; /* nom fichier, maxi 64 car. */
  long   place;       /* place (deplacement depuis début)*/
  char   buf<TAILLEMAX>;          /* data */
  int    nbre;         /* nbre de car. à écrire */
};
struct lecture {       /* structure requête read */
  string fichier<64>; /* nom fichier, maxi 64 car. */
  long   place;       /* place (déplacement depuis début)*/
  int    nbre;        /* nbre de car. à lire */
};
struct replecture {    /* structure réponse read */
  int   nbrelu;       /* nbre de car. effectivement lus */
  char  buf<TAILLEMAX>;          /* data */
};
```



```
/* exb.x  (suite) */

/* noms de programme (service RPC), version et
   fonctions distantes, copiées de exa.h pour
   montrer la similitude */

program EXB_PROG {
  version EXB_V1 {
   /* les 2 fonctions disponibles dans ce service */
      replecture EXAREAD (lecture) =1;  /* fonc 1 */
      int EXAWRITE(ecriture) =2;     /* fonction 2 */
  } = 2;  /* version 2 */
} = 0x20000002; /* numéro de prog. service RPC */
```


Ensuite on écrit les sources des programmes client, serveur et des adaptateurs, en s'aidant des compilations rpcgen, puis on crèe les exécutables :

```
[vayssade@linux exb]$ ls
exb.x
[vayssade@linux exb]$ rpcgen exb.x ; ls
exb_clnt.c  exb.h  exb_svc.c  exb.x  exb_xdr.c

! demander à rpcgen de faire des squelettes pour les
! adaptateurs client et serveur
[vayssade@linux exb]$ rpcgen -Ss -C exb.x > exbfonc.c
[vayssade@linux exb]$ rpcgen -Sc -C exb.x > temp_cli.c
```



```
nedit exbfonc.c
! reprendre code des fonctions de ../exa/exafonc.c

nedit exbclistub.c
! inspiré de temp_cli.c et ../exa/exaclistub.c

cp ../exa/exaclient.c ./exbclient.c
nedit exbclient.c

gcc -o client exbclient.c exbclistub.c exb_clnt.c exb_xdr.c
gcc -o serveur exb_svc.c exbfonc.c exb_xdr.c
```

L'exécution donne :

```
        ! préparer fichier data de test
cp ../exa/exatest.dat ./exatest.txt

        ! dans une première fenêtre
./serveur
        ! dans une deuxième fenêtre
./client

[root@linux root]# rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
```



```
 536870914    2   udp  32771
 536870914    2   tcp  32777

[root@linux root]# rpcinfo -u localhost  536870914 2
program 536870914 version 2 ready and waiting

[vayssade@linux exb]$ cp ../exa/exatest.dat ./exatest.txt
[vayssade@linux exb]$ ./client
lire portion de exatest.txt sur localhost
lu 12 car. = -hij-klm-nop
ecrire portion de exatest.txt sur localhost
ecrit 12 car.
relire même portion de exatest.txt sur localhost
lu 12 car. = 1234567890--
```



```
/* exbclient.c : client des services RPC */
#include "exaintf.h"
char buf[32];  int rep;
char buffer[] = "1234567890--";
char file[] = "exatest.txt";
char host[] = "localhost";

main() {
  printf("lire portion de %s sur %s\n",file, host);
  rep = distread(host, file, 8, buf, 12);
  printf("lu %d car. = %s \n",rep,buf);

  printf("ecrire portion de %s sur %s\n",
         file, host);
  rep = distwrite(host, file, 8, buffer, 12);
  printf("ecrit %d car. \n",rep);

  printf("relire même portion de %s sur %s\n",
         file, host);
  rep = distread(host, file, 8, buf, 12);
  printf("lu %d car. = %s \n",rep,buf);
}
```

67

```
/* exbclistub.c : stub client des services RPC */
#include "exb.h"  /* from template */

int distread  ( char *host, char *file,
                long place, char *buffer, int size){
  // variables locales pour emballage/déballage
    CLIENT *clnt;
    replecture  *result_1;
    lecture  exaread_2_arg;
    int  *result_2;
    ecriture  exawrite_2_arg;

    /* connexion au serveur RPC */
    clnt= clnt_create(host, EXB_PROG, EXB_V1,"udp");
    if(clnt==NULL) {clnt_pcreateerror(host);exit(1);}

    /* préparer la structure d'appel */
    exaread_2_arg.fichier = file;
    exaread_2_arg.place = place;
    exaread_2_arg.nbre = size;

    /* faire l'appel RPC */   ...
```

68

```
/* exbclistub.c : stub client des services RPC */
...
 /* faire l'appel RPC */
    result_1 = exaread_2(&exaread_2_arg, clnt);
    if (result_1 == (replecture *) NULL) {
         clnt_perror (clnt, "call failed"); }

    clnt_destroy(clnt); /* liberer handle connexion */

 /* récupérer résultat, le ranger dans buffer client */
 /* contenu de result_1 n'est défini qu'après
    exaread_2() :un tableau "buf" déclaré dans exb.x
    est traduit par rpcgen en une struture contenant:
          int buf_len;    la longueur du tableau
          void * buf_val; l'adresse du tableau       */

 /* copier data reçues ds buffer fourni par appelant */
    strncpy (buffer,result_1->buf.buf_val,
                    result_1->buf.buf_len);
    return result_1->buf.buf_len;
}
```

69

```
/* exbfonc.c : serveur des services RPC */
 * implémentation des fonctions de service  */
#include "exb.h"
#define TAILLEMAX 1024 /* taille max read ou write */
/* devrait être dans exb.h. Bug rpcgen sur linux ? */
#include <fcntl.h>
static int repwrite; // write renvoie nbre car.écrits
static int bufalloc=0; // allouer buffer une fois

replecture *
exaread_2_svc(lecture *lire, struct svc_req *rqstp)
{ static replecture  repread;
  int id;
  if (bufalloc==0) { bufalloc=1;
      repread.buf.buf_val = (char *)malloc(TAILLEMAX);
      repread.buf.buf_len = TAILLEMAX; }
  id = open(lire->fichier, O_RDONLY);
  if(id==-1){repread.nbrelu=-1;return(void *)&repread;}
  if(lseek(id,lire->place, SEEK_SET)==-1) {
     repread.nbrelu=-2; return (void *)&repread;}
  repread.nbrelu = read(id, repread.buf.buf_val,
                            lire->nbre);
  repread.buf.buf_len = repread.nbrelu;
  close(id);  return &repread; }
```

70

Autres possibilités des RPCs

Diffusion : Il est possible de diffuser ("broadcast") une requête avec le transport UDP par la primitive **clnt_broadcast()**.

Démarrage et arrêt du serveur par le client :

Le démon **inetd** peut lancer d'autres démons mais aussi des services RPC. Le paragraphe suivant est extrait de la documentation BSD de inetd.

Le programme inetd est lancé lors du démarrage par /etc/rc (voir rc(8)). Il écoute le réseau pour détecter les connexions sur certaines sockets internet. Lorsqu'une connexion est trouvée, il détermine à quel service correspond cette socket et appelle le programme qui peut satisfaire la requête. Ce dernier est appelé avec le service de

sockets et ses description d'entrée, sortie et erreur standard. Une fois le programme terminé, inetd poursuit son écoute (sauf dans quelques cas décrits plus loin). Essentiellement, inetd permet de faire tourner un démon pour en appeler plusieurs autres, réduisant ainsi la charge du système.

Lors de son exécution, inetd détermine sa configur ation à partir d'un fichier qui, par défaut, est /etc/inetd.conf. Pour les services ONC RPC, le nom est celui qui apparait dans le fichier /etc/rpc.

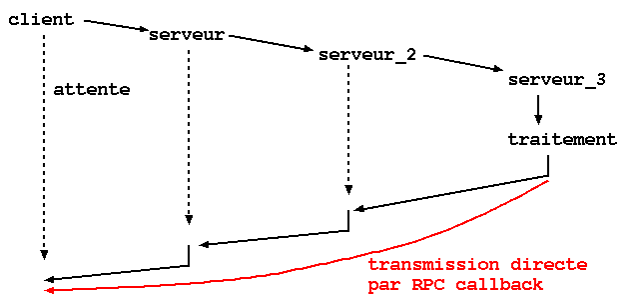
Pour utiliser le mécanisme, le client fait un rexec() du serveur avec le numéro du programme obtenu en consultant le portmapper. Le serveur est lancé par inetd. Le client attends quelques secondes et appelle le serveur.

On arrête le serveur en invoquant une fonction de celui-ci qui se

termine par un exit().

RPC et "callback" RPC

On peut se trouver dans une situation ou un serveur RPC fait lui-même appel par RPC à un autre serveur. Dans ce cas, il peut être plus efficace que le serveur "final" renvoie la réponse directement à l'appelant de départ plutôt que de remonter successivement tous les appels.

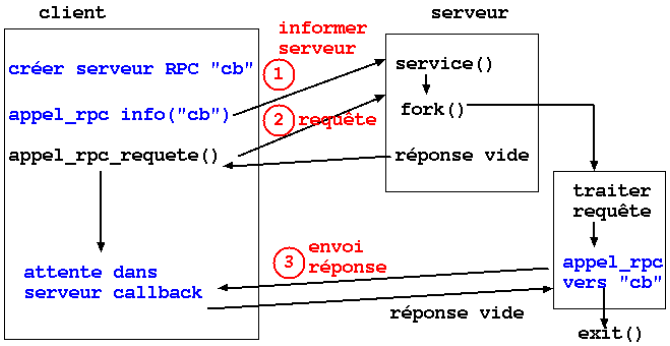


On prépare l'appel en callback en créant deux interfaces. Par exemple si on utilise rpcgen :

- **ser.x** : pour l'interface du serveur,
- **serb.x** : pour l'interface "callback" du client.

Dans ser.x on ajoute une fonction info(adr_callback) qui va permettre au client de donner au serveur le nom de la machine du client (host) et l'identité du seveur callback qu'il aura au préalable enregistré auprès du portmapper de la machine cliente.

Puis, le client envoie sa requête au serveur. Le serveur renvoie une réponse vide qui sert d'acquitement, et le serveur fork un fils qui va traiter la requête et faire un appel RPC vers le serveur callback du client pour transmettre au client les résultats de sa requête.



Ch08 - Le RPC. Exemples de questions

➤ Situez (par un schéma) le RPC dans la pile de protocoles IP.
Réponse : diapo 5.

➤ Donner par un schéma le principe du RPC.
Réponse : diapo 6.

➤ Quel est, l'élément *fondamental* qui sépare l'appel de procédure habituel de l'appel par RPC ? (nota : le mot réseau n'est pas la bonne réponse).
Réponse : diapo 7 : espaces virtuels identiques versus différents.

➤ Détaillez les étapes de fonctionnement du RPC. Réponse : diapo 8.

➤ Quelles caractéristiques de l'application permettent de choisir UDP

comme transport du RPC ? Réponse : diapo 10.

➤ Faites un schéma détaillé des différents éléments impliqués dans un appel RPC avec le RPC "ONC".
Réponse : diapo 12.

➤ Décrivez le rôle du "portmapper".
Réponse : diapos 16 et 17.

➤ Décrire comment est composée une application RPC construite avec le compilateur "rpcgen".
Réponse : diapos 41 à 43.

➤ Donner un exemple de contenu du fichier d'entrée du compilateur "rpcgen".
Réponse : diapo 44.

➤ Quelle précaution indispensable doit-on prendre dans le serveur pour le stockage des données destinées au client ?
Réponse : diapo 70 (et 51) : déclaré en static.

➤ Comment fonctionne le "rappel par RPC" ("RPC callback") ?
Réponse : diapo 76.

Ch08 - Le RPC.

Fin du chapitre.

Ch09 - Le Web et http

- Le Web : présentation générale
- L'URL et le protocole HTTP
- Les transactions HTTP et les différentes requêtes
- Le Web, HTTP et HTML
- Quelques exemples HTML

Le Web et http : fonctionnement du protocole

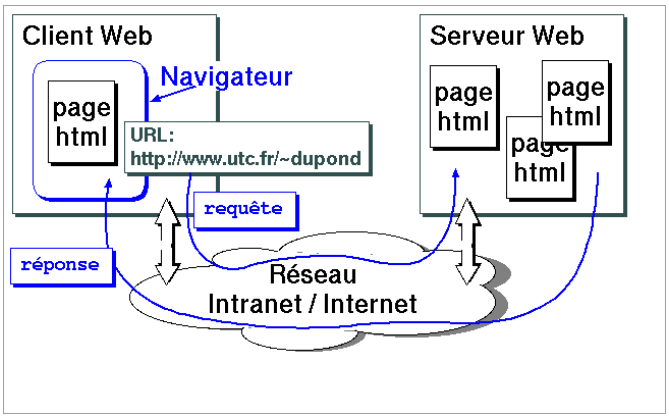
Le web : si vous ne savez pas ce que c'est, vous venez sans doute d'une autre planète ! !

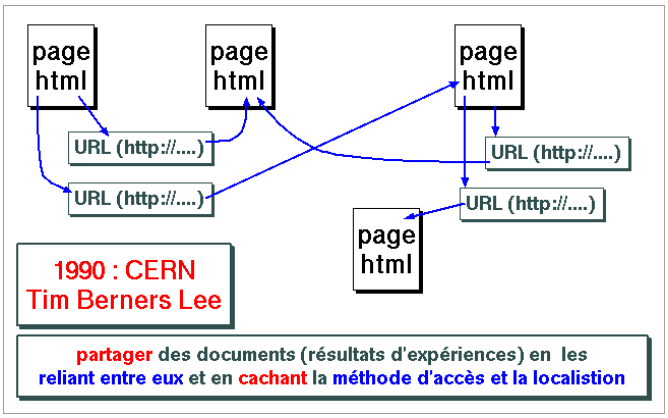
Le "oueb" est fondé sur le protocle "**http**" (Hyper Text Transfer Protocole).

Comme souvent il a une origine "accidentelle". Une technique un peu "bricolée" pour des usages internes s'est répandue partout pour une raison unique : elle est d'utilisation **facile**.

L'objectif des concepteurs était qu'un grand nombre de clients puissent accéder à des données réparties sur une collection de serveurs, sans avoir à se soucier de la localisation de ces serveurs.

➤ l'accès à un document est basé sur le concept d'**URL** : Universal Ressource Locator. Le mot important ici est "**Universal**".





Le Web : le début

➤ 1990 : création

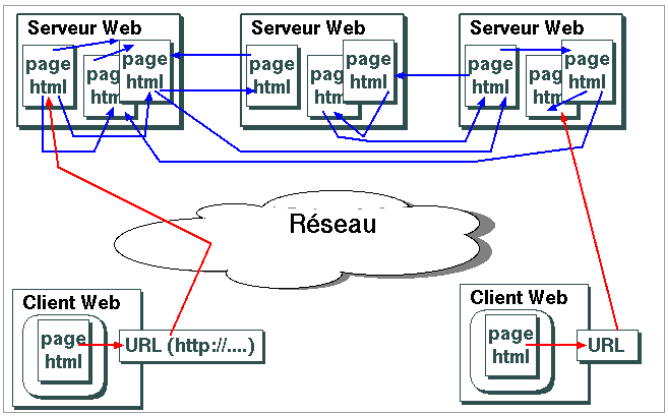
➤ 1993 : au NCSA, Univ.Illinois à Urbana-Champaign : Mosaic le premier browser (Arpenteur, Fureteur, Navigateur, Explorateur) ;

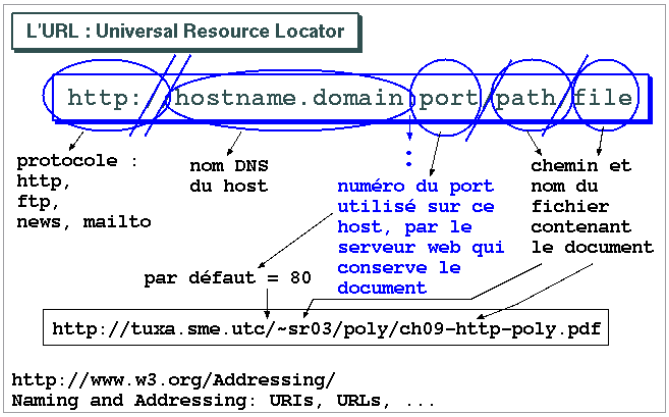
➤ 1994 : Netscape Navigator

➤ Serveurs : **Apache**, Netscape, SUN HotJava, ...

La référence sur les protocoles du web c'est le **W3C** (World Wide Web Consortium) :

http ://www.w3.org/





L'URL

L'URL est fourni au navigateur (soit parce qu'on a cliqué sur un lien dans une page affichée, soit qu'on l'a entré au clavier).

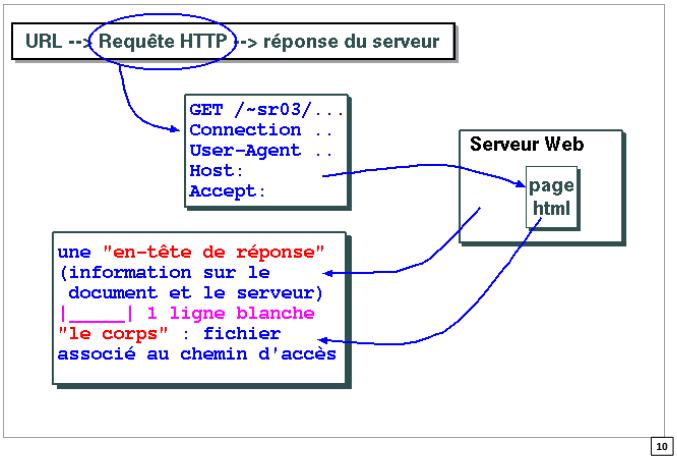
Le navigateur analyse l'URL selon le schéma précédent et envoi une requête au serveur web décrit par cet URL.

Par exemple : http ://tuxa.utc.fr/~sr03/td1.html provoque l'envoi au serveur de :

```
GET /~sr03/td1.html HTTP/1.1
Connection : Keep-Alive
User-Agent : Mozilla
Host : tuxa.utc.fr
Accept-Charset : iso-8859-5, unicode-1-1
Accept : image/gif, image/x-xbitmap, image/jpeg, */*
```

L'URL : requête au serveur

- **GET** : navigateur demande un document au serveur ;
- **/~sr03/td1.html** : chemin et nom du document depuis la racine du serveur ;
- **HTTP/1.1** : en utilisant le protocole version 1.1 ;
- **Connection : Keep-Alive** : demande de maintenir la connexion TCP ouverte jusqu'à une fermeture explicite ;
- **User-Agent : Mozilla** le client s'identifie ;
- **Host : tuxa.utc.fr** le host à qui la requête est envoyée (car un même serveur peut être connu sous plusieurs noms DNS différents).
- **Accept : ...** : les types de documents acceptés par le navigateur.

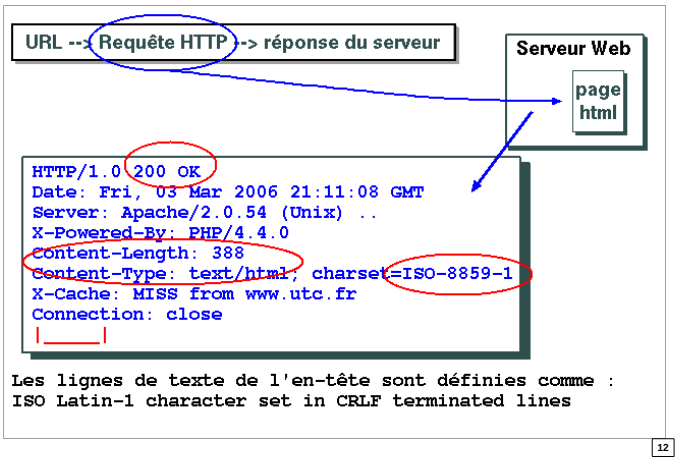


10

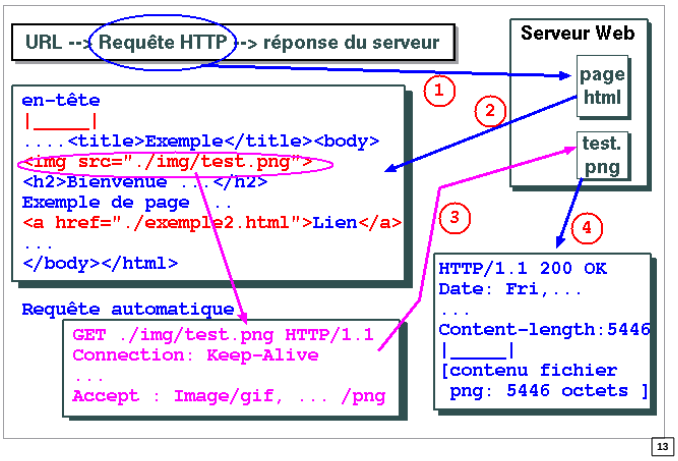
Réponse du serveur

```
> telnet www.utc.fr 80
Trying 195.83.155.17...
Connected to floyd.utc.fr.  Escape character is '^]'.
GET / HTTP/1.0
User-Agent: Mozilla
Host:  www.utc.fr
Accept: image/gif, image/x-xbitmap, image/jpeg, */*

HTTP/1.0 200 OK
Date: Fri, 03 Mar 2006 21:11:08 GMT
Server: Apache/2.0.54 (Unix) mod_jk/1.2.13 PHP/4.4.0
X-Powered-By: PHP/4.4.0
Content-Type: text/html
X-Cache: MISS from www.utc.fr
Connection: close

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

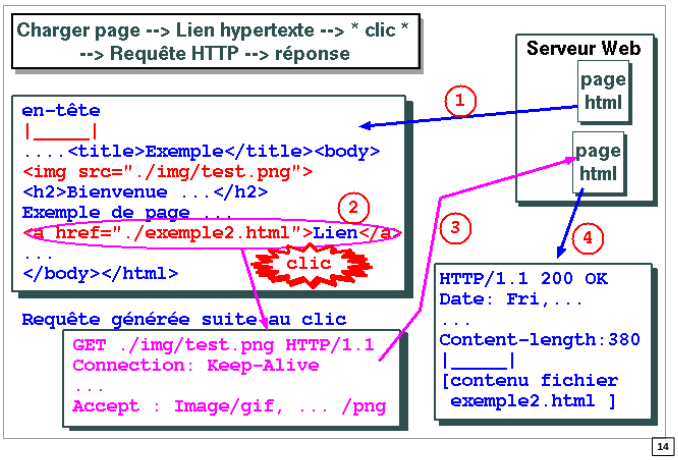
11



12



13



14

Le protocole HTTP

HTTP est un protocole qui reste simple car il **utilise une connexion TCP**.

Sur cette connexion TCP s'échanges des **suites de lignes de texte** séparées par de **simples lignes blanches (vides)**.

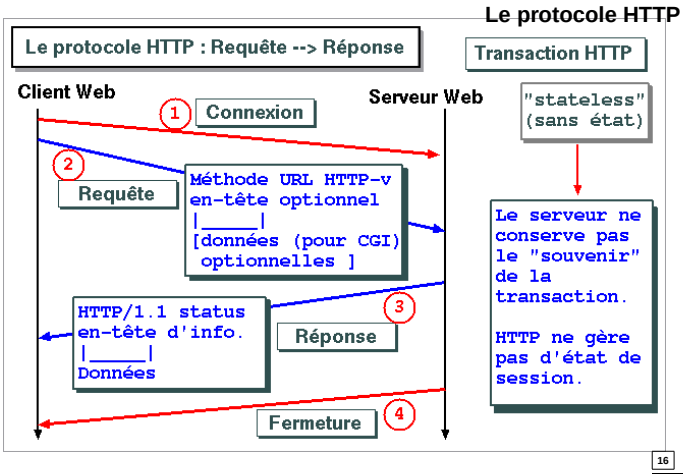
L'exemple vu plus haut avec l'envoi d'une requête grâce à un simple client telnet montre la simplicité de l'interaction.

Lorsque l'on doit transférer des données binaires (par exemple des images), on les fait précéder d'une ligne de texte qui **indique leur longueur**.

TCP garantissant la correction du flux, ces conventions simples suffisent à faire des échanges fiab les entre le client et le serveur.

Pour plus d'infos sur le protocole :
http ://www.w3.org/Protocols/HTTP/HTTP2.html

Le protocole HTTP



16

HTTP : Requêtes simples

GET

```
Le client veut juste
demander un document:
GET /chemin/doc.htm HTTP/1.1
```

HEAD

```
Le client veut juste
de l'information sur
un document, mais pas
le document lui-même :
HEAD /chemin/doc.htm HTTP/1.1
```

```
Le client
veut
seulement
récupérer
de
l'information
```

17

HTTP : Requêtes avec envoi d'informations

POST

```
On fournit de l'information au serveur
(par ex. issue d'un formulaire) par:
  POST /chemin/cgi-bin/form.php HTTP/1.1
  en-tête
  |_____| ligne blanche
  var1=data1&var2=data2&....
```

GET + infos

```
GET + Format "URL encodé"
<FORM>...method=GET>
l'info. saisie dans la forme est passée au serveur par:

  GET /ch.../form.php?var1=data1&var2=data2 HTTP/1.1
```

fichier ? var = data & var = data ...

18

HTTP : Requêtes modifiant le contenu du serveur

POST

```
POST permet de charger un fichier sur le
serveur en créant un nouveau document :

  POST /chemin/cgi-bin/form.php HTTP/1.1
  en-tête
  Content-type: multipart/form-data:
  boundary=arandomstringc0p4Jq0M2Yt
  Content-Length: 1213
  |_____| ligne blanche
  --arandomstringc0p4Jq0M2Yt
    ici ... données du fichier ...
  --arandomstringc0p4Jq0M2Yt--
```

➡ http://www.w3.org/Protocols/rfc1341/7_2_Multipart.html

19

HTTP : Requêtes modifiant le contenu du serveur

PUT — destiné à modifier le contenu d'un URL existant

```
PUT /chemin/fichier.html HTTP/1.1
en-tête
...
Content-Length: 1213
|_____| ligne blanche
<html>
<head>....</head>
<body>
  <p> ... texte ... </p>
</body></html>
```

20

HTTP : autres Requêtes du client

DELETE — destiné à effacer un URL

```
  DELETE /chemin/fichier.html HTTP/1.1
  en-tête
  ...
Serveur répond (si succès):
  HTTP/1.1 200 OK
  Date: ...
  .....
  <h1>URL deleted.</h1>
```

TRACE — voir ce que devient une requête à travers un proxy

OPTIONS — Demander les options disponibles pour un URL donné. Serveur répond (par exemple) : Public: GET, HEAD, POST

21

Le protocole HTTP : codes de réponse

Série	Signification
101-199	Information
200-299	Requête client réussie
300-399	Requête client redirigée. Une autre action est nécessaire pour récupérer le document.
400-499	Requête client incomplète ou erronée.
500-599	Erreur dans le serveur

Le Web, HTTP et HTML (➤ http ://www.w3.org/MarkUp/)

HTTP est un protocole basé sur l'échange de messages en texte.

Les principaux documents du Web sont des fichiers textes. On ajoute à ces textes une **structure** et à l'aide de **balises** ("tags") tels que <title>, <h2>, <head>, <p>, <ol>, <li>, etc ...

HTML est un langage à balise issu d'un langage antérieur et bien plus complexe **SGML** défini par la communauté des documentalistes et gestionnaires de bibliothèques pour décrire complètement des documents de tout type.

SGML n'a jamais été implémenté complètement dans un système informatique, mais il a donné naissance à HTML puis à **XML**.

Après la définition et le succès d'usage de XML, on a fait évoluer HTML en **XHTML** qui reprend et étend HTML en XML.

HTML : premiers exemples

Les balises donnent une **structure** aux documents HTML :

```
<HTML>
<HEAD>
<TITLE>Titre du document</TITLE>
</HEAD>
<BODY>
<P>Insérer du texte ou des images ici</P>
<P>Un paragraphe</p>
</BODY>
</HTML>
```

Un document HTML devrait de plus commencer par une déclaration de type de document :

< !DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2//EN">

Le programme **tidy** permet de vérifier la correction d'un document html :**> tidy page.html** va donner sur stdout la liste des erreurs ou warning par rapport à la norme (même si la plupart des browsers affichent "correctement" un document contenant des erreurs).

Autre exemple :

```
<HTML>
   <HEAD>
   <!-- commentaire, (en fait du SGML ignoré par HTML)  -->
   <TITLE>Titre affiché dans la barre de titre du browser</TITLE>
   </HEAD>
<BODY>
<H1>Ceci est un titre de niveau 1 (le plus gros)</H1>
<p><b>Un premier paragraphe :</b> HTTP est basé sur des
fichiers textes.</p>
<p><i>Un deuxième paragraphe :</i> Le web : bla bla ...</p>
```



```
<p><b><i>Un troisième :</i></b></p> <u>exemple souligné.</u>
<br><hr><br>
Les balises <inconnues> sont ignorées.<br>
<pre>
du texte "pré-formaté",
        . qui apparaît non modifié,
        . avec les mêmes sauts de lignes,
        . et les mêmes          espaces.
</pre>
</BODY></HTML>
```

HTML décrit la structure du document, ainsi (hélas) que des indications sur la façon d'imprimer certains éléments. Mais cette façon dépend de la configur ation du browser sur laquelle l'auteur du document n'a pas d'action.

Ce mélange des genres entre structure et "aspect" est mauvais et est corrigé par l'utilisation des **CSS** et de **XHTML**.

HTML : liens hypertextes

```
<HTML>
<HEAD>  <TITLE>titre</TITLE>  </HEAD>
<BODY>
<H3>Titre de niveau 3 </H3>
<p><b>Un premier paragraphe :</b> Exemple de liens HTTP.</p>
<br>
Un lien "absolu" :
<A HREF="http://www.utc.fr/interne">Aller page interne UTC</A>
<br>
Un lien "relatif" : <A HREF="./test3.html">voir test3.html</A>
<br><hr>
```



```
<pre> Pour "dessiner" avec du texte "pré-formaté", il faut utiliser
une fonte fixe telle que "Courier" :
<font face="Courier New" size=+2 color="#ffff00">
        +-------------+
        | un "dessin" |
        +-------------+
</font></pre>
</BODY></HTML>
```

Codes de couleur spécifiés par : color="#RRGGBB".
RR, GG ou BB = 00 à FF (de 0 à 256 valeurs de rouge, vert, bleu).
Exemple : "#FFFF00" = 100% de rouge + 100% de vert + 0 = jaune.
➤ Voir **http ://www.w3.org/MarkUp/Guide/Style**.

HTML : liens hypertextes ; étiquettes dans le document

```
<HTML><HEAD><TITLE>
Exemple de document avec renvois internes</TITLE></HEAD><BODY>
<A NAME="debut"></A><!-- définition d'un étiquette -->
<H3>Exemple de document avec renvois internes : début.</H3>
<UL><LI><A HREF="#label1">Texte 1</A></LI>
    <LI><A HREF="#label2">Texte 2</A></LI></UL>
<HR>
<A NAME="label1"></A><H3>Texte 1</H3>
<P>Texte 1 ......</p>
<A HREF="#debut">debut</A></P>
<HR>
<A NAME="label2"></A><H3>Texte 3</H3>
<P>Texte 3 ...... On peut aussi faire référence à une étiquette
dans un autre fichier :
<A HREF="test7.html#debut">aller à "debut" dans test7.html.</A></p>
<A HREF="#debut">debut</A></P>
</BODY></HTML>
```

29

HTML : tables

```
<HTML><HEAD><TITLE>
Exemple de page construite avec une table</TITLE></HEAD><BODY>
<table cellspacing=10 cellpadding=5 width=100\% border=1>
<tr><!-- rangée 1 -->
     <td width=10\%> cellule de gauche </td>
     <td width=80\%> cellule du milieu </td>
     <td width=10\%> cellule de droite </td>
</tr>
<tr><!-- rangée 2 -->
        <td width=10\%> </td> <td width=80\%>
<p>cellspacing=10  => espace entre deux cellules</p>
<p>cellpadding=5  => espace entre la bordure d'une cellule et le
   texte où l'image à l'intérieur de la cellule.</p>
<p>width=100\%  => utiliser toute la largeur de la page.</p>
<p>border=1  => un pixel de large pour la bordure: trace un cadre.</p>
        </td><td width=80\%> </td> </tr>
</table></BODY></HTML>
```

30

HTML : formes de saisie

```
<HTML><HEAD><TITLE>
Exemple de page avec une forme</TITLE></HEAD><BODY>
<H3>Exemple de page avec une forme de saisie.</H3>
<FORM METHOD=GET ACTION=cgi-bin/test8.cgi>
<P>Entrez vos Noms et Adresse :<BR>
<P>Nom : <INPUT NAME=nom SIZE=40>
<P>Adresse :<INPUT NAME=adresse  SIZE=60>
<P><INPUT TYPE=submit VALUE=Envoyer>
      <INPUT TYPE=reset VALUE=Effacer>
</FORM></BODY></HTML>
```

ACTION=cgi-bin/test8.cgi ➤ les serveurs web sont souvent configurés pour n'accepter les "cgi" que dans des répertoires particuliers que l'on peut protéger (**drwx–x–x**). Ces fichier ".cgi" peuvent être écrits dans **n'importe quel** langage (PHP, perl, python, C, C++, bash, ...).

31

HTML : formes de saisie ; exemples de widgets

```
<FORM METHOD=GET ACTION=cgi-bin/test9.cgi>
<P>Entrez votre nom et vos choix :<BR>
<P>Nom : <INPUT TYPE=TEXT NAME=nom SIZE=40>
<P>Choisir 1 parmi  3 :
<INPUT TYPE=radio NAME=choix1  value="un">numéro un
<INPUT TYPE=radio NAME=choix1  value="deux">numéro deux
<INPUT TYPE=radio NAME=choix1  value="trois">numéro trois
<P>Cocher si urgent : <INPUT TYPE=checkbox Name=urg>
<p>choisir  1 dans menu :
<SELECT NAME=fichier SIZE=3>
<OPTION>fichier1.txt <OPTION>fichier2.txt
<OPTION>fichier3.txt <OPTION>fichier4.txt  <OPTION>fichier5.txt
</SELECT>
<P><INPUT TYPE=submit VALUE=Envoyer><INPUT TYPE=reset VALUE=Effacer>
</FORM>
```

Rem : <SELECT .. SIZE=3> => menu avec "ascenceur" ;
Si SIZE=1 => menu "déroulant".

32

Client "pull"

Il est possible, de faire demander un rafraichissement automatique de la page par le navigateur. C'est alors le navigateur qui redemande la page : le serveur n'a pas besoin de mémoriser l'identité du client.

doc.html

```
 1 <HTML>
 2 <HEAD>
 3 <!- demo client pull : HTML 3.0 (Netscape Navigator >= 1.1) ->
 4 <META HTTP-EQUIV="Refresh" CONTENT=5>
 5 <TITLE>Utilisation client pull
 6 pour appeler un autre document</TITLE>
 7 </HEAD><BODY>
 8 <P>Insérer du texte ou des images ici</P>
 9 <img src="penguin.png">
10
```


doc.html (suite)

```
11 </BODY></HTML>
```

doc.php3

```
 1 <HTML>
 2 <HEAD>
 3 <!- demo client pull : HTML 3.0 (Netscape Navigator >= 1.1) ->
 4 <META HTTP-EQUIV="Refresh" CONTENT=5>
 5 <TITLE>Utilisation client pull
 6 pour appeler un autre document</TITLE>
 7 </HEAD><BODY>
 8 <P>Insérer du texte ou des images ici</P>
 9 <P>Date, Heure =
10 <? setlocale ('LC_TIME', 'fr');
11 print(strftime("%A %d %B %Y %H :%M :%S"));?>
12
```


doc.php3 (suite)

```
13 </BODY></HTML>
```

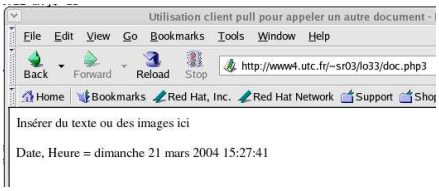


FIG. 1: doc.php3

Selon ce printcipe, on peut mettre en place des chaines de pages qui s'appellent les unes les autres.

doc1.html

doc1.html (suite)

```
 1 <HTML><HEAD>
 2 <META HTTP-EQUIV="Refresh" CONTENT="5; URL=doc2.html">
 3
 4 <TITLE>Utilisation client pull
 5 pour appeler un autre document</TITLE>
 6 </HEAD><BODY>
 7 <font size='+2'>
 8 <P>Insérer du texte ou des images ici</P>
 9 <P>ici doc1.html qui va appeler doc2.html</P>
10 </font>
11 </BODY></HTML>
```

doc2.html

```
 1 <HTML><HEAD>
 2 <META HTTP-EQUIV="Refresh" CONTENT="5; URL=doc1.html">
```


doc2.html (suite)

```
 3
 4 <TITLE>Utilisation client pull
 5 pour appeler un autre document</TITLE>
 6 </HEAD><BODY>
 7 <font size='+2'>
 8 <P>Insérer du texte ou des images ici</P>
 9 <P>ici doc2.html qui appelle doc1.html</P>
10 </font>
11 </BODY></HTML>
```

Un autre exemple, avec une interaction avec le client est donné ci-dessous.

sdoc1.html

```
 1 <HTML><HEAD>
 2 <META HTTP-EQUIV="Refresh" CONTENT="5; URL=sdoc2.html">
 3
```


sdoc1.html (suite)

```
 4 <TITLE>Utilisation client pull
 5 pour appeler un autre document</TITLE>
 6 </HEAD><BODY>
 7 <font size='+2'>
 8 <P>Insérer du texte ou des images ici</P>
 9 <P>ici sdoc1.html :  clic sur stop ou appelle sdoc2.html</p>
10 <CENTER>
11 <A HREF="stop.html"><IMG SRC="stop.gif" BORDER=0></A>
12 </CENTER>
13 </font>
14 </BODY></HTML>
```

FIG. 2: sdoc1.html

sdoc2.html

```
 1 <HTML><HEAD>
 2 <META HTTP-EQUIV="Refresh" CONTENT="5; URL=sdoc3.html">
```


sdoc2.html (suite)

```
 3
 4 <TITLE>Utilisation client pull
 5 pour appeler un autre document</TITLE>
 6 </HEAD><BODY>
 7 <font size='+2'>
 8 <P>Insérer du texte ou des images ici</P>
 9 <P>ici sdoc2.html qui appelle sdoc3.html</P>
10 </font>
11 </BODY></HTML>
```

FIG. 3: sdoc2.html

sdoc3.html

```
 1 <HTML><HEAD>
 2 <META HTTP-EQUIV="Refresh" CONTENT="5; URL=sdoc1.html">
 3
 4 <TITLE>Utilisation client pull
 5 pour appeler un autre document</TITLE>
```


sdoc3.html (suite)

```
 6 </HEAD><BODY>
 7 <font size='+2'>
 8 <P>Insérer du texte ou des images ici</P>
 9 <P>ici sdoc3.html qui appelle sdoc1.html</P>
10 </font>
11 </BODY></HTML>
```

stop.html

```
 1 <HTML><HEAD>
 2 <TITLE>Utilisation client pull
 3 pour appeler un autre document</TITLE>
 4
 5 <script language="JavaScript">
 6         function pushgo() {
 7                 alert("on continue");
```


stop.html (suite)

```
 8                 location = 'sdoc1.html';
 9         }
10         function pushstop() {
11                 alert("on arrete");
12                 location = 'arret.html';
13         }
14 </script>
15
16 </HEAD><BODY>
17 <P>Insérer du texte ou des images ici</P>
18 <P>
19 <form>
20 <input  type="button" name="Button1" value="continuer ?"
21         onclick="pushgo()">
22 <input  type="button" name="Button2" value="arreter ?"
23         onclick="pushstop()">
24 </form>
```


stop.html (suite)

```
25 </BODY></HTML>
```

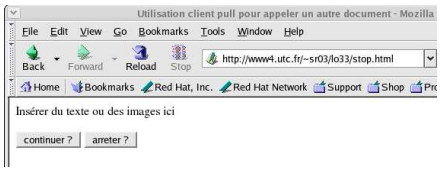


FIG. 4: stop.html

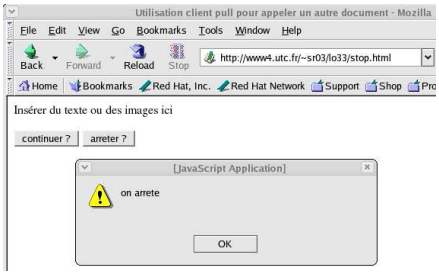


FIG. 5: stop.html javascript confirm

arret.html

```
 1 <HTML><HEAD>
```


arret.html (suite)

```
 2 <TITLE>Utilisation client pull
 3 pour appeler un autre document</TITLE>
 4 </HEAD>
 5 <BODY>
 6 <P>Insérer du texte ou des images ici</P>
 7 <font size='+2'>
 8 <P>ici arr&ecirc;t.html</p>
 9 </font>
10 <hr>
11 </BODY></HTML>
```

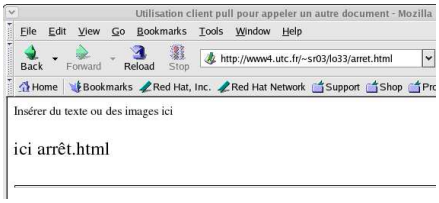



FIG. 6: arret.html

Ch09 - Le Web et http.

Les questions sur ce chapitre sont reportées à la fin du ch.11.

Fin du chapitre.

Ch10 - Les CSS

- Les CSS
- Les différents types
- Quelques exemples de feuilles CSS

Première feuille CSS : css01.htm

```
<HTML><HEAD><TITLE>fichier css01.htm</TITLE>
<STYLE TYPE="text/css">
<!-- /* Hide content from old browsers */
P {
font-size: 10pt;
font-family: Geneva, sans-serif;
color: #000000;
}
UL {
font-size: 12pt;
font-family: Geneva, sans-serif;
color: #FF0000;
}
/* end hiding content from old browsers */ -->
</STYLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF" TEXT="#000000"
 LINK="#008080" VLINK="#000080">

<font size="+2"> Début du fichier css01.htm <br>
<hr>
<P>Paragraphe ... bla bla bla
</P>
<hr><br>
<UL><LI>Paragraphe PA... bla bla bla</LI>
</UL>
<hr><br>
Fin du fichier css01.htm <br>
</font>
</BODY></HTML>
```

Début du fichier css01.htm
Paragraphe ... bla bla bla
• Paragraphe PA... bla bla bla
Fin du fichier css01.htm

<P>...</P> il FAUT refermer toutes les balises

Les types de CSS :

Il y a quatres types de CSS :

- **inline,**
- **ID,**
- **class,**
- **html ;**

Ils s'appliquent **dans cet ordre de priorité** : d'abord les "inline", puis, s'il y en a ceux définis par **ID**, puis ceux définis par **class**, et enfin le html classique. D'où le nom de styles "en cascade".

Attention : le mot clé "ID" étant aussi utilisé par JavaScript, il y a un conflit potentiel. Dans ce cas utiliser la méthode "class".

Exemple des types de CSS : css02.htm

```
<HTML><HEAD><TITLE>fichier css02.htm</TITLE>
<STYLE TYPE="text/css">
P { font-size: 10pt; font-family: Geneva,
    sans-serif; color: #000000; }
#pa { Background-Color: #00EEFF; color: #FF0000;
  font-size:12pt;}
P.marge { font-size: 11pt; margin-left:2.5cm;
color: #0000FF; }
</STYLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF" TEXT="#000000"
 LINK="#008080" VLINK="#000080">
<font size="+2"> Début du fichier css02.htm <br>
Formatage "html" font size="+2".<br>
<P id="pa">Paragraphe
formatage ID="pa" ...
bla bla bla bla bla bla.
</P>
<P>Paragraphe formatage de
la classe P ...
bla bla bla bla bla bla.
</P>
```

Début du fichier css02.htm
Formatage "html" font size="+2".
Paragraphe formatage ID="pa" ... bla bla bla bla bla bla.
Paragraphe formatage de la classe P ... bla bla bla bla bla bla.
Paragraphe formatage de la classe P ... bla bla bla bla bla bla mais avec du texte en format ID="pa" au milieu du paragraphe.
Paragraphe formatage de la classe P.marge ... bla bla bla bla bla bla. Bla bla bla bla bla bla.
Paragraphe formatage inline avec du texte en format ID="pa" au milieu du paragraphe.
Fin du fichier css02.htm
Retour au formattage par défaut html.

Exemple des types de CSS : css02.htm

```
P.marge { font-size: 11pt; margin-left:2.5cm;
color: #0000FF; }
</STYLE>
.......  .........
<P>Paragraphe formatage de la classe P ...
bla bla bla bla bla bla<br>
mais <span id="pa">avec du texte en format
ID="pa"</span> au milieu du paragraphe.
</P>
<P class="marge">Paragraphe formatage de la classe
P.marge ... <br>
bla bla bla bla bla bla. Bla bla bla bla bla bla.
</P>
<P style="Background-Color:#FFFF00;
   font-family:Helvetica;
   font-size:18pt;">
Paragraphe formatage inline
<span id="pa">avec du texte <br>
en format ID="pa"</span>
au milieu du paragraphe.
</P>
Fin du fichier css02.htm <br>
Retour au formattage
par défaut html.
</font>
</BODY></HTML>
```

Début du fichier css02.htm
Formatage "html" font size="+2".
Paragraphe formatage ID="pa" ... bla bla bla bla bla bla.
Paragraphe formatage de la classe P ... bla bla bla bla bla bla.
Paragraphe formatage de la classe P ... bla bla bla bla bla bla mais avec du texte en format ID="pa" au milieu du paragraphe.
Paragraphe formatage de la classe P.marge ... bla bla bla bla bla bla. Bla bla bla bla bla bla.
Paragraphe formatage inline avec du texte en format ID="pa" au milieu du paragraphe.
Fin du fichier css02.htm
Retour au formattage par défaut html.

Différentes façons d'introduire les CSS dans les pages html

Il y a quatre façons d'introduire les CSS dans les pages html :

- **inline** Styles : <P style="..."
- **Enbeded** Style Sheet (feuilles incorporées) : <style type="text/css">...</style
- **Linked** Style Sheet (feuilles liées, dans un fichier externe) :
  <link rel="stylesheet" type="text/css" href="monstyle.css" />
- **Imported** Style Sheet (feuilles de style importées) :

```
<style type="text/css">
   @import url(mon_serveur/mon_style.css);
</style>
```


Exemple de "Linked Style" CSS : css03.htm et css03.css

css03.htm

```
<HTML><HEAD><TITLE>css03.htm</TITLE>

<LINK HREF='css03.css' REL='stylesheet' TYPE='text/css'>

</HEAD>
<BODY BGCOLOR="#FFFFFF" TEXT="#000000"
 LINK="#008080" VLINK="#000080">
<font size="+2"> Formatage "html" font size="+2".<br>
<hr>
<P id="pa">Paragraphe formatage ID="pa" ...
bla bla bla bla bla bla.
</P>
<P>Paragraphe formatage de la classe P  ...
bla bla bla
</P>
<P class="marge">Paragraphe formatage de la classe
P.marge ... <br>
bla bla bla bla bla bla. Bla bla bla bla bla bla.
</P>
<hr>
Retour au formattage par défaut html.
</font>
<br>html : après fermeture &lt;/font>.
</BODY></HTML>
```


Exemple de "Linked Style" CSS : css03.htm et css03.css

css03.css

```
:STYLE TYPE="text/css">
:!-- /* Hide content from old browsers */

' { font-size: 10pt; font-family: Geneva, sans-serif;
    color: #000000; }
'pa { Background-Color: #00EEFF; color: #FF0000;
      font-size:12pt;}
'.marge { font-size: 11pt; margin-left:2.5cm;
  color: #0000FF; }

'* end hiding con
:/STYLE>
```

Formatage "html" font size="+2".
Paragraphe formatage ID="pa" ... bla bla bla bla bla bla.
Paragraphe formatage de la classe P ... bla bla bla
Paragraphe formatage de la classe P.marge ... bla bla bla bla bla bla. Bla bla bla bla bla bla.
Retour au formattage par défaut html.
html : après fermeture .

Exemple de style CSS plus complexe : css04.htm et css04.css

```
<HTML><HEAD><TITLE>css04.htm</TITLE>

<LINK HREF='css04.css' REL='stylesheet' TYPE='text/css'>

</HEAD>
<BODY BGCOLOR="#FFFFFF" TEXT="#000000"
 LINK="#008080" VLINK="#000080">
<hr>
<P>Paragraphe ... bla bla bla
 bla bla bla bla bla bla bla bla
</P>
<P id="P2">
Paragraphe ... bla bla bla bla bla bla bla bla bla bla bla
 bla bla bla bla bla bla bla bla bla bla bla bla bla
 bla bla bla bla bla bla bla bla bla bla bla bla bla
</P>

<P id="P3">
Paragraphe ... blu blu blu blu blu blu blu blu blu blu blu
 blu blu blu blu blu blu blu blu blu blu blu blu blu
 blu blu
 <SPAN id="HI">blo blo blo</SPAN>
 blu blu blu blu blu blu blu blu
</P>
<hr>
</BODY></HTML>
```

9

Exemple de style CSS plus complexe : css04.htm et css04.css

```
<STYLE TYPE="text/css">
<!-- /* Hide from old browsers - fichier css04.css - */

P { color: #000000;
font-size: 10pt; font-family: Geneva, sans-serif; }

#P2 { color: #0000FF;
position:absolute;top:50;left:80; width:110;
font-size: 12pt; font-family: Geneva, sans-serif; }

#P3 {color:green; color:rgb(0,255,0);
background-color:rgb(222,222,222);
position:absolute;top:100;left:120;
font-size:14pt;width:150;
font-family: Geneva, sans-serif; }

#HI {color:green;
background-color:rgb(255,0,0);
font-size:14pt; }

/* end hiding content from old browser
</STYLE>
```

10

Autre exemple de style CSS : css05.htm

```
<div id="retrait">
<hr>
<P>.....
<IMG src="p
<P>.....
<br clear
<hr>
</div>

#retrait {
text-align:justify;
margin-left:10%;
margin-right:10%;
background-color:yellow;
}
```

11

Autre xemple de style CSS : css05.htm

```
<P class="cadreg">
   NEdit is a GUI
   pro- vides mous
   editing style.
</p>

<P class="cadred">
   NEdit is a GUI
   provides mouse
   style.
</p>

<P style="background-color:lightgreen;">
</P>

.cadred{ float:right;
   font-weight:bold; text-align:left;
   width:300; background-color:yellow;
   padding:10; marging:10; border-color:black;
   border-width:2; border-style:solid;
}
```

12

Autre exemple de style CSS : css11.htm+css11.css

```
#retrait {
text-align:justify;
margin-left:10%;
margin-right:10%;
background-color:yellow;
}

<div id="retrait" class="cadreg">
<P>
Paragraphe ... bla bla bla
 bla bla bla bla bla bla
</P>
<IMG src="penguin.png" align="left">
<P style="background-color:lightgreen;">
          NEdit is a GUI style editor
</P>
<P class="cadred">
Paragraphe ... blo blo blo
 blo blo blo blo blo
</P>
</div>

.cadreg{ float:left;
  font-weight:bold; text-align:left;
  width:350; background-color:lightgreen;
  padding:10; marging:10;
  border-color:black; border-width:2; border-style:solid;
}
```

css11.htm
css11.css

13

Ch10 - Les CSS

Propriétés des CSS

Voir **http ://www.w3.org/TR/REC-CSS1** pour la liste complète et leur description (environ 90 pages).

- **background** : le fond de page (couleur, image, fixe, ...)
- **color** : la couleur du texte
- **font** : la police de caractères (famille, taille, style, forme, graisse)
- **list-style** : les types de listes
- **margin border padding** : ce qui encadre un élement pour faire une boîte (de l'extérieur vers l'intérieur)
- **propriétés du texte** : espacement horizontal et vertical, décorations, indentations, alignement, justification.

Ch10 - Les CSS

- **position des éléments** : en haut, en bas, à droite, à gauche, se recouvrant, absolue, visible, caché, ...

Unités de mesures utilisées par les CSS

- **absolues :** in, cm, mm, pt, pc (picas)
- **relatives :** %, em, en, ex, px (pixels)

- em, en, ex : relatif à la taille des lettres m, n, x dans la police courante ("current font"),
- px (pixels) dépend de la résolution de l'écran utilisé ;
- attention : l'indication % de la largeur peut donner des résultats bizarres dans les éléments imbriqués (table dans un table) ;

Ch10 - Les CSS

Propriétés des CSS : background and color

- background-color
- background-image , background-repeat, background-attachment, background-position

```
BODY {color: black; background: white }
BODY {  background: red url(pendant.gif);
        background-repeat: repeat-y;
        background-attachment: fixed; }
H1 { color: maroon }
EM { color: \#f00 }              /* \#rgb */
EM { color: \#ff0000 }           /* \#rrggbb */
EM { color: rgb(255,0,0) }      /* integer range 0 - 255 */
EM { color: rgb(100\%, 0\%, 0\%) } /* float range 0.0\% - 100.0\% */
```

Propriétés des CSS : color 1

La couleur peut donc être spécifiée paar un **nom** ou par une **un groupe de 3 valeurs RGB**.

Les valeurs sont codées sur un octet donc [000–255]. Ce qui donne $(2^8)^3 = 2^{24}$ ~= 16 millions.

Évidemment, on n'a pas défini 16 millions de noms ! !

On a oublié qui avait créé cette liste dans les premières versions du **"X Window System"** (appelé aussi **X11** ou **X**), au **MIT** en **1984** (voir http ://en.wikipedia.org/wiki/X_window_system et http ://en.wikipedia.org/wiki/X11_color_names).

La liste contient **140** noms de couleurs associées chacune à un triplet de valeurs dans le fichier **/usr/X11R6/lib/X11/rgb.txt** :

```
255 255 255     white        0   0 128      navy
  0   0   0     black        0   0 255      blue
```


Propriétés des CSS : color 2

Parmi ces couleurs, **16** correspondent au tristement célèbre écran "VGA" des PCs et sont reprises en HTML 3.2 et 4.01 :

```
black = "\#000000"        green = "\#008000"
silver = "\#C0C0C0"       lime = "\#00FF00"
gray = "\#808080"         olive = "\#808000"
white = "\#FFFFFF"        yellow = "\#FFFF00"
maroon = "\#800000"       navy = "\#000080"
red = "\#FF0000"          blue = "\#0000FF"
purple = "\#800080"       teal = "\#008080"
fuchsia = "\#FF00FF"      aqua = "\#00FFFF"
```

En plus des 16 couleurs ci-dessus et des 140 couleurs X11 (pas tout à fait 140 car certains noms ont les mêmes valeurs RGB), on a

dû définir un sous-ensemble des 16 millions de couleurs possibles pour s'accomoder des cartes graphiques d'entrée de gamme ne disposant que d'une petite mémoire donc incapables d'afficher simultanément 16 millions de couleurs.

D'où la définition du sous-ensemble dit**"browser safe palette"**.

Il s'agit des combinaisons de 6 graduations également réparties sur [000-255] pour chacune des 3 couleurs de base RVB :

```
En décimal :  000  051  102  153  204  255
En hexa :      00   33   66   99   CC   FF
```

Voir une table de couleur dans :
http ://www.w3.org/MarkUp/Guide/Style

Propriétés des CSS : font

➤ font-family , font-style , font-variant ,font-weight , font-size

```
BODY { font-family: gill, helvetica, sans-serif }
BODY { font-family: "new century schoolbook", serif }
H1, H2, H3 { font-style: italic }    normal | italic | oblique
H3 { font-variant: small-caps }      normal | small-caps
P { font-weight: normal }   /* 400 */
H1 { font-weight: 700 }     /* bold */
        normal | bold | bolder | lighter | 100 | 200 | 300 |
        400 | 500 | 600 | 700 | 800 | 900
P { font-size: 12pt; }   xx-small | x-small | small |
P { font-size: larger }  medium | large | x-large | xx-large
EM { font-size: 150\% }  xx-large | larger | smaller
P { font: bold italic large Palatino, serif }  "font" est un
raccourci pour donner tous les choix en une propriété
```


Propriétés des CSS : Text properties

word-spacing et letter-spacing : indique une longueur qui **s'additionne** à l'espacement par défaut résultat de la prise en compte des autres propriétés du texte (police, justification,...).
Ex : H1 { word-spacing : 1em ; letter-spacing : 0.1em}

Autres :

```
text-decoration: underline || overline || line-through || blink
vertical-align:  sub | super | top | text-top | middle | bottom |
                 text-bottom | <percentage>
text-transform:  capitalize | uppercase | lowercase | none
text-align:      left | right | center | justify
text-indent:     <length> | <percentage>
line-height:     normal | <number> | <length> | <percentage>
                 si <number> : facteur x font-size
```


Propriétés des CSS : Box properties

Il y a trois coquilles entourant chaque élément :

```
margin---            Pour chaque coquille xxx il y a quatre
 | border---         propriétés :
 |  | padding---     xxx-top xxx-right xxx-bottom xxx-left
 |  |  | element    Plus une "xxx" donnant la même valeur
                     à toutes les xxx-yyy
Les valeurs peuvent être: <length> | <percentage> | auto
```

border est plus complexe et se décline en plusieurs propriétés : border-color, border-style et border-width (avec les dérivés border-top-width,...).

La propriété **"border"** est un raccourci permettant de donner la même valeur sur tous les côtés.

Les propriétés **'width'** et **'height'** spécifient la taille de l'élément.

'float' et **'clear'** indiquent la position de l'élément dans le flux et si du texte peut "couler" autour de l'élément.

Propriétés des CSS : list-style

'list-style-type' : disc | circle | square | decimal | lower-roman | upper-roman | lower-alpha | upper-alpha | none

'list-style-image' : <url> | none

'list-style-position' : inside | outside (marqueur de liste dans le texte ou à l'extérieur ; défaut : outside).

'list-style : raccourci pour spécifier les 3 précédenta en une commande.

Ch10 - Les CSS.

Les questions sur ce chapitre sont reportées à la fin du ch.11.

Fin du chapitre.

Ch11 - JavaScript

- Le langage Javascript
- JavaScript, CSS et les menus

©Michel.Vayssade@UTC.fr – Université de Technologie de Compiègne,UV SR03 2006 - Architectures Internet.

1

js01.htm

```
:html>
:head>
:/head>
'eci est du html.<br>
  <script language="JavaScript">
:!--  hide script from old browsers
     document.write("Ceci est écrit par \
                     JavaScript!<br><hr>")
'/ end hiding contents -->
  </script>
:br>
.a suite est du html.<br>
:/body>
:/html>
```

Ceci est du html.
Ceci est écrit par JavaScript!
La suite est du html.

2

js02.htm

```
<html>
<head><!-- fichier js02.htm -->
  <script language="JavaScript">
     function pushbutton() {
       alert("Hello!");
  }
 </script>
</head>
<body>
<form>
  <input type="button" name="Button1"
  value="Push me" onclick="pushbutton()">
</form>
</body>
</html>
```

Push me

[JavaScript Application] Hello! OK

3

js02.htm → js021.htm + js022.js

```
!avigateurs modernes: possible stocker code javascript
     (>=V4)                    dans des fichiers séparés
<html>
<head>
  <script language="JavaScript" src="./js022.js">
  </script>
</head>
<body>
<form>
  <input type="button" name="Button1"
  value="Push me" onclick="pushbutton()">
 </form>
</body>
</html>
```

```
function pushbutton() {
       alert("Hello!");
}
```

4

js03.htm

```
<html>
<head><!-- js03.htm -->
<script language="JavaScript">
<!--  hide script from old browsers
  function getname(str) {
    document.write("ici dans getname<br>");
    document.write(document.URL);
    alert("Hi, "+ str+"!");
    location.replace(document.URL);
  }
// end hiding contents -->
</script>
</head>
<body>
Please enter your name:
<form>
  <input type="text"
   name="nom" value=""
   onBlur=
   "getname(this.value)">
</form>
</body>
</html>
```

Please enter your name:
aaaa (1) clic (2) aaaa (3) clic en dehors

ici dans getname
file:///home/vayssade/srt/ch11-jscript/js03.htm?nom=aaaa

[JavaScript Application] Hi, aaaa! OK

5

js04.htm

```
:html>
:body>
'his is a simple HTML- page.
:br>
.ast changes:

  <script language="JavaScript">
  <!--  hide script from old browsers

    document.write(document.lastModified)

  // end hiding contents -->
  </script>

:/body>
:/html>
```

This is a simple HTML- page.
Last changes: Thu 24 Mar 2005 10:19:05 AM CET

6

js05a.htm

```
<html> <head><!-- fichier js05a.htm -->
<script language="JavaScript">
     function info1() {
       alert("mot en gras = tag <b>");
     }
     function info2() {
       alert("infos : explications");
     }
</script></head>
<body>
Cliquer sur les mots
<b onClick="info1()"><u>en gras soulignés</u></b> ou
passer la souris sur les mots en gras pour un complément
<b onMouseOver="info2()">d'infos</b>.<br>
</body>
</html>
```

Cliquer sur les mots **en gras soulignés** ou passer la souris sur les mots en gras pour un complément **d'infos**.

[JavaScript Application] infos : explications OK

7

js05.htm

```
html><head>
script language="JavaScript">
     function info1() {
       choix = confirm("changer de page vraiment ?");
       if (choix) {
         parent.location.href="js051.htm"
       } else {
         parent.location.href="js05.htm"
       }
     }
/script>
/head>
body>
'age <b>js05.htm</b><br>
'liquer sur le bouton
our changer de page.<br>
form>
<INPUT TYPE="button"
 VALUE="aller sur js051.htm"
 onClick="info1()">
/form>
/body></html>
```

Page **js05.htm**
Cliquer sur le bouton pour changer de page.
aller sur js051.htm

[JavaScript Application] changer de page vraiment ? OK Cancel

8

js051.htm

```
<html> <head>
<script language="JavaScript">
     function info1() {
       choix = confirm("changer de page vraiment ?");
       if (choix) {
         parent.location.href="js05.htm"
       } else {
         parent.location.href="js051.htm"
       }
     }
</script>
</head>
<body>
Page <b>js051.htm</b><br>
Cliquer sur le bouton pour changer de page.<br>
<form>
 <INPUT TYPE="button"
  VALUE="aller sur js05.htm"
  onClick="info1()">
</form>
</body></html>
```

Home Bookmarks Red Hat, Inc. Red Hat Network Supp

Page **js051.htm**

Cliquer sur le bouton pour changer de page.

aller sur js05.htm

9

js06.htm

```
:script language="JavaScript">
 today = new Date()
 document.write("L'heure est : ",today.getHours(),":",
                                 today.getMinutes())
 document.write("<br>")
 document.write("La date est : ", today.getDate(),"/")
 document.write(today.getMonth()+1,"/",today.getYear())
 document.write("<br>getFullYear()=",today.getFullYear())
/script>
```

Home Bookmarks Red Hat, Inc.

L'heure est : 14:20
La date est : 24/3/105
getFullYear()=2005

10

js07.htm

```
<html>
<head><!-- fichier js07.htm -->
<script language="JavaScript">

function RandomNumber() {
  today = new Date();
  num= Math.abs(Math.sin(today.getTime()));
  return num;
}
</script>
</head>

<body>
<script language="JavaScript">
   document.write("Un nombre aléatoire : ",
                  RandomNumber());
</script>
</body>
</html>
```

Home Bookmarks Red Hat, Inc. Red Hat Network Suppo

Un nombre aléatoire : 0.9995703526291909

11

js08.htm

```
<html><head><!-- fichier js08.htm -->
<script language="JavaScript">
function WinOpen() {
   msg=open("frtest1.htm","DisplayWindow",
    "toolbar=no,directories=no,menubar=yes");
   msg.document.write(
   "<HTML><HEAD><TITLE>Yo!</TITLE></HEAD>");
   msg.document.write("<BODY>");
   msg.document.write(
   "<CENTER><h2><B>This is cool!</B></h2></CENTER>");
   msg.document.write("<br><br>Fin.</BODY></HTML>");
   msg.document.close();
}
</script></head>
<body>
Ouvrir une nouvelle fenêtre du navigateur :
<form><input type="button" name="Button1"
       value="Open new window" onclick="WinOpen()">
</form>
<p>msg=open("frtest1.htm", "DisplayWindow", "toolbar=no,
directories=no,menubar=yes");</p>
<p>On peut associer à la nouvelle fenêtre les propriétés
suivantes :</p>
<ul><li>toolbar, menubar
    <li>location, directories, status, copyhistory
    <li>scrollbars, resizable, width=pixels, height=pixels
</ul>
</body></html>
```

12

js08.htm

```
function WinOpen() {
   msg=open("frtest1.htm","DisplayWindow",
    "toolbar=no,directories=no,menubar=yes");
   msg.document.write(
   "<HTML><HEAD><TITLE>Yo!</TITLE></HEAD>");
   msg.document.write("<BODY>");
   msg.document.write(
   "<CENTER><h2><B>This is cool!</B></h2></CENTER>");
   msg.document.write("<br><br>Fin.</BODY></HTML>");
   msg.document.close();
}
```

Yo! - Mozilla

File Edit View Go Bookmarks Tools Window Help

This is cool!

Fin.

13

js09.htm

```
<HTML>
<HEAD><!-- fichier js09.htm -->
<title>Frames</title>
</HEAD>
  <FRAMESET ROWS="50%,50%">
      <FRAME SRC="frtest1.htm" name="fr1">
      <FRAME SRC="frtest2.htm" name="fr2">
  </FRAMESET>
</HTML>
```

50%

50%

Home Bookmarks Red Hat, Inc. Red Hat Network Sup

This is our first frame! frtest1.htm

This is our second frame! frtest2.htm

Done

14

js091.htm

```
<HTML> <HEAD><!-- js091.htm -->
<title>Frames</title>
</HEAD>
<FRAMESET COLS="50%,50%">
  <FRAMESET ROWS="50%,50%">
    <FRAME SRC="frtest1.htm">
    <FRAME SRC="frtest2.htm">
  </FRAMESET>
  <FRAMESET ROWS="33%,33%,33%">
    <FRAME SRC="cell.htm">
    <FRAME SRC="cell.htm">
    <FRAME SRC="cell.htm">
  </FRAMESET>
</FRAMESET>
</HTML>
```

Home Bookmarks Red Hat, Inc. Red Hat Network »

This is our first frame! frtest1.htm

This is our second frame! frtest2.htm

This is cell.htm 33%

This is cell.htm 33%

This is cell.htm 33%

Done

50% 50%

15

js10.htm

```
<HTML><HEAD><!-- js10.htm -->
<title>Frames</title>
</HEAD>
  <FRAMESET ROWS="50%,50%">
    <FRAME SRC="js101.htm" name="fr1" noresize>
    <FRAME SRC="js102.htm" name="fr2">
  </FRAMESET>
</HTML>

<HTML><!-- js101.htm --><HEAD>
<script language="JavaScript">
<!-- Hiding
   function hi() {
     document.write("Hi!<br>"); }
   function yo() {
     document.write("Yo!<br>");
     document.close();          }
   function bla() {
     document.write("bla bla<br>"); }
// -->
</script></HEAD><BODY>
Ceci est notre frame 1.
</BODY></HTML>

<HTML><!-- js102.htm --><body>
<p>Ceci est notre "frame" 2.</p>
<FORM NAME="buttonbar">
  <INPUT TYPE="button" VALUE="Hi" onClick="parent.fr1.hi()">
  <INPUT TYPE="button" VALUE="Yo" onClick="parent.fr1.yo()">
  <INPUT TYPE="button" VALUE="Bla" onCLick="parent.fr1.bla()">
</FORM></BODY></HTML>
```

Home Bookmarks Red Hat, Inc.

Ceci est notre frame 1.

Ceci est notre "frame" 2.

Hi Yo Bla

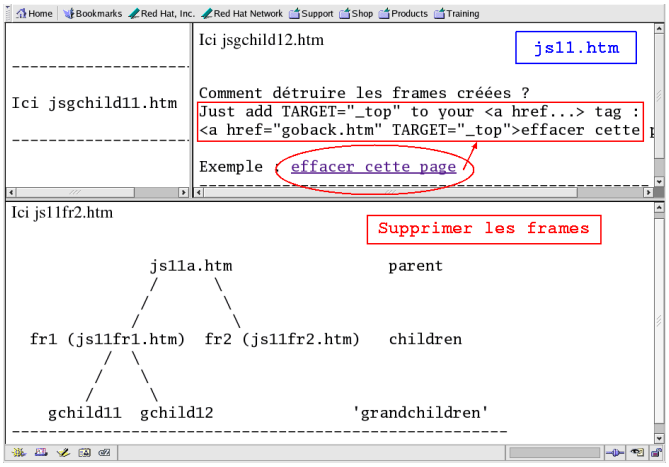
Done

Home Bookmarks

Yo!

marche 1 fois ! Pourquoi ? -> view frame source

16



17

Action sur le navigateur

js12.htm

```
html>
head><!-- js12.htm -->
script language="JavaScript">
unction statbar(txt) {
  window.status = txt;

/script>
/head>

body>
ci js12.htm : démo écriture
ans barre d'état (statusbar).<br>

form>
input type="button" name="look" value="Write!"
onclick="statbar('Hi! This is the statusbar!');">

input type="button" name="erase"
alue="Erase!" onclick="statbar('');">

/form>
/body></html>
```

js12.htm : démo écriture dans barre d'état (statusbar).

Write! Erase!

Hi! This is the statusbar!

18

Action sur le navigateur

js13.htm

```
<html>
<head><!-- js13.htm -->

<script language="JavaScript">
function statbar(txt) {
   window.status = txt;
   setTimeout("erase()",2000);
}
function erase() {
   window.status="";
}
</script>
</head>
<body>

<a href="js13.htm"
 onMouseOver="statbar('Disappearing in 2 sec.!');
 return true;">
link</a>

</body></html>
```

link

Disappearing in 2 sec.!

19

js14.htm

Action sur le navigateur

```
<html><head><!-- js14.htm -->
<script language="JavaScript">

var scrtxt="Ici un message "+
" et la suite du message ...";
var lentxt=scrtxt.length;
var width=100;
var pos=1-width;

function scroll() {
  pos++;
  var scroller="";
  if (pos==lentxt) { pos=1-width;}
  if (pos<0) {
    for (var i=1; i<=Math.abs(pos); i++) {
          scroller=scroller+" ";}
    scroller=scroller+scrtxt.substring(0,width-i+1); }
  else{
    scroller=scroller+scrtxt.substring(pos,width+pos); }
  window.status = scroller;
  setTimeout("scroll()",150);
  }
</script></head>
<body onLoad="scroll(); return true;">
Ici la page js14.htm.<br>
Un texte défile dans la barre de status.
</body></html>
```

Ici la page js14.htm.
Un texte défile dans la barre de status.

Ici un message et la suite du message ...

20

js15.htm

```
<html><head><!-- fichier js15.htm -->
<script language="JavaScript">
<!-- Hide
function writedoc() {
var aa = "abc def";
var mytab= new Array(3)
mytab[0]=123
mytab[1]="yes/no"
mytab[2]=102030
   document.write("<P>");
   document.write("éléments du tableau :"+mytab.length+"<br>");
   document.write("élément 0: "+mytab[0]+"<br>");
   document.write("élément 1: "+mytab[1]+"<br>");
   document.write("élément 2: "+mytab[2]+"<br>");
   document.write("élément 3: "+mytab[3]+"<br>");
}
// -->
</script>
</head>
<body>
<FORM NAME="buttonbar">
 <INPUT TYPE="button" VALUE="Back" onClick="history.back()">
 <INPUT TYPE="button" VALUE="Home" onClick="location='js01.htm'">
 <INPUT TYPE="button" VALUE="Next" onClick="history.forward()">
<br>
<INPUT TYPE="button" VALUE="write document" onClick="writedoc()">
</FORM>
<p>On peut aussi utiliser : history.go(-1) et
history.go(1) ...</p>
</body></html>
```

Back Home Next
write document
On peut aussi utiliser : history.go(-1) et history.go(1) ..

éléments du tableau :3
élément 0: 123
élément 1: yes/no
élément 2: 102030
élément 3: undefined

21

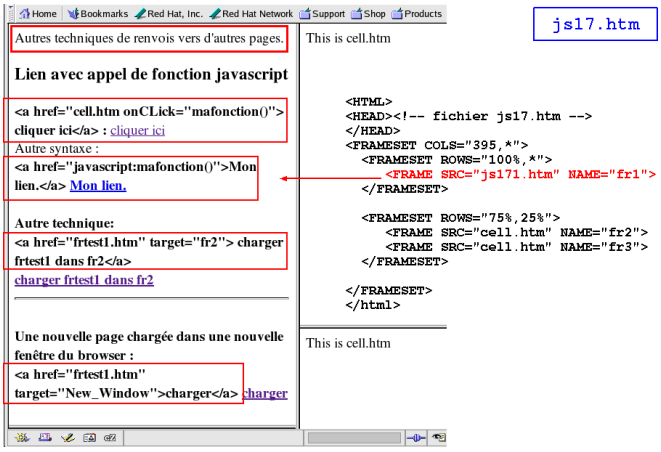
js16.htm

```
<HTML><HEAD><!-- fichier js16.htm -->
<title>Frames</title>
</HEAD>
<FRAMESET COLS="105,*">
  <FRAMESET ROWS="100%,*">
     <FRAME SRC="load2.htm" NAME="fr1">
  </FRAMESET>

  <FRAMESET ROWS="75%,25%">
     <FRAME SRC="cell.htm" NAME="fr2">
     <FRAME SRC="cell.htm" NAME="fr3">
  </FRAMESET>

</FRAMESET>
</HTML>

 <HTML><HEAD><!-- load2.htm -->
 <script language="JavaScript">
   function loadtwo(page2, page3) {
     parent.fr2.location.href=page2;
     parent.fr3.location.href=page3;
 }
 </script>
 </HEAD><BODY>
 <FORM NAME="buttons">
      <INPUT TYPE="button" VALUE="2 in a row"
           onClick="loadtwo('frtest1.htm','frtest2.htm')">
 </FORM>
 </BODY></HTML>
```

2 in a row
This is cell.htm
This is cell.htm
Done

2 in a row
This is our first frame! frtest1.htm
This is our second frame! frtest2.htm
Done

22



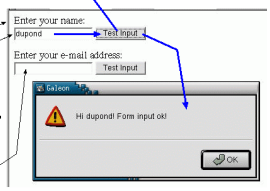
23

```
<HTML><HEAD><!-- fichier js171.htm -->
<script language="JavaScript">
  function mafonction () {
     parent.fr2.location.href="frtest1.htm";
     parent.fr3.location.href="frtest2.htm";  }
</script>
</HEAD><BODY>
Autres techniques de renvois vers d'autres pages.<br>

<h3>Lien avec appel de fonction javascript</h3>
<b>&lt;a href="cell.htm onCLick="mafonction()">
cliquer ici&lt;/a> : </b>
<a href="cell.htm" onCLick="mafonction()">cliquer ici</a>
<br>
Autre syntaxe : <br>
<b>&lt;a href="javascript:mafonction()">Mon lien.&lt;/a>
<a href="javascript:mafonction()">Mon lien.</a>
<br><br>
Autre technique: <br>
&lt;a href="frtest1.htm" target="fr2">
charger frtest1 dans fr2&lt;/a><br>
<a href="frtest1.htm" target="fr2">charger frtest1 dans fr2</a>
<br><hr><br>
Une nouvelle page chargée dans une nouvelle fenêtre
du browser :<br>
&lt;a href="frtest1.htm" target="New_Window">charger&lt;/a>
<a href="frtest1.htm" target="New_Window">charger</a>
</BODY></HTML>
```

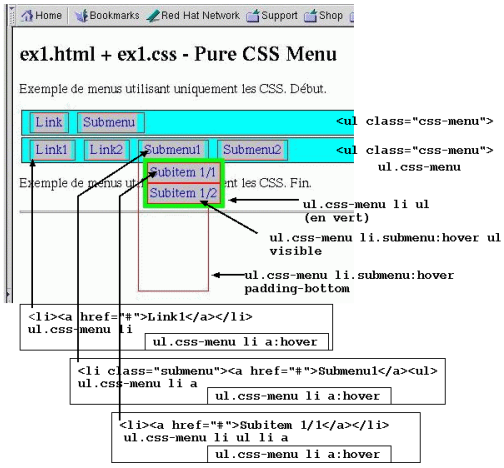
24

js19.htm

```
<html>
<head><!-- fichier js19.htm -->
<script language="JavaScript">
function test1(form) {
  if (form.text1.value == "")
    alert("Please enter a string!")
  else {
   alert("Hi "+form.text1.value+"! Form input ok!"); }
}
function test2(form) {
  if (form.text2.value == "" ||
      form.text2.value.indexOf('@', 0) == -1)
        alert("No valid e-mail address!");
  else alert("OK!");
}
</script>
</head><body>
<form name="first">

Enter your name:<br>
<input type="text" name="text1">
<input type="button" name="button1"
       value="Test Input"
       onClick="test1(this.form)">
<P>Enter your e-mail address:<br>
<input type="text" name="text2">
<input type="button" name="button2"
       value="Test Input" onClick="test2(this.form)">
</form></body></html>
```



25



26

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html><head><title>Ex1 - Pure CSS Menu</title>
<meta http-equiv="Content-Type" content="text/html;
  charset=ISO-8859-1">
<link rel="stylesheet" type="text/css" href="ex1.css">
</head><body>
  <h2>ex1.html + ex1.css - Pure CSS Menu</h2>
 <p>Exemple de menus utilisant uniquement les CSS. Début.</p>
  <div id="menubar">
    <ul class="css-menu">
      <li><a href="#">Link</a></li>
      <li class="submenu">
        <a href="#">Submenu</a>
        <ul>
          <li><a href="#">Subitem 1</a></li>
          <li><a href="#">Subitem 2</a></li>
        </ul>
      </li>
    </ul>
  </div>
    <ul class="css-menu">
      <li><a href="#">Link1</a></li>
      <li><a href="#">Link2</a></li>
      <li class="submenu"><a href="#">Submenu1</a><ul>
          <li><a href="#">Subitem 1/1</a></li>
          <li><a href="#">Subitem 1/2</a></li>
      </ul></li>
      <li class="submenu"><a href="#">Submenu2</a><ul>
          <li><a href="#">Subitem 2/1</a></li>
          <li><a href="#">Subitem 2/2</a></li>
          <li><a href="#">Subitem 2/3</a></li>
      </ul></li>
    </ul>
 <p>Exemple de menus utilisant uniquement les CSS. Fin.</p>
<hr></body></html>
```

27

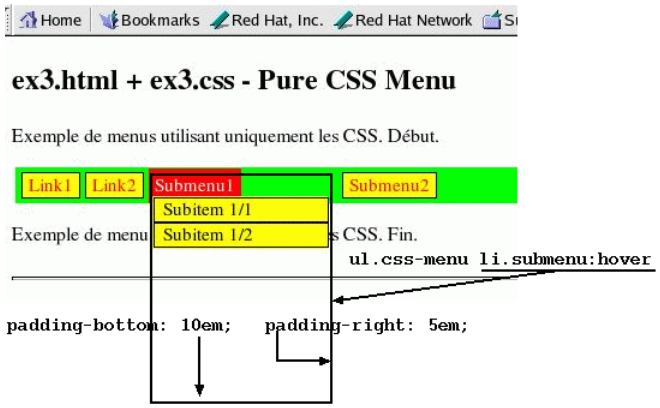
```
/* la barre de menu */                          /*  ex1.css  */
ul.css-menu
{ display: block;  background-color: aqua;
  /* margin: espace entre 2 barres;
     padding : espace autour des objets contenus */
  margin: 2px;  padding: 6px;
  /* cadre noir autour de la barre */
  border-style: solid; border-width: 1px; color: black;
}
/* une entrée dans la barre de menus */
ul.css-menu li /* les "li" sous "ul.css-menu" = top level */
{ /* afficher en ligne (horizontale) les top-levels */
  display: inline;
  /* list-style: none;  inutile en inline */
  position: relative;
  margin: 3px;  padding: 3px;
  /* cadre rouge autour de chaque entrée de la barre */
  border-style: solid; border-width: 1px; color: red;
}
/* la partie anchor d'une entrée de la barre */
ul.css-menu li a
{ margin: 1px;  padding: 3px;  background-color: silver;
  /* texte des entrées de la barre (dans les "a") */
  text-decoration: none;
}
/* changer apparence entrée top niv sélectionnée */
ul.css-menu li a:hover
{ color: #FFF;    /* selection: texte blanc sur fond rouge */
  background-color: red;  /* fond élément menu sélectionné */
}
```

28

```
/* sous-menu qui s'ouvre (en vert) sous une entrée toplevel */
ul.css-menu li ul  /* "ul" sous un "li" = sous-menu */
{ display: block; position: absolute; left: 0;
  top: 1em; /*required for NS7.x/Gecko 1.0.x but not Gecko 1.3*/
  visibility: hidden;  /* sous-menu cachés par défaut */
  margin: 5px;  padding: 5px;  background-color: lime;
  /* width: 9em; */
  z-index: 1000;
}
/* fix up the submenu itemss voice-family lines screen correct
   CSS values from browsers that improperly lay out block-level
   boxes and have broken CSS parsers (IE5.5/Win)
*/
ul.css-menu li ul li
{ margin: 0 0 0 -1.5em;
  padding: 0; display: block;
  width: 100%;  margin-left: 0;
  voice-family: "\"}\"";
  voice-family: inherit;
}
/* entrée dans un sous-menu */
ul.css-menu li ul li a
{ display: block; margin: 0;
/*   padding: 0 0 0 5%;
  width: 95%; */
}
```

```
/* pour sous-menu ouvert,
   zone active assez haute,
   sinon reste pas ouvert */
ul.css-menu li.submenu:hover
{ padding-bottom: 10em;  }

/* quand entrée barre est
   activée, rendre visible les
   "ul" en-dessous */
ul.css-menu li.submenu:hover ul
{ left:0; visibility: visible;}
```

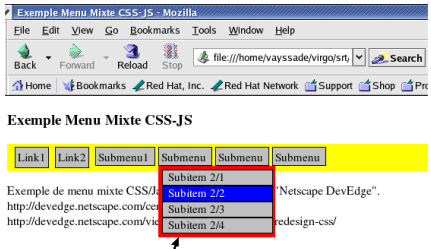
29



30

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html><head><title>Ex3 - CSS Menu</title>
<meta http-equiv="Content-Type" content="text/html;
 charset=ISO-8859-1">
<link rel="stylesheet" type="text/css" href="ex3.css">
</head><body>
  <h2>ex3.html + ex3.css - Pure CSS Menu</h2>
 <p>Exemple de menus utilisant uniquement les CSS. Début.</p>

  <div id="menubar">
    <ul class="css-menu">
      <li><a href="#">Link1</a></li>
      <li><a href="#">Link2</a></li>
      <li class="submenu"><a href="#">Submenu1</a><ul>
          <li><a href="#">Subitem 1/1</a></li>
          <li><a href="#">Subitem 1/2</a></li>
      </ul></li>
      <li class="submenu"><a href="#">Submenu2</a><ul>
          <li><a href="#">Subitem 2/1</a></li>
          <li><a href="#">Subitem 2/2</a></li>
          <li><a href="#">Subitem 2/3</a></li>
      </ul></li>
    </ul>
  </div>
 <p>Exemple de menus utilisant uniquement les CSS. Fin. </p>
<hr></body></html>
```

31



exmix.html

cssjsmenu.css

cssjsmenu.js

Forme css appelée par l'évènement Javascript "onmouseover" et placée en position absolue calculée par javascript juste sous l'entrée correspondante.

32

```
<html><head><title>Exemple Menu Mixte CSS-JS</title>
...
<link rel="stylesheet" type="text/css" href="cssjsmenu.css">
<script type="text/javascript" src="cssjsmenu.js"></script>
<script type="text/javascript">
  <!--
  function init()
  { cssjsmenu('navbar');  }
  // -->
</script></head>
<body onload="init()">
  <h3>Exemple Menu Mixte CSS-JS</h3>
  <div id="navbar">
    <ul class="nde-menu-system">
      <li><a href="#">Link1</a></li>
      <li>....... un élément <li> pour chaque entrée
          ....... de la barre de menus
      <a href="#">Submenu1</a><ul>
          <li><a href="#">Subitem 1/1</a></li>
          <li><a href="#">Subitem 1/2</a></li></ul>
      </li>
      <li>..........
    </ul>
  </div>
  <p>Exemple de menu mixte CSS/Javascript, inspiré du site
  "Netscape DevEdge".....
  </p></body></html>
```

exmix.html

33

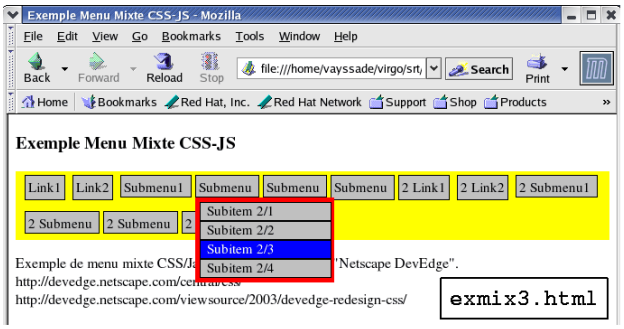
exmix.html

cssjsmenu.css

cssjsmenu.js

```
a fonction cssjsmenu explore l'arbre constitué
ar les entées <UL>, <LI> dans <div id="navbar">

lle construit des tableaux internes qui pour chaque
lément <UL> ou <LI> enregistre la liste des éléments
en-dessous", afin de les rendre
oit visibles: child.style.visibility = 'visible';
              (dans parent.onmouseover)
oit cachés :  child.style.visibility = 'hidden';
              (dans parent.onmouseout)

ci se sont les évènements javascript (mouseover,
ouseout)qui déclenchent le changement de status
l'éléments de la page (visible/hidden).
eur apparence dans le cas visile étant fixée par
a feuille CSS associée.
```

34



exmix3.html

Les mêmes programmes (css, js) que l'exemple précédent.Mais ici la barre de menus "déborde" car le nombre d'éléments qu'elle contient la rend plus large que la page : il y a passage à la ligne et la barre de menus apparait sur deux lignes.

35

Changer un élément HTML dynamiquement

Un élément HTML contenu dans une balise élément peut être remplacé dynamiquement par JavaScript :

```
<script language="JavaScript">
document.getElementById('cible').innerHTML="autre chose";
</script>
ou
document.getElementById('cible').innerHTML=
"<img src=number0.gif></img>";
```

D'autres balises que <DIV> peuvent être utilisées, par exemple une cellule d'un tableau à qui on a donné un identifiant ID="etiquette" peut avoir son "innerHTML" remplacé.

Charger une page HTML dynamiquement dans une IFRAME

Une IFRAME (Inline Frame) est un cadre **dans lequel** ou peut faire charger dynamiquement (par une ligne de JavaScript) une page html complète.

Cette fonction n'est disponible que sur les navigateurs récents.

L'inclusion d'une IFrame se fait par :

<iframe name="IFrameName"></iframe>

Les paramètres possibles d'une IFrame sont :

```
frameborder  yes|no      height  nombre|pourcentage
scrolling    yes|no      width   nombre|pourcentage
```

```
Changée par :
frames'IFrameName'.location.href='http://www.utc.fr/unepage.html'
```



```
ou
document.all.IFrameName.src='./unepage.html'

Autres paramètres :
marginwidth="x"   marginheight="x"
hspace="x"    vspace="x" : espaces blanc autour de l'iframe
```

Ces lignes de JavaScript peuvent être mise en paramètres d'un appel onClick="" dans un bouton par exemple.

Charger une page dans une IFRAME dans un élément <DIV>

La méthode <DIV id=cible> et IFRAME peuvent être combinées :

```
document.getElementById('cible').innerHTML =
"<iframe src='http://www...unepage.html";
```

Si on a plusieurs IFRAME dans une page, on peut leur donner des noms (<iframe src="page.html" name="nom"></iframe>) et utiliser un attribut target="nom" dans un lien pour envoyer la cible de ce lien hypertexte dans la IFrame "nom".

Le document dans la iframe peut accéder le document contenant le iframe par le mot clé **parent**.

Les évènements intrinsèques

```
onload  onunload  onclick  ondblclick  onmousedown  onmouseup
onmouseover  onmousemove  onmouseout  onfocus  onblur
onkeypress  onkeydown  onkeyup  onsubmit  onreset
onselect  onchange
```

Plus d'infos dans :

http ://www.w3.org/TR/REC-html40/interact/scripts.html

Exemple : rollovers avec 'onMouseover' et 'onMouseout'

```
onmouseout="this.style.backgroundColor=''"
onmouseover="this.style.backgroundColor='burlywood'">
onmouseout="this.style.fontWeight='normal'"
onmouseover="this.style.fontWeight='bold'"
```

Ch11 - JavaScript

Fin du chapitre.

Questions sur ch09-http, ch10-css, ch11-jscript

➤ Indiquez (par un schéma) le protocole de base du Web et le concept de liaison entre documents.
Réponse : ch09, diapos 2 et 3.

➤ Quel est l'organisme chargé de la normalisation du Web ?
Réponse : ch09, diapo 5.

➤ Décrivez les éléments principaux d'un URL.
Réponse : ch09, diapo 7.

➤ Quelle est la forme d'une requête envoyée par un navigateur à un serveur web et quelle est la forme de la réponse ?
Réponse : ch09, diapo 10. Une suite de lignes de texte.

➤ Qui provoque le chargement des images insérées dans un page web ?
Réponse : ch09, diapo 13. Le navigateur analyse la page recue et

refait une requête pour chaque référence à une image trouvée dans la page.

➤ Quelle est la particularité du protocole HTTP ?
Réponse : ch09, diapo 16. Utiliser TCP et être "sans état".

➤ Décrire le format d'une requête "GET"+"URL encodé".
Réponse : ch09, diapo 18.

➤ Qu'est-ce que HTML ? Réponse : ch09, diapos 23 à 26.

➤ Quel est le mécanisme par lequel des données saisies dans un navigateur sont passées à un programme s'exécuant sur le serveur web ?
Réponse : ch09, diapos 31 et 32. <FORM ACTION=programme> ...

➤ Quel est le principe des CSS ?

Réponse : ch10, diapos 3 à 4. Pour chaque type d'élément d'une page html définir la façon dont il est "dessiné" dans la page.

➤ Quelle sont les différentes façons d'introduire les CSS dans les pages html ?
Réponse : ch10, diapo 6 (et suivantes).

➤ Comment sont spécifiées les couleurs des attributs CSS ?
Réponse : ch10, diapos 16 à 18

➤ Quelle sont les trois propriétés qui encadrent un élément CSS ?
Réponse : ch10, diapo 22.

➤ Qu'est-ce que JavaScript ? Par qui est-il exécuté ?
Réponse : ch11, diapo 2 et 3. Un langage interprété par le navigateur.

➤ Comment JavaScript interagit-il avec l'utilisateur ?
Réponse : ch11. Par gestion évènements onClick, onMouseOver, CSS :hover, ...

(diapo vide)

(diapo vide)

(diapo vide)

Ch12 - PHP

- Le langage PHP
- PHP et les sessions

PHP : un langage de script côté serveur. Objectifs :

- permettre au concepteur de site web **d'exécuter un traitement** avant d'envoyer une page au navigateur
- instructions PHP sont exécutées **par le serveur web**
- permettre de construire des pages **créées dynamiquement**

Conditions de fonctionnement de PHP :

- code PHP ne fonctionne que si PHP à été installé sur le serveur (ex. mod_php de Apache)
- source HTML envoyé au navigateur par le serveur ne contient **aucune trace** du code PHP
- code PHP est placé directement dans le code source HTML, encadré par deux balises **<?php et ?>**
- Un fichier source php peut contenir une **alternance** de parties "html" et de parties "php".
- **Syntaxe** de PHP très proche du **C**.

Premier script : p01.php3

p01.php3

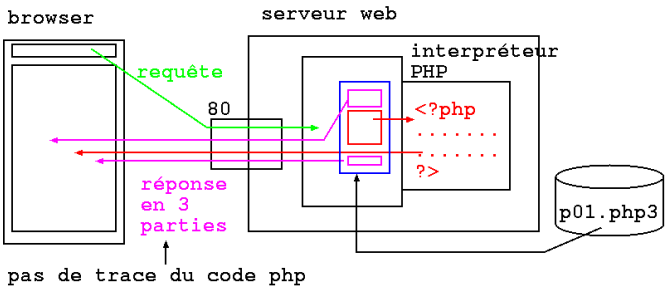
```
<HTML><HEAD>
<TITLE>fichier p01.php3</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<font size="+1"> Début du fichier php01.htm <br>
Ici code html.<br>
<hr>
<?php
 echo "<P>Hello World ! écrit par du code PHP.";
 echo "</P>";
?>
Retour au code html.<br>
</BODY></HTML>
```

Début du fichier p01.php3
Ici code html.
<?php
Hello World ! écrit par du code PHP.
?>
Retour au code html.

Netscape: Source of: http://www4.utc.fr/~lo33/php/p01.php3

```
<HTML><HEAD><TITLE>fichier p01.php3</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<font size="+1"> Début du fichier p01.php3 <br>
Ici code html.<br>
<hr>
<P>Hello World ! écrit par du code PHP.</P>Retour au code html.<br>
</BODY></HTML>
```


Fonctionnement de PHP



Types de données PHP : p02.php3

PHP possède trois types de données : les entiers (integer), les réels (double) et les chaines de caractères (string).

C'est un langage **faiblement typé** : on peut mettre la chaine "123" dans une variable, lui **ajouter** 4, on obtient, **suivant le contexte** soit le nombre 127, soit la chaine "127" (voir $foo += 3 ; et $faa dans l'exemple p02.php3).


```
<HTML><HEAD>
<TITLE>fichier p02.php3</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF">
<font size="+1"> Début du fichier p02.php3 <br>
Démo des types de données PHP.<br><hr>
<?php
 echo "<P>écrit par PHP :<br>";

$foo = "123";  // $foo is string (ASCII 49,50,51)
 print ("<br>foo= $foo\n");
$foo++;       // $foo is the string "124"
 print ("<br>foo++= $foo\n");
$foo += 3;    // $foo is now an integer (127)
 print ("<br>foo + 1= $foo\n");
$faa = "abc $foo def";
 print ("<br>faa = $faa\n");
$foo = $foo + 1.3;  // $foo is now double (128.3)
 print ("<br>foo + 1.3= $foo\n");
$foo = 5 + "10 Little Piggies";
 print ("<br>foo= $foo\n");
$foo = 5 + "10 Small Pigs";
 print ("<br>foo= $foo\n");

 echo "</P>";
?>
Retour au code html.<br>
</BODY></HTML>
```

langage faiblement typé

écrit par PHP :
foo= 123
foo++= 124
foo + 1= 127
faa = abc 127 def
foo + 1.3= 128.3
foo= 15
foo= 15

PHP : les opérateurs

Opérateurs arithmétiques

$a + $b	Addition
$a - $b	Soustraction
$a * $b	Multiplication
$a / $b	Division
$a % $b	Modulo

Opérateurs sur les chaines : concaténation

$a = "Hello " ;
$b = $a . "World !" ; (avec opérateur "." et non "+")
// maintenant $b contient "Hello World !"

++> collection de fonctions sur les chaines, et support des expressions régulières.

Opérateurs binaires "bit à bit"

$a & $b	ET	bits à 1 dans a ET b mis à 1
$a | $b	OU	bits à 1 dans a OU b mis à 1
~$a	NON	bits inversés
<<	décalage	bits décalés vers la gauche
>>	décalage	bits décalés vers la droite

```
$a = 5;          // 0000 0101
$b = 12;         // 0000 1100
$c = $a & $b;    // $c vaut donc  0000 0100 soit 4
$d = $a | $b;    // $d vaut donc  0000 1101 soit 13
$e = ~ $a;
// $e contient le complément de 5, soit 1111 1010
// (avec devant autant de 1 que vous voulez) (= -6)
```

Opérateurs logiques (booléens)

$a and $b	ET	vrai si a ET b vrais
$a or $b	OU	vrai si a OU b vrai
$a xor $b	OU exclusif	vrai si a OU b vrais, mais par les 2 vrais
! $a	NON	vrai si a faux et inverse
$a && $b	ET	vrai si a ET b vrais
$a || $b	OU	vrai si a OU b vrai

Note : && et || font la même chose que "and" et "or", mais avec une précedence supérieure, ce qui évite parfois de mettre des parenthèses. Astuce vaseuse à éviter pour la lisibilité du code.

On dispose également tous les **opérateurs de comparaison** classiques : **== , != , < , <= , > , >=** .

PHP : instructions de contrôle

Instructions de contôle diverses

(condition) ? (valeur si vrai) : (valeur si faux)
if .. elseif .. elseif .. else ..
switch(expr) .. case expr : .. default : ..
while(expr) ...
do...while(expr) ;
for(init ; test ; incr ;)...
break ; quitter une boucle

Exemple : **p03.php3**

p03.php3

```
<HTML><HEAD>
<TITLE>fichier p03.php3</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF">
<font size="+1"> Début du fichier p03.php3 <br>
Ici code html.<br><hr>
<?php
 echo "<P>Hello World ! écrit par du code PHP.</P>";
 $a = "12";
 $b = "34";
 if ($a > $b) print "<br>a est > b";
 if ($a < $b) print "<br>a est &lt; b";
 if ($a > $b) {
   print "<br>a plus grand que b";
   print "<br>donc b plus petit que a";
 } else {
   print "<br>a plus petit que b";
   print "<br>donc b plus grand que a";
 }
 $a = "123";
?>
<br>Retour au code html.<br>
<?php if ($a=="123"): ?>
<br>A = 123<br>
<?php endif; ?>
</BODY></HTML>
```

Début du fichier p03.php3
Ici code html.
Hello World ! écrit par du code PHP.
a est < b
a plus petit que b
donc b plus grand que a
Retour au code html.
A = 123

<?php
?>
<?php if($a=="123"):?>
<?php endif; ?>

PHP : tableaux

```
<HTML><HEAD><TITLE>fichier tab.php3</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<font size="+1"> Début du fichier tab.php3 <br>
Ici code html. Démo des tableaux php.<br>
<hr>
<?php
 echo "<P>Hello World ! écrit par du code PHP.";
 echo "</P>";
?>
<hr>
<?php
$tab1=array("a","X","c");
$tab1[1]="b";
$tab1[4]="e";
echo "<br>tab1=$tab1";
$lng=count($tab1);
echo "<br>lng=$lng";
for ($i=0; $i<=$lng; $i++)
 { echo"<br>tab1[$i]=$tab1[$i]";
 }
?>
<hr>
```

Début du fichier tab.php3
Ici code html. Démo des
Hello World ! écrit par
tab1=Array
lng=4
tab1[0]=a
tab1[1]=b
tab1[2]=c
tab1[3]=
tab1[4]=e

Tableau indexé par nombres

tab.php3 (suite)

```
<?php
$sem = array( "lun"=>11, "mar"=>12, "mer"=>14,
              "jeu"=>15, "ven"=>16 );
$sem["sam"]=17;
$sem["dim"]=18;

$total=0;
$ln = count($sem);
reset($sem);
while( list($key, $value) = each($sem) ) {
 echo "<br> Jour $key , fait : $value";
 $total = $total + $value;
}
echo "<br>Total = $total";
?>

<hr><br>Retour au code html.<br>
</BODY></HTML>
```

tableau associatif
avec mécanisme d'exploration
(itérateur)

```
Jour lun , fait : 11
Jour mar , fait : 12
Jour mer , fait : 14
Jour jeu , fait : 15
Jour ven , fait : 16
Jour sam , fait : 17
Jour dim , fait : 18
Total = 103
```

Tableau indexé par des chaines

PHP : autres possibilités du langage

Création et appels de fonctions
Création de classes et instanciation d'objets
Fonctions d'Entrée/Sortie et accès aux fichiers sur le disque du serveur : fopen(), fputs(), fclose(), ...

Fonctions : nombreuses bibliothèque

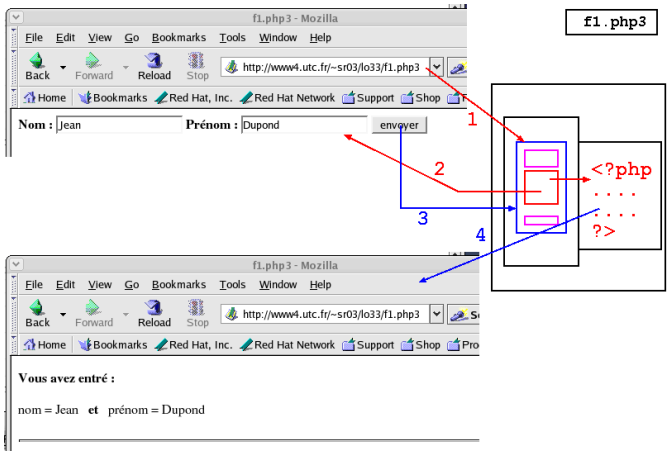
- gestion des échanges avec le navigateur,
- gestion des images,
- création de graphiques,
- accès à des bases de données,
- accès à des serveurs IMAP et LDAP,
- ...

Interaction entre PHP et le navigateur client

PHP facilite beaucoup la tâche du programmeur de scripts CGI :

- accès automatique et simple aux différents champs des formulaires
- si un formulaire a pour cible (paramètre "action=") un script PHP, les **valeurs des champs** du formulaire sont **affectées à des variables PHP** ayant le même nom que le champ :
- par exemple, la valeur du **champ "nom"** se retrouve dans la variable PHP "**$nom**".

Exemple avec un formulaire : f1.php3



```
<?
print("<HTML><HEAD><TITLE>f1.php3</TITLE></HEAD><BODY>");

     print("<FORM ACTION=\"f1.php3\" METHOD=\"POST\" >\n");
/* est-ce le premier appel ou le retour du formulaire ? */
if ( !$premier ) {
     $premier = 1;
     print("<FORM ACTION=\"f1.php3\" METHOD=\"POST\" >\n");
     print("<B>Nom : </B>");
     print("<INPUT TYPE=\"text\" NAME=\"nom\" >\n");
     print("<B>Prénom : </B>");
     print("<INPUT TYPE=\"text\" NAME=\"prenom\" >\n");
     print(
     "<INPUT TYPE=\"hidden\" NAME=\"premier\" VALUE=1 >\n");
     print(
"<INPUT TYPE=\"submit\" NAME=\"envoi\" VALUE=\"envoyer\" >\n");
     print("</FORM>\n");
     print("</BODY></HTML>\n");

} else {   /* retour */
     print("<B><P>Vous avez entré :</B><BR>\n");
     print(
   "<P>nom = $nom   <B>et</B>   prénom = $prenom\n");
     print("<br><hr>");
     print("</BODY></HTML>\n");
}
?>
```

17

Code HTML envoyé au browser

lors du premier appel

```
<html><head><title>f1.php3</title>
</head><body>
<form ACTION="f1.php3" METHOD="POST" >
<b>Nom : </b><input TYPE="text" NAME="nom" >
<b>Prénom : </b><input TYPE="text" NAME="prenom" >
<input TYPE="hidden" NAME="premier" VALUE=1 >
<input TYPE="submit" NAME="envoi" VALUE="envoyer" >
</form>
</body></html>
```

lors du deuxième appel

```
<html><head>
<title>f1.php3</title>
</head><body>
<b><p>Vous avez entré :</b><br>
<p>nom = Jean   <b>et</b>   prénom = Dupond
<br><hr>
</body></html>
```

18

PHP et les sessions

Qu'est-ce qu'une **session** ?

HTTP a été conçu dans un contexte de confiance ("trusted") : tous les ordinateurs connectés sur le site du CERN étaient bien sûr considérés comme autorisés.

Pour des raisons de simplicité de réalisation le protocole HTTP a été conçu **sans état** : si vous faites une série de "n" requêtes sur un site web depuis un même client, cela ne fait pas de différence que les requêtes soient faites en 10 minutes ou qu'elles s'étalent sur une journée, avec des interruptions de plusieurs heures entre les requêtes.

Le serveur ne fait pas la différence entre les deux cas.

Puis le web est sorti de son contexte initial et a été progressivement utilisé à autre chose.

En particulier, la simplicité d'utilisation de l'interface, et son expansion lui donnant un caractère de plus en plus universel, a conduit à l'utiliser pour des **opérations de type transaction**.

Une **transaction** est une **série d'échanges liés entre eux** : le résultat de la requête "n" est **dépendant** du résultat des requêtes "n-1", "n-2", ...

Exemples :

- achat d'un billet (train, avion),
- achats sur un site de commerce (gestion d'un panier d'achats),
- inscription en ligne à des UVs, ...

Cette notion de transaction est plus facile à gérer dès lors que les échanges se font dans le cadre d'une **session**.

Une session sera une suite d'échanges requête-réponse dans laquelle **chaque requête sera identifiée par le serveur comme étant en provenance du même client**.

Pour cela il faudra pouvoir conserver les valeurs de certaines variables tout au long de la session.

Exemples de variables que l'on peut souhaiter conserver :

- login, mot de passe,
- identifiant de client (associé à un compte client),
- valeur de la variable "langage" pour un site multilingue, ...

Les solutions "classiques" n'utilisent que du HTML :

- garder la valeur des variables de session dans les variales "hidden" d'une balise <form>, et les "faire suivre" du client vers le serveur et inversement pendant toute la session. Soit dans un URL-encodé (limité à 255 caractères), soit par une méthode PUT. Mais les valeurs circulent en clair. Le client peut "tricher" et les modifier .
- stocker les données à conserver dans un "cookie" (petit fichier stocké par le navigateur sur le poste client). Là aussi la sécurité n'est pas garantie,

de plus si le client refuse les cookies, le site ne fonctionne plus.

PHP4 a introduit des fonctions permettant de gérer les sessions assez simplement.

Le serveur va gérer **côté serveur** la conservation de la valeur des variables tout au long de la session.

Il va attribuer à chaque session un **identifiant de session unique**.

Il suffira de faire passer ce seul identifiant de page en page tout au long de la session.

PHP stocke les variables d'une session dans un fichier du répertoire session.save_path défini dans la configuration du serveur PHP.

Par exemple une page "login.php" contient une forme qui envoie ses données avec POST et contient les variables "lelogin" et "lepass".

Quand l'utilisateur fait "submit", la page cible du submit est activée et on peut dans son code PHP, récupérer les valeurs des variables du formulaire :


```
$monlogin = $_POST['lelogin'];
$monpass = $_POST['lepass'];
```

On va alors ouvrir la session par : `session_start();`

- S'il y a déjà une session ouverte, les variables de cette session (le contexte) est "restitué", c'est-à-dire rendu disponible dans l'exécution PHP en cours.
- S'il n'y a pas de session en cours, un nouvel **ID de session** est généré et un contexte de session créé sur le serveur. On va revenir après sur la gestion de l'ID.

On vérifie l'existence du login et la correction du mot de passe, par exemple en interrogeant une base de données (MySQL, OpenLDAP,...).

On stocke des variables dans le contexte de la session par :

```
$_SESSION['name'] = $monlogin;
$_SESSION['logged'] = 1;
```


On peut faire la même chose avec :

```
$name = $monlogin; session_register("name");
$logged = 1; session_register("logged");
```

La seconde méthode revient à créer une variable **globale** qui sera définie dans toutes les pages qui comporteront un session_start(). Il faudra donc faire attention à des collisions avec des variables locales à une page ayant le même nom.

Après exécution de session_start() les variables de session sont utilisables par exemple par :

```
if($_SESSION['logged']==1){...}
if(session_is_registered("logged") && $logged==1){...}
```

Gestion de l'ID de session PHP

À l'exécution de session_start(), PHP cherche l'ID de session (s'il ne le

trouve pas il en crèe un nouveau).

Comment cherche-t-il ?

Le plus simple : le cookie : PHP pose sur le client un cookie qui contient l'ID de session. Ce cookie a une durée de vie nulle, ce qui fait qu'il est détruit si le navigateur est arrêté. Il suffit de configurer PHP sur le serveur avec "session.use_cookies" à 1. Alors la gestion de l'ID est entièrement prise en charge par PHP.

Passer l'id de session dans l'URL : fonctionne quelque soit le paramétrage du navigateur. Mais elle implique une gestion assez lourde. En effet, il faut alors ajouter à tous les liens pointant sur des pages du site gérant la session un paramètre contenant l'ID de session :

```
"index.php?".session_name()."=".session_id()
ou $sid = session_name()."=".session_id();
printf('<a href="script.php?%s"> Link </a>', $sid);
```


Cette méthode change les URLs du site mais ne demande pas d'action particulière dans les pages cibles du lien : lors du session_start(), PHP récupère l'ID dans les variables de l'URL.

Si ce paramètre est le seul à être ajouté, cela ne suffit pas pour perturber les moteurs de recherche qui continuent à indexer les pages ainsi définies .

Ajouter l'ID aux URLs : si on compile PHP avec l'option –enable-trans-sid et qu'on le configure avecsession.use_trans_sid à 1, PHP ajoute l'ID aux URLs de manière invisible. **Mais** il ne fait cela que pour les URLs relatifs à la racine du site ("./page.php"). De plus cela charge le serveur qui doit "parser" chaque page envoyée pour en transformer les URLs.

La fonction **<?php phpinfo();?>** donne la liste de toutes les options de compilation et configur ation d'un serveur PHP existant.

Supprimer des variables de session

$_SESSION['mavariable']="" ; ou $_SESSION['variable'] = FALSE ; ou session_unregister("variable") suivant la façon dont la variable a été créée.

Dernière astuce pour déboguer : le code suivant imprime toutes les variables de la session en cours :

echo "Var. de session : <pre>" ; print_r($_SESSION) ; echo "</pre>" ;

Terminer une session

PHP n'a pas de moyen de savoir si un utilisateur met simplement longtemps à lire une page ou bien s'il a quitté le site.

PHP utilise donc une variable de garde (timeout) **gc_maxlifetime** qui est une variable de configur ation du serveur PHP. Au delà de cette limite une session non utilisée est détruite.

Le test des sessions inactives est fait toutes les "n" requêtes seulement avec une probabilité égale à gc_probability.

L'idéal est bien sûr de mettre sur le site un bouton "déconnexion" qui point sur une page exécutant session_destroy().

Remarque : dans le cas où on a utilisé $_SESSION pour stocker les variables, il faut le mettre à vide par $_SESSION = array() ; avant de détruire la session.

Les sessions ayant, par construction, une durée de vie limitée, ne permettent pas de conserver d'informations d'une visite à une autre. Pour cela il faut utiliser un stockage permanent sur le serveur (fichiers , base de données).

Sessions et indexation

les variables transmises dans l'URL des pages dynamiques peuvent gêner leur indexation par les robots des sites de recherche comme Google ou Yahoo.

En effet, si un identifiant de session est transmis dans l'URL, alors, à

chaque visite du robot, une même page aura une URL différente puisque PHP crèe automatiquement un nouvel ID.

Certains robots reconnaissent l'identifiant de session PHP et sont capables de l'enlever lorsqu'il est à la fin de l'URL. Toutefois, si en plus de l'identifiant le site passe d'autres paramètres dans l'URL, alors le robot ne suit pas le lien et n'indexe pas la cible.

Pour éviter cela certains sites de commerce déploient tout un arsenal dit URL Rewriting consistant à transformer des URLs tels que :
article.php?id=12&page=2&rubrique=5 non indexables en des URLs "propes" tels que article_12_2_5.html pour permettre leur indexation.

Questions sur ch12-PHP

➤ Quel est le principe de fonctionnement de PHP ?
Réponse : diapos 1 à 4.

➤ Quels sont les types de tableaux manipulables en PHP ?
Réponse : diapos 12 et 13.

➤ Quand la cible "action=" d'un formulaire HTML pointe sur une page PHP, comment PHP récupère-t-il la valeur des variables de la forme ?
Réponse : diapos 15.

➤ Si une page PHP contient une forme de saisie HTML et renvoie la réponse par ACTION= sur elle-même, comment doit-elle être programmée ?
Réponse : diapos 17.

➤ Qu'est-ce qu'une session et pourquoi en a-t-on besoin ?
Réponse : diapos 19 et 20.

➤ Comment PHPgère-t-il les sessions ?
Réponse : diapos 22 à 24.

Ch12 - PHP

Fin du chapitre.

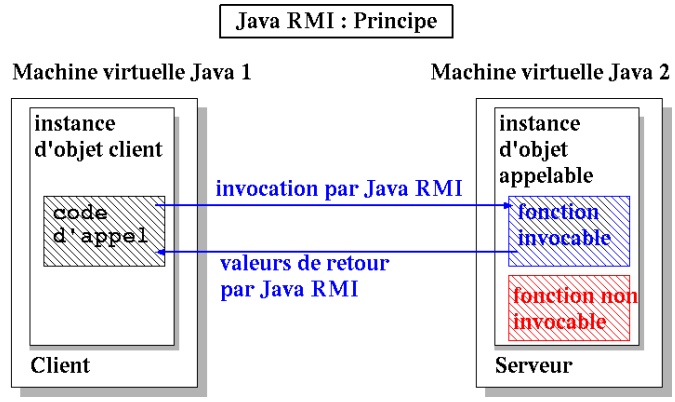
Ch13 - Java RMI : Remote Method Invocation

- Java RMI : présentation
- Java RMI : premier exemple
- Java RMI : objet ordinaire et objet distant

Java RMI : Remote Method Invocation

C'est l'implémentation du **RPC en Java**, permettant d'appeler une méthode d'un objet géré dans une autre machine virtuelle (JVM) que celle du code appelant.

- Permettre à du code d'un objet tournant dans une machine virtuelle d'invoquer une méthode d'un objet tournant *dans une autre JVM*.
- les deux JVM peuvent être deux machines tournant dans des process différents sur la même machine ou sur des **machines différentes** reliées par un réseau TCP/IP.
- la machine contenant **l'objet appelé** est le **serveur**,
- la machine contenant **l'objet appelant** est le **client**.



Java RMI : propriétés

- il suffit de deux lignes de code côté serveur pour "**enregistrer**" l'objet appelé et "exposer" ses méthodes,
- il suffit d'une ligne de code en plus côté client pour **obtenir une référence** sur l'objet distant,
- le serveur doit définir la classe et instancier un objet distant (remote),
- le client ET le serveur doivent avoir accès à une **interface** qui déclare les **méthodes** qui peuvent être **invoquées** à distance,
- le client et le serveur doivent définir un "RMISecurityManager()" ;

Java RMI : propriétés

- par l'invocation de méthodes sur l'objet distant, le client peut **passer des objets Java en paramètres** et **recevoir des objets en retour** : ceci est réalisé par la capacité de "sérialisation" des objets de Java ;
- le client et le serveur peuvent être compilés sur la même machine (puis les .class déplacés) ou bien compilés séparément,

Java RMI : accès et nommage

- le mécanisme de "bootstrap" (point de départ) pour qu'un client **obtienne des références à des objets distants** est basé sur **l'enregistrement de noms** d'objets dans un registre, qui peut ensuite être accédé par les clients à l'aide de fonctions et d'une syntaxe de type "URL" : **rmi ://host :port/name**
- **fonctions de la classe "Naming" :**
  - **bind** (String_name, RemoteObj) associe un objet à un nom,
  - **list** (String_URL) renvoie un tableau des noms d'objets dans le registre,
  - **lookup** (String_URL_obj) renvoie une référence sur l'objet,
  - **rebind** (String_name, RemoteObj) remplace un objet par un nouveau,
  - **unbind** (String_name) supprime un nom du registre ;

Java RMI : premier exemple

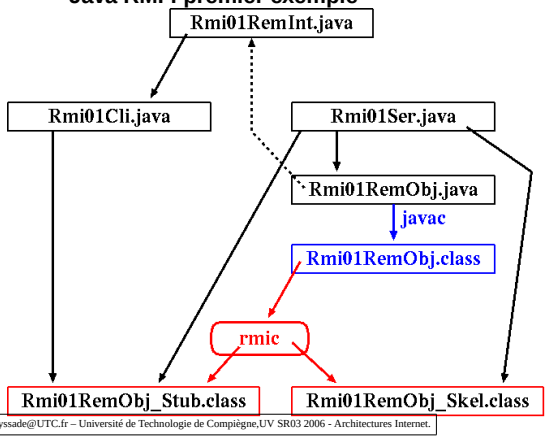
Comme premier exemple, on montre une version RMI d'un "Hello World" ; c'est donc un programme minimal. Il requiert 4 fichiers sources :

- **Rmi01Cli.java :** le programme **client** qui va invoquer une méthode d'un objet distant,
- **Rmi01Ser.java :** le programme **serveur** qui va instancier l'objet distant et l'enregistrer pour le rendre accessible,
- **Rmi01RemInt.java :** un fichier de définition d'une **interface** qui déclare les signatures des méthodes d'un objet distant qui peuvent être invoquées par un client à distance,
- **Rmi01RemObj.java :** un fichier de définition de la classe des **objets distants** dont les méthodes déclarées dans Rmi01RemInt.java seront invocables à distance ;

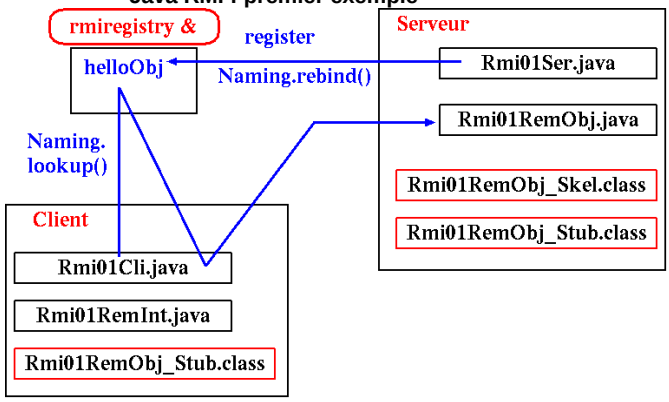
En plus des 4 sources, il faut deux utilitaires : **rmic** et **rmiregistry**.

- **rmic :** il fait la "compilation" des fichiers "_Stub" et "_Skel" : à partir de Rmi01RemObj.class , il fabrique :
  - **Rmi01RemObj_Stub.class** (pour le client ET le serveur),
  - **Rmi01RemObj_Skel.class** (pour le serveur).
- **rmiregistry :** il crée et maintient un registre des objets d'un serveur, objets dont des méthodes peuvent être invoquées à distance par des clients ; Il peut être invoqué de 2 façons :
  - par le serveur à l'exécution : LocateRegistry.createRegistry(1099) ;
  - comme un process indépendant par : > rmiregistry &
- le serveur enregistre un objet par : Naming.rebind("helloObj", theRemoteObj) ;
- le client accède à un objet par : Rmi01RemInt c1 = (Rmi01RemInt)Naming.lookup(partOfUrl+ "helloObj") ;

Java RMI : premier exemple



Java RMI : premier exemple



Compilation et exécution

```
/rmi1> ls *.java
Rmi01Cli.java   Rmi01RemInt.java  Rmi01RemObj.java  Rmi01Ser.java
/rmi1> javac *.java
/rmi1> ls *.class
Rmi01Cli.class  Rmi01RemInt.class Rmi01RemObj.class  Rmi01Ser.class

/rmi1> rmic Rmi01RemObj
/rmi1> ls *.class
Rmi01Cli.class    Rmi01RemObj.class    Rmi01RemObj_Stub.class
Rmi01RemInt.class  Rmi01RemObj_Skel.class  Rmi01Ser.class

==> création de Rmi01RemObj_Stub.class (côté client et serveur)
  et Rmi01RemObj_Skel.class (côté serveur)

/rmi1> java Rmi01Ser
Remote obj helloObj ready to use

/rmi1> java Rmi01Cli
Hello à vous! Bonjour!
```


Client

```
/* Rmi01Cli.java : an RMI hello world program. */

import java.rmi.*;
import java.rmi.server.*;

public class Rmi01Cli{

  public static void main(String[] args){
    System.setSecurityManager(new RMISecurityManager());
    String partOfUrl = "rmi://localhost/";
    // remplacer par "rmi://ultraxxx.utc/" si besoin
    try{
      Rmi01RemInt c1 =
        (Rmi01RemInt)Naming.lookup(partOfUrl + "helloObj");
      System.out.println(c1.helloRemoteObj("Bonjour!"));
    }catch(Exception e){System.out.println("Erreur "+ e);}
    System.exit(0);
  }//end main
}//end class Rmi01Cli
```

appel distant → c1.helloRemoteObj("Bonjour!")

c1 = référence de l'objet de nom "helloObj"

Serveur

```
/* Rmi01Ser.java : le serveur distant appelé par RMI */
/* ce serveur "contient" l'objet "helloObj" qui possède
   la méthode helloRemoteObj          */
import java.rmi.*;
import java.rmi.server.*;
import sun.applet.*;
import java.rmi.registry.LocateRegistry;

public class Rmi01Ser{
  public static void main(String args[]){
    System.setSecurityManager(new RMISecurityManager());

    try{
      Rmi01RemObj theRemoteObj = new Rmi01RemObj();
      LocateRegistry.createRegistry(1099);
      Naming.rebind("helloObj", theRemoteObj);
      System.out.println("Remote obj helloObj ready to use");
    }catch(Exception e){System.out.println("Erreur: " + e);}
  }//end main
}//end class Rmi01Ser
```

créer une instance d'objet distant

créer le registry (un serveur de noms d'objets)

enregistrer l'instance d'objet distant dans le registre sous le nom "helloObj"

L'interface

```
//File Rmi01RemInt.java
import java.rmi.*;

public interface Rmi01RemInt extends Remote{

  String helloRemoteObj(String client)
                                  throws RemoteException;
}//end Rmi01RemInt definition d'interface
```

"extends Remote" est obligatoire : seuls les objets qui implémentent "Remote" peuvent être invoqués par RMI.

Dans l'objet distant Rmi01RemObj.java on trouve :

public class Rmi01RemObj extends UnicastRemoteObject implements Rmi01RemInt { ....

Dans Rmi01Cli.java
on trouve :
Rmi01RemInt c1 = (Rmi01RemInt)Naming.lookup(partOfUrl + "helloObj");
c1.helloRemoteObj("Bonjour!")

L'objet distant

```
/* Rmi01RemObj.java : l'objet remote RMI pour hello world */

 import java.rmi.*;
 import java.rmi.server.*;

 public class Rmi01RemObj extends UnicastRemoteObject
 implements Rmi01RemInt{

 public Rmi01RemObj() throws RemoteException{
   super();
 }//end constructor

 // méthode invocable à distance
 public String helloRemoteObj(String client)
        throws RemoteException{
        return "Hello à vous! " + client;
 }//end method()

}//end class Rmi01RemObj
```

"extends UnicastRemoteObject" est obligatoire : hérite de la capacité à se comporter comme une extrémité d'un lien de communication.

Appel RMI avec échange d'objet distant et d'objet ordinaire

Cet exemple est destiné à montrer la différence entre les objets "**ordinaires**" (i.e. non-distants) et les objets "**distants**" ("Remote") lorsque des **objets sont passés en argument** ou en **valeur de retour** lors d'un appel RMI.

Le client : invoque successivement une méthode sur un objet ordinaire, puis sur un objet "remote" : **Rmi05Client.java**

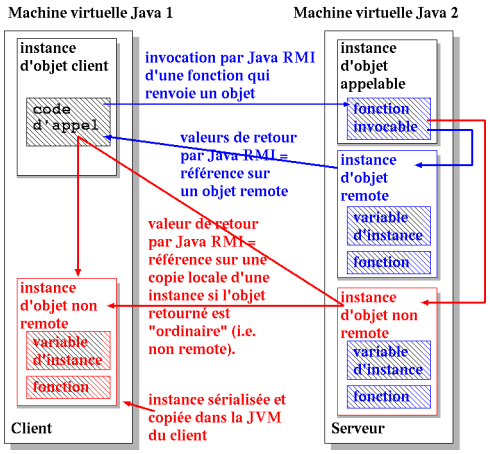
Les interfaces : **Rmi05RemIntA.java** et **Rmi05RemIntB.java**

Le serveur : **Rmi05Server.java**

Les implémentations des classes de service :

Rmi05RemObjA.java et **Rmi05RemObjB.java**

Java RMI avce échange d'objets



Compilation et exécution

```
/rmi5> ls
Rmi05Server.java    Rmi05RemIntA.java   Rmi05RemObjA.java
Rmi05Client.java    Rmi05RemIntB.java   Rmi05RemObjB.java

/rmi5> javac *.java
Note: Rmi05Client.java uses a deprecated API.
Recompile with "-deprecation" for details.
1 warning       //ignorer ce warning
/rmi5> rmic Rmi05RemObjA     //construire _Stub et _Skel
/rmi5> rmic Rmi05RemObjB
```

```
/rmi5> java Rmi05Server    //lancer serveur
        -- init objet Rmi05RemObjB --
Remote obj A ready to use
```

RemObjA et RemObjB sont tous les deux des objets "remote".
Mais une fonction de RemObjA renvoie en valeur de retour un objet RemObjB

Le serveur ne déclare qu'un objet A dans le registre.
Le client va invoquer :
- une fonction renvoyant un objet ordinaire : refToObjA.getDateObj()
- une fonction renvoyant un objet remote : refToObjA.getRemObj()

©Michel.Vayssade@UTC.fr – Université de Technologie de Compiègne,UV SR03 2006 - Architectures Internet.

Compilation et exécution

```
/rmi5> java Rmi05Client    //1er run client
        -- obtenir une ref sur obj serveur --

        -- 1er test -- objet ordinaire --
        Demander au serveur un OBJET date ordinaire.

demander la date a cet objet date= 1
faire date=32 dans objet date
re-demander la date a cet objet date= 2

re-demander l'OBJET DATE au serveur
demander la date au nouvel OBJET DATE : 1

        -- 2eme test -- objet remote --
        Demander au serveur un OBJET date remote.

demander la date a objet date remote= 99
faire date=33 dans objet date remote
re-demander la date a objet date remote= 33

RE-Demander au serveur l'OBJET date remote.
re-demander date au nouveau objet date remote= 33
```

initialisé sur le serveur

modifiée LOCALEMENT sur le CLIENT

initialisé sur le serveur

initialisé sur le serveur

modifiée À DISTANCE sur le SERVEUR

L'instance sur le serveur à conservé la valeur modifiée

©Michel.Vayssade@UTC.fr – Université de Technologie de Compiègne,UV SR03 2006 - Architectures Internet.

Rmi05Client.java

```
/* Rmi05Client.java */
 import java.rmi.*;
 import java.rmi.server.*;
 import java.util.*;

 public class Rmi05Client{
     public static void main(String[] args){
     System.setSecurityManager(new RMISecurityManager());
     String partOfUrl = "rmi://localhost/";
     try{
     System.out.println("-- obtenir une ref sur obj serveur --");
       Rmi05RemIntA refToObjA =
          (Rmi05RemIntA)Naming.lookup(partOfUrl + "unObjetA");

    System.out.println("-- 1er test -- objet ordinaire --");
       System.out.println(
              "Demander au serveur un OBJET date ordinaire.");
       Date theDate = refToObjA.getDateObj();
       System.out.println("demander la date a cet objet date= "
                                        + theDate.getDate());

       System.out.println("faire date=32 dans objet date");
       theDate.setDate(32);
       System.out.println("re-demander la date a cet objet date= "
                                        + theDate.getDate());

       System.out.println("re-demander l'OBJET DATE au serveur");
       theDate = refToObjA.getDateObj();
       System.out.println("demander la date au nouvel OBJET DATE: "
                                        + theDate.getDate());
```


Rmi05Client.java

```
 System.out.println("-- 2eme test-- objet remote --");
     System.out.println(
            "Demander au serveur un OBJET date remote.");
     Rmi05RemIntB theObj = refToObjA.getRemObj();
     System.out.println("demander la date a objet date remote= "
                                        + theObj.getData());

     System.out.println("faire date=33 dans objet date remote");
     theObj.setData(33);
     System.out.println(
            "re-demander la date a objet date remote= "
                                        + theObj.getData());
     System.out.println(
            "RE-Demander au serveur l'OBJET date remote.");
     theObj = refToObjA.getRemObj();
     System.out.println(
            "re-demander date au nouveau objet date remote= "
                                        + theObj.getData());

   }catch(Exception e){System.out.println("Error " + e);}
   System.exit(0);
  }//end main
}//end class Rmi05Client
```

Dans ce second test, toutes les actions sur "theObj" se font à distance:
c'est l'instance de l'objet sur le serveur qui est accédée et modifiée.

Rmi05RemIntA.java et Rmi05RemIntB.java

```
/* Rmi05RemIntA.java */
 import java.rmi.*;
 import java.util.*;

 public interface Rmi05RemIntA extends Remote{
   Date getDateObj() throws RemoteException;
   Rmi05RemIntB getRemObj() throws RemoteException;

 }//end Rmi05RemIntA definition
```

Définition de l'objet remote A et des deux fonctions "exposées" :
ces 2 fonctions renvoient un objet (ordinaire: Date ou remote: Rmi05RemIntB)

Définition de l'objet B, déclaré remote, qui est renvoyé comme valeur de retour par l'une des fonctions de l'objet A.

```
/* Rmi05RemIntB.java */
 import java.rmi.*;

 public interface Rmi05RemIntB extends Remote{
   void setData(int data) throws RemoteException;
   int getData() throws RemoteException;

 }//end Rmi05RemIntB definition
```

©Michel.Vayssade@UTC.fr – Université de Technologie de Compiègne,UV SR03 2006 - Architectures Internet.

Rmi05Server.java

```
/* Rmi05Server.java */
/*
 Un serveur destiné à montrer la différence entre les objets
 "ordinaires" (i.e. non-distants) et les objets "distants"
 ("Remote") lorsque des objets sont passés en argument ou en
 valeur de retour.
*/
 import java.rmi.*;
 import java.rmi.server.*;
 import sun.applet.*;
 import java.rmi.registry.LocateRegistry;

public class Rmi05Server{
   public static void main(String args[]){
     System.setSecurityManager(new RMISecurityManager());
     try{
       Rmi05RemObjA remoteObjA = new Rmi05RemObjA();

       LocateRegistry.createRegistry(1099);
       Naming.rebind("unObjetA", remoteObjA);

       System.out.println("Remote obj A ready to use");
     }catch(Exception e){System.out.println("Error: " + e);}
   }//end main
 }//end class Rmi05Server
```

Le serveur se contente d'instancier un objet A.
C'est l'objet A qui va instancier les objets renvoyés (Date et B).

©Michel.Vayssade@UTC.fr – Université de Technologie de Compiègne,UV SR03 2006 - Architectures Internet.

Rmi05RemObjA.java

```
/* Rmi05RemObjA.java */
/* Cette classe implemente Remote indirectement à travers
   l'héritage par Rmi05RemIntA.  */
 import java.rmi.*;
 import java.rmi.server.*;
 import java.util.*;

 public class Rmi05RemObjA extends UnicastRemoteObject
                                 implements Rmi05RemIntA{

   private Date theDate = new Date();
   private Rmi05RemObjB theRemoteObj;

   public Rmi05RemObjA() throws RemoteException{
     theRemoteObj = new Rmi05RemObjB(99);
   }//end constructor

   public Date getDateObj() throws RemoteException{
     return theDate;
   }//end getDateObj()

   public Rmi05RemIntB getRemObj() throws RemoteException{
     return theRemoteObj;
   }//end getRemObj()

 }//end class Rmi05RemObjA
```

instancie un objet Date ordinaire

dans le constructeur, A instancie un objet B et l'initialise (99).

l'objet B va stocker la valeur d'initialisation dans une variable d'instance. Ceci n'est fait qu'une fois puisque l'objet B n'est créé qu'une fois. Si la valeur est modifiée, c'est la même variable d'instance qui est écrasée, puisqu'il n'y en a qu'une !

©Michel.Vayssade@UTC.fr – Université de Technologie de Compiègne,UV SR03 2006 - Architectures Internet.

Rmi05RemObjB.java

```
/* Rmi05RemObjB.java */
/* Cette classe implemente Remote indirectement a travers
 l'heritage par Rmi05RemIntA.  */
 import java.rmi.*;
 import java.rmi.server.*;
 import java.util.*;

 public class Rmi05RemObjB extends UnicastRemoteObject
                           implements Rmi05RemIntB{
   private int data;

   public Rmi05RemObjB(int data) throws RemoteException{
     System.out.println("-- init objet Rmi05RemObjB --");
     this.data = data; //initialize the object
   }//end constructor

   public void setData(int data) throws RemoteException{
     this.data = data;
   }//end setData()

   public int getData() throws RemoteException{
     return data;
   }//end getData()
 }//end class Rmi05RemObjB
```


Questions sur ch13-JavaRMI

➤ Quel est le principe de Java RMI ?
Réponse : diapos 2 et 3.

➤ Que fait le client RMI pour accéder aux objets distants ?
Réponse : diapo 4.

➤ Que peut faire le client RMI pour échanger des informations avec le serveur RMI ?
Réponse : diapo 5.

➤ Comment le client "trouve-t-il" les objets distants ?
Réponse : diapo 6.

➤ Quels sont les deux utilitaires indispensables à la préparation d'une connexion RMI ? Quelle est la fonction de chacun d'eux ?
Réponse : diapo 8.

➤ Qu'est-ce qui fait le lien (au niveau du code) compilé entre le client RMI et le serveur RMI ?
Réponse : diapos 7 et 9. Ils doivent tous deux utiliser la même interface.

➤ Que doit faire le serveur RMI pour se préparer à un appel distant ?
Réponse : diapo 13.

➤ Que doit faire le client RMI pour appeler une méthode de l'objet distant ?
Réponse : diapo 12.

➤ Expliquez à l'aide d'un schéma la différence de comportement lors d'un appel sur un objet ordinaire et sur un objet "distant".
Réponse : diapo 17.

(diapo vide)

Ch13 - JavaRMI

Fin du chapitre.

(diapo vide)

(diapo vide)

Ch14 - Java, le client-serveur et le web

- Le client-serveur et le web
- Java et le web - La plateforme Java
- Architectures client-serveur et Java

Le client-serveur et le web

client : entité de calcul qui va requérir un service auprès d'une ressource ;

serveur : entité de gestion de ressource qui va répondre à des requêtes de service.

➤ Les clients sont consommateurs de services et ressources. Les serveurs sont producteurs de services et ressources.

➤ La division des tâches entre client et serveur est un problème difficile. La "meilleure" solution dépendant des techniques disponibles à un moment donné et ... des modes !

Évolution du mode Client-Serveur

Client-Serveur traditionnel "deux-tiers" ➤ Client-Serveur "trois-tiers".

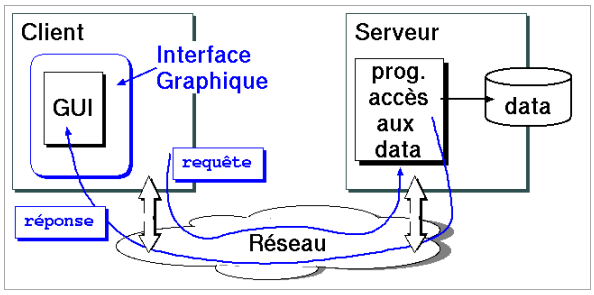


FIG. 1: Client-Serveur deux-tiers

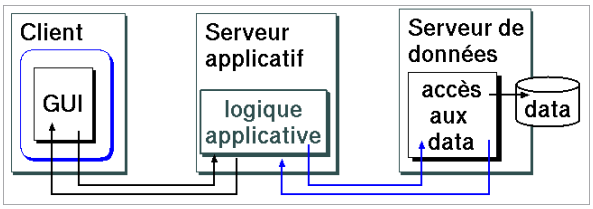


FIG. 2: Client-Serveur trois-tiers

Le client-serveur traditionnel a évolué vers une architecture plus complexe, mais plus souple : le client-serveur dit "3 tiers", constitué de 3 éléments indépendants interagissant. Les différentes raisons de cette évolution seront détaillées plus loin dans le chapitre sur les **serveurs d'applications**.

Client - Serveur : la "collision" avec le Web

Les applications client/serveur traditionnelles faisaient communiquer une application sur un ordinateur personnel (PC ou Mac), chargé de gérer toute l'interactivité avec l'utilisateur avec un serveur chargé de stocker, conserver et consolider les données.

Puis à partir de 94 une nouvelle sorte d'application client/serveur s'est répendue à côté de l'existant : le web, avec un brower-client accédant des serveurs-web.

La simplicité d'accès au web (apprentissage = 0) a fortement poussé à une convergence. D'autant plus que le web était la seule application client-serveur facilement déployable sur l'Internet.

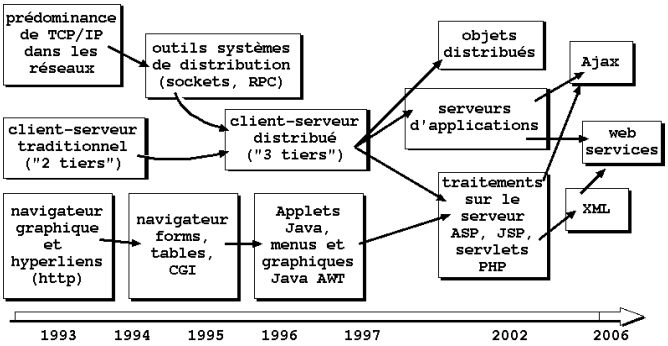


FIG. 3: Le Client-Serveur et le Web

Java et le web

En 5 ans l'Internet est passé du statut de "jouet privé" pour les univesitaires et chercheurs à celui d'une application client/serveur universelle.

L'application clé ("the killer application") qui a provoqué ce changement est le World Wide Web (www, ou le web, ou "la toile").

Le web a évolué rapidement depuis sa création qui peut être datée de l'apparition du browser Mosaic, fin 93. Au début de 97, le web comptait 120 000 serveurs et 20 millions d'utilisateurs, la taille du web doublant tous les 2 mois (+3000 serveurs chaque jour !) et le nombre d'utilisateurs croissant de 1 million par mois.

Parallèlement, les techniques d'interaction avec l'utilisateur du web ont augmenté en interactivité.

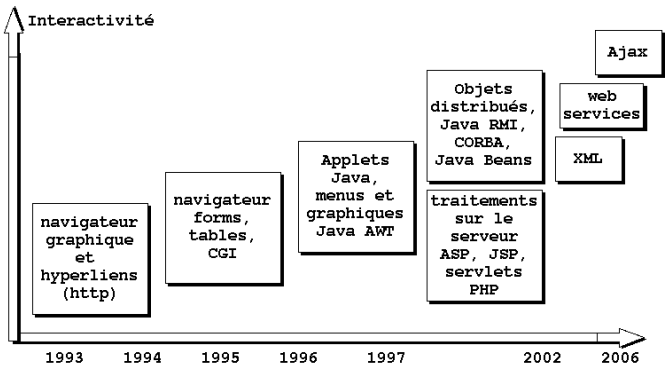


FIG. 4: L'interactivité sur le web

Java a été perçu comme un moyen de fournir au concepteur de sites interactifs les éléments dont manquaient les navigateurs de l'époque.

Java a permis un tout nouveau modèle d'interactivité sur le web : il permet d'écrire des "programmes composants", appelés "Applets", qui peuvent être téléchargés depuis un serveur web vers un client (dans un browser), qui, s'il est compatible java, pourra les exécuter localement sur le poste du client.

Les Applets permettent de distribuer à travers le web du contenu exécutable en même temps que des données.

Tout ceci explique pourquoi l'usage de java s'est développé alors qu'on avait pas vraiment besoin d'un langage de plus.

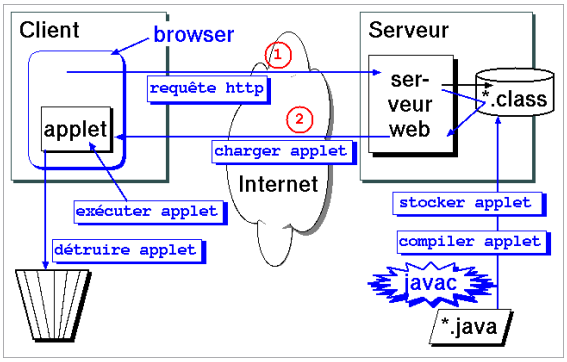


FIG. 5: Client-Serveur en Java

Une applet java est une unité de code mobile (et doit donc être portable). Java atteind la portabilité en compilant les sources (.java) en une suite de **bytescodes** (.class) qui sont les instructions primitives de la **machine virtuelle java (JVM)** ("Java Virtual Machine").

La machine virtuelle java prend en charge tous les détails de "bas niveau" du fonctionnement d'un programme ("gory details") tels que modèle mémoire (big endian) identique sur toutes les JVMs quelque soit le matériel sous-jacent.

La création des bytescode par le compilateur java "javac" représente 80% du travail d'un compilateur traditionnel. Les 20% qui restent sont faits par la JVM au moment de l'exécution.

Ensuite la JVM **interprète** le bytecode.

Java est plus qu'un langage : c'est une plateforme complète de

support d'exécutions de programmes, plus proche d'un système d'exploitation que d'un compilateur associé à une bibliothèque.

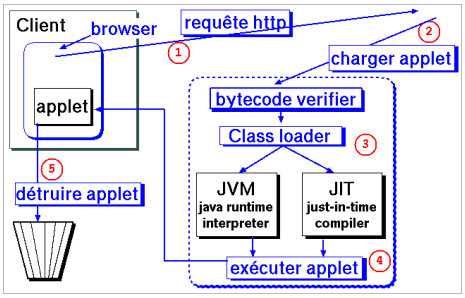


FIG. 6: Java : les applets

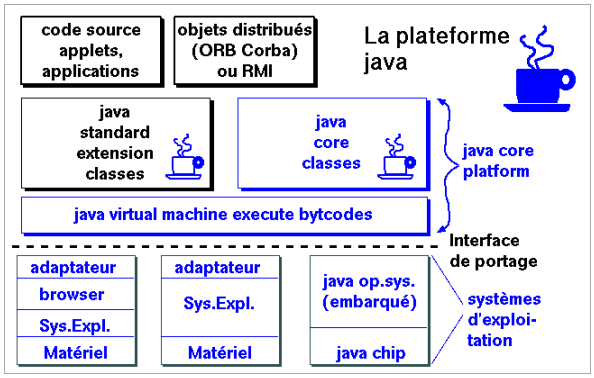


FIG. 7: La plateforme Java

Introduction au modèle de programmation du système Java

Introduction générale

Java n'est pas seulement un langage. C'est aussi une synthèse des évolutions de l'informatique, depuis la création de Smaltalk jusqu'à celle du web.

été 1993 : envol du web, développé au CERN ;

23 mai 1995 : annonce de Java par Sun ;

Java apparrait à la croisée de 3 évolutions : celle des ordinateurs, celle des langages et celle des réseaux.

Il est utile de distinguer le **langage java** du **système java** (on parle aussi de la **plateforme java**).

Le système java

- "Java is not a JAL" ! (Just Another Language),
- l'art de la programmation est dans la gestion de la complexité : complexité du problème à résoudre **plus** complexité de la machine avec laquelle on essaie de le résoudre,
- Java est sans doute le premier langage pour lequel un des objectifs majeurs de sa conception est de diminuer la complexité de conception et de maintenance de programmes,
- –> réduire le temps nécessaire au programmeur, et la difficulté, pour produire du code robuste,
- –> moitié moins long que pour produire un équivalent C++
- donc, plus facile de :
  – créer des programmes,

  – travailler en groupe à la création de programmes,
  – construire des interfaces utilisateur,
  – exécuter les programmes sur différentes machines,
  – écrire des programmes qui communiquent sur un Internet.
- il ne faut pas penser aux fonctions de Java en termes de **codage**, mais en termes de **design**.

Le langage java

Java possède des caractéristiques modernes permettant d'atteindre des objectifs de productivié, rentablité, pérennité et sécurité. En particulier, java est :

- orienté objet,
- multithread,

- fortement et statiquement typé,
- à liaison dynamique,
- gérant la mémoire,
- simplifié,
- sécurisé.

Chacun de ces points sera détaillé plus loin dans le chapitre sur le langage java.

Du langage java à la plateforme Java

Il y a une autre des caractéristiques de Java dont on a pas encore parlé, et qui est une des plus importantes : la **portabilité**.

Les premiers essais de portabilité ont été ceux des langages procéduraux traditionnels : Cobol, Fortran, Pascal, C. Mais bien qu'un

programme C, par exemple, soit en théorie portable sur de nombreuses machines, en pratique l'énorme diversité des bibliothèques dépendantes du système interdisent d'espérer qu'un programme de taille conséquente puisse fonctionner à l'identique par simple recompilation.

Ceci a conduit un certain nombre de projets à modifier la structure de la relation entre le langage et le système d'exploitation, en faisant "remonter" un maximum de fonctions, habituellement implantées dans les bibliothèques du système, au niveau du langage lui-même. On peut citer dans cette optique : Smaltalk, Ada et Objective C / Next. Au passage, on réduit le système d'exploitation à ses fonctions de base (gestion mémoire, gestion multitâche, gestion des communications entre process) pour aboutir au concept de mico-

noyau (Mach, Chorus).

La plateforme Java, en intercalant la machine virtuelle Java entre le système d'exploitation et le langage proprement dit et ses bibliothèques, franchit une étape de plus vers la portabilité universelle des applications.

Architecture de la plateforme Java

La plateforme Java est constituée d'un ensemble de composants qui sont tous, sauf la machine virtuelle, des bibliothèques de classes écrites en Java. Mais à la différence des bibliothèques C des systèmes d'exploitation, il ne s'agit pas d'une simple collection de routines juxtaposées, mais d'un ensemble hiérarchique rendu cohérent par la nature orientée objet du langage Java.

Architectures client-serveur et Java

La plateforme Java a été conçue pour faciliter l'implantation d'applications client-serveur. Particulièrement, pour rendre facile le déploiement de la partie cliente sur un grand nombre de postes, et sa mise à jour, deux opérations qui, dans le cadre des applications client-serveur traditionnelles ont vite fait de tourner au cauchemar pour les administrateurs systèmes, à cause de l'instabilité de la plateforme client "Windows xx".

La section suivante va montrer comment le système Java permet de faire évoluer les architectures client-serveur.

Architectures client-serveur et Java

Les schémas présentent successivement différentes types d'architecture client-serveur, qui diffèrent par le type de code (GUI : gestion interface utilisateur, applicatif (traitements fonctionnels) ou accès aux données (base de données SQL ou objet)) qui est confié au programme client ou à la plateforme serveur. On montre successivement :

1. Architecture client-serveur traditionnelle à deux niveaux avec les traitements sur le client.
2. Architecture client-serveur traditionnelle à deux niveaux avec les traitements sur le serveur.
3. Architecture client-serveur à trois niveaux : séparation plus nette

entre l'interface utilisateur, les traitements et l'accès aux données.

4. Architecture client-serveur à deux niveaux avec arpenteur, requête HTTP et CGI.
5. Architecture client-serveur à trois niveaux : navigateur, CGI, traitements et accès aux données.
6. Architecture client-serveur à deux niveaux avec arpenteur et téléchargement de code Java à la demande. Les objets Java peuvent ouvrir un canal de communication direct avec la source de données.
7. Architecture client-serveur à trois niveaux avec arpenteur et téléchargement de code Java à la demande. Les objets Java téléchargés sur le client gèrent l'interface et communiquent avec des

objets applicatifs sur un serveur par des canaux ORB ou RMI. Les objets applicatifs ouvrent un canal de communication direct avec la source de données. Il y a indépendance entre le code de présentation, le code fonctionnel et le code d'accès aux données.

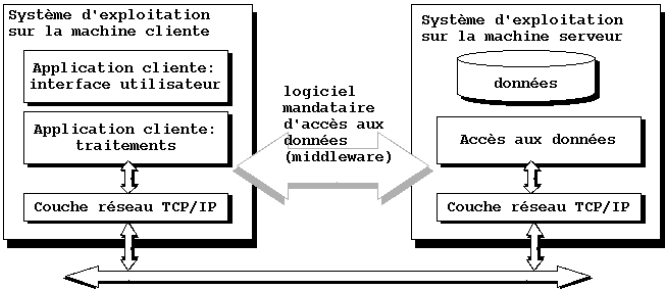


FIG. 8: Client-Serveur et Java 1/7

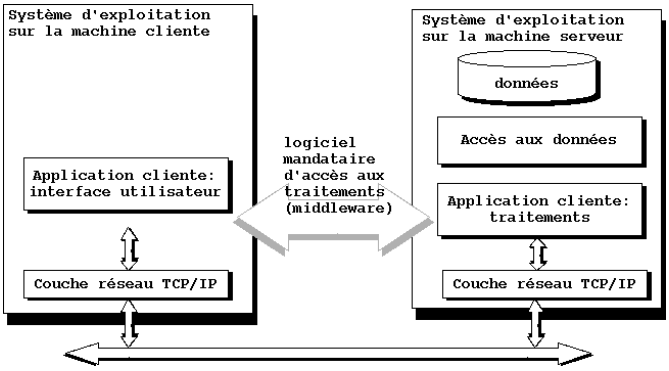


FIG. 9: Client-Serveur et Java 2/7

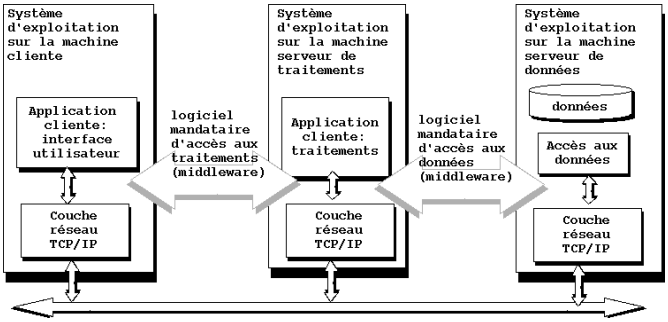


FIG. 10: Client-Serveur et Java 3/7

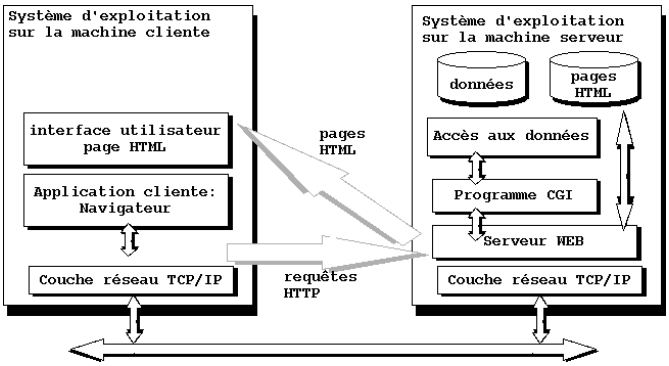


FIG. 11: Client-Serveur et Java 4/7

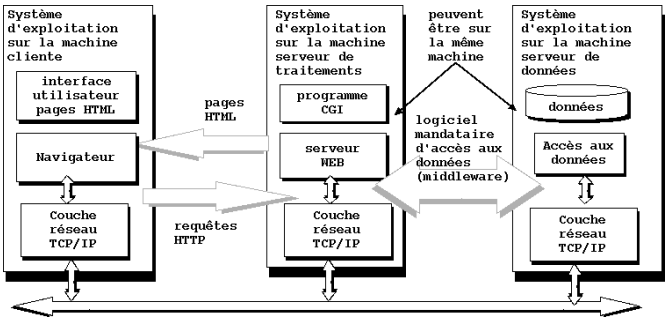


FIG. 12: Client-Serveur et Java 5/7

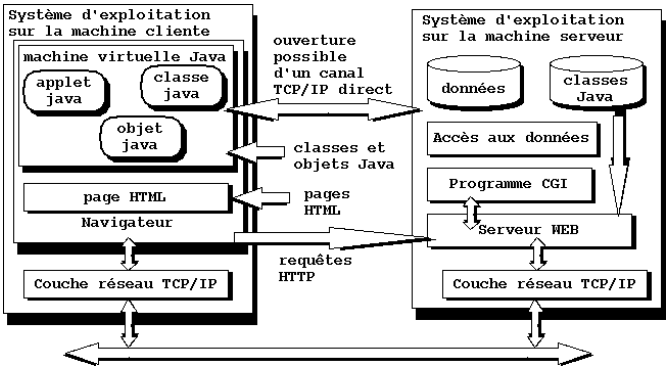


FIG. 13: Client-Serveur et Java 6/7

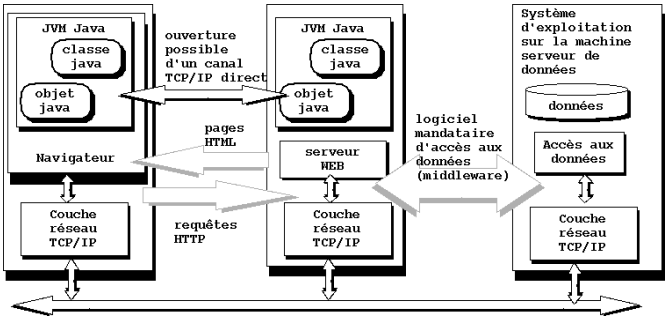


FIG. 14: Client-Serveur et Java 7/7

Questions sur ch14 - Java et le Web

➤ Quel est le principe du client-serveur "trois-tiers" (schéma) ?
Réponse : diapo 4.

➤ Citez plusieurs techniques d'insteraction entre client et serveur web, apparues depuis la création du web.
Réponse : diapo 6.

➤ Décrivez le cycle de vie d'une applet java.
Réponse : diapo 10 (+ diapo 12 si on veut faire du zèle).

➤ Qui exécute le code java ?
Réponse : diapo 11.

➤ Quel est l'objectif principal de la plateforme Java ?
Réponse : diapo 17. La portabilité.

➤ Qu'aporte java dans les architectures client-serveur ?
Réponse : diapos 29 et 30. La possibilité d'une interaction directe entre le client et le serveur sans passer par les requêtes HTTP, avec lesquelles il est difficile de gérer un dialogue de type session.

Ch14 - Java et le web

Fin du chapitre.

SR03 2004 - Cours Architectures Internet - Le système Java
©Michel.Vayssade@utc.fr – Université de Technologie de Compiègne.

15 SR03 2004 - Cours Architectures Internet - Le système Java

15.1 L'architecture du système Java : introduction

Le système Java c'est l'ensemble du langage Java, de la machine virtuelle java (la JVM), de l'API Java et des fichiers de classes Java.

On a toujours utilisé des outils pour aider le concepteur dans les tâches de programmation. Ce n'est que recemment ces outils sont devenus plus complexes, et surtout interconnectés. Les microprocesseurs fabriqués en très grande série n'ont pas seulement donné des stations de travail et des ordinateurs personnels, ils se sont aussi insérés dans un grand nombre d'objets, objets qui sont maintenant de plus en plus souvent connectés à un réseau : fax, matériels vidéo, systèmes de contrôle d'accès, automobiles, etc...

Ce nouvel environnement d'objets interconnectés et disposant d'une capacité de traitement local de l'information, crèe de nouvelles opportunités, mais aussi de nouveaux problèmes techniques pour la mise en œuvre.

Le système Java est un outil bien adapté à ce nouvel environnement et permet de résoudre ou contourner certaines difficultés.

Première difficulté : la diversité

Un réseau typique sera constitué de nombreuses machines de type différents, utilisant des systèmes d'exploitation différents. Java contourne cette difficulté en permettant de créer des programmes indépendants de la plate-forme sous-jacente. Ceci grâce à la machine virtuelle java.

Deuxième difficulté : la sécurité

En plus de leur potentiel "positif", les réseaux peuvent aussi être utilisés dans des buts délictueux : vol ou destruction d'informations, perturbation volontaire du bon fonctionnement du réseau ou des machines

connectées au réseau. Java offre au programmeur d'applications différents moyens de contrôler l'environnement d'exécution (et donc les actions possibles) des programmes Java.

Écrire des logiciels complexes est une entreprise difficile et mal maîtrisée. Il en résulte de nombreux logiciels "fragiles" ou "instables" : les "plantages" de certains logiciels sont devenus tellement habituels que de nombreux utilisateurs nouveaux de l'informatique ne songent même plus à s'en offusquer. Imaginez un instant une automobile ou un poste de télévision qui tomberait en panne tous les deux jours ! L'architecture de Java incorpore un certain nombre de techniques qui visent à réduire cette fragilité, et à empêcher qu'elle ne se propage (un programme qui "se" plante, et qui en plus "plante la machine" !).

Aujourd'hui, presque tous les utilisateurs ont été confrontés au problème des "versions". On produit un document avec un logiciel, et on veut échanger ce document avec quelqu'un d'autre ... qui ne peut pas le réutiliser parce que "ce n'est pas la bonne version". Il faut alors que l'un des deux "installe la bonne version" : les ennuis commencent. En effet une version donnée d'un logiciel exigera pour fonctionner la "bonne" version du système d'exploitation ou d'un composant du système d'exploitation, et cette bonne version pour le logiciel "A" ne sera peut-être pas la même que celle requise par le logiciel "B", dont l'utilisateur a AUSSI besoin pour travailler avec une troisième personne ... D'autre part pour installer une nouvelle version, il faut disposer d'une source : disquettes, cd-rom ou site Internet, contenant une copie du logiciel à installer. Cette source doit être construite et mise à jour pour toute version nouvelle de tout logiciel. De plus, l'utilisateur, n'a pas toujours la compétence nécessaire pour installer correctement le logiciel. Il faut alors que quelqu'un, dans l'organisation, se charge de faire ou d'organiser les installations multiples. Ceci peut devenir un vrai cauchemar sur des parcs de plusieurs centaines ou plusieurs milliers de machines.

Ce problème du déploiement d'une version nouvelle d'un logiciel sur de nombreuses machines, éventuellement de types différents, est fortement facilité par l'architecture de Java.

15.2 L'architecture du système Java : description

Le système Java c'est l'ensemble du langage Java, de la machine virtuelle java (la JVM), de l'API Java et des fichiers de classes, chacun de ces quatre éléments étant défini par une spécification publiée par Sun Microsystems.

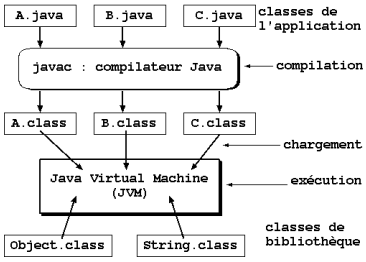


FIG. 156: Fonctionnement de Java

- Les programmes sources, écrits par le développeur, sont stockés dans des fichiers ".java" ;
- Le compilateur "javac" transforme les sources en "bytescode" et génère un fichier ".class" pour chaque classe java ;
- Les fichiers de classes sont envoyés du serveur au client, sur la machine locale ou à travers un internet (réseau TCP/IP),
- Les fichiers .class sont chargés dans la JVM ;
- La JVM interprète les "bytescode" contenus dans les .class. Ceux-ci peuvent faire appel à des objets pré-définis dans les bibliothèques des Java-API.
- Les bibliothèques d'objets pré-définis. Ces objets implémentent les accès aux ressources systèmes en fournissant aux programmes Java une interface commune, identique sur tous les systèmes.

Prises ensemble, la machine virtuelle java et les API java forment une "plateforme" pour laquelle les programmes Java sont compilés. La plateforme JVM + Java API est aussi appelée "Java Runtime System". Ce "système d'exécution java" peut lui-même être implémenté sous forme d'un logiciel et donc être porté sur toute sorte de matériels et de systèmes d'exploitation différents, comme illustré sur la fig ure ci-dessous.

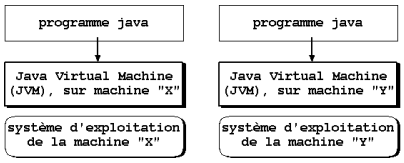


FIG. 157: Java exécute le même binaire partout

La machine virtuelle Java (JVM)

La JVM est le cœur du système Java. C'est une machine **abstraite** dont les spécifications définissent les caractéristiques que doivent posséder **toutes** les machines virtuelles Java. Mais ces spécifications laissent toute liberté aux concepteurs sur la façon de réaliser la machine : on peut faire des JVM uniquement logicielles (par exemple s'exécutant au sein d'un navigateur), mais aussi des machines entièrement matérielles, gravées dans un circuit intégré.

Le travail de la JVM est de charger les fichiers de classes contenant les bytecoodes à exécuter. Le chargeur de classes ("class loader") mets à disposition de la machine d'exécution les classes du programme utilisateur ainsi que toutes les classes de l'API Java utilisées par le programme (et seulement celles là).

JIT et JNI : Just In Time Compiler et Java Native Interface

La JVM peut être implantée de plusieurs façons :
- sous la forme d'un interpréteur : les bytecodes sont lus et interprétés "à la volée",
- sous la forme d'un "compilateur juste à temps" : les bytecodes sont lus et compilés à la volée, puis les classes compilées sont stockées localement par la JVM. Quand elles sont réexécutées, cette nouvelle exécution est beaucoup plus rapide,
- dans les deux cas ci-dessus, le machine peut être soit logicielle soit matérielle.

De plus, pour interagir avec le système d'exploitation de la machine hôte, la JVM appelle des **méthodes natives** implantées dans des bibliothèques précompilées pour la machine hôte. Toutefois, il faut être averti du fait que l'interface avec ces méthodes natives (le JNI) ne fait pas partie

de la spécificati on obligatoire et donc que les programmes Java qui font appel à des méthode natives peuvent perdre le bénéfice de la portabilité du code. Dans ce cas, le concepteur doit arbitrer entre l'avantage de la vitesse d'exécution des méthodes natives et ceux de la portabilité, à moins de faire le portage de la bibliothèque native sur tous les systèmes cibles.

15.3 L'architecture du chargeur de classes de la JVM

La JVM possède un chargeur de classes dont l'architecture est flexible, ce qui permet à un programme de charger les classes selon une méthode choisie par lui. Une application Java peut soit utiliser le chargeur de classes primordial, soit utiliser des "objets chargeurs de classes". Le chargeur primordial fait partie de toute implémentation d'une JVM. C'est lui qui charge les classes "connues" ("trusted") et les classes de l'API Java, ces dernières généralement depuis le disque local.

À l'exécution, une application Java peut installer des objets chargeurs de classes qui vont charger des classes à leur façon, par exemple en les téléchargeant depuis une autre machine sur le réseau ("downloading").

La JVM considère comme connues (trusted) toutes les classes chargées à travers le chargeur primordial et comme suspectes, toutes les classes chargées par un objet chargeur. Les objets chargeurs sont écrits en Java, compilés en bytecode, chargés dans la machine virtuelle et instanciés comme n'importe quel autre objet. Ils sont juste une partie du code exécutable d'une application Java. Les objets chargeurs permettent d'étendre une application à l'exécution. Pour toute classe chargée, la JMV garde trace du chargeur par qui elle a été incluse dans l'application.

Quand une classe fait référence à une autre classe, la machine virtuelle appelle la classe référencée à travers le même chargeur que celui qui a chargé la classe appelante. Ceci, pour les invocations de classes depuis une autre classe. Une requête de chargement explicite, sans invocation de la classe, ne suit évidemment pas cette règle, sinon toutes les classes seraient "filles" du chargeur primordial.

Une conséquence de cette politique de chargement est que, par défaut, les classes ne "voient" que les autres classes chargées par le même chargeur de classes. Par ce biais, le concepteur d'application peut créer plusieurs espaces de nommage à l'intérieur d'une même application Java.

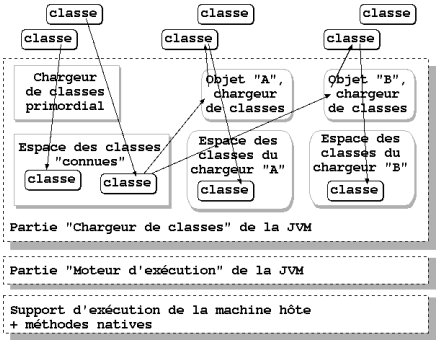


FIG. 158: Java : le chargeur de classes

Les classes chargées par des chargeurs de classes différents n'ont pas accès l'une à l'autre, sauf si l'application le permet explicitement. On peut donc dans une application faire une séparation entre des classes chargées depuis des sources différentes, par exemple, grâce à ce mécanisme, et donc contrôler les interactions entre classes de provenances différentes. Ceci pour, par exemple, protéger des classes "amies" de classes "hostiles".

Exemple : dans un navigateur, on lance une application Java qui installe un chargeur de classes (un "applet class loader") qui "sait" comment chercher les fichiers de classe sur un serveur HTTP. L'application Java lancée par le navigateur va généralement créer un objet chargeur pour chaque source différente sur le réseau depuis laquelle elle va charger des classes. Ainsi les classes de deux sources différentes ne peuvent pas interférer, pusiqu'elles sont dans des espaces de nommage différents.

Le fichier de classes Java

Les fichiers de classes Java contiennent la forme binaire des programmes Java (le bytecode) produite par le compilateur Java et interprété par la machine virtuelle. Ce binaire est donc le même pour toutes les plateformes, puisque ce sont les machines virtuelles qui font l'interface entre le bytecode Java et la plateforme. Les machines virtuelles sont écrites une fois pour chaque type de plateforme, ensuite tous les bytecodes sont exécutables sur toutes les machines virtuelles. Le programmeur d'application

Java n'a donc pas à se soucier de produire un exécutable pour chaque type de machine ou de système d'exploitation.

Le bytecode masque deux éléments dépendants de la plateforme matérielle : les instructions machines (assembleur) du processeur et l'ordre de rangement des octets dans les mots mémoire (big-endian comme sur le Power-PC ou little-endian comme sur la famille Intel X86). Dans un fichier de classe java, l'ordre est big-endian pour tous.

De plus, le bytecode a été conçu pour générer des fichiers compacts, facilement téléchargeables à travers un réseau.

Dernier avantage : les classes java étant liées dynamiquement (à l'exécution), ne sont téléchargées que celles qui seront effectivement utilisée lors d'une exécution particulière. Ceci réduit aussi la quantité de code binaire à charger à travers le réseau.

L'API Java

L'API Java est un ensemble de bibliothèques dynamiques (appelées à l'exécution) qui donnent accès de façon standard aux ressources du système hôte de la machine virtuelle. Quand on écrit un programme Java, on suppose que toutes les classes définies dans l'API Java sont disponibles dans la machine virtuelle sur laquelle le programme va s'exécuter. Cette supposition est raisonnable puisque l'API fait partie des spécifications requises de toute implantation d'une machine virtuelle.

C'est parceque les classes constituant les API Javi sont implantées pour chaque type de machine hôte, que les programmes Java qui n'utilisent que ces classes sont indépendants des machines hôtes.

De plus, avant de faire toute action potentiellement dangereuse, les classes de l'API Java demandent une permission au gestionnaire de sécurité ("security manager"). Celui-ci peut, par exemple, interdire l'accès au disque local.

15.4 Le langage Java

Le langage a parfois été défini sous une forme plaisant par C++– : il contient des spécificités de type "orienté-objet" comme le C++, mais certaines possibilités "dangeureuses" du C++ ont été retirées et certaines modifiées, dans le but de rendre les programmes plus sûrs. C'est un langage complet et moderne (en 1997) qui inclut les fonctionnalités suivantes :

- orienté objet,
- multithread,
- fortement et statiquement typé,
- à liaison dynamique,
- gérant la mémoire,
- simplifié,
- sécurisé.

orienté objet : il respecte les principes d'encapsulation, d'abstraction et d'héritage ; il possède une notion de classe obligant le programmeur à structurer ses données. Toutefois, "l'objet" est une mode qui n'a pas toujours apporté les gains de productivité attendus parce qu'on a fortement sous estimé le changement de mentalité nécessaire pour concevoir et programmer avec les concepts objets.

Mais Java reste utile car ses autres caractéristiques, associées à la plateforme Java, compensent largement ce barrage initial :

- les ateliers de développement masquent en grande partie la complexité,
- il permet un bonne maintenance,
- il prend en charge les applications distribuées en réseau.

L'héritage entre les classes est "simple" : une classe ne peut hériter que **d'une** autre classe. L'héritage de propriétés mulitples, qui est parfois utile, peut se faire à travers l'héritage des **interfaces**, qui lui, peut être multiple.

Java résoud les problèmes de collision de noms des classes et méthodes en introduisant le concept de "package" (inventé par Ada), qui est un espace de nomage.

multithread : Java permet de créer très facilement des flots d'exécution concurrents ("threads"). Ceci permet d'écrire des programmes plus réactifs, plus efficaces du point de vue de la charge du système hôte, et qui utilisent les machines multiprocesseurs.

fortement et statiquement typé : Java effectue un contrôle statique (à la compilation) strict des types.

à liaison dynamique : En java, il n'y a pas d'édition de liens. L'existance des bibliothèques et des classes appelées dans ces bibliothèques se fait au moment de la compilation. Leur code est chargé dans la JVM au

moment de l'exécution. Le compilateur et la JVM retrouvent les classes compilées en bibliothèque car celles-ci sont structurées en paquetages rangés dans une arborescence précise. Il est donc indispensable de découper les projets en hiérarchies de classes, structurées en paquetages.

gérant la mémoire : Java gère la mémoire (code et données) par un ramasse miettes (garbagge collector) qui libère la mémoire de tous les objets inutilisés. Le programmeur n'a donc plus besoin d'allouer et de relacher des zones de mémoire (malloc, dealloc), source très fréquente de bogues dans les programmes C et C++ ("memory leaks") ("delete" oubliés ou mal placés). De plus le langage Java ne possède pas de pointeurs. Ceci supprime une source de très nombreuses erreurs (en C et C++).

simplifié :la syntaxe est simple, les bizarreries d'autres langages ayant été supprimées.

sécurisé : de nombreuses vérifications sont faites par l'environnement avant d'exécuter du code Java : un champ "privé" d'une classe restera innacessible (pas de pointeurs), l'accès aux ressources locales est contrôlé, le débordement de la pile est contrôlé, etc ...

Java n'inclut aucune technique logicielle nouvelle ou expérimentale, mais plutôt fait une combinaison nouvelle de techniques connues, essayées et démontrées dans d'autres langages. Ceci en fait un outil puissant et général que l'on peut appliquer dans de nombreux cas indépendamment du fait que l'on travaille ou non sur une application de réseau.

La gestion des exceptions est incluse dans le langage : la documentation d'une méthode doit indiquer les exceptions qu'ell déclenche ("throw").

Certains ont pu dire que Java avait gardé les bons côtés de C++ et enlevé les éléments potentiellement dangereux.

15.5 Java ou C++ comme langage général ?

On estime que par rapport à C++, Java apporte un gain de productivité (ceci au prix d'une vitesse d'exécution plus faible). Cette productivite est essentiellement due aux restrictions que Java impose dans la manipulation directe de la mémoire (la très grosse majorité des bogues dans les programmes C ou C++ sont dus à des accès à des zones mémoires

non-initialisées ou à des zones mémoires allouées ou désallouées à des moments incorrects : C++ est quasiment un "pousse au crime" à ce sujet).

Java possède un opérateur **new**, comme C++, pour allouer de la mémoire sur le tas ("heap") pour un nouvel objet. Mais, à la différence de C++, Java ne possède pas d'opérateur **delete** que l'on doit utiliser en C++ pour libérer la mémoire occupée par un objet dont on a plus besoin : il suffit de cesser de référencer l'objet et le ramasse miette ("garbagge collector") va, un certain temps plus tard récupérer cette zone mémoire. En C++, pour des objets complexes, il est souvent difficile de savoir quand un objet n'est plus référencé. On peut ainsi oublier de faire le delete, conduisant à un programme "mange mémoire" ("memory leak" : la mémoire restante devient faible), soit mettre le delete trop tôt, soit le mettre deux fois, conduisant à une mémoire corrompue qui provoque un "crash" du programme d'une façon qu'il est difficile de relier à la véritable cause de l'erreur.

Une autre particularité de Java protège la mémoire : Java ne permet pas de faire des conversions arbitraires de pointers ("cast"). Java impose un strict respect des types. Si on a une reférence sur un objet de type "A", on ne peut manipuler l'objet que comme un objet de type "A". On ne pourra convertir la référence sur "A" en une référence sur un objet de type "B" que si "A" et "B" sont dérivées l'une de l'autre et que l'objet est réellement de la classe dérivée (donc des deux types).

Une troisième façon pour Java de protéger la mémoire est par la vérification des bornes des tableaux ("array bound checking"). En C, C++ ou Fortran, on peut déclarer un tableau TAB[100] et accéder à TAB[200] ou à TAB[-10]. Bien sûr ces accès en dehors du tableau vont se faire sur un autre objet rangé en mémoire. Ce type d'accès "hors limite" est une cause fréquente de bogues, lorsque les indices de tableaux sont le résultats de calculs, où d'indirections complexes. Lors d'un accès de ce type, Java va lever une exception.

Une dernière particularté de Java supprime aussi une cause de nombreuses erreurs : avant tout accès à un objet, Java va vérifier que la référence à l'objet n'est pas **null**, et lever une exception si c'est le cas. En C++, utiliser un pointeur nul entraine le "crash" du programme.

Inconvénients

Bien sûr, toutes ces fonctions de Java – test des bornes des tableaux, vérification de la cohérence des conversions de type, test des références

null, et ramasse miettes – font qu'un programme Java sera intrinsèquement plus lent que son équivalent en C++.

Mais le principal effet sur la performance vient du caractère interprété du bytecode Java : en s'exécutant sur une machine virtuelle qui interprète le bytecode, un programme Java va tourner 10 à 30 fois moins vite que son équivalent C++ compilé en instructions natives de la machine hôte. Ce compromis sur la performance a principalement été accepté en échange de l'indépendance de la plateforme.

Heureusement des techniques intermédiaires peuvent limiter cette pénalité : un compilateur "juste à temps" procure un gain d'un facteur 7 à 10 par rapport à l'interprétation pure. Ceci permet de faire des applications qui s'exécutent **suffisamment** rapidement. De plus, la vitesse brute d'exécution dans l'unité centrale n'est **plus** le facteur le plus important dans la performance **perçue** par l'utilisateur : dans de nombreux cas, les programmes passent la plupart de leur temps à attendre des données en provenance de disques ou de réseaux.

Certains programmes ont toutefois réellement **besoin** de s'exécuter très vite. Une façon de contourner cette difficulté de la vitesse d'exécution, peut être d'identifier les paries critiques du code (là où on doit aller vite) et de les compiler en tant que méthodes natives et de les stocker dans les bibliothèques dynamiques de la JVM de la machine hôte. Il faudra bien sûr payer la performance obtenue par une compilation pour chaque type de système et une installation sur chaque machine cliente.

Une alternative "finale" est de compiler l'ensemble des classes utilisées par l'application en un fichier exécutable natif et monolithique qui sera utilisé comme un exécutable C ou C++. On abandonne l'indépendance vis-à-vis de la plateforme, mais on conserve toutefois les avantages en gain de productivité apportés par Java (tout le développement et la mise au point se fait en mode interprété).

Les compromis architecturaux de Java

Les concepteurs du langage ont fait les choix de compromis qui ont le plus de sens dans un contexte d'applications réparties sur un réseau. Le principal compromis est celui de la vitesse. Il est compensé dans de nombreux cas pratiques par le fait que la vitesse d'exécution du code n'est pas l'élément majeur dans le confort d'utilisation. Il est compensé aussi par les gains obtenus en matière de robustesse. Les programmes Java

s'exécutent moins vite pour de nombreuses raisons, qui toutes apportent par ailleurs un bénéfice :

- l'interprétation du bytecode est 10 à 30 fois plus lent qu'un équivalent compilé en code natif,
- la compilation "juste à temps" apporte un gain de 7 à 10 sur l'interprétation, mais il reste une pénalité,
- les programmes Java sont liés dynamiquement (recherche à l'exécution d'une méthode invoquée),
- la machine virtuelle peut avoir à attendre qu'un fichier de classe lui parvienne à travers le réseau,
- les bornes des tableaux sont testées à chaque accès,
- tous les objets sont créés sur le tas (les objets éphémères ne sont pas créés sur la pile comme en C),
- toutes les utilisations d'une référence vers un objet sont vérifiées ne pas être nulles,
- le ramasse miette est connu comme moins efficace qu'une gestion mémoire directe,
- les types primitifs sont les mêmes sur toutes les plateformes au lieu d'être adaptés au plus efficace,
- les chaines de caractères sont toujours en Unicode qui est plus long à traiter que l'ASCII.

Un autre compromis est celui de la perte de contrôle de la gestion de la memoire. Le ramasse miette fait des programmes plus robustes et plus faciles à concevoir, mais ajoute un degré d'incertitude sur la performance à l'exécution. On ne peut pas décider quand le ramasse miette va décider de récupérer les zones mémoires orphelines, ni combien de temps il va mettre. Ceci limite l'usage de Java dans les applications de type temps réel où il est important d'avoir un temps d'exécution prédictible.

Encore un autre compromis est celui qui provient de la volonté d'indépendance de la plateforme. Celle-ci est obtenue par les bibliothèques qui implantent les API. Mais ces API sont un intermédiaire vers des fonctionalités du système d'exploitation de la machine hôte. Or, tous les systèmes d'exploitation n'ont pas la même liste de fonctionnalité. Si l'une d'entre elles est présente sur certains systèmes et pas sur d'autres, faut-il écarter cette fonction de la liste des fonctions disponibles dans l'API, donc utilisable en Java ? Ou faut-il la simuler (plus ou moins bien) dans les implantations de l'API sur les systèmes qui n'ont pas cette fonction ? Si la deuxième solution est choisie, et que la façon dont la fonction man-

quante est simulée ou implantée, ne plait pas au programmeur Java, celui-ci doit-il l'écarter de ses programmes, se créant ainsi un sous-ensemble de l'API, "plus petit dénominateur commun" des fonctions disponibles ? Ceci est plus particulièrement vrai lors de l'écriture d'interfaces utilisateur faisant appel à l'API "AWT" (Abstract Window Toolkit).

Enfin un dernier compromis a été fait pour obtenir un systéme de liens dynamiques. Dans un programme C++ un appel de fonction est compilé en une adresse directement utilisable à l'exécution. En Java quand une classe référence une autre classe, cette référence est implantée par le stockage de la chaine de caractères du nom de la classe "appelée". Un fichier de classe Java contient des références symboliques (symbôles stockés sous forme de chaines de caractères) aux classes appelées, aux méthodes, aux arguments et à leurs types, aussi bien pour les classes "externes" au fichier, que pour les classes et méthodes internes, définies dans le fichier.

Tendances futures

La pénalité en performance des programmes Java actuels ne doit pas être considérée comme définitive : des techniques combinées de compilation "juste à temps" et de mesures de profils ("profiling") juste à temps pourraient permettre de s'approcher d'un rapport de pénalité de 2 ou 3 vis-à-vis d'un équivalent natif en C++. À ce niveau la pénalité sera indiscernable à l'utilisateur final pour de nombreuses applications.

De même dans le domaine des interfaces utilisateur, on commence à voir apparaitre des composants Java, fondés sur les API standard de Java (l'AWT) qui implémentent des fonctions d'interface non présentes dans l'AWT et qui les rendent ainsi disponibles sur toutes plateformes. On pourrait voir ainsi apparaitre une interface avec "l'aspect et le comportement" ("look and feel") de la plateforme Java, comme on a aujourd'hui "l'aspect Mac", "l'aspect Motif" ou "l'aspect Windows".

15.6 Indépendance vis-à-vis des plateformes

La raison clé du succès de Java dans un environnement distribué en réseau est le fait que le système Java permet de créer des binaires exécutables qui vont pouvoir tourner sur de multiples plateformes, tout en restant inchangés (compilés une seule fois).

Le système Java c'est l'ensemble du langage Java, de la machine virtuelle java (la JVM), de l'API Java et des fichiers de classes Java. L'architecture du "Système Java" permet d'écrire des programmes indépendants des plateformes, mais permet aussi d'écrire des programmes dépendants d'une plateforme. L'indépendance est une option.

Tous les composants du système Java (le langage, l'API, la JVM et les fichiers de classe) jouent un rôle dans l'obtention de binaires indépendants des plateformes. Mais le principal support provient de la plateforme Java : l'ensemble de la JVM et des API Java.

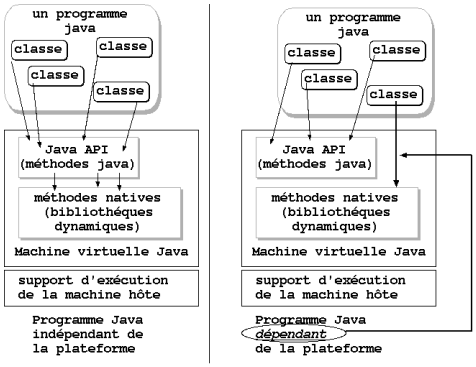


FIG. 159: Java : indépendance de la plateforme

Un programme Java qui n'interagit que avec les fonctions définies dans l'API Java pourra s'exécuter sur toute plateforme possédant une machine virtuelle java.

Le langage intervient dans l'indépendance en définissant l'étendue et le comportement de ses types primitifs. Alors que en C et C++ l'étendue ("range") d'un int est déterminée par sa taille, et sa taille est dépendante de la plateforme (4 octets sur certaines machines, 8 sur d'autres). Un int java est toujours un nombre signé sur 32 bits en complément à 2.

Les fichiers de classes définissent un format binaire spécifique de la machine virtuelle java, et non pas d'une plateforme particulière.

De plus, la plateforme java est extensible ("scalable") : elle peut être

implantée aussi bien sur de grosse machines que sur des microprocesseurs embarqués, comportant très peu de ressources.

Les versions de la plateforme Java

L'ensemble des bibliothèques de base de l'édition standard de la plateforme java est appelé "Java Core API" et sa disponibilité est garantie sur toute plateforme java de l'édition standard. Mais il existe des extensions et des restrictions.

extensions : des extensions appelées "Java Standard Extensions API" ont été définies pour la téléphonie, le commerce, l'audio, la vidéo, la 3D, ...

restrictions : il existe 3 éditions de la plateforme java, qui comportent de moins en moins de fonctions prédéfinies dans les API. Elles sont destinées à des environnements embarqués disposant de très peu de ressources :

- l'édition embarquée : "Java Embedded Platform",
- l'édition personnelles : "Java personal Platform",
- l'édition carte à puce : "Java Card Platform".

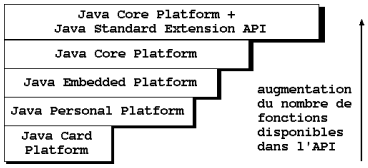


FIG. 160: Java : différentes éditions

Un autre facteur de complexité dans le support de l'indépendance des programmes est l'évolution des versions des APIs. Certaines fonctions peuvent être ajoutées, et (plus rarement) certaines devenir obsolètes. Un programme utilisant des fonctions récentes ne fonctionnera pas sur une machine virtuelle ancienne version. Les nouvelles versions de la machine virtuelles vont mettre un certain temps à se diffuser sur tous les ordinateurs connectés à un réseau.

L'utilisation des méthodes natives (ne faisant pas partie de l'API standard) est un autre facteur empêchant la portabilité. Si ces méthodes sont nécessaires et que l'on désire rester portable, il faut "porter" ces méthodes natives sur toutes les plateformes cibles.

Certains vendeurs fournissent également avec leur JVM des bibliothèques Java non standard qui viennent s'ajouter aux APIs standard de la JVM. Si ces méthodes Java sont implantées en utilisant uniquement des méthodes de l'API standard, elles pourront être téléchargées avec le programme pour permettre l'exécution de celui-ci. Il n'y a pas de perte de généralité, mais une pénalité sur le temps de chargement. Par exemple un programme Java qui utilise les AFC (Application Foundation Class) de Microsoft, ne pourra tourner sur Unix que si on charge avec lui les binaires des classes AFC utilisées par le programme.

La machine virtuelle étant implémentée différemment par différents vendeurs sur différentes machines, il est fortement conseillé, pour qu'un programme soit portable d'éviter les deux ecceuils suivants :

- ne pas dépendre de la vitesse de traitement pour assurer la correction du programme, (en particulier du moment où seront exécutés les routines de finalisation des objets non référençés, libérés par le ramasse miettes),
- ne pas dépendre de la priorité des threads pour assurer la correction du programme.

Ceci à cause de la liberté laissée aux implémenteurs dans la spécification du ramasse miettes et des threads.

Une partie très difficile d'une application Java qui doit fonctionner sur plusieurs plateformes est l'interface utilisateur. L'AWT ne contient qu'un ensemble de base de composants d'interface. Des bibliothèques telles que Microsoft AFC, Netscape IFC ou Sun JFC fournissent des composants plus élaborés. Mais créer une interface avec laquelle les utilisateurs des différentes plateformes se sentent tous à l'aise n'est pas une tâche aisée.

En conclusion, les programmes Java ne sont pas en général indépendants des plateformes à un degré tel que l'on puisse se contenter de mettre un programme au point sur une plateforme : il convient de le tester sur toutes les plateformes sur lesquelles on prétend que le programme fonctionne.

On peut fournir une liste d'étapes à suivre pour réaliser sans déboires un programme Java indépendant des plateformes :

1. choisir une liste de plateformes, sur lesquelles on va indiquer que le programme tourne,
2. choisir une version de la plateforme Java qui soit suffisamment diffussée sur ces systèmes,

3. sur chaque système cible, choisir une implémentation de la plateforme Java, sur laquelle on annoncera que le programme fonctionne ;
4. écrire le programme de telle sorte qu'il n'accède au système hôte que à travers l'API standard de Java,
5. écrire le programme de telle sorte qu'il ne dépende pas du moment d'exécution des routines de finalisation ni de la façon dont les priorités des threads sont gérées,
6. concevoir une interface utilisateur qui fonctionne correctement sur toutes les plateformes,
7. tester le programme sur toutes les plateformes cibles.

La politique de l'indépendance des plateformes

En plus de l'API standard, les vendeurs de JVM peuvent ajouter des extensions. Ces extensions peuvent être proposées pour "faciliter" la tâche du programmeur, mais aussi pour "vérouiller" celui-ci sur la plateforme du vendeur, et l'inciter à écrire des programmes qui ne fonctionnent que sur la plateforme du vendeur. Bien entendu, un vendeur concurrent va faire la même chose. Le développeur a alors le choix entre profiter des avantages des extensions ou faire l'effort de développer de façon indépendante et ne pas être lié à une plateforme unique.

15.7 La sécurité dans le système java

L'utilisation intensive des réseaux implique d'apporter une attention pqrticulière à la sécurité. Java ayant été conçu en vue d'une utilisation intensive sur des réseaux de machines diverses, a inclus dés sa conception un modèle de sécurité assez complet.

Le système Java inclut un modèle de sécurité qui a pour rôle premier de protéger l'utilisateur final de programmes hostiles qui seraient téléchargés à travers le réseau depuis une source non sûre, par exemple en chargeant une page html dans un navigateur. Pour cela Java fournit une "boîte à sable" ("sandbox") dans laquelle un programme Java non-sûr peut faire ce qu'il veut, mais dont il ne peut pas franchir les frontières. Cette "boîte à sable" interdit en particulier aux applets de :

– lire ou écrire sur le disque local,

– établir une connexion réseau vers toute machine sauf celle depuis laquelle elle vient ;
– créer un nouveau process,
– charger une nouvelle bibliohèque dynamique,
– appeler directement une méthode native.

Naturellement, si vous êtes raisonnablement paranoïaque, vous avez besoin d'être convaincu que la "boîte à sable" ("Sandbox") n'a pas de point de faiblesse avant de croire qu'elle peut protéger votre machine de tout programme hostile. Pour assurer cette absence de faiblesse, le modèle de sécurité Java implique tous les aspects de l'architecture.

Les composants responsables du bon fonctionnement de la "boîte à sable" sont :

– l'architecture du chargeur de classes,
– le vérificat eur de fichiers de classes,
– les fonctions de sécurité incluses dans la machine virtuelle et dans le langage,
– le gestionnaire de sécurité et l'API Java.

L'une des forces du modèle de sécurité Java est que deux de ses composants, le chargeur de classes et le gestionnaire de sécurite, sont paramétrables. Ceci permet de créer une politque de sécurité adaptée pour une application particulière.

L'architecture du chargeur de classes

Le chargeur de classes est le première ligne de défense. Il garde la frontière des bibliothèques sûres en empêchant les classes non-sûres ("untrusted") de prétendre à être considérées comme "sûres". Il réalise ceci en fournissant des espaces de nommage différents pour des classes chrgées par différents chargeurs de classes. Un espace de noms est un ensemble de noms uniques pour des classes chargées. Il est maintenu par la machine virtuelle.

Dans la machine virtuelle, des classes chargées dans des espaces de noms différents ne peuvent même pas détecter leur présence mutuelle, sauf si on fournit explicitement un mécanisme qui leur permet d'interagir.

Quand on écrit un chargeur de classes, on crée un environnement dans lequel s'exécute le code chargé. Pour éviter que cet environnement ne comporte des trous de sécurité, il faut respecter un certain nombre de règles quand on écrit le chargeur. En général on désire que le chargeur

protège les limites des classes reconnues sûres ("trusted"), par exemple l'API Java ou un package local.

Java permet aux classes d'un même paquetage ("package") de s'accorder l'une l'autre des droits d'accès particuliers qui ne sont pas accordés aux classes extérieures au paquetage. Or, rappelons nous que les classes se désignent (se référencent) par leur nom. Ainsi si une classe, chargée par le chargeur "mon_chargeur" reçoit une requête pour charger depuis le paquetage "inconnu" une classe qui se fait appeler "java.lang.virus", cette classe "virus" prétend par son nom faire partie de l'API Java. Si elle est chargée, elle se verra accorder les mêmes droits que ceux de l'API, ce qui n'est certainement pas ce que l'on désire. Le chargeur "mon_chargeur" doit donc être écrit pour empêcher cela.

Pour cela, le chargeur doir refuser de charger les classes qui prétendent faire partie d'un paquetage "reconnu sûr", mais qui ne sont pas stockées dans le container local lui-même reconnu sûr ("trusted local repository"). Le chargeur "mon_chargeur" doit alors utiliser la méthode suivante :

1. le chargeur personnel passe d'abord toutes les requêtes de chargement classes au chargeur primordial,
2. le chargeur primordial cherche dans l'entrepôt local "reconnu sûr" la classe demandée ; si elle est trouvée, c'est elle qui est fournie (ainsi un paquetage malicieux ne peut pas remplacer une classe d'une API connue). Si la classe n'est pas trouvée, le chargeur primordial renvoie un statut d'échec au chargeur personnel ;
3. le chargeur personnel doit alors vérifier si la classe demandée ne prétends pas faire partie d'un paquetage reconnu sûr (si elle le prétend et qu'elle en fait vraiment partie, alors le chargeur primortdial l'aurait trouvée) ; Si la classe le prétend, alors le chargeur personnel doit lever une exception de sécurité ("throw security exception") ;

Autre exemple : si vous ne voulez pas qu'une classe chargée par le chargeur personnel puisse charger une classe d'un paquetage local "mon_prog", le chargeur personnel doit vérifier les requêtes et lever une exception si l'une d'elle est adressée à une classe de "mon_prog" par une classe chargée par "mon_chargeur".

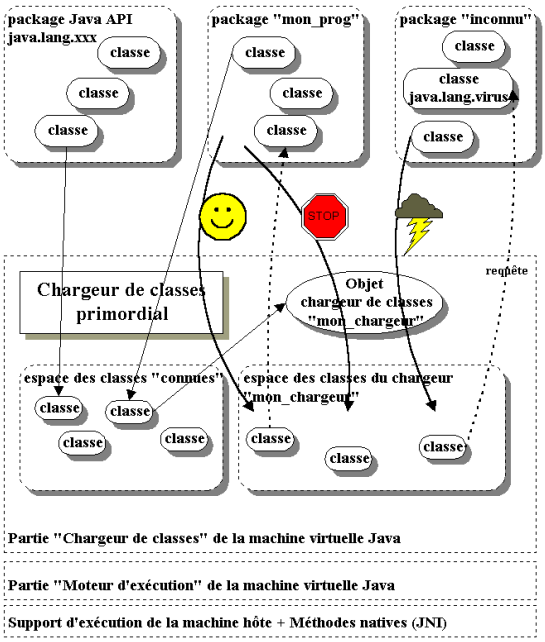


FIG. 161: Sécurité dans le chargeur de classes

Comme c'est par le nom complet de la classe (tel que "java.lang.virus") que le chargeur peut savoir si la classe demandée est ou prétend être une classe d'un paquetage protégé, le chargeur devra être pourvu d'une liste des noms des paquetages protégés, restreints, interdits. Le chargeur devra alors :

1. si la classe demandée est dans un paquetage "interdit", lever une exception de sécurité, sinon aller en 2,
2. passer la requête au chargeur primordial ; si celui-ci fournit la classe, on continue, sinon on va en 3,

3. s'il existedes paquetages sûrs auxquels le chargeur n'est pas autorisé à ajouter des classes, et que la classe demandée fait partie d'un tel paquetage, il doit lever une exception, sinon aller en 4,
4. le chargeur personnel charge la classe selon sa méthode (par exemple téléchargement à travers un réseau) ; si c'est un succès, il retourne la classe, sinon il retourne le statut "no class definit ion found".

Le vérificateur de fichiers de classes

Le vérificateur de fichiers de classes s'assure que les fichiers binaires ".class" contenant le bytecode exécutable ont une structure correcte. En effet, quand une classe est téléchargée depuis un réseau, on a aucun moyen de savoir si ce fichier a été créé par un compilateur java "normal" ou bien "forgé" par un pirate dans le but de compromettre le bon fonctionnement de la machine virtuelle.

Phase 1 : vérifications internes

Durant cette phase, on vérifie tout ce qu'il est possible de vérifier en regardant uniquement le contenu du fichier de classe. La première vérification concerne la robustesse : si une classe erronée contient une instruction de branchement qui saute en-dehors du code de la classe, son exécution pourrait conduire à un crash de la machine virtuelle. La JVM vérifie donc que toutes les instructions de branchement pointent sur une instruction valide à l'intérieur de la classe.

On va d'abord vérifier la structure : la classe doit commencer par les 4 octets du "nombre magique" : 0xCAFEBABE. Puis on teste si la classe n'est pas tronquée et si elle ne comporte pas d'octets supplémentaires. Ceci est possible car chaque composant d'un fichier de classe doit indiquer son type et sa taille. Ensuite on vérifie que chaque composant est une instance bien formée de son type. Par exemple, on vérifie que chaque descripteur de méthode est une chaine de caractère bien formée conforme à la grammaire de description des paramètres d'une méthode.

Le vérificateur s'assure aussi que la classe adhère à toutes les contraintes qui lui sont imosées par la spécific ation du langage Java. Par exemple, toute classe sauf la classe object doit avoir une super-classe.

Ensuite la JVM opère une analyse de flux de la chaine de bytecodes constituant les méthodes de la classe. Le flux de bytcodes est constitué d'une série de code opérations ("opcodes") de 1 octet, chacun suivi de 0,

1 ou plusieurs opérandes. L'activité consistant à exécuter les bytecodes l'un aprés l'autre constitue un flot d'exécution ("thread") à l'intérieur de la JVM. Chaque flot possède sa propre pile java, constituée de formes ("frame"). Chaque invocation d'une méthode possède sa propre forme qui est une section de mémoire pour les variables locales et les résultats intermédiaires. La partie de la forme où sont rangés les résultats intermédiaires est appelée "pile d'opérandes" ("operand stack"). Un bytecode peut utiliser des données stockées dans ses opérandes, dans les variables locales de la forme et dans la pile d'opérandes.

Le vérificateur de bytecode va faire un grand nombre de tests : que aucune variable locale n'est accédée avant d'être définie, que les champs d'une classe sont remplis avec des valeurs du bin type, que les méthodes sont invoquées avec le bon nombre d'arguments et que ceux-ci sont du bon type, ...

Phase 2 : vérification des références symboliques

La phase 1 est exécutée une fois, dés le chargement de la classe dans la JVM. La phase 2 est reportée juste avant l'exécution. Durant la phase 2, la JVM suit toutes les références contenues dans la classe à vérifier jusqu'aux classes référencées, pour être sur qu'elles sont correctes. Cette phase 2 est en fait une partie du processus de liaison dynamique.

La liaison dynamique est le processus qui va résoudre les références symbolique (une classe définie par son nom) en des références directes : la JVM va (1) trouver les classes référencées (et les charger si nécessaire), (2) remplaçer la référence symbolique par une référence directe (un pointeur). La JVM garde une table de correspondance des références symboliques et directes de façon à ne pas faire deux fois le travail de recherche si une classe est utilisée plusieurs fois. Quand c'est un champ ou une méthode d'une classe qui est référençée, la JVM vérifie l'existance de ce champ ou de cette méthode avant de remplacer le symbôle par un lien direct.

Références symboliques et compatibité binaire

– Quand une classe référence un champ ou une méthode d'une classe en bibliothèque, il peut arriver que l'on doive modifier la classe référencée. Bien sûr, le compilateur n'a pas de moyen de savoir quelles classes possèdent une référence vers la classe qu'il est en train de recompiler ; il ne peut donc pas faire une recompilation automatique

de toutes les classes utilisatrice de la classe de bibliothèque qui vient d'être modifiée. On distingue alors deux types de modifications : celles qui sont compatibles binaires, et celles qui ne le sont pas.
– Par exemple si une classe "a" contient la méthode "m1", référencée par une classe de "b", et que l'on change le nom de "m1" en "m2", l'invocation "a.m1()" va échouer. Cette modification n'est pas compatible binaire. Par contre, si on ajoute la méthode "m2" à "a" et que l'on modifie le contenu de la méthode "m1" en lui faisant appeler "m2", mais sans changer ni son nom ni sa liste d'arguments, l'invocation "a.m1()" continuera à fonctionner : c'est une modific ation compatible binaire. Dans ce dernier cas, il n'est pas nécessaire de recompiler toutes les classes utilisatrices de la bibliothèque pour qu'elles continuent à fonctionner correctement.
– Le langage Java spécifie les modifi cations que l'on peut faire à une classe, tout en lui conservant sa compatibilité binaire.

Les fonctions de sécurité incluses dans la JVM

La machine virtuelle n'accorde aux programmes que des moyens sûrs et structurés d'accéder à la mémoire :
– conversions de type vérifiées et sûres,
– accès mémoires structurés (pas d'arithmétique des pointeurs),
– ramasse miette automatique (pas de "de-alloc" intenpestif),
– test des bornes des tableaux (pas d'accès à Tab[-10], ...),
– test des références vis-à-vis de null (pas d'accès à travers un pointeur non initialisé).

L'interdiction de tout accès mémoire non structuré empêche un programme malicieux d'aller lire ou écrire n'importe où dans la mémoire gérée par la machine virtuelle. Ceci est non seulement imposé par les règles du langage, mais est aussi inlclut dans la définition des codes opératoires des bytecodes. Ainsi même en écrivant "à la main" une séquence de bytecodes, il est peu probable de pourvoir accéder en dehors des zones mémoires "officiellementé déclarées par la classe. Ceci constitue une solide barière contre la manipulation imprévue de la mémoire de la JVM.

Les barières de sécurité de la machine virtuelle peuvent cependant être pénétrées par l'utilisation des méthodes natives. Celles-ci sont en effet écrites en C ou C++ avec tous les défauts de contrôle d'accès à la mémoire de ces langages. Pour cette raison, le gestionnaire de sécurité dis-

pose d'une méthode dont le rôle est de dire si un programme sera autorisé à charger une bibliothèque dynamique, ce qui est nécessaire avant d'invoquer une méthode native. Les applers, par exemple, ne sont pas autorisés à charger une bibliothèque dynamique, et donc ne peuvent pas installer leur propres méthodes natives. Ils peuvent cependant appeler une méthode native de l'API Java. Si cette méthode contient un trou (sous forme d'accès non prévu à la mémoire), la sécurité de la JVM est menaçée.

Le mécanisme final de sécurité est le traitement des exceptions, particulièrement des exceptions de sécurité : quand un flot lève une telle exception, c'est ce flot qui doit être arrêté, mais pas la machine virtuelle toute entière.

Le gestionnaire de sécurité et l'API Java

Les chargeurs de classe permettent de protéger entre elles des classes chargées dans la machine virtuelle. Mais pour protéger les éléments extérieurs à la JVM on a besoin du gestionnaire de sécurité : celui-ci va définir (de façon paramétrable) les limites extérieures du "bac à sable" dans lequel évoluent les programmes Java.

Pour chaque action potentiellement non-sûre ("unsafe"), l'API Java interroge le gestionnaire de sécurité. Celui-ci posséde, pour chaque action de ce type, une méthode qui indique si l'action est permise depuis le "bac à sable". Le nom de chacune de ces méthodes commence par "check" : par exemple checkRead() indique si le flot appelant est autorisé à lire un fichier, etc ... Avanr d'essayer une action "sensible" l'API Java appelle la méthode "check" correspondante du gestionnaire de sécurité. Soit celle-ci autorise, elle fait alors un simple "return", soit elle refuse, elle lève alors une exception.

Les principales actions vérifiées avant exécution sont :
– accepter une connexion socket depuis un hôte et un port spécifiés,
– modifier les caractéristiques d'un flot (changer de priorité, arrêter, ...),
– ouvrir une connexion socket vers un hôte et un port spécifiés,
– créer un nouveau chargeur de classe,
– détruire un fi chier spécifié,
– créer un nouveau process,
– sortir de l'application ;
– charger une bibliothèque dynamique contenant des méthodes na-

tives,
- attendre une connexion sur un port local spécifié,
- charger une classe depuis un paquetage spécifié (concerne les chargeurs de classe) ;
- ajouter une nouvelle à un paquetage spécifié (concerne les chargeurs de classe),
- accède ou modifie certaines propriétés du système d'exploitation,
- lire un fichi er spécifié,
- écrire dans un fichier spécifié.

Il reste, dans la version courante (1.1) deux actions potentiellement dangereuses qui ne sont pas vérifiées :
- allouer de la mémoire jusqu'à ce qu'il n'en reste plus,
- lançer de nouveaux flots(threads) jusqu'au nombre maximum possible.

Ces deux actions de type "déni de service" peuvent rendre un navigateur inutilisable par manque de ressources pour continuer normalement.

D'autre types d'applets hostiles peuvent :
- envoyer des e-mails intenpestifs depuis la machine de l'utilisateur,
- provoquer des bruits désagréables, même aprés que l'on ai quitté la page web,
- afficher des images ou des animations "choquantes".

Bien qu'on ne puisse installer qu'un seul gestionnaire de sécurité, on peut lui faire gérer plusieurs politiques de sécurité. Il est en effet possible de savoir, quand une méthode "check" est appelée, si la classe à l'origine de l'appel a été chargée par un chargeur ajouté, et si oui, par lequel. On peut aussi savoir d'où venait la classe qui a provoqué la vérification.

15.8 Mobilité du code dans le système Java

L'une des raisons fondamentales de l'importance et du succès de Java est sa capacité à produire du logiciel **mobile sur un réseau**. Cette capacité a été considérée par beaucoup comme un saut technologique.

Évolution de l'informatique et importance de la mobilité du code

L'informatique a subit de grandes évolutions à travers une série d'étapes et de modèles d'utilisation, liés aux évolutions de la technique, en parti-

culier à celle des semi-conducteurs et des circuits intégrés (VLSI) :
- partage d'une machine centrale par de nombreux utilisateurs,
- utilisation de machines personnelles isolées et indépendantes,
- mise en réseau des machines personnelles et connexion à une machine centrale, avec un développement d'applications client-serveur 2 tiers (1 client - 1 serveur),
- généralisation de l'utilisation des réseaux, toutes les machines sont connectées, les applications deviennent plus complexes et évoluent vers le client-serveur 3 tiers (1 client sur une machine personnelle, 1 serveur intermédiaire responsable de traitement de données et 1 serveur "d'arrière plan" responsable du maintient de bases de données communes),
- les applications sur un ensemble de machines connectées effacent peu à peu la différence entre clients et serveurs (chaque partie jouant tour à tour l'un ou l'autre rôle) pour devenir de véritables applications distribuées ;
- la mobilité du logiciel sur le réseau représente donc une étape de plus dans l'évolution du modèle d'utilisation de l'informatique.

Cette évolution ne s'est pas faite sans inconvénients. Du côté des utilisateurs le côté positif est largement dominant : ceux-ci peuvent disposer d'applications mieux adaptées à leurs besoins et à leur façon de travailler. L'inconvénient est toutefois une complexité plus grande. Du côté des administrateurs de systèmes on peut raisonnablement affirmer que le bilan ne comporte que des aspects négatifs : la gestion de centaines de machines diverses à qui on doit faire exécuter correctement les mêmes programmes est un véritable cauchemar du point de vue de l'administration système. Ceci est fortement agravé par le caractère très primitif, sur le sujet de l'administration système, des systèmes d'exploitation disponibles sur les ordinateurs personnels.

C'est pourquoi, l'apparition d'un "système logiciel" permettant de rendre un nouveau programme disponible à l'identique pour des centaines, voire des milliers de machines, en ne l'installant que sur un seul serveur a été accueillit par les administrateurs de systèmes et de réseaux comme un quasi miracle. En 1998, on attend de voir si le miracle a une petite chance de se réaliser.

L'évolution du modèle d'utilisation vers des ordinateurs personnels a été rendue possible par la révolution des microprocesseurs, avec leur capacité en augmentation rapide et leurs prix, simultanément, en baisse ra-

pide. Parallèlement, caché sous l'évolution des logiciels vers de plus en plus de mobilité et de connexion en réseau, se cache les capacités croissantes et les prix en baisse de la bande passante des réseaux de télécommunication. L'augmentation rapide de la bande passante solvable permet d'envisager de faire passer de nouveaux types d'information sur les réseaux informatiques : du son, de la voix, de la vidéo, des applications informatiques complètes, de la gestion d'interface utilisateurs, le tout résultant en un outil fortement interactif.

Dans ce nouveau modèle de code mobile, le code et les données perdent leur distinction pour devenir du contenu. Avant cette évolution, les utilisateurs étaient contraints à penser l'utilisation de l'informatique en terme de logiciels et de versions de ces logiciels. Ces versions évoluant, ils avaient à décider quand passer à la version suivante et installer la ou les nouvelles versions de leurs logiciels, afin de disposer d'outils à jour. Avec le nouveau modèle l'utilisateur est poussé à penser en termes de contenu des services.

Le logiciel est alors fournit comme un fluxauto-évolutif faisant partie d'un contenu interactif. On n'a plus besoin de le maintenir, il suffit de l'utiliser. Le logiciel prends alors l'aspect d'un bien de consommation ("a software appliance").

De nombreux services au contenu auto-évolutif vont (ou devront) partager avec les biens de consommation ménagers deux caractéristiques : une fonction dédiée et une interface simple. Quand vous décidez d'utiliser un grille-pain, vous ne vous attendez pas à devoir lire un manuel au préalable ! De même la fonctionalié de nombreux services de contenu devra être aussi dédiée (1 seul objectif) et l'utilisation aussi simple.

Un autre exemple est celui d'une page web : d'un certain point de vue, la page html ressemble à un programme avec des instructions et des paramètres, d'un autre point de vue, elle ressemble à des données. La distinction entre données et programme s'est estompée. Mais les personnes qui naviguent sur le web s'attendent à ce que les pages évoluent, sans qu'il soit besoin d'actions délibérées de leur part (tel que installer la nouvelle version !).

Le support de l'architecture Java pour la mobilité du code

Le support de l'architecture Java pour la mobilité du code commence avec le support de l'indépendance vis-à-vis des plateformes et celui de la

sécurité. Bien que non strictement nécessaires, ces deux éléments aident à rendre praticable la mobilité du code. Le support de la mobilité est concentré sur le temps nécessaire à charger les codes binaires à travers le réseau. Java réponds à ce problème en rejetant le modèle de programme traditionnel (un gros fichier monolithique contenant tous les binaires suceptibles d'être utilisés par le programme). À la place, Java découpe le programme en un grand nombre de petites unités : les fichiers de classes Java.

Les fichiers de classes voyagent sur le réseau indépendamment les uns des autres. De plus comme Java supporte la liaison dynamique, l'utilisateur n'a pas besoin d'attendre que tout le programme soit chargé pour commençer son exécution. Le programme démarre dès que la première classe arrive.

L'exécution d'un programme Java débute à la méthode main() de l'une des classes. Les autres classes sont chargées et reliées dynamiquement au fur et à mesure que l'application en a besoin. Si une classe n'est jamais effectivement utilisée dans une exécution particulière du programme, elle n'est jamais téléchargée.

En plus de cette liaison dynamique, qui permet de réduire le traffic sur le réseau, les fichiers de classe sont conçus pour être compacts. Bien que ceci conduise à de petits fchiers, le foncionnement du protocole http 1.0 fait que chaque fichier de classe d'une applet est l'objet d'une requête http individuelle. La majeure partie du temps d'attente est alors occupée par les protocoles d'échange ("handshacking") pour chaque requête de fichier. Pour cette raison la version 1.1 de Java a défini des fichiers "JAR" (Java ARchive) capables de contenir plusieurs fichiers de classes, à envoyer à travers le réseau en une seule requête, ce qui réduit fortement le surcoût du traitement des requêtes. Plus, les données à l'intérieur du fichier "JAR" peuvent être compréssées, ce qui réduit encore le temps de transit.

L'Applet, un exemple de Java mobile sur un réseau

Java a été adopté si vite et si largement, non seulement parce qu'il est apparu la bonne technologie au bon moment, mais aussi parcequ'il a bénéficié d'un bon marketing.

1. il avait un nom facile à retenir ("cool" !),

2. il était, dans la pratique, gratuit (toujours un bon point pour un acheteur !),
3. il a été accroché au web au bon moment : juste quand Netscape cherchait à transformer "Navigator" d'un visualiseur d'hypertexte graphique en une plateforme de programmation à part entière.

... et la clé de ce succès a été la création d'une forme spéciale de programme Java, destinée à s'exécuter à l'intérieur d'un navigateur : l'applet.

L'applet Java exhibe tous les avantages de l'aspect "orienté réseau" de Java : indépendance de la plateforme, mobilité et sécurité. L'indépendance vis-à-vis de la plateforme a été l'une des principales motivations de la création du web, et l'applet Java s'y conforme exactement : un applet Java peut s'exécuter sur n'importe quelle plateforme sur laquelle un navigateur avec une machine virtuelle java y est installé.

Un navigateur exécute un appler java quand il rencontre une balise <applet ...> ... </applet> dans une page html. Cette balise fournit suffisament d'information au navigateur pour lui permettre d'afficher l'applet. Le navigateur passe l'information à la machine virtuelle Java qui initialise l'applet en appelant la méthode init() de la classe de démarrage de l'applet. Une fois l'initialisation de l'applet terminée, celui-ci apparait comme faisant partie de la page web.

15.9 La Machine Virtuelle Java (JVM)

L'expression "Machine Virtuelle Java" peut, selon le contexte, désigner trois choses différentes :
- la **spécification** de la Machine Virtuelle Java,
- une **implémentation** de la JVM,
- une **instance** d'exécution de la JVM.

La durée de vie d'une instance de JVM

Une instance de JVM a une mission claire : exécuter une application Java. L'instance nait quand une application est lancée, elle meurt quand l'application se termine. Si on lance simultanément 3 applications Java sur la même machine, chacune d'elles va s'exécuter dans sa propre instance de l'implémentation locale de la machine virtuelle.

Exemple


```
class Echo {
    public static void main (String[] args) {
          int len = args.length ;
          for (int i = 0; i < len; ++i) {
             System.out.print ("arg[" + i + "]= ");
             System.out.print (args[i] + " ") ;
          }
          System.out.println() ;
 }
}
```

Cette application va imprimer les arguments trouvés sur sa ligne de commande.

Elle est construite pour être lancée depuis une ligne de commande du système, **mais pas** depuis un navigateur.

La méthode **main(), doit** être "public static void" et prendre un String comme unique paramètre. Pour lancer cette application, il faut donner à l'instance de la machine créée le nom de la classe initiale (celle qui contient la méthode main()).

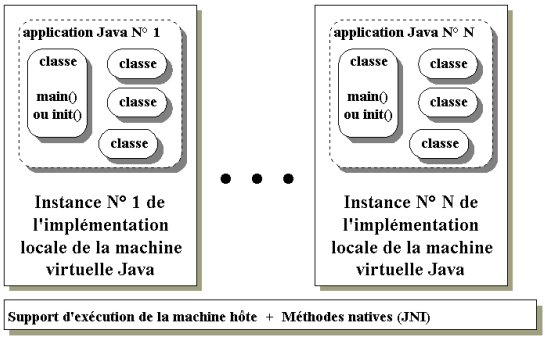


FIG. 162: La JVM

Par exemple : java Echo Hello World. La commande "java" lance une instance de la JVM, le premier argument "Echo" lui donne le nom de la classe initiale (il doit y avoir dans le répertoire courant un fichier Echo.class),

les deux mots suivants sur la ligne sont considérés comme des arguments qui seront passés à main() dans le String "args".

Sous Win95, avec le JDK de Sun, il faut ajouter l'emplacement du JDK dans le PATH (dans le fichier autoexec.bat) :

```
PATH C:\WINDOWS;C:\WINDOWS\COMMAND;C:\;C:\DOS;C:\JDK1.1.4\BIN
```

On crée un fichier Echo.java contenant le source java. Le source est compilé par le compilateur "javac", et le résultat (le bytecode) stocké dans le fichier Echo.class

La commande "java Echo" va chercher dans le répertoire courant un fichier Echo.class qui doit contenir la classe "Echo", et l'exécuter.

Attention ! Les noms du fichier et de la classe doivent être identiques et respecter la casse.

```
>javac Echo.java
>java Echo Hello World.
arg[0]= Hello arg[1]= World.
```

L'architecture de la JVM

Dans la spécification de la JVM, son comportement est décrit en termes de sous-systèmes, zones mémoire, types de données et instructions. Le but est de décrire strictement le comportement des implémentations. La figure suivante décrit l'architecture de la JVM.

Parmi les zones mémoires d'une instance de machine virtuelle (donc pour une application java), certaines sont partagées par tous les flots(threads), par exemple la zone des méthodes (le code exécutable) et le tas (heap) où sont stcokés les objets créés pendant l'exécution. Chaque flot créé possède sont propre registre et sa propre pile jave (contenant l'mpilement des formes correspondant aux invocations de méthodes).

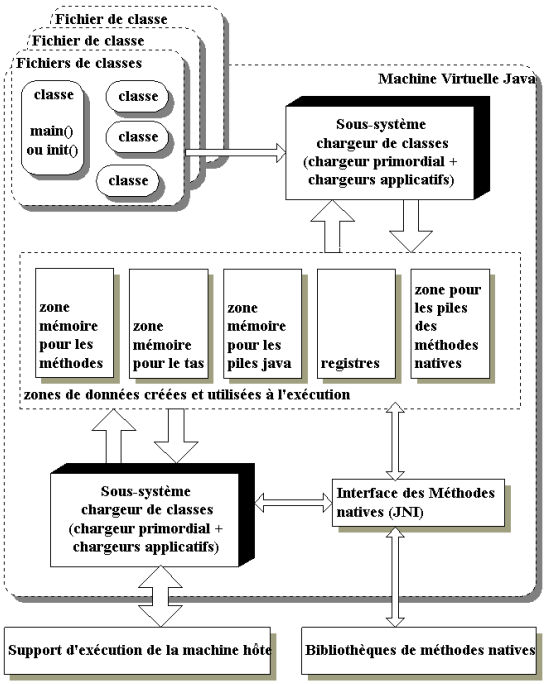


FIG. 163: Architecture de la JVM

Les piles java sont constituées d'une chaine de "formes" (stack frames), chacune contenant l'état d'une invocation de méthode java (l'équivalent d'un appel de fonction C). Chaque flot, n'a qu'un seul registre, le "pointeur courant" ("PC") indiquant la prochaine instruction à exécuter par le flot. Il n'y a pas de registres de données stockant des variables temporaires ou locales. Toutes les variables locales sont mises sur la pile java associée au flot. Ce choix a été fait pour simplifier la liste d'instructions de

la machine virtuelle et permettre son implémentation dans de très petites machines embraquées (e.g. JavaCard).

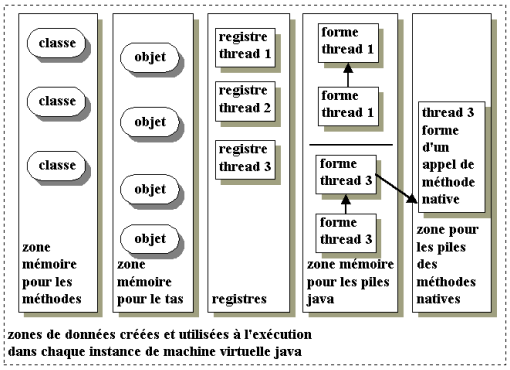


FIG. 164: Zones mémoire de la JVM

Les types de données de la JVM

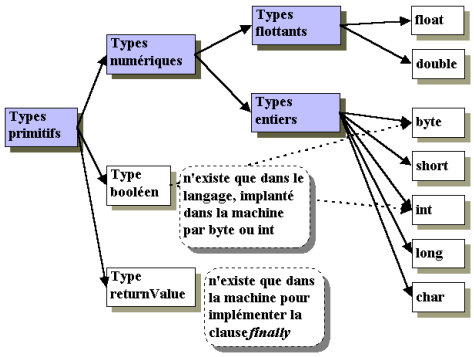


FIG. 165: Les types de données de la JVM

Dans la spécification de la JVM, les types de données et les opérations sur ces types sont strictement définies. Il y a deux grandes catégories de types : les types primitifs et les types référence. Les types primitifs sont décrits ci-dessous, ils existent à la fois dans la spécification du langage et dans celle de la machine virtuelle, sauf 2 exceptions annotées dans le tableau :

Les types "référence" contiennent des "pointeurs" vers des objets créés dynamiquement.

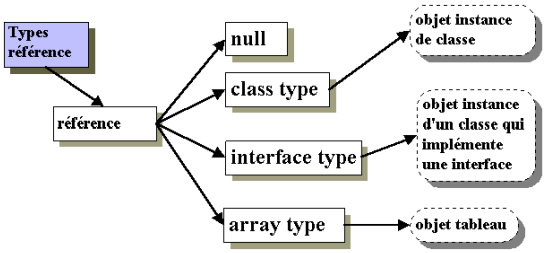


FIG. 166: les types référence de la JVM

La spécification de la JVM défi nit l'étendue des valeurs prises par chaque type de données, mais elle ne définit pas sa taille en mémoire. Cette dernière n'est pas accessible au programmeur et n'a pas d'influence sur le comportement du programme (il est choisit par l'implémenteur pour le meilleur compromis coût/efficacité sur le matériel considéré). Le tableau ci-dessous donne les étendues ("range") des types primitifs :

La spécification Java impose que les types de données de base aient toujours les définitions du tableau, quelque soient la taille du mot mémoire de la machine réelle qui implémente la machine virtuelle.

Type	Étendue (range)
byte	complément à 2 signé 8 bits (-2 7 à +2 7-1 inclus)
short	complément à 2 signé 16 bits (-2 15 à +2 15-1 inclus)
int	complément à 2 signé 32 bits (-2 31 à +2 31-1 inclus)
long	complément à 2 signé 64 bits (-2 63 à +2 63-1 inclus)
char	charactère UNICODE, non-signé 16 bits (0 à 2 16-1 inclus)
float	flottantsimple précision 32 bits IEEE-754
double	flottantdouble précision 64 bits IEEE-754
reference	null ou référence vers un objet sur le tas
returnValue	adresse d'une instruction bytecode à l'intérieur de la même méthode

SR03 2004 - Cours Architectures Internet - Le système Java
Fin du chapitre.
©Michel.Vayssade@utc.fr – Université de Technologie de Compiègne.

.

Ch16 - Programmation Java

- Application java et applet java
- Exemples d'applets java
- Java : gestion d'évènements.
- Java : Event Listeners dans toolkits awt et swing

Application java et applet java

Le système Java permet de créer deux types de "programmes" :
- une "application java",
- une "applet java".

Nota : on verra plus loin d'autres façons d'exécuter du code Java : les JSP, les Servlets, les Beans et les EJBs. Toutefois dans ces quatres cas, il s'agit de portions de codes s'exécutant dans le cadre d'une **application**. La différence entre application et applet est essentiellement une différence **système** : quel est le process dans le cadre duquel le code s'exécute.

Une **application** est un programme java "standalone". Elle s'exécute dans le cadre d'une JVM instanciée dans un process indépendant, créé par le système d'exploitation. Elle s'exécute indépendam-

ment d'un navigateur.

Une **applet** est un programme java téléchargeable depuis un navigateur. Il s'exécute dans le cadre d'une instance de JVM créée par le navigateur.

application		
source	**compilation**	**exécution**
hello.java	javac hello.java	java hello
applet		
source	**compilation**	**exécution**
HelloWorld.java	javac HelloWorld.java	appel par browser

Exemple d'appel d'applet dans une page html :

```
<APPLET CODE="HelloWorld" WIDTH=150 HEIGHT=25></APPLET>
```


application java

```
(1) créer source :

367 lo33 ultra:~/public_html/java> more hello.java

class HelloWorldApp {
   public static void main(String[] args) {
        System.out.println("Hello World!"); //Display string.
   }
}

(2) compiler : javac hello.java

925 lo33 ultra:~/public_html/java> java hello
```



```
Can't find class hello

mv hello.class --> HelloWorldApp.class

le fichier doit avoir le nom de la classe

(3) exécuter :

919 lo33 ultra:~/public_html/java> java HelloWorldApp
Hello World!
```

applet java

```
(1) créer source
```



```
368 lo33 ultra:~/public_html/java> more HelloWorld.java

import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello world!", 50, 25);
    }
}

(2) compiler : javac HelloWorld.java

(3) créer un fichier HTML qui appelle l'applet compilé
```



```
926 lo33 ultra:~/public_html/java> more ../hel-java.html
<HTML>
<HEAD><TITLE>Un exemple d'applet java</TITLE></HEAD>
<BODY>
L'applet apparait ici (cadre gris):
<APPLET CODE="HelloWorld" WIDTH=150 HEIGHT=25>
</APPLET>
</BODY></HTML>

(4) l'exécution se fait quand on affiche le fichier
    HTML dans un browser

Note : l'applet (le fichier HelloWorld.class) doit se
```



```
====   trouver dans le même répertoire que le fichier HTML,
       sauf si on indique un autre répertoire dans le tag :
        <APPLET CODE=AppletSubclass CODEBASE=aURL
                WIDTH=anInt HEIGHT=anInt>
        </APPLET>

Note : avec le JDK, Sun fournit l'application "appletviewer":
====
930 lo33 ultra:~/public_html> appletviewer hel-java.html &
[5] 8079
```

Le point d'entrée de toute application java est la méthode "main" : public static void main(String[] args) ...

Toute application java doit comporter une méthode "main". C'est

elle qui "prends la main" et lance l'exécution d'autres méthodes.

Si main est absente, erreur :
In class Machin : void main(String argv[]) is not defined

main est déclarée :

- **public :** peut être appelée par tout objet,
- **static :** c'est une "class method",
- **void :** ne retourne aucune valeur.

Les arguments de la méthode main : un seul tableau "args" contenant "n" chaines :

```
public class Echo {
     public static void main (String[] args) {
         for (int i = 0; i < args.length; i++)
             System.out.println(args[i]);
```



```
         }
     }
```

Seuls les arguments sont passés (pas le nom du programme) :

```
> java mon-appli arg-1 arg-2

args.length    contient 2
args[0]        contient la chaine "arg-1"
args[1]        contient la chaine "arg-2"
```

Toute applet java doit comporter l'une des trois méthodes "**init**" , "**start**" ou "**paint**" : c'est elle qui "prends la main" et lance l'exécution d'autres méthodes.

Paramètres d'une applet : ils sont aux applets ce que les arguments sont aux applications.


```
  <APPLET CODE=AppletSubclass.class WIDTH=anInt HEIGHT=anInt>
  <PARAM NAME=parameter1Name VALUE=aValue>
  <PARAM NAME=parameter2Name VALUE=anotherValue>
  </APPLET>
ex:
     <APPLET CODE="Animator.class" WIDTH=460 HEIGHT=160>
     <PARAM NAME="imageSource" VALUE="images/Beans">
     <PARAM NAME="backgroundColor" VALUE="0xc0c0c0">
     ......
     </APPLET>
```

Remarque : les "....." ci-dessus, peuvent contenir un code alternatif HTML, destiné aux navigateurs qui ne comprennent pas le tag <applet>.

Exemple de lecture de paramètres :

```
        <param name=MESS value="arg from html">
        <param name=X value=10>
lu par :
message = getParameter("Mess");
x = Integer.parseInt(getParameter("X"));
```

Utilisation des classes et objets

En java une méthode ou une variable n'existe que dans une classe ou un objet (objet = instance d'une classe).

Tout programme java doit donc définir une ou des classes :
class ma_classe ...

La ligne : System.out.println("Hello World !") ; montre l'utilisation d'une variable de classe et d'une méthode d'instance.

D'une variable de classe : System.out
"out" est une variable de classe, associée à une classe et non pas à une instance d'une classe.

De même on peut créer des méthodes de classe :

- variable de classe appelée par : nom_classe.nom_var
- méthode de classe appelée par : nom_classe.nom_method

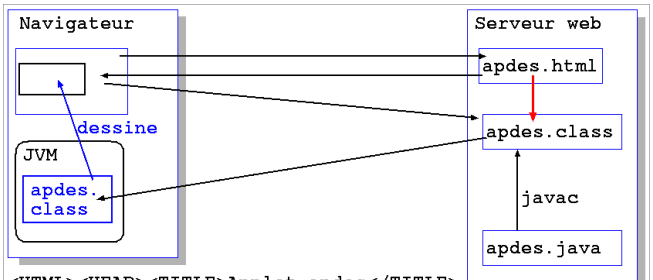
Les méthodes et variables qui ne sont pas "de classe" sont des méthodes d'instance et des variables d'instance : pour les utiliser, il faut y faire référence **depuis un objet**.

Explication de la ligne de code citée plus haut :
Quand la classe "System" est chargée dans l'application, elle instan-

cie la classe "PrintStream" et assigne le nouvel objet "PrintStream" à la variable de classe "out".

Comme on dispose d'une instance de "PrintStream", représentée par l'objet "System.out", on peut appeler l'une de ses méthodes d'instance, ici "println" en postfixant la référence d'objet par le nom de la fonction : System.out.println().

Exemples d'applets : apdes.java




```
<HTML><HEAD><TITLE>Applet apdes</TITLE>
</HEAD><BODY>
<h3>Applet apdes : dessin sur le
    "contexte graphique"</h3>
<APPLET code="apdes" width=500 height=300>
</APPLET>
</BODY></HTML>
```

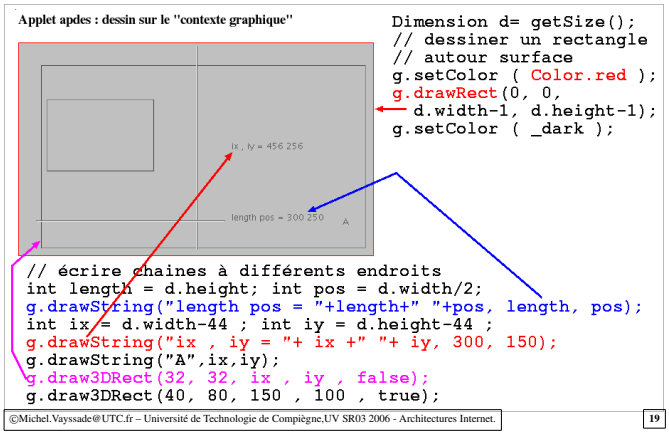


```
/* Exemple d'applet simple utilisant objet Graphics: apdes.java */
import java.applet.*;
import java.awt.*;

public class apdes extends Applet {
        private Color _light, _dark ;

   public void init() {
        _light = getBackground().brighter().brighter();
        _dark  = getBackground().darker().darker();
        Graphics g = this.getGraphics();
        paint(g);
   }
   public void paint ( Graphics g)
```

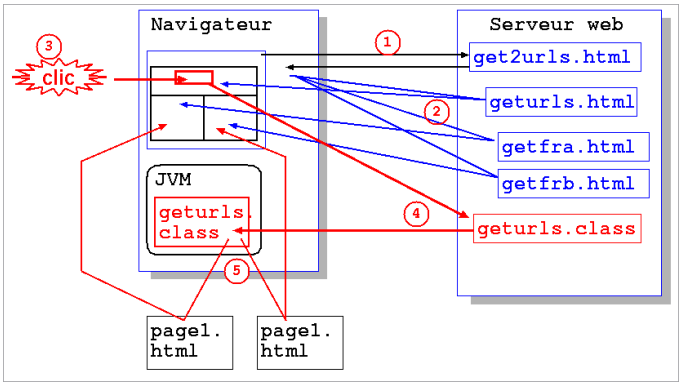
```
{     Dimension d= getSize();
      // dessiner un rectangle autour surface
      g.setColor ( Color.red );
      g.drawRect(0, 0, d.width-1 , d.height-1);
      g.setColor ( _dark );

      // écrire chaines à différents endroits
      int length = d.height; int pos = d.width/2;
      g.drawString("length pos = "+length+" "+pos, length, pos);
      int ix = d.width-44 ; int iy = d.height-44 ;
      g.drawString("ix , iy = "+ ix +" "+ iy, 300, 150);
      g.drawString("A",ix,iy);
      g.draw3DRect(32, 32, ix , iy , false);
      g.draw3DRect(40, 80, 150 , 100 , true);
```



```
      // dessiner une ligne verticale en relief
      g.setColor ( _light );
      g.drawLine (pos, 5, pos, length-10);
      g.setColor ( _dark );
      g.drawLine (pos+1, 5, pos+1, length-10);
      // dessiner une ligne horizontale en creux
      g.setColor ( _dark );
      g.drawLine (25, pos, length-10, pos);
      g.setColor ( _light );
      g.drawLine (26, pos+1, length-11, pos+1);
   }//fin paint
}//fin class apdes
```

Exemples d'applets : geturls.java




```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Frameset//EN">
<HTML> <!-- get2urls.html -->
<HEAD><TITLE>Applet get 2 URLs -> 2 Frames</TITLE></HEAD>

<frameset rows="120,*">
  <frame src="./geturls.html">
  <frameset cols="*,*">
        <frame src="./getfra.html" name="fra">
        <frame src="./getfrb.html" name="frb">
  </frameset>
</frameset>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML> <!-- geturls.html -->
<HEAD><TITLE>Applet get 2 URLs</TITLE></HEAD>
<BODY BGCOLOR="GREY">
<APPLET CODE="geturls" WIDTH=600 HEIGHT=100></APPLET>
```



```
</BODY></HTML>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML> <!-- getfra.html -->
<HEAD><TITLE>Frame "fra"</TITLE></HEAD>
<BODY BGCOLOR="lightblue">
<H3 ALIGN="CENTER">url1 s'affichera ici</H3>
</BODY></HTML>

// applet geturls.java -
// fait 2 requêtes web simultanées et affiche
// les résultats dans deux frames.

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
```

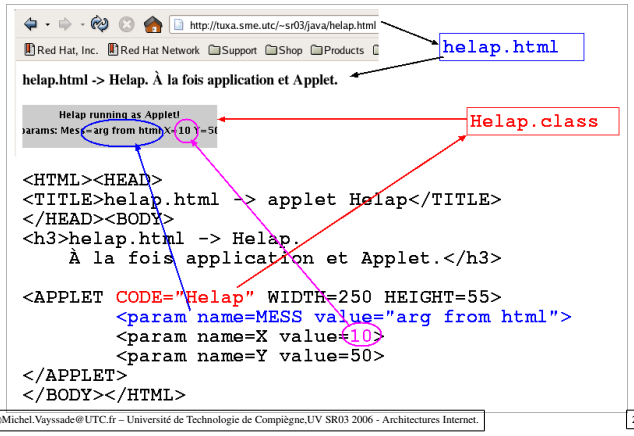


```
public class geturls extends Applet
                        implements ActionListener {
  private Button envoiget;
  private String urls = {       "http://tuxa.sme.utc/~sr03/",
                                "http://tuxa.sme.utc/~sr02/"};
  private String frame= {"fra","frb"};

  public void init() {
    setBackground(Color.white);
    setFont(new Font("Serif", Font.BOLD, 18));

    envoiget = new Button("Envoi 2 requêtes");
    envoiget.addActionListener(this);
    add(envoiget);
  }
```



```
// quand clic bouton -> appel event listener de CETTE instance
// ( car addActionListener(this) )

  public void actionPerformed(ActionEvent event) {
    for(int i=0; i<urls.length; i++) {
      try {
        getAppletContext().showDocument(new URL(urlsi), framei);
      } catch(MalformedURLException mue) {}
    }
  }//fin actionPerformed

}//fin geturls
```

Exemples d'applets : Helap.java




```
// Helap.java : à la fois application et Applet

import javax.swing.*;
import java.awt.event.*;
import java.awt.FlowLayout;

// JApplet = Applet + swing
public class Helap extends JApplet {
  private static String message = "A default message";
  private static int x = 20;
  private static int y = 30;
  private static boolean isapplet = true; // mis à vrai dans main

  public void init() {
  System.out.println("Helap exec init");
  getContentPane().setLayout( new FlowLayout() );
```



```
  if (isapplet)
    {   // get param from html
        message = getParameter("Mess");
        x = Integer.parseInt(getParameter("X"));
        y = Integer.parseInt(getParameter("Y"));
        getContentPane().add(new JLabel("Helap running as Applet"));
    } else {
        getContentPane().add(new JLabel("Helap running as application"));
    }
   String outmess = "params: Mess="+message+" X="+x+" Y="+y;
   getContentPane().add(new JLabel(outmess));
}// end init

  public static void main(String args) {
  System.out.println("Helap exec main");
  JFrame frame = new JFrame("jframe créée par main");
```



```
  frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
                        System.out.println("--windowClosing--");
                        System.exit(0); }              });
  JApplet applet = new Helap();
  isapplet = false;
    if (args.length ¯ 3)
    {   System.out.println("Usage: java Helap message x y");
        System.exit(0);
    } else {
      message = args0;
      x = Integer.parseInt(args1);
      y = Integer.parseInt(args2);
    }
  frame.getContentPane().add(applet);
  frame.setSize(250, 70); // larg hter
```



```
  applet.init();
  applet.start();
  frame.setVisible(true);
}// end main

}// end Helap class
```


Évènements en Java

Java utilise le modèle habituel des "callback" pour gérer l'interface utilisateur graphique (GUI) dans les toolkits **awt** d'abord puis **swing**.

Les différents composants de l'awt (Abstract Windows Toolkit) ou de swing peuvent générer des **évènements**.

On trouve dans java **deux façons** de gérer les évènements :

1. la gestion traditionnelle, du AWT version 1.0,
2. la gestion par les "events listeners" à partir du AWT version 1.2, reprise dans swing.

Évènements en Java : modèle awt 1.0

Dans ce modèle, les composants se voient notifier les évènements par l'invocation par le toolkit d'une méthode ayant une certaine signature.
Principaux évènements :

```
mouseDown       (Event evt, int x, int y)
mouseUp         (Event evt, int x, int y)
mouseEnter      (Event evt, int x, int y)
mouseExit       (Event evt, int x, int y)
mouseMove       (Event evt, int x, int y)
mouseDrag       (Event evt, int x, int y)
gotFocus        (Event evt, Object what)
lostFocus       (Event evt, Object what)
keyDown         (Event evt, int key)
keyUp           (Event evt, int key)
action          (Event evt, Object what)
```


Il suffit de surcharger l'une de ces méthodes pour capter l'évènement correspondant

Ensuite, à la fin de notre fonction, si on sort par "return false" l'évènement est transmis aux autres gestionnaires d'évènements et si on sort par "return true", c'est la fin du traitement pour cet évènement.

Exemple d'évènements en Java avec le modèle awt 1.0 : Scribble1.java

L'applet est appelé par Scribble1.html :

```
<HTML>
<HEAD><TITLE>Scribble1 Applet</TITLE></HEAD>
<BODY BGCOLOR="lightgrey">
<b>Scribble1 Applet dans le cadre ci-dessous :</b>
<table BORDER="1" ><tr><td>
<APPLET CODE="Scribble1" WIDTH=400 HEIGHT=300></APPLET>
```

```
</td></tr></table>
</BODY></HTML>
```

Son exécution donne :

Le code source de l'applet :

```
// This example is from the book _Java in a Nutshell_
```



```
// Written by David Flanagan. Copyright (c)1996 O'Reilly
// Modifié pour remplacer fonctions "deprecated" de java 1.0
import java.applet.*;
import java.awt.*;

public class Scribble1 extends Applet {
    private int last_x = 0;
    private int last_y = 0;
    private Color current_color = Color.black;
    private Button clear_button;
    private Choice color_choices;

    // Called to initialize the applet.
    public void init() {
        // Set the background color
        this.setBackground(Color.white);
```



```
        // Create a button and add it to the applet.
        // Also, set the button's colors
        clear_button = new Button("Clear");
        clear_button.setForeground(Color.black);
        clear_button.setBackground(Color.lightGray);
        this.add(clear_button);

        // Create a menu of colors and add it to the applet.
        // Also set the menus's colors and add a label.
        color_choices = new Choice();
        color_choices.addItem("black");
        color_choices.addItem("red");
        color_choices.addItem("yellow");
        color_choices.addItem("green");
        color_choices.setForeground(Color.black);
```



```
        color_choices.setBackground(Color.lightGray);
        this.add(new Label("Color: "));
        this.add(color_choices);
    }

    // Called when the user clicks the mouse to start a scribble
    public boolean mouseDown(Event e, int x, int y)
    {
        last_x = x; last_y = y;
        return true;
    }

    // Called when the user scribbles with the mouse button down
    public boolean mouseDrag(Event e, int x, int y)
    {
        Graphics g = this.getGraphics();
```



```
        g.setColor(current_color);
        g.drawLine(last_x, last_y, x, y);
        last_x = x;
        last_y = y;
        return true;
    }

    // Called when the user clicks the button or chooses a color
    public boolean action(Event event, Object arg) {
        // If the Clear button was clicked on, handle it.
        if (event.target == clear_button) {
            Graphics g = this.getGraphics();
            Rectangle r = this.getBounds();
            g.setColor(this.getBackground());
            g.fillRect(r.x, r.y, r.width, r.height);
            return true;
```



```
        }
        // Otherwise if a color was chosen, handle that
        else if (event.target == color_choices) {
            String colorname = (String) arg;
            if (arg.equals("black")) current_color= Color.black;
            else if (arg.equals("red")) current_color= Color.red;
            else if (arg.equals("yellow"))
                                current_color = Color.yellow;
            else if (arg.equals("green"))
                                current_color = Color.green;
            return true;
        }
        // Otherwise, let the superclass handle it.
        else return false;
    }
}
```


Évènements en Java, modèle Java 2 SE : events listeners

Le modèle de gestion des évènements introduit avec J2SE (en fait il a commençé avec 1.1) utilise la notion de "events listeners".

Le traitement des évènements dans le modèle Java est basé sur le concept de "**event listener**" : un objet intéressé par la réception d'un évènement, va s'enregistrer auprès de l'objet source de l'évènement, et s'ajouter ainsi à la liste des "event listeners". Quand la source génère un évènement, elle va notifier à tous les objets listeners enregistrés, que cet évènement s'est produit.

Exemple de composants et d'évènements :

Composant	Évènement généré	Signification
Button	ActionEvent	Clic sur un bouton
Checkbox	ItemEvent	Choix d'une case à cocher
MenuItem	ActionEvent	Choix d'un item de menu

La source notifie à un "event listener" qu'un évènement s'est produit, en invoquant une méthode de l'objet intéressé, et en passant un objet "event" en argument à cette méthode.

Tous les objets qui se déclarent comme "event listener", doivent **implémenter une interface** ("Interface ActionListener") correspondant à cet évènement.

À chaque évènement correspond une (ou 2) interfaces. Par exemple :

Évènement généré	Listener Interface	Méthodes
ActionEvent	ActionListener	
ItemEvent	ItemListener	
MouseEvent	MouseListener	mouseClicked()
		mousePresed()
		mouseRelease()
	MouseMotionListener	mouseDragued()

Par convention, toutes les méthodes d'un "event listener" reçoivent un argument unique qui est un objet évènement du type correspondant à celui du listener.

Pour chacune des interfaces "listener" comportant plus d'une méthode, java.awt.event définit une classe "adaptateur" qui contient un corps vide pour chacune des méthodes. Ainsi l'objet "event listener" n'a qu'à sous-classer l'adaptateur (héritage) et surcharger les seules méthodes qui l'intéresse.

L'enregistrement d'un objet en tant que "event listener" auprès d'une source se fait par appel à une méthode dont le nom respecte une convention :
`addMouseListener()`
`addMouseMotionListener()`
etc...

Le retrait éventuel de la liste des listener, si l'objet n'est plus intéressé par la notification de ct évènement par :

`removeMouseListener()`, ...

Donc la séquence pour gérer les évènements générés par un objet (souvent un widget) est la suivante :

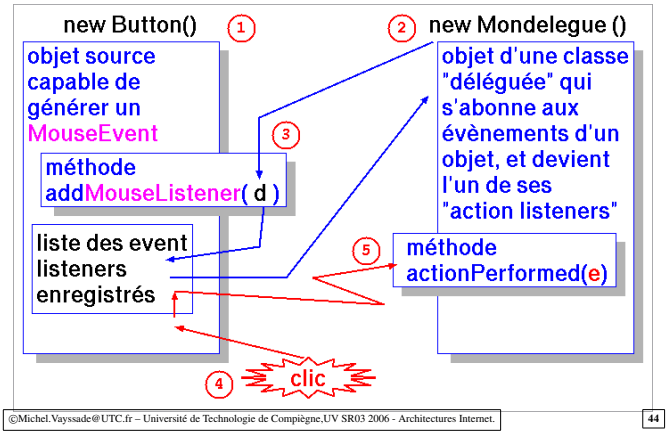
1. créer une instance d'une classe déléguée, cette classe doit posséder une méthode publique **actionPerformed()**,
2. enregistrer cette instance comme "actionListener" de l'objet dont on veut traiter les évènements (par exemple un bouton),
3. quand l'objet génère un évènement (par exemple lors d'un clic sur le bouton), il va appeler la fonction actionPerformed() de chacune des instances enregistrées en tant que "event listener".

Le nom "actionPerformed" de la méthode qui sera appelée lors de l'évènement est imposé : l'appelant est construit ainsi par convention. On assure la présence de cette méthode à la compilation car le délégué doit faire "implements ActionListener".

Rappel sur les interfaces java

Une interface java provoque un "héritage" particulier. L'interface ne fait **que définir des signatures de fonctions**. Mais elle ne contient pas de code pour ces fonctions.

La classe qui implémente une interface **doit** donc redéfinir toutes les fonctions spécifiées dans l'interface et leur attribuer du code (éventuellement vide d'ailleurs).

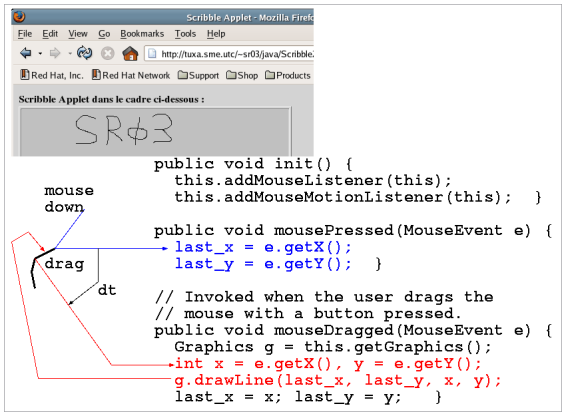



```
import java.awt.* ;        // accéder à Frame et Button
import java.awt.event.* ; // accéder à ActionListener

public class Simplevent extends Frame {
  public Simplevent () {                    // constructeur
    Button monBouton = new Button ("Hep!");
    // créer délégué et l'enregistrer auprès du bouton
    monBouton.addActionListener ( new MonDelegue() );
    add ( monBouton );
  }

  public static void main (String args[]) {
    Frame f = new Simplevent() ;   // créer la fenêtre
    f.setSize(200, 200);           // la dimensionner
    f.setVisible (true);           // l'afficher
  }
}

// classe déléguée gérant évènements sur le bouton
class MonDelegue implements ActionListener {
  public void actionPerformed (ActionEvent e) {
    System.out.println("Ici MonDelegue e="+e);
    System.exit(0); }
}
```

1 2 3 4 clic 5

le nom de la méthode est imposé

Autre exemple d'évènements par event listener : Scribble.java




```
// This example is from _Java Examples in a Nutshell_.
// http://www.oreilly.com Copyright (c) 1997 by David Flanagan
// This example is provided WITHOUT ANY WARRANTY either
// expressed or implied. You may study, use, modify, and
// distribute it for non-commercial purposes.

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/* A simple applet that uses the Java 1.1 event handling model */
public class Scribble extends Applet
              implements MouseListener,  MouseMotionListener {
  private int last_x, last_y;

  public void init() {
```



```
    // Tell this applet what MouseListener and MouseMotionListener
    // objects to notify when mouse and mouse motion events occur.
    // Since we implement the interfaces ourself,
    // our own methods are called.
    this.addMouseListener(this);
    this.addMouseMotionListener(this);
  }
  // A method from the MouseListener interface.  Invoked when the
  // user presses a mouse button.
  public void mousePressed(MouseEvent e) {
    last_x = e.getX();
    last_y = e.getY();
  }
  // A method from the MouseMotionListener interface.
  // Invoked when the user drags the mouse with a button pressed.
  public void mouseDragged(MouseEvent e) {
```

```
    Graphics g = this.getGraphics();
    int x = e.getX(), y = e.getY();
    g.drawLine(last_x, last_y, x, y);
    last_x = x; last_y = y;
  }
  // The other, unused methods of the MouseListener interface.
  public void mouseReleased(MouseEvent e) {;}
  public void mouseClicked(MouseEvent e) {;}
  public void mouseEntered(MouseEvent e) {;}
  public void mouseExited(MouseEvent e) {;}

  // The other method of the MouseMotionListener interface.
  public void mouseMoved(MouseEvent e) {;}
}
```


Différentes façons de retrouver l'origine d'un évènement

```
Création de composants :
        btna = new Button("aaa");
        btnb = new Button("bbb");
Enregistrement d'un event listener pour chacun :
        btna.addActionListener(this);
        btnb.addActionListener(this);
Retrouver la source :
public void  actionPerformed(ActionEvent evt) {
        Object src = evt.getSource();
        if(src instanceof Button)
                if( (Button)src == btna ) {...}
// ou alors :
        String srctag = evt.getActionCommand();
        if(srctag.equals("aaa")) {...}
```

On peut aussi, comme dans Simplevent.java vu plus haut, créer une instance différente d'une classe déléguée pour chaque source.

Un exemple de gestion d'évènements "exceptions utilisateur"

```
// This example is from _Java Examples in a Nutshell_.
// http://www.oreilly.com Copyright (c) 1997 by David Flanagan
// Here we define some exception types of our own.
// Exception classes generally have constructors but
// no data or other methods. All these do is to call
// their superclass constructors.
class MyException extends Exception {
    public MyException() { super(); }
    public MyException(String s) { super(s); }
}
class MyOtherException extends Exception {
    public MyOtherException() { super(); }
    public MyOtherException(String s) { super(s); }
}
```



```
class MySubException extends MyException {
    public MySubException() { super(); }
    public MySubException(String s) { super(s); }
}

// This class demonstrates defining, throwing and handling
// exceptions. Try invoking it in the following ways and
// try to understand the output:
//  java throwtest        //  java throwtest 99
//  java throwtest one    //  java throwtest 2
//  java throwtest 0      //  java throwtest 3
//  java throwtest 1

public class throwtest {
    // This is the main() method.  Note that it uses two
    // catch clauses to handle two standard Java exceptions.
```



```
    public static void main(String argv) {
        int i;

        // First, convert our argument to an integer
        // Make sure we have an argument and that
        // it is convertible.
        try {
            i = Integer.parseInt(argv0);
        }
        catch (ArrayIndexOutOfBoundsException e) {
            // argv is empty
            System.out.println("Must specify an argument");
            return;
        }
        catch (NumberFormatException e) {
           // argv0 not an integer
```



```
            System.out.println("Specify an integer argument.");
            return;
        }
        // Now, pass that integer to method a().
        a(i);
    }//end main

    // This method invokes b(), which is declared to throw
    // one type of exception.  We handle that one exception.
    public static void a(int i) {
        try {
            b(i);
        }
        catch (MyException e) {                    // Point 1.
            // Here we handle MyException and
            // its subclass MyOtherException
```



```
            if (e instanceof MySubException)
                System.out.print("MySubException: ");
            else
                System.out.print("MyException: ");
            System.out.println(e.getMessage());
            System.out.println("Handled at point 1");
        }
    }//end a(i)

    // This method invokes c(), and handles one of the
    // two exception types that that method can throw. The other
    // exception type is not handled, and is propagated up
    // and declared in this method's throws clause.
    // This method also has a finally clause to finish up
    // the work of its try clause.  Note that the finally clause
    // is executed after a local catch clause, but before a
```



```
    // a containing catch clause or one in an invoking procedure.
    public static void b(int i) throws MyException {
        int result;
        try {
            System.out.print("i = " + i);
            result = c(i);
            System.out.print(" c(i) = " + result);
        }
        catch (MyOtherException e) {                // Point 2.
            // Handle MyOtherException exceptions:
            System.out.println(
                        "MyOtherException: " + e.getMessage());
            System.out.println("Handled at point 2");
        }
        finally {
            // Terminate output we printed above with a newline.
```

```
            System.out.print("\n");
        }
    }//end b(i)

    // This method computes a value or throws an exception.
    // The throws clause only lists two exceptions, because
    // one of the exceptions thrown is a subclass of another.
    public static int c(int i) throws MyException,
                                      MyOtherException {
        switch (i) {
            case 0: // processing resumes at point 1 above
                throw new MyException("input too low");
            case 1: // processing resumes at point 1 above
                throw new MySubException("input still too low");
            case 99:// processing resumes at point 2 above
                throw new MyOtherException("input too high");
```



```
            default:
                return i*i;
        }
    }//end c(i)
}
```

```
> java throwtest              | > java throwtest 1
Must specify an argument      | i = 1
> java throwtest aaa          | MySubException: input still too low
Must specify integer argument | Handled at point 1
> java throwtest 0            | > java throwtest 99
i = 0                         | i = 99MyOtherException:input too high
MyException: input too low    | Handled at point 2
Handled at point 1            | > java throwtest 44
                              | i = 44 c(i) = 1936
```


Questions sur Ch16 - Programmation Java

➤ Quelle est la différence entre une application java et une applet java ?
Réponse : diapos 2 et 3.

➤ Quelle est le point d'entrée d'une application java ?
Réponse : diapos 8 et 9.

➤ Une applet java peut-elle provoquer le chargement d'une nouvelle page web ? Comment ?
Réponse : diapo 24.

➤ Un programme java peut-il être exécuté aussi bien comme application que comme applet ? À quelle condition ?
Réponse : diapos 26 à 29. Il sufft que la classe "extends Applet" et qu'elle possède une fontion main().

➤ Décrire le fonctionnement de la gestion des évènements dans Java 2. Un schéma serait le bienvenu.
Réponse : diapos 43 et 45.

➤ Peux-t-on déclarer et utiliser ses propres exceptions dans un programme java ? Comment ? Réponse : diapos 51, 56 et 57.

Ch16 - Programmation Java

Fin du chapitre.

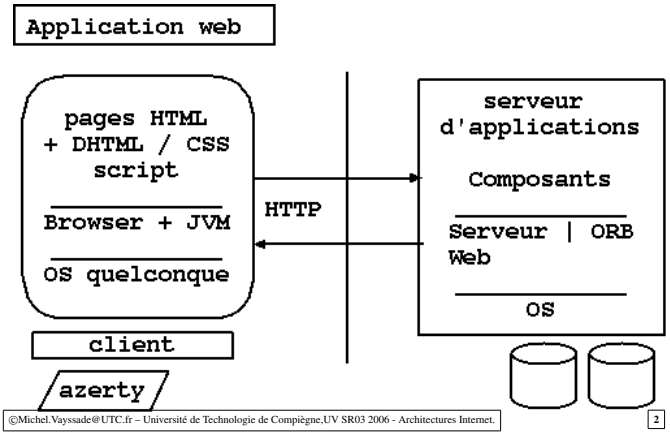
(diapo vide)

Ch17 - Servlets et JSP

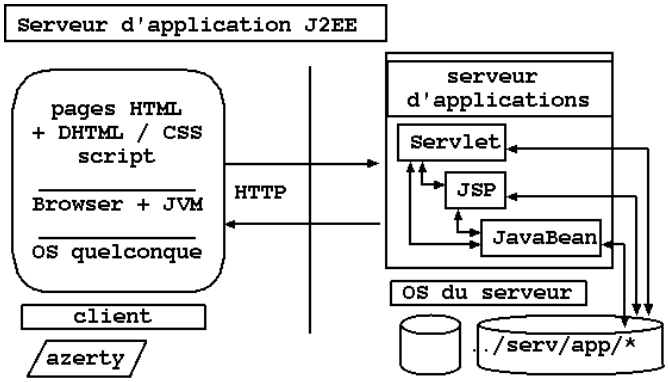
Introduction : Qu'est-ce qu'une servlet et une JSP ?

Servlets et JSP sont des composants logiciels destinés à être exécutés dans un **serveur d'application J2EE**.

Application web



Serveur d'application J2EE



Une servlet est une classe java qui est capable de lire le contenu d'une requête HTTP et de produire (sur son flux de sortie) une réponse HTTP.

Une JSP est un squelette de page HTML qui contient des portions de code java, comme PHP, entre les balises **<%** et **%>**.

Servlets et JSP peuvent s'appeler les unes les autres.

Servlets, JSP et JavaBeans : aussi bien les servlets que les JSP peuvent appeler un JavaBean.

Les Servlets et les JSP avec Tomcat

Qu'est-ce que Tomcat ? Tomcat 4 est un **conteneur Servlet/JSP**. Il implémente "the Servlet 2.3 and JavaServer Pages 1.2 specifications from Java Software".

D'après ses auteurs, c'est "a useful platform for developing and deploying web applications and web services".

C'est un produit du projet "Jakarta" de Apache.

La documentation, sur l'installation de sunserv, est accessible à l'URL :
http ://sunserv.utc :8080/tomcat-docs/index.html

Sur sunserv, on peut vérifier que le serveur Tomcat est présent par la commande suivante :

```
1556 lo33 sunserv:~/mv/servlet> ps -ef | grep -i jak
http 1895  1 0 09:16:39 pts/47 0:09 /usr/java/bin/../bin/sparc/
           native_threads/java -classpath /usr/local/JAKARTA/ja
lo33 19016 7212  0 09:18:14 pts/26  0:00 grep -i jak

si non : su puis :
/etc/init.d/tomcat start
```


La commande "list" du gestionnaire de Tomcat permet de connaitre la liste des applications tournant sur le serveur :

```
Commande par entrée de l'URL :
        http://sunserv.utc:8080/manager/list
Réponse =
OK - Listed applications for virtual host localhost
/examples:running:0
/webdav:running:0
/kap6:running:0
/tomcat-docs:running:0
/manager:running:0
/:running:0
```

Le gestionnaire de Tomcat demande un Username et un Password. Ce sont des informations spécifiques à Tomcat.

Premiers exemples

Pour exécuter une "application web", il faut d'abord compiler cette application, puis **l'installer** dans Tomcat. Ceci se fait par la commande :

```
http://www4.utc.fr:8080/manager/install?
       path=/lo33&war=file:/usr/uvs/lo33/lo33/mv/servlet

la réponse de Tomcat est :

OK - Installed application at context path /lo33
```

Le résultat de cette commande (encore une fois envoyée à Tomcat comme une URL, par l'intermédiaire du navigateur), est que le **contenu du répertoire** de l'utilisateur sera **considéré** par le serveur Tomcat comme la **racine** des applications réportoriées sous le nom **/lo33**.

L'appel de l'URL : **http ://sunserv.utc :8080/lo33/** va donner dans le navigateur la liste des fichiers du répertoire associé.

La mise en place de l'application :

Elle doit obéir à des **règles précises** concernant le placement des **sources JSP**, et des fichiers "**.class**" :

```
1684 lo33 sunserv:~/mv/servlet> ls -R
.:
HelloWorld.java       StringBean.jar        isprime.html
IsPrimeServlet.class  StringBean.java       manifest.tmp
IsPrimeServlet.java   StringBean.jsp        verifjsp.jsp
StringBean.class      WEB-INF
./WEB-INF:
classes
./WEB-INF/classes:
HelloWorld.class      IsPrimeServlet.class  cwp
./WEB-INF/classes/cwp:
StringBean.class
```


L'exemple ci-dessus contient **deux servlets (HelloWorld.java et IsPrimeServlet.java)**, et un **JavaBean (StringBean.java)** appelé par **une JSP (StringBean.jsp)**.

Les .class des servlets **doivent** être stockés dans le répertoire : **/racine/WEB-INF/classes**.

Un JavaBean situé dans le **package "test"** est compilé en un .class qui **doit** être stocké dans le répertoire **/racine/WEB-INF/classes/test**.

Une fois placé dans WEB-INF/classes le .class de la servlet va être appelée par l'URL :
http ://sunserv.utc :8080/lo33/servlet/HelloWorld

Dans cette URL, "/lo33" désigne la racine de l'application installée dans le serveur Tomcat et "**servlet/HelloWorld**" peut être vu comme un **nom logique** (il n'y a aucun répertoire qui s'appelle "servlet") désignant **racine/WEB-INF/classes/HelloWorld**.

On pourra appeler la servlet par une URL de ce type dans un fichier html, comme par exemple, ci-dessous pour l'appel de IsPrimeServlet après avoir lu un argument dans une "form" html :

```
2009 lo33 sunserv:~/mv/servlet> more isprime.html
<HTML>
<HEAD><TITLE>servlet 2</TITLE></HEAD>
<BODY>
<H1>Test sur les nombres</H1>
```



```
<FORM Method="post"
 Action="http://sunserv.utc:8080/lo33/servlet/IsPrimeServlet">
Entrez un nombre:
<INPUT type=text size=10 name=number>
<INPUT type=submit value="Est-il premier?">
</FORM>
</BODY></HTML>
```


Compilation

Pour compiler une servlet, il faut ajouter au **CLASSPATH** l'emplacement de la bibliothèque (.jar) qui implémente **l'API des servlets** dans Tomcat :

```
setenv CLASSPATH \
  ".:/usr/local/JAKARTA/jakarta-servletapi-4/lib/servlet.jar"
compilation :
javac HelloWorld.java
```


Installation dans un répertoire privé

Normalement les applications sont installées dans le répertoire "webapps" de l'installation de Tomcat. Ceci pose un problème de droits d'accès au répertoire. Pour le contourner, on installe le produit dans un répertoire privé.

Il faut que ce répertoire soit accessible à Tomcat et que les fichiers .class soient lisibles par Tomcat. Donc faire "chmod o+x" pour tous les répertoires et "chmod o+r" pour les .class et les JSP.

```
http://www4.utc.fr:8080/manager/install? \
path=/lo33&war=file:/usr/uvs/lo33/lo33/mv/servlet .
OK - Installed application at context path /lo33
si erreur : "cannot access directory"
chmod o+x mv/servlet
```



```
puis, recharger :
http://www4.utc.fr:8080/manager/reload?path=/lo33 .
OK - Reloaded application at context path /lo33

si on fait :

http://sunserv.utc:8080/manager/list .
OK - Listed applications for virtual host localhost
/examples:running:0
/webdav:running:0
/lo33:running:1  <- - - - - - - - en plus.
/tomcat-docs:running:0
/manager:running:0
/:running:0
```

Exécution d'une servlet

La servlet "HelloWorld.java" a été compilée, et le .class mis dans le répertoire **./WEB-INF/classes** situé sous la "racine" déclarée à Tomcat par la commande :
http ://www4.utc.fr :8080/manager/install ? donnée plus haut.

Quand on entre dans un navigateur l'URL
http ://sunserv.utc :8080/lo33/servlet/HelloWorld Tomcat exécute dans sa machine virtuelle java le code java contenu dans : **./WEB-INF/classes/HelloWorld.class**.

Attention ! le mot "servlet" dans l'URL ci-dessus ne fait **pas** référence à un répertoire portant le nom "servlet". C'est un **marqueur logique**. Tomcat va le **remplacer** par le chemin réel qui lui a été donné dans la commande "install".

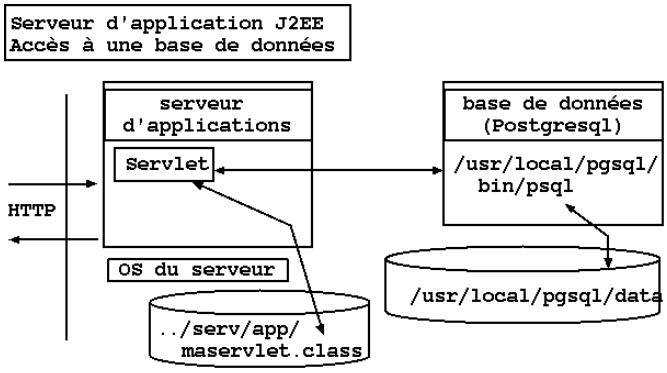
HelloWorld.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {
    public void doGet( HttpServletRequest request,
       HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<head>");
        out.println("<title>Hello World!</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Hello World!</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Page html fabriquée à la volée.

envoyée sur la sortie standard,
que Tomcat a connecté au socket TCP
sur lequel est branché le navigateur client

Servlets et connexion à une base de données



Servlets et connexion à une base de données (ex. Postgresql)

Les gestionnaires de bases de données sont des systèmes complexes qui gèrent eux-mêmes leur espace disque et qui implantent un contrôle d'accès (Username/Password) propre à la base de donnée. De plus sur une même installation, le programme gestionnaire gère **plusieurs** bases de données.

La connexion a une base de données va donc nécessiter un processus en trois étapes :

- **Connexion** au gestionnaire (MySQL, Oracle, PostGresql, Sybase, Informix, DB2),
- **Choix de la base** que l'on veut accéder,
- **Identification** : Username/Password de la base de données.

Exemple d'accès depuis un programme Java :

```
import java.sql.*;
..........
     username = "lo33xxx";
     password = "baselo33";
     // The URL to connect to the database
     // Syntax: jdbc:TYPE://machine:port/DB_NAME
     url = "jdbc:postgresql://sunserv:5432/baselo33";

     Connection conn;
     conn = DriverManager.getConnection(url, username, password);
```


Les insformations peuvent aussi être passées dans les arguments :

```
java ExecuteSQL -d postgresql.Driver \
-u lo33xxx -p yyyy jdbc:postgresql://sunserv:5432/baselo33
args parsed.
load driver. driver loaded.
open a connection.
connection opened.
sql> select * from auteur
+----------------------+------------------------------+
|        auteur        |            fichier           |
+----------------------+------------------------------+
|  ........            | ......                       |
```


LanceRequete.html et FormLeft.html

```
<html>
<head><title>frasme de lance requete</title></head><body>
<frameset cols="30%,*">
<frame src = "FormLeft.html" name = "left">
<frame src =
 "http://sunserv.utc.fr:8080/lo33db/servlet/LanceRequete"
 name = "right">
</frameset>
</body></html>
```

LanceRequete.html

```
<html>
<head><title>Lancer requete database</title></head>
<body>
<P>
<form action=
  "http://sunserv.utc.fr:8080/lo33db/servlet/LanceRequete"
  target="right">
<input type=text size=30 name=requete>
<br>
<input type=submit>
</form>
</body></html>
```

appel de la servlet "LanceRequete"

LanceRequete.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
import java.io.*;

public class LanceRequete extends HttpServlet {
      String username;  String password;
      String url;       String requete ="";
      String nom;
      int nbre;

    public void doGet( HttpServletRequest request,
       HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Lance requete!</title>");
        out.println("</head>");
        out.println("<body>");
```

LanceRequete.java 1/3

début de la page HTML générée dynamiquement

LanceRequete.java

LanceRequete.java 2/3

```
// ---- configure START
username = "lo17xxx";
password = "dblo17";
// The URL to connect to the database
// Syntax: jdbc:TYPE://machine:port/DB_NAME
url = "jdbc:postgresql://sunserv/dblo17";
// ---- configure END

String requete;
requete = request.getParameter("requete");
if (requete != null) {
   // INSTALL/load the Driver (Vendor specific Code)
   try {
       Class.forName("org.postgresql.Driver");
       } catch(java.lang.ClassNotFoundException e) {
         System.err.print("ClassNotFoundException: ");
         System.err.println(e.getMessage());
       }
```

emplacement de la base et paramètres d'accès

chaque base de données a son propre driver d'accès

LanceRequete.java

LanceRequete.java 3/3

connexion à la BdD

```
try{ Connection con;
     Statement stmt;
     // Establish Connection to database
     //  at URL with username and password
     con = DriverManager.getConnection(url,
                                 username, password);
     stmt = con.createStatement();
     // Send query and bind to result set
     ResultSet rs= stmt.executeQuery(requete);
     ResultSetMetaData rsmd=rs.getMetaData();
     nbre=rsmd.getColumnCount();
     while (rs.next()) {
         for (int i=1; i<=nbre;i++){
           nom = rsmd.getColumnName(i);
           String s = rs.getString(nom);
           out.print(s);
         }
     out.print("<p>");
     }
out.println("</body>");
out.println("</html>");
// Close resources
stmt.close(); con.close();
}//try
```

Boucle sur chaque ligne du "ResultSet": Pour chaque ligne, boucle pour afficher le contenu de chaque colonne

HTML envoyé au navigateur

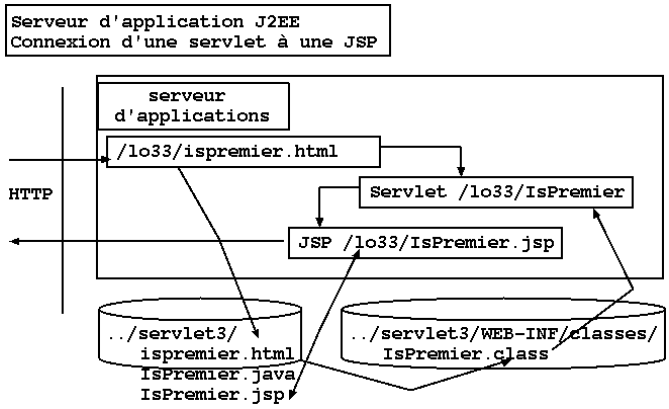
Servlets, JSP et JavaBeans : interactions

Presque toutes les interactions sont possibles (sinon faciles) entre les servlets, les JSP et les JavaBeans :

- une page html appelle une servlet,
- une servlet propage une requête vers une JSP,
- une JSP appelle une servlet,
- une servlet instancie un JavaBean et lui passe de l'informations ;
- une JSP appelle un JavaBean pour lui demander un calcul,
- une JSP appelle un JavaBean déja instancié par une servlet et lui demande des informations que la servlet lui a passé,
- ...

Les spécifications des servlets et des JSP prévoient les API et les syntaxes pour réaliser toutes ces opérations.

Connexion d'une Servlet à une JSP



Exemple de Connexion d'une Servlet à une JSP

```
lo33 sunserv:~/servlet3> ls ispremier.html IsPremier.*
IsPremier.java  IsPremier.jsp   ispremier.html
lo33 sunserv:~/servlet3> echo $CLASSPATH
.:/usr/local/JAKARTA/jakarta-servletapi-4/lib/servlet.jar
lo33 sunserv:~/servlet3> javac IsPremier.java
lo33 sunserv:~/servlet3> mv IsPremier.class WEB-INF/classes/

http://www4.utc.fr:8080/manager/install?
path=/lo33&war=file:/usr/uvs/lo33/lo33/mv/servlet3
ou  http://www4.utc.fr:8080/manager/reload?path=/lo33
puis : http://www4.utc.fr:8080/lo33/ispremier.html
==> affichage de la forme, saisie du nombre, envoi
Page JSP IsPremier.jsp appelée par servlet IsPremier.java
Nombre demandé était : 17 Ce nombre est premier.
```


Exemple de Connexion d'une Servlet à une JSP

Pour pouvoir appeler une page JSP depuis une servlet, on permet à la servlet de renvoyer la requête ("**forward request**") en utilisant un "**RequestDispatcher**".

On obtient un ré-émetteur en appelant la **méthode "getRequestDispatcher" du "ServletContext"** en lui fournissant en argument un URL relatif à la racine du serveur.

Exemple : **String url = "/mespages/mapagejsp.jsp" ; RequestDispatcher dispatcher = getServletContext().getRequestDispatcher(url) ;**

Une fois que l'on a un "dispatcher", on utilise sa méthode "**forward()**" pour transférer complètement le contrôle à l'URL associée à ce dispatcher : **dispatcher.forward (request, response) ;**

Ici request et response sont des variables contenant HttpServletRequest et HttpServletResponse.

On peut aussi **inclure** dans la réponse de la servlet la réponse produite par la page JSP "appelée", puis continuer le traitement dans la servlet : **dispatcher.include (request, response) ;**

Ces deux fonctions renvoient ServletException et IOException.

ispremier.html

ispremier.html

```
<HTML>
<HEAD><TITLE>servlet 3 et JSP</TITLE></HEAD>
<BODY>
<H1>Test sur les nombres</H1>

<FORM Method="post"
 Action="http://sun.utc:8080/lo33/servlet/IsPremier">
Entrez un nombre:
<INPUT type=text size=20 name="lenombre">
<INPUT type=submit value="Est-il premier?">
</FORM>

</BODY></HTML>
```

IsPremier.java

IsPremier.jsp

IsPremier.java

```
public class IsPremier extends HttpServlet
{
public void service( HttpServletRequest req,
                     HttpServletResponse res)
    throws ServletException, IOException
    { String val;
      int number = Integer.parseInt(
              (req.getParameterValues("lenombre"))[0]);
      boolean prime = isPrime(number);
      if (prime) { val = new String ("est premier");
      } else { val = new String ("n'est pas premier");
      }
      req.setAttribute("estil", val);
      gotoPage("/IsPremier.jsp", req, res);
    }
    // utility routine forwarding request
    private void gotoPage( String url,
                           HttpServletRequest request,
                           HttpServletResponse response)
    throws IOException, ServletException
    {
      RequestDispatcher dispatcher =
         getServletContext().getRequestDispatcher(url);
      dispatcher.forward (request, response);
    }
```

IsPremier.java

requête propagée vers /IsPremier.jsp
après lui avoir ajouté l'attribut "estil"
dont la valeur contient la réponse à afficher

IsPremier.jsp

IsPremier.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<!--
IsPremier.jsp : example de jsp appelée par une servlet
-->
<HTML>
<HEAD><TITLE>Using JSP from Servlets</TITLE></HEAD>
<BODY>
<h3>Page JSP IsPremier.jsp appelée par servlet
    IsPremier.java</h3>
<p>
Nombre demandé était: <%= request.getParameter("lenombre") %>
<br/>
Ce nombre <%= request.getAttribute("estil") %>.
<br/>
</p>
<hr><br/>
</body></html>
```

Cette page JSP est activée par le "forward" de la requête par la servlet.
Elle récupère la valeur d'un paramètre et celle d'un attribut et se charge de les afficher (envoi de HTML au navigateur).

Passage d'information depuis la servlet vers la page JSP

1. passer l'info par des **attributs de la requête** :
   passer (depuis la servlet : req.setAttribute("estil", val) ;
   récupérer (dans la jsp) : **<%= request.getAttribute("estil") %>**
2. passer par l'**utilisation d'un Bean** présent dans la dir de la servlet, ce Bean étant ensuite utilisé par la JSP.

Passage d'information par l'utilisation d'un Bean : il y a 3 méthodes : par **HttpServletRequest, HttpSession ou ServletContext** :

1. **HttpServletRequest :** data utilisables par la JSP seulement **dans CETTE requête** :

```
UneClasse value = new UneClasse(...);
request.setAttribute("key",value);
```

la page JSP récupère l'info par :

```
<jsp:useBean id="key" class="UneClasse" scope="request" />
```


2. **HttpSession :** data utilisables par la JSP **dans CETTE requête ET dans les requêtes suivantes du même client** :

```
UneClasse value = new UneClasse(...);
HttpSession session = request.getSession(true);
session.setAttribute("key",value);
```

la page JSP récupère l'info par :

```
<jsp:useBean id="key" class="UneClasse" scope="session" />
```

3. **ServletContext :** data utilisables par la JSP dans **cette requête ET dans les requêtes suivantes de n'importe quel client** :

```
UneClasse value = new UneClasse(...);
getServletContext().setAttribute("key",value);
```

la page JSP récupère l'info par :

```
<jsp:useBean id="key" class="UneClasse" scope="application" />
```


Exemple d'interaction html -> Servlets -> JavaBeans -> JSP -> JavaBeans

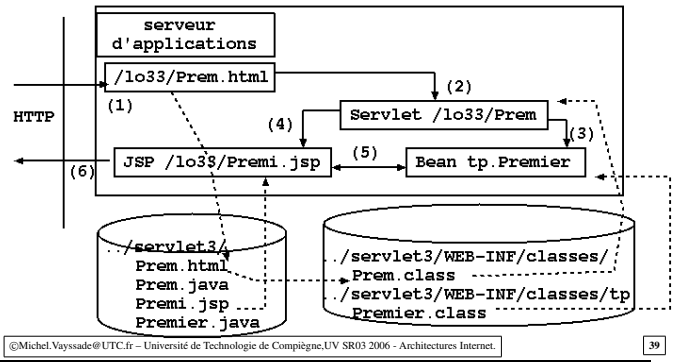
Cet exemple va montrer une interaction entre les **servlets, les JSP et les JavaBeans** :
une page html **(1)(Prem.html)** appelle une servlet **(2)(Prem.java)** qui instancie un JavaBean **(3)(Premier.java)**, lui passe un argument, puis renvoie la requête vers une JSP **(4)(Premi.jsp)** ;

cette JSP appelle **(5) le JavaBean** instancié précédemment par la servlet, lui demande **(5) des paramètres** puis envoie **(6) la page html** formattée au navigateur.

À la fin de la JSP, on montre un exemple de modification d'un paramètre du Bean par l'accès depuis la JSP aux fonctions du Bean (**<jsp :setProperty ..**).

Interactions Servlet, JSP, JavaBean

Serveur d'application J2EE
Connexion d'une servlet à une JSP
avec passage d'informations par un JavaBean



Prem.html

Prem.html

```
<HTML>
<HEAD><TITLE>servlet 3 , bean et JSP</TITLE></HEAD>
<BODY>
<H1>Test sur les nombres</H1>

<FORM Method="post"
 Action="http://sunserv.utc:8080/lo33/servlet/Prem">
Entrez un nombre:
<INPUT type=text size=10 name="lenombre">
<INPUT type=submit value="Est-il premier ? ">
</FORM>

</BODY></HTML>
```

Prem.java → Prem.class

Prem.java

```
import tp.Premier;
.......
public class Prem extends HttpServlet
{
public void service( HttpServletRequest req,
   HttpServletResponse res)
   throws ServletException, IOException
   {
// int number = Integer.parseInt(
//    (req.getParameterValues("lenombre"))[0]);

   Premier val = new Premier ();
// val.setLeNum(number);
// val.setUnAutre(123);
   val.setLeNum((req.getParameterValues("lenombre"))[0]);

   req.setAttribute("estil", val);
   gotoPage("/Premi.jsp", req, res);
 }
    private void gotoPage( String url,
                           HttpServletRequest request,
                           HttpServletResponse response)
    throws IOException, ServletException
    { RequestDispatcher dispatcher =
          getServletContext().getRequestDispatcher(url);
      dispatcher.forward (request, response);   }
public String getServletInfo()
 { return "A servlet to test for primeness"; }
}
```

Prem.java

Premier.java

setLeNum

Premi.jsp

41

Premier.java

```
package tp;
/** A simple bean been called by a servlet
 *  store a value and give it to a calling JSP */
public class Premier {
  private int num = 0;
  private boolean is;
  private String is0 = "n'est pas premier";
  private String is1 = "est premier";
  private String isPrem = is0;
  private String leNum;
// a bean class must have a zéro-arg constructor, or no constr.
// a bean class should have no public instance variables
// persitant values should be access with methods setXyz, getXyz

  public void setLeNum(String n) {
    this.num = Integer.parseInt(n); // keep init val in local
    this.is = isPrime(this.num); // calculate and keep result
    isPrem=is0;
    if (this.is) isPrem=is1;  }
  public String getLeNum() {
    this.leNum = ""+this.num;
    return(this.leNum);  }
  public String getIsPrem() {
    return(this.isPrem);    }
  static boolean isPrime(int i) {...... }
//end class Premier
```

Premier.java

42

Premi.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML ...">
<!-- Premi.jsp: exemple de jsp appelée par une servlet -->
<HTML><HEAD><TITLE>Using JSP from Servlets</TITLE></HEAD>
<BODY>
<h3>Page JSP Premi.jsp appelée par servlet Prem.java</h3>
<p>Nombre demandé était :
<%= request.getParameter("lenombre") %>
<br/>
Ce nombre <%= request.getAttribute("estil") %>.
<br/>
Appel du bean "Premier.java" du package "tp".
<br/>
<jsp:useBean id="estil" class="tp.Premier" scope="request" />
Le nombre
<jsp:getProperty name="estil" property="leNum" />
<jsp:getProperty name="estil" property="isPrem" />
<br/>
Change
<jsp:setProperty name="estil" property="leNum" value='<%= "5" %>'/>
<br/>
New =  <jsp:getProperty name="estil" property="leNum" />
<jsp:getProperty name="estil" property="isPrem" />
</p>
<hr><br/>
</body></html>
```

renvoie : tp.Premier@4845aa = référence sur instance du bean

"estil" est le nom donné à l'instance du bean par la servlet dans : req.setAttribute("estil", val);

43

Questions sur Ch17 - Servlets et JSP

➤ Dans quel contexte utilise-t-on servlets et JSP ?

Réponse : diapos 1 et 3.

➤ Donner en 2 lignes chacune les défnition des servlets et JSP ?

Réponse : diapo 4.

➤ Qu'est-ce que Tomcat ?

Réponse : diapo 5.

➤ En quoi consiste "l'installation" d'une application dans Tomcat ?

Réponse : diapos 9 à 11.

➤ Qu'est-ce qu'une servlet et où s'exécute-t-elle ?

Réponse : diapo 19.

➤ Faites un schéma des relations entre servlet, JSP et Bean.

©Michel.Vayssade@UTC.fr – Université de Technologie de Compiègne,UV SR03 2006 - Architectures Internet.

Réponse : diapo 28.

➤ Quelle relation lie les 2 lignes de code suivantes :

```
req.setAttribute("estil", val);
<jsp :getProperty name="estil" property="leNum" />
```

Réponse : diapo 43.

Ch17 - Servlets et JSP

Fin du chapitre.

©Michel.Vayssade@UTC.fr – Université de Technologie de Compiègne,UV SR03 2006 - Architectures Internet.

(diapo vide)

©Michel.Vayssade@UTC.fr – Université de Technologie de Compiègne,UV SR03 2006 - Architectures Internet.

Ch18 - CORBA

- CORBA et les applications réparties
- CORBA et les objets ; Principe de Corba ; Stubs, skeleton.
- L'ORB
- Un premier exemple avec Corba
- CORBA : le POA

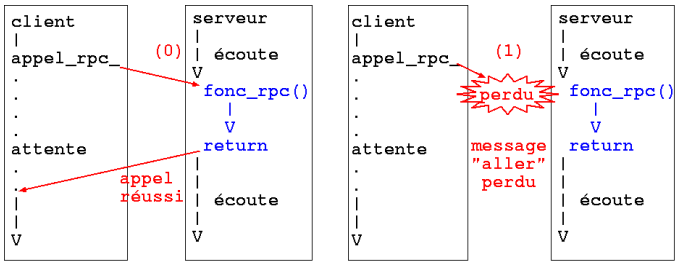
CORBA et les applications réparties

Généralités sur les applications réparties

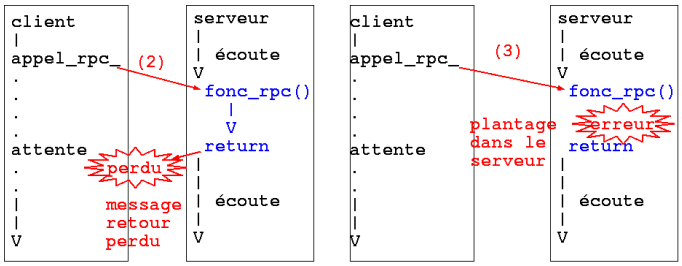
Le premier type d'applications réparties est le RPC. Il y a plusieurs modes d'invocation du RPC :

- **synchrone :** c'est le RPC classique vu auparavant : le client est bloqué jusqu'à reception des résultats renvoyés par le serveur.
- **synchrone différé :** le client lance l'appel, puis continue ses traitements ; Il demande ensuite périodiquement à la souche si le serveur a envoyé les résultats.
- **asynchrone :** le client lance l'appel et poursuit son exécution. Une routine d'interruption est exécutée quand les données de retour du serveur parviennent sur le site du client.

Applications réparties et fiabilité



Applications réparties et fiabilité



Applications réparties et fiabilité

Selon les différentes façons de se dérouler (bien ou mal) pour un appel RPC, il peut être **difficile de décider** ce qui doit être fait.

Si l'appel RPC déclenche un traitement côté serveur ayant une action sur une base de données :

- Pour **(1)** il **faut refaire** la demande ;
- Pour **(2)** il ne faut **SURTOUT PAS la refaire**,
- Pour **(3)** on est **soit** dans un cas de type **(1)** soit dans un cas de type **(2)** en fonction de l'endroit du plantage.

Dans ce cas (modif d'une base), il faut recourir à des **mécanismes de type "transaction"** permettant de passer d'un état **initial cohérent à un état final cohérent** ("**commit**").

Si une erreur survient on pourra annuler la transaction ("**rollback**") et revenir à l'état cohérent de départ avant de réessayer.

CORBA et les applications réparties

Corba est une **spécification d'un "bus à requêtes"**.

C'est une architecture **complexe** car elle permet d'assurer la **portabilité**, et **l'interopérabilité** entre environnements de **fournisseurs différents**.

Mais aussi de permettre l'écriture d'**applications hétérogènes** : invoquer **depuis du code C++** par exemple des **traitements effectués en Java** sur la même machine ou sur une autre machine. Ceci en pouvant changer le client ou le serveur de machine, sans avoir à recompiler.

Le bus CORBA, s'appelle un ORB : Object Request Broker.

CORBA : Le mécanisme de base

Le mécanisme de base de CORBA est le RPC (pas LE RPC ONC de SUN ni le RPC DCE, mais un appel à distance ayant le même type de fonctionnement).

On a donc dans Corba :

- un fichier de**description d'interface en IDL**,
- un **compilateur d'IDL** qui va créer les amorces client et serveur (dans Corba l'amorce client s'appelle une **souche ("stub")** et l'amorce serveur s'appelle un **squelette ("skeleton")**),
- un fichier **source du code client**,
- un fichier **source d'implémentation des objets serveurs**,
- un fichier **source du code du serveur**.

CORBA : Le mécanisme de base

En pratique, **l'ORB se présente commme un bibliothèque** : linkée avec le client et avec le serveur, elle va effectuer les opérations de **transcodage** (marshalling) des arguments de l'appel, d'**envoi des données** sur un socket internet (après connexion au serveur), de **récupération des résultats**, etc ...).

Force de Corba : avoir définit des spécification précises (+ de 1000 pages) des formats d'échange de données. C'est ceci qui permet d'**invoquer un objet** sur un serveur utilisant l'ORB du **fournisseur "A"** depuis un code linké avec l'ORB du **fournisseur "B"**.

Le client et le serveur pouvant se trouver sur des **machines différentes** (big/little endian, 32/64 bits) et être écrits dans des **langages différents**.

Les objets distribués

En 1989 création du consortium "OMG" pour promouvoir l'idée:

orienté objet + client-serveur = objets distribués

Moyen : promouvoir l'adoption de standards pour gérer les objets distribués, en publiant des spécifications

Particularité : Un standard ne peut être adopté par l'OMG que s'il en existe une implémentation. (donc que quelqu'un a prouvé que "ça marche") !

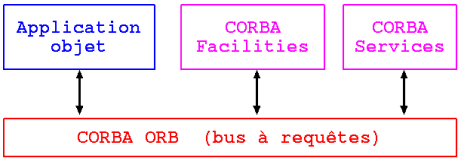
L'OMG a développé OMA : Object Management Architecture, avec un modèle de référence.
L'OMG compte un peu plus de 500 membres, et s'est fait une réputation pour adopter très vite des spécifications.

Ceci a permis le développement de nombreuses implémentations interopérables.

L'architecture OMA : CORBA

L'architecture OMA comporte 4 parties :

- un composant principal **l'ORB** : Object Request Broker (mandataire de requêtes) ;
- des services pour développer les objets distribués : CORBA Services ;
- des **services utilisés par les objets** des applications distribuées : CORBA Facilities ;
- les **applications distribuées** elles-mêmes (clients et serveurs).



CORBA : Common Object Request Broquer Architecture

Ce sont les objets eux-mêmes qui deviennent **serveur ou client** d'un autre objet.

Pour distinguer un objet-serveur d'un process-serveur on parle parfois d'objet "côté serveur". Mais un objet côté serveur peut lui-même être client d'un objet côté client !

Tous ces objets peuvent résider dans des process différents sur des **machines différentes**, n'importe où sur l'Internet.

CORBA a défini des **protocoles d'échanges** de messages entre objets. Le respect de ces protocoles permet l'**interopérabilité** entre fournisseurs et entre langages.

CORBA : Gestion des Objets

Les objets d'un environnement distribué doivent pouvoir :

- **accepter des requêtes** : c'est l'**interface** IDL qui décrit les requêtes acceptées : l'interface fait office de**contrat** pour les clients en indiquant les **services fournis** par un objet.
- fournir un moyen de créer de nouvelles instances : un constructeur ou "object factory".
- gérer leur espace de stockage ;
- gérer leur cycle de vie : création, stockage/persistance, destruction.

CORBA : Gestion des Objets

Les objets d'un environnement distribué doivent aussi pouvoir supporter :

- **l'encapsulation** : attributs accessibles seulement à traves les méthodes ;
- **l'héritage** : spécialiser une nouvelle classe en ajoutant ou surchargeant une classe existante ;
- **le polymorphisme** : comportements différents en fonction du type effectif de l'objet dans la hiérarchie d'héritage ;

En bref, conserver toutes les caractéristiques de la programmation objet.

CORBA : Interface et Implémentation

L'interface d'un objet **spécifie son comportement**. Elle déclare les services que l'objet est prêt à rendre et **cache** les détails de **l'implémentation** grâce à l'encapsulation.

==> donc, tant que l'objet a le comportement spécifié par son interface, son implémentation peut changer.

Ce principe est fondamental pour CORBA.

L'OMA adhère au modèle objet "classique" : toutes les méthodes sont contenues dans une classe.

Dans CORBA les objets sont connus exclusivement par leur interface, sans rien savoir de leur implémentation.

Pour cela, CORBA inclut un **langage IDL** (Interface Definition Language) qui permet de **décrire les interfaces** de tous les objets.

L'interface est l'équivalent CORBA d'une classe. Les **requêtes** à un objet CORBA sont toujours **relayées à travers une référence à un objet**. Toutes les **invocations** sont faites **sur la référence** "ObjRef" d'un objet.

IDL n'es PAS un langage de programmation : c'est un **langage pour exprimer des types**, en particulier des types "interface". Il n'a ni construction de contrôle, ni itérateurs.

Les types IDL

types de base : boolean, char, octet, enum, short, unsigned short, long, unsigned long, float, double, any, object.

types construits : string, struct, union, array, sequence.

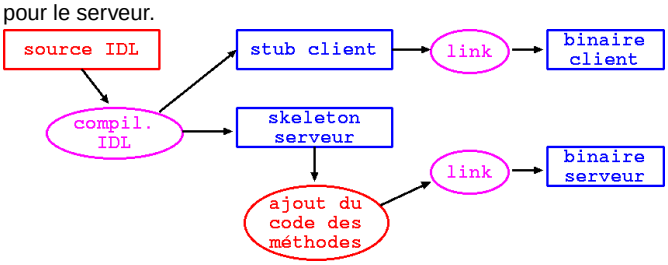
Exemples :

```
interface Person {
    readonly attribute Gender sex;
    readonly attribute Date birthdate;
    attribute string name;
};

interface Employee:Person {
```

```
    readonly attribute ssn;

    void addEmployer(Employer emp);
    void deleteEmployer(Employer emp);
    short numEmployers();
    Employer Employer(short index) throws exOutofBounds;
};
```


Stubs et Skeletons

source IDL –> (compilateur IDL) –> stub pour le client et skeleton pour le serveur.



stub : linkés statiquement au code client, les stubs sont produits complets par le compilateur IDL ;

skeleton : linkés statiquement au code serveur, après avoir été

"enrobés" par le code d'implémentation de chaque méthode (d'où leur nom).

Une **référence d'objet CORBA** est passée depuis le client jusqu'au serveur, mais pas l'objet lui-même.

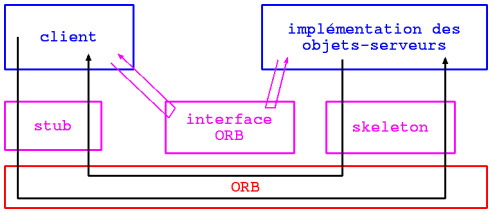
Les stubs constituent le mécanisme d'invocation d'un objet, côté client.

Les skeletons constituent l'interface côté serveur recevant les requêtes.

L'architecture spécifie que les objets CORBA sont**accessibles sans spécifier leur localisation** (i.e. "location transparent") ;

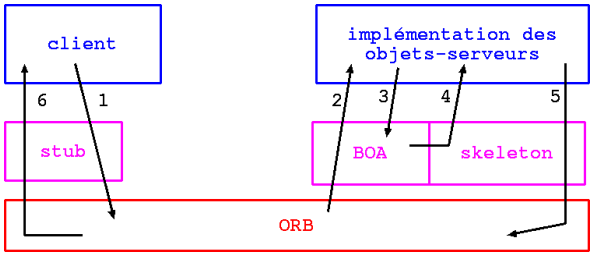
==> L'implémentation peut être dans le même process, dans un autre process de la même machine ou dans une autre machine ; CORBA prend en charge le transport des requêtes aux objets serveurs quelque soit l'endroit où ils sont situés.

L'interface ORB fournit des services (supposés largement disponibles) aussi bien aux clients qu'aux serveurs.



L'invocation de l'ORB

Côté serveur, un composant particulier, le **BOA (Basic Object Adaptor)** est chargé d'**activer l'implémentation des objets**. Les requêtes passent d'abord par le BOA avant d'être renvoyées vers les méthodes des objets une fois que ceux-ci sont prêts.



Une invocation suit le chemin suivant :

1. le client appelle une méthode à travers un stub,
2. l'ORB passe la requête au BOA qui active l'implémentation,
3. l'implémentation appelle le BOA pour lui indiquer qu'elle est prête,
4. le BOA passe la requête à une méthode de l'implémentation, via le skeleton,
5. l'implémentation renvoie les résultats ou les exceptions au client, à travers l'ORB.

L'activation du BOA

Si une interface est responsable de décrire le type de l'implémentation, l'adaptateur d'objet est responsable du "style" de cette implémentation, c'est-à-dire de la façon dont elle va être activée.

Il y a 4 styles d'activation :

- **activation par méthode** (per method) : un nouveau serveur est démarré à chaque fois qu'une méthode d'un objet est invoquée.
- **activation partagée** (shared) : un serveur supporte plusieurs objets actifs simultanément.
- **activation non partagée** (unshared) : un serveur ne supporte qu'un seul objet actif. Il peut gérer de nombreuses invocations de méthodes tant que ces méthodes appartiennent à un même objet.
- **serveur persistant** (persitant) : le serveur est toujours actif et ne requiert pas d'activation. Il est supposé toujours disponible tant que la machine est opérationnelle.

Persistance des objets

Le sous-groupe ODMG (Object Data Management Group) a standardisé la façon dont un OODBMS (Object Oriented Data Base Management System) interopère avec un ORB pour jouer le rôle d'un système de stockage persistant.

Un OODBMS a des fonctions différentes d'une simple implémentation d'objets, telles que :

- contrôler et gérer l'identité des objets,
- rester actif de longues périodes de temps,
- gérer les accès concurrents,
- récupération et retour à un état cohérent après une activation suite à une faute ;

RPC versus ORB

Tous les mandataires ("brokers") prélèvent un prix pour leurs services.

En quoi l'invocation d'une méthode à travers un ORB diffère-t-elle d'un RPC ?

Les mécanismes sont très similaires :

- appel de fonction à distance,
- passage par des adaptateurs ("stubs") côté client et côté serveur,
- empaquetage et dépaquetage des arguments par les adaptateurs,
- utilisation "transparente" du service de transport du réseau sous jacent,

- compilation et ligature séparée des codes clients et serveurs,
- description des fonctions appelées par un langage de définition (IDL) ;

Avec le RPC, on appelle une fonction spécifique , qui existe en dehors de tout contexte. Les données sont séparées et fournies au moment de l'appel.

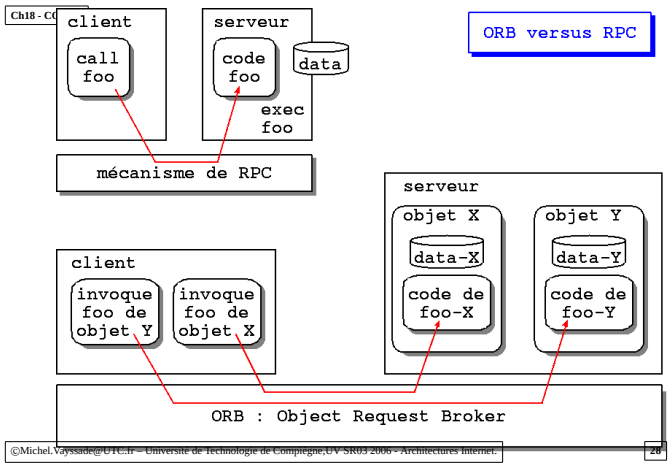
Avec l'ORB, on appelle une méthode à l'intérieur d'un objet spécifique . Cet objet peut encapsuler des données internes et les utiliser dans l'exécution de la fonction.

Différentes classes d'objets peuvent répondre de façon différente à un appel à la même méthode, par le biais du polymorphisme. La méthode appelée va travailler avec les données spécifiques à l'instance

de l'objet appelé.

La méthode d'invocation par un ORB à une grande précision : l'appel va à un objet précis qui contrôle des données précises et qui utilise la fonction spécifique associée à sa propre classe.

À contrario, l'appel RPC n'est pas spécifique : toutes les fonctions de même nom sont les mêmes, par définition.



Évidemment, la plupart des ORBs sont construits au-dessus d'un service RPC. On est donc amené à payer une pénalité en performances pour ce service plus fin.

Ceci dit, les implémentations récentes d'ORB montrent que :

- dans le même espace d'adressage (invocation intra-process), les ORB ont des coûts équivalents à l'invocation d'une méthode virtuelle C++ ;
- dans un contexte distribué, entre deux machines, un appel ORB est plus rapide qu'un appel PVM ;

Avec ces performances ont peut envisager d'utiliser les ORBs même pour des objets à grain très fin, et pas seulement pour du calcul distribué à gros grain.

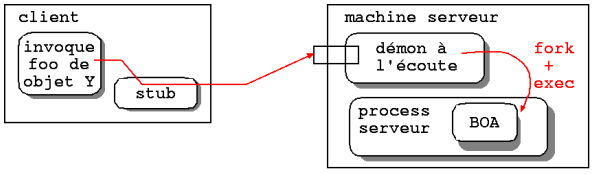
Implémentation des ORBs

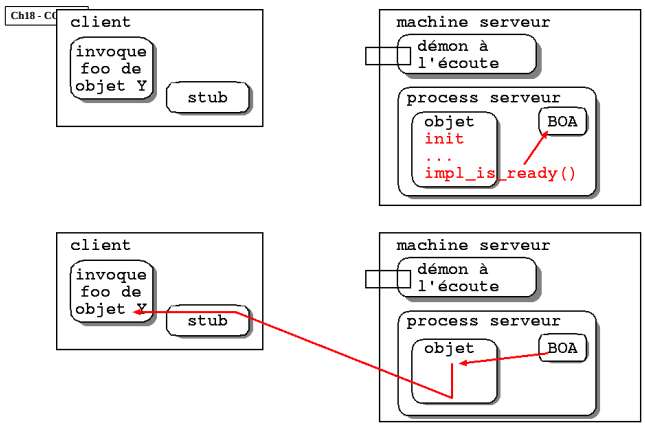
Le mécanisme d'échange sur le réseau étant, in fine , un socket, il faut bien qu'il y ait, côté serveur un process "démon" qui écoute sur un port de la machine serveur l'arrivée des requêtes.

Une implémentation possible d'un ORB peut alors être la suivante :

1. le client invoque un démon qui écoute sur un port connu,
2. le démon "fork" puis "exec" l'implémentation du serveur, incluant entre autre le BOA,
3. quand le serveur est lancé, il instancie les objets de service et appelle la fonction "impl_is_ready()" du BOA (ou bien "obj_is_ready() dans le cas d'une implémentation non partagée) ;
4. le BOA passe l'invocation à l'instance de l'objet,
5. l'objet sert la requête et retourne les résultats au client,

6. si le client fait une autre requête, elle arrive directement à l'implémentation lancée à l'étape 2,
7. la suite du dialogue se passe entre l'objet client et l'objet serveur.



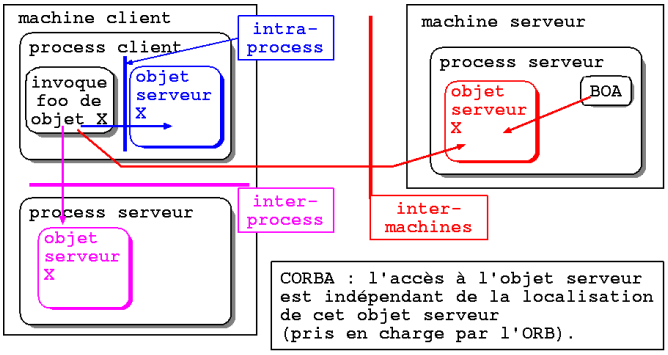


Accès transparent

L'un des buts de l'architecture OMA est que le programmeur du client puisse accéder à un objet **sans avoir besoin** de savoir à l'avance **où**, sur le réseau, **l'objet existe** : la localisation effective est cachée. Les objets doivent être "location transparent".

Un objet doit apparaitre comme étant le même, qu'il soit dans le même process, dans un autre process de la même machine ou dans un process sur une autre machine. Cette interaction doit se faire de la même façon si les objets sont gérés à travers deux ORBs de constructeurs différents.

Les détails de la localisation ne font pas partie du "contrat" proposé par l'objet et spécifié dans son interface IDL.



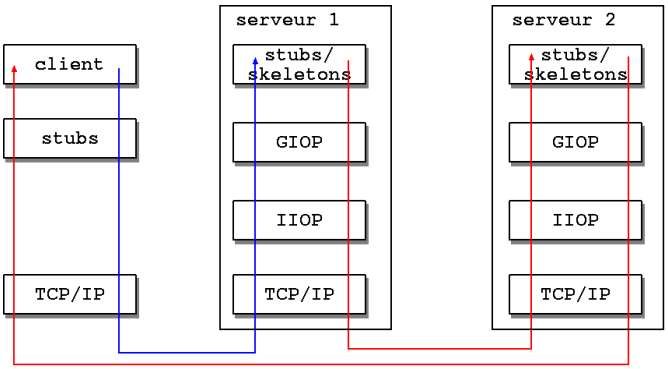
Interopérations entre ORBs

L'implémentation de CORBA étant libre, il faut un ensemble de protocoles bien spécifiés pour que deux implémentations puissent interagir.

C'est le rôle de **GIOP** (General Inter-ORB Protocol), qui spécifie le type et le format des messages que les ORBs doivent s'envoyer pour coopérer.

La sous-couche **IIOP** (Internet Inter-ORB Protocol) permet de "projeter" GIOP sur TCP/IP.

GIOP + IIOP => obligatoires pour une interopération entre ORBs "out-of-the-box".

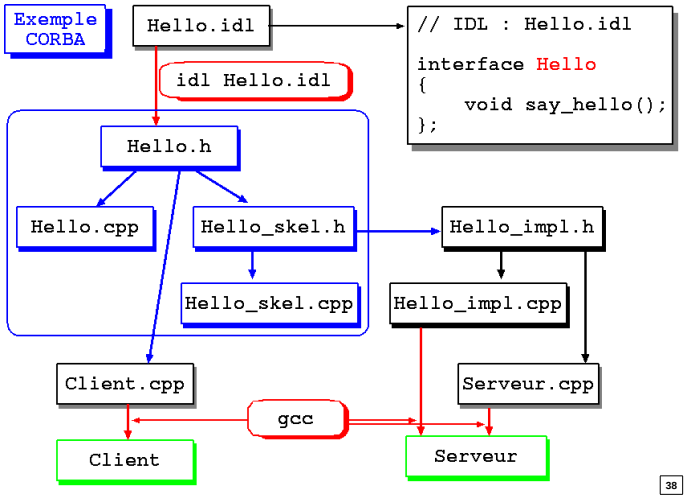


L'environnement et les Services Corba

Un ORB CORBA doit fournir plus que le simple mécanisme de passage de messages dont les objets ont besoin pour communiquer entre eux à travers des langages, des outils, des plateformes et des réseaux, tous hétérogènes.

Il fournit aussi l'environnement pour gérer ces objets, annoncer leur présence sur le réseau et décrire leurs métadonnées.

Tout ceci est mis en oeuvre par une série de "**services CORBA**", que presque tous les objets CORBA seront amenés à utiliser. On y trouve par exemple : **notification d'évènements**, persistance, cycle de vie, **nommage**, contrôle des accès concurrents, **transactions**, collections, temps, sécurité, etc ...



Exemple CORBA

```
ex-echo> nedit Hello.idl
ex-echo> echo $PATH
/progs/OB/bin:....

ex-echo> idl Hello.idl
ex-echo> ls
Hello.h     Hello_skel.cpp    Hello.cpp
Hello.idl   Hello_skel.h

ex-echo> pico Hello_impl.h
ex-echo> pico Hello_impl.cpp

ex-echo> nedit Server.cpp
ex-echo> nedit Client.cpp
ex-echo> ls *.cpp
Client.cpp  Hello.cpp  Hello_impl.cpp
Hello_skel.cpp  Server.cpp

ex-echo> gcc -c Hello.cpp -I.
ex-echo> gcc -c Hello_impl.cpp -I.-Wno-deprecated
ex-echo> gcc -c Hello_skel.cpp -I.
```



```
ex-echo> echo $LD_LIBRARY_PATH
/lib:/usr/lib:/usr/local/lib:/progs/OB/lib:/progs/JOB/lib

ex-echo> gcc -o Client Client.cpp Hello.o -I. \
-Wno-deprecated -lOB -lJTC -lpthread -ldl

ex-echo> gcc -o Server Server.cpp Hello.o Hello_skel.o \
Hello_impl.o -I. -Wno-deprecated -lOB -lJTC -lpthread -ldl

ex-echo> ./Server
Hello World!

ex-echo> ./Client

ex-echo> ls Hello.ref
Hello.ref
ex-echo> more Hello.ref
IOR:01aeefbf0e00000049444c3a48656c6c6f3a312e30000000010000
000000000700000000010102000d000000747578612e736d652e757463
00000d9625000000abacab3131313134343532363032005f526f6f7450
4f410000cafebabe426d327a000000000000000010000000100000002000
000001366600010001000200000002000010001000105090101000100000
0000010100
```

```
// C++  Client.cpp
#include <OB/CORBA.h>
#include <Hello.h>
#include <fstream.h>
int run(CORBA::ORB_ptr);

int main(int argc, char* argv[])
{  // Same as the server
   int status = EXIT_SUCCESS;
   CORBA::ORB_var orb;
   try
   {  orb = CORBA::ORB_init(argc, argv);
      status = run(orb);
   }
   catch(const CORBA::Exception&)
   {
      status = EXIT_FAILURE;
   }
   if(!CORBA::is_nil(orb))
   { try {  orb -> destroy(); }
         catch(const CORBA::Exception&)
         {  status = EXIT_FAILURE; }
   }
   return status;
}
```

inclusion des symboles de la bibliothèque ORB et des définitions générées par le compilateur IDL

initialiser l'ORB

puis, appeler notre fonction de travail

détruire la connexion à l'ORB

41

```
//--- run() ---
int run(CORBA::ORB_ptr orb)
{
   const char* refFile = "Hello.ref";
   ifstream in(refFile);
   char s[2048];
   in >> s;

   CORBA::Object_var obj = orb -> string_to_object(s);

   Hello_var hello = Hello::_narrow(obj);

   hello -> say_hello();

   return 0;
}
```

Récupération de la forme "stringifiée" de l'identifiant de l'objet serveur.

Cet identifiant stringifié est transformé en référence sur un objet

La référence sur un objet est convertie en une référence sur un objet de type Hello

Sur cet objet de type Hello on peut appliquer la fonction déclarée dans l'interface.

Les variables Xxxx_var sont des "smart pointers" qui relachent la mémoire allouée dès que la variable n'est plus visible ("out of scope").

42

```
// Serveur.cpp
#include <OB/CORBA.h>
#include <Hello_impl.h>
#include <fstream.h>
#include <iostream.h>
int run(CORBA::ORB_ptr);

int main(int argc, char* argv[])
{
   int status = EXIT_SUCCESS;
   CORBA::ORB_var orb;
   try
   {  orb= CORBA::ORB_init(argc, argv);
      status = run(orb);
   }
   catch(const CORBA::Exception&)
   {
      status = EXIT_FAILURE;
   }
   if(!CORBA::is_nil(orb))
   { try {  orb -> destroy(); }
     catch(const CORBA::Exception&)
         { status = EXIT_FAILURE; }
   }
   return status;
}
```

inclusion des symboles de la bibliothèque ORB et des définitions générées par le compilateur IDL

initialiser l'ORB

puis, appeler notre fonction de travail

détruire la connexion à l'ORB

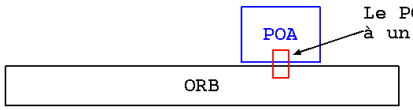
43

```
//----- run() -----
int run(CORBA::ORB_ptr orb)
{
  CORBA::Object_var poaObj =
          orb -> resolve_initial_references("RootPOA");
  PortableServer::POA_var rootPoa =
          PortableServer::POA::_narrow(poaObj);
  PortableServer::POAManager_var manager =
          rootPoa -> the_POAManager();                    1

  Hello_impl* helloImpl = new Hello_impl();
  PortableServer::ServantBase_var servant = helloImpl;
  Hello_var hello = helloImpl -> _this();                 2

  CORBA::String_var s = orb -> object_to_string(hello);
  const char* refFile = "Hello.ref";
  ofstream out(refFile);
  out << s << endl;
  out.close();                                            3

  manager -> activate();

  orb -> run();                                           4

  return EXIT_SUCCESS; }
```

Créer la forme "stringifiée" de l'identifiant de l'objet serveur.

44

```
//----- run() -----
int run(CORBA::ORB_ptr orb)
{
  CORBA::Object_var poaObj =
          orb -> resolve_initial_references("RootPOA");
  PortableServer::POA_var rootPoa =
          PortableServer::POA::_narrow(poaObj);
  PortableServer::POAManager_var manager =
          rootPoa -> the_POAManager();                    1

.....Récupérer une référence sur le RootPOA et la
}     transformer en objet de type POA.
      Puis récupérer la référence sur le manager du POA.
```



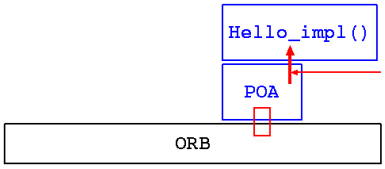
Le POA se connecte à un port TCP.

45

```
//----- run() -----
int run(CORBA::ORB_ptr orb)
{ .....

  Hello_impl* helloImpl = new Hello_impl();
  PortableServer::ServantBase_var servant = helloImpl;
  Hello_var hello = helloImpl -> _this();                 2
...
}
```

Instancier un objet d'implémentation.
Déclarer que cet objet est un servant du POA.

Ce servant va incarner un "objet CORBA" par l'appel à _this().



appel à objet Hello : le POA, par le servant appelle l'implémentation Hello_impl()

46

```
//----- run() -----
int run(CORBA::ORB_ptr orb)
{ .....

  CORBA::String_var s = orb -> object_to_string(hello);
  const char* refFile = "Hello.ref";
  ofstream out(refFile);
  out << s << endl;
  out.close();                                            3
...
}
```

Créer la forme "stringifiée" de l'identifiant de l'objet serveur.

```
ex-echo> ls Hello.ref
Hello.ref
ex-echo> more Hello.ref
IOR:01aeefbf0e00000049444c3a48656c6c6f3a312e30000000..
....
```

Pour accéder à un objet CORBA sur le serveur, le client a besoin d'une référence CORBA.
Ici, on passe cette référence au client par l'intermédiaire de sa forme "string" dans un fichier.
Dans une "vraie" application, on utilise un service de nommage (annuaire).

47

```
//----- run() -----
int run(CORBA::ORB_ptr orb)
{ .....

  manager -> activate();

  orb -> run();                                           4

  return EXIT_SUCCESS;
}
```

On active le manager pour lui indiquer qu'il peut accepter les requêtes des clients.

On appelle la boucle d'attente de l'ORB.

Il n'y a pas de retour de cette fonction, sauf si une fonction exécute un "shutdown" de l'ORB.

Après cet appel, le serveur est en attente des appels des clients.

Tout le traitement se fait dans les fonctions des objets serveurs déclarés (comme les callbacks d'une interface graphique ou comme un serveur RPC ou un serveur JavaRMI).

48

```
// C++ Hello_impl.h     dérivée du skeleton POA_Hello
//                      définie dans Hello_skel.h
#include <Hello_skel.h>

class Hello_impl : public POA_Hello, public
      PortableServer::RefCountServantBase
{
public:
        virtual void say_hello()
                throw(CORBA::SystemException);    };
```

```
// C++  Hello_impl.cpp
// code de la classe d'implémentation
#include <iostream.h>
#include <OB/CORBA.h>
#include <Hello_impl.h>

void Hello_impl::say_hello()
        throw(CORBA::SystemException)
{
   cout << "Hello World!" << endl;    }
```

49

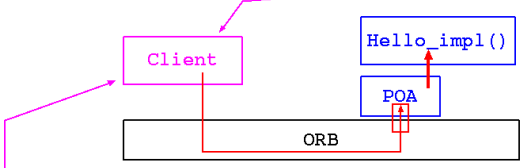
Implémentation en java

```
ex-echo> which jidl
/progs/OB/bin/jidl

ex-echo> jidl --package hello Hello.idl
ex-echo> ls hello/
HelloHelper.java  Hello.java              HelloPOA.java
HelloHolder.java  HelloOperations.java    _HelloStub.java
ex-echo> cd hello
ex-echo/hello> nedit Hello_impl.java
ex-echo/hello> nedit Server.java &
ex-echo/hello> nedit Client.java &

ex-echo/hello> echo $JAVA_HOME
/progs/j2sdk
ex-echo/hello> setenv CLASSPATH .:/progs/JOB/lib/
ex-echo/hello> cd ..
ex-echo> javac hello/*.java
ex-echo> setenv CLASSPATH .:/progs/JOB/lib/:/progs/JOB/ob/lib
ex-echo> java -Xbootclasspath/p:$CLASSPATH hello.Server
Hello World!

ex-echo> setenv CLASSPATH .:/progs/JOB/lib/:/progs/JOB/ob/lib
ex-echo> java -Xbootclasspath/p:$CLASSPATH hello.Client
```

à écrire

50

```
ex-echo> ./Server
Hello World!

ex-echo> java -Xbootclasspath/p:$CLASSPATH hello.Client
```




```
ex-echo> java -Xbootclasspath/p:$CLASSPATH hello.Server
Hello World!

ex-echo> ./Client
```

Serveur C++ <-> Client java ou Serv. java <-> Cli. C++

51

Serveur C++ <->Client java ou Serv. java <->Cli. C++

```
ex-ob-echo> iordump Hello.ref
[No valid IOR found on the command line, assuming -f]
IOR #1:
byteorder: big endian
type_id: IDL:Hello:1.0
Profile #1: iiop
iiop_version: 1.2
host: tuxa.sme.utc
port: 38443
object_key: (37)
171 172 171  49  49  49  49  52 "...11114"
 52  54  48  56  52  51   0  95 "460843._"
 82 111 111 116  80  79  65   0 "RootPOA."
  0 202 254 186 190  66 109  82 ".....BmR"
171   0   0   0   0            ".....".
Native char codeset:
  "ISO 8859-1:1987; Latin Alphabet No. 1"
Char conversion codesets:
  "ISO 646:1991 IRV (International Reference Version)"
  "X/Open UTF-8; UCS Transformation Format 8 (UTF-8)"
Native wchar codeset: "ISO/IEC 10646-1:1993; UTF-16,
  UCS Transformation Format 16-bit form"
```

L'IOR est un identifiant "universel" (compris par toutes les implémentations d'ORB Corba). Il contient toute l'information pour retrouver l'objet CORBA.

52

```
// Java 2  Client.java
package hello;
public class Client
{  public static void main(String args[])
   {  // Same as the server
      java.util.Properties props= System.getProperties();
      props.put("org.omg.CORBA.ORBClass",
                "com.ooc.OBServer.ORB");
      props.put("org.omg.CORBA.ORBSingletonClass",
                "com.ooc.CORBA.ORBSingleton");
      int status = 0;
      org.omg.CORBA.ORB orb = null;
      try
      {   orb = org.omg.CORBA.ORB.init(args, props);
          status = run(orb); }
      catch(Exception ex)
      {   ex.printStackTrace(); status = 1; }
      if(orb != null)
      { try
        {  orb.destroy(); }
        catch(Exception ex)
        {  ex.printStackTrace(); status = 1; }
      }
      System.exit(status);
   }
```

53

```
//----- run() -----
   static int run(org.omg.CORBA.ORB orb)
   { org.omg.CORBA.Object obj = null;
    try
 {  String refFile = "Hello.ref";
    java.io.BufferedReader in =
            new java.io.BufferedReader(
       new java.io.FileReader(refFile));
    String ref = in.readLine();
    obj = orb.string_to_object(ref);
 }
 catch(java.io.IOException ex)
 {  ex.printStackTrace();
    return 1; }

 Hello hello = HelloHelper.narrow(obj);

 hello.say_hello();

 return 0;
   }
}
```

récup. réf sur objet

convertir référence en objet de type Hello

invoquer la fonction de l'objet Hello instancié par le serveur

54

```
// Java  hello/Server.java
package hello;
public class Server
{ public static void main(String args[])
  { java.util.Properties props= System.getProperties();
    props.put("org.omg.CORBA.ORBClass",
              "com.ooc.OBServer.ORB");
    props.put("org.omg.CORBA.ORBSingletonClass",
              "com.ooc.CORBA.ORBSingleton");
    int status = 0;
    org.omg.CORBA.ORB orb = null;
    try
    { orb = org.omg.CORBA.ORB.init(args, props);
      status = run(orb); }
    catch(Exception ex)
    {  ex.printStackTrace(); status = 1; }
    if(orb != null)
    { try
      {  orb.destroy(); }
      catch(Exception ex)
      {  ex.printStackTrace(); status = 1; }
    }
    System.exit(status);
  }
```

55

```
//----- run() -----
   static int run(org.omg.CORBA.ORB orb)
            throws org.omg.CORBA.UserException
   { org.omg.PortableServer.POA rootPOA =
     org.omg.PortableServer.POAHelper.narrow(
         orb.resolve_initial_references("RootPOA"));
     org.omg.PortableServer.POAManager manager =
                        rootPOA.the_POAManager();
     Hello_impl helloImpl = new Hello_impl();
     Hello hello = helloImpl._this(orb);
   try
   { String ref = orb.object_to_string(hello);
     String refFile = "Hello.ref";
     java.io.PrintWriter out = new java.io.PrintWriter(
                 new java.io.FileOutputStream(refFile));
     out.println(ref);
     out.close();
   }
   catch(java.io.IOException ex)
   {  ex.printStackTrace(); return 1; }
   manager.activate();
   orb.run();
   return 0;
   }
 }
```

client

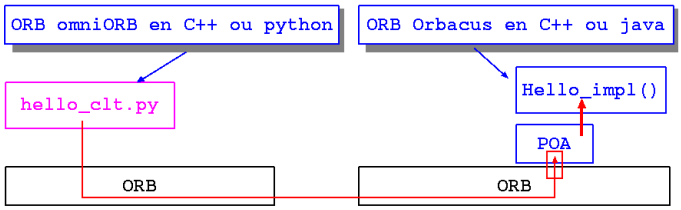
mettre le serveur en attente des appels des clients

56

```
// Java Hello_impl.java
package hello;
public class Hello_impl extends HelloPOA
{
    public void say_hello()
    {
       System.out.println("Hello World!");}
}
```

```
// Generated by the ORBacus IDL to Java Translator
public abstract class HelloPOA
...
public org.omg.CORBA.portable.OutputStream
   _invoke(String opName,
.....
   final String[] _ob_names =
   { "say_hello"
   };
   while(_ob_left < _ob_right)
     int _ob_res= _ob_names[_ob_m].compareTo(opName);

   switch(_ob_index)
      {  case 0: // say_hello
.....
```


Interactions entre ORBs de constructeurs différents



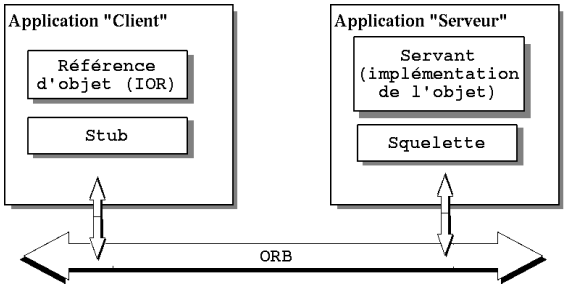

```
ex-echo> setenv PATH /usr/local/omniORB-4.0.5/bin:$PATH
ex-echo> omniidl -bpython Hello.idl
ex-echo> nedit hello_clt.py
ex-echo> ./Server
Hello World!

ex-echo> more Hello.ref
IOR:0105fcbf0e00......

ex-echo> ./hello_clt.py IOR:0105fcbf0e00......
```



```
#!/usr/bin/env python
import sys
from omniORB import CORBA  # Import the CORBA module
import _GlobalIDL # Import stubs for the global module

# Initialise the ORB
orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
# Get the IOR of an Echo object from the command line
ior = sys.argv[1]
# Convert the IOR to an object reference
obj = orb.string_to_object(ior)

# Narrow reference to an Echo object
eo  = obj._narrow(_GlobalIDL.Hello)
if eo is None:
    print "Object reference is not an Hello"
    sys.exit(1)
if len(sys.argv) == 3:
    message = sys.argv[2]
else:
    message = "Hello from Python"

result = eo.say_hello()  # Invoke say_hello operation
print "I said '%s'." % (message)
```


CORBA : le POA, fonctions

Le dialogue CORBA :
[Client–>Stub–>ORB–>..–>ORB–>Skeleton–>Server]
peut se résumer par la figure ci-dessous :



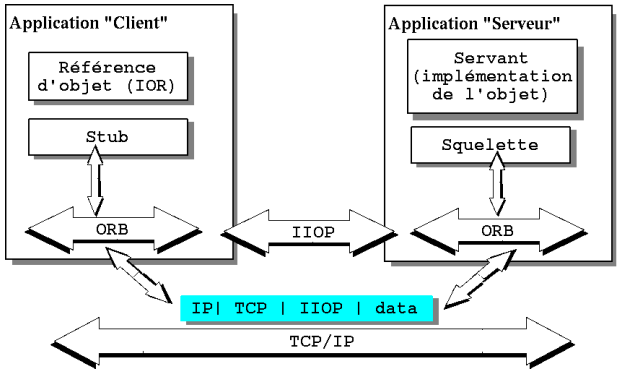
En fait, ce schéma est la vision "fonctionnelle" de CORBA. Il masque deux éléments importants :

- la **communication** entre le client et le serveur,
- la **mise en service et l'appel aux objets** depuis l'ORB.

L'ORB est en pratique une bibliothèque linkée avec le client d'une part et le serveur d'autre part.

Ces deux instances de la **bibliothèque-ORB** vont communiquer entre elles en échangeant des **messages dans un protocole qui leur est propre** et qui fait partie de la spécification de Corba.

C'est la définition de ce protocole qui permet à deux bibliothèque-ORB de deux constructeurs différents de dialoguer entre elles.

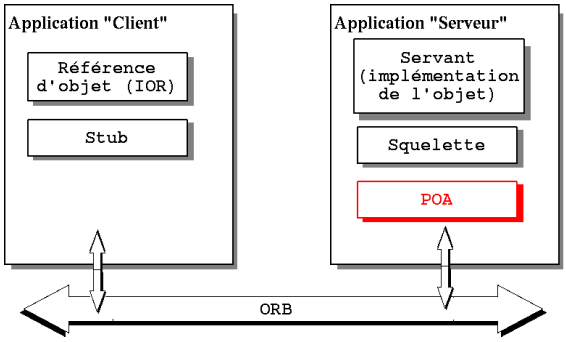


Ce protocole est l'un des protocoles de la famille **GIOP**. Si les machines supportant le client et le serveur utilisent TCP/IP comme liaison réseau, alors le protocole utilisé par les ORBs est "**IIOP**".

Dans l'implantation d'un serveur Corba on trouve l'autre élément supplémentaire : **l'adaptateur d'objets** ("object adapter").

Le **POA** (Portable Object Adaptor) sert d'intermédiaire entre l'ORB et l'implémenatation du serveur.

En février 1998, la spécification CORBA 2.2 a retiré le BOA au profit du POA. Le POA fournit en plus un ensemble d'outils permettant d'affiner le comportement du serveur.



Que fait le POA ?

Son rôle est d'assurer la **gestion des différents objets** qui se situent à l'intérieur de l'application serveur. Il va :

- activer les objets (en fonction des demandes des clients),
- transmettre une invocation d'un client vers un objet particulier,
- créer et détruire des objets,
- déterminer l'état d'un objet,
- ...

Quand on passe du BOA au POA, seul le code du serveur a besoin d'être modifié. Les clients restent identiques.

Avec le POA on utilise une nouvelle terminologie pour prendre en compte les fonctions supplémentaires. Une application utilisant un POA possède donc les éléments suivants :

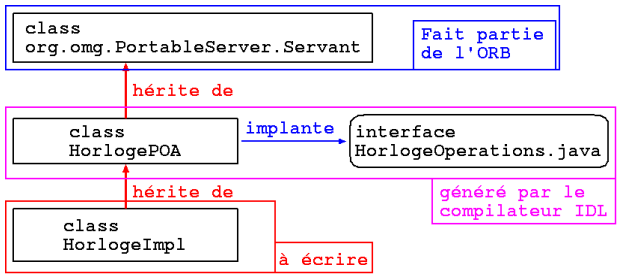
- **servant :** représente l'objet créé sur/par le serveur (anciennement appelé "implémentation de l'objet" ;
- **serveur :** c'est l'application qui loge un ensemble de servants et permet aux applications clientes d'invoquer ces objets ;
- **client :** c'est l'application qui invoque des services proposés par les objets servants ;
- **référence d'objet :** ce que le client voit d'un objet servant par l'intermédiaire de l'interface IDL ; le client utilise une référence d'objet pour accéder à un servant par l'intermédiaire du "stub", de l'ORB, du POA et du squelette qui, ensemble, masquent au client la non-proximité de l'objet réel. Cette référence est appelée **IOR** (Interoperable Object Reference).

Un servant avec une approche par héritage

Dans le cas du POA, les fichiers générés par le compilateur IDL sont un peu différents. On trouve pour XXX.idl :

- _XXXStub.java : la souche du client,
- XXX.java : l'interface XXX,
- XXXOperations.java : l'interface d'opérations de XXX,
- XXXPOA.java : le squelette côté serveur,
- XXXHelper.java et XXXHolder.java les classes Helper et Holder de XXX.

L'implantation du servant va hériter du squelette. Mais ici, il y a une différence fondamentale par rapport au BOA : le squelette n'implante pas l'interface XXX, mais **directement XXXOperations**.



Le servant (la classe XXXImpl) n'a plus de lien avec l'interface XXX.

Le but est d'introduire une **séparation** entre la **définition** fonctionnelle de l'objet Corba (symbolisée par l'interface XXX) et **l'implé-**

mentation qui en est faite (XXXImpl).

Ceci, afin de pouvoir proposer différentes politiques de gestion des instanciations des objets servants au sein du serveur.

Ceci sera fait par les **politiques du POA** :

- politique de threading,
- politique de durée de vie,
- politique d'unicité d'identité d'objet,
- politique d'assignation d'identité,
- politique de rétention des servants,
- politique de gestion des requêtes,
- politique d'activation implicite.

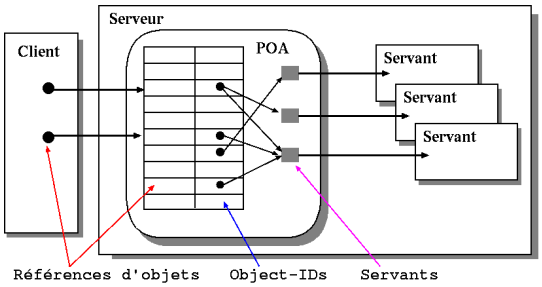
Pour cela on va modifier l'association**référence <–> instance**.

Avec l'ancien BOA, on avait une relation bijective entre la référence d'objet passée/utilisée par le client et l'instance d'objet crée/appelée par le serveur :

```
Serveur
 xxx = new XXXImpl();
 ref = object_to_string(xxx);
  |
  \------ "ref" ------> Client
                        remotexxx = string_to_object(ref);
                        remotexxx.operation();
class XXXImpl                          |
public void operation () <--- invoque --/
```

Pour chaque "**new XXXImpl()**" il y a une référence et une seule.

Dans le cas du POA, on veut introduire une **table d'indirection**, gérée par le POA, et distinguer **trois notions** :
référence d'objet <–> Object-ID <–> Servant (instance).

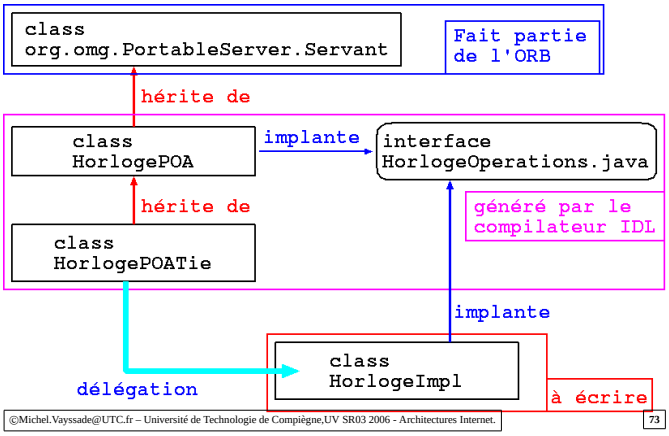


Un servant avec une approche par délégation

Pour les servants développés en Java, on peut, pour les mêmes raisons que dans le BOA vouloir utiliser l'approche par délégation.

L'utilisation se fait en passant un référence sur une instance de XXXImpl à une instance de la classe de délégation :

```
HorlogeImpl   hor = new HorlogeImpl();
HorlogePOATie tie = new HorlogePOATie(hor);
```



RootPOA et arborescence de POA

L'une des nouveautés du POA est que l'on puisse en avoir plusieurs, organisés en une **hiérarchie à partir du RootPOA**. Chaque POA peut avoir plusieurs POA-fils .

L'intérêt est lié au fait que chaque POA peut avoir des règles ou politiques de fonctionnement différentes.

Chaque POA gère un ensemble de servants et leur transmet les invocations en provenance des clients. La façon dont les servants et les invocations sont gérées est totalement paramétrable.

Chaque POA est géré par un **gestionnaire de POA**. Il y a au moins un gestionnaire de POA et il peut être unique et gérer tous les POAs du serveur.

Le gestionnaire de POA ("**POAManager**") contrôle le traitement des requêtes en provenance des clients à destination du POA.

Suivant l'état dans lequel il est, les requêtes seront rejetées, mises en attente ou traitées.

Avant que des requêtes puissent être traitées il faudra donc faire passer le POA dans l'état actif.

La classe org.omg.PortableServeur fournit les méthodes pour convertir de l'un vers l'autre les trois types d'identifiants d'objets et pour associer un servant à un POA.

Questions sur Ch18 - CORBA

➤ Montrer les différents cas de réussite ou d'échec lors d'un appel à distance.
Réponse : diapos 3 et 4.

➤ Qu'est-ce que Corba ?
Réponse : diapo 6.

➤ Quel est le mécanisme de base de Corba ?
Réponse : diapo 7.

➤ Quelle est la cible d'une requête Corba ?
Réponse : diapo 15. Une méthode d'un objet, à travers une référence sur l'objet.

➤ Quel est le langage de définition des interfaces Corba ?

Réponse : diapo 16.

➤ Décrire le mécanisme d'invocation d'objet distant à travers un ORB Corba.
Réponse : diapos 21 et 22.

➤ Qu'est-ce que l'ORB apporte par rapport au RPC ?
Réponse : diapos 26 à 28.

➤ Expliquer les 3 lignes de codes ci-dessous extraites d'un client corba.

```
CORBA : :Object_var obj = orb -> string_to_object(s) ;
Hello_var hello = Hello : :_narrow(obj) ;
hello -> say_hello() ;
```

Réponse : diapo 42.

➤ Est-il possible dans une application Corba d'utiliser des langages

différents pour clients et serveurs et aussi des bibliothèques de fournisseurs différents côté client et côté serveur ?
Réponse : oui aux deux questions, Corba a précisément été fait pour peermettre cela.

Ch18 - CORBA

Fin du chapitre.

(diapo vide)

Ch19 - Corba : IDL

- CORBA : l'IDL, les types, compilation
- Création du serveur et du client. Références d'objets.
- Classes Helper et fonction narrow
- IDL et java : classes Holder
- Héritage des interfaces IDL
- Corba : renvoi d'exceptions
- Corba : Héritage et délégation.

De l'IDL à l'application

Un serveur CORBA est une application qui met à disposition des objets. Chacun de ces objets implante une interface décrite au moyen d'un fichier IDL.

La description IDL du serveur décrit ce que le serveur propose aux clients.

Ce fichier IDL est compilé pour générer des fichiers sources qui devront être compilés (au sens des langages (C,C++,Java,...) et linkés avec le serveur d'une part et le client d'autre part.

Dans l'implantation par héritage, la classe d'implantation (écrite par l'auteur du serveur) va hériter du squelette (généré par le compilateur IDL).

Exemple d'interface **Horloge.idl :**

```
interface Horloge
{ string get_time();
};
```

Plus généralement :

```
interface Nom_de_l_interface
{ // corps de l'interface
  attribute type un_attribut;
  ....
  type une_fonction (params);
  ....
};
```

Avec pour les fonctions :


```
    Type_de_retour nom_fonction ( liste paramètres );
```

Et pour les structures :

```
struct nom_structure
{
  type x;
  type y;
  ...
};
```

On peut créer ses propres types :

```
     typedef sequence vecteur;
     vecteur v;
ou
typedef struct une_structure
```



```
{
  type x;
  ....
};
une_structure s;
```


Les types de base de CORBA sont les suivants :

```
boolean       octet
char          wchar           string          wstring
short         unsigned short      == short_java
long          unsigned long       == int_java
long long     unsignedlong long   == long_java
float         double              == float_java et double_java
long double                       ## pas_en_java
void  == retour vide d'une fonction
```


Compilation de la description IDL de l'interface

Le compilateur "idl2java" est utilisé pour compiler la description IDL de l'interface :

```
idl2java Horloge.idl
==> va créer plusieurs fichier parmi lesquels
    2 ont des noms normalisés :
        _HorlogeImplBase.java   amorce serveur = squelette
        _HorlogeStub.java       amorce client  = souche

ou en général pour  Xyyy.idl :
        _XyyyImplBase.java
        _XyyyStub.java
```

On va souvent appeler la classe d'implantation XyyyImpl.java (bien

que ce ne soit pas obligatoire, on peut très bien l'appeler "truc.java").

Cette classe va hériter du squelette _HorlogeImplBase.java et contenir le code effectuant le travail "promis" par l'interface :

```
HorlogeImpl.java =
public class HorlogeImpl extends _HorlogeImplBase
{
  publis string get_time()
  {
    java.util.Calendar cal = new java.util.GregorianCalendar();
    String str = ""+ cal.get(java.util.Calendar.HOUR_OF_DAY);    str =
  }
}
```

Le client va utiliser cette fonction de façon classique en instanciant un objet Horloge et en invoquant la fonction get_time() de cet objet :

```
public class Client
{
 public static void main (String  args)
 {
 ..........// initialisations de CORBA
 //instanciation_à_distance_de_l_objet_CORBA_HorlogeImpl
 obj = orb.string_to_object(refobjhor);
 Horloge hor = HorlogeHelper.narrow(obj);
 System.out.println ( "Heure=" + hor.get_time() );
 }
}
```


Création du serveur

Avant de pouvoir utiliser l'objet, il va falloir créer l'application serveur qui va être linkée avec la bibliothèque CORBA pour pouvoir **se mettre à l'écoute des connexions** des clients sur le réseau, **instancier les objets** demandés et **passer les appels** (invocations de méthodes) des clients **aux instances** des objets.

Le serveur va comporter cinq étapes :

1. initialiser l'ORB
2. initialiser l'adaptateur d'objets (BOA ou POA)
3. créer un objet d'implantation (ici de type HorlogeImpl)
4. enregistrer et activer cet objet auprès de l'adaptateur d'objets
5. attendre les demandes des clients

La bibliothèque CORBA se présente comme un package contenant une classe
org.omg.CORBA.ORB. Elle possède une fonction init() pouvant être appelée sous deux formes :

- org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init() ;
- org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(String [] args, java.util.Properties prop) ;

La deuxième forme permet de passer à l'ORB des arguments en provenance de la ligne de commande :

```
 public static void main (String  args)
 {
   org.omg.CORBA.ORB orb= org.omg.CORBA.ORB.init(args, null);
   .....
 }
```


L'initialisation de l'adaptateur d'objet dépend de l'adaptateur. Dans le cas d'un BOA elle est dépendante de l'ORB utilisé, mais la plupart ont une fonction ressemblant à :
org.omg.CORBA.BOA boa = orb.BOA_init (args , null) ;

L'initialisation d'un adaptateur d'objet de type "POA" est normalisée. Le code peut être plus général et devient portable d'un ORB à un autre ORB.

Pour créer un objet d'implémentation, il suffit d'invoquer le constructeur de l'objet :

```
HorlogeImpl hor = new HorlogeImpl() ;
```

Il faut ensuite enregistrer et activer cet objet auprès du POA :

```
Hello hello = helloImpl._this(orb) ; // enregistrer manager.act
// activer
```


Ceci ajoute l'objet à la liste des objets connus et prêts à répondre aux demandes des clients (c'est un peu l'équivalent de l'enregistrement d'un serveur RPC dans le portmapper).

Ensuite, il ne reste qu'à attendre les demandes des clients :

```
orb.run() ;
```

On ne ressort de cet appel que si une erreur se produit ou si le serveur est arrêté.

Création du client

Le client ne comporte que trois étapes :

1. initialiser l'ORB,
2. récupérer la référence de l'objet que l'on veut utiliser,
3. utiliser l'objet (invoquer des méthodes de l'objet).

Le code du client aura la forme suivante :

```
public class Client
{
 public static void main (String  args)
 {
 // initialisations de CORBA
 org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init( args, null );
```



```
 // récupérer la référnce de l'objet
 refobjdist = récupération_de_la_référence("identifiant_objet")
 objetdistant = traduction_de_la_référence(refobjdist)
 // convertir la réf. en objet du type souhaité
 Horloge hor = HorlogeHelper.narrow (obj);
 // utiliser l'objet
 System.out.println ( "Heure=" + hor.get_time() );
 }
}
```

Il faut maintenant détailler un peu les références d'objets et leur échange entre serveurs et clients ainsi que les relations entre classes utilisateurs et classes CORBA et les notions de "Helper" et "narrow".

Les références d'objets

Une référence d'objet contient **toutes les données nécessaires pour retrouver cet objet**. On peut la comparer à une URL.

Une référence est attribuée à un objet lors de l'opération helloImpl._this(orb) ;.

Elle permet de localiser l'objet : nom de la machine hôte du serveur, numéro du port sur lequel le serveur écoute les requêtes et un identifiant unique qui permet de distinguer l'objet des autres objets gérés par le même serveur.

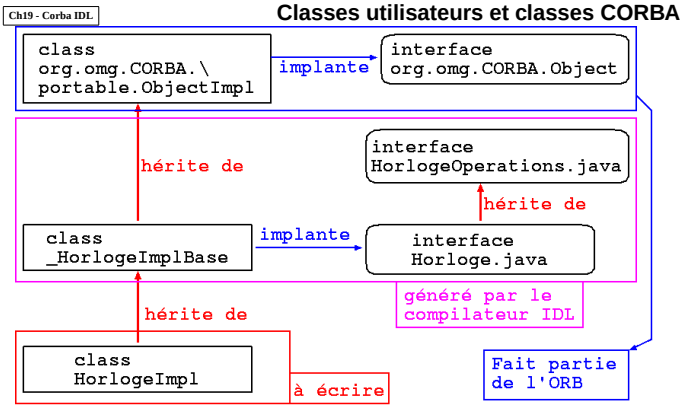
La **référence est donc générée par le serveur**. Il faut donc trouver un moyen de la **transmettre aux clients** potentiels.

Dans les applications complexes déployées à grande échelle, on utilisera un **service de noms** ("naming service") fournissant un **annuaire des objets disponibles**.

Pour simplifier cette opération dans les premiers exemples on va utiliser deux fonctions de service de CORBA : **object_to_string** et **string_to_object** permettant de traduire une référence en une chaine de caractères et inversement.

Nous ferons passer cette chaine de caractères du serveur au client par un moyen extérieur à CORBA.

Classes utilisateurs et classes CORBA



Notions de "Helper" et "narrow"

L'interface org.omg.CORBA.Object représente la référence d'objet.

L'objet d'implantation (ici HorlogeImpl.java) hérite de _HorlogeImplBase.

La classe org.omg.CORBA.portable.ObjectImpl implante les méthodes de cette interface, et permet à l'adapteur d'objet de récupérer les informations caractérisant la référence de l'objet.

object_to_string :

```
HorlogeImpl hor = new HorlogeImpl();
String ref = orb.object_to_string(hor);
```

string_to_object :


```
org.omg.CORBA.Object obj = orb.string_to_object(ref);
```

Lorsque l'on recrée la référence à partir de la chaine de caractère, on obtient une référence accessible uniquement à travers l'interface org.omg.CORBA.Object.

Il va falloir convertir cette référence "générale" en une référence "spécifique" sur un objet de type (ici) Horloge.

Or "Horloge" n'hérite pas directement de "org.omg.CORBA.Object". On ne peut donc pas faire un transtypage (cast).

Pour contourner ce problème, on utilise une classe spéciale, générée par le compilateur IDL à partir de la description de l'interface Horloge. Cette classe s'appelle "**classe Helper**". Elle porte le nom de l'interface suivi de "Helper" : ici HorlogeHelper.

La classe "Helper" contient en particulier une **méthode "narrow"** spécialement construite pour convertir une référence d'objet en une référence vers l'interface Horloge.

Il faut que l'objet que l'on donne en paramètre de "narrow" implante l'interface vers laquelle on essaie de le convertir.

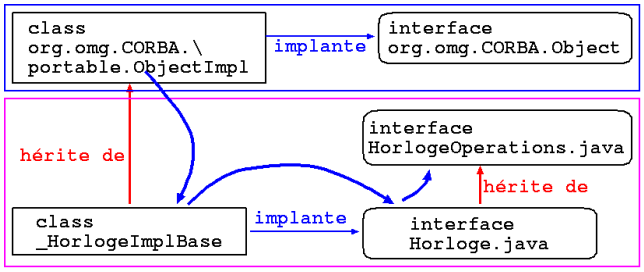
Ce sera le cas ici si l'objet est bien à l'origine du type HorlogeImpl qui hérite de _HorlogeImplBase qui elle-même implante Horloge.

Dans le cas contraire, lors de l'essai de conversion, on reçoit une exception de type :
org.omg.CORBA.BAD_PARAM (exception système CORBA qui hérite de
org.omg.CORBA.SystemException).

Que fait "narrow" ?

La fonction "narrow" convertit la référence en une référence vers la souche qui implante l'interface Horloge.

Quand on utilise "narrow" sur le client, on a la situation suivante :



La souche contient donc toutes les méthodes de l'interface que l'on pourra alors invoquer :

```
// convertir la réf. en objet du type souhaité
```



```
Horloge hor = HorlogeHelper.narrow (obj);
// utiliser l'objet
System.out.println ( "Heure=" + hor.get_time() );
```

Remarque : À chaque appel à une méthode d'un objet distant, des erreurs peuvent se produire (serveur planté, timeout dû au réseau, etc ...), c'est pourquoi il est prudent de mettre ces appels dans un bloc try/catch :

```
try
{ Horloge hor = HorlogeHelper.narrow (obj);
  String heure = hor.get_time();
}
catch (org.omg.CORBA.SystemException ex)
{  ex.printStackTrace();  }
```

La compilation et l'exécution

```
horloge> ls
Client.java   HorlogeImpl.java   Serveur.java   Horloge.idl

horloge> java org.openorb.compiler.IdlCompiler\
 -d . -boa -notie Horloge.idl

horloge> ls
Client.java          HorlogeHolder.java      Serveur.java
Horloge.idl          HorlogeImpl.java        _HorlogeImplBase.java
Horloge.java         HorlogeOperations.java  _HorlogeStub.java
HorlogeHelper.java

horloge> javac *.java
```

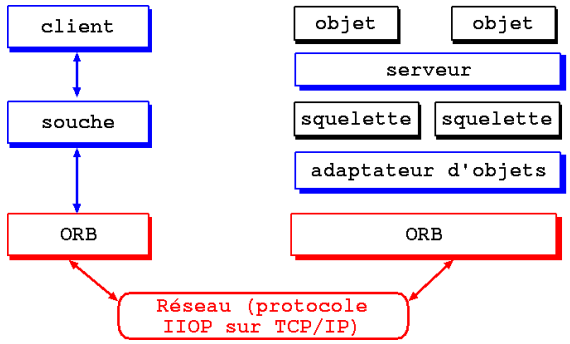


```
horloge>
running serveur with :

horloge> java -Dorg.omg.CORBA.ORBClass=org.openorb.CORBA.ORB
-Dorg.omg.CORBA.ORBSingletonClass=org.openorb.CORBA.ORBSingleton
Serveur
 Le serveur est pret...

running client with :

horloge> java -Dorg.omg.CORBA.ORBClass=org.openorb.CORBA.ORB
-Dorg.omg.CORBA.ORBSingletonClass=org.openorb.CORBA.ORBSingleton
Client
 Heure =  16:27
```


Conclusion partielle

L'ORB traduit les requêtes entre clients et serveurs en un protocole général indépendant du langage d'implantation, ce qui permet d'appeler un objet java depuis un objet C++ ou l'inverse.

On a la structure suivante entre client et serveur :



CORBA : les bases de l'IDL (Interface Definition Language)

Surcharge de noms interdite en IDL

Si une interface définit déja une fonction string = get_time(), et que l'on veut ajouter une fonction void get_time(hh,mm), il **faudra** lui donner un autre nom, la surcharge de noms étant interdite en IDL.

On a vu qu'il était possible de définir des structures en IDL :

```
struct nom_structure
{ type x;
  type y;
  ...
};
```


Ces structures peuvent être passées en paramètres des fonctions ou bien reenvoyées comme valeur de retour. Par exemple :

```
struct heure_min
{ long hh;
  long mm;
};
interface Time_hm
{  heure_min get_time_hm();
};
```

Traduction des structures IDL en Java

Quand on compile la structure IDL ci-dessus, on va générer **trois** nouveaux fichiers :
heur_min.java , heur_minHelper.java et heur_minHolder.java.

heur_min.java : Ce fichier contient la traduction en Java de la structure sous forme d'une classe, chaque membre de la structure devenant un attribut public de la classe. Des constructeurs sont ajoutés pour initialiser les attributs.

La fonction ci-dessus renvoie une structure contenant deux attributs.

Pour renvoyer deux valeurs nous aurions aussi pu écrire une fonction qui renverrait deux valeurs (équivalent du passage par référence en C) comme ci-après :

```
interface Heur_min
{  void get_heur_min(out long hh, out long mm);
};
```

En effet, un paramètre d'opération en IDL peut être spécifié de trois

façons :

```
long get_fonc (in  long param);
void get_fonc (out long param);
void get_fonc (inout long par);
```

Les classes Holder

Le choix de conception du langage Java fait que celui-ci ne peut passer les arguments à une fonction **que par valeur**.

Ainsi, pour les types simples (int, long, double ...) une fonction Java ne peut pas modifier un paramètre d'appel et renvoyer la nouvelle valeur à son appelant.

Pour pouvoir faire cette opération, il faut passer à la fonciton une **référence sur une instance** d'une classe et **modifier un attribut** de la classe.

Donc pour pouvoir implanter les sémantiques **out et inout** de CORBA, il faut, en Java, **encapsuler** les arguments dans des classes : c'est le rôle des classes **holder**.

Exemple : Ci-dessous on montre un exemple **val.java** et un exemple **ref.java** qui illustrent cette particularité du langage Java :

val.java :

```
public class val
{
  public static int value = 5;
  public void change( int i)
  {
        i = i * 2;       //<=***** accès par VALEUR
  }
  public static void main(String args)
  {
     val une_val = new val();
     System.out.println("value="+une_val.value);
```



```
     une_val.change(une_val.value);
     System.out.println("value="+une_val.value);
  }
}
1090 lo33 b116serv:~/mv/java> javac val.java
1091 lo33 b116serv:~/mv/java> java val
value=5
value=5          //<=***** accès par VALEUR
                 //pas de modif à l'extérieur
```

ref.java :

```
public class ref
{
  private class hold
```



```
  {
     public int value;
     public hold ()      // constructeur
     {}
     // constructeur avec initialisation
     public hold (int init)
     {  value = init; }
     public void change( hold i)
     {
        i.value = i.value * 2;
     }
  }

  public static void main(String args)
```



```
  {
     ref une_ref = new ref();
     // créer classe interne à ref
     hold une_val= une_ref.new hold(5);

     System.out.println("value="+une_val.value);
     une_val.value = 7;
     System.out.println("value="+une_val.value);
                        // appel avec une instance
                        // de classe en argument
                        // ==> passage par RÉFÉRENCE
     une_val.change(une_val);
     System.out.println("value="+une_val.value);
  }
```



```
}
1106 lo33 b116serv:~/mv/java> javac ref.java
1107 lo33 b116serv:~/mv/java> java ref
value=5
value=7
value=14        //<== valeur interne modifiée
```

Quand on passe an paramètre à une fonction java, non pas un type simple mais **un objet**, **on peut modifier la valeur des attributs de l'objet**.

Donc pour les **paramètres out et inout** de CORBA, il faut définir des **classes "conteneur"** pour chaque type utilisé dans un paramètre modifié. Ces classes sont les**classes "Holder"** automatiquement créées par le compilateur IDL.

Ces classes portent le nom du type accolé au mot "Holder".

Exemple : heure_min fonc () ; ==> heure_minHolder.java

Pour les types de base de CORBA les bibliothèques CORBA possèdent déja la collection des classes holder correspondantes :

```
org.omg.CORBA.BooleanHolder
org.omg.CORBA.ShortHolder   etc ...
```

Les classes holder possèdent un attribut public "value" qui correspond à la valeur qu'elles contiennent.

Traduction des interfaces IDL en Java

Les interfaces IDL deviennent des interfaces Java et chaque opération de l'interface devient une méthode publique de l'interface Java.

Fichiers générés

La compilation d'une interface horloge.idl par idl2java génère les fichiers suivants :

- **_horlogeStub.java** : la souche côté client,
- **_horlogeImplBase.java** : le squelette côté serveur,
- **horlogeOperations.java** : une interface contenant les signatures des fonctions définies dans horloge.idl
- **horloge.java** : l'interface représentant l'objet horloge ; elle hérite de **horlogeOperations.java** et de **org.omg.CORBA.portable.IDLEntity**,

ce qui en fait un objet CORBA accessible à distance,

- **horlogeHelper.java** : fournit l'opération narrow aux clients,
- **horlogeHolder.java** : permet d'utiliser un paramètre de type horloge en mode out ou inout.

Les attributs des interfaces IDL

L'IDL permet de définir des attributs :

```
attribute type_de_l_attribut nom_de_l_attribut ;
// exemple d'attribut de type long:
     attribute long age;
// un attribut peut être en lecture seule :
     readonly attribute long solde;
```

Le compilateur IDL va générer une ou deux fonctions pour représenter un attribut : une pour l'accès en lecture et une pour l'accès en

écriture.

```
attribute long total; //===>
        long total();           // lecture
        void total(long val);   // écriture
```

Utilisation de références d'objets dans les interfaces IDL

Il est possible d'utiliser une **référence d'objet** comme type de **paramètre d'une fonction** ou comme membre d'une structure de données.

Une référence d'objet est symbolisée par le nom de l'interface correspondant à cet objet.

Par exemple, si on a créé une interface IDL "**compte_bancaire**" dans un fichier compte_bancaire.idl, il est possible d'utiliser des ré-

férences à des objets CORBA de type "compte_bancaire" dans une autre interface, par exemple une interface "banque.idl".

```
interface banque
{  compte_bancaire creer_compte (
              in string titulaire, in string adresse);
};
```

Ici l'opération "creer_compte" renvoie une référence vers un **objet CORBA de type "compte_bancaire"**.

Les héritages et les interfaces IDL

L'IDL supporte l'héritage : une interface peut hériter de une ou plusieurs interfaces. Exemple :

```
interface Base
{
  void f();
};
interface AutreBase
{
  void g();
};
interface PremierHeritier : Base
{
  void h();
```



```
};
interface DeuxièmeHeritier : Base, AutreBase
{
  void z();
};
```

Remarque 1 : la surcharge de méthodes est interdite en IDL. Ainsi PremierHeritier ne peut pas déclarer une opération "f()", même si sa signature est différente de celle de "Base :f()".

Remarque 2 : un objet de type DeuxièmeHeritier pourra être converit au choix en objet de type : DeuxièmeHeritier, Base, AutreBase ou org.omg.CORBA.Object

Problème d'implémentation en Java avec l'approche par héritage

Dans l'approche par héritage la classe d'implantation XyyyImpl.java hérite de la classe _XyyyImplBase.java générée par le compilateur IDL.

Or le langage Java interdit l'héritage multiple. Ainsi la classe PremierHeritierImpl va hériter de _PremierHeritierImplBase.

La classe PremierHeritier implémente l'interface PremierHeritier qui elle-même hérite de l'interface base.

Donc elle doit implanter les fonctions définies dans PremierHeritier **et** dans Base.

Donc la fonction "Base :f()" doit être définie dans PremierHeritier. Malheureusement, on ne peut pas la définir implicitement en faisant hériter PremierHeritierImpl de BaseImpl, ce qui serait normal dans la logique de l'héritage puisque BaseImpl (qui implémente l'interface

Base) possède de ce fait une fonction f().

Il faut alors redéfinir "f()" dans l'implantation de PremierHeritierImpl.

```
public class BaseImpl extends _BaseImplBase
 {
   public void f()
   { ...//code de f...
   }
 }
public class PremierHeritierImpl extends _PremierHeritierImplBase
 {
   public void f()
   { ...//code de f...  à réimplanter car on ne peut
        //              pas hériter aussi de BaseImpl
   }
```



```
   public void h()
   { ...//code de h...
   }
 }
```

Cette **limitation est due à Java** et à conduit l'OMG à définir une approche particulière pour conserver les bénéfices de l'héritage et de la réutilisation du code : l'approche par délégation ("tie").

Cette approche, détaillée en fin de chapitre, va permettre de se dégager de l'obligation pour une classe d'implémentation d'hériter du squelette du serveur.

Ses fonctions (qui possèdent le code à exécuter) seront appelées par l'intermédiaire d'une **classe "Tie"** qui elle héritera de _PremierHeritierImplBase, proposera des fonctions de même signature que

celles de PremierHeritierImpl, fonctions "vides" qui ne feront que **transmettre l'appel** à la vraie fonction de la classe PremierHeritierImpl.

Connexion d'objets par des fonctions de la classe ORB

L'adaptateur d'objet "BOA" n'a pas été suffisamment spécifié lors de sa normalisation. Il en est résulté des différences d'utilisation entre des ORBs de fournisseurs différents, empêchant la portabilité complète du code.

À partir de CORBA 2.3, le BOA ne fait plus partie de la spécification. Il a été remplacé par le POA : "Portable Object Adapter".

Toutefois on a pu constater que dans la très grande majorité des cas seules 5 fonctions du BOA étaient utilisées :

- connexion d'un objet,
- activation d'un objet,
- déconnexion d'un objet,

- désactivation d'un objet,
- attente des requêtes clients.

On a donc ajouté 3 fonctions directement à la classe ORB pour simplifier l'écriture de serveurs simples :

- orb.connect(obj)
- orb.disconnect(obj)
- **orb.run()** pour se mettre en attente des clients

Les initialisations sont alors faites de façon transparentes et portables.

Toutefois, dans les application plus complexes, il est utile d'utiliser les fonctionnalités suplémentaires apportées par le POA.

Compléments sur l'IDL et l'environnement CORBA

Le langage IDL possède encore un certain nombre de carcatéristiques destinées à faciliter la réalisation d'applications complexes :

- règles de visibilité de l'IDL,
- les modules,
- les constantes,
- les énumérations,
- les exceptions,
- les opérations "oneway",
- les tableaux,
- les séquences,
- les alias (typedef),
- les unions.

L'environnement CORBA propose également :

- le type Any,
- les clauses de précompilation,
- les contextes d'opérations,
- les références initiales,
- le service d'annuaire interopérable,
- les URLs CORBA.


```
e4> jidl --package calc Calculatrice.idl
e4> ls calc
CalculatriceHelper.java       _CalculatriceStub.java
CalculatriceHolder.java       DivisionParZeroHelper.java
Calculatrice.java             DivisionParZeroHolder.java
CalculatriceOperations.java   DivisionParZero.java
CalculatricePOA.java
e4> cd calc
e4> nedit Serveur.java
e4> nedit Client.java
e4> nedit Calculatrice_impl.java
e4> setenv CLASSPATH .:/progs/JOB/lib/
e4> cd ..
e4> javac calc/*.java
e4> setenv CLASSPATH .:/progs/JOB/lib/:/progs/JOB/ob/lib
e4> java -Xbootclasspath/p:$CLASSPATH calc.Serveur
poa manager créé
Addition = 5.0 + 3.0
Division = 5.0 / 0.0

e4> java -Xbootclasspath/p:$CLASSPATH calc.Client
5 + 3 = 8.0
Une division par zero s'est produite...
La division demandee etait 5.0 / 0.0
```

Application CORBA renvoyant une exception


```
// ----------------------------------------------------
// Une application gérant les exceptions
// JExtrait du livre "Au coeur de Corba - Jérôme Daniel"
// ----------------------------------------------------

// DIVISION PAR ZERO

exception DivisionParZero
{
 float op1;
 float op2;
};

// CALCULATRICE

interface Calculatrice
{
 float add ( in float nb1, in float nb2 );

 float div ( in float nb1, in float nb2 )
       raises ( DivisionParZero );
};
```



```
// Client.java
// application Calculatrice, inspirée de
// "Au coeur de Corba - Jérôme Daniel" éd.Vuibert,ch.4
package calc;
public class Client
{  public static void main(String args[])
   {  // Same as the server
      java.util.Properties props= System.getProperties();
      props.put("org.omg.CORBA.ORBClass",
                "com.ooc.OBServer.ORB");
      props.put("org.omg.CORBA.ORBSingletonClass",
                "com.ooc.CORBA.ORBSingleton");
      int status = 0;
      org.omg.CORBA.ORB orb = null;
      try
      {   orb = org.omg.CORBA.ORB.init(args, props);
          status = run(orb);
      }
      catch(Exception ex)
      {   ex.printStackTrace(); status = 1; }
      if(orb!=null){ try
                     { orb.destroy(); }
                     catch(Exception ex)
                     { ex.printStackTrace();status=1;}}
      System.exit(status);
   }
```

Client

```
//----- run() -----
  static int run(org.omg.CORBA.ORB orb)
  { org.omg.CORBA.Object obj = null;
    try
    {String refFile = "Calc.ref";
     java.io.BufferedReader in= new java.io.BufferedReader(
                              new java.io.FileReader(refFile));
     String ref = in.readLine();
     obj = orb.string_to_object(ref);   }
    catch(java.io.IOException ex)
    {  ex.printStackTrace(); return 1; }

    Calculatrice calc = CalculatriceHelper.narrow(obj);
    try{  // utiliser l'objet Calculatrice
      System.out.println("5 + 3 = " +
                          calc.add((float)5,(float)3) );
      System.out.println("5 / 0 = " + calc.div(5,0) );
    }
    catch ( DivisionParZero ex )
    { System.out.println("Une division par zero
                           s'est produite...");
      System.out.println("La division demandee etait "
                          +ex.op1+" / "+ex.op2);
    }
    return 0; }\\fin run
}\\fin classe
```

Client (suite)

57

```
// Serveur.java  application Calculatrice, inspirée de
// "Au coeur de Corba - Jérôme Daniel" éd.Vuibert,ch.4
package calc;
public class Serveur
{
  public static void main(String args[])
  {
    java.util.Properties props = System.getProperties();
    props.put("org.omg.CORBA.ORBClass",
                "com.ooc.OBServer.ORB");
    props.put("org.omg.CORBA.ORBSingletonClass",
                "com.ooc.CORBA.ORBSingleton");
    int status = 0;
    org.omg.CORBA.ORB orb = null;
    try
    { orb = org.omg.CORBA.ORB.init(args, props);
      status = run(orb);
    }
    catch(Exception ex)
    {  ex.printStackTrace(); status = 1; }
    if(orb!=null){ try
                   {  orb.destroy(); }
                   catch(Exception ex)
                   { ex.printStackTrace(); status=1;} }
    System.exit(status);
  }
```

58

```
//----- run() -----
  static int run(org.omg.CORBA.ORB orb)
            throws org.omg.CORBA.UserException
  {
    org.omg.PortableServer.POA rootPOA =
            org.omg.PortableServer.POAHelper.narrow(
            orb.resolve_initial_references("RootPOA"));
    org.omg.PortableServer.POAManager manager =
                        rootPOA.the_POAManager();
    System.out.println("poa manager créé");
    // Creation de l'objet Horloge
    Calculatrice_impl calc_impl= new Calculatrice_impl();
    Calculatrice calc = calc_impl._this(orb);
    try
    { String ref = orb.object_to_string(calc);
      String refFile = "Calc.ref";
      java.io.PrintWriter out= new java.io.PrintWriter(
                  new java.io.FileOutputStream(refFile));
      out.println(ref); out.close();
    }
    catch(java.io.IOException ex)
    { ex.printStackTrace(); return 1; }
    manager.activate();
    orb.run();
    return 0;
   }
  }
```

59

```
// Calculatrice_impl.java
package calc;
import org.omg.CORBA.*;

/* Implantation de l'interface IDL Calculatrice  */
public class Calculatrice_impl extends CalculatricePOA
{
   // Operation IDL "add"
   //
   public float add(float nb1, float nb2)
   {
      System.out.println("Addition = "+nb1+" + "+nb2);
      return nb1 + nb2;
   }
   // Operation IDL "div"
   //
   public float div(float nb1, float nb2)
              throws DivisionParZero
   {
      System.out.println("Division = "+nb1+" / "+nb2);
      if ( nb2 == 0 ) throw new DivisionParZero(nb1,nb2);
      return nb1 / nb2;
   }
}
```

renvoi d'une exception

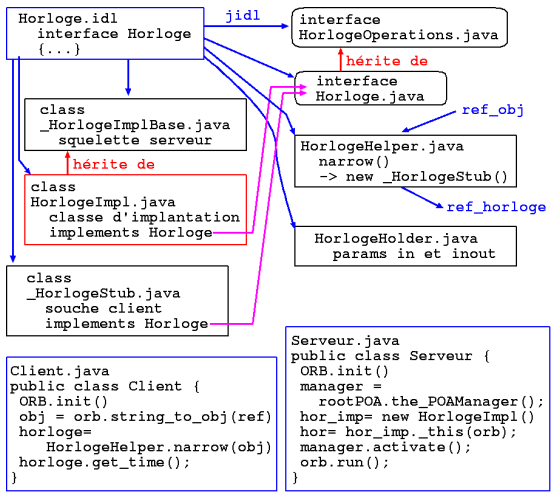
60

Ch19 - Corba IDL

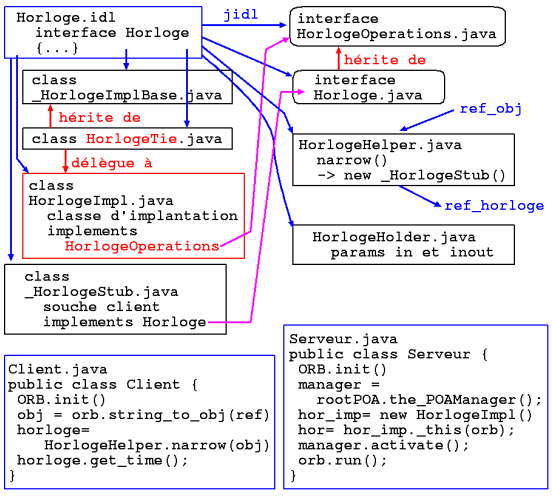
Résumé de la création d'une application CORBA avec approche par héritage

Résumé de la création d'une application CORBA avec approche par délégation ("tie") pour serveur en Java

61



62



63

Ch19 - Corba IDL

La délégation s'opère par association de l'instance hor = new HorlogeImpl() qui contient le code d'implantation (car implements HorlogeOperations) qui devient le "tie object" associé à une instance de la classe _HorlogeTie par h_tie = new _HorlogeTie(hor), classe tie (c'est elle la classe de délégation), qui est le squelette du serveur et va "déléguer" les appels des clients vers "l'objet tie" (ici "hor") qui contient le code effectif.

Ainsi le **code effectif (dans HorlogeImpl)** peut-il être **invoqué sans** que l'on ai besoin de le faire **hériter d'une classe générée par l'ORB**.

La **classe d'implantation** est alors "libérée" de son obligation d'hériter du squelette : elle **peut donc hériter d'une autre classe** d'implantation, ce qui permet de construire une **hiérarchie de classes**

64

d'implantation pour bénéficier de la réutilisation du code dans ces diverses classes.

Ceci est un problème spécifique à Java qui ne permet pas l'héritage multiple des classes (mais permet d'implémenter plusieurs interfaces).

Explications complémentaires

Dans le schéma par héritage, la classe HorlogeImpl hérite de la classe _HorlogeImplBase, qui elle-même implémente l'interface Horloge.

Or, on voudrait pouvoir éviter cet héritage, de façon à conserver la possibilité pour la classe HorlogeImpl d'hériter d'une autre classe d'implantation.

La classe _HorlogeImplBase est une classe abstraite possédant des méthodes publiques.

Le but est que ces méthodes soient disponibles sur les objets instanciés de classes héritant de _HorlogeImplBase, car ce sont les méthodes de base d'appel à distance par l'ORB (comme _invoke).

Donc, il faut enregistrer auprès du POA un objet qui hérite de ces méthodes de base.

Mais l'objet enregistré doit **aussi** posséder les fonctions de l'interface distante.

Or, la classe qui les possède est justement (par définition) la classe HorlogeImpl, celle dont on voudrait éviter de la faire hériter de _HorlogeImplBase pour se garder la possibilité de la faire hériter d'une autre classe.

L'astuce de la méthode par délégation consiste à créer une **classe intermédiaire ("tie")** qui va hériter de _HorlogeImplBase, qui va aussi **posséder toutes les fonctions** déclarées dans l'interface distante, et qui sera enregistrée auprès du POA.

Bien sûr, la classe "tie" ne possède pas l'implémentation des fonctions de l'interface distante (on ne va pas mettre deux fois l'implémentation !).

Simplement, chacune des fonctions "d'interface" de la classe "tie" va se contenter, quand elle est appelée, de **"renvoyer l'appel"** vers la vraie fonction correspondante de la classe d'implémentation HorlogeImpl.

On peut voir dans le listing ci-dessous d'une classe "tie" créée par le compilateur IDL, comment celle-ci enregistre une référence sur un objet instancié de la classe d'implémentation HorlogeImpl pour pouvoir renvoyer les appels sur les fonctions de cet objet.


```
public class HorlogeTie extends _HorlogeImplBase
{
 // Private reference to implementation object
 private HorlogeOperations _tie;
 // Constructor
 public HorlogeTie( HorlogeOperations tieObject )
 {
  _tie = tieObject; }

 // Get the delegate
 public HorlogeOperations _delegate()
 {
  return _tie; }

 // Set the delegate
 public void _delegate( HorlogeOperations delegate_)
 {
  _tie = delegate_; }

 // Operation get_time
 public java.lang.String get_time()
 {
  return _tie.get_time(); }
}
```

HorlogeTie.java

un appel à get_time() provoque en fait l'exécution de la fonction get_time() de la classe déléguée

Questions sur Ch19 - Corba IDL.

➤ Qu'est-ce que l'IDl de Corba ?
Réponse : diapo 2.

➤ Que doit faire un serveur Corba ?
Réponse : diapo 10.

➤ Que doit faire un client Corba ?
Réponse : diapo 14.

➤ À quoi sert la fonction **narrow** ?
Réponse : diapos 21 et 22.

➤ À quoi servent les classes **Holder** ?
Réponse : diapo 33 et 38.

➤ Que propose Corba pour gérer les exceptions ?

Réponse : diapo 55 et 57. Déclarer ses propres exceptions dans l'IDL et les récupérer dans le client.

Ch19 - Corba IDL

Fin du chapitre.

Ch20 - XML et XSL

- XML (diapo 2)
- XSL (diapo 22)
- Xpath (diapo 48)
- XSLT (diapo 66)
- XML Schema (diapo 137)

XML : eXtensible Markup Language

Introduction : XML est un **ensemble** de techniques : les documents XML eux-mêmes, les DTD (Document Type Definition) (ou maintenant les xml-schemas) et les "feuilles" XSL (eXtensible Stylesheet Language).

Définition : Un document XML est constitué d'une suite de mots contenant des **données textuelles** et des **balises de marquage**. Ces balises construisent une structure constituée d'une **arborescence de noeuds**.

Le balisage doit se conformer à certaines **contraintes de forme** afin que le **parseur XML** puisse reconstruire l'arborescence rapidement et de façon non ambigüe.

Dans ce cas le document XML est dit **bien formé**. Un document "mal formé" est inutilisable. Ce n'est pas un document XML. Face à un document mal formé, un parseur XML s'arrête avec un message d'erreur.

En plus des données de texte et des balises, on peut associer au document XML une **DTD** ou un **Schema** qui va jouer le rôle d'une grammaire indiquant des contraintes **sur le contenu**.

Exemple de contrainte imposée par une DTD : une balise <auteur> **devra** comporter : une balise <nom>, une balise <prénom> et une balise <e-mail>.

Un document XML conforme à sa DTD est dit valide.

Un parseur sachant aussi vérifier si un document XML est conforme

à sa DTD est dit **parseur validant**.

Première feuille XML : xml01.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE document SYSTEM "doc.dtd">
<document>
<header><title>Titre du document</title>
<subtitle>Un sous-titre pour le document</subtitle>
<authors>
  <author>
    <name>Michel Vayssade</name><shortname>MV</shortname>
    <email>Michel.Vayssade@utc.fr</email>
  </author>
</authors>
```



```
<organization>
  <name>Université de Technologie de Compiègne</name>
  <shortname>UTC</shortname>
</organization>
<date>28 Mars 2002</date>
</header>
..........
</document>
```

Première feuille XML : la DTD : xml01.dtd

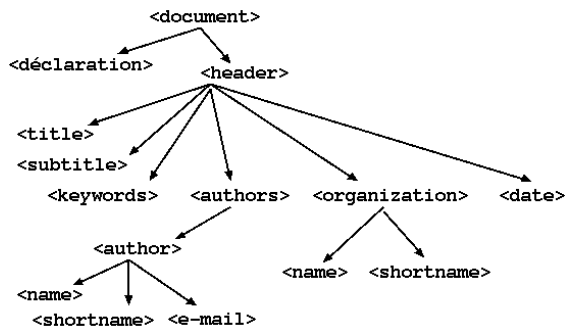
```
<?xml version="1.0" encoding="US-ASCII"?>
<!ELEMENT header (title,subtitle,keywords?,authors,organization,date)>
<!ELEMENT title (#PCDATA)*>
<!ELEMENT subtitle (#PCDATA)*>
```



```
<!ELEMENT keywords (keyword+)>
<!ELEMENT keyword (#PCDATA)>
<!ELEMENT authors (author+)>
<!ELEMENT author (name,shortname?,email?)>
<!ELEMENT name (#PCDATA)*>

Nota : * = 0 ou plus; ? = 0 ou 1; + = 1 ou plus;
```

Explications : la DTD stipule qu'une en-tête de document **doit** spécifier un titre et un sous-titre, qu'elle **peut** comporter une liste de mots clés (keywords ?), ... Puis qu'un auteur est décrit par un nom (obligatoire), un prénom (facultatif) et un e-mail (facultatif).

Ce document peut aussi être présenté sous la forme d'un arbre :



Nota : un document XML peut aussi contenir des **références** vers

d'autres fichiers . Lors du traitement tout se passe comme si le fichier référencé était inclus (copié) dans le document initial. Cet autre document peut, lui aussi, contenir des références vers d'autres documents. Dans le vocabulaire XML le fichier référencé est appelé **entité référence** ou **entité analysable**.

Nota : un document XML peut aussi contenir des **références vers des caractères** ; elles sont appelées **entités caractères**. Par exemple la chaine **<** est l'entité caractère chargée d'insérer dans le texte le caractère **<** qui ne peut pas y être inséré directement puisqu'il sert à délimiter les balises.

Remarque sur les parseurs XML : la spécification XML est très précise sur la façon dont un "parseur XML conforme" doit réagir aux

erreurs du fichier source XML analysé.

Les parseurs ne doivent laisser passer **aucune erreur**. La conséquence est qu'un parseur XML est un **petit** programme.

De plus, le passage du texte linéaire avec les balises à la représentation en arbre (ainsi que le passage inverse) sont faciles à réaliser. IL en résulte que les parseurs XML sont petits, rapides, fiab les et faciles à trouver et gratuits.

Conséquence de la structure XML : un document XML étant constitué d'une suite de caractères, son transfert vers un autre programme est simplifié : un document XML est toujours prêt à circuler sur un réseau.

Contrat : si un document XML spécifie une **DTD**, celle-ci repré-

sente un **contrat** entre le fournisseur des données XML et le consommateur de ces données.

Traitement : un document XML peut être vu comme une arborescence. Le traitement des arbres est quelque chose de bien compris et de maitrisé.

Avec XML a été développé un langage (**XPath**) qui permet de faire référence à des **ensembles de noeuds** (tous les descendants, tous les suivants, etc ...) ainsi qu'un langage (**XSLT**) permettant de décrire des transformations du document XML en un autre document (XML, HTML, texte, LaTeX, etc ...).

Utilisations typiques de XML :

- création de documents sources qui seront ensuite transformés vers des cibles diverses (HTML, LaTeX, PDF,...),
- encodage de données extraites ou à destination d'une base de données,
- configur ation de programmes informatiques (voir l'outil ant de Apache),
- création de DTD et de vocabulaires (nouvelles balises) spécifiques à un domaine.

Par exemple, J2EE utilise des conteneurs dans lesquels les composants d'une application sont déployés. La façon dont un composant interagit avec son conteneur est indiquée dans un fichier XML : c'est le **descripteur de déploiement**. De même, des fichiers XML

sont utilisés pour l'échange de données entre conteneurs.

XML (avec XSLT) permettant une **transformation aisée** de formats de données constitue une **passerelle** très pratique entre des formats issus d'applications héritées d'une part, et des applications nouvelles d'autre part.

Standardisation de XML : XML est issu de spécifications publiées par le **W3C** (WWW Consortium) : **http ://www.w3.org**

Structure d'un document XML

Exemple :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- commentaire (après la ligne ?xml, TOUJOURS première) -->
<!DOCTYPE document SYSTEM "doc.dtd"> <!-- réf. vers la DTD -->
<document>                            <!-- racine du document -->
<header>                              <!-- élément -->
  <title>Titre du document</title>    <!-- élément fils de header -->
  <subtitle>Un sous-titre pour le document</subtitle>
  <authors>                  <!-- autre élément fils de header -->
    <author>                 <!-- élément fils de authors -->
      <name>Michel Vayssade</name><shortname>MV</shortname>
      <email>Michel.Vayssade@utc.fr</email>
    </author>
```



```
  </authors>
  <organization>
    <name>Université de Technologie de Compiègne</name>
    <shortname>UTC</shortname>  <!-- élément fils de organization -->
  </organization>
  <date>28 Mars 2002</date>     <!-- élément fils de header -->
</header>
..........
</document>
```


Un document XML "bien formé" obéit aux règles suivantes :

- **toute balise** d'ouverture doit possèder une balise de **fermeture**,
- les balises ne peuvent **pas se chevaucher**,
- un document XML n'a qu'un seul élément racine,
- les noms des éléments doivent être conformes aux conventions XML
- les noms des éléments (balises) sont **sensibles à la casse**.

Éléments vides :parfois un élément ne contient pas de données (par exemple <cover titlepage="no" tableofcontents="no" />). On le termine par "/>" accolés (pas d'espace entre / et >).

Entités caractères et entités prédéclarées

Les documents XML se composent donc de **balises** définissant

des **éléments**, de données textuelles à analyser (**PCDATA** "Parsed Content DATA"), de données, valeurs d'attributs (**CDATA**) et de **sections CDATA** dont le contenu n'est pas analysé.

Deux caractères servant au balisage sont **interdits** partout : **<** et**&**.

Dans un contenu d'élément, trois autres caractères doivent être déguisés : **>**, **"** et **'**. Ces caractères doivent être mis dans le document sous une forme déguisée dite "entité caractère".

On utilise soit la valeur de leur représentation (par exemple pour < : #60 en décimal ou #x3C en hexa, soit un nom prédéfini (pour < c'est "lt"). Dans les deux cas, la référence esr délimitée par **&... ;**

On entre ainsi :

- pour **<** : **&lt ;**

- pour **&** : **&amp ;**
- pour **>** : **&gt ;**
- pour **"** : **&quot ;**
- pour **'** : **&apos ;**

Sections CDATA

Elles servent à **déguiser une portion de texte** contenant des caractères spéciaux tels que < ou &, par exemple un fichier XML servant d'exemple dans un exposé sur XML. Elles commencent par **< ![CDATA[** et finissent par**]]>**.

```
<![CDATA[   !<-- balise marquant le début de la section CDATA

        <?xml version="1.0" encoding="ISO-8859-1"?>
        <!-- un exemple XML -->
```



```
        <document>              <!-- racine du document -->
        ....
        </document>

]]>    !<-- balise marquant la fin de la section CDATA
```

Conséquence : la section CDATA ne doit pas contenir la séquence]]>. Si c'était le cas (comme par exemple dans le source XML qui a servi à générer ce document), cette séquence doit être masquée (en insérant un blanc :]] > ou découpée en deux séquences CDATA).

Les composantes de la "famille XML"

Le **W3C** a publié une série de **recommandations** décrivant un ensemble de "briques de base" pour l'écriture d'applications web interopérables et d'échanges de données structurées.

- **XML 1.0 :** décrit la syntaxe des documents XML et les règles que doivent respecter les parseurs. Un document XML est **bien formé** s'il est conforme à la syntaxe XML.
- **DTD :** permet de créer des **modèles** pour un types de documents. Une DTD établit les règles de définition de la structure d'un document. Un document XML est **valide** s'il est conforme à sa DTD.
- **schémas :** la définition des DTD ayant quelques faiblesses d'expressivité, le W3C a travaillé sur une méthode plus complète de

définition de la structure d'un document. Les "schémas" étant complexes à implémenter et à utiliser, il n'existe de parseur validant vis-à-vis des schémas que depuis peu.

- **espaces de noms :** ("namespaces") permettent d'utiliser plusieurs vocabulaires ayant des balises en commun dans un même document en préfixant la balise par le nom du namespace (par ex. <pers :title> et <xhtml :title>).
- **Xpath :** langage de requête permettant d'extraire des parties précises d'un document XML (en fait des portions bien délimitées de l'arbre associé au document).
- **XSL-XSLT :** XSLT est un langage de transformation qui va permettre d'extraire des portions d'un document source XML (en les

désignant par une expression Xpath) et les transformer puis les écrire dans un document destination (qui peut être XML ou texte).

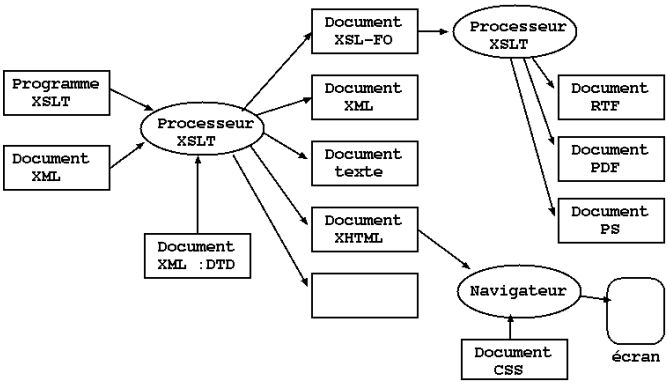
- **XSL-FO :** langage XML de description de pages imprimables de haute qualité.
- **Xlink et Xpointer :** création de liens entre documents XML.
- **DOM :** (Document Object Model) API de gestion de documents XML depuis un langage (C, Java,...).
- **SAX :** (Simple API for XML) API de gestion de documents XML depuis un langage (C, Java,...) complémentaire du DOM.

XSL : Introduction

XSL (eXtensible Stylesheet Language) est une **Recommandation** du **W3C** divisée en deux parties :

- **XSLT** (XSL Transform) langage XML servant à décrire les transformation d'un document XML ;
- **XSL-FO** (XSL Formating Objects) langage XML pour la description de pages imprimables de haute qualité typographique (Recommandation du 15 octobre 2001, qui n'est pas encore complètement implémentée).

Ces deux langages ont pour but d'être intégrés dans une chaine de production de documents comme sur la figure suivante :



XSLT est un langage interprété. Il lui faut donc un interpréteur souvent appelé "processeur XSLT". En "Open Source" on trouve les processeurs **Xt, Xalan, Saxon, XSLTproc, ...**

```
xsltproc is a command line tool for applying XSLT
stylesheets to XML documents. It is part of libxslt, the
XSLT C library for GNOME.
```

Donner une "petite" idée de XSLT est difficile car ce langage ne ressemble en rien aux langages procéduraux tels que C ou Java.

Seuls des langages fonctionnels comme Lisp, Prolog ou Caml peuvent donner une idée : on **déclare des règles** qui seront appliquées à des éléments du fichier source pour produire des éléments du fichier destination.

Un programme XSLT est une suite d'instructions d'extraction de données. Une instruction XSLT va chercher un élément correspondant à un **modèle ("template")** et instancier la valeur de ce modèle dans le document destination.

Examinons un petit exemple constitué d'un fichier de données source XML, d'une DTD et de 5 programmes XSLT.

- **uv.xml** : fichier source XML
- **uv.dtd** : fichier contenant la DTD de uv.xml
- **uv-1.xsl** : premier programme XSLT
- **uv-1.out** : sortie générée par uv-1.xsl appliqué à uv.xml
  La commande avec xsltproc est :$ xsltproc –output uv-1.out uv-1.xsl uv.xml
  uv-1.xsl extrait tous les éléments de uv.xml , dans l'ordre de leur

apparition dans uv.xml

- **uv-2.xsl** : programme XSLT qui extrait certains éléments seulement
- **uv-2.out** : sortie produite par uv-2.xsl
- **uv-3.xsl** : extrait les horaires de tous les groupes (pas seulement le premier)
- **uv-3.out** : sortie produite par uv-3.xsl
- **uv-4.xsl** : programme XSLT qui contrôle complètement la sortie (il n'y a pas de texte en dehors des instructions xsl),
- **uv-4.out** : sortie produite par uv-4.xsl
- **uv-5.xsl** : programme XSLT avec un meilleur contrôle de la sortie (pas d'exploration implicite d'un élément composite),
- **uv-5.out** : sortie produite par uv-5.xsl

uv.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE polytex SYSTEM "uv.dtd">
<uv>
  <enseignants>
      <un-enseignant>
         <nom>Jean-Marc Philippe</nom>
         <statut>UTC</statut>
      </un-enseignant>
      <un-enseignant>
         <nom>Pierre Arthur</nom>
         <statut>extérieur</statut>
      </un-enseignant>
  </enseignants>
```



```
  <cours>
    <horaire>
       <jour>mercredi</jour>
       <heure>14h15</heure>
    </horaire>
    <enseignant>
       <nom>Jean-Marc Philippe</nom>
    </enseignant>
  </cours>
  <td>
    <groupe-td>
      <horaire>
         <jour>lundi</jour>
         <heure>08h00</heure>
```



```
      </horaire>
      <enseignant>
         <nom>Jean-Marc Philippe</nom>
      </enseignant>
      <inscrits>
        <étudiant>Jean Dupond</étudiant>
        <étudiant>Jacques Durand</étudiant>
      </inscrits>
    </groupe-td>
    <groupe-td>
      <horaire>
         <jour>mardi</jour>
         <heure>10h15</heure>
      </horaire>
```



```
      <enseignant>
         <nom>Pierre Arthur</nom>
         <statut>extérieur</statut>
      </enseignant>
      <inscrits>
        <étudiant>Jean Lefevre</étudiant>
        <étudiant>Paul Smith</étudiant>
      </inscrits>
    </groupe-td>
  </td>
</uv>
```

uv.dtd

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```



```
<!--* DTD pour un document UV *-->
<!--Nota : * = 0 ou plus; ? = 0 ou 1; + = 1 ou plus;-->

<!ENTITY % uv "(enseignants,cours?,td*)">

<!ENTITY % enseignants "(un-enseignant+)">
<!ENTITY % cours "(horaire,enseignant+)">
<!ENTITY % horaire "(jour,heure)">
<!ENTITY % un-enseignant "(nom,statut)">

<!ENTITY % td "(groupe-td+)">
<!ENTITY % groupe-td "(horaire,enseignant,inscrits)">
<!ENTITY % enseignant "(nom)">
<!ENTITY % inscrits "(étudiant+)">
```



```
<!ENTITY % étudiant "(#PCDATA)">
```

uv-1.xsl

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output cdata-section-elements="verbatim"
            doctype-system="uv.dtd" method="xml" indent="no"
            encoding="ISO-8859-1" />

<xsl:template match="/">
  <xsl:apply-templates/>
</xsl:template>
```

```
</xsl:stylesheet>
```

uv-1.out

```
<?xml version="1.0" encoding="ISO-8859-1"?>


        Jean-Marc Philippe
        UTC


        Pierre Arthur
        extérieur
```



```
        mercredi
        14h15

etc ...
! on remarque que cette feuille XSLT produit tous les contenus
! du fichier source sur la sortie standard, y compris les
! formatages (indentations, retours ligne).
```

uv-2.xsl

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```



```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output encoding="ISO-8859-1" />

<xsl:template match="/">
écrire ce texte dans le fichier destination
puis extraire un élément du fichier source XML :
<xsl:value-of select="uv/cours/horaire/jour" />

extraire un élément composite :

<xsl:value-of select="uv/enseignants" />

puis un autre :
<xsl:value-of select="uv/td/groupe-td/horaire" />
```



```
</xsl:template>

</xsl:stylesheet>
```

uv-2.out

```
<?xml version="1.0" encoding="ISO-8859-1"?>

écrire ce texte dans le fichier destination
puis extraire un élément du fichier source XML :
mercredi

extraire un élément composite :
```



```
        Jean-Marc Philippe
        UTC


        Pierre Arthur
        extérieur



puis un autre :

        lundi
```



```
        08h00
```

uv-3.xsl

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Tr
<xsl:output encoding="ISO-8859-1" />

<xsl:template match="/">
écrire ce texte dans le fichier destination
puis extraire un élément du fichier source XML :
<xsl:value-of select="uv/cours/horaire/jour" />

extraire un élément composite :
```



```
<xsl:value-of select="uv/enseignants" />

puis un autre :
<xsl:for-each select="uv/td/groupe-td/horaire">
  <xsl:value-of select="." />
</xsl:for-each>

</xsl:template>

</xsl:stylesheet>
```

uv-3.out

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```



```
écrire ce texte dans le fichier destination
puis extraire un élément du fichier source XML :
mercredi

extraire un élément composite :


        Jean-Marc Philippe
        UTC


        Pierre Arthur
```

```
        extérieur


puis un autre :

        lundi
        08h00

        mardi
        10h15
```

uv-4.xsl


```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Tr
<xsl:output encoding="ISO-8859-1" />

<xsl:template match="/">
<xsl:text>écrire ce texte dans le fichier destination
puis extraire un élément du fichier source XML : </xsl:text>
<xsl:value-of select="uv/cours/horaire/jour" />
<xsl:text>
extraire un élément composite :</xsl:text>
<xsl:value-of select="uv/enseignants" />

<xsl:text>puis un autre :</xsl:text>
<xsl:for-each select="uv/td/groupe-td/horaire">
```



```
  <xsl:value-of select="." />
</xsl:for-each>

</xsl:template>

</xsl:stylesheet>
```

uv-4.out

```
<?xml version="1.0" encoding="ISO-8859-1"?>
écrire ce texte dans le fichier destination
puis extraire un élément du fichier source XML : mercredi
extraire un élément composite :

        Jean-Marc Philippe
```



```
        UTC


        Pierre Arthur
        extérieur

  puis un autre :
        lundi
        08h00

        mardi
        10h15
```

uv-5.xsl


```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output encoding="ISO-8859-1" />

<xsl:template match="/">
<xsl:text>écrire ce texte dans le fichier destination
puis extraire un élément du fichier source XML : </xsl:text>
<xsl:value-of select="uv/cours/horaire/jour" />
<xsl:text>
extraire un élément composite :</xsl:text>
<xsl:value-of select="uv/enseignants" />

<xsl:text>puis un autre :</xsl:text>
<xsl:for-each select="uv/td/groupe-td/horaire">
```



```
  <xsl:text>&#10;</xsl:text>
  <xsl:value-of select="./jour" />
  <xsl:text> : </xsl:text>
  <xsl:value-of select="./heure" />
</xsl:for-each>

</xsl:template>

</xsl:stylesheet>
```

uv-5.out

```
<?xml version="1.0" encoding="ISO-8859-1"?>
écrire ce texte dans le fichier destination
puis extraire un élément du fichier source XML : mercredi
```



```
extraire un élément composite :

        Jean-Marc Philippe
        UTC


        Pierre Arthur
        extérieur

  puis un autre :
lundi : 08h00
mardi : 10h15
```


XPath : Introduction

XPath est un langage permettant de désigner de façon précise des **ensembles d'éléments** de l'arbre XML source, en vue de leur appliquer un traitement spécifié en XSLT.

Par le moyen **d'expressions XPath**, on définit un **chemin de localisation**. Ce chemin peut être relatif au noeud courant ou absolu (s'il commence par "/", dans ce cas il part de la racine du document).

Dans les exemples précédents on a vu des chemins de localisation dans les **attributs** "select" des balises xsl : **select="uv/td/groupe-td/horaire"**.

Remarque 1 : la syntaxe d'une expression XPath n'est pas du XML. En effet, ces expressions sont appelées à être utilisées comme **valeurs** d'attributs de balises xsl.

Or, on ne peut **jamais** mettre **une balise XML à l'intérieur d'une autre balise** : une syntaxe telle que <balise attribut=<a>valeur n'est pas valide en XML.

Remarque 2 : il y a une analogie entre les "chemins" XPath et les chemins vers un fichier dans l'arborescence d'un système de fichiers unix.

Il faut toutefois se méfier de cette analogie : un chemin unix tel que "uv/td/groupe-td/horaire" pointe sur **un** fichier "horaire". Le chemin XPath ayant la même forme désigne **tous** les éléments <horaire> de **tous** des éléments <groupe-td> de **tous** les éléments <td>.

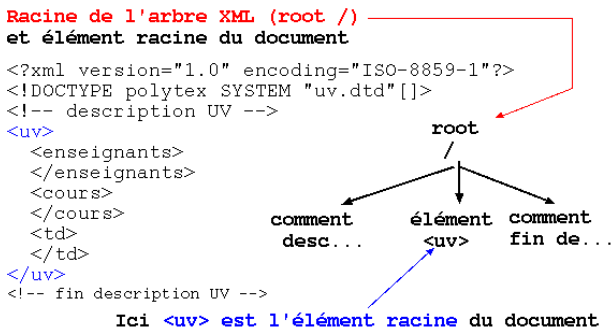
Il est souvent préférable de **lire** les chemins XPath **à l'envers**. Ainsi, "uv/td/groupe-td/horaire" se lira : "tous les éléments horaires rattachés à des éléments groupe-td rattachés à des éléments td rat-

tachés à des éléments uv".

XPath est fait pour explorer des documents XML. Il va donc utiliser l'arbre associé au fichier XML. Les chemins de localisation permettent de se déplacer dans cet arbre en sélectionnant des ensembles de noeuds de l'arbre.

Il y a dans l'arbre XML, **sept** types de noeuds :

- **root : /** la racine de l'arbre. C'est un noeud "virtuel". Il n'est pas visible dans le document. Il est créé virtuellement par XPath et XSLT car un arbre a besoin d'une racine ! Ne pas confondre avec **l'élément racine** du document qui est seulement la balise de "plus haut niveau".



- **élément : <balise>** un noeud élément XML

- **text : ..bla bla ..** un noeud texte, partie d'un élément
- **attribut : <xxx attrib="valeur">** un attribut d'un élément
- **namespace :** un nom permettant de qualifier (<nom :balise> des éléments, déclaré par exemple comme : xmlns :xi="http ://www.w3.org/2001/ qui crèe le namesapce "xi",
- **processing-instruction : < ? traitement arg1 arg2 ?>** appel à un traitement externe au processeur XSLT
- **comment : <– ... –>** commentaire.

Noeud contexte : à partir d'un noeud de l'arbre (appelé noeud contexte), un chemin de localisation ("location path") permet de désigner un ensemble de **voisins** du noeud courant choisis dans une **direction** que l'on appelle un **axe de localisation**.

Forme d'un chemin de localisation :

```
LocationPath = "/"?, LocationStep, ("/",LocationStep)*
```

Un chemin de localisation est donc une **suite d'étapes** de localisation séparées par des "/" avec éventuellement un **"/" au début** qui indique alors un chemin **absolu** (sinon c'est un chemin **relatif**).

L'évaluation d'un chemin de localisation donne un **node-set** c'est-à-dire un ensemble de noeuds, sans répétition et non ordonné.

Chaque étape du processus consiste en fait à **éliminer** les noeuds de l'arbre qui ne correspondent pas à cette étape du chemin.

Forme d'une étape de localisation :

```
LocationStep = Axis, "::", NodeTest, Predicate*
exemple : child::horaireattribute::type=été
```

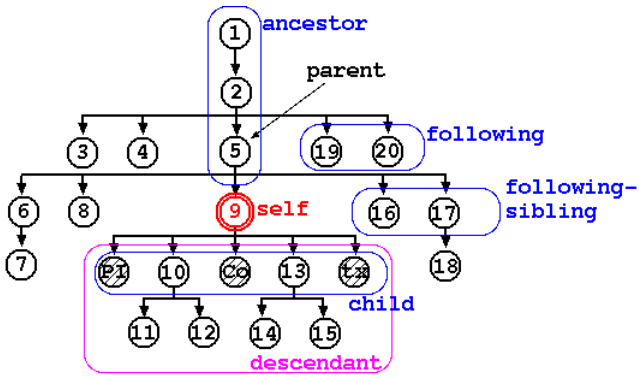

- **un Axe :** l'ensemble de départ est fait des noeuds choisis selon une direction : parents, enfants, frères en amon, frères en aval,...
- **un NodeTest (un déterminant) :** c'est un premier filtre permettant de ne garder que certains noeuds (par exemple tous les éléments <horaire>).
- **0 ou n prédicats :** filtre(s) suplémentaires (par exemple ne garder que les éléments ayant un attribut "truc=val").

Axes de localisation : XPath en définit treize.

- **child**
- **descendant**
- **parent**
- **ancestor**

- **self**
- **following-sibling**
- **preceding-sibling**
- **following**
- **preceding**
- **descendant-or-self**
- **ancestor-or-self**
- **attribute**
- **namespace**

Exemple (d'après XSLT fondamental, Ph.Drix, édi.Eyrolles 2002).



L'arbre ci-dessus correspond au fichier ci-dessous :

```
<1>
   <2>
      <3 />
      <4 />
      <5>
         <6>
           <7 />
         </6>
         <8 />
         <9>
            <? processing-instruction ?>
            <10>
               <11 />
```



```
               <12 />
            </10>
            <!-- commentaire -->
            <13>
               <14 />
               <15 />
            </13>
            du texte
         </9>
         <16 />
         <17>
            <18 />
         </17>
      </5>
```



```
      <19 />
      <20 />
   </2>
</1>
```

L'ordre de lecture des éléments est celui de l'arbre. Les attributs et les domaines nominaux viennent après leur parent et avant les enfants de leur parent.

Indice de proximité : cet indice associé à chaque noeud d'un ensemble va donner l'ordre d'énumération des éléments de l'ensemble.

Pour les axes directs, l'indice augmente quand on s'éloigne du noeud contexte en suivant l'ordre de lecture du document. Pour les axes rétrogrades, l'indice augmente quand on s'éloigne du noeud contexte vers le début du document.

Le NodeTest (ou déterminant) est une fonction, qui, appliquée à un noeud d'un ensemble désigné par un axe de localisation, va dire si le noeud doit ou non rester dans l'ensemble. Ce déterminant peut être :

- **un nom :** child : :horaire -> garder les éléments <horaire> fils du noeud courant.
- **une * :** child : :* -> garder tous les fils du noeud courant.
- **un type de noeud :** child : :text() -> garder tous les fils du noeud courant qui sont de type "text". Il y a cinq types possibles :
  - **text()**
  - **comment()**
  - **processing-instruction()**
  - **processing-instruction("xxx")** "pi" dont le nom est "xxx"

- **node()** garder tous les noeuds de l'axe

Prédicats

Un prédicat est une fonction booléenne qui appliquée à un ensemble de noeud (node-set) produit un nouvel ensemble qui ne contient que les noeuds pour qui le prédicat est vrai.

Le résultat étant lui aussi un "node-set", on peut appliquer les prédicats "en cascade" :

```
       Axe::NodeSetprédicat-1prédicat-2...
Exemple:
       child::*self::figureposition()=2
       sélectionne le deuxième élément "figure" parmi les élements
       fils du noeud courant
```



```
Exemple:
       child::paragraphe child::*self::figureposition()=2
       sélectionne les éléments "paragraphe" fils du noeud courant
       qui possèdent au moins deux éléments de type "figure" (car
       s'il n'y a pas de figure en position 2, le node-set résultat
       est vide).
```

Évaluation d'un chemin de localisation

Si on part d'un node-set contenant "n" noeuds, pour évaluer une étape de localisation vis-à-vis de ce node-set, on évalue "n" fois cette étape par rapport à chacun des noeuds.

Ces "n" évaluations produisent chacune un node-set. Le node-set global résultant est la réunion ensembliste (on enlève les doublons) de chacun des node-set produits par chacune des "n" étapes.


```
Exemple :
  uv/td/groupe-td/horaire
1-  à partir du noeud courant, on garde les éléments "uv"
2-  à partir de l'ensemble résultat, on garde les éléments "td"
3-  à partir de chaque noeud "td" de l'ensemble résultat
    précédent, on garde tous les éléments "groupe-td",
    on a donc maintenant un node-set contenant tous les
    éléments "groupe-td" de tous les "td" de toutes les "uv"
4-  pour chaque "groupe-td" du node-set produit en 3,
    on garde les éléments "horaire"
```

Formes courtes des chemins de localisation

Ces formes courtes étant employées dans la quasi totalité des exemples de codes sources XSLT disponibles, il est indispensable

de les connaitre.

```
Forme longue       Forme courte
------------       ------------
child::nom         nom
child::*           *
attribute::nom     @nom
attribute::*       @*
[position()=x]     [x]
self::node()       .
parent::node()     ..
/descendant-or-self::node()/   //
```

```
Exemple :
  bloc[3]/figure[@type='gif'][1]
  sélectionner les "figure" ayant un attribut type='gif' qui
```

```
sont aussi le premier ([1]) enfant du troisième ([3])
élément "bloc" du noeud contexte.
```


XSLT : eXtensible Markup Language Transform

Introduction : XSLT est un langage de **transformation de documents XSL**.

C'est un langage **déclaratif**, qui utilise des **motifs** ("pattern").

Un "programme" XSLT va rechercher la **concordance des motifs** ("pattern matching") avec les données d'un document XML.

Puis, quand une concordance est trouvée, il va appliquer au motif du document un **modèle de transformation** ("template").

Les deux modèles de transformation fondamentaux de XSLT étant **xsl :value-of** et **xsl :apply-templates**.

Structure d'un document XSLT : un programme XSLT est d'abord un fichier XML dont le domaine nominal est **"http ://www.w3.org/1999/XSL/Transform"**, abrévié généralement en **xsl :**, et dont la racine est l'élément **xsl :stylesheet**.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
..........
</xsl:stylesheet>
```

Un élément XSLT est un élément XML du domaine nominal "xsl :", c'est-à-dire une balise du type **<xsl :nom_balise...>**.

Une instruction XSLT est un élément XSLT. Les éléments fils direct de la racine **<xsl :stylesheet ..>** sont dits "instructions de premier niveau".

Régles de transformation :

XSLT est un langage déclaratif ("à la prolog") dans lequel on décrit des règles de transformation à appliquer quand le cas se présente lors de la lecture du fichier source XML à traiter.

En général, l'ordre dans lequel ces règles sont énoncées dans le fichier XSLT n'a pas d'influence sur le résultat.

Le processeur XSL parcours l'arbre du document XML d'entrée et applique à certains des éléments de cet arbre une règle de transformation choisie parmi l'ensemble des règles constituant le programme XSLT.

Une règle est composée de deux parties :

- **un motif :** ("pattern"), exprimé en **XPath**, il dit si on doit traiter

l'élément courant ou pas ;

- **un modèles de transformation :** ("template"), qui, si le motif correspond, indique **par quoi** remplacer l'élément courant

```
Forme d'une règle XSLT :
    <xsl:template match="..pattern..">
       <!-- modèle de transformation -->
       ....
    </xsl:template>
```

Exemple de programme XSLT :

l'exemple ci-dessous s'applique au fichier **uv.xml** donné plus haut.

```
<?xml version="1.0" encoding="ISO-8859-1"?><!-- uv-8.xsl -->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```



```
<xsl:output  method="text" encoding="ISO-8859-1" />

<xsl:template match="/">
    <xsl:text>&#10;Infos sur l'UV :</xsl:text>
    <xsl:apply-templates/>
</xsl:template>

<xsl:template match="enseignants">
    <xsl:text>Les enseignants de l'UV :</xsl:text>
    <xsl:apply-templates/>
</xsl:template>

<xsl:template match="un-enseignant">
    <xsl:text>Enseignant </xsl:text>
```



```
    <xsl:value-of select="./nom" />
    <xsl:text> (</xsl:text>
    <xsl:value-of select="./statut" />
    <xsl:text>).</xsl:text>
</xsl:template>

<xsl:template match="cours">
    <xsl:text>Le cours :</xsl:text>
    <xsl:apply-templates/>
</xsl:template>

<xsl:template match="td">
    <xsl:text>Les TDs :</xsl:text>
    <xsl:apply-templates/>
```



```
</xsl:template>

<xsl:template match="groupe-td">
    <xsl:text>TD : </xsl:text>
    <xsl:value-of select="./horaire/jour" />
    <xsl:value-of select="./horaire/heure" />
    <xsl:text>, enseignant : </xsl:text>
    <xsl:value-of select="./enseignant/nom" />
    <xsl:text>.</xsl:text>
</xsl:template>

</xsl:stylesheet>
```

L'exécution par la commande **$ xsltproc uv-8.xsl uv.xml** donne le résultat suivant :

```
Infos sur l'UV :
  Les enseignants de l'UV :
      Enseignant Jean-Marc Philippe (UTC).
      Enseignant Pierre Arthur (extérieur).

  Le cours :

       mercredi
       14h15


       Jean-Marc Philippe
```

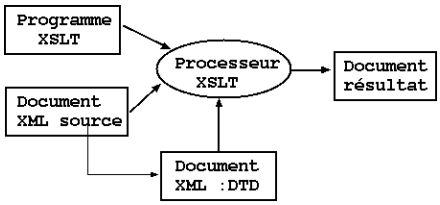


```
 Les TDs :
   TD : lundi08h00, enseignant : Jean-Marc Philippe.
   TD : mardi10h15, enseignant : Pierre Arthur.
```

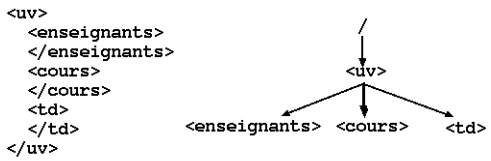
Dans cet exemple, les règles de transformation produisent du texte composé d'un **mélange** de **valeurs incluses** dans le fichier XSLT et de **valeur extraites** du fichier XML qui est le document source.

Principe de fonctionnement d'un processuer XSLT

Écrire un "programme" XSLT revient à prévoir le comportement du processeur XSLT face à la feuille de style XSLT que l'on écrit.



Le processeur XSLT commence par construire l'arbre XML à partir du fichier source XML qu'on lui donne à traiter. Rappelons que les deux représentations (arbre et fichier XML) sont équivalentes.



Le processeur XSLT opère des transformations sur l'arbre source pour l'arbre résultat.

La sérialisation de l'arbre résultat produit le document XML résultat.

Le processeur XSLT ne modifie jamais l'arbre source : il ne peut que construire un nouvel arbre.

La conséquence est que les seules opérations utiles consistent à

greffer un nouveau morceau à l'arbre résultat en cours de construction.

Du point de vue des documents XML, la greffe d'un morceau à un arbre revient à **concaténer un morceau de document** à un document en construction.

Or il est beaucoup plus facile de **décrire** des manipulations de documents. C'est pourquoi, les transformations XSLT sont souvent décrites en termes de construction de document à partir de fragments de documents.

Traitement du document source XML

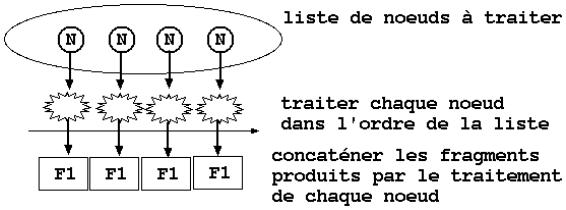
Au départ, on constitue une liste de noeuds contenant uniquement la racine du document source XML. On traite cette liste de noeuds

sources initiale.

Le traitement d'une liste de noeuds sources consiste à traiter chaque noeud de la liste dans l'ordre où il apparait dans la liste.

Le traitement de chaque noeud produit un fragment de document résultat.

Le document résultat est la concaténation, dans l'ordre, de ces fragments.



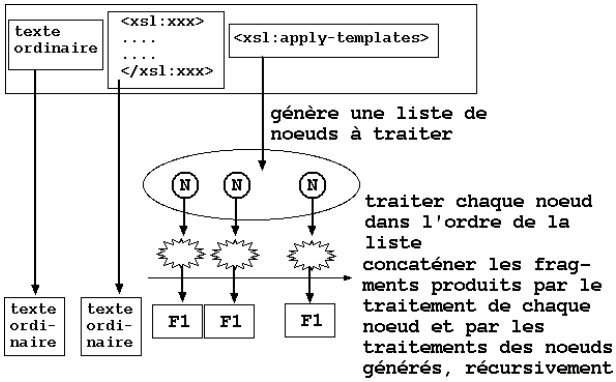
Traitement d'un noeud source

Traiter un noeud consiste à chercher dans la feuille de style XSLT, la règle dont le motif (pattern) correspond ("match") au noeud traité et à appliquer cette règle avec pour noeud courant le noeud en cours de traitement et pour liste courante la liste de noeuds source.

Si aucune règle explicite n'est trouvée, une règle par défaut s'applique.

Si plusieurs règles sont trouvées, un mécanisme de calcul de priorité permet d'un sélectionner une et une seule (sinon, le programme XSLT est incorrect).

Appliquer une règle consiste à instancier le modèle de transformation contenu dans la règle. Ceci produit un fragment de document qui est le résultat du traitement du noeud source en cours.



L'instanciation d'un modèle consiste à produire le fragment résultat.

Pour cela, tout de qui est du texte brut ou du texte XML, en dehors du domaine nominal XSLT (autrement dit, en dehors des balises <xsl :xxx>), dans le modèle est recopié tel quel.

Le reste est formé d'éléments XSLT (i.e., le texte compris dans les balises <xsl :xxx> ... </xsl :xxx>).

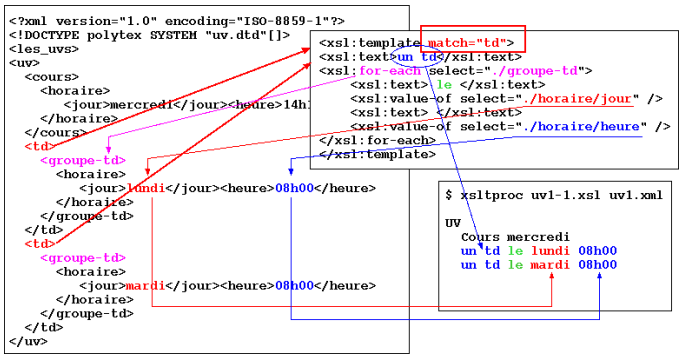
Ces éléments XSLT sont remplacés par leur valeur. Un élément XSLT de ce type s'appelle une **instruction XSLT**.

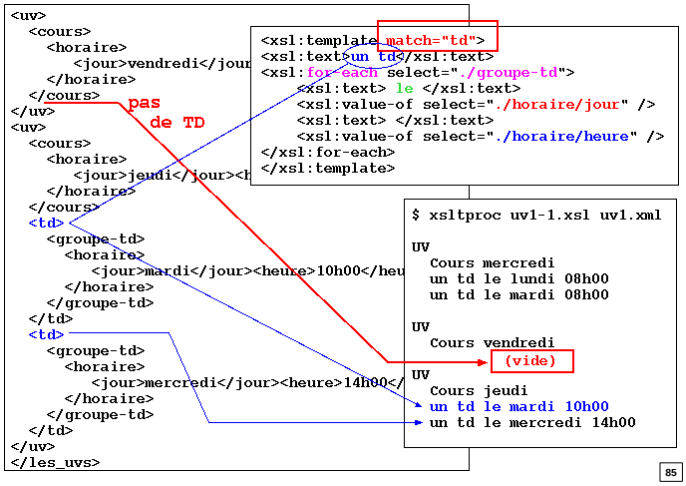
L'évaluation de l'une des deux instruction XSLT **<xsl :apply-templates>** et **<xsl :for-each>** produit une **nouvelle liste de noeuds sources** qui est **récursivement traitée**.

Motifs ("patterns")

C'est la valeur de l'attribut **match** dans la définition de la règle XSLT. Cet attribut doit représenter une **expression XPath**. Cette expression va désigner un ensemble de noeuds de l'arbre source.

Un motif utilisant un chemin de localisation relatif (c'est-à-dire qui ne commence pas par un /), ne peut être évalué que par rapport à un noeud contexte. Avec certains noeuds contexte, un motif peut donner un ensemble vide.





Dans l'exemple ci-dessus, le motif "td" s'applique successivement à chaque noeud courant. Quand ce noeud est une "uv" contenant des "td", il produit un ensemble contenant tous les noeuds "td" de cette "uv", et pour chaque noeud de l'ensemble, il applique la règle associée au motif.

Le but d'un motif est de désigner un noeud ou un groupe de noeud sur lesquels on désire appliquer une certaine règle.

Dans l'exemple, lorsqu'on applique le motif "td" à un noeud contexte "uv", il y a concordance du motif pour chaque noeud "td" fils direct de "uv".

La **concordance de motif n'est examinée** par XSLT **que sur l'axe "child"** (sinon, il faudrait examiner tout l'arbre à chaque fois).

Ceci n'est pas une limitation. En effet, le but d'un motif est de dési-

gner un noeud ou un groupe de noeud sur lesquels on désire appliquer une certaine règle.

L'intérêt d'un motif est son **pouvoir discriminant**.

Une règle qui s'appliquerait à trop de noeuds de l'arbre ne serait pas très utile.

Une conséquence de cette limitation à l'axe "child" est **l'interdiction dans un motif** de l'utilisation de **"./" ou "../"** puisque ceci permettrait d'étendre la recherche à tout l'arbre.

Remarque : si au cours du processus de traitement plusieurs règles sont éligibles, c'est la **plus spécifique** qui aura **priorité**.

Instruction XSLT : xsl :value-of

Exemple : une instruction "value-of" située dans un modèle de transformation, sera remplacée par la valeur textuelle de ce qui est désigné dans son attribut "select".

```
<xsl:template match=" motif ">
  <xsl:text>.. du texte ..</xsl:text>
  <xsl:value-of select="... chemin de localisation ..." />
</xsl:template>
```

Ainsi, dans l'exemple précédant, <xsl :value-of **select="./horaire/jour"** /> est remplacé, quand le motif s'applique au noeud courant, par la **valeur textuelle du noeud <jour>**.

Remarque : en fait, ce qui est désigné par un chemin de localisation est un "node-set".

Si ce node-set comporte plusieurs noeuds, sa valeur textuelle est celle du noeud source **qui arrive en premier** dans l'ordre de lecture du document.

Ainsi, si on modifie l'exemple "xml09" plus haut en mettant deux noeuds <jour> dans un groupe-td/horaire, **seul le premier** sera utilisé par <xsl :value-of select="./horaire/jour" />

```
      <horaire>
         <jour>dimanche</jour><heure>08h00</heure>
         <jour>lundi</jour><heure>08h00</heure>
      </horaire>
> xsltproc uv1-1.xsl uv2.xml
```



```
produisant :
  un td le dimanche 08h00
ou
  un td le lundi 08h00

suivant l'ordre des noeuds <jour>.
```


Instruction XSLT : xsl :apply-templates

Fonction : une instruction "apply-templates" est remplacée par le fragment de document qui résulte du **traitement de la liste des enfants** du noeud courant. Ces éléments enfants sont pris **dans l'ordre de lecture** du document source.

Par exemple :

```
<xsl:template match="/">
    <xsl:apply-templates/>
</xsl:template>
```

Cette règle va prendre un à un tous les enfants du noeud racine et leur appliquer le traitement, c'est-à-dire, chercher pour chacun si une des autres règles s'applique (concordance de motif). Dans l'exemple

"xml09" plus haut, elle applique les transformations à tous les enfants "<uv>" de la racine "<les_uvs>".

Si aucune règle du programme XSLT n'est applicable (aucune concordance de motif avec le noeud courant), alors une règle par défaut s'applique. Elle dépend de la nature du noeud à traiter.

Règle par défaut pour la racine ou un élément de l'arbre :

On reporte le traitement sur les enfants du noeud courant.

```
<xsl:template match="/|*">
    <xsl:apply-templates/>
</xsl:template>
```


Règle par défaut pour un noeud de type text ou attribute :

On ajoute un fragment de document en prélevant la **valeur du noeud courant**.

```
<xsl:template match="text()|attribute()::*">
    <xsl:value-of select="." />
</xsl:template>
```

L'application récursive de ces deux règles par défaut explique qu'un programme XSLT vide de règle, appliqué sur un fichier XML produise la suite des textes contenus comme valeur des différents noeuds.

Remarque : ceci peut avoir des conséquences en apparence bizarres quand le fichier source xml contient des erreurs.

Une erreur courante étant une faute de frappe dans une balise, conduisant à la présence d'un noeud qui ne correspond à aucun motif.

On peut alors essayer la méthode suivante pour trouver l'erreur : introduire dans le programme XSLT une **règle "attrape-tout"** qui imprime le nom de chaque élément sur lequel elle est appelée.

```
<xsl:template match="*">
  élément traité : balise <xsl:value-of select="local-name(.)" /> =
  <xsl:apply-templates/>
</xsl:template>
```

Appliqué au fichier uv.xml, ceci va produire :

```
> xsltproc uv-0a.xsl uv.xml
<?xml version="1.0" encoding="ISO-8859-1"?>
```


Instruction XSLT : xsl :sort

Fonction : une instruction "sort" est une instruction de tri qui ne s'emploie que comme **complément** à "xsl :apply-templates" ou "xsl :for-each" pour traiter les membres du node-set produit par ces deux instructions dans un ordre différent de celui du document source xml.

Elle ne peut apparaitre qu'en "deuxième niveau", dans le modèle de transformation d'un "apply-templates"ou d'un "for-each".

Instruction typique :

```
<xsl:template match="..motif..">
     <xsl:for-each select="...">
         <xsl:sort select=".." order=".." case-order=".."
```


L'attribut optionnel **select** permet de modifier la liste des noeuds à traiter.

Au lieu de prendre tous les enfants directs de ".", la valeur de select (un chemin de localisation) est calculée, ce qui donne un node-set dont les noeuds sont pris dans l'ordre de lecture du document source xml.

Il est prudent de se limiter dans le select à un sous-ensemble des descendants du noeud courant, même si ce n'est pas imposé, sinon on risque d'introduire une récursion infinie dans le traitement.

Instruction XSLT : xsl :for-each

Fonction : une instruction "for-each" va, (et c'est la seule avec l'instruction "apply-templates"), lors de l'instanciation du modèle qui la contient, **provoquer la création d'une nouvelle liste** de noeuds, traitée récursivement.

On a déja vu un exemple d'utilisation dans "uv-3.xsl", "uv-4.xsl", "xml08" et "xml09".

Instruction typique :

```
<xsl:template match="..motif..">
     <xsl:for-each select="...">
         ... texte ou instructions XSLT ...
     </xsl:for-each>
```



```
</xsl:template>
```

L'instruction "for-each" construit une nouvelle liste de noeuds d'après la valeur de son attribut "select" et lance un traitement sur cette liste.

La **différence avec** l'instruction "**apply-templates**" est que la **règle à appliquer** à chaque noeud de la liste n'est **pas cherchée dans l'ensemble** du programme XSLT, mais que l'on **applique** à chaque noeud de la liste le **même modèle** de transformation **défini entre les balises <xsl :for-each ..> et </xsl :for-each>**.

Instruction XSLT : xsl :sort

Fonction : une instruction "sort" est une instruction de tri qui ne s'emploie que comme **complément** à "xsl :apply-templates" ou "xsl :for-each" pour traiter les membres du node-set produit par ces deux instructions dans un ordre différent de celui du document source xml.

Elle ne peut apparaitre qu'en "deuxième niveau", dans le modèle de transformation d'un "apply-templates"ou d'un "for-each".

Instruction typique :

```
<xsl:template match="..motif..">
     <xsl:for-each select="...">
         <xsl:sort select=".." order=".." case-order=".."
```



```
                    lang=".." data-type=".." />
         ... texte ou instructions XSLT ...
     </xsl:for-each>
</xsl:template>
```

"xsl :sort" est toujours vide, et elle doit se trouver placer avant le modèle de transformation de la balise "apply-templates"ou "for-each" associée.

Attribut select : défini la clé de tri, chaîne de caractère à extraire de l'élément en cours. La valeur par défaut est la valeur textuelle du noeud courant.

Attribut order : ordre du tri (ascending ou descending), par défaut ascendant.

Attribut case-order : upper-first ou lower-first. La valeur par défaut dépend de la langue utilisée.

Attribut lang : un des codes de langues définis par XML. La valeur par défaut dépend de l'environnement de traitement ("locale" sous unix).

Attribut data-type : la nature de la clé : text ou number (défaut = text). Remarque : un type "date" aurait été très utile mais n'a pas été encore défini par la norme.

Tri à clés multiples :

```
<xsl:template match="..motif..">
     <xsl:for-each select="...">
         <xsl:sort select="nom" />
         <xsl:sort select="prénom" />
         ... texte ou instructions XSLT ...
     </xsl:for-each>
</xsl:template>
```


Instruction XSLT : xsl :copy-of

Fonction : une instruction "copy-of" est instanciée sous la forme d'une copie conforme des éléments sélectionnés par son attribut select=".. chemin de localisation ..".

Instruction typique :

```
<xsl:template match="..motif..">
     ... texte ou instructions XSLT ...
     <xsl:copy-of select="..." :>
     ... texte ou instructions XSLT ...
</xsl:template>
```

Si le résultat du select est une valeur terminale (texte, booléen, nombre), l'effet de "copy-of" est le même que celui de "value-of".

Mais, si le résultat est un node-set, alors le node-set est "sérialisé" : celci produit un fragment de document constitué de la concaténation des fragments produits en sérialisant chacun des noeuds du node-set, pris dans l'ordre de lecture du document source xml.

En terme d'arbre XML, "copy-of" provoque un duplicata récursif d'un sous-arbre.

Instructions de programmation XSLT

Introduction : les instructions de programmation XSLT permettent :

- conditionner l'instanciation d'un modèle à une expression booléenne : **xsl :if, xsl :choose**.
- manipuler des variables ou des paramètres : **xsl :variable, xsl :param**.
- manipuler des modèles nommés ("named templates") : **xsl :template name="nom" et xsl :call-template**.

C'est tout !

- pas de boucle
- pas d'évolution de la valeur d'une variable ; donc pas de compteur, pas d'incrémentation, pas d'effet de bord, pas de break, pas de return, ...

- pas de tableaux, pas de struct, pas de class, pas de listes, pas de hash-tables, ...

Et pourtant : XSLT est un langage "Turing-complet" !

En effet : les modèles nommés peuvent être appelés récursivement, on peut tester une condition, on peut créer des pseudo-tableaux en construisant un arbre que l'on convertit en chaine de caractère et en l'explorant avec des fonctions prédéfinies de XPath telles que "substring", "substring-after", "substring-before".

Or il a été montré que l'on pouvait tout programmer en ne disposant que des tableaux, de la récursion et des tests.

Bien entendu, c'est inutilisable dans la pratique.

Mais : la plupart des moteurs XSLT disponibles possèdent une extension permettant de convertir un arbre "RTF" (Result Tree Fragment) qui est normalement "write only" en un node-set sur lequel on peut appliquer n'importe quelle expression XPath et donc en extraire nominativement une information individuelle. La version suivante de la norme (XSLT 2.0) devrait intégrer cette extension en unifiant les notions de "RTF" et de "node-set".

Ceci dit, la programmation XSLT reste difficile car c'est une programmation fonctionnelle pure étrangère à tout ce qui est familier.

De plus le langage est très "spartiate" et la bibliothèque de fonctions peu développée. XSLT n'est pas conçu pour faire de l'algorithmique, mais pour faire des transformations d'arbres XML.

Il faut donc voir les quelques fonctions de programmation comme des compléments à la transformation d'arbres XML.

Exemple d'instruction :

```
<xsl:template match="..motif..">
    <xsl:value-of select="." />
    <xsl:if test="not(position() = last()">
        <xsl:text>,</xsl:text>
    </xsl-if>
</xsl:template>
==> ajoute une "," après chaque élément sauf le dernier.
```


Instructions xsl :if et xsl :choose

Fonction : ces instructions permettent une instanciation conditionnelle en fonction de la valeur d'une expression booléenne.

Instruction typique xsl :if :

```
<xsl:template match="..motif..">
    <xsl:if test=" .. expression ..">
        ... texte ou instructions XSLT ...
    </xsl-if>
</xsl:template>
```

L'expression XPath, argument de l'attribut "test=" est évaluée et transformée en booléen : il sera vrai si c'est un nombre non nul, une chaîne non nulle ou un node-set non vide.

Il n'y a **pas** de "else". Si on veut une alternative, il faut utiliser xsl :choose.

Instruction typique xsl :choose :

```
<xsl:template match="..motif..">
    <xsl:choose>
      <xsl:when test=" .. expression ..">
          ... texte ou instructions XSLT ...
      </xsl-when>
      <xsl:when test=" .. expression ..">
          ... texte ou instructions XSLT ...
          autant de when que l'on veut, mais au moins un
      </xsl-when>
      <xsl:otherwise>
```



```
          ... texte ou instructions XSLT ...
      </xsl:otherwise>
    <xsl:choose>
</xsl:template>
```

L'expression XPath, argument de l'attribut "test=" des différents "when" sont **évaluées en séquence**.

Dès que l'une est vraie, le modèle associé est instancié, et on "sort" du choose.

Si aucune n'est vraie, et qu'il existe une clause "otherwise", son modèle est instancié.

Si aucune expression de "when" n'est vraie et qu'il n'y a pas de "otherwise", aucun modèle n'est instancié.

Si plusieurs conditions sont vraies, seule la première évaluée à vrai verra son modèle instancié.

Instruction xsl :variable

Fonction : une instruction "xsl :variable" permet de **créer** une association nom/valeur. Une fois crée on **ne peut pas changer** la valeur d'une variable.

La seule instruction permettant d'affecter une valeur à une variable est la déclaration de la variable, déclaration qui ne peut être faite qu'une fois.

Instruction typique :

```
<xsl:template match="..motif..">
     ... texte ou instructions XSLT ...
     <xsl:variable name="mavar" select=".. expression XPath .." />
     <xsl:choose>
```



```
        <xsl:when test="$mavar &lt;=1" >
            <xsl:text>mavar est inférieure ou égale à 1</xsl:text>
        </xsl:when>
        <xsl:otherwise>
            <xsl:text>mavar est supérieure à 1</xsl:text>
        </xsl:otherwise>
    </xsl:choose>
    ... texte ou instructions XSLT ...
</xsl:template>
```

La variable "mavar" est définie au début d'un modèle de transformation et utilisée dans le modèle par la syntaxe "$mavar".

Exemple : le programme XSLT ci-dessous :

```
<?xml version="1.0" encoding="ISO-8859-1"?><!-- uv1-2.xsl -->
```



```
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output  method="text" indent="no" encoding="ISO-8859-1" />

<xsl:template match="/">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="uv">
    <xsl:text>UV</xsl:text>
    <xsl:variable name="nbretd" select="count(./td)" />
    <xsl:choose>
        <xsl:when test="$nbretd &lt;=1" >
            <xsl:text>- au plus un td -</xsl:text>
```



```
        </xsl:when>
        <xsl:otherwise>
            <xsl:text>- plusieurs tds -</xsl:text>
        </xsl:otherwise>
    </xsl:choose>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="cours">
    <xsl:text>Cours </xsl:text>
    <xsl:value-of select="./horaire/jour" />
</xsl:template>
                $
<xsl:template match="td">
```



```
<xsl:text>un td</xsl:text>
<xsl:for-each select="./groupe-td">
    <xsl:text> le </xsl:text>
    <xsl:value-of select="./horaire/jour" />
    <xsl:text> </xsl:text>
    <xsl:value-of select="./horaire/heure" />
</xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Appliqué au fichier uv1.xsl, produit le résultat ci-dessous :

```
> xsltproc uv1-2.xsl uv1.xml
```



```
UV- plusieurs tds -
  Cours mercredi
  un td le lundi 08h00
  un td le mardi 08h00

UV- au plus un td -
  Cours vendredi

UV- plusieurs tds -
  Cours jeudi
  un td le mardi 10h00
  un td le mercredi 14h00
```


Instructions xsl :param

Fonction : une instruction "xsl :param" ressemble beaucoup à "xsl :variable".

La différence est que la valeur donnée par l'attribut select lors de la déclaration est la **valeur par défaut** du paramètre.

Cette valeur par défaut pourra être remplacée soit lors de l'appel du processeur XSLT par une valeur donnée sur la ligne de commande, soit lors de l'appel d'un template nommé (call-template).

Instruction typique :

```
<xsl:template match="..motif..">
     ... texte ou instructions XSLT ...
     <xsl:param name="monparam" select="'default_val'" />
```



```
    <xsl:choose>
       <xsl:when test="$mavar != val" >
           <xsl:text>monparam non égal à val</xsl:text>
       </xsl:when>
       <xsl:otherwise>
           <xsl:text>monparam égal à val</xsl:text>
       </xsl:otherwise>
    </xsl:choose>
    ... texte ou instructions XSLT ...
</xsl:template>
```

Instructions xsl :call-template et xsl :template name="nom"

Fonction : une instruction <xsl :template name="nom"> est un modèle nommé destiné à factoriser une série de transformations qui sont utilisées à l'identique dans plusieurs règles du programme XSLT.

Forme de l'instruction :

```
Une règle :
                <xsl:template match="..motif..">
                     ... texte ou instructions XSLT ...
                </xsl:template>
Un modèle nommé :
                <xsl:template name="nom_du_modèle">
```



```
                     ... texte ou instructions XSLT ...
                </xsl:template>
```

Un modèle nommé peut comporter des arguments formels qui seront remplis par des instructions **<xsl :with-param name="par_i" select="..."> ** dans l'appel au modèle nommé par **<xsl :call-template ..>**

Instruction typique :

```
<xsl:template match="..motif..">
     ... texte ou instructions XSLT ...
     <xsl:call-template name="exemple">
         <xsl:with-param name="par_1" select="...">
         <xsl:with-param name="par_2" select="...">
     </xsl:call-template>
```



```
</xsl:template>

<xsl:template name="exemple">
     <xsl:param name="par_1" />
     <xsl:param name="par_2" />
     ... texte ou instructions XSLT ...
     ... utilise un paramètre par : $par_1
     ... exemple :
     <xsl:value-of select="$par_1" />
</xsl:template>
```


Instructions de création <xsl :text>

Fonction : une instruction <xsl :text> permet d'insérer du texte litéral dans le document de sortie. On en a déjà vu de nombreux exemples.

Un élément <xsl :text> ne doit pas contenir de balises.

Il recopie le texte de son modèle de transformation dans le document résultat.

L'utilisation systématique de balises <xsl :text>, à l'exclusion de texte inséré dans les modèles de transformation en dehors des balises permet de mieux contrôler le format de sortie, en particulier les caractères "blanc" et **"à la ligne" (&#10 ;)**.

Création de texte XML par un élément source littéral

Fonction : plutôt que de sortir un fichier texte, on peut vouloir produire un autre document XML.

On va alors insérer les balises XML à copier dans le document résultat dans les modèles XSL. On pourra donner des contenus à ces balises avec des éléments calculés extraits du fichier source par des instructions XSL.

Exemple uv-9.xsl

```
<?xml version="1.0" encoding="ISO-8859-1"?><!-- uv-9.xsl -->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output  method="text" encoding="ISO-8859-1" />
```



```
<xsl:template match="/">
    <xsl:apply-templates/>
</xsl:template>

<xsl:template match="enseignants">
</xsl:template>
<xsl:template match="un-enseignant">
</xsl:template>
<xsl:template match="cours">
</xsl:template>

<xsl:template match="td">
    <xsl:text>Les TDs :</xsl:text>
    <xsl:apply-templates/>
```



```
</xsl:template>

<xsl:template match="groupe-td">
    <xsl:text>TD : </xsl:text>
    <xsl:value-of select="./horaire/jour" />
    <xsl:value-of select="./horaire/heure" />
    <xsl:text>, enseignant : </xsl:text>
    <xsl:value-of select="./enseignant/nom" />
    <xsl:text>.</xsl:text>
</xsl:template>

</xsl:stylesheet>
```

L'exemple uv-9.xsl appliqué à uv.xml produit :


```
Les TDs :
  TD : lundi08h00, enseignant : Jean-Marc Philippe.
  TD : mardi10h15, enseignant : Pierre Arthur.
```

L'exemple uv-9x.xsl ci-dessous est modifié pour produire un fichier XML calculé à l'aide du document source :

```
<?xml version="1.0" encoding="ISO-8859-1"?><!-- uv-9x.xsl -->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Tr
<xsl:output  method="xml" encoding="ISO-8859-1" />

<xsl:template match="/">
    <xsl:apply-templates/>
</xsl:template>
```

```
<xsl:template match="enseignants">
</xsl:template>
<xsl:template match="un-enseignant">
</xsl:template>
<xsl:template match="cours">
</xsl:template>

<xsl:template match="td">
  <td>
  <titre>
    <xsl:text>Les TDs :</xsl:text>
  </titre>
    <xsl:apply-templates/>
```



```
  </td>
</xsl:template>

<xsl:template match="groupe-td">
  <untd>
    <jour>
       <xsl:value-of select="./horaire/jour" />
    </jour>
    <heure>
       <xsl:value-of select="./horaire/heure" />
    </heure>
    <xsl:text>&#10;          </xsl:text>
    <enseignant>
       <xsl:value-of select="./enseignant/nom" />
```



```
    </enseignant>
  </untd>
</xsl:template>

</xsl:stylesheet>
```

À noter : le changement de méthode de sortie par <xsl :output method="xml" encoding="ISO-8859-1" />.

uv-9.xsl appliqué à uv.xml produit :

```
  <td><titre>Les TDs :</titre>
    <untd><jour>lundi</jour><heure>08h00</heure>
          <enseignant>Jean-Marc Philippe</enseignant></untd>
```



```
    <untd><jour>mardi</jour><heure>10h15</heure>
          <enseignant>Pierre Arthur</enseignant></untd>
  </td>
```

Remarque : toutes les balises (<td>, <jour>, ... insérées directement dans le programme XSL aurait pu être remplaçées par des instructions <xsl :element name="td"> .. <xsl :element> , <xsl :element name="jour"> .. <xsl :element>, etc...

Bien que plus compliquée, cette méthode a un avantage, en effet, le contenu de l'attribut name=".." de l'instruction <xsl :element> peut être **calculé** en écrivant **<xsl :element name="{$var}">**.

Ici les {} qui entourent la variable $var ont pour rôle d'indiquer au processeur XSLT que le contenu de name="" n'est pas une valeur

litérale comme d'habitude, mais une valeur à calculer, c'est-à-dire qu'il doit remplacer $var par sa valeur et non pas insérer la chaîne de 4caractères $,v,a,r !

Ceci s'appelle une valeur différée d'attribut ("Attribute Value Template").

Complément : l'instruction <xsl :attribute> permet d'ajouter un attribut et sa valeur **dans** une balise XML que l'on construit dans le document résultat.

```
Ainsi :
    <enseignant>
      <xsl:attribute name="Nom">
         <xsl:value-of select="./enseignant/nom" />
      </xsl:attribute>
```



```
    </enseignant>
Produira :
    <enseignant Nom="Jean-Marc Philippe" />
```


Autres instructions XSLT

XSLT comporte un certain nombre d'instructions qui n'ont pas été montrées car elles dépassent le cadre d'une introduction.

- **xsl :message**
- **xsl :key**
- **xsl :attribute-set**
- **xsl :copy**
- **xsl :comment**
- **xsl :processing-instruction**
- **xsl :number**
- **xsl :include**
- **xsl :import**
- **xsl :apply-imports**

(diapo vide)

XML - Schema – Motivations

W3C Recommendation 28 October 2004

Le XML Schema est un langage XML dont le but est de permettre de décrire les contraintes et le contenu de documents XML.

Les Schemas ont pour objectif de remplacer les DTD qui ne permettent pas d'exprimer certaines contraintes.

Un XML Schema défini :

- les **éléments** qui peuvent apparaitre dans un document,
- les **attributs** permis,
- quels éléments sont des **fils d'éléments**,
- **l'ordre et le nombre** des éléments fils ,
- si un élément est vide ou s'il peut contenir du texte,
- les **types de données** des éléments ou attributs,

- les **valeurs par défaut** ou les valeurs fix es des éléments ou attributs.

Les Schemas vont, petit à petit, remplacer les DTD. En effet, les Schemas :

- sont **extensibles** (on peut utiliser un Schema dans un autre Schema),
- sont **plus riches** que les DTD,
- sont **écrits en XML** (donc **validables**, éditables, manipulables par XSLT),
- supportent les **types de données**,
- supportent les **espaces de noms** (namespaces).

Pour un document XML, être "bien formé", ce n'est pas suffisant.

Même un document bien formé peut contenir des erreurs. Par exemple la balise <quantité>120</quantité> peut être conforme à la DTD, mais l'application pourrait n'accepter que les nombres compris entre 0 et 100.

Un Schema pourra exprimer cette **contrainte** :

```
<xsd:simpleType name="quantité">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="0"/>
    <xsd:maxInclusive value="100"/>
  </xsd:restriction>
</xsd:simpleType>
```


De même une date telle que 05-12-2005 pourra être interprétée soit comme 05 décembre 2005, soit comme 12 mai 2005. Le type date des Schema impose le format YYYY-MM-DD évitant toute erreur d'interprétation.

Sur le site du W3C :
XML Schema Part 0 : Primer Second Edition
W3C Recommendation 28 October 2004
http ://www.w3.org/TR/xmlschema-0/

.
.
.
.

XML - Schema – Exemple

Un **premier exemple** :

Le document XML ci-dessous fait référence à une DTD :

```
<?xml version="1.0"?>
<!DOCTYPE note SYSTEM "note.dtd">
<note>
  <to>Jean</to>
  <from>Pierre</from>
  <head>Réunion projet</head>
  <body>En salle B123, reportée à 10h00</body>
</note>
```

Cette DTD indique le contenu légal de chacun des éléments :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--* DTD pour un document note *-->
```



```
<!ELEMENT note (to, from, head, body)>
<!ELEMENT to   (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT head (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

Le même document faisant référence à un XML Schema :

```
<?xml version="1.0"?>
<note   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="note.xsd">
..............
</note>
```

Note : il serait possible de ne mettre aucune référence dans le co-

cument instance, en laissant le processeur validant trouver lui-même le Schema :

```
<?xml version="1.0"?>
<note>
..............
</note>
```

Le Schema est un document XML qui spécifie la forme du document instance, avec la façon de construire les éléments composites et le contenu légal des éléments :

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="note">
       <xs:complexType>
```



```
         <xs:sequence>
           <xs:element name="to"   type="xs:string"/>
           <xs:element name="from" type="xs:string"/>
           <xs:element name="head" type="xs:string"/>
           <xs:element name="body" type="xs:string"/>
         </xs:sequence>
       </xs:complexType>
   </xs:element>
</xs:schema>
```

Un autre **exemple**, plus complet, extrait de la documentation publiée par le W3C : **http ://www.w3.org/TR/xmlschema-0/**.

Un document XML décrit un bon de commande (purchase order) :

.

```
<?xml version="1.0"?>
```

```
<purchaseOrder orderDate="1999-10-20">
   <shipTo country="US">
      <name>Alice Smith</name>
      <street>123 Maple Street</street>
      <city>Mill Valley</city>
      <state>CA</state>
      <zip>90952</zip>
   </shipTo>
   <billTo country="US">
      <name>Robert Smith</name>
      <street>8 Oak Avenue</street>
      <city>Old Town</city>
      <state>PA</state>
      <zip>95819</zip>
```



```
   </billTo>
   <comment>Hurry, my lawn is going wild<!/comment>
   <items>
      <item partNum="872-AA">
         <productName>Lawnmower</productName>
         <quantity>1</quantity>
         <USPrice>148.95</USPrice>
         <comment>Confirm this is electric</comment>
      </item>
      <item partNum="926-AA">
         <productName>Baby Monitor</productName>
         <quantity>1</quantity>
         <USPrice>39.98</USPrice>
         <shipDate>1999-05-21</shipDate>
```



```
      </item>
   </items>
</purchaseOrder>
```

Ce document consiste en un **élément principal** (purchaseOrder), et quatre sortes d'**éléments fils** (shipTo, billTo, comment, items).

Certains des **sous-éléments contiennent à leur tour des éléments**. Ceci sur **plusieurs niveaux**, jusqu'à arriver à des éléments contenant des **données simples**.

Un élément contenant soit d'autres éléments, soit des attributs est dit de type "**complexType**". Les autres sont des éléments "**simpleType**" qui ne peuvent contenir ni éléments, ni attributs.

Le Schema XML décrivant l'instance de document XML "purchaseOrder" précédente :


```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
     Purchase order schema for Example.com.
     Copyright 2000 Example.com. All rights reserved.
    </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
  <xsd:element name="comment" type="xsd:string"/>
  <xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
      <xsd:element name="shipTo" type="USAddress"/>
      <xsd:element name="billTo" type="USAddress"/>
```



```
      <xsd:element ref="comment" minOccurs="0"/>
      <xsd:element name="items"  type="Items"/>
    </xsd:sequence>
    <xsd:attribute name="orderDate" type="xsd:date"/>
  </xsd:complexType>

  <xsd:complexType name="USAddress">
    <xsd:sequence>
      <xsd:element name="name"   type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city"   type="xsd:string"/>
      <xsd:element name="state"  type="xsd:string"/>
      <xsd:element name="zip"    type="xsd:decimal"/>
    </xsd:sequence>
```



```
    <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
  </xsd:complexType>
    .
  <xsd:complexType name="Items">
    .<xsd:sequence>
      <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="productName" type="xsd:string"/>
            <xsd:element name="quantity">
              <xsd:simpleType>
                <xsd:restriction base="xsd:positiveInteger">
                  <xsd:maxExclusive value="100"/>
                </xsd:restriction>
```



```
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="USPrice"  type="xsd:decimal"/>
            <xsd:element ref="comment"   minOccurs="0"/>
            <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
          </xsd:sequence>
          <xsd:attribute name="partNum" type="SKU" use="required"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>

  <!-- Stock Keeping Unit, a code for identifying products -->
  <xsd:simpleType name="SKU">
```



```
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{3}-A-Z{2}"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Le préfixe "**xsd**" est utilisé, par **convention**, pour désigner **l'espace de noms XML Schema**, bien qu'on puisse en choisir un autre.

Dans le Schema on trouve des "element", des "complexType", et des "simpleType".

```
  <xsd:element name="comment" type="xsd:string"/>

  <xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
```

```
      <xsd:element name="shipTo" type="USAddress"/>
      <xsd:element name="billTo" type="USAddress"/>
      <xsd:element ref="comment" minOccurs="0"/>
      <xsd:element name="items"  type="Items"/>
    </xsd:sequence>
    <xsd:attribute name="orderDate" type="xsd:date"/>
  </xsd:complexType>

  <xsd:simpleType name="SKU">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{3}-A-Z{2}"/>
    </xsd:restriction>
  </xsd:simpleType>
```

Dans un XML Schema, il faut distinguer entre les **définitions** qui

crèent de nouveaux types, et les **déclarations** qui autorisent des éléments et attributs ayant des noms spécifiques à apparaitre dans une instance de document.

Ainsi les lignes :

```
<xsd:complexType name="USAddress">
  <xsd:sequence>
  ..................
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
</xsd:complexType>
```

indiquent que si un élément "USAddress" apparait dans un document, alors, cet élément doit être conforme à cette **définition** : même nombre, dans le même ordre, et que, de plus, cet élément **doit** conte-

nir un attribut country="US" (car déclaré"fix ed").

Remarque : tous les **attributs** ne peuvent être que de **type simple**. Ils ne peuvent pas contenir d'éléments ni d'autres attributs. En effet, les attributs sont à l'intérieur d'une balise, or en XML, une balise ne contient jamais d'autres balises (<balise <autre_balise>...</a..» est interdit en XML).

La plupart des éléments associent un **nom** à un **type** :
<xsd :element **name="comment" type="xsd :string"**/>
<xsd :element **name="shipTo" type="USAddress"**/>

Il est possible de mettre des déclarations d'éléments qui **reprennent** une définition existante en lui ajoutant une contrainte :
<xsd :element **ref=**"comment" minOccurs="0"/>

Contraintes d'occurence sur les éléments

Exemple : <xsd :element ref="comment" **minOccurs="0"**/> indique que cet élément est optionnel.

Pour imposer sa présence on mettra minOccurs >= 1.

On peut aussi limiter le nombre d'éléments :
<xsd :element ref="comment" **maxOccurs="5"**/>
ou <xsd :element ref="comment" **maxOccurs="unbounded"**/> .

Un **attribut** est présent une seule fois ou pas du tout. De plus il peut être "**required**", "**optional**", ou "**prohibited**". Exemple :
<xsd :attribute name="partNum" type="SKU" **use="required"**/>

... à suivre dans : **http ://www.w3.org/TR/xmlschema-0/**

Questions sur Ch20 - XML, XSLT

➤ Qu'est-ce que XML ?
Réponse : diapo 2.

➤ Quelle différence entre un document XML **bien formé** et un document XML **valide** ?
Réponse : diapo 3.

➤ En XML qu'est-ce une "entité caractère" ?
Réponse : diapos 8, 16 et 17.

➤ Quelles sont les règles définissant un document XML bien formé ?
Réponse : diapo 15.

➤ Comment commence et fini une portion de texte contenant des caractères spéciaux tels que "<" ou "&" ?

Réponse : diapo 17.

➤ Faites un schéma indiquant la place du processeur XSLT dans une chaine de transformation de documents.
Réponse : diapo 23 ou 75.

➤ Décrire en 2 phrases en quoi consiste un "programme" XSLT.
Réponse : diapos 24, 25 et 66.

➤ Écrire un exemple de fichier XML et sa DTD associée qui décrit que l'enseignant "Jean Machin" est responsable de l'UV "truc" qui comporte un cours de 2 heures et un TD de 2 heures.
Réponse : <uv><code>truc ..<resp><nom>Machin <prenom>Jean.. <enseignement> <cours>2h .. <td>2h ..
et <!ELEMENT enseignement (cours ?,td*,tp*)> <!ELEMENT cours (#PCDATA)*> etc ..

➤ Que désigne le chemin de localisation "uv/td/groupe-td/horaire" ?
Réponse : diapo 49.

➤ Dessinez un arbre XML dans lequel vous choisirez un noeud "self" et indiquerez l'ensemble des noeuds "descendant-or-self" et l'ensemble "following-sibling"
Réponse : diapo 56.

➤ Que fait l'instruction xsl :apply-templates que l'on retrouve un peu partout dans un programme XSLT ?
Réponse : diapo 91.

➤ Avec quelles autres instructions XSLT utilise-t-on l'instruction xsl :param ?
Réponse : xsl :call-template et xsl :template name="nom"

➤ Quelles sont les deux instructions XSLT permettant de program-

mer des choix en fonction du contenu du fichier XML analysé ?
Réponse : xsl :if et xsl :choose

➤ Quelle différence entre xsl :copy-of et xsl :value-of ?
Réponse : diapo 103.

➤ Quel est l'avantage principal des Schemas sur les DTDs ?
Réponse : diapo 138.

Ch20 - XML, XSLT

Fin du chapitre.

Ch21 - XMLRPC et Web Services

- XMLRPC
- Web Services

XML-RPC : appel RPC utilisant XML

Introduction : What is XML-RPC ?

It's a spec and a set of implementations that allow software running on disparate operating systems, running in different environments to make procedure calls over the Internet.

It's remote procedure calling using **HTTP as the transport** and **XML as the encoding**. XML-RPC is designed to be as simple as

possible, while allowing complex data structures to be transmitted, processed and returned.

An XML-RPC message is an **HTTP-POST request**. The body of the request is in XML. A procedure executes on the server and the value it returns is also formatted in XML.

Procedure parameters can be scalars, numbers, strings, dates, etc. ; and can also be complex record and list structures.

Example of an XML-RPC request :

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
```



```
Content-length: 181


<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param> <value><i4>41</i4></value> </param>
  </params>
</methodCall>

La réponse sera :
---------------
```



```
HTTP/1.1 200 OK
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:08 GMT
Server: UserLand Frontier/5.1.2-WinNT

<?xml version="1.0"?>
<methodResponse>
  <params>
    <param> <value><string>South Dakota</string></value> </param>
  </params>
</methodResponse>
```


Scalar <value>s

<i4> or <int>	four-byte signed integer	-12
<boolean>	0 (false) or 1 (true)	1
<string>	ASCII string	hello world
<double>	double-precision signed	-12.214
<dateTime.iso8601>	date/time	19980717T14 :08 :55
<base64>	base64-encoded binary	eW91IGNhbid0IHJlYW=

<struct>s

```
A <struct> contains <member>s and each <member> contains a <name> and a <value>.
<struct>
  <member>
    <name>lowerBound</name>
    <value><i4>18</i4></value>
```



```
  </member>
  <member>
    <name>upperBound</name>
    <value><i4>139</i4></value>
  </member>
</struct>

<struct>s can be recursive, any <value> may contain a <struct> or any
including an <array>
```

<array>s

```
A value can also be of type <array>.
An <array> contains a single <data> element,
which can contain any number of <value>s.
```



```
<array>
<data>
<value><i4>12</i4></value>
<value><string>Egypt</string></value>
<value><boolean>0</boolean></value>
<value><i4>-31</i4></value>
</data>
</array>

You can mix types as the example above illustrates.

<arrays>s can be recursive, any value may contain an <array>
or any other type, including a <struct>
```

La spécification peut être trouvée à : **http ://www.xmlrpc.com/spec**

Les implémentations sont nombreuses :
http ://www.xmlrpc.com/directory/1568/implementations

Entre autres : **AOLserver, Apache, C/C++, Cold Fusion, Delphi/Kylix, Eiffel, Flash, Java, JavaScript, KDE, Macintosh OS X, Mozilla, Perl, PHP, Python, Rebol, Scheme, Tcl, WebObjects, Zope.**

XML-RPC : Exemples d'exécution

Avec apache-xmlrpc

Installation à partir d'une version binaire: xmlrpc-1.2-b1.tar.gz

définir le jar xml-rpc dans le classpath :
```
    > setenv CLASSPATH .:/local1/xmlrpc-1.2-b1/xmlrpc-1.2-b1.jar
```

dans le source du serveur ajouter (après server.addHandler()
la ligne : server.start();

compiler le source utilisant xml-rpc :
```
    > javac JavaServer.java
    > javac JavaClient.java
```
exécuter :

dans une fénêtre :
```
    > java JavaServer 20000
```
dans une autre fenêtre :
```
    > java JavaClient
    Sum: 8, Difference: 2
```

Avec PHP-XML-RPC

Décompresser dans /local1 xmlrpc-1.0.99.2.tar.gz
décompresser : il y a des exemples et deux bibliothèques :
35272 Jan 13 2003 xmlrpc.inc 12539 Jan 13 2003 xmlrpcs.inc

Copier les deux bibliothèques ".inc" dans le "include path"
de PHP : /usr/share/pear/
On trouve le include path en lisant le message d'erreur dans

le log de PHP :
```
# tail /var/log/httpd/error_log
client 172.26.247.2 PHP Warning:  main(): Failed opening
'xmlrpcs.inc' for inclusion (include_path='.:/usr/share/pear')
```

Nota: suite à une évolution de PHP4 par rapport à ces deux bibliothèque, il faut aller mettre en commentaire les deux fonctions xmlrpc_decode() et xmlrpc_encode() de xmlrpc.inc

dans les clients PHP :
```
     include 'xmlrpc.inc';
```
dans les serveurs PHP :
```
     include 'xmlrpc.inc';
     include 'xmlrpcs.inc';
```


clients et serveurs PHP sont appelés depuis un navigateur, il faut donc les rendre accessibles au serveur web :
```
> ls ~/public_html/testrpc/
cli1.php  clidate.php      intros.php  ser3.php
cli2.php  clidiffjava.php  ser1.php    serdiff.php
cli3.php  clidiff.php      ser2.php
```

Puis on appelle le client, qui lui-même provoque la création du serveur : http://tuxa.sme.utc/~sr03/testrpc/cli1.php .
XML-RPC PHP Demo cli1.php -> ser1.php
Envoi messge au serveur ser1.php : calcul (7 * 2).
Résultat: 14.

XML-RPC : Sources des exemples

Premier exemple : serveur java standalone, client java

Code source du serveur

```
import java.util.Hashtable;
import org.apache.xmlrpc.*;

public class JavaServer {
    public JavaServer () {
        // Our handler is a regular Java object. It can have a
        // constructor and member variables in the ordinary fashion.
        // Public methods will be exposed to XML-RPC clients.
    }
    public Hashtable sumAndDifference (int x, int y) {
```



```
        Hashtable result = new Hashtable();
        result.put("sum", new Integer(x + y));
        result.put("difference", new Integer(x - y));
        return result;
    }
    public static void main (String  args) {
        try {
            // Invoke me as <http://localhost:port/RPC2>.
            WebServer server = new WebServer(8080);
            server.addHandler("sample", new JavaServer());
            server.start();

        } catch (Exception exception) {
            System.err.println("JavaServer: " + exception.toString());
```



```
        }
    }
}
```

Code source du client

```
import java.util.Vector;
import java.util.Hashtable;
import org.apache.xmlrpc.*;

public class JavaClient {

  // The location of our server.
  private final static String server_url =
```



```
      //"http://xmlrpc-c.sourceforge.net/api/sample.php";
      "http://tuxa.sme.utc:20000/RPC2";

  public static void main (String  args) {
    try {
        // Create an object to represent our server.
        XmlRpcClient server = new XmlRpcClient(server_url);

        // Build our parameter list.
        Vector params = new Vector();
        params.addElement(new Integer(5));
        params.addElement(new Integer(3));

        // Call the server, and get our result.
```

```
        Hashtable result = (Hashtable) server.execute(
                        "sample.sumAndDifference", params);
        int sum = ((Integer) result.get("sum")).intValue();
        int difference =
            ((Integer) result.get("difference")).intValue();

        // Print out our result.
        System.out.println("Sum: " + Integer.toString(sum) +
                           ", Difference: " +
                           Integer.toString(difference));

    } catch (XmlRpcException exception) {
        System.err.println("JavaClient: XML-RPC Fault #" +
                Integer.toString(exception.code) + ": " +
```



```
                exception.toString());
    } catch (Exception exception) {
      System.err.println("JavaClient:"+ exception.toString());
    }
  }
}
```

Appel du serveur RPC Java depuis un script PHP

```
Dans une fenêtre sur tuxa :
> setenv CLASSPATH .:/local1/xmlrpc-1.2-b1/xmlrpc-1.2-b1.jar
> java JavaServer 20000

Entrer dans le navigateur :
     http://tuxa.sme.utc/~sr03/testrpc/clidiffjava.php
```



```
Réponse :

XML-RPC PHP Demo
Sum: 8, Difference: 2
```

Source du client PHP

```
<html>
<head>
<title>XML-RPC PHP Demo</title>
</head>
<body>
<h1>XML-RPC PHP Demo</h1>
```



```
<?php
include 'xmlrpc.inc';
// Make an object to represent our server.
$server = new xmlrpc_client('/RPC2',
                            'tuxa.sme.utc', 20000);
// Send a message to the server.
$message = new xmlrpcmsg('sample.sumAndDifference',
                         array(new xmlrpcval(5, 'int'),
                               new xmlrpcval(3, 'int')));
$result = $server->send($message);

// Process the response.
if (!$result) {
    print "<p>Could not connect to HTTP server.</p>";
```



```
} elseif ($result->faultCode()) {
    print "<p>XML-RPC Fault #" . $result->faultCode() . ": " .
        $result->faultString();
} else {
    $struct = $result->value();
    $sumval = $struct->structmem('sum');
    $sum = $sumval->scalarval();
    $differenceval = $struct->structmem('difference');
    $difference = $differenceval->scalarval();
    print "<p>Sum: " . htmlentities($sum) .
        ", Difference: " . htmlentities($difference) . "</p>";
}
?></body></html>
```


Appel du serveur PHP depuis un script PHP

Code source du serveur PHP (fichier serdiff.php)

```
<?php
  include 'xmlrpc.inc';
  include 'xmlrpcs.inc';

  function sumAndDifference ($params) {
      // Parse our parameters.
      $xval = $params->getParam(0);
      $x = $xval->scalarval();
      $yval = $params->getParam(1);
      $y = $yval->scalarval();
```



```
    // Build our response.
    $struct = array('sum' => new xmlrpcval($x + $y, 'int'),
              'difference' => new xmlrpcval($x - $y, 'int'));
    return new xmlrpcresp(new xmlrpcval($struct, 'struct'));
  }
  // Declare our signature and provide some documentation.
  // (The PHP server supports remote introspection. Nifty!)
  $sumAndDifference_sig = array(array('struct', 'int', 'int'));
  $sumAndDifference_doc = 'Add and subtract two numbers';

  new xmlrpc_server(array('serdiff.sumAndDifference' =>
              array('function' => 'sumAndDifference',
                    'signature' => $sumAndDifference_sig,
                    'docstring' => $sumAndDifference_doc)));
```



```
?>
```

Code source du client PHP (fichier clidiff.php)

```
Il est le même que celui qui appelle le serveur Java (voir
plus haut), avec les modifications suivantes :
Les lignes :
  $server = new xmlrpc_client('/RPC2', 'sunserv.utc', 8080);
  $message = new xmlrpcmsg('sample.sumAndDifference',
sont remplacées par :
  $server = new xmlrpc_client('/~lo33/xmlrpc/php/serdiff.php',
                              'sunserv.utc', 80);
  $message = new xmlrpcmsg('serdiff.sumAndDifference',
            $
```

Appel du serveur PHP depuis un client Java

Code source du client Java

Il est le même que celui qui appelle le serveur Java (voir plus haut) avec les modifications suivantes :

Les lignes :

```
    public class JavaClient {
    "http://sunserv.utc:8080/RPC2";
    (Hashtable) server.execute("sample.sumAndDifference", params);
```

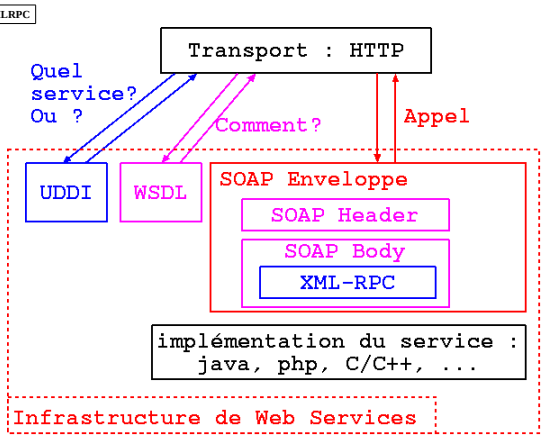
sont remplacées par :

```
    public class JavaClientPHP {
    "http://sunserv.utc:80/~lo33/xmlrpc/php/serdiff.php";
    (Hashtable) server.execute("serdiff.sumAndDifference", params);
```


Qu'est-ce que les Web Services ?

Les Web Services sont constitués d'un ensemble de standards :

- **XML**, pour décrire les informations et pour les messages échangés,
- **UDDI**, pour trouver les services,
- **WSDL**, pour décrire le fonctionnement des Web services,
- **SOAP**, pour exécuter à distance les WEB services (par un XML-RPC).



SOAP

- a framework for exchanging XML-based information in a network
- SOAP used to be an acronym : Simple Object Access Protocol
- "This is no longer the case." (it is neither simple nor has anything to do with objects)
- the currently most hyped XML/Web service technology
- mostly just hot air...

WSDL (Web Service Description Language)

- an XML-based language for describing network services
- WSDL descriptions of capabilities and locations of services
- like an interface description language for Web services
- communication using SOAP or direct HTTP

UDDI (Universal Description, Discovery, and Integration)

- provides a registry mechanism for clients and servers to find each other
- uses SOAP for communication

WSDL is an XML-based language that allows formal descriptions of the interfaces of Web services :

- which **interactions** does the service provide ?
- which **arguments and results** are involved in the interactions ?
- which **network addresses** are used to locate the service ?
- which communication **protocol** should be used ?
- which **data formats** are the messages represented in ?

So what are the benefits ?

- an interface description is a **contract** between the server developers and the client developers

- having formal descriptions **allows tool support**, e.g. code template generators (see the Apache AXIS project).

A WSDL description of a "stock quote" service :

```
<?xml version="1.0"?>
<definitions name="StockQuote"
             targetNamespace="http://example.com/stockquote.wsdl"
             xmlns:tns="http://example.com/stockquote.wsdl"
             xmlns:xsd1="http://example.com/stockquote.xsd"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
            xmlns="http://www.w3.org/2000/10/XMLSchema">
```



```
      <element name="TradePriceRequest">
        <complexType>
          <all>
            <element name="tickerSymbol" type="string"/>
          </all>
        </complexType>
      </element>
      <element name="TradePrice">
         <complexType>
           <all>
             <element name="price" type="float"/>
           </all>
         </complexType>
      </element>
```



```
    </schema>
  </types>

  <message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
  </message>

  <message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
  </message>

  <portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
      <input message="tns:GetLastTradePriceInput"/>
```

```
      <output message="tns:GetLastTradePriceOutput"/>
    </operation>
  </portType>

  <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetLastTradePrice">
      <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
```



```
      </output>
    </operation>
  </binding>

  <service name="StockQuoteService">
    <documentation>My first service</documentation>
    <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
      <soap:address location="http://example.com/stockquote"/>
    </port>
  </service>

</definitions>
```

- some data types are defined using XML Schema
- some simple message types are defined from the data types

- the port type describes a single operation GetLastTradePrice, which uses the message types for input/output
- the binding tells that the communication is through SOAP
- the port associates the binding with the URI http ://example.com/stockquote where the running service can be accessed

–> **this is all we need to write clients for the service** !

Using SOAP typically means :

- when sending messages, put them into a SOAP envelope
- when receiving messages, take them out of the SOAP envelope

An envelope is a wrapper containing :

- header - information to intermediaries (network nodes on the message path)

- a body - the actual contents (depending on the application)

Exemple d'implémentation

➤ **xml.apache.org/axis - Apache AXIS** (Apache Extensible Interaction System)

- a Java-based implementation of SOAP+WSDL
- largely allows the programmer to forget these technologies
- typically used together with Tomcat
- highly recommended !

Autre exemple : les serveurs d'application J2EE (e.g. JBOSS) incluent une implémentation des Web Services, en simplifiant leur utilisation (par ex. depuis eclipse avec JBOSS).

Constructing application-to-application Web services is **simple**, ***in principle*** : **SOAP, WSDL, and UDDI** are

- unavoidable frameworks for XML communication and service description
- **surprisingly complicated** and their **benefits are not always obvious**, although the basic ideas behind them are great

Questions sur Ch21 - XMLRPC

➤ Qu'est-ce que XMLRPC ?
Réponse : diapo 2.

➤ Qu'est-ce qui est échangé entre client et serveur lors d'un appel XMLRPC ?
Réponse : diapos 2 à 4.

➤ Que faut-il ajouter à un XMLRPC pour en faire un Webservice ?
Réponse : diapos 27 et 28. Une enveloppe SOAP.

Ch21 - XMLRPC

Fin du chapitre.

(diapo vide)

Ch22 - XML sur le Web

- XML, le navigateur et les styles : exemple livre*.xml
- XML et le DOM dans un navigateur : exemple exrss*.xml
- RSS
- Ajax
- XUL

Exemple livre*.xml

- **livre1.xml** : XML seul -> pas de formattage.
- **livre2.xml** : –> livre.css et livredtd.dtd
  ==> formattage et vérification.
- **livre3.xml** : comme livre2.xml, **avec en plus** des éléments **html** dans le fichier.
- **livre5.xml** : –> remplace l'appel à un style css par un appel à une "feuille de style" **xsl**.
- **livre6.xml** : –> appel à un style **xsl** qui fait lui-même **appel** à un style **css**.

livre1.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<livre>
  <titre>Le Cid</titre>
  <auteur>Corneille</auteur>
  <editeur>Hachette</editeur>
</livre>
```

livre1.xml

This XML file does not appear to have any style

- <livre>
  <titre>Le Cid</titre>
  <auteur>Corneille</auteur>
  <editeur>Hachette</editeur>
  </livre>

This XML file does not appear to have any style

+ <livre></livre>

livre2.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<?xml-stylesheet href="livre.css" type="text/css"?>
<!DOCTYPE livre SYSTEM "livredtd.dtd">
<livre>
  <titre>Le Cid</titre>
  <auteur>Pierre Corneille</auteur>
  <editeur>Hachette</editeur>
</livre>
```

livre2.xml

livre.css

```
titre
{ font-size: 24pt; color: blue;
}
auteur
{ font-size: 16pt; color: red;
}
editeur
{ font-size: 12pt; color: black;
}
```

Le Cid Pierre Corneille Hachette

livre3.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<?xml-stylesheet href="livre.css" type="text/css"?>
<!DOCTYPE livre SYSTEM "livredtd.dtd">
<livre xmlns:html="http://www.w3.org/1999/xhtml">
  <titre>Le Cid -</titre>
  <html:br/><html:hr />
  <auteur>Pierre Corneille -</auteur>
  <html:u>Ceci en html</html:u>
  <br/><hr />
  <editeur>Hachette</editeur>
</livre>
```

livre3.xml

inclusion et utilisation
de l'espace de nommage html

Le Cid -

Pierre Corneille - Ceci en html Hachette

livre5.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<?xml-stylesheet href="livre5.xsl"
                 type="application/xml"?>
<!DOCTYPE livre SYSTEM "livredtd.dtd">
<livre xmlns:html="http://www.w3.org/1999/xhtml">
  <titre>Le Cid -</titre>
  <html:br/><html:hr />
  <auteur>Pierre Corneille -</auteur>
  <html:u>Ceci en html</html:u>
  <html:br/><html:hr />
  <editeur>Hachette</editeur>
</livre>
```

livre5.xml

Depuis xsl : le titre = Le Cid -

Depuis xsl : auteur = Pierre Corneille -

Ceci en html Hachette

livre5.xsl

```
<?xml version="1.0"  encoding="ISO-8859-1" ?>
<xsl:stylesheet version="1.0" type="application/xml"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="livre">
    <html>
      <xsl:apply-templates/>
    </html>
  </xsl:template>
  <xsl:template match="titre">
    <P><hr/>Depuis xsl : le titre =
      <xsl:apply-templates/>
    </P>
  </xsl:template>
  <xsl:template match="auteur">
    <P>Depuis xsl : auteur =
      <xsl:apply-templates/>
    </P>
  </xsl:template>
</xsl:stylesheet>
```

livre5.xsl

Depuis xsl : le titre = Le Cid -

Depuis xsl : auteur = Pierre Corneille -

Ceci en html Hachette

livre6.xsl

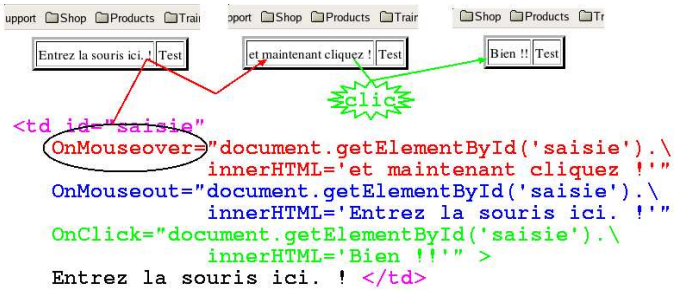
livre6.xml est identique à livre5.xml
(sauf: il appelle livre6.xsl)
livre6.xsl est quasi identique à livre5.xsl
il possède en plus l'instruction d'appel de
la feuille css:

```
<xsl:template match="livre">
   <xsl:processing-instruction name="xml-stylesheet">
       <xsl:text>href="livre.css"
                 type="text/css"</xsl:text>
   </xsl:processing-instruction>
   <xsl:element name="livre">
       <xsl:apply-templates/>
   </xsl:element>
</xsl:template>
```

livre6.xsl

Depuis xsl : le titre = Le Cid - Depuis xsl : auteur = Pierre Corneille - Ceci en html Hachette

Le DOM : Premier exemple : dom1.html




```
<td id="saisie"
   OnMouseover="document.getElementById('saisie').\
                innerHTML='et maintenant cliquez !'"
   OnMouseout="document.getElementById('saisie').\
                innerHTML='Entrez la souris ici. !'"
   OnClick="document.getElementById('saisie').\
                innerHTML='Bien !!'" >
   Entrez la souris ici. ! </td>
```


Le DOM : Premier exemple : dom1.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML..."
"http://www.w3.org.../DTD/xhtml1-strict.dtd">
<HTML><HEAD>
<meta http-equiv="Content-Type" content="text/html;
 charset=iso-8859-1">
<TITLE>Page test DOM 1</TITLE></HEAD>
<BODY>
<table cellpadding="2" cellspacing="2"
       border="3" align="center"><tr>
 <td id="saisie"
    OnMouseover="document.getElementById('saisie').\
                 innerHTML='et maintenant cliquez !'"
    OnMouseout="document.getElementById('saisie').\
                 innerHTML='Entrez la souris ici. !'"
    OnClick="document.getElementById('saisie').\
                 innerHTML='Bien !!'" >
    Entrez la souris ici. ! </td>
 <td id="truc"
 OnMouseover="document.getElementById('truc').\
                          bgColor='yellow'"
 OnClick="this.bgColor='red'">
 Test</td>
</tr></table>
</BODY></HTML>
```


Formatage d'un document XML par des CSS

On va examiner comment on peut traiter dans un navigateur un fichier purement XML. On va prendre l'exemple d'un fichier au format RSS et montrer comment il apparait en fonction de la façon de le décrire.

- **html** : si le fichier a pour nom **xxx.html**, seul le **contenu** des balises s'affiche , sans aucun formattage.
- **xml** : si le fichier a pour nom **xxx.xml**, et ne possède aucun élément de style, le navigateur affiche **l'arbre XML**.
- **xml + css** : si le fichier contient une indication de style comme :
  `<?xml-stylesheet href="exrss2.css" type="text/css"?>`
  alors, chaque élément est affiché selon les indications du .css

- **xml + css + javascript** : on peut ajouter au fichier précédent une instruction :
  `<xhtml :script type="text/javascript" src="exrss3.js" />`
  Si cette instruction est mise **au début** du fichier , elle n'agit pas sur le contenu, car celui-ci n'est pas encore disponible dans le DOM. Par contre mise **à la fin** du fichier , elle va changer le formattage de certains des éléments affichés .

Exemple : exrss1.xml

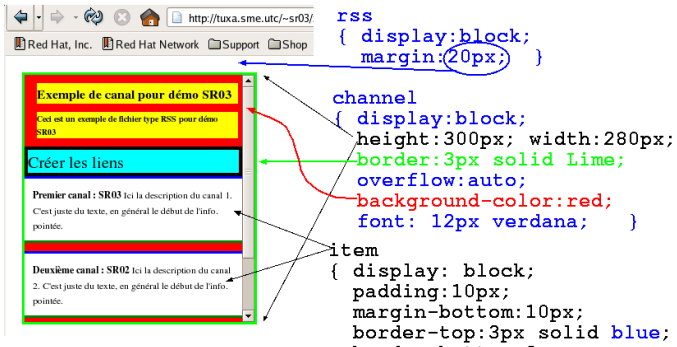
```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<rss version="0.91" xmlns:xhtml="http://.../xhtml">
  <channel>
    <title>Exemple de canal pour démo SR03</title>
    <link>http://tuxa.sme.utc/~sr03</link>
    <description>Ceci est un exemple...  </description>
    <language>en-us</language>
    <item>
      <title>Premier canal : SR03</title>
      <description>Ici la description ...</description>
      <link>http://tuxa.sme.utc/~sr03</link>
    </item>
    <item>
      <title>Deuxième canal : SR02</title>
      <description>description du canal 2</description>
      <link>http://tuxa.sme.utc/~sr02</link>
    </item>
    <item>
      <title>Troisième canal : responsable UV</title>
      <description>Ici ... canal 3. ...</description>
      <link>http://www.utc.fr/~vayssade</link>
    </item>
  </channel>
</rss>
```


Une feuille de style est associée à un fichier XML dans son préambule :

```
<?xml version="1.0"?>
<?xml-stylesheet href="monstyle.css" type="text/css"?>
```

Dans le fichier css on va indiquer les formatages des différents éléments du fichier XML.

Mais css ne fait que formatter du texte. Ici nous voudrions en plus **interpréter** certains éléments : par exemple que les éléments "items" soient des liens hypertextes.




```
rss
{ display:block;
  margin:20px;   }

channel
{ display:block;
  height:300px; width:280px;
  border:3px solid Lime;
  overflow:auto;
  background-color:red;
  font: 12px verdana;    }

item
{ display: block;
  padding:10px;
  margin-bottom:10px;
  border-top:3px solid blue;
  border-bottom:3px
                 solid green;
  background-color:#fff;    }
```


On va écrire une fonction javascript qui va utiliser l'API du DOM pour récupérer le contenu de certains éléments et les transformer.

```
  </channel>
<- doit etre mis a la fin pour avoir des elements sur lesquels agir --
<xhtml:script type="text/javascript" src="exrss3.js" />
</rss>
```

On va pour chaque élément <item>, récupérer son élément <title> pour en faire un lien :

On extrait la partie texte qui devient la partie texte d'un nouvel élément <a> donc d'un lien, le tag <link> du fichier de départ devenant l'élément "href" donc la cible du lien.

Puis, on remplace dans le DOM (dans l'arbre XML décrivant le document affiché) le fils <title> de l'item par le nouvel élément de type <a>.

Le fait de remplacer un élément du DOM du document affiché provoque une réinterprétation de celui-ci pas le browser (on a changé l'arbre XML, ce qui s'affiche est donc différent).

```
var xhtml = "http://www.w3.org/1999/xhtml";
var allItems = document.getElementsByTagName("item");
for (var i=0;i<allItems.length;i++)
{   var itemElm = allItems[i];
    var titleElm = itemElm.getElementsByTagName("title").item(0);
    var titleText = "==="+titleElm.firstChild.nodeValue+"---";
    var linkElm = itemElm.getElementsByTagName("link").item(0);
```



```
    var linkURL = linkElm.firstChild.nodeValue;

    var newLinkElm = document.createElementNS(xhtml,"a");
    var txtNode = document.createTextNode(titleText);
    newLinkElm.setAttributeNS("","href",linkURL);
    newLinkElm.style.display = "block";
    newLinkElm.style.color = "blue";
    newLinkElm.appendChild(txtNode);
    itemElm.replaceChild(newLinkElm,titleElm);  }
```

Une autre méthode serait d'utiliser **XLink** pour créer des hyperliens : dans ce cas la fonction "hyperlien" est donnée à un élément XML du DOM en lui attachant un attribut XLink.

Il suffit donc d'attacher cet attribut aux éléments désirés :


```
titleElm.setAttributeNS(xlink,"type","simple");
titleElm.setAttributeNS(xlink,"show","replace");
titleElm.setAttributeNS(xlink,"href",linkURL);
```

On insère ce code dans une fonction javascript exécutée au chargement du fichier XML car appelée depuis le XML par :

```
<xhtml :script type="text/javascript" src="exrss3.js" />
```

L'intérêt principal de cette méthode est qu'elle permet de transformer en lien hypertexte n'importe quel élément du document (paragraphe, case d'un tableau, image, partie encadrée par , etc...) et pas seulement un morceau de texte.

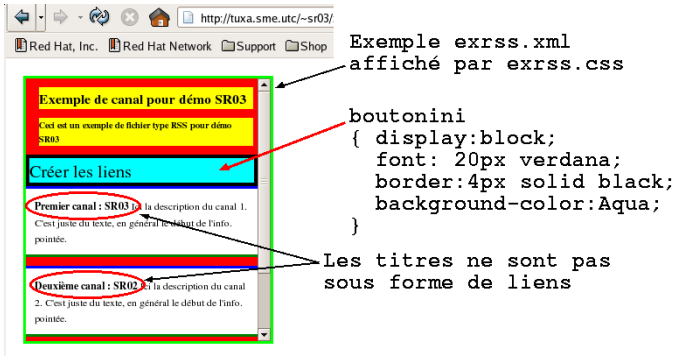
Modification dynamique d'un document XML

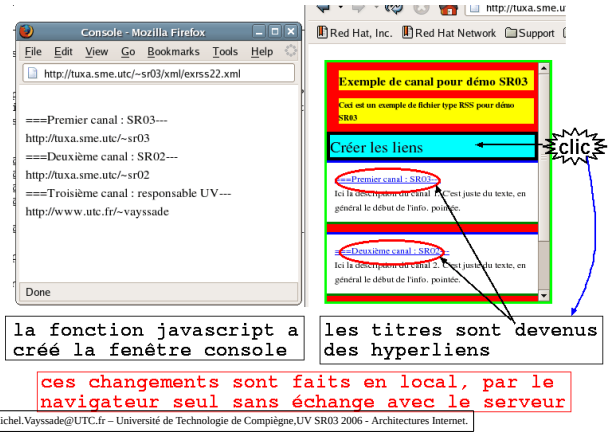
L'exemple suivant va montrer, à partir d'une modification simple de l'exemple précédent, la possibilité de modifier le document de façon dynamique **sans intervention du serveur** : tout se passe dans le navigateur.

On introduit l'élément `<boutonini>Makelinks</boutonini>` dans le fichier XML. Il va servir de "bouton".

Le code de création des hyperliens est encapsulé dans une fonction makeliens() qui ne sera exécutée que si elle est appelée. Ici elle sera appelée par un clic sur l'élément "boutonini". L'association étant réalisée en ajoutant à la fin du fichier exrss3.js le code javascript suivant :

```
var but = document.getElementsByTagName('boutonini');
but[0].onclick = makeliens;
```



RSS : qu'est-ce que c'est ?

Le signe : au départ c'était sans doute (Really Simple Syndication).

Il y a eu aussi d'autres appellations qui ont circulé :

- Rich Site Summary
- R(DF) Site Summary
- Rich Site Syndication
- Rich Syndication Standard

Peu importe, **RSS** fournit un moyen de gérer une **collection d'entêtes** ("headlines"), chaque élément étant accompagné d'un **lien** et d'un **résumé**. Le lien pointe sur une page web actualisée ou bien (c'est souvent le cas) sur une entrée (lien #cible dans une page) d'un blog.

L'idée est la suivante : un **fournisseur d'information** maintient et met à jour fréquement un fichier XML décrivant une série de**canaux**. Chaque canal comporte un certain nombre d'**items**. Un item pointe sur un élément d'information qui a été ajouté ou actualisé (cours de bourse, prévision météo, nouvelle entrée dans un blog, résultat sportif, communiqué d'agence de presse, etc ...).

Un utilisateur va **"s'abonner"** à plusieurs sources et utiliser une **application web** ou un plugin dans un browser pour **aller chercher** régulièrement la nouvelle version du fichier XML et afficher le résultat. Ceci se fait sans que l'utilisateur ai à intervenir (notion d'abonnement) : l'information se met à jour sur son écran.

Ajax

Qu'est-ce qu'Ajax ?

"AJAX" itself is a geeky acronym : "**A**synchronous **Ja**vaScript + **X**MLHttpRequest".

Au coeur d'Ajax, il y a l'objet **xmlhttprequest** qui permet des transferts de données **asynchrones** entre le serveur et le navigateur.

Supposons que nous ayons une page html composée de plusieurs éléments ; y compris un lien vers un fichier de style CSS et un lien vers un fichier de script javascript.

Avec un chargement synchrone, le script javascript doit être analysé avant de passer à l'élément suivant. Etc ...

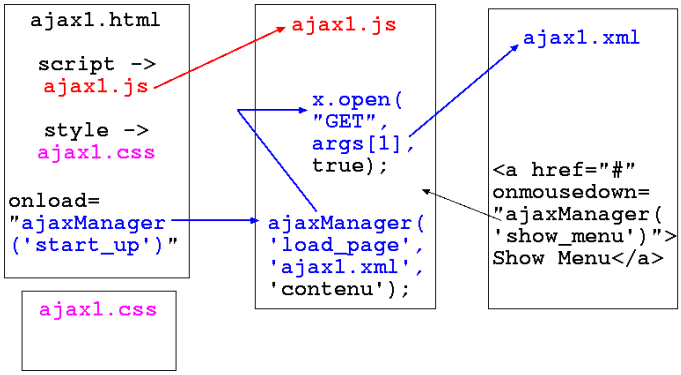
Tous les éléments de la section <head> doivent être pareillement traités avant que l'on commence à charger les éléments de la page elle-même. Or dans le cas de la section <head> se chargement se fait séquentiellement.

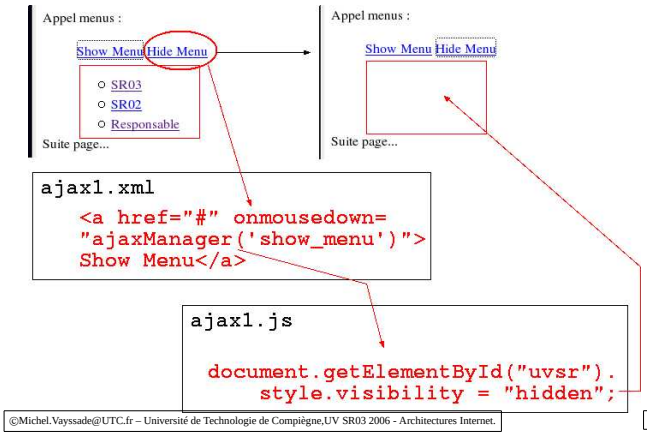
Une fois analysé les éléments de la section <body>, ils peuvent être chargés en utilisant plusieurs connexions. La plupart des serveurs web accepent de 2 à 4 connexions en parallèle pour charger dans le navigateur les éléments de la page (images, plugsins, ...).

Le fait de charger et analyser tous les éléments de la section <head> d'abord peut ralentir de façon importante l'affichage de la page. Surtout avec la tendance à mettre beaucoup de CSS et de Javascript.

L'idée pour contourner cette difficulté est d'utiliser un bout de JavaScript minimal au début, puis de récupérer tout le reste par l'intermédiaire de l'objet xmlhttprequest.

Ceci permet d'utiliser des connexions multiples (une par appel à xmlhttprequest) à la place d'une seule dans laquelle les chargements sont faits l'un après l'autre.





Ex : ajax1.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head><scripttitle>Hiding Menu</title>
<script src="ajax1.js" type="text/javascript">
</script>
<style href="ajax1.css" type="text/css">
</style>
</head>
<body onload="ajaxManager('start_up')">
<p>Appel menus :</p>
<div id="contenu">
</div>
Suite page...
</body></html>
```


Ex : ajax1.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<menu>
  <item>
    <p xmlns="http://www.w3.org/1999/xhtml" id="menus">
      <a href="#" onmousedown="ajaxManager('show_menu')">Show Menu</a>
      <a href="#" onmousedown="ajaxManager('hide_menu')">Hide Menu</a>
    </p>
  </item>
  <item>
    <ul xmlns="http://www.w3.org/1999/xhtml" id="uvsr">
        <li><a href="http://tuxa.sme.utc/~sr03/">
                SR03</a></li>
        <li><a href="http://tuxa.sme.utc/~sr02/" class="menu">
```



```
                SR02</a></li>
        <li><a href="http://www.utc.fr/~vayssade/" class="menu">
                Responsable</a></li>
    </ul>
  </item>
</menu>
```

Ex : ajax1.js

```
function ajaxManager()
{
var args = ajaxManager.arguments;
switch (args0)
   {
   case "load_page":
```

```
        if (document.getElementById) {
           var x = new XMLHttpRequest(); }
        if (x) {
            x.onreadystatechange = function()
            {
               if (x.readyState == 4 && x.status == 200)
                   { el = document.getElementById(args2);
                     el.innerHTML = x.responseText;
                   }
            }
            x.open("GET", args1, true);
            x.send(null);
        }
        break;
```



```
   case "start_up":
        ajaxManager('load_page', 'ajax1.xml', 'contenu');
        break;
   case "hide_menu":
        document.getElementById("uvsr").style.visibility = "hidden";
        break;
   case "show_menu":
        document.getElementById("uvsr").style.visibility = "visible";
        break;
   }
}
```

Valeurs possibles de x.readyState quand une XMLHttpRequest est en cours :

```
0   || uninitialized || Object is not initialized with data.
```



```
1   || Loading       || Object is loading its data.
2   || loaded        || Object has finished loading its data.
3   || interactive   || User can interact with the object even
                        though it is not fully loaded.
4   || complete      || Object is completely initialized.
```

À noter : XMLHttpRequest ne peut pas importer directement des images, des objets flash, des svg. La solution est alors de mettre ces objets dans une page html et d'importer cette page.

La page html importée ainsi devient partie intégrante du DOM de la page courante.

Qu'est-ce que XUL ?

The **XML User interface Language (XUL)** is a markup language for **creating rich dynamic user interfaces**.

C'est, en fait, l'architecture de construction de **Mozilla**. Elle est disponible comme étant une partie de **Gecko**.

Elle a été conçue pour être **portable** : elle est disponible sur toutes les versions de Windows, MacOS, Linux et autres Unix.

Avec XUL et les autres composants Gecko on peut créer des applications interactives complexes. Pas seulement des navigateurs.

–> http ://www.xulplanet.com

XUL a été conçue pour créer l'interface de Mozilla, aussi bien le

navigateur que le client mail et l'éditeur de pages.

Avec XUL on peut **créer une interface en utilisant un langage à balise** ("a markup language") : on utilise les **CSS** pour définir l'apparence et **JavaScript** pour le comportement.

XUL propose un ensemble assez riche de **widgets** pour créer des menus, des barres de menus, des barres d'outils, des arbres, etc...

XUL est un **langage XML**. On peut donc utiliser tous les standards XML pour manipuler l'interface crée : XSLT, Xpath, DOM,.... Le tout est directement supporté par Gecko.

XUL peut aussi être utilisé pour créer des applications locales ("standalone") qui **embarquent le moteur Gecko** ou bien qui peuvent être utilisées comme "plugins" d'un navigateur.

Ainsi une fonction de XUL appelée **overlay** permet des créer des **extensions au navigateur**, telles que ajouter une barre d'outils personalisée ou changer les menus.

Gecko supporte aussi des techniques de **"web services"** comme **XML-RPC, SOAP et WSDL**.

Standards and other Technologies Supported by XUL

```
* HTML 4.0 and XHTML
* CSS2 and some of CSS3
* DOM2 and some of DOM3
* XML
* MathML
* RDF
* XML-RPC
* SOAP (Simple Object Access Protocol)
* WSDL (Web Services Description Language)
* ECMAScript/Javascript
* XBL (eXtensible Bindings Language)
* GIF, JPG, PNG, BMP, ICO images
* XSLT
* XPath
* Simple XLinks
* XPointer
* XML Base
* FIXPtr
* HTTP 1.1
* POP3
* IMAP
* LDAP
```


Comment fonctionne XUL ?

Dans Mozilla, XUL est géré à peu près comme HTML : quand on entre l'URL d'une page html dans le champ adresse du navigateur, celui-ci charge le source de la page, puis le texte chargé est interpéte par le moteur de rendu pour produire le résultat graphique, lui-même organisé en un **arbre** (le DOM).

CSS, images, etc ... sont utilisés pour contrôler la présentation de cet arbre d'objets.

XUL fonctionne à peu près de la même façon.

Comme XUL et HTML sont utilisés de la même façon, on peut charger l'un et l'autre **soit depuis un serveur, soit depuis les fichiers locaux**.

Par dessus ce mécanisme de base de Gecko, Mozilla propose un

système d'installation dans le cadre du son système **"chrome"**.

Celui-ci (chrome) permet d'enregistrer des packages ayant des privilèges particuliers tels que lire des fichiers locaux, gérer les préférences de l'utilisateur, gérer les "bookmarks", etc...

Cette extension par package est la façon dont Mozilla et Firefox permettent l'ajout de fonctionnalités au navigateur.

Une extension est un ensemble de fichiers XUL, JavaScript, CSS et images "empaquetés" dans un fichier unique.

L'utilisateur a l'impression que l'extension a modifié le navigateur, mais en réalité le code est entièrement séparé et l'extension peut être facilement désinstallée.

Un package n'est pas obligé d'utiliser les overlays. Si un package ne le fait pas, il peut tout de même être accédé sur le système de fichiers local par une **URL spéciale : chrome ://**.

Ne pas oublier qu'un package accédé par chrome :// acquière les privilèges spéciaux déjà évoqués.

Il est intéressant de remarquer que **Mozilla lui-même** n'est en fait qu'un **ensemble de packages** contenant des fichiers XUL, JavaScript et CSS. Ces fichiers sont accédés par un URL chrome : et possèdent les privilèges associés comme n'importe quel autre package.

Si on veut utiliser XUL sur un site web, il suffit de poser les fichiers XUL sur le serveur comme des fichiers HTML et de les charger dans le browser par un URL. Il suffit d'ajouter dans la configuration du serveur web le "content type" 'application/vnd.mozilla.xul+xml'.

C'est ce "content type" qui va informer Mozilla de la différence entre ce fichier et du HTML ou du XML.

Donc, Mozilla peut gérer trois types de documents : HTML, XML et XUL. Le type XUL étant un sous-type du type XML.

Les fichiers d'un même package sont en général rassemblés sous la forme d'un **fichier JAR** qui peut être créé et examiné par un utilitaire de type ZIP.

Un tel fichier contient en général trois parties :

- **Content** - Windows definition and scripts
- **Skin** - Style sheets, images and other theme specific files
- **Locale** - Locale specific files

En résumé

- **Mozilla fait le rendu de HTML, XML et XUL** en utilisant le **même** moteur de rendu et les CSS pour définir la présentation ;
- **XUL** peut être **chargé** soit depuis un site distant soit depuis le système de fichiers local ou bien **installé** comme un package et accédé par une URL chrome :// (c'est ce que font les extensions au navigateur).
- **le URLs Chrome** peuvent être utilisés pour accéder des packages installés qui vont tourner avec des privilèges ;
- **HTML, XML et XUL** sont traités comme des types différents.

Ch22 - XML sur le Web. Exemples de questions

➤ Que se passe-t-il si on envoie un fichier XML "brut" à un navigateur moderne ?
Réponse : diapo 3.

➤ Que suffit-il de faire pour qu'un fichier XML soit "rendu" par navigateur moderne ?
Réponse : diapo 4. stylesheet href="xxx.css".

➤ Que **doit-on** faire pour pouvoir insérer du html dans un fichier xml et que ce html soit interprété comme tel par le navigateur ?
Réponse : diapo 5. faire appel dans le xml au **namespace** html (xmlns :html="...w3...".

➤ Que se passe-t-il si on ajoute à un document XML une référence

de type stylesheet href="xxx.xsl" ?
Réponse : diapos 6 et 7. Le fichier xsl désigné va être utilisé **à la place** de la méthode par défaut du navigateur pour rendre le fichier XML.

➤ Qu'est-ce que le DOM ?
Réponse : diapos 9 à 22. C'est le fait de rendre accessible par une API JavaScript les éléments de l'arbre XML représentant le document affiché dans le navigateur.

➤ Qu'est-ce que RSS ?
Réponse : diapos 23 et 24.

➤ Qu'est-ce que Ajax ?
Réponse : diapo 25.

➤ Qu'est-ce que Ajax permet de faire ?

Réponse : diapos 28, 29 et suivantes. Il permet de modifier l'arbre du DOM de façon dynamique et d'interagir en même temps avec le serveur web par l'envoi de requêtes asynchrones XMLHttpRequest.

➤ Qu'est-ce que XUL ?
Réponse : diapos 36 et 37.

➤ Comment fonctionne XUL ?
Réponse : diapos 40 et 41.

Ch22 - XML sur le Web.

Fin du chapitre.

Les serveurs d'applications

Différence entre objets métiers, classes et composants ?

- **objets métiers:** issu de la phase de modélisation (souvent UML)
- **classe:** c'est l'implémentation dans un langage d'un objet
- **composant:** code binaire conforme à un modèle (COM ou EJB), regroupe plusieurs classes et services techniques.

==> seul le composant à une réalité physique lorsqu'il est instancié dans la mémoire du serveur d'application.

Serveurs d'applications : innovation vraie ou concept marketing ?

les deux !

issus des expériences de développement d'applications transactionnelles à fort taux d'accès dans un contexte Internet.

objectif : développer une approche visant à réduire le plus possible la part de code "technique" (communication, sécurité, transaction, persistance, accès aux bases de données).

Différence entre serveur d'application et serveur web ?

À l'origine serveur de pages html statiques, le serveur web a été étendu par l'ajout de scripts côté serveur : CGI, ASP, PHP, servlets, JSP. Il est désormais capable de fournir des services applicatifs.

Un serveur d'application est un **conteneur** pour applications construites sur le modèle à trois niveaux "3 tiers" (présentation, logique métier, stockage de données).

L'évolution conduit à une fusion entre les deux technologies.

Situation de COM et EJB par rapport à Corba ?

Les serveurs d'applications offrent une solution globale de mise en production d'applications à base de composants.

Corba 2 se situe une **couche en dessous**. Il sert de socle technique pour l'implémentation de certains EJBs, mais de façon transparente.

Corba n'est **pas** un modèle de composants mais une spécification de **communication d'objets distribués**, plus quelques services de "*plomberie*" (nommage, évènements,...).

Corba 3 devrait introduire un modèle de composants indépendant du langage. Il est à craindre qu'il n'arrive trop tard pour s'imposer.

L'utilisation d'un serveur d'application fait-elle changer l'infrastructure existante ?

Heureusement non. C'est même le contraire : beaucoup de serveurs d'application ont été installés pour "consolider" une vision globale d'un ensemble existant hétérogène.

Les serveurs d'applications

Le modèle d'architecture <u>à trois niveaux</u>

À l'origine le but du modèle à trois niveaux (en anglais "three-tier"), était de distinguer 3 niveaux physiques : le poste client, la machine de traitement et la machine de gestion de base de données.

Le but étant d'optimiser la répartition de la charge de travail entre les postes de travail et le serveur de données en introduisant un intermédiaire chargé de passer les requêtes et les réponses.

Par extension le modèle a désigné une architecture logique séparant les **fonctions** de **présentation**, de **traitement** et de **données**.

Aujourd'hui les objectifs de ce type d'architecture sont :

- améliorer la vitesse de développement de nouvelles application par la réutilisation de **"briques logicielles"** existantes,
- autoriser l'accès aux applications par des canaus variés : intranet, Internet (personnel itinérant), WAP, postes de travail variés,
- réutiliser l'existant ("legacy applications" : applications héritées,
- permettre une évolution de l'un des trois niveaux de façon relativement indépendante;

Différents modèles de répartition

Il y a plusieurs façon de répartir les différentes couches de traitements entre les 3 niveaux.

Les limites du "client-serveur"

Client-serveur traditionnel :

- serveur ne traite que les données
- client accède aux données par l'intermédiaire d'un **middleware** chargé d'optimiser les requêtes SQL du client
- middleware propriétaire : un par fournisseur de base de données
- middlewares très complexes à utiliser
- pas de normalisation
- succès fin des années 80
- utilisation de l'interface windows par l'intermédiaire d'AGL (delphi)

Pour palier aux inconvénients de ce modèle, se sont généralisés des **moniteurs de traitements** proposant aux clients des "services" sous forme de procédures stockées. On aboutit alors à un modèle client-serveur de traitements.

On a alors un modèle à trois niveaux : client, serveur de traitements, base de données. **Exemple de moniteurs de ce type :** Tuxedo de BEA, CICS d'IBM

L'échec de la distribution

- prise en compte existant centralisé
- + limites des réseaux
- => **contraint** architectes à déployer des systèmes distribués

C'était un mal nécessaire, non un fin en soi, tellement la distribution compliquait le système (de la conception à l'exploitation).

Conséquences de la distribution :

- distribution des données ==> complexité de la gestion de la donnée de référence
- distribution ==> complexité de mise à jour et synchronisation
- distribution ==> impossibilité d'une mise à jour transactionnelle quand l'un des éléments du système peut ne pas être présent (cas des application "B2B" (business to business) ou l'échelle (réseaux distants, voire l'Internet), ne permet plus de garantir la disponibilité de tous les éléments

Conséquences de l'utilisation de middlewares propriétaires :

- chaque base de donnée à son propre middleware
- si une application utilise plusieurs bases de données, de constructeurs différents (cas fréquent), il faut les installer tous sur les postes avec une gestion cauchemardesque des chaines (version d'OS, DLLs, versions des applications, etc ...)

Et les objets distribués apparurent !

- moyen astucieux de distribuer données ET traitements,
- moyen d'y accéder de façon normalisée
- deux standards :
  - COM/DCOM
  - CORBA

MAIS :

- complexité des protocoles,
- interdépendances créées entre les objets,
- complexité de la distribution des annuaires décrivant les **composants**, leur **localisation** et leurs **interfaces**.

Cette complexité devrait être prise en compte par des ateliers de développement.
Malheureusement ceux-ci sont arrivés tard, sont complexes à utiliser et manquent de maturité.

Finalement, l'application du concept de distribution viendra du web qui a détourné l'intérêt des architectes.

Il semble se dégager un consensus pour estimer que les technologies utilisées sur les postes de travail seront issues du web.

Les technologie web

Le succès du web est dû d'abord à sa simplicité et à son efficacité pour la diffusion de documents.

Toutefois l'ajout d'une intelligence applicative se fera dans la douleur : CGI, puis extensions propriétaires aux serveurs : ASP, PHP, JSP (Java Server Pages). Le but étant de faciliter le plus possible la génération automatique des pages.

L'échec du "network computer"

Le succès du web a conduit certains (SUN, Oracle, IBM) a essayer de remplaçer le poste de travail par un "super navigateur" chargé de toute l'interaction avec l'utilisateur.

Mais plusieurs erreurs ont été commises :

- manque de standard, chacun "ramant" dans son coin pour imposer sa solution,
- n'avoir pas apporté de réponse à l'existant (bureautique et applications client-serveur qu'il fallait continuer à utiliser, puisqu'on ne pouvait pas tout jeter d'un coup),
- la lenteur de téléchargement et d'exécution des applications java sur ces "pizza box" à empêcher toute utilisation professionnelle :
  - les réseaux étant à l'époque insuffisants (ethernet 10 partagé),
  - les interfaces réseaux des serveurs saturant et le logiciel

réseau (la pile IP) faisant goulôt d'étranglement dans les serveurs,
  - les CPUs de ces "NC" étant trop faibles pour compenser le surcoût de l'interprétation de Java, à une époque ou les compilateurs JIT ne parvenaient que jusqu'à un rapport 5 au mieux.

Du mainframe en mode texte au mainframe en mode graphique => retour à la case départ !

On a vu successivement :

- les grappes de terminaux alphanumériques 3270 sur des mainframes IBM
- la connexion de PCs en mode "fat client"--"serveur de données"
- le déferlement du web pour la diffusion d'information
- les balbutiements de la logique applicative intégrée au web
- le retour au pragmatisme marqué par une **consolidation** des serveurs, qui en fait le retour à **une logique centralisée**.

C'est le besoin d'offres de service sur des canaux de distribution hétérogène qui a (semble-t-il définitivement) imposé cette solution qui simplifie les modèles de distribution au profit d'une architecture multi-niveaux (3 ou 4), séparant **présentation**, **logique métier** et **persistance des données**.

Restait à mettre en oeuvre ce modèle de façon économique (en temps de développement et de maintenance).

C'est le rôle dévolu aux **serveurs d'application**.

Alors tout le monde est d'accord ?

Oui, ... mais :

- les concepts ne sont pas triviaux,
- les outils ne sont pas matures,
- le niveau d'expertise pour mettre en oeuvre un serveur d'application est élevé,
- les bénéfices de l'utilisation de ces technologies ne sont perceptibles qu'à moyen terme (coût du ticket d'entrée en termes d'acquisition de compétences).

Les architectures multi-niveaux à base de composants hébergés dans des serveurs d'applications sont une tendance lourde du marché.

Les architectures transactionnelles

Le transactionnel provient des mainframes IBM des années 70, où des ***milliers*** de terminaux alphanumériques (en mode texte) sont connectés à une machine centrale.

Le terme ***transaction*** désigne alors un ***ensemble d'écrans*** qui doivent s'enchainer pour conduire à une modificatio durable de la base de données. Ces transactions sont bien sûr mono-moniteur et mono-base (CICS ou IMS sur DB2).

Le modèle DTP

Un modèle transactionnel a été proposé par l'**X/Open** : le modèle DTP : Distributed Transaction Service, processus mettant en oeuvre des ordinateurs en réseau.

Le modèle propose :

- une API permettant d'invoquer le moniteur transactionnel (TM) : l'interface **TX**,
- une API utilisée par le moniteur transationnel pour invoquer les Ressource Manager (RM) : l'interface **XA**. ces RM sont les sources de données ou les traitements distribués.

Ces interfaces permettent de mettre en oeuvre le protocole de validation à deux phases (**two phases commit**).

Le Transation Manager la cohérence : propriétés **ACID**:

- **Atomicité**: toutes les mises à jour ou aucune,
- **Cohérence**: on passe d'un état cohérent à un autre état cohérent,
- **Isolation**: les résultats ne deviennent visibles aux autre processus qu'une fois la transaction validée,
- **Durabilité**: une fois la transaction validée, le système garantit que les modifications sont durables, même en cas de panne.

Les **TM** sont mis en oeuvre dans des moniteurs transactionnels tels que Tuxedo, les **RM** se trouvent dans les principaux SGBD/R (Sybase, Oracle, DB2) et dans certains MOM (Middleware Orientés Messages) tel que MQSeries (IBM).

L'interopérabilité des TM et des RM est assurée par le support de **XA**.

Limite : le modèle DTP repose sur une **hypothèse très contraigante** : la transaction doit être **courte**. En cas de panne du TM après le début de la phase 1, la correction du protocole de validation n'est pas garantie.

Objets et transactions

En plus de l'interopérabilité (qui est le point fort de CORBA: langages hétérogènes, machines hétérogènes, fournisseurs d'ORBs différents), l'OMG a dû intégrer la notion de de transaction. On parle alors **d'objet transactionnel**.

Le service transactionnel de Corba est décrit dans les spécifs "Corba services".

Il assure l'exécution de transactions impliquant plusieurs objets et sources de données réparties, en conservant les propriétés ACID.

Les composants de service de transactions Corba sont :

- OTS, Object Transaction Service, compatible avec le modèle DTP de l'X/Open,
- le serveur transactionnel, gère les objets modifiés au cours d'une transaction,
- le serveur de restauration (commit/rollback),
- le client transactionnel, modifie les objets et active la validation.

On trouve de nombreuses implémentations de ce service dans les ORBs commerciaux, entre autres :

- Orbix OTM d'Iona,
- Visibroker ITS d'Inprise-Borland.

Même si les concepts de transactions sont simples à comprendre, ils **sont toujours compliqués à mettre en oeuvre**. Ils font partie des **services de base attendus** des serveurs d'applications.

Le poste de travail

Les serveurs d'applications permettent la mise en oeuvre de différents types d'architectures :

- client-serveur traditionnel Win32,
- applications Java,
- applications Win32 émulées,
- applications web.

L'application client-serveur traditionnel Windows

- Le client est une appli win (.exe), **développée avec un outil visuel** (**L4G**) (Power Builder, Visual Basic, Delphi, ...).
- Elle utilise des composants via le réseau par l'intermédiaire d'un **middleware** (logiciel d'intermédiation): **DCOM** ou

CORBA/IIOP/RMI.

- Ceci implique la présence de **proxies**: passerelle entre l'application et le middleware, qui doit être fournie par l'éditeur du L4G.
- **Nota:** les proxies dont on parle ici sont les **stub et skeletons** des appels RPC et Corba.

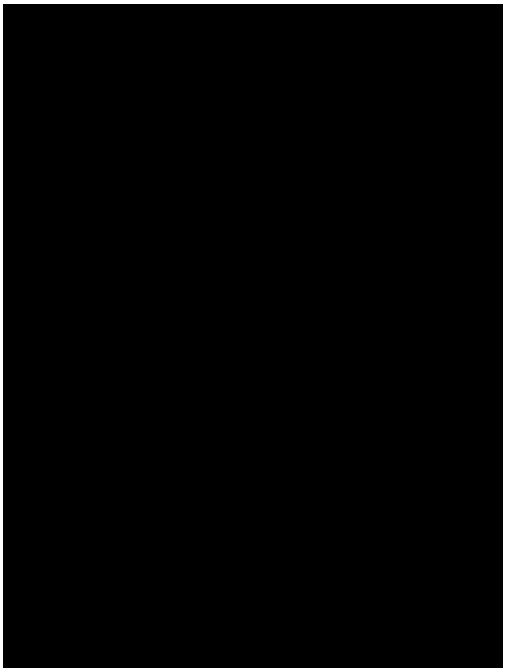
Avantages

Les ateliers L4G sont des produits matures et stabilisés.
Ils permettent de développer rapidement des interfaces graphiques.

Inconvénients

- Le démarrage à distance de composants serveurs nécessitent la présence **sur le poste client** de bibliothèques DCOM ou IIOP
- Les proxies (composants d'accès) doivent être redéployés sur chaque client si l'interface est modifiée.
- DCOM impose de les enregistrer dans la base de registres.
  Il faut aussi y enregistrer le **nom de la machine serveur**.
- L'utilisation de DCOM ou IIOP s'accomode mal des **coupes feu (firewalls)**, il faut alors **encapsuler** DCOM ou IIOP dans HTTP (**HTTP tunneling**).
- Ce type de client n'a de sens que lorsque la configuration des postes clients est totalement maîtrisée.
- Déployer du COM/DCOM reste un casse-tête : versions d'OS, services packs, clés de Registres, versions de DCOM, ...

Application ou applets Java



Avantages

- Utilisation "simple" puisque Corba spécifie les proxies en Java.
- Java garanti "en théorie" la portabilité.
- Le mécanisme **"d'auto-déploiement"** à travers un navigateur est simple et désormais performant (par les conteneurs **.JAR**).
- Les ateliers Java ont ratrapé les L4G traditionnels (JBuilder, Visual Café, Visual Age,...)

Inconvénients

- L'utilisation des fonctions évoluées d'interface homme-machine nécessite la présence d'un "runtime" récent (Java 2).
- Ceci nécessite la maîtrise des postes clients à cause de petites différences dans l'implémentation des JVM;
- Les applications Java restent plus lentes au démarrage.
- La promesse de Java *"write once, run everywhere"* reste une promesse. Ainsi une même appli Java essayée sur Netscape, Internet Explorer, JRE-Solaris, Linux, Mac verra des différences, de "petites" à "rédhibitoires" (polices, alignements, refraichissements, raccourcis,...).

Émulation Windows

- MetaFrame de Citrix, couplé avec la version Windows Terminal Server (WTS) permet d'exécuter des applications windows sur un serveur NT en déportant l'affichage sur le poste client (comme en X11, 20 ans après).
- Prérequis : installér un client WTS ou Cytrix sur le poste client (clients Citrix existent pour w9x, wnt, w2k, Mac et les principaux unix).
- Le client Citrix utilise le protocole **ICA** au-dessus de TCP/IP et est **peu gourmant** (30 kilobits/sec par client).

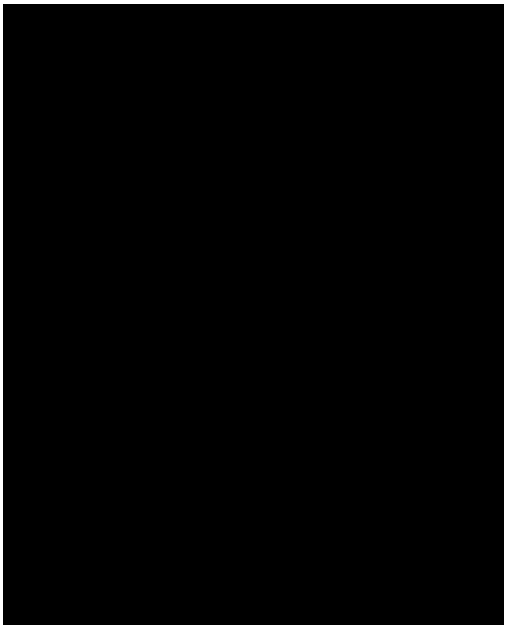
Avantages

- Accès aux applis windows sans les coûts de déploiement.
- Utilisation à distance sur lignes à faible débit.
- Accès aux applis windows sur postes unix.

Inconvénients

- Coût : une licence Citrix + une WNT **par poste client**
- Certaines applis ne fonctionnent pas en mode multi-utilisateur, ou bien doivent être modifiées.

Application web



- Utilisation de HTTP entre le client et le serveur.
- On utilise alors un serveur web "en frontal" des applications serveurs. Il reçoit les requêtes des postes clients par HTTP, et les transmet aux applications serveurs par une passerelle (JSP, Servlet, PHP).

- Le client *logique* des composants serveurs n'est plus le poste client, mais l'extension "passerelle" du serveur web, qui se charge de retraduire les échanges en HTTP.

Avantages

- Résoud au mieux les problèmes de déploiement.
- Utilise une nouvelle interface utilisateur : **la page HTML et le lien hypertexte**.
- Seule méthode permettant un accès "universel".

Inconvénients

- La richesse et l'ergonomie de l'interface est moindre que celle des interfaces natives.
- Cet inconvénient se résorbera au fur et à mesure de la généralisation du DHTML et des CSS.

Miser sur les technologies issues du web est un bon pari.

Les composants

Qu'est-ce qu'un composant ?

Le composant est un **objet** qui adhère à un **modèle**, c'est-à-dire qu'il supporte un ensemble prédèfini de **méthodes**.
Ces méthodes (choisies par l'auteur du modèle de composant et non pas l'auteur du composant) permettent : • d'appeler (d'instancier puis d'exécuter) le composant: un client extérieur au serveur d'application provoque le chargement du composant dans le serveur puis son exécution au sein du serveur d'applications. • d'être construit par un atelier de type L4G grâce à des méthodes **d'introspection**.

Les méthodes disponibles dans le composant sont regroupées sous forme **d'interfaces** (on retrouve l'**IDL**).
Le serveur d'application doit supporter le transactionnel, la sécurité, la persistance, l'accès concurrent, etc ... sans ajout de code spécifique à ces fonctions dans le composant.
Ceci diminue la complexité de mise en oeuvre d'une architecture à trois niveaux.

Les modèles de composants

- COM/DCOM, issu de OLE, appelé ActiveX,
- les EJB (Enterprise JavaBean), d'origine SUN et IBM,
- Zope serveur d'applications Python,
- ? Corba 3, si des implémentations arrivent à temps.

Les deux principaux sont **COM** et **EJB**.

Le modèle COM

Les composants COM peuvent être en C++, VB, ...développés le plus souvent avec un outil (Delphi, PowerBuilder, ...).
==>Le choix de COM impose *de facto* la plate-forme WNT sur le serveur intermédiaire.

Le modèle EJB

C'est un environnement d'exécution adapté à des objets écrits en Java.

La spécification de cet environnement, faite par SUN et IBM s'appelle **Enterprise Java Beans**.

C'est une architecture en couches:

- l'EJB server contient l'EJB Container et lui fournit des services de bas niveau,

- L'EJB Container est l'environnement dans lequel s'exécute l'EJB (le Beans).
- les clients se connectent à une représentation du Bean fournie par le conteneur qui route les requêtes vers le Bean.

L'EJB Container fournit des services aux Beans :

- support du mode transactionnel,
- gestion des instances multiples, pool d'instances, cache, ...
- persistance (à partir de version 2.0),
- sécurité (par le biais d'ACLs au niveau du Bean ou des méthodes du Bean),
- (en option) une administration des Beans et une gestion des versions.

Le Bean

Le fournisseur d'un Bean doit fournir une **home interface (EJBHome)** et une **remote interface**.

La séquence d'appel ressemble à la suivante :

- le client appelle JNDI (service de nommage) et récupère une **référence** sur un objet implémentant EJBHome.
- le client appelle une méthode de EJBHome pour obtenir une **référence** sur l'interface Remote (qui dérive de EJBObject),

On trouve deux types de Beans :

- les **Session Beans** créés en réponse à **une** requête **d'un** client, ils peuvent être **sans état** ou **avec ét&t**, c'est-à-dire garder trace d'un appel à l'autre de valeurs dans leurs variables memebres;
- les **Entity Beans** ont vocation à "vivre" longtemps et à être partagés par différents clients. Leur persistance peut être gérée par le conteneur ou par eux-mêmes. Ils peuvent être réenrants ou non.

Déploiement des Beans
Un serveur EJB s'appuie le plus souvent sur un **moteur Corba** qui fournit les services de bas niveau (localisation d'objets, gestion d'évènements, appel de méthodes distantes, ...). Mais ce n'est pas obligatoire.
Ainsi, EJB encapsule les aspects Corba de bas niveau : IDL, BOA, POA, ...

Le serveur d'applications

Les services de base du moteur de composants

Un serveur d'applications doit disposer de 4 services de base :

- l'accès aux composants,
- l'optimisation de l'accès aux ressources locales et distantes,
- la gestion transactionnelle,
- la répartition dynamique de charge.

L'accès aux composants

- permettre aux clients de **localiser**, **instancier** et **utiliser** les composants,
- un service de nommage (JNDI : Java Naming and Directory Interface, pour les EJBs, interface générique d'accès aux annuaires). L'implémentation peut utiliser LDAP, un fichier, une base de données, le DNS ...
- après localisation et obtention d'une référeence, le client va utiliser l'objet à travers son interface EJBObject en passant par RMI ou IIOP.
- dans le cas de COM, l'annuaire est **distribué** dans la base de registre des clients, ce qui complique beaucoup l'utilisation sauf dans le cas d'une application web/COM ou seuls quelques serveurs participent à l'accès DCOM.

L'optimisation de l'accès aux ressources locales et distantes

Les composants sont de **gros consommateurs** de ressources (mémoire, UC, E/S, réseau). L'optimisation des ressources va porter sur :

- en local, gestion des threads et de la mémoire : mécanismes de pré-instanciation, et de pools d'objets;
- interaction entre composants : sémaphores, IPCs, ...
- vers l'extérieur : connexion aux bases de données (pool de connexion, ne pas réouvrir une session de base de données chaque fois);

La gestion transactionnelle

La gestion de transaction peut être faite par programme (dans le code du composant) ou de façon déclarative. Le serveur se charge du reste : début de transaction (start), validation (commit) ou annulation (rollback).

La répartition dynamique de charge ou *fail over*

Un composant doit pouvoir être déployé sur plusieurs serveurs afin d'augmenter la capacité à monter en charge (scalability) et la robustesse.
Les serveurs EJB les plus performants disposent de **services de routage des requêtes** vers les machines les moins chargées d'une grappe (algorithmes simples de round-robin ou plus complexes avec poids sur la puissance et la charge).

Dans le cas de Beans de session, il est préférable de router toutes les requêtes d'un même client sur la même machine afin d'éviter une migration des données internes du Bean qui serait plus coûteuse que la répartition de charge. On utilise alors l'adresse IP du client pour l'orienter vers la même machine.

L'administration et l'exploitation

Dépendant du produit utilisé. Devrait proposer :

- packaging et distribution
- inventaire et gestion de versions (existe rarement)
- outils de surveillance, ordonnancement, sauvegarde

Sur ces points la plupart des produits manquent de maturité.

Sécurité

Elle se décompose en trois niveaux :

- **authentification** (qui se connecte ?) : centralisation sur un annuaire LDAP
- **contrôle d'accès** (untel as-t-il le droit de ?) : il est souhaitable de pouvoir gérer des autorisations avec une granularité au niveau des méthodes des objets (et pas seulement des objets eux-mêmes);
- **intégrité et confidentialité** (utilisation de SSL pour les échanges HTTP et IIOP avec des clés DES ou RC4 de 128 bits)

La plateforme J2EE

Les entreprises aujourd'hui

- apparition d'Internet **==>** nouveau créneau commercial
- changement radical : **l'information détenue** par une entreprise devient de plus en plus **précieuse**
- **==>** vital d'exploiter au mieux l'information détenue
- **==>** adapter les nouvelles techniques (i.e. le web) aux applications et systèmes hérités ("legacy systems")

Paysage technique et économique en mutation rapide. **==>**

- capacité de réactions rapides
- productivité élevée en programmation : capacité à développer et déployer des applications de façon efficace et rapide
- tâche complexe à cause multitude de technologie et standards (html, http, javascript, php, cgi, perl, dhtml, css, xml, xhtml, xslt, java, jsp, servlets, j2ee, rpc, rmi, corba, jdbc, jndi, ldap, ...)
- les applications web doivent fonctionner en **permanence**, et garantir la fiabilité des transactions,
- les informations détenues sont précieuses ==> assurer leur confidentialité,
- les applications doivent pouvoir évoluer, gérer une montée en charge ("scalability") et utiliser au mieux les ressources système
- utiliser les données présentes dans les systèmes d'information existant

Architecture système

Architecture à deux niveaux

Les systèmes traditionnels sont organisés en une architecture à deux niveaux séparant les **données** de la **logique de traitement et de présentation** (business logic).

Dans cette architecture le PC-client fait tout le travail de logique et présentation et le serveur gère l'échange de données entre les clients et la base de données.

Ceci peut conduire à de très nombreuses demandes de données pour pouvoir présenter un écran, pénalisant le trafic réseau et l'accès à la base de données.

De plus, l'approche à deux niveaux pose un grave problème de **maintenance**. En effet, la moindre modification entraine l'obligation de modifier tous les postes clients. Le résultat pouvant être des clients utilisant des versions **différentes** de la même application.

C'est pour éliminer ce risque et réduire cette difficulté que la communauté informatique a créé la notion d'**architecture à trois niveaux.**

Architecture à trois niveaux

Les trois niveaux sont distincts et pourvus chacun d'un ensemble **d'interfaces bien défini**.

- le niveau **présentation**, constitué d'une interface graphique,
- le niveau **métier**, contenant la logique applicative,
- le niveau **données**, contient les données pouvant provenir de plusieurs sources (BDD, fichiers, annuaire, ...).

Architecture multiniveaux

En fait, on peut avoir intérêt à découper encore plus l'application en distinguant dans le deuxième niveau, la logique de présentation et la logique métier, ainsi que des **services d'infrastructure** fournissant des fonctionnalités aux composants de l'application (messagerie, annuaire, transactionnel, persistance), fonctionnalités qu'il n'est plus nécessaire de coder.

Architecture d'entreprise

Jusqu'ici nous avons décrit l'architecture **d'une** application.

Mais dans une entreprise il peut y avoir des dizaines, voire des centaines d'applications.

Pour transformer un système d'applications multiniveaux en un système d'entreprise, il "suffit" **d'étendre le niveau intermédiaire** de sorte qu'il accueille plusieurs **objets applicatifs**.

Ces objets applicatifs devront être **pourvus d'une interface** pour pouvoir coopérer les uns avec les autres.

Une interface est un **contrat**: chaque objet publie, annonce, garanti, via son interface les données qu'il accepte ou fourni et les actions qu'il fera.

Il est alors possible de disposer d'un ensemble d'applications partageant un ensemble de **composants** communs.

Dans une telle architecture, il est évidemment fondamental d'éviter les changements dans la définition des interfaces. Pour cela il est conseillé de les faire aussi générique que possible.

La plateforme J2EE

J2EE est l'une des solutions pour répondre aux besoins exprimés ci-dessus. J2EE spécifie à la fois :

- une infrastucture de gestion des applications, permettant de les exécuter,
- un ensemble d'API (extensions à Java) pour concevoir des applications.

J2EE spécifie les **rôles** et les **interfaces** pour les applications, ainsi que l'environnement d'exécution. Cet environnement d'exécution fourni les services d'infrastructure (transaction, persistance, annuaire, ...) qui étaient traditionnellement dévelopés dans les codes applicatifs.

J2EE ne spécifie ni la nature ni la structure de l'environnement d'exécution. À la place, il introduit un **conteneur**, et via les API de J2EE, il élabore un **contrat** entre les conteneurs et les applications.

Les API J2EE

- JDBC
- RMI / IIOP
- EJB
- Servlets
- JSP
- JMS
- JNDI
- Java Transaction
- Java Mail

Architecture J2EE \ Les conteneurs

Une plateforme commerciale J2EE comporte un ou plusieurs conteneurs. Un conteneur est un environnement d'exécution Java2 chargé de gérer des composants applicatifs et de leur donner accès aux API J2EE.

- conteneur web = machine JVM exécutant des composants **servlets** et/ou **JSP**
- conteneur EJB = machine JVM exécutant des composants **EJB**

Les composants (Servlets, JSP, EJB) sont des programmes Java2 développés, compilés puis hébergés à l'intérieur d'un conteneur.

Pour cela, en plus du code Java2, le développeur doit fournir un **descripteur de déploiement**. C'est un fichier XML **décrivant le composant**. Il fournit au conteneur les éléments nécessaires pour gérer le composant.

En utilisant le **contrat du composant** (inclus dans le descripteur de déploiement), le conteneur pourra créer des instances du composant à l'intérieur de la JVM du conteneur, puis invoquer des méthodes sur ces instances.

C'est un peu comme lorsque l'on ajoute de nouvelles fonctions RPC à une bibliothèque RPC. Ces fonctions doivent s'enregistrer auprès du portmapper en donnant une référence correspondant à leur définition IDL.

Pour que cela fonctionne, le conteneur suppose que le composant a étendu ou mis en oeuvre un certain nombre d'interfaces. Il peut donc appeler les méthodes correspondantes. Si un composant "oublie" d'implémenter une interface nécessaire, le conteneur n'a aucun moyen pour appeler ces méthodes et le composant ne sera pas utilisable au sein du conteneur (même si c'est par ailleurs un programme Java fonctionnel).

Avec J2EE tous les composants applicatifs sont instanciés et initialisés dans la JVM du conteneur.

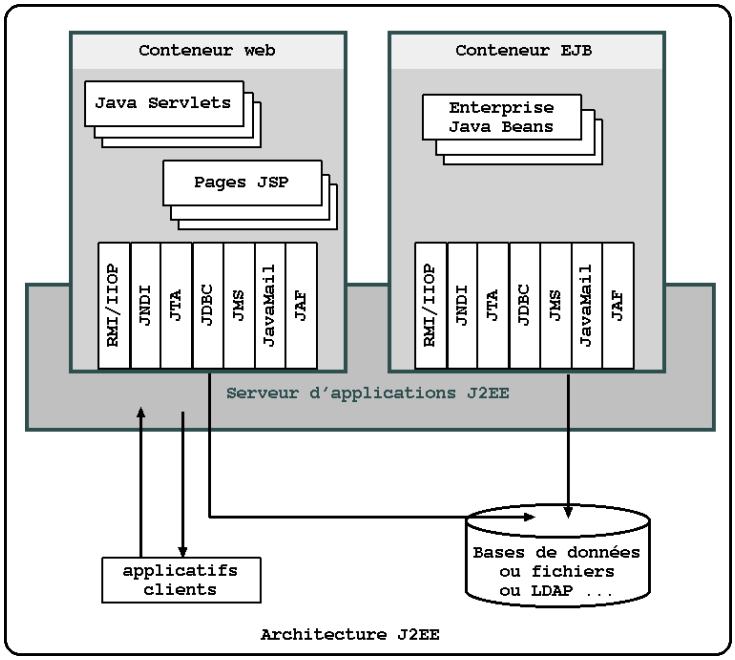
Les composants étant **distants** par rapport aux clients, ceux-ci ne peuvent pas appeler directement les méthodes des instances des

composants. Le conteneur est la seule chose connue des clients. Les composants doivent donc être conformes au contrat pour pouvoir être utilisés.

Le contrat sont les interfaces que le conteneur **doit** implémenter, et les classes qui contiennent cette implémentation. **Plus** quelques règles et conventions à appliquer lors de la définition du composant, afin que le conteneur puisse "deviner" les noms de certaines fonctions.

Exemples :

- les servlets doivent étendre la classe `java.servlet.http.HttpServlet` et mettre en oeuvre certaines méthodes obligatoirement (e.g. `doGet()`, `doPut()`).
- les JSP doivent étendre la classe `java.servlet.jsp.HttpJspPage`,
- les EJB doivent implémenter les interfaces `javax.ejb.EJBHome` et `javax.ejb.EJBObject` et une des deux `javax.ejb.SessionBean` ou `javax.ejb.EntityBean`.



Architecture J2EE -- Les composants

Les Servlets

Les servlets sont des programmes **côté serveur** embarquant de la logique (du code) dans le processus de réponse à une requête http. Ils permettent d'étendre les fonctionnalités du serveur et d'introduire du contenu dynamique dans HTML. Pour J2EE, il faut les servlets v2.2.

Les JSP (Java Server Pages)

Les JSP permettent d'embarquer (d'inclure) un composant dans une page (comme un programme PHP) et de laisser ce composant générer la page qui sera effectivement renvoyée au client.

C'est en réalité une extension des servlets : en effet, quand il rencontre un composant JSP (en code source Java2), le serveur web **compile cette page JSP en une servlet**, puis invoque la servlet et renvoie le résultat au client web. Le serveur gère un cache des JSP déja compilées et invoque directement les servlets correspondantes.

Les EJB (Enterprise JavaBeans)

Il y en a de plusieurs sortes :

- **les Beans de session avec état** : c'est un objet transitoire qui permet de représenter dans ses variables l'interaction avec un client. Il exécute des requêtes du client en interrogeant d'autres beans ou la base de données. Il est détruit lorsque le client termine sa session.
- **les Beans de session sans état** : ils sont utilisés pour mettre en oeuvre des services ne demandant pas de conserver un état du client,
- **les Beans entité** : c'est un **objet persistant**, qui modélise des données dans ses variables. Il peut être utilisé simultanément par plusieurs clients.

Ch24 - Introduction à J2EE

Introduction à J2EE

Les **EJB (Enterprise Java Beans)** sont une architecture de composants destinée à créer des applications distribuées, multi-tiers et extensibles ("scalables").

De plus, elle rend possible la création de **serveurs d'application extensibles dynamiquement** (on peut ajouter des fonctions sans arrêter le serveur, sans relinker, sans rebooter).

Les EJBs ont été introduits avec la publication de la spécification 1.0 en mars 1998. De nombreuses compagnies proposent des produits conformes aux spécifications EJB : IBM, BEA, Oracle, Tandem,

Sybase, Visigenic, Iona, ...

Historique du Client/Serveur

Le "bon vieux temps"

Au début était le "**mainframe**". L'état de l'art en matière d'informatique dans les années 60 consistait en grosses machines, très chères, utilisées pour les opérations de gestion : paie, comptabilité, facturation, ...

Tout le monde était content : les vendeurs qui gagnaient beaucoup d'argent sans se fatiguer, les acheteurs qui se sentaient important en signant des gros chèques, les gestionnaires grands prêtres et gardiens du temple ou trônait LA machine, tous ... sauf peut-être les utilisateurs.

Les miniordinateurs en temps partagé des années 70 ont rendu les machines plus accessibles, mais les traitements restaient centralisés.

Les premiers micro ordinateurs, dans les années 80, ont transformé le paysage monolithique de l'informatique d'entreprise en des centaines de petits îlots de traitement et de stockage d'informations, devenant très vite incohérents les uns par rapport aux autres.

Le Client/Serveur à la rescousse

L'architecture client/serveur a longtemps été la seule réponse à la double contrainte du besoin de **centraliser le contrôle** des données (pour assurer leur cohérence et leur intégrité) et du besoin de **disséminer l'accès** à ces données.

Mais les nombreux défauts du client/serveur traditionnel ont conduit à le faire évoluer : on est d'abord passé du client/serveur 2 tiers au client-serveur 3 tiers, qui permet de donner de la souplesse au système et facilite l'interopérabilité entre systèmes d'origines différentes.

... et l'objet vint

Puis le client/serveur a évolué vers les techniques de programmation objet.

CORBA est une architecture permettant à des objets qui s'exécutent dans des applications différentes de s'appeler les uns les autres.

Des applications héritées (legacy applications) peuvent être encap-

sulées dans un service CORBA et ainsi, interopérer avec des applications nouvelles.

Les EJBs, qui ont été concus pour être compatibles avec CORBA, sont une autre façon de faire inter-opérer des objets.

Les EJBs et les serveurs d'application extensibles

L'idée de base derrière les EJBs est de fournir un cadre permettant de **brancher des composants dans un serveur existant** de façon à **étendre les fonctionalités du serveur**.

Il faut bien noter que les EJBs ont très peu à voir avec les simples JavaBeans. La spécification des EJBs est**entièrement** différente de celle des JavaBeans.

La spécification des EJBs indique comment les EJBs interagissent avec le serveur et avec les clients.

Elle indique aussi comment les EJBs assurent la compatibilité avec CORBA, et définit précisément les responsabilités de chaque composant du système.

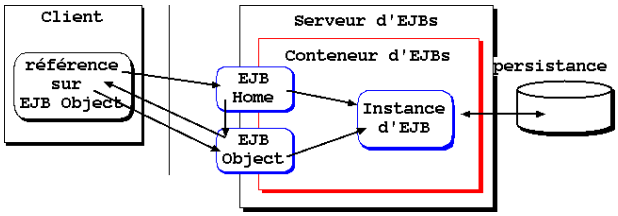
Les objectifs des EJBs

- **rendre facile la création d'applications** par les développeurs.
  Les EJBs libèrent le développeur de la nécessité de gérer dans le détail les threads, les transcactions, la répartition de charge, la persistance, etc ...
  Tout en permettant, s'il en est besoin, d'utiliser des APIs de "bas niveau" pour adapter le comportement par défaut du système pour une application particulière.
- la spécification définit la structure du système de fonctionnement des EJBs ("framework") et définit les**contrats** entre chaque élément de cette structure.
  Créer un EJB est un travail **très** différent de la création d'un serveur d'EJBs.

- de même que les JavaBeans créés par différents outils (du genre Delphi de Borland) peuvent être associés en une seule application, des EJBs créés par des outils de différents vendeurs peuvent **interopérer** et être **combinés** pour produire une application.
  Les EJBs sont un moyen **standard** de créer des applications client/serveur en Java.
- les EJBs sont compatibles avec les autres APIs Java, peuvent interopérer avec des applications non-Java et avec CORBA.

Le fonctionnement d'un système client/serveur à base d'EJBs

Un système client/serveur à base d'EJBs est composé de quatre éléments : le conteneur d'EJBs, le composant EJB, l'objet EJB et l'objet d'accueil EJB Home (**"EJB container"**, **"EJB component", "EJB object" et "EJB Home"**).



Le conteneur d'EJBs ("EJB container")

Le conteneur est le **support d'exécution** du composant.

Il lui fournit des **services** tels que la gestion des transactions, des versions, la mobilité, l'extensibilité, la persistance et la sécurité.

Par exemple, si le composant décide qu'une transaction en cours doit être avortée, il le signale, selon l'API prévue, à son conteneur, et c'est le conteneur qui va se charger d'effectuer tous les "rollbacks" nécessaires.

Typiquement, de multiples instances de composants EJB coexistent dans un seul conteneur d'EJBs.

Le **conteneur est l'endroit ou "vit" le beans**. Il lui fournit un environnement sûr, extensible et transactionnel dans lequel le bean peut

opérer.

Le **conteneur est invisible aux clients** : il n'y a pas d'API des conteneurs pour les clients.

Quand un bean est instancié sur un serveur (i.e. dans un conteneur), le conteneur **fournit deux implémentations** :

- une implémentation de **l'interface EJBHome** (l'interface d'accueil) pour le bean,
- et une implémentation de **l'interface remote** du bean : l'objet EJB ("EJB Object").

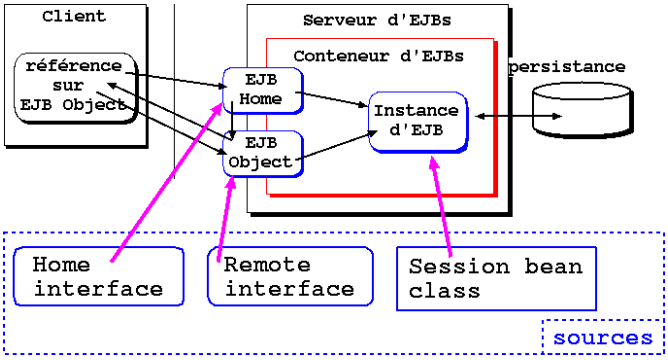
Ces deux implémentations sont construites automatiquement à partir des sources des **interfaces EJBHome et remote** fournies avec le composant.

Le conteneur est aussi responsable d'**enregistrer l'interface EJBHome auprès de JNDI**.

L'**interface remote**, qui joue le rôle du "skeleton" dans CORBA, est générée automatiquement quand le bean est installé sur le serveur.

L'implémentation de l'interface remote est "**l'objet EJB**". Cet objet EJB expose toutes les méthodes publiques du bean.

Le composant EJB lui-même n'implémente pas l'interface remote bien qu'il dispose d'une implémentation de toutes les fonctions présentes dans cette interface, avec les mêmes signatures.



Le composant EJB ("EJB component")

Un composant EJB s'exécute à l'intérieur d'un conteneur d'EJBs qui, lui-même, s'exécute à l'intérieur d'un serveur d'EJBs.

Tout serveur qui peut héberger un conteneur d'EJBs et lui fournir les services définis par la spécification peut devenir un serveur d'EJBs.

Ceci a permis à de nombreux vendeurs d'étendre les fonctionnalités de leurs serveurs pour les transformer en serveurs d'EJBs.

Un **composant EJB** est une classe Java qui implémente la **logique de l'application** (la logique métier ("business logic")).

Elle doit être accompagnée de la définition des interfaces "Home" et "remote" du composant.

Toutes les autres classes du système EJB fournissent des services aux composants ou supportent les accès par les clients.

L'objet EJB ("EJB object")

Les clients exécutent des méthodes des EJBs distants par le biais de l'objet EJB.

L'objet EJB implémente l'interface remote du composant EJB. Cette interface remote, liste les **méthodes de la logique métier** qui sont accessibles aux clients.

Les méthodes du composant font le travail réel, utile, pour lequel le composant a été créé.

Les **composants EJBs** et les **objets EJBs** sont des classes **séparées**, bien que, vues de l'extérieur, elles semblent identiques.

C'est simplement qu'elles implémentent **non pas** la même interface, mais plus subtilement la **même liste de méthodes** (avec les mêmes signatures).

Les objets EJBs font cela en implémentant l'interface distante (remote interface) du composant EJB, alors que le composant EJB se "contente" d'implémenter ces méthodes sans pour cela implémenter (au sens Java, avec le mot clé "implements") l'interface remote.

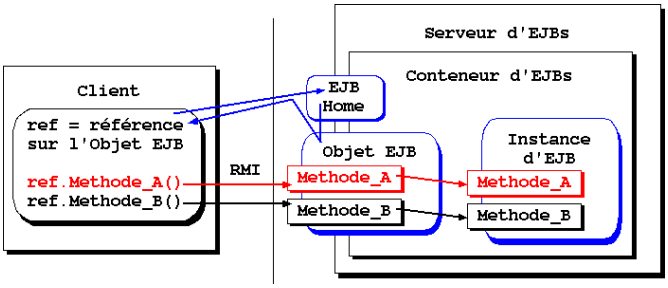
Le composant s'exécute sur le serveur dans le conteneur et implé-

mente la logique métier, alors que l'objet client s'exécute sur le client et fait des appels à distance **vers les méthodes de l'objet EJB** qui **renvoie ces appels** vers les méthodes du composant..

L'objet EJB est **"créé" par le conteneur** lors de l'installation du composant EJB sur le serveur :

Le programmeur de l'EJB n'écrit **que** le source du composant et les deux interfaces (distante et d'accueil).

Le conteneur utilise les informations trouvées dans la description de l'interface et **l'instrospection Java** sur le code du composant EJB pour **créer le code implémentant la classe "EJB object"** qui va servir **d'intermédiaire** entre les clients et le composant EJB.



Le client va récupérer (par l'intermédiaire de l'interface d'accueil) une **référence distante sur l'objet EJB**.

Cette référence lui permettra de faire des **appels RMI** des méthodes disponibles de l'objet référencé.

Les méthodes de l'objet EJB dont la référence RMI a été passée au client ont été construites automatiquement par le conteneur pour être compatibles avec les appels par RMI, et transmettre l'appel à la méthode de même nom du composant EJB. Le client n'a jamais d'accès direct au composant.

Analogie : si on imagine que le composant est un magnétoscope, alors l'objet EJB représente **le récepteur de la télécommande** du composant et la **référence de l'objet EJB** passée au client est la télécommande.

Quand on appuie sur un bouton de la télécommande (quanf on appelle une méthode de la référence distante), un signal est envoyé

au récepteur sur le magnétoscope (un appel RMI est envoyé à l'objet EJB).

Le récepteur actionne le même moteur que le bouton correspondant dans le magnétoscope : le résultat est une **action** dans le magnétoscope (l'objet EJB appelle la méthode de même nom du composant, celle-ci contient le code effectif qui fait l'action).

L'objet d'accueil ("EJB Home")

Son implémentation est construite automatiquement à partir des sources de **l'interfaces EJBHome** fournies avec le composant.

Le conteneur **enregistre l'interface EJBHome auprès de JNDI**.

Ainsi le client aura un moyen standard d'obtenir une référence sur le bean :

```
Context ctx = new InitialContext();
Object beanref = ctx.lookup("nom du bean");
```

Pour pouvoir ensuite invoquer les méthodes métier :

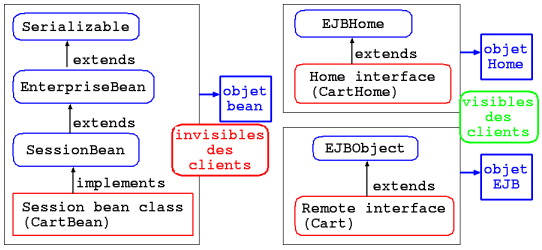
```
beanref.Methode_A()
```


Cycle de vie des EJBs

En plus de l'interface remote, un composant EJB définit une **home interface**, ou **interface d'accueil**, qui permet de retrouver le composant.

L'objet EJB est **généré automatiquement** par les outils de compilation des EJBs, à l'image des fichiers sources générés par les compilateurs IDL de CORBA.

Cet outil de génération est fourni avec le conteneur d'EJB. L'objet EJB joue le rôle du "skeleton" dans CORBA alors que la référence distante fournie au client représente le "stub".



Pour le client, l'objet EJB ressemble exactement à un objet du domaine d'application, mais il s'agit juste d'un "déguisement" ou d'un remplacant de l'EJB réel qui, lui, s'exécute dans le conteneur.

Quand le client appelle à distance une méthode de l'objet EJB,

celle-ci appelle à son tour la même méthode de l'objet "composant EJB", avec les mêmes arguments.

Ceci est le concept central de fonctionnement d'un système client/serveur à base d'EJBs.

L'intérêt de cette étape intermédiaire est de **séparer le comportement des clients** de la **gestion des instances** d'EJBs par le conteneur.

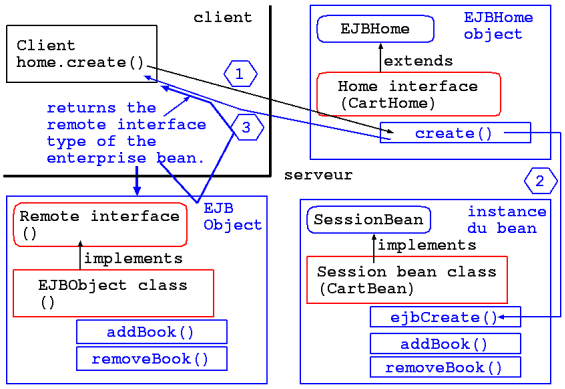
Un EJB est composé d'un fichier "jar" contenant la classe du composant EJB, les 2 fichiers sources décrivant l'interface "remote" et l'interface "Home" et d'un fichier XML appelé **descripteur de déploiement**.

Le descripteur de déploiement décrit le **type de bean** (session, entité, avec état, sans état, persistant ou non) et les **services** qu'il

va utiliser (sécurité, transactions), ainsi que les noms du bean et des classes que le conteneur doit créer pour **"Home", "remote" et "bean"**.

Un programme client contacte un serveur et demande que le serveur crèe un EJB pour effectuer du traitement de données pour le compte du client.

Le serveur répond en créant une instance de composant EJB, et **en renvoyant au client un mandataire ("proxy")** de cet EJB sous la forme d'une **référence distante** vers **l'objet EJB**.



L'objet EJB a la même interface que l'EJB réel, mais ses méthodes ne font que s'exécuter suite à une invocation à distance ("RMI") depuis le client pour **renvoyer l'appel** sur les méthodes correspondantes de l'EJB réel.

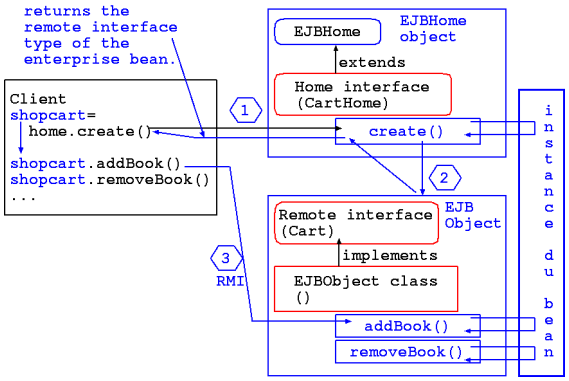
Le client utilise alors "l'objet EJB" comme si c'était l'objet réel, sans avoir besoin de savoir que le vrai travail est effectué à distance dans la mémoire du serveur d'EJBs.

Chaque classe de composant EJB possède une interface d'accueil ("home interface"). Cette interface définit des méthodes pour créer, initialiser, détruire, et (dans le cas des beans entité) trouver des instances de l'EJB sur le serveur.

L'interface "home" d'un EJB est un contrat entre l'EJB et son conteneur. Ce contrat indique comment on construit, détruit et retrouve

l'EJB.

L'interface "**EJB home**" étends (hérite de) **javax.ejb.EJBHome** qui définit les fonctions de base pour une interface "home".



Toutes les méthodes de cette interface doivent être compatibles avec RMI.

L'interface "EJB home" définit aussi une ou plusieurs méthodes "create" (distinguées par leur signature).

La **valeur de retour** de ces méthodes "create" est une **référence sur l'objet** java implémentant l'**interface remote du composant EJB**.

Quand un client veut créer un bean EJB sur le serveur, il appelle d'abord le service JNDI (Java Naming Directory Interface) qui lui permets de localiser une ressource par son nom.

Une fois récupérée le référence sur un objet "Home interface", le client appelle une des méthodes "create" de cet objet pour provoquer la création d'une instance de l'EJB sur le serveur.

La méthode de l'interface home retournée au client fait un appel RMI sur le conteneur de l'EJB, qui, à son tour, crée l'objet instance d'EJB et renvoie au client **une référence sur l'objet EJB**.

Le client peut alors appeler (par RMI) les méthodes de l'objet EJB, chacune d'elle contenant en fait uniquement le code qui "renvoie" l'appel sur la méhode correspondante du composant EJB instancié sur le serveur.

Une interface EJB remote peut définir les méthodes qu'elle veut, avec la seule restriction que tous les arguments d'appel et les valeurs de retour soient **compatibles avec RMI** et que chaque méthode contienne une gestion de l'exception **java.rmi.RemoteException**.

Les deux types d'EJB : beans de session et beans entités

Pour construire un EJB, on commence par implémenter les méthodes "métier" : par exemple pour un bean "compte_bancaire" on créera une méthode "débit", une méthode "crédit" et une méthode "solde".

Il faut aussi implémenter une des deux interfaces **SessionBean** ou **EntityBean**. Ces interfaces gèrent les interactions du bean avec le conteneur et sont invisibles aux clients.

Les beans de session ("session beans")

Un bean de **session** est une instance d'EJB associée à un client **unique**. Les beans de sessions sont généralement non persistants

(bien qu'ils puissent l'être) et peuvent éventuellement participer à des transactions.

En particulier, les beans de session ne survivent pas aux crashs du serveur.

Exemple de bean de session : un bean, qui, dans un serveur web, est associé à un client et conserve la "trace" du chemin suivi par le client à travers le site.

Quand le client quitte le site, ou après un temps d'inactivité, ce bean de session est détruit.

Les beans entités ("entity beans")

Un bean entité **représente de l'information stockée** de façon permanente dans une base de données. Un bean entité représente un accès aux données et donne accès à ces données à de multiples clients.

Les beans entités survivent aux crashs de serveurs, car lorsque le serveur reboote, il peut reconstituer le bean à partir des données stockées de façon permanente.

Dans une base de données relationnelle, un bean entité pourrait représenter une rangée d'une table, dans une base de données objet, il représenterait un objet de la base.

Pour résumer, les **beans de session** représentent les **échanges**

entre le client et le serveur et les **beans entités** les **données** stockées de façon permanente.

Chaque EJB a un identifiant unique : pour un EJB entité, cet identifiant joue le rôle d'une clé primaire dans une base de données. Pour un bean de session, l'identifiant joue le rôle de la référence IOR dans CORBA : une valeur unique permettant de retrouver et utiliser l'EJB.

Les EJBs : conclusion partielle

Le but fondamental d'une technologie de serveur de composants est de répondre à la question : **comment créer un serveur dont les méthodes sont extensibles à l'exécution ?**

Autrement dit : **Comment ajouter au serveur une fonctionnalité**, puis utiliser aussitôt les clients correspondant, sans que l'on ai eu besoin de recompiler le serveur, ni même de le redémarrer ?

La réponse des EJBs est de créer le client en faisant appel à des fonctions définies dans une interface distante, les fonctions de cette interface distante étant implémentées sur le client sous forme d'appels RMI vers un objet du serveur préalablement instancié, qui lui implémente des méthodes de même signature (même nom, mêmes

arguments) qui réalisent le travail efféctif.

Le serveur d'EJBs utilise JNDI pour publier la liste des interfaces disponibles.

Le client utilise **JNDI pour localiser l'interface home**, puis celle-ci pour **créer d'abord** et **ensuite utiliser** le composant EJB.

C'est cette **association tardive** ("late binding") entre un **nom** de composant et la **référence permettant de l'utiliser** qui rend le système EJB extensible à l'exécution, et pas seulement après reboote ou recompilation.

L'architecture des EJBs, en utilisant la plateforme Java, garanti l'indépendance du code écrit dans les méthodes des EJBs et, grâce au respect des spécifications des EJBs, permet d'écrire des composants qui s'exécuteront sur n'importe quel serveur d'EJBs, même de

fournisseurs différents ("write once, run anywhere").

L'une des applications les plus difficiles à écrire et mettre au point sont les applications **transactionnelles et distribuées**. Une fonction clé du système des EJBs est le support des transactions distribuées.

Le système EJB permet d'écrire des applications qui accèdent des bases de données multiples, distribuées sur plusieurs machines et accédées à travers plusieurs serveurs d'EJBs.

Une autre fonction clé du système d'EJB est qu'il est neutre vis-à-vis du protocole de communication entre objets. Ainsi un serveur d'application EJB peut invoquer d'autres objets soit par RMI (pure Java) soit par IIOP (CORBA).

Des conteneurs spécialisés peuvent faire apparaitre des applications existantes non-java comme des Beans, permettant au déve-

loppeur d'EJBs d'accéder à ces applications sans en apprendre leur spécificités .

Un des buts de conception des EJBs a été de permettre une **simplification** de l'écriture d'**applications distribuées**, en transformant des fonctions habituellement codées **manuellement par de simple déclarations** de propriétés des EJBs.

Par exemple, la mise en place de la sécurité ou l'usage des transactions peut se faire simplement par une déclaration dans le composant lui-même.

De plus, les EJBs permettent de développer les deux principaux modèles d'application :

1. **session :** le client débute une session avec un **objet qui réagit comme une application**, exécutant un certain travail pour

le compte du client, travail pouvant inclure une ou plusieurs transactions, et des interrogations de type SQL,

2. **entité :** le client accède un objet du serveur qui **représente une entité de la base de donnée** ; toutes les **modifications sur l'objet sont stockées** de façon permanente dans la base de données ; ce modèle est plus souvent utilisé avec des bases de données objet.

Le processus de développement avec les EJBs

Prenons l'exemple d'un caddie électronique ("shopping cart") pour acheter des objets ou des services sur un site web.

1. On **écrit les méthodes** contenant la "**logique métier**" avec un outil de développement tel que JBuilder, Java Workshop, Café, ...

   Après la compilation, le Bean est **empaqueté** (packaged) dans un fichier de type EJB JAR.

   Ce fichier est un fichier JAR ordinaire avec en plus un **descripteur de déploiement** ("DeploymentDescriptor") sous la forme d'une instance de classe sérialisée qui contient les informations sur la sécurité, le comportement du bean vis-à-vis des ransactions, etc ...

2. On **installe** le Bean sur un serveur en utilisant un **outil de déploiement** fournit par le vendeur.

On commence par écrire l'interface remote du bean :

```
public interface ShoppingCart extends javax.ejb.EJBObject {
  boolean addItem(int itemNumber)
                 throws java.rmi.RemoteException;
  boolean purchase() throws java.rmi.RemoteException;  }
```

Puis on écrit la classe du composant EJB (le bean), avec l'implémentation des méthodes métiers et d'une méthode d'initialisation dont le nom est obligatoirement ejbCreate().

Cette méthode ejbCreate() sera appelée par la méthode create() de l'interface d'accueil. Mais il faudra d'abord obtenir une référence sur un objet implémentant cette interface d'accueil (ici CartHome).


```
public class ShoppingCartEJB implements SessionBean {
   public boolean addItem (int itemNumber) { // code métier }
   public boolean purchase()               { // code métier }
   public ejbCreate (String accountName, String account) {
              // code d'initialisation de l'objet EJB  }   }
```

On remarquera que la classe de l'EJB implémente SessionBean mais **pas** ShoppingCart.

La classe EJB possède toutefois une méthode publique pour chacune de celles déclarées dans l'interface. Elle possède en plus au moins une méthode "ejbCreate".

C'est l'EJBObject, généré par le conteneur lors de l'installation du bean qui implémente l'interface remote ShoppingCart.

Comme les beans de session n'implémentent pas la persistance

automatique, des accès explicites aux bases de données doivent être codés, si besoin, dans les méthodes métier.

Un client commence par chercher l'EJBHome en utilisant JNDI. Il obtient ainsi la racine de la hiérarchie de nommage JNDI.

L'appel à InitialContext() pourra comporter un argument "properties" donnant l'adresse du serveur J2EE et le port sur lequel il écoute.

Ensuite le client appelle la fonction de recherche du bean :

```
Context ctx = new InitialContext();
Object cartref = ctx.lookup("nom du bean");
```

L'interface d'accueil (Home) est définie de la façon suivante :

```
public interface CartHome extends javax.ejb.EJBHome {
   Cart create(int itemNumber)
              throws java.rmi.RemoteException; }
```


Ensuite, il faut transtyper la référence vers l'objet Home à l'aide d'une méthode "narrow". L'utilisation de narrow (qui n'est pas obligatoire si on a un client java communiquant avec le serveur par RMI) permet de rendre le client indépendant du protocole entre lui et le serveur.

```
CartHome home = (CartHome)javax.rmi.
    PortableRemoteObject.narrow(cartref, CartHome.class);
```

Maintenant le client peut alors provoquer la création d'une instance d'objet EJB puis appeler les méthodes déclarées dans l'interface distante :

```
cart = home.create();
cart.addItem( n );  cart.purchase();
```


Questions sur Ch24 - J2EE

➤ Qu'est-ce que J2EE ?
Réponse : diapos 1 et 6.

➤ Faites le schéma de fonctionnement d'un système client/serveur utilisant les EJBs.
Réponse : diapo 9.

➤ Dans un EJB, qu'est-ce qui est visible du client ?
Réponse : diapos 11 et 13.

➤ Dans l'architecture EJB, quel est le rôle de "l'objet EJB" ?
Réponse : diapo 17.

➤ Dans l'architecture EJB, quel est le rôle de "l'objet d'accueil" ("EJB Home") ?

Réponse : diapo 21.

➤ Quels sont les deux types de beans et qu'est-ce qui les distingue ?
Réponse : diapos 32 à 34.

Ch24 - J2EE

Fin du chapitre.

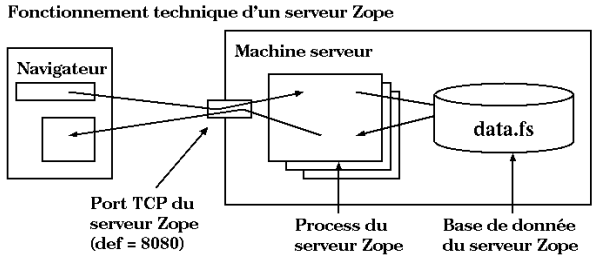
Ch25 - Zope

Zope : qu'est ce que c'est ?

Zope est un serveur d'applications Internet :

- **open-source**,
- **fonctionnel et fiable**,
- utilisable seul comme serveur web ou en collaboration avec Apache,
- utilisé suffisamment : il y a une **communauté active**,
- **maintenu** : développé par Zope Inc. qui l'a mis en open-source et vend du service (création de sites web dynamiques et e-commerce).

Fonctionnement d'un serveur Zope

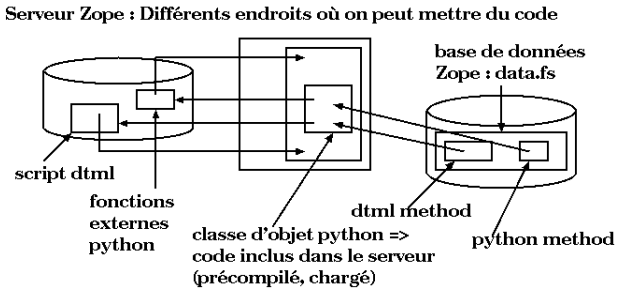


Toutes les requêtes se traduisent par l'exécution de code dans le serveur.

Ce code fabrique "à la volée" la page html renvoyée au client.

Zope utilise deux langages :

- ➤ un langage de script ("**dtml**") : similaire à PHP dans son principe, un script dtml peut être "mélangé" à une page html et être exécuté à la suite d'une requête ; il va de même construire dynamiquement le code html renvoyé au client. Le client n'a aucun moyen de voir le source dtml.
- ➤ le langage natif de Zope : **python**. Il est possible d'ajouter à un site Zope des scripts python, mais aussi de **créer de nouveaux objets Zope** qui seront des instances d'une classe python que l'on va écrire et intégrer dans le serveur Zope.



En quoi Zope est-il un **serveur d'applications** ?

Les process Zope (multithread) conservent le code applicatif en

mémoire : il n'y a pas de surcoût lié à la création d'un process à chaque requête ;

bon, mais le mod-CGI et le mod-PHP de Apache fait la même chose ...

Le code en mémoire peut garder un **état** => plus facile d'implémenter des sessions.

Le concepteur de l'application peut choisir de construire son site avec des pages entièrement statiques ou bien des pages entièrement dynamiques, avec tous les intermédiaires possibles.

Le serveur Zope gère une collection d'objets organisés selon une hiérarchie arborescente comme un ensemble de répertoires et de fichiers .

Parmi ces objets, on trouve :

- des répertoires,
- des scripts dtml,
- des scripts python,
- des objets applicatifs (en python).

Fonctionnement d'un serveur Zope : La hiérarchie d'objets

Faire une requête sur un serveur Zope revient le plus souvent à "appeler" un des objets de la hiérarchie. L'objet va alors exécuter sa méthode "view", ce qui a pour effet de se faire voir (montrer le rendu de l'objet) dans le navigateur du client.

Tout objet a une méthode "view" : s'il n'en définit pas, il hérite de celle de son objet parent.

Plus exactement, plutôt que d'héritage (réservé aux instances d'objets python) ; on parle, sous Zope, **d'acquisition**.

Si un objet ne définit pas la méthode "view", le serveur va remonter la hiérarchie des objets à la recherche d'une telle méthode, et exécuter sur l'objet la première qu'il trouve.

Fonctionnement d'un serveur Zope : L'acquisition

L'acquisition est différente de l'héritage car elle est **dynamique**. Si on place une nouvelle méthode dans la hiérarchie, "plus près" de l'objet invoqué, c'est cette nouvelle méthode qui sera invoquée par le mécanisme d'acquisition, et non pas la méthode de la classe dont hérite l'objet par construction.

==> l'acquisition fonctionne lors de la recherche d'une méthode **dans la hiérarchie des objets et répertoires** et non pas seulement dans la hiérarchie d'héritage.

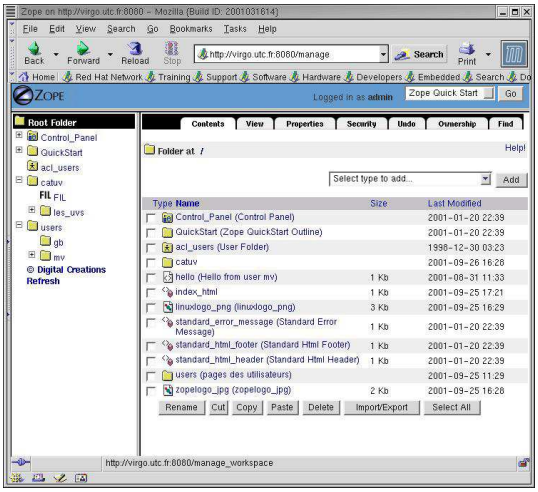
Quand un objet est situé dans un répertoire, les "objets-méthodes" de ce répertoire sont vus comme des méthodes de l'objet. On peut les invoquer, les appeler entre elles et passer des paramètres.

Stockage des données par Zope

Zope peut accéder aux bases de données classiques (MySQL, PostGreSQL, Oracle, Sybase, ..).

Mais il possède aussi sa propre base de données objets : il y stocke des **objets python sérialisés**.

Zope peut aussi accéder au système de fichiers de la machine hôte et **manipuler des fichiers**.



Conclusion sur Zope

Zope est un vrai serveur d'application, comportant des caractéristiques intéressantes :

- puissance fonctionnelle,
- développement rapide (une fois passée la courbe d'apprentissage),
- maintenance facile (le code python est facile à lire),
- s'intègre facilement à un existant,
- gère les protections et la sécurité,
- multi-plateforme (grâce à python).

SR03 Ch25. Fin du chapitre.

(diapo vide)

Ch26 - Introduction à PVM

Introduction à PVM ("Parallel Virtual Machine")

PVM est un **logiciel** (programmes et bibliothèques) permettant d'utiliser un **ensemble de machines** reliées par un **réseau IP** comme des processeurs d'une **machine virtuelle parallèle**, pour des applications de calcul parallèle.

PVM est issu d'un projet du ORNL (Oak Ridge National Laboratory) en 1989. Les sources sont presque depuis le début du domaine public.

PVM est disponible sur pratiquement toutes les machines connues : tous les Unix, Linux, Windows, MAC-OS, Open-VMS, IBM-MVS, ...

Il est très facile à installer, il peut même être installé en mode utilisateur dans le répertoire privé d'un utilisateur.

PVM n'exige pour fonctionner entre deux machines que deux choses :

1. avoir été **installé sur les deux machines**,
2. que l'utilisateur **puisse faire "rsh"** d'une machine à l'autre.

PVM va permettre à l'utilisateur d'écrire des **applications parallèles** en provoquant l'exécution de programmes ("task") sur les machines distantes.

Grâce à l'empaquetage des données échangées par la bibliothèque XDR, PVM peut fonctionner **entre des machines hétérogènes** (constructeur différents, modèles mémoire différents).

Par exemple, on pourra "enroler" dans une machine PVM des stations "sparc", des serveurs IBM Power-PC, des PC Intel sous Linux et d'autres sous Windows, des Macintosh, ... tout cela pour coopérer à une même application.

PVM est interfacé avec les deux principaux langages du monde du calcul scientifique intensif : **C et Fortran**.

Si PVM est installé **sur une machine multiprocesseurs**, il pourra utiliser le **mémoire partagée** pour faire communiquer plus efficacement entre elles les tâches allouées à chacun des processeurs.

Les principes de foncionnement de PVM

Chaque machine physique participant à la "machine PVM" sera identifiée par un numéro (un entier). PVM crée une table de correspondance entre les numéros de processeur de la machine virtuelle et les adresses IP des machines physiques correspondantes.

L'application est découpée (par le programmeur) en **tâches**. Une tâche PVM sera implémentée par un process. PVM va attribuer à chaque tâche un numéro d'identification unique : le **TID**.

Le TID est différent du numéro de processeur, car on peut décider de faire exécuter plusieurs tâches sur le même processeur.

En fait, le TID est composé de la concaténation du N° de processeur et du PID du process implémentant la tâche plus deux bits de

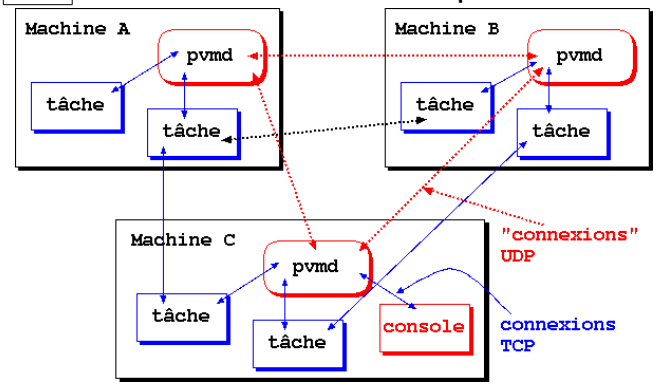
contrôle "S" (démon PVM où tâche) et "G" (Group multicast)

En effet, il est possible de créer des groupes de tâches vers lesquelles on pourra faire du multicast.

Le mécanisme de communication de base de PVM est le passage de messages entre tâches ("message passing").

La bibliothèque PVM implémente des **protocoles spécifiques, au-dessus de TCP et de UDP** pour réaliser au mieux cette communication inter-tâches par messages.

L'implémentation de PVM



PVM est composé de trois parties :

1. le **démon PVM (pvmd)** qui doit être lancé sur chaque machine qui participe à la machine virtuelle ; ces démons écoutent les messages en provenance de la **console PVM** et font le routage des messages entre tâches ; ils coopèrent par l'intermédiaire de sockets UDP ;
2. la bibliothèque pvm (**libpvm**) contient des fonctions pour envoyer et recevoir des messages, ainsi que pour préparer le contenu des messages (appel à XDR) ;
3. une **console PVM**, tâche particulière, résultat de la commande **pvm** qui va servir à gérer le cycle de vie des tâches, des démons pvm et donc de la machine virtuelle PVM.

Configurer l'environnement et la machine virtuelle PVM

Configurer l'environnement

PVM a besoin de variables d'environnement indiquant l'endroit où est installée la bibliothèque.

Comme ces variables d'environnement doivent être définies dans **tous** les process participant à une machine virtuelle, y compris ceux créés à distance par rsh, il est indispensable de mettre cette définition dans un fichier exécuté par le système lors de la création d'un process.

Ce fichier dépend du shell par défaut utilisé. Par exemple pour le shell tcsh ce sera soit **.tcshrc**, soit, s'il est absent **.cshrc** ; pour un shell bash, ce sera **.bashrc**.

On mettra dans ce shell une ligne du type **source ~/set_pvm_env** ou bien **. ~/set_pvm_env**.

Le fichier set_pvm_env contiendra par exemple :

```
# set up env var for pvm
setenv PVM_ROOT /usr/local/pvm/pvm3

if ($?PVM_ROOT) then
        setenv PVM_ARCH `$PVM_ROOT/lib/pvmgetarch`

# uncomment the following line if you want the PVM
# executable directory to be added to your shell path.
       set path=($path $PVM_ROOT/bin/$PVM_ARCH)
       set path=($path $PVM_ROOT/lib/$PVM_ARCH)
```



```
       set path=($path $PVM_ROOT/lib)
       set path=($path $PVM_ROOT/include)

setenv MANPATH `printenv MANPATH`:$PVM_ROOT/man
endif
```

Dans le cas d'un shell bash, les commandes "**setenv VAR value**" sont remplaçées par la syntaxe **export VAR=value** .

Afin d'éviter de permettre l'utilisation de rsh sans avoir à fournir un couple username/password à chaque création de process, on utilise, entre les machines servant de support à la machine virtuelle PVM, la fonction ".rhosts".

Si la machine "bravo" contient dans le fichier **~/.rhosts** de l'utilisateur "sr03nn" les lignes suivantes :


```
echo
golf
kilo
```

alors, on pourra faire un **rsh bravo** depuis une de ces trois machines, **sans avoir à donner de mot de passe**. Un login shell sera créé sur bravo, sous le compte du user "sr03nn".

Configurer la machine virtuelle PVM

La configuration de la machine virtuelle PVM consiste à **indiquer combien de "processeurs virtuels"** on désire utiliser dans notre application, et sur quelles machines physiques vont être créés (sous forme de démons pvmd) ces processeurs virtuels.

Ceci se fait par un fichier "pvm_hostfile" qui a l'aspect suivant :


```
# fichier hostfile de config d'une machine virtuelle PVM
bravo ep=$HOME/pvm
echo  ep=$HOME/pvm
golf  ep=$HOME/pvm
kilo  ep=$HOME/pvm
```

Ici on crèe 4 processeurs virtuels. Dans chaque cas on défini la variable pvm "ep" qui indique **le chemin d'accès aux exécutables PVM**. C'est-à-dire, à quel endroit, le programme pvm lancé depuis une tâche sur un processeur virtuel devra chercher l'exécutable.

Dans l'exemple ci-dessus, tous les chemins sont identiques car on est dans le cas simple ou toutes les machines citées font partie du même ensemble NFS. Et donc, l'utilisateur "sr03nn" a le même répertoire "HOME" sur toutes les machines.

Mais ceci n'est pas obligatoire, on pourrait ajouter à la machine PVM une machine physique située ailleurs (éventuellement, même, sur un autre site géographique) et pour laquelle les noms de répertoires sont différents.

Par exemple l'utilisateur "sr03nn" pourrait installer PVM dans son compte "vega" et utiliser son username "jdupond" sur vega pour lancer une tâche PVM depuis sunserv, en ajoutant à son fichier "pvm_hostfile" une ligne du type :

```
vega.utc.fr lo=jdupond dx=pvm/pvmd ep=$HOME/pvmexe
```

Le fichier "pvm_hostfile" est un fichier de données qui va être passé en paramètre de la commande **pvm** qui lance la console pvm : **> pvm pvm_hostfile**.


```
77 sr03 sunserv:~/pvm> pvm pvm_hostfile
pvm> conf
3 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED
                 sunserv    40000 SUN4SOL2    1000
                   bravo    80000 SUN4SOL2    1000
                    golf   100000 SUN4SOL2    1000
pvm>
```

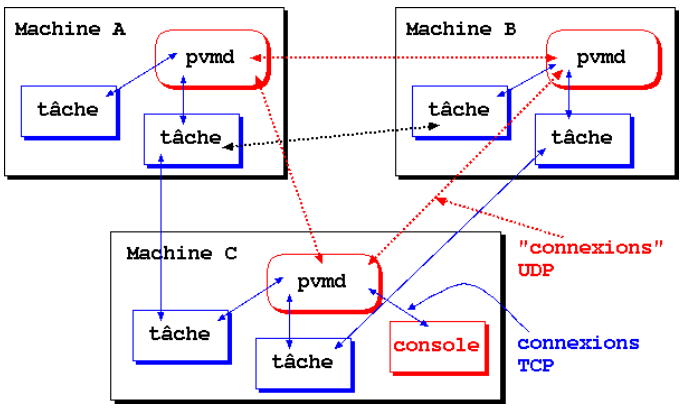
On a maintenant une machine virtuelle constituée de 3 processeurs.

Attention ! Il a deux façons de sortir de la console pvm :

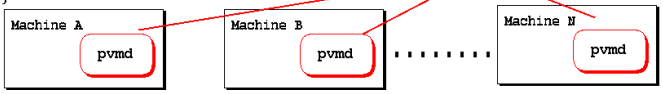
- **pvm> quit** sort de la console pvm **sans arrêter les démons** : la machine virtuelle reste en fonction.
- **pvm> halt** provoque l'arrêt de tous les démons avant de sortir de la console. C'est la commande à utiliser pour "nettoyer" la machine virtuelle.

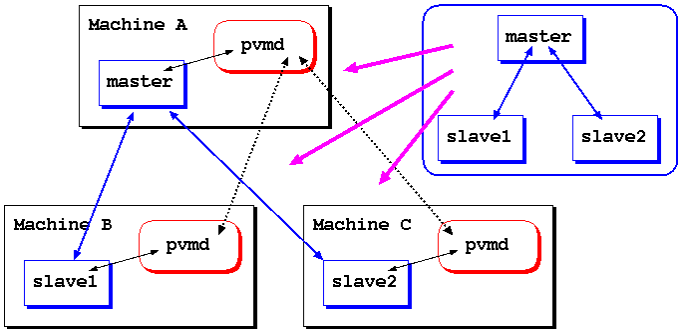
Autre commandes utiles :

- **pvm> ps** : liste des process pvm en cours ;
- **pvm> kill numéro_process_pvm** : tuer un processus pvm ;
- **pvm> pstat numéro_process_pvm** : afficher le status d'un processus pvm ;

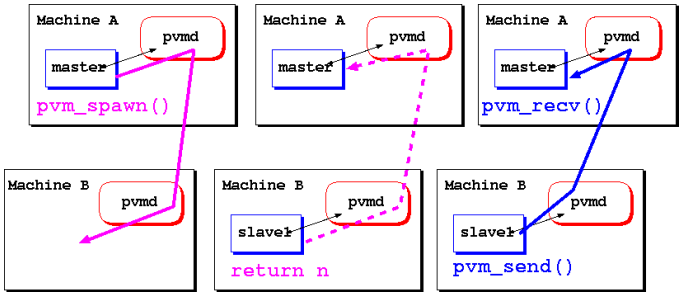


```
/* config.c
cc -O -I/usr/share/pvm3/include  -o config ./config.c
-L/usr/share/pvm3/lib/LINUXI386 -lpvm3
*/
#include <pvm3.h>
#include <stdio.h>
main()
{ struct pvmhostinfo *hostp;
 int i, nhost, narch, info;

 info = pvm_config ( &nhost, &narch, &hostp );

 for ( i=0 ; i < nhost ; i++ )
    printf("machine %d: %s\n", i, hostp[i].hi_name);
}
```





PVM : "master-slave" paradigm
+ projection tâches sur machine virtuelle




```
#include <stdio.h>
#include "pvm3.h"

int hello_you (char *machine, int flag, int ntask)
{
 int cc, tid;
 char buf[100];

 cc = pvm_spawn("./pvm/hello_other",
                (char**)0, flag, machine, ntask, &tid);
 if (cc == 1) {
   cc = pvm_recv(-1, -1);
   pvm_bufinfo(cc, (int*)0, (int*)0, &tid);
   pvm_upkstr(buf);
   printf("from t%x: %s\n", tid, buf);

 } else
   printf("can't start hello_other on %s\n",machine);
 return cc;
}
main()
{ int cc;
  printf("i'm t%x\n", pvm_mytid());

  cc = hello_you("", 0, 1);
  cc = hello_you("smesta05a.sme.utc", 1, 1);
  cc = hello_you("smesta05b.sme.utc", 1, 1);
  pvm_exit(); exit(0);
}
```

hello.c → hello_other.c

Nbre de tâches lancées

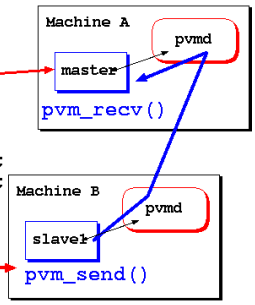
bloquant

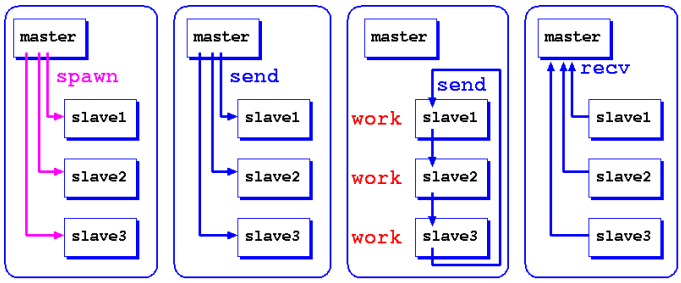
Nbre de copies à lancer

flag=0 : PVM choisit la machine d'exécution
flag=1 : PVM exécute sur la machine demandée


```
#include "pvm3.h"

main()
{
 int ptid;
 char buf[100];

 ptid = pvm_parent();

 strcpy(buf,"hello, world from ");
 gethostname(buf+strlen(buf), 64);

 pvm_initsend(PvmDataDefault);
 pvm_pkstr(buf);
 pvm_send(ptid, 1);

 pvm_exit(); exit(0);
}
```

hello_other.c

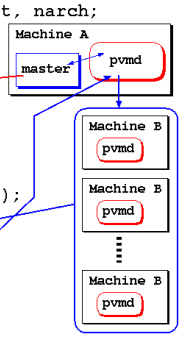





```
#include <stdio.h>
#include "pvm3.h"
#define SLAVENAME "slave1"

main()
{    int mytid;        /* my task id */
     int tids[32];     /* slave task ids */
     int n, nproc, numt, i, who, msgtype, nhost, narch;
     float data[100], result[32];
     struct pvmhostinfo *hostp[32];

     /* enroll in pvm */
     mytid = pvm_mytid();

     /* Set number of slaves to start */
     /* Can not do stdin from spawned task */
     if( pvm_parent() == PvmNoParent ){
        puts("How many slave programs (1-32)?");
        scanf("%d", &nproc);
     }
     else{
        pvm_config( &nhost, &narch, hostp );
        nproc = nhost;
        if( nproc > 32 ) nproc = 32 ;
     }
```

master1.c

1/3



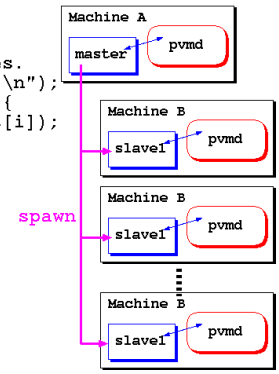

```
/* start up slave tasks */
numt=pvm_spawn(SLAVENAME, (char**)0,0,"",nproc,tids);

 if( numt < nproc ){
    printf("Trouble spawning slaves.
        Aborting. Error codes are:\n");
    for( i=numt ; i<nproc ; i++ ) {
       printf("TID %d %d\n",i,tids[i]);
    }
    for( i=0 ; i<numt ; i++ ){
        pvm_kill( tids[i] );
    }
    pvm_exit(); exit(1);
 }
```

demandé

obtenu

master1.c

2/3



master1.c

3/3

```
    /* Begin User Program */
    n = 100;
    /* initialize_data( data, n ); */
    for( i=0 ; i<n ; i++ ){
         data[i] = 1;       }

    /* Broadcast initial data to slave tasks */
      pvm_initsend(PvmDataDefault);
      pvm_pkint(&nproc, 1, 1);
      pvm_pkint(tids, nproc, 1);
      pvm_pkint(&n, 1, 1);
      pvm_pkfloat(data, n, 1);
    pvm_mcast(tids, nproc, 0);

    /* Wait for results from slaves */
    msgtype = 5;
    for( i=0 ; i<nproc ; i++ ){
       pvm_recv( -1, msgtype );
       pvm_upkint( &who, 1, 1 );
       pvm_upkfloat( &result[who], 1, 1 );
       printf("I got %f from %d\n",result[who],who);
    }
    /* Program Finished exit PVM before stopping */
    pvm_exit();
}
```

empaqueter

dépaqueter

slave1.c

1/2

```
main()
{   int mytid;        /* my task id */
    int tids[32];     /* task ids   */
    int n, me, i, nproc, master, msgtype;
    float data[100], result;
    float work();
    mytid = pvm_mytid();  /* enroll in pvm */
    msgtype = 0; /* Receive data from master */
    pvm_recv( -1, msgtype );
       pvm_upkint(&nproc, 1, 1);
       pvm_upkint(tids, nproc, 1);
       pvm_upkint(&n, 1, 1);
       pvm_upkfloat(data, n, 1);
    /* Determine which slave I am (0 -- nproc-1) */
    for( i=0; i<nproc ; i++ )
       if( mytid == tids[i] ){ me = i; break; }
    /* Do calculations with data */
    result = work( me, n, data, tids, nproc );
    /* Send result to master */
    pvm_initsend( PvmDataDefault );
       pvm_pkint( &me, 1, 1 );
       pvm_pkfloat( &result, 1, 1 );
       msgtype = 5;
    master = pvm_parent();
    pvm_send( master, msgtype );
    pvm_exit(); /* Exit PVM before stopping */
}
```

dépaqueter

empaqueter

slave1.c

2/2

```
float work(me, n, data, tids, nproc )
    /* Simple example: slaves exchange
    data with right neighbor (wrapping)*/
    int me, n, *tids, nproc;
    float *data;
{
    int i, dest;
    float psum = 0.0, sum = 0.0;
    for( i=0 ; i<n ; i++ ){
       sum += me * data[i];
    }
    /* illustrate node-to-node
       communication */
    pvm_initsend( PvmDataDefault );
       pvm_pkfloat( &sum, 1, 1 );
       dest = me+1;
       if( dest == nproc ) dest = 0;
    pvm_send( tids[dest], 22 );
    pvm_recv( -1, 22 );  bloquant
       pvm_upkfloat( &psum, 1, 1 );

    return( sum+psum );
}
```

```
1  slave1(){
     send(me+1,ty)
     recv( -1, ty)

2  slave1(){
     send(me+1,ty)
     recv( -1, ty)

3  slave1(){
     send(me+1,ty)
     recv( -1, ty)
```


Questions sur Ch26 - PVM

➤ Qu'est-ce que PVM ?

Réponse : diapo 1.

➤ Faites un schéma du principe de fonctionnement de PVM.

Réponse : diapo 6 ou 16.

➤ PVM peut-il utiliser pour un même calcul des machines de type différent (par exemple des machines Intel , SUN-Sparc et IBM-PowerPC mélangées) ?

Réponse : Oui.

➤ Comment sont lancées les différentes tâches d'une application PVM ?

Réponse : diapos 18 à 20.

➤ Dans une application PVM peut-on exécuter des programmes différents dans chacune des tâches virtuelles ?

Réponse : diapo 20. Oui : il sufft de changer l'argument du pvm_spawn().

➤ Quand deux tâches PVM s'échangent un paquet de données que doivent-elles faire à l'envoi et à la réception ?

Réponse : diapo 26. Empaqueter et dépaqueter en utilisanet les fonctions PVM ad-hoc qui tiennent compte du problème big/little endian.

Ch26 - PVM

Fin du chapitre.

(diapo vide)

Ch27 - Portails

- Qu'est-ce qu'est (ou n'est pas) un portail ?
- Fonctions d'un portail.
- Implications du portail
- Portail et contrôle d'accès.
- Avantages d'un portail
- Exemples

Qu'est-ce qu'est (ou n'est pas) un portail ?

Il convient de distinguer :

- **un site web institutionnel** : accessible de partout sans identification (consultation anonyme) et sans authentification (consultation non contrôlée).
- **un portail** : accessible aux seules personnes **enregistrées** moyennant une identification et une authentification.

Fonctions d'un portail.

C'est-à-dire que :

- le site web institutionnel est destiné à informer le public,
- le portail est destiné à fournir au personnel (et pour une université aussi aux étudiants inscrits) un "environnement numérique de travail" **mobile**.

C'est-à-dire pouvoir **faire en déplacement**, à l'aide d'un simple navigateur internet, les **mêmes choses** que ce que l'on peut faire **à l'intérieur** des locaux.

Donc le site institutionnel continue d'exister à côté du portail.

Implications du portail.

Faire "ce que l'on peut faire depuis l'intérieur" implique d'avoir **accès à des applications**.

Mais ces applications ne doivent être accessibles qu'aux personnes **autorisées**.

Or en "passant par le web", on élimine la méthode traditionnelle de contrôle d'accès aux applications qui est de vérifier d'abord le contrôle d'accès à la machine qui supporte l'application (le couple infernal "username / password").

Il s'ensuit que l'application doit faire elle-même le contrôle d'accès.

Portail et contrôle d'accès.

Il serait bien entendu **ingérable** d'ajouter à chaque application une gestion d'une liste de couples "username / password".

Les applications vont donc **déléguer le contrôle d'accès**.

Cette délégation est faite à un **service d'authentification centralisé** ("CAS" : Central Authentication Service), qui lui-même aura recours à un **annuaire centralisé**.

Le "CAS" va servir de **mandataire** pour effectuer l'authentification.

Conséquence intéressante du "CAS" commun

Si toutes les applications utilisent le même mandataire, alors, l'authentification peut être faite **une seule fois** en permettant l'accès à **plusieurs** applications.

C'est ce qu'on appelle le **"SSO" : Single Sign On**.

Quand on a "badgé" à l'entrée, on a accès à toutes les pièces.

Techniquement, le CAS, après vérification de la paire "username / password", qui sera donc **unique**, va produire un **certificat** crypté qui sera déposé sur le poste de travail de l'utilisateur sous la forme d'un **cookie**.

Ensuite, pour chaque accès nouveau, le certificat sera réclamé par le portail et fournit par le navigateur **sans intervention de l'utilisateur**.

À la fin de la session de travail, l'utilisateur **doit**, soit faire **logout**, soit **fermer le navigateur** de façon à **détruire le certificat**.

Comme on ne peut jamais être certain que ceci sera toujours fait, le certificat possède une **date de péremption**.

Conséquence du "CAS" commun sur les applications

Quand un utilisateur authentifié appelle une nouvelle application, celle-ci va récupérer le certificat grâce au "cookie" déposé sur le poste de travail et présenter ce certificat au mandataire (au "CAS"), qui en vérifie la validité et la date et l'heure de péremption.

- ceci implique que les applications accessibles incluent la méthode d'authentification par mandataire .

Pour les applications qui utilisent les méthodes standard des systèmes d'exploitation, ceci peut être obtenu sans modifier l'application, mais simplement en ajoutant, au niveau du système, l'appel au mandataire. Sinon il est nécessaire de modifier l'application pour lui ajouter l'authentification par mandataire.

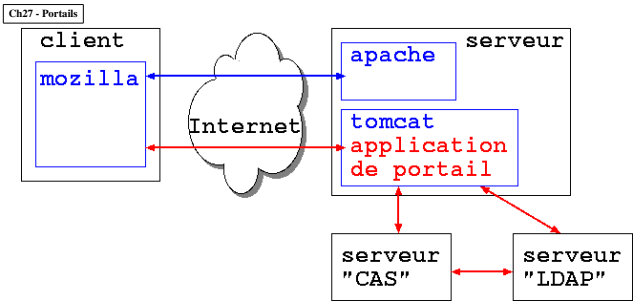
Environnement d'infrastructure du Portail.

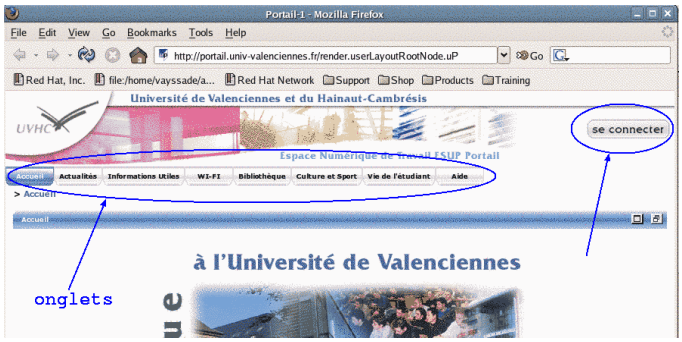
L'objectif d'un portail est d'offrir un espace numérique de travail qui donne un accès à un certain nombre de services.

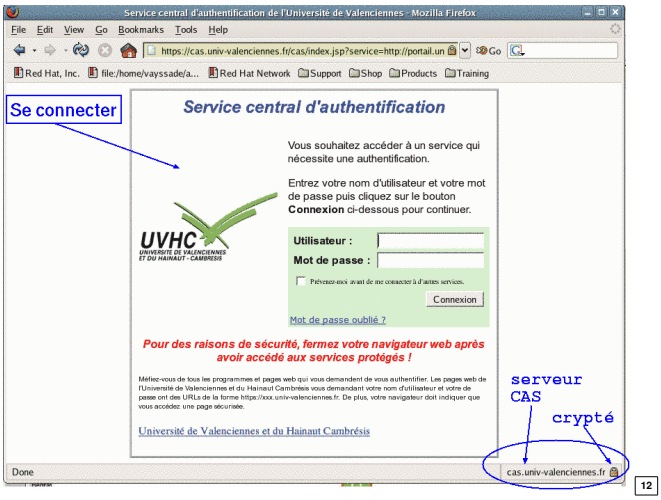
Par exemple : annuaire, courrier électronique, emplois du temps, forums, espaces disques partagés, agenda partagé, application de saisie de bons de commandes, d'ordres de missions, etc ...

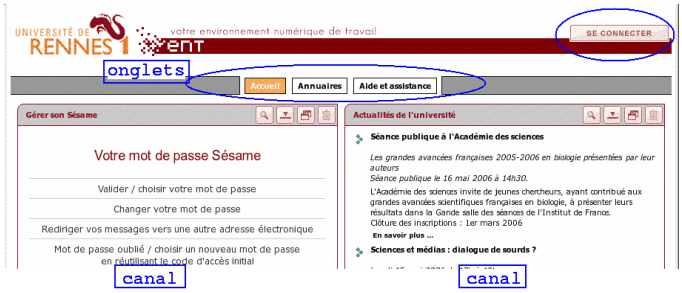
Il est souhaitable qu'un portail se présente sous forme **d'écrans personnalisés** en fonction du profil de l'utilisateur. Mais aussi que celui-ci puisse, dans une certaine mesure, modifier l'apparence des écrans en fonction de ses préférences.

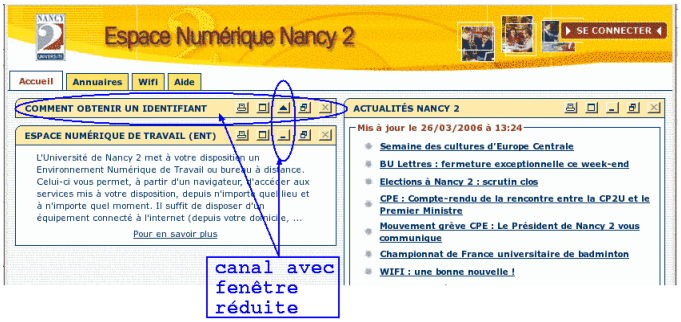
Par exemple, un utilisateur ajoutera sur sa page le **canal** "informations sportives", un autre le canal "évènements musiquaux", ...











Ch27 - Portails

Fin du chapitre.

(diapo vide)

Ch28 - Applications de sécurité

- ➤ PKI et cryptage à clés publiques
- ➤ ssh : "secure sh"
- ➤ VPN : Virtual Private Network (aka tunnel)
- ➤ SSO : Single Sign On
- ➤ LDAP : Annuaire pour TCP/IP

Qu'est-ce qu'une PKI ?

PKI = Public Key Infrastructure

Cryptage symétrique

Un texte en clair ("plaintext") (ou n'importe quelle information numérique : image, fichier de data, ...) peut être transformé en un texte encrypté ("cyphertext") par un algorithme utilisant un nombre appélé **clé** ;

Si l'algorithme et la clé sont bien choisies (surtout si la clé est assez longue) il sera **en pratique impossible** de décrypter le résultat du cryptage autrement qu'en utilisant la clé.

Remarque : le mot important ici est "en pratique".

Si une information a une durée de vie de 24 heures, un cryptage qui

ne peut être "cassé" qu'en 3 jours de calculs sera considéré comme sûr.

Si le **même** nombre est utilisé pour crypter et décrypter, le cryptage est dit **symétrique**.

Il est assez facile dans les cryptages symétriques d'utiliser de très longues clés car ceci impacte peu le temps de calcul. Si la clé est assez longue le temps de calcul pour décrypter sans la clé même sur les machines les plus rapides peut

Un nombre de 9 chiffres (1 milliards de clé possibles) est équivalent à une clé binaire de 30 bits.

Une clé de 128 bits donne environ 3,4 10^{38} clés possibles.

Si toutes ces clés étaient écrites sur des feuilles de papier, la pile

de papier aurait une hauteur à peu près 1 million de fois plus grande que la taille estimée de l'univers.

Malgré cela, le calcul du cryptage avec une clé de 128 bits est relativement rapide : ainsi PGP qui utilise un cryptage symétrique à clé de 128 bits peut crypter 1 mégaoctet en 10 secondes sur un PC 486 à 100Mhz (et décrypter en 12 sec.).

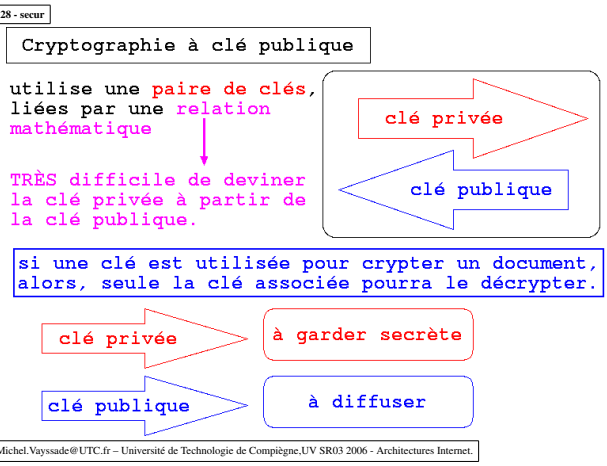
Cryptage asymétrique (à clé publique)

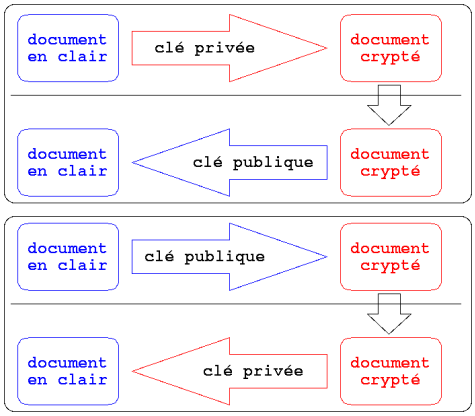
Jusqu'en 1976 le cryptage symétrique était le seul connu.

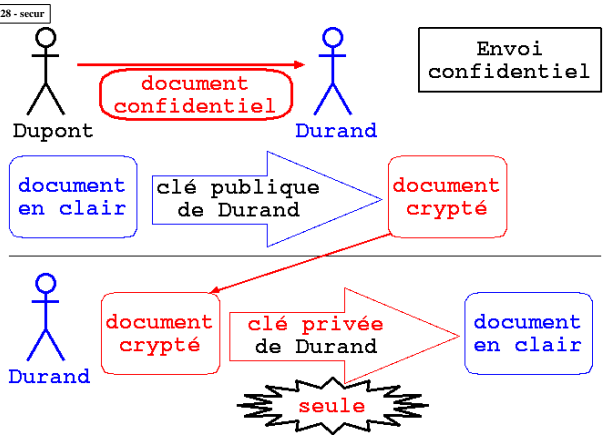
La conséquence est que pour échanger des messages cryptés, les deux protagonistes devaient avoir chacun une copie de la clé.

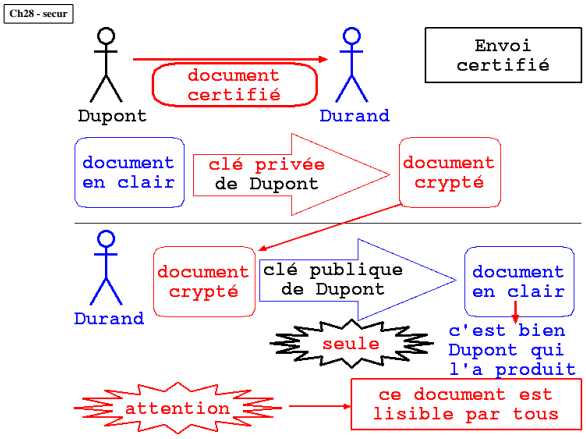
En 1976 Whitfield Diffie a développé une nouvelle approche appélée **Cryptage à clé publique**.

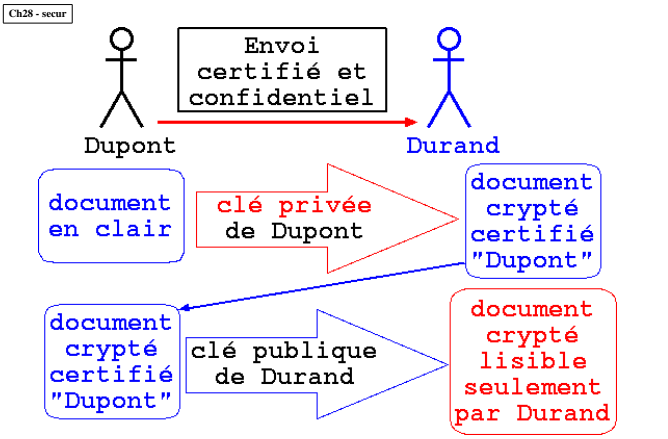
Chaque personne génère **deux clés** : une **clé privée à garder secrète** et une **clé publique à diffuser**.











En pratique

Le chiffrement à clé publique est très consommateur de ressources (calcul).

On crypte le document avec un algorithme de chiffrement symétrique (calcul rapide). La clé de chiffrement symétrique étant générée aléatoirement.

Seule la clé de chiffrement symétrique est chiffrée avec la clé publique.

La clé publique change tout

Avant, il fallait un échange de clé **préalable** à une communication. Cet clé de départ devait être échangée par un moyen **autre que le réseau** (problème de poule et de l'oeuf !).

Après, il sufft de publier les clés puliques sur un site web. Après cette publication il est possible de faire des échanges sécurisés et/ou authentifés **même entre des personnes qui n'ont eu aucun échange préalable**.

Système de cryptage : attention

La sécurité doit reposer sur le secret de la clé, jamais sur le secret de l'algorithme.

Si quelqu'un tente de vous vendre un système de cryptage dont l'algorithme est gardé secret, refusez !

Même chose si l'algorithme n'a pas été sujet à plusieurs années d'examen et de tests de craquage de la part de la communauté scientifique du cryptage.

La sécurité d'un système de cryptage ne doit reposer que sur deux éléments : le secret de la clé et la difficulté à trouver la clé secrète à partir de la clé publique.

Comment les clés sont-elles construites ?

Dans les systèmes à clé public actuels les clés sont en général constituées d'une paire telle que chaque clé soit construite à partir du produit de deux nombres premiers.

Or entre 2 et 2^{1024}, la quantité de nombres premiers est bien sûr très inférieure à 2^{1024} (i.e. les nombres qui sont premiers sont plutôt "rares").

Il est donc plus facile de les essayer tous. C'est pourquoi les systèmes à clé publique utilisent des clés de grande longueur.

Une clé de 1024 bits telle que celles utilisées par PGP est équivalente à une clé symétrique de 96 bits.

Exemple de RSA

En résumé, la méthode est la suivante :

1. choisir deux nombres premiers **p** et **q** très grands et calculer **n = p * q**.
2. calculer **z = (p-1)(q-1)** et choisir un nombre entier **e** premier avec **z**.
3. choisir un nombre entier **d** tel que **e * d = 1(mod z)** (d est l'inverse de e dans l'arithmétique mod z).
4. détruire p et q : la sécurité de l'algorithme repose sur la difficulté de décomposer **n** en ses facteurs premiers.
5. la clé publique est le couple d'entiers **(e, n)**.
6. la clé privée est le couple d'entiers **(d, n)**.

L'encodage se fait par élévation à la puissance "e mod n". On applique ce calcul à des blocs de chiffres d'amplitude inférieure à n pour faire des calculs modulo n. Donc on traite le message par blocs successifs.

Le décodage se fait au moyen de l'élévation à la puissance "d mod n".

Ce calcul d'élévation à une puissance est très long. C'est pourquoi en pratique on ne l'utilise pas pour coder le message complet. On utilise l'algorithme à clé publique **pour coder une clé symétrique** assez longue, générée de façon aléatoire et **qui ne servira qu'une fois**.

Le fait que la clé symétrique ne serve qu'une fois rend inopérentes les méthodes d'attaque basées sur la connaissance de textes cryptés et de leur équivalent en clair.

Le nombre "RSA-155" utilisant une clé de 512 bits a été cassé en 1999 après 8000 années-MIPS de calcul plus 224 heures de CRAY-C916.

Le nombre "RSA-174" (nombre à 174 chiffres) a été cassé en 2003.

Le nombre "RSA-200" a été cassé en 2005. Il a fallut près d'une année de recherche suivie de 3 mois de calculs sur un cluster de 80 Opterons à 2.2Ghz.

"RSA-640" (640 binary digits soit 193 decimal digits) a été cassé par la même équipe en novembre 2005.

Il y a une prime de 100,000$ pour qui factorisera RSA-1024.

Nota : il existe des puces informatiques à 10$ capables de tester 200 millions de clés DES par seconde.

Signature numérique

Pour authentifier un message (le signer ou le sceller) on a dit plus haut qu'il suffisait de le crypter avec sa propre clé privée.

En pratique on évite car le calcul est trop long. Sur le même principe on a développé des systèmes de signature numérique qui utilisent la clé privée et un **algorithme de hachage**.

C'est le cas par exemple du système **MD5**.

- La signature ne peut être imitée. Elle prouve que le document a bien été signée avec telle clé privée.
- Elle appartient à un seul document.
- Le document ne peut pas être modifié sans rendre la vérification fausee.
- La signature ne peut être reniée (aucune autre clé ne pourrait la produire).

ssh

ssh remplace rlogin et rsh pour se connecter et exécuter des commandes sur une machine distante.

ssh peut aussi renvoyer des connexions X11 ou n'importe quels ports TCP/IP à travers le canal sécurisé.

`ssh [options] [user@]hostname [command]`

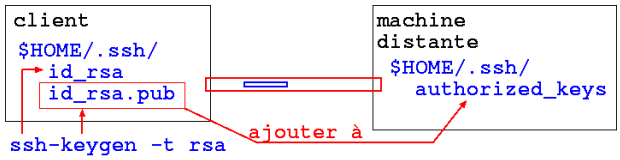
Exemple : `ssh sr03@tuxa.sme.utc`.
ou `ssh -X sr03@tuxa.sme.utc` pour forcer le renvoi de X11.

ssh et authentification RSA

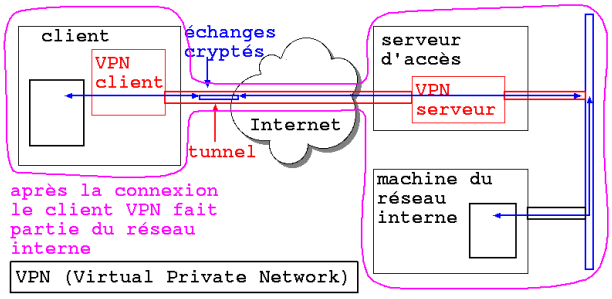
Chaque utilisateur crée une paire de clés publique / privée. Il donne la clé publique au serveur et conserve la clé privée.

Le fichier $HOME/.ssh/authorized_keys contient la liste des clés publiques autorisées à se connecter.

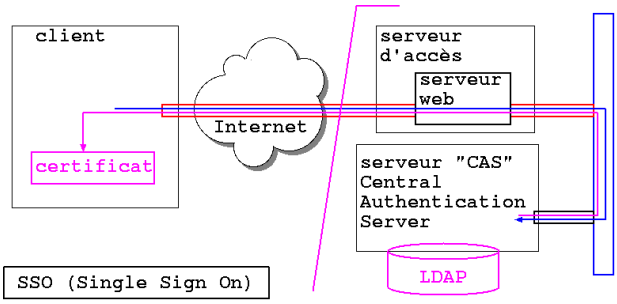
Quand un utilisateur se connecte, le serveur et le client ssh vérifient l'autorisation par un échange de type challenge / réponse.

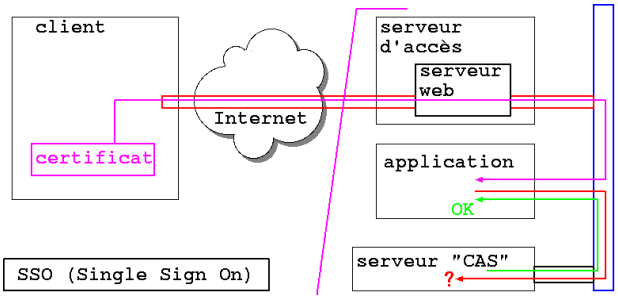


VPN



SSO





(diapo vide)

LDAP

LDAP : Lightweight Directory Access Protocol : un protocole d'annuaire sur TCP/IP.

Annuaire : une base de données d'informations accessible sur le réseau. C'est une base spécialisée dont l'objectif est de **répondre très vite à des interrogations**.

➤ contrairement à un SGBD elle sera très performante en lecture mais peu en écriture.

➤ Les serveurs LDAP sont conçus pour stocker une **grande diversité** de données (beaucoup d'entrées), mais **chacune de faible volume**, ce qui permet de les organiser pour un accès très rapide en lecture.

➤ En savoir plus : **http ://www.cru.fr/ldap/**

La fonction d'un annuaire LDAP est de **centraliser des informations** en les rendant accessibles **rapidement** à des **applications**.

Par exemple l'accès sera suffisamment rapide pour que l'on puisse remplacer la vérification "username / password" dans le fichier /etc/passwd ou /etc/shadow par une interrogation LDAP, sans que l'utlisateur observe un ralentissement de la procédure de login.

LDAP est l'adaptation de la norme DAP d'accès aux annuaires X500 de l'OSI au protocole TCP/IP.

Un annuaire LDAP stocke des objets avec des **attributs normalisés** et permet des interrogations multicritères performantes.

LDAP définit comment l'information contenue est organisée et référencée et quels sont les types de données reconnus.

Le protocole décrit a communication client-serveur : les commandes pour se **connecter ou se déconnecter**, et pour utiliser les entrées de la base : **rechercher, comparer, créer, modifier ou effacer**.

Le protocole définit également des communications **serveur-serveur** qui permettent de **synchroniser** des serveurs (replication) ou de construire une **hiérarchie** (referral).

LDAP utilise sa propre base de données au format **dbm** (Berkeley).

Le DIT : Directory Information Tree

Les données LDAP forment un **arbre** similaire à un système de fichiers unix. Chaque noeud de l'arbre est une entrée de l'annuaire : une **DSE** ("directory service entry"). Au sommet se trouve la racine appelée aussi **suffixe**.

Ce modèle est copié du modèle X500, mais simplifié car son espace de nommage est **local** alors que celui de X500 se veut **mondial** (comme celui du DNS).

Une entrée peut être un **objet abstrait** (un username, un groupe) ou un **objet physique** (une personne, une imprimante, ...).

Une entrée contient des champs de valeurs appelés **attributs**.

Chaque serveur possède une entrée spéciale **rootDSE** (root directory specific entry) qui contient la description de l'arbre et des contenus.

L'ensemble de ces descriptions s'appelle le **schéma** : il décrit les classes d'objets et leurs attributs.

Une entrée de l'annuaire contient une suite de couples "**attribut=valeur**".

Par exemple : `uid=vayssade,ou=si,dc=utc,dc=fr`.

Aujourd'hui (2006) il est fortement conseillé de choisir comme suffixe (racine de l'arbre) un attribut relié au nom de domaine DNS. Ceci permet de trouver automatiquement le serveur LDAP d'un domaine DNS (entrée SRV du DNS).

Par exmple on choisira **dc=utc,dc=fr** et dans le serveur DNS du domaine **utc.fr** on mettra une entrée indiquant quelle machine contient le serveur LDAP.

Nota : **dc = Domain Component**.

Les objets que peut contenir un annuaire LDAP sont normalisés et référencés par un **OID** (object identifier directory) dont la liste est tenue à jour par l'**IANA** ("Internet Assigned Numbers Authority"). Rappel : l'IANA est l'organisation qui gère aussi la liste des "ports bien connus" UDP et TCP.

Un OID est une séquence de nombres entiers séparés par des points. Ils sont alloués de manière hiérarchique, ce qui permet de donner des délégations d'autorité.

Chaque entrée de l'arbre est référencée de façon unique par le **chemin** qui conduit de la racine à cette entrée. Ce chemin complet s'appelle le **DN** (Distinguished Name). On peut aussi utiliser des chemins relatifs **RDN**.

Questions sur Ch28 - Sécur.

➤ Qu'est-ce système de cryptographie à clé publique ?
Réponse : diapo 6.

➤ Quelles sont les deux applications d'un système de cryptographie à clé publique ?
Réponse : diapos 8 et 9. Envoi certifié et envoi confidentiel.

➤ Qu'est-ce qu'un VPN ? Réponse : diapo 21.

➤ Qu'est-ce qu'un SSO ? Réponse : diapo 22.

➤ Qu'est-ce que LDAP ? Réponse : diapo 25.

Ch28 - Sécur

Fin du chapitre.