

Protocoles de communication

=> Ingénierie du logiciel dans les réseaux informatiques

- autopsie d'un accident...
- structure des protocoles de communication
- règles d'ingénierie pour la conception rigoureuse de protocoles

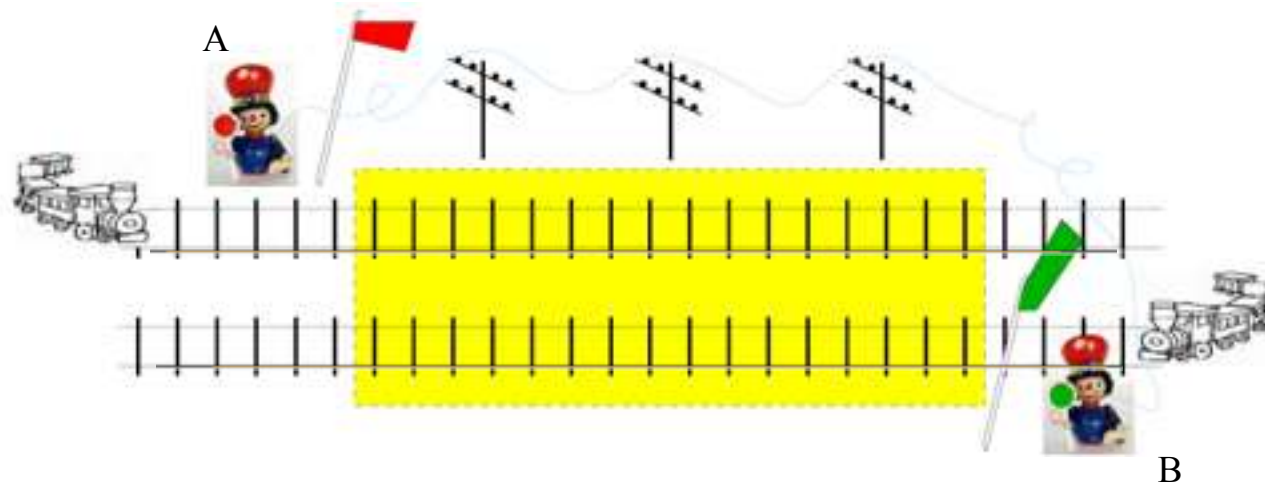
Protocoles de communication...

Definition et caractéristiques

- Un protocole est un ensemble de règles qui gouvernent l'interaction entre processus parallèles dans les systèmes distribués
 - La conception de protocoles n'est pas une discipline à part, elle est liée à la fois aux réseaux et à l'ingénierie des systèmes
 - Décrire complètement et sans ambiguïté un protocole est difficile
 - Prouver qu'un protocole est correct est une tâche encore plus ardue
- => Le but du cours n'est pas d'enseigner de nouveaux protocoles, mais de montrer comment on peut (bien) les décrire

Autopsie d'un accident...

Accident du tunnel de Clayton, 1861



- un train peut entrer dans le tunnel si le sémaphore est vert
- en passant le sémaphore, le train le met au rouge automatiquement ; en cas de défaillance du système, c'est l'opérateur qui agite un drapeau rouge
- c'est l'opérateur qui remet le sémaphore au vert quand il est sûr que le train a quitter le tunnel
- deux télégraphes permettent aux opérateurs de s'échanger quelques messages
 - « *Train in tunnel* »
 - « *Tunnel is clear* »
- Pour plus de sécurité, un 3^e message est prévu : "*Has the train left the tunnel?*"

Autopsie d'un accident...

Accident du tunnel de Clayton, 1861 : ce qui s'est passé

- Un train passe le sémaphore A (vert), mais le sémaphore ne passe pas au rouge.
- L'opérateur réagit : il envoie le message "*Train in tunnel*" et agite un drapeau rouge pour arrêter les trains suivants.
- Cependant, un 2^e train est arrivé trop vite et est passé au vert.
- Heureusement, le conducteur a entrevu le drapeau rouge in extremis.
- Un 3^e train arrive et s'arrête
- L'opérateur envoie à nouveau le signal "*Train in tunnel*" pour avertir l'autre qu'il y a 2 trains dans le tunnel.
- Le protocole n'a pas prévu ce cas.
- Le problème pour l'opérateur A est maintenant de savoir quand les deux trains ont quitté le tunnel afin d'autoriser le 3^e à entrer.
- Pour alerter son collègue, l'opérateur A envoie le signal "*Has the train left the tunnel?*"
- Lorsque le premier train sort du tunnel, l'opérateur B répond "*Tunnel is clear*"
- Ne sachant pas si les 2 trains ont quitté le tunnel, l'opérateur A attend un certain temps, puis autorise le 3^e train à entrer.
- Malheureusement, le conducteur du 2^e train, qui avait vu le drapeau, s'était arrêté dans le tunnel. Après un certain temps de réflexion, il a même fait marche arrière...

=> **23 morts et 176 blessés**

Autopsie d'un accident...

Accident du tunnel de Clayton, 1861 : conclusion

- Il est difficile d'établir la responsabilité de cet accident.
- A partir du moment où le 2^e train est entré dans le tunnel, il n'y avait plus moyen de récupérer la situation.
- L'ensemble des messages disponibles sur les télégraphes était incomplet.
- Le bon sens des opérateurs n'était pas suffisant.
- On aurait tendance à blâmer le dispositif automatique de déclenchement du sémaphore plutôt que le concepteur du protocole, et pourtant ...

La réaction est toujours la même :

I could not imagine that could ever happen

Au début, beaucoup d'accidents étaient dus au manque de moyens de communication, mais on a constaté ensuite qu'il était surtout très difficile d'établir des règles non ambiguës de communication.

Structure d'un protocole...

- **Pour définir un protocole**, nous devons définir un ensemble de règles :
 - définir comment un message est encodé
 - comment une transmission est démarrée et terminée, etc
- **Deux types d'erreurs** sont difficiles à éviter :
 - la conception d'un ensemble incomplet de règles (incomplétude)
 - la conception de règles contradictoires (inconsistance)
- **Cela requiert** :
 - de définir tous les éléments essentiels du protocole
 - une discipline pour les définir en séparant les éléments indépendants
- **La vraie difficulté** vient du parallélisme. Le nombre de scénarios est en général énorme et difficile à appréhender

Structure d'un protocole...

Les Cinq éléments d'un protocole

- Le service à fournir par le protocole
- Les hypothèses sur l'environnement dans lequel le protocole s'exécute
- Le vocabulaire des messages utilisé par le protocole
- Le format de chaque message
- Les règles procédurales utilisées durant la communication

La 5è partie est la plus délicate à définir et la plus difficile à vérifier.

Structure d'un protocole...

Un exemple

1. Le service :

- Transfert de fichiers de texte comme séquences de caractères via une ligne téléphonique en se protégeant contre les erreurs de transmission (supposée toutes détectables)
- Bidirectionnel : 2 transferts en sens opposés possibles

2. Les hypothèses :

- Composé de 2 utilisateurs et d'un canal de transmission
- On suppose que chaque utilisateur peut soumettre une requête et attendre qu'elle s'exécute
- Le canal peut modifier les messages, mais pas les perdre, les dédoubler, les réarranger, ni en insérer un autre.

3. Vocabulaire : trois messages

- ack : un message + un acquit positif
- nak : un message + un acquit négatif
- err : un message erroné

Structure d'un protocole...

Un exemple (suite)

5. Les règles procédurales :

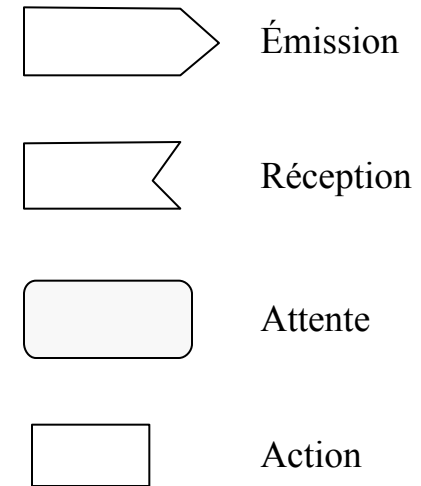
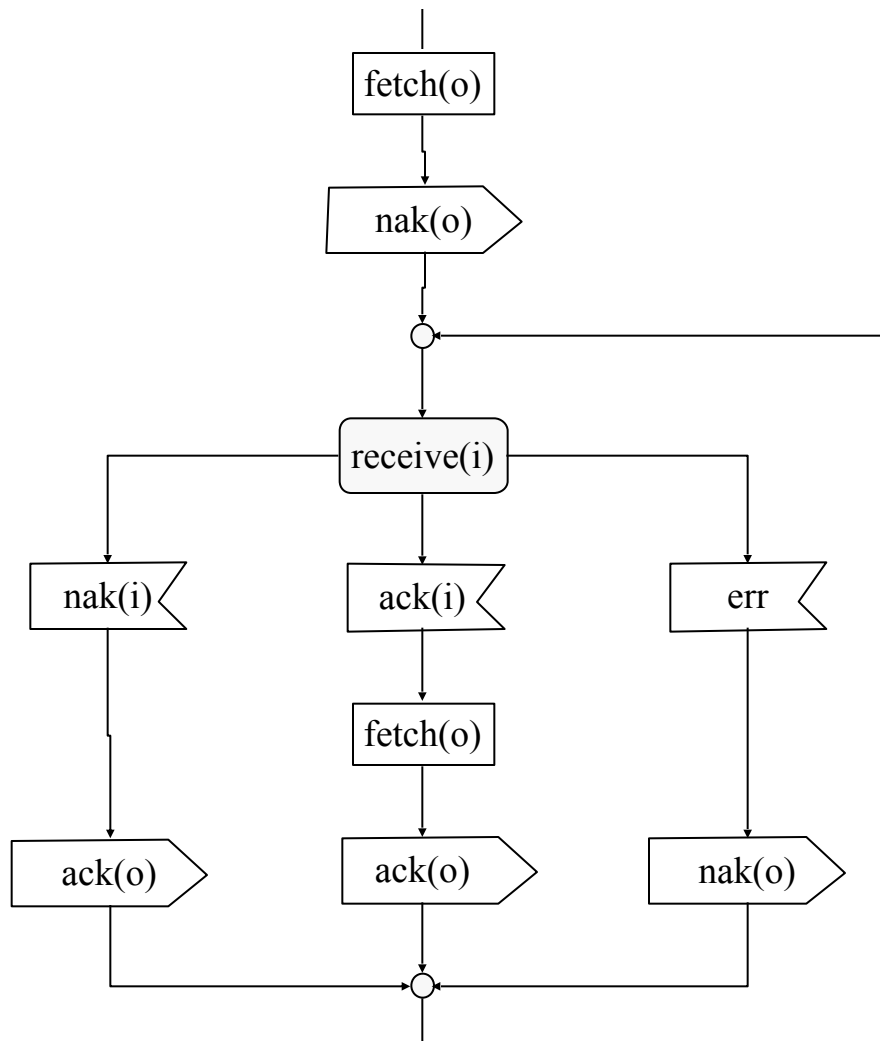
Soit 2 entités A et B

1. Si la réception précédente est sans erreur, le prochain message sur le canal opposé contiendra un acquit positif.
2. Si la réception précédente est erronée, le prochain message sur le canal opposé contiendra un acquit négatif.
3. Si la réception précédente est erronée ou contient un acquit négatif, le prochain message sera une retransmission du message émis précédemment; sinon, ce sera un nouveau message
4. Si un utilisateur n'a pas de caractère à émettre, il peut envoyer un caractère « null »
5. A démarre le transfert.

Structure d'un protocole...

Un exemple (suite)

Représentation des règles procédurales :



Protocole correct ?

Structure d'un protocole...

Notion vocabulaire et de format

```
Type PDU is
    record
        control : tag,
        data : char
    endrecord
Type tag is {ack, nak, err}
```

Attention : il s'agit de formats abstraits, et non d'un codage

Exemple de codage :

```
record
    control : bit,
    data : ASCII char,
    parity : bit
endrecord
+ des fonctions de calculs d'erreurs
```

Structure d'un protocole...

Notion de règles procédurales

- Exprimées au **niveau d'abstraction adéquat** (pas de détail inutile (codage))
- **Doivent être complètes, consistantes et non ambiguës**
 - => nécessite un formalisme ou un langage approprié
 - Il n'existe pas de méthode permettant de s'assurer a priori de la complétude et de la consistance des règles
 - => vérification a posteriori
- **La difficulté vient du parallélisme**
 - => Les diagrammes de séquences ne conviennent pas. Ils permettent au mieux d'illustrer un scénario d'erreur.
- Un protocole n'est pas correct **dans l'absolu** ; un protocole est correct par rapport à un ensemble de **propriétés** (la spécification de son service)
 - => Nécessité de formaliser les propriétés ou le service attendu

Règles d'ingénierie...

- **Simplicité — Concevoir des protocoles légers**
 - Faciles à comprendre, à implanter, à maintenir à jour, et souvent plus performants
- **Modularité — Une hiérarchie de fonctions**
 - Un protocole complexe doit être construit à partir de modules plus simples qui interagissent de façon simple et bien définie
 - Il ne faut pas mélanger des fonctions indépendantes (p. ex. contrôle d'erreur et contrôle de flux) dans un même module
 - Un protocole doit être indépendant du système d'exploitation, de l'encodage des données, des formats d'adresse, des débits utilisés...
 - ⇒ **Facilite l'ouverture** (portabilité), **l'extensibilité** ou le changement de modules...
- **Un protocole doit être bien formé**
 - Pas de sur-spécification : pas de code non exécutable
 - Pas de sous-spécification : pas de réception non prévue
 - Borné : pas de débordement de tampon
 - Auto-stabilisant : en cas d'erreur, retour dans un état stable après récupération

Règles d'ingénierie...

- **Robustesse**

- Concevoir un protocole qui fonctionne bien dans des circonstances normales est facile
- C'est l'inattendu qui défie le concepteur
- Il faut faire un minimum d'hypothèses sur l'environnement
- Un bon design doit résister à l'évolution (vitesse des lignes, taille du réseau, ...)
- Il faut éviter l'ajout de fonctions inutiles qui pourraient s'avérer être un frein à une évolution ultérieure.

- **Consistance : un protocole peut être incorrect de plusieurs façons**

- Interblocages (deadlocks) : toutes les entités sont bloquées en attente de message
- Cycles non productifs (livelocks) : le système boucle indéfiniment sans progresser
- Terminaisons impropres : sans satisfaire certaines conditions

Règles d'ingénierie...

quelques règles de bonnes conception

1. Etre sûr que le problème est **bien défini** :
 - il faut énumérer tous les critères et contraintes avant de commencer
2. Définir **le service à fournir** avant de concevoir le protocole
 - le quoi avant le comment
3. Principe **KISS** (Keep It Simple, Stupid)
4. Séparer les fonctionnalités orthogonales.
5. Un bon design résout **une classe de problèmes** plutôt qu'un problème isolé.
 - Ne pas se restreindre inutilement.
 - Ne pas introduire de détails inutiles.
6. Procéder par **prototypages de haut niveau** avant d'implémenter
7. Implémenter, mesurer les performances, et si nécessaire optimiser.
8. Vérifier que l'implémentation est équivalente au prototype de haut niveau.

DISTRIBUTED PROTOCOLS

Principles and Paradigms

ARCHITECTURES

Architectural Styles (1)

Styles of architecture is formulated in terms of

- The way that components are connected to each other
- The data exchanged between components
- How these elements are jointly configured into a systems

Architectural Styles (1)

Component is

- a modular unit
- with well-defined required and provided interfaces that is replaceable within its environments

Architectural Styles (2)

Important styles of architecture for distributed systems

- Layered architectures
- Object-based architectures
- Data-centered architectures
- Event-based architectures

Architectural Styles (3)

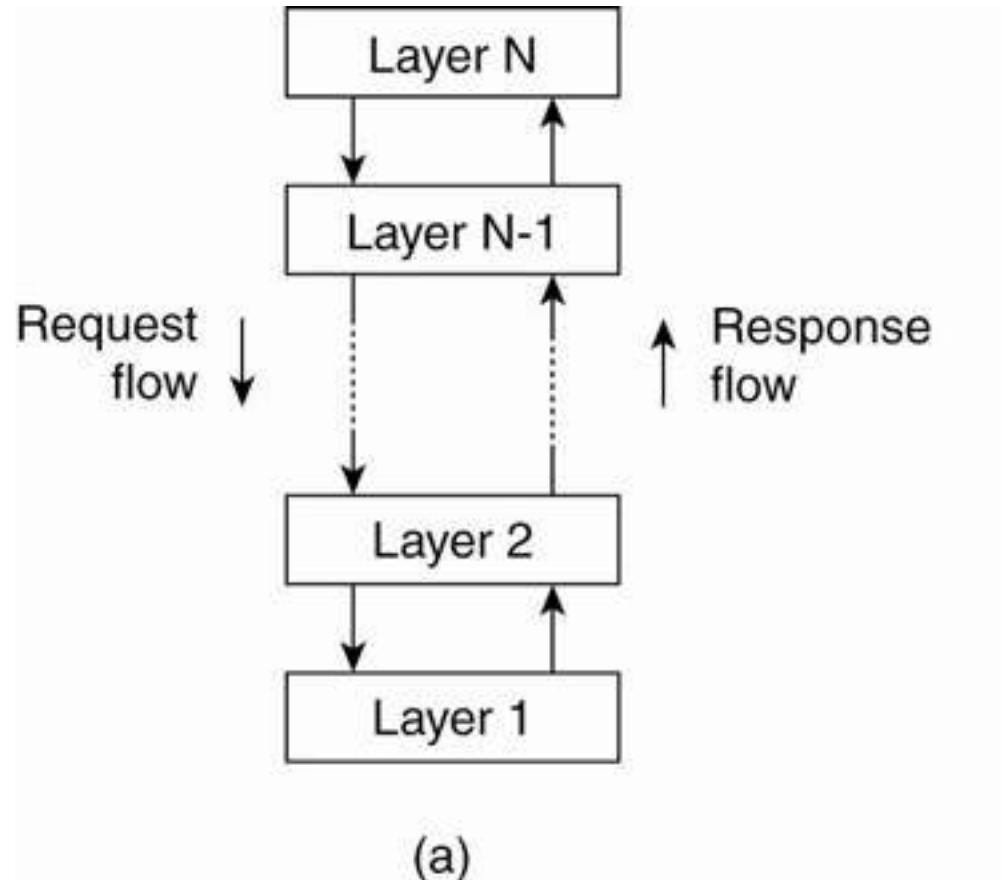


Figure 2-1. The (a) layered architectural style and ...

Architectural Styles (4)

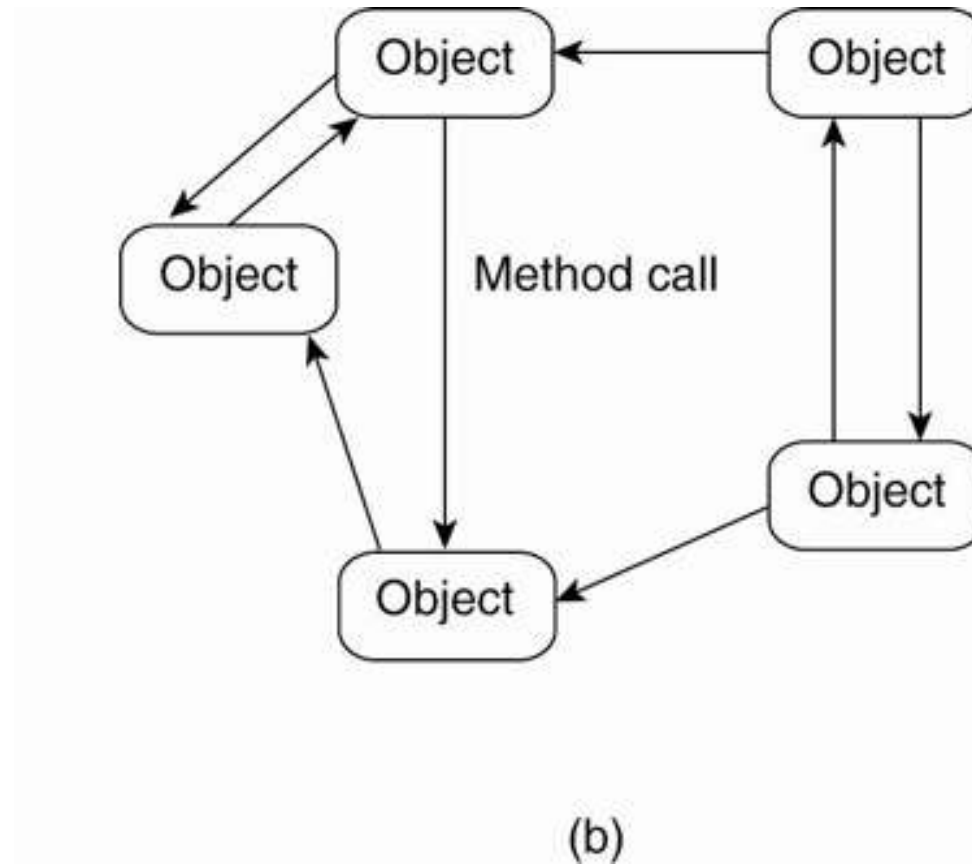


Figure 2-1. (b) The object-based architectural style.

Architectural Styles (5)

Event-based architectural style

- Processes communicate through the propagation of events, which may carry data
- Event are associated with publish/subscribe systems
 1. Processes publish events
 2. Middleware ensures that only those processes that subscribed to those events will receive them
 3. They (processes) need not explicitly refer to each other

Architectural Styles (5)

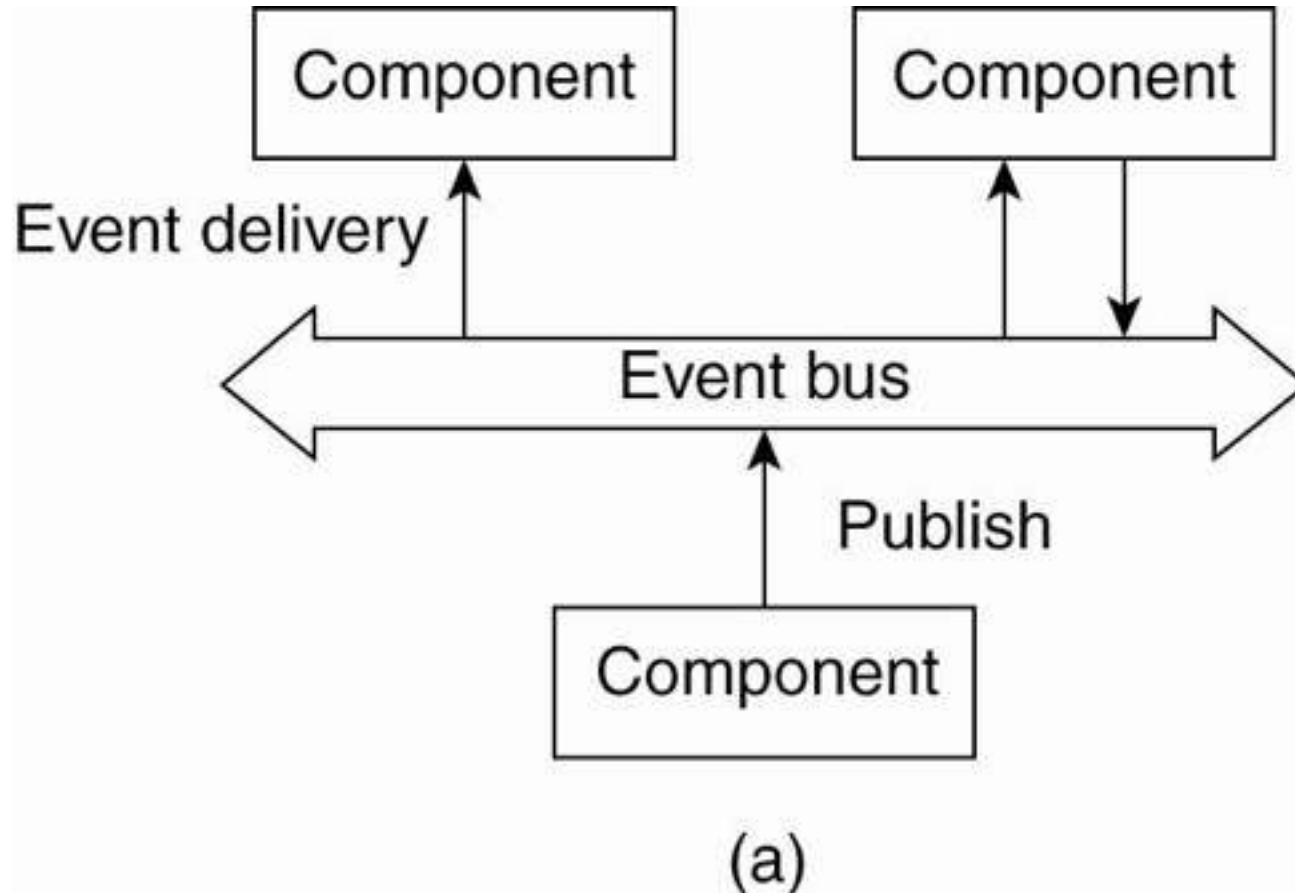


Figure 2-2. (a) The event-based architectural style and ...

Architectural Styles (5)

shared data-space architectural style

- Processes communicate through a common repository
- Can be combined with Event-based architectures yielding to a shared data spaces
 1. processes are now decoupled in time
 2. they need not both be active when communication takes place

Architectural Styles (6)

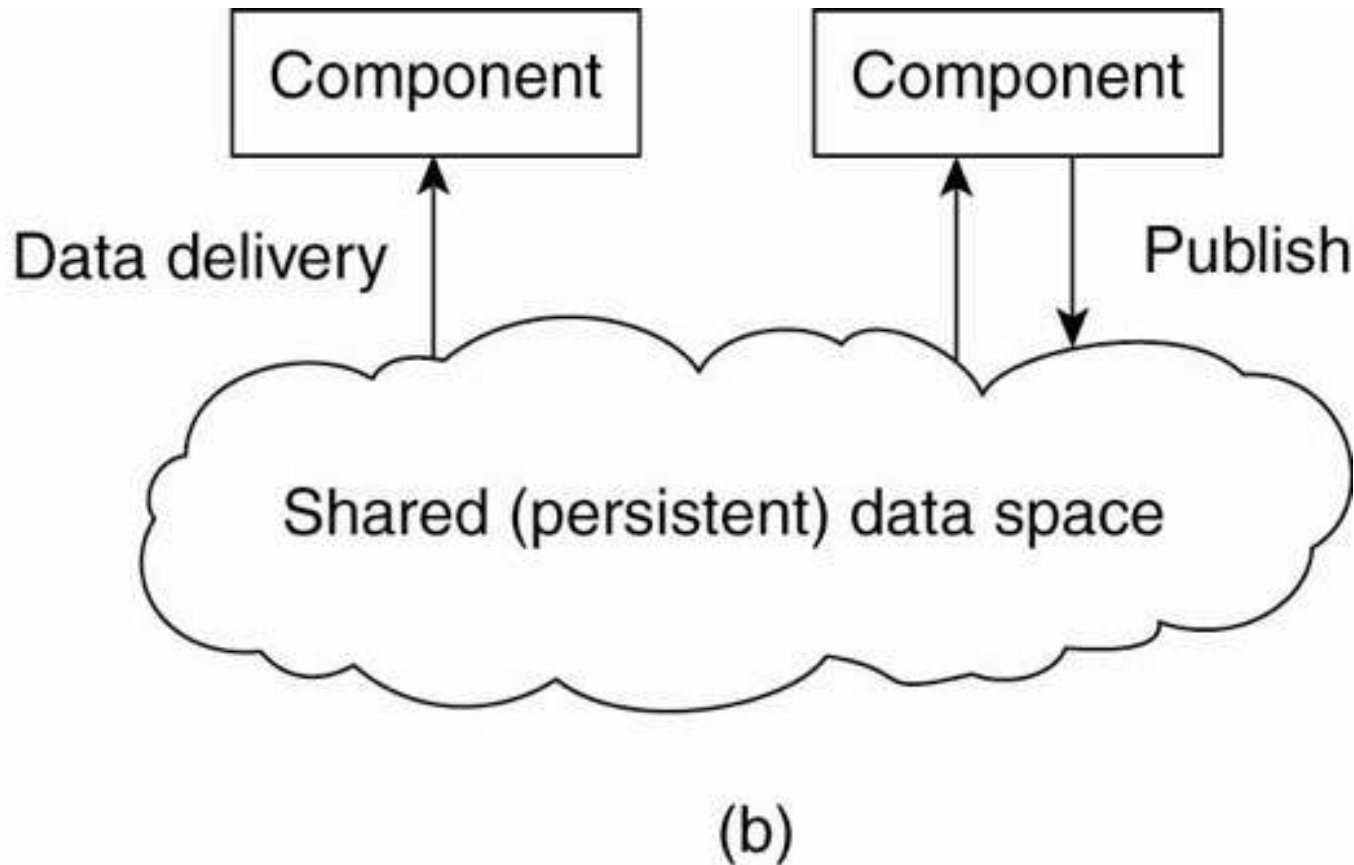


Figure 2-2. (b) The shared data-space architectural style.

Client-Server model

In the basic Client-Model, processes are divided into two groups

- A server: it is a process implementing a specific service
- A client: it is a process that requests a service from server by sending it a request and subsequently waiting for server's reply
- It can be implemented by means of a simple connectionless protocol
- or by means of reliable protocol

Centralized Architectures

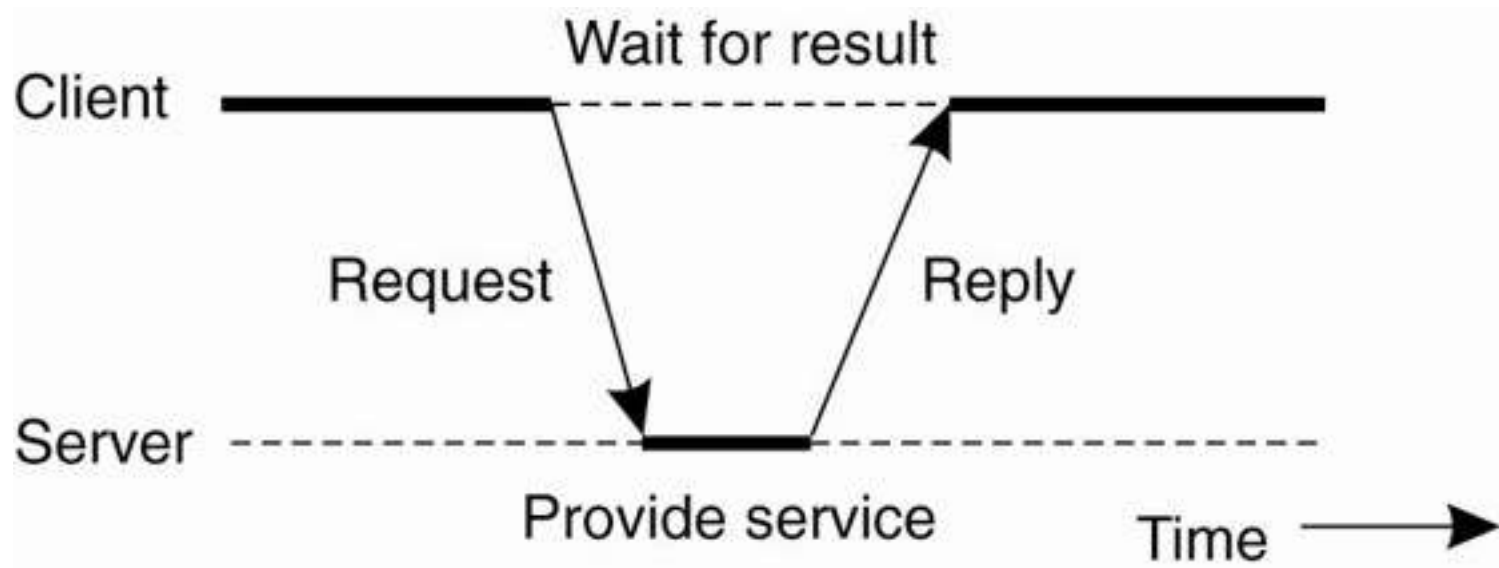


Figure 2-3. General interaction between a client and a server.

Application Layering (1)

Recall previously mentioned layers of architectural style

- The user-interface level
contains all that is necessary to directly interface with the user, such as display management
- The processing level
contains the applications
- The data level
manages the actual data that is being acted on

Application Layering (1)

Many client-server applications can be constructed from three different pieces

- a part that handles interaction with a user
- a part that operates on a database of file systems
- a middle part that generally contains the core functionality of an application

this middle part is logically placed at the processing level

Application Layering: Examples

Internet search engine

- user interface

a user type in a string of keywords and is subsequently presented with a list of titles of Web pages

- back end

It is formed by a huge database of Web pages that have been prefetched and indexed

- the core of the search

it is a program that transforms the user's string of keywords into one or more database queries

1. It subsequently ranks the results into a list
2. transforms that list into a series of HTML pages
3. this information retrieval is typically placed at the processing level

Application Layering (2)

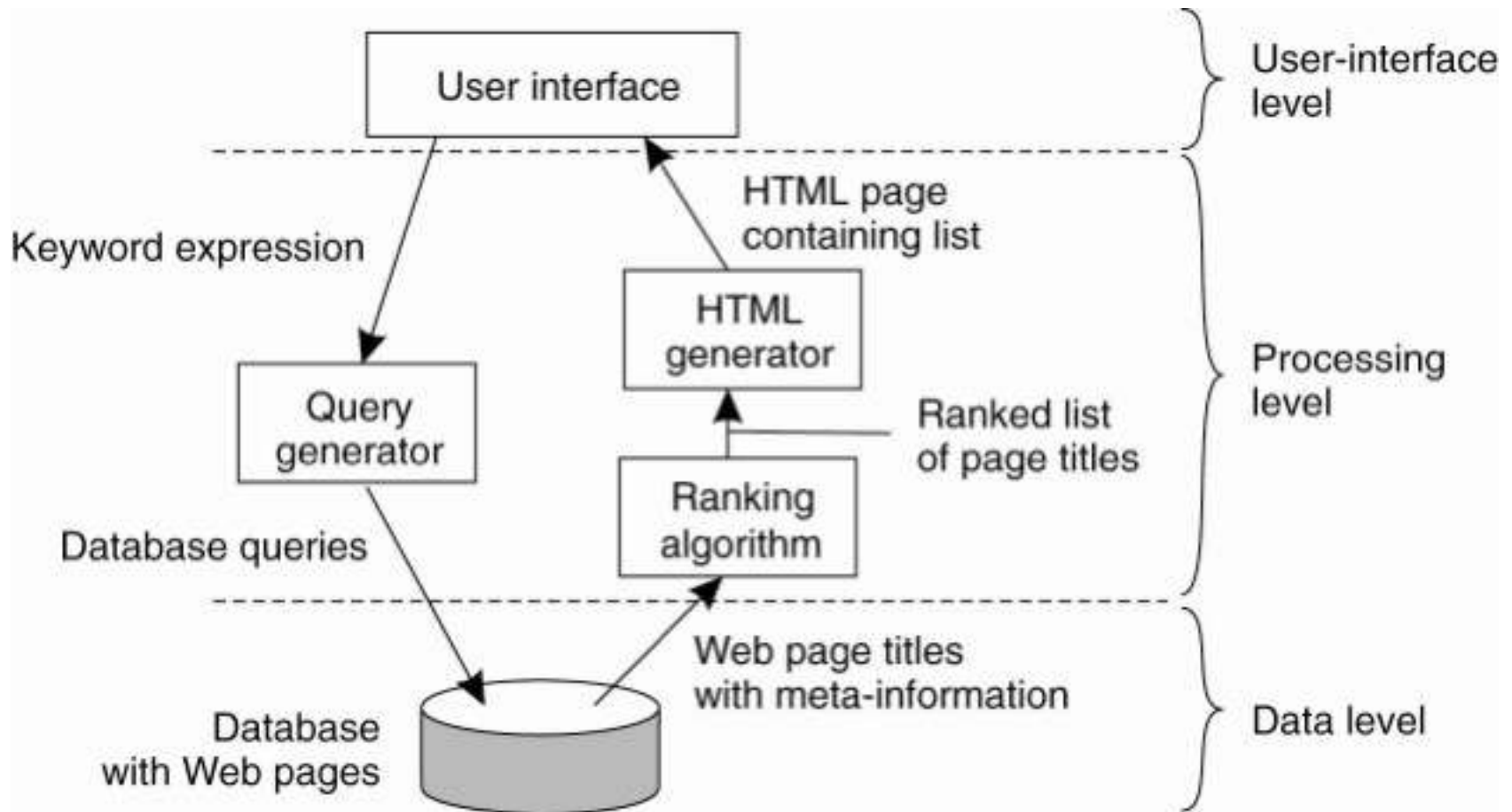


Figure 2-4. The simplified organization of an Internet search engine into three different layers.

Multitiered Architectures (1)

Three logical levels

Different possibilities for physically distributed a client-server application across several machine

The simplest organization is to have only two types of machines:

- A client machine containing only the programs implementing (part of) the user-interface level
- A server machine containing the rest,
 - the programs implementing the processing and data level

Multitiered Architectures (2)

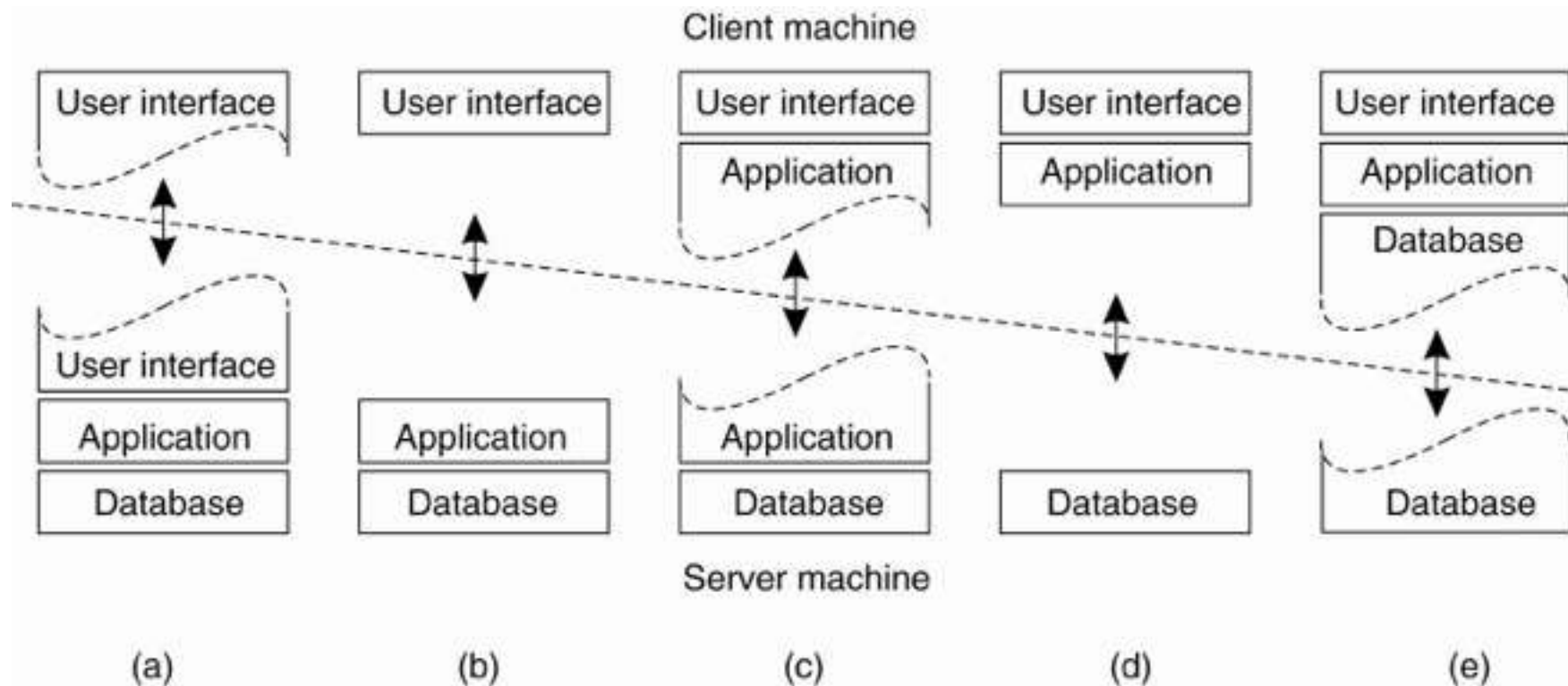


Figure 2-5. Alternative client-server organizations (a)–(e).

Multitiered Architectures (3)

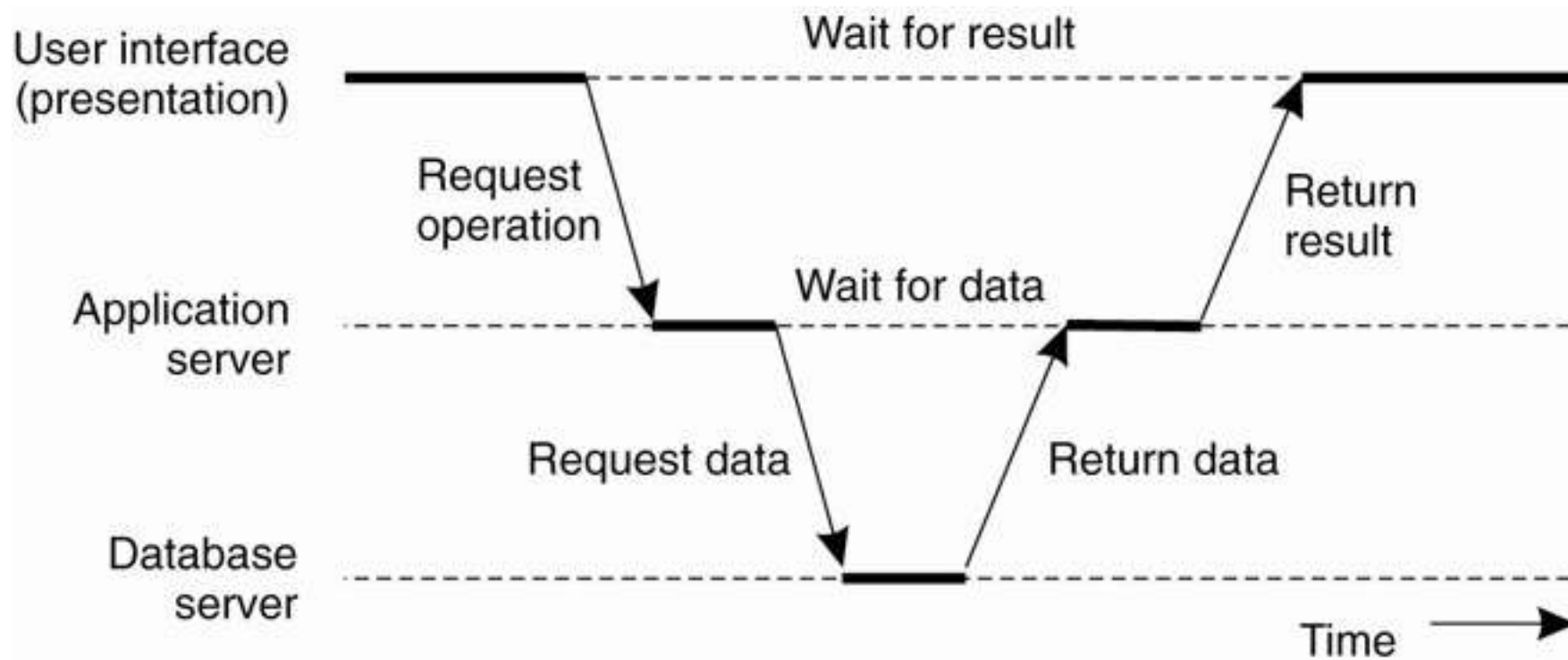
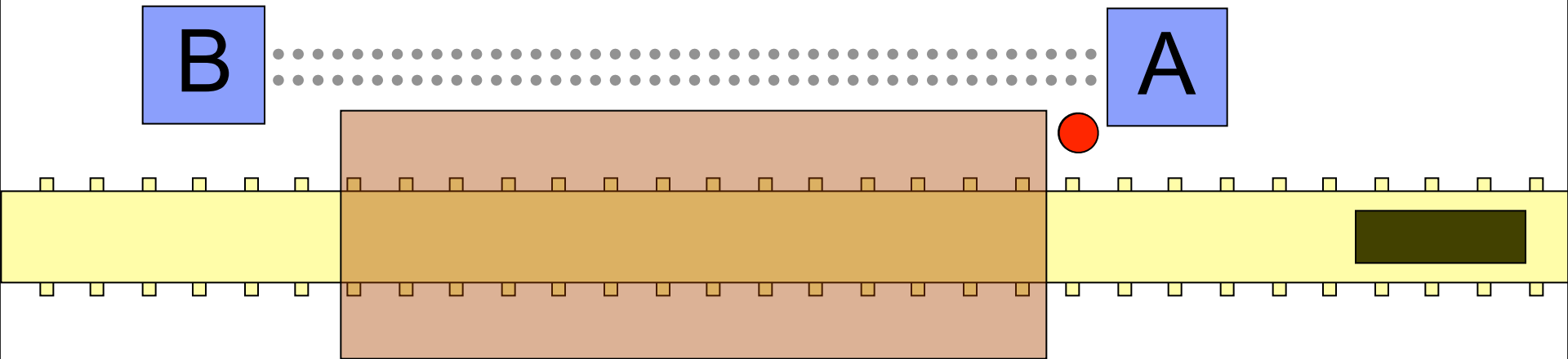


Figure 2-6. An example of a server acting as client.

Communication Protocols: Early Beginnings

- Problem of designing efficient and unambiguous communication protocols existed even before the advent of computers
- Number of systems for communication over long distances
 - Fire signals and torch telegraph (458 BC)
 - Chappe's semaphore/optical telegraph (Claude Chappe 1793)
 - Electric telegraph (William Cooke 1837)
- Problems with early communication systems
 - Number of messages that can be communicated by fire signals is limited (and previously determined) - Unforeseen things can't be communicated
 - Synchronization problems
 - What happens if two messages arrive simultaneously to a semaphore from opposite directions (relay messages)?
 - Electric telegraphs could not prevent some railway crashes due to misunderstandings among signalmen
- Common problem: unexpected occurrences, which require immediate attention lead to protocol failures → expect the unexpected

Malfunctioning Semaphore



The first train enters the tunnel – the semaphore fails to turn the light red!

Signalman A sends a message to signalman B, that a train is in the tunnel.

The second train approaches the tunnel.

Signalman A runs to the track and signals to the train to stop and sets the light to red, manually.

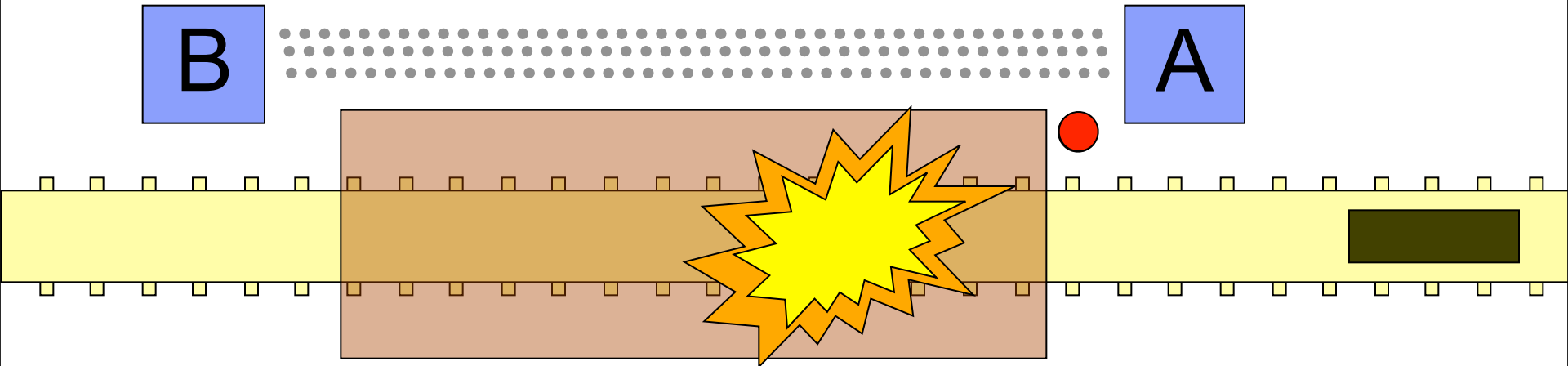
The first train exits the tunnel.

B sends a message to A, that the first train has exited the tunnel.

A sets the signal to green and lets the second train pass.

The second train enters and leaves the tunnel.

Clayton Tunnel Accident - 25 August 1861



The first train enters the tunnel - the semaphore fails to turn the light red!

Signalman A sends a message to signalman B, that a train is in the tunnel.

B sends a message to A, that the train has exited the tunnel.

Signalman A runs to the track to get the light to red and signal other trains to stop, and he sets the light to green.

But a second train arrives and enters the tunnel before he has time to change the light to red. The driver sees something but doesn't have time to stop.

The second train driver decides to back out of the tunnel.

Signalman A sends another message to signalman B, that a train is in the tunnel.

The third train enters the tunnel.

The third train approaches the tunnel and stops.

The second and third trains collide.

The driver of the second train is suspicious that something is wrong and stops in the middle of the tunnel.

Protocols as Languages

- An informal definition of a communication protocol:
 - An agreement for exchange of information in a distributed system
- A more careful definition looks like a language definition:
 - Defines a precise **format** of valid messages (**syntax**)
 - Defines **procedure rules** for data exchange (**grammar**)
 - Example: transmission initiator and responder follow different procedures
 - Defines a **vocabulary** of valid messages that can be exchanged, with their meaning (**semantics**)
 - Example: control messages, acknowledgements, etc.
- Grammar of a protocol must be logically consistent and complete:
 - Rules should specify clearly (unambiguously) what is allowed and what is forbidden
 - This is a difficult requirement to meet, in practice

The Five Elements of a Protocol

- A protocol specification consists of five distinct parts:
 1. The **service** to be provided by the protocol
 2. The **assumptions** about the environment in which the protocol is executed
 3. The **vocabulary** of messages used to implement the protocol
 4. The **encoding** (format) of each message in the vocabulary
 5. The **procedure rules** guarding the consistency of message exchanges

Requirements of Designing a Protocol

- Hidden requirement:
 - Rules for exchange of information is an obvious requirement
 - There should be **agreement** between sender and receiver about those rules
- Protocol design results in more than one implementation
 - Different implementations leave room for short-cuts or improvements
 - Such differing implementations may not work together
 - Example: many implementations of IBM's Bisync protocol rule out any hopes of communicating among different implementations
- Solution to multiple implementations:
 - Institution of international standardization bodies
 - Other possibility could have been more strict guidelines for the design, specification, and implementation of protocols

Assumptions

- The “environment” in which the protocol is to be executed consists minimally of two users of the file transfer service and a transmission channel.
- The users can be assumed to simply submit a request for file transfer and await its completion.
- The transmission channel is assumed to cause arbitrary message distortions, but not to lose, duplicate, insert, or reorder messages.
- We will assume here that a lower-level module is used to catch all distortions and change them into undistorted messages of type *err*.

Protocol Vocabulary

- The protocol vocabulary defines three distinct types of messages:
 - *ack* for a message combined with a positive acknowledgment,
 - *nak* for a message combined with a negative acknowledgment,
 - *err* for a message with a transmission error.
- The vocabulary can be succinctly expressed as a set:

$$V = \{ \mathbf{ack}, \mathbf{err}, \mathbf{nak} \}$$

Message Format (Signaling)

- Each message consists of a control field identifying the message type and a data field with the character code
- The general form of each message can now be represented symbolically as a simple structure of two fields:

{tag, data }

- which in a C-like specification may be specified in more detail as follows:

enum control { ack, nak, err };

struct message {

enum control tag;

unsigned char data;

};

- The message structure itself contains two fields: a tag of type control, and a data field declared as an unsigned character (one byte).

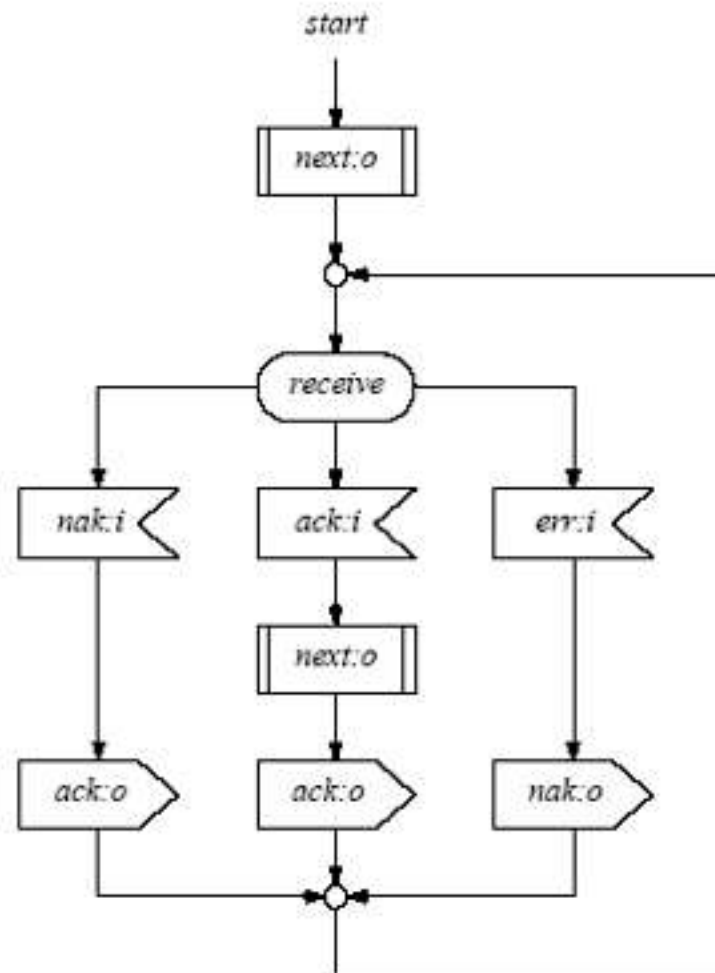
Procedure Rules

- The procedure rules for the protocol were **informally** described as follows:
 1. *“If the previous reception was error-free, the next message on the reverse channel will carry a positive acknowledgment; if the reception was in error it will carry a negative acknowledgment.”*
 2. *“If the previous reception carried a negative acknowledgment, or the previous reception was in error, retransmit the old message; otherwise fetch a new message for transmission.”*

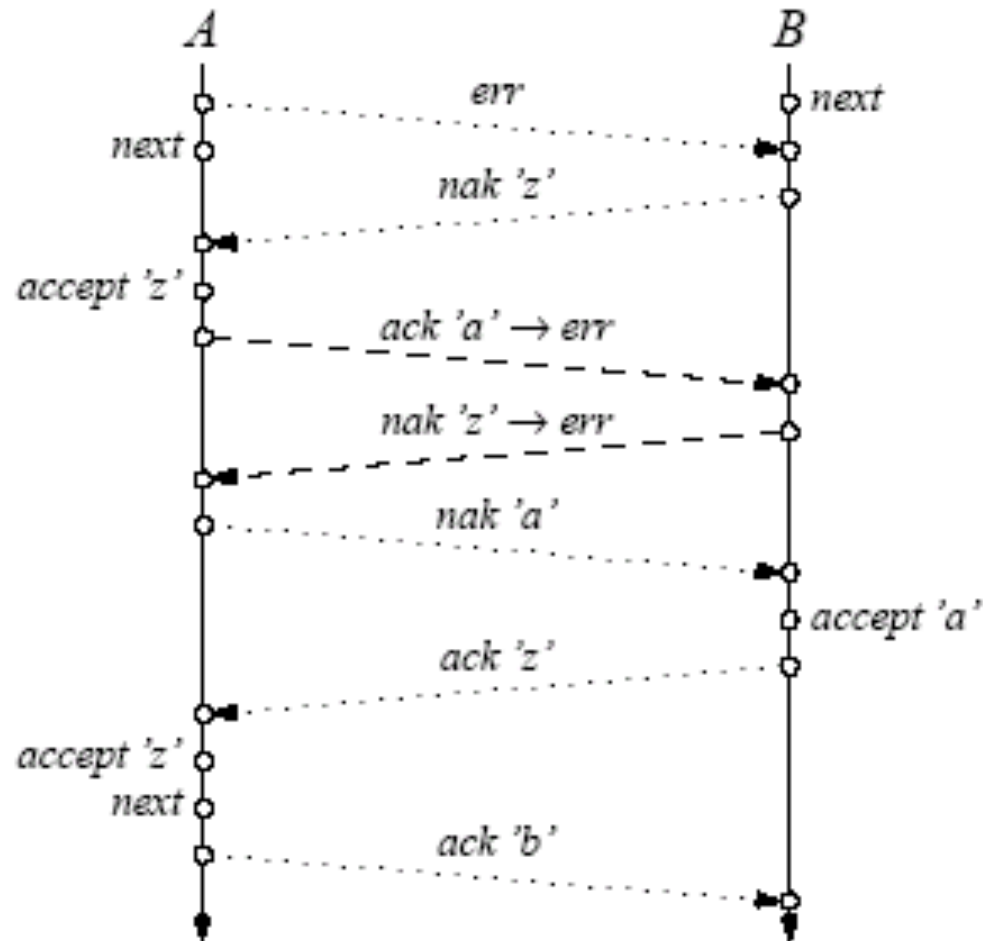
Procedure Rules

- The procedure rules for the protocol were **informally** described as follows:
 1. *“If the previous reception was error-free, the next message on the reverse channel will carry a positive acknowledgment; if the reception was in error it will carry a negative acknowledgment.”*
 3. *“If the previous reception carried a negative acknowledgment, or the previous reception was in error, retransmit the old message; otherwise fetch a new message for transmission.”*

Procedure Rules (State Transition Diagram)



Protocol Problems (Time Sequence Diagram)



Well-Formed Protocols

- Well formed protocol is:

1. ***bounded***

- *It cannot overflow known system limits*
limited capacity of message queues

2. ***self-stabilizing***

- *always return to a desirable state within a finite number of transitions*

3. ***self-adapting***

- *for instance, the rate at which a data are sent to the rate at which the data links can transfer them*

Protocols Design: Consistency

- Consistency:

1. **Deadlock**

- *No further protocol execution is possible*

2. **Livelocks**

- *Execution sequences that can be repeated indefinitely*

3. **Improper termination**

- *the completion of a protocol execution without satisfying the proper termination conditions*

Protocol Examples

- *Clayton Tunnel Protocol*
 - *Service:*
 - *Assumptions*
 - *Vocabulary:*
 - *Encoding:*
 - *Procedure rules:*

Communication interprocessus

Trois points importants:

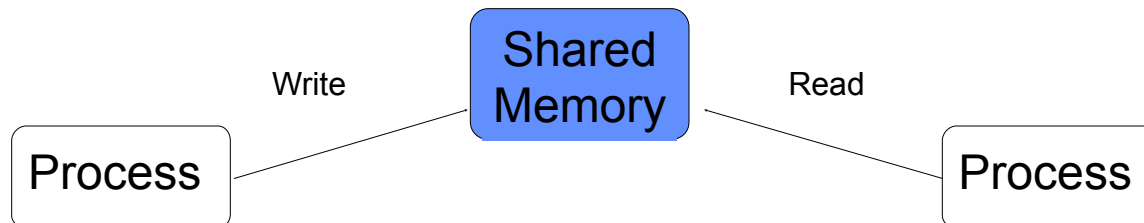
- Comment partager les ressources?
- Concurrence pour l'utilisation des ressources
- Synchronisation

Communication Mechanisms for Concurrent Systems

- **Message passing involves sending and receiving messages through a channel**

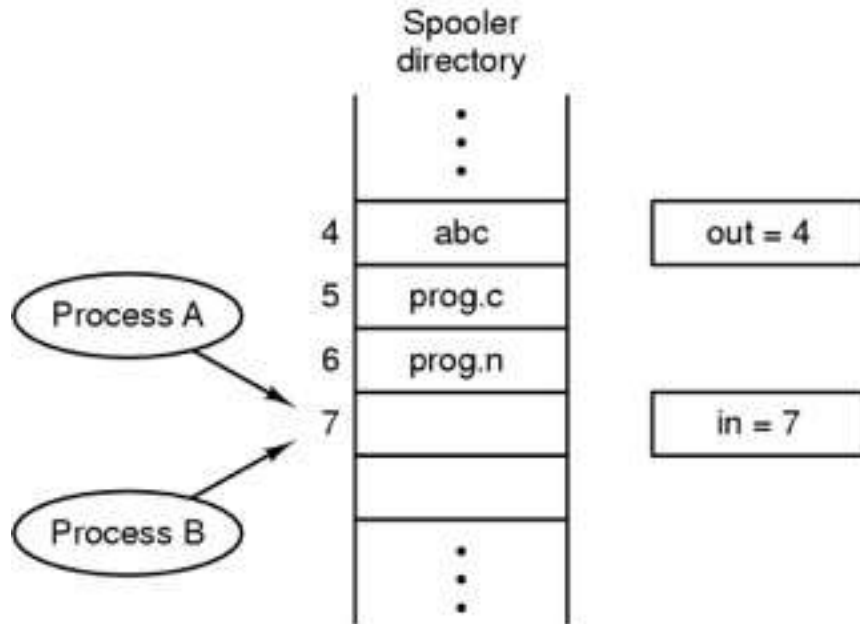


- **In the Shared Memory Approach memory is common to both processes, and they can read and write to the memory**



- **The two are equivalent. We will only consider shared memory**

Conditions de concurrence



Deux processus veulent accéder à la mémoire partagée en même temps.

Par exemple: Les processus A et B veulent utiliser la file d'impression.

1. A voit que la prochaine case disponible est 7 puis est remplacé par B
2. B voit que la prochaine case disponible est 7, y place le nom de son fichier, puis est remplacé par A
3. A place le nom de son fichier dans une case qu'il croit libre écrasant le nom donné par B

Sections Critiques

L'**exclusion mutuelle** est une méthode qui permet de s'assurer que si un processus utilise une variable ou un fichier partagé, les autres processus seront exclus de la même activité.

Une **section critique** (ou **région critique**) est la partie du programme à partir de laquelle on accède à la mémoire partagée.

Sections Critiques: good properties

- **Mutual Exclusion:** No two threads are in their critical sections at the same time.
- **Deadlock-freedom:** If a thread is trying to enter its critical section, then some thread, not necessarily the same one, eventually enters its critical section.
- **Starvation-freedom:** If a thread is trying to enter its critical section, then this thread must eventually enter its critical section.
- Starvation-freedom is a stronger property than Deadlock-freedom.

Sections Critiques:good properties

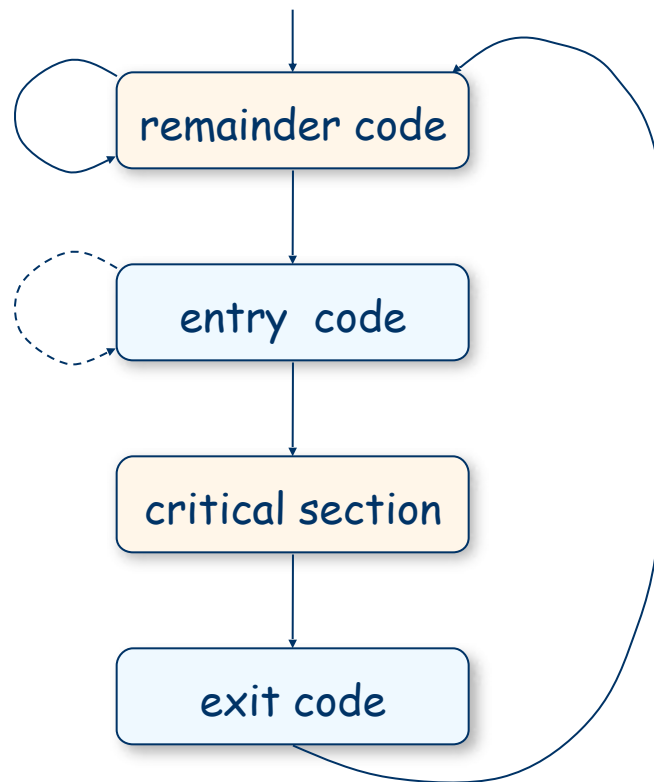
Conditions pour obtenir l'exclusion mutuelle

- **Mutual Exclusion:** Un seul processus à la fois dans la section critique
- Aucune supposition sur le nombre et la vitesse des CPUs
- Aucun processus hors de la section critique ne peut bloquer un autre processus.

Interblocages (**Deadlock-freedom**) : toutes les entités sont bloquées en attente de message

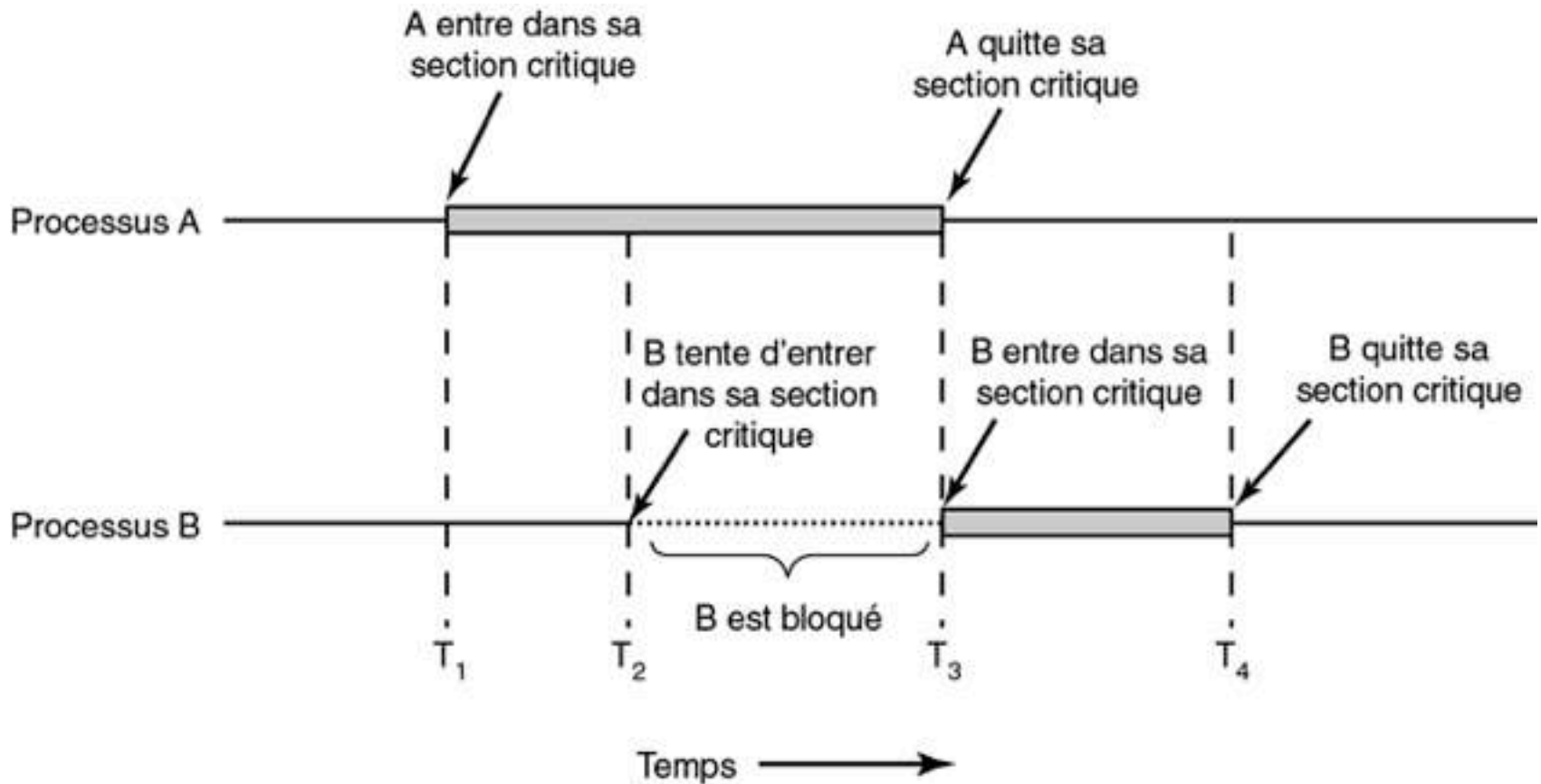
- **Starvation-freedom:** Aucun processus ne doit attendre indéfiniment pour entrer en section critique.

The mutual exclusion problem



The problem is to design the entry and exit code in a way that guarantees that the mutual exclusion and deadlock-freedom properties are satisfied.

Sections critiques (3)



© Pearson Education France

Exclusion mutuelle à l'aide des sections critiques.

Solutions à l'exclusion mutuelle

- Désactivation des interruption:
 - Instruction disable
 - On ne peut pas permettre aux usagers d'utiliser une telle instruction
- Variable de verrou
 - Un processus peut entrer seulement si $VERROU=0$ Il met alors $VERROU=1$ pour empêcher les autres processus d'entrer
 - Cette solution ne fonctionne pas!

Alternance stricte

```
while (TRUE) {  
    while (turn != 0)    /* loop */ ;  
    critical_region( );  
    turn = 1;  
    noncritical_region( );  
}
```

(a)

```
while (TRUE) {  
    while (turn != 1)    /* loop */ ;  
    critical_region( );  
    turn = 0;  
    noncritical_region( );  
}
```

(b)

Proposition de solution au problème de la section critique

(a) Processus 0. (b) Processus 1.

Solution de Peterson (1981)

```
#define FALSE 0
#define TRUE  1
#define N     2                                /* nombre de processus */
int turn;                                       /* à qui le tour ? */
int interested[N];                             /* toutes valeurs initialement 0 (FALSE) */

void enter_region(int i) {                     /* le processus est 0 ou 1 */
    int other;                                /* nombre des autres processus */
    other = 1 - i;                            /* l'opposé du processus */
    interested[i] = TRUE;                     /* montre que i est intéressé */
    turn = i;                                 /* au tour de i */
    while (turn == i && interested[other] == TRUE) ;
                                                /* instruction null */
}

void leave_region(int i) {                     /* i: processus qui quitte */
    interested[i] = FALSE;                    /* indique le départ de la section critique */
}
```

L'instruction TSL (Test and Set Lock)

TSL R, LOCK est une opération atomique équivalente à:

R=LOCK

LOCK=1

enter_region:

TSL REGISTER,LOCK	copie lock dans le registre et le définit à 1
CMP REGISTER,#0	lock était-il à 0 ?
JNE enter_region	s'il n'était pas à 0, boucle
RET	retourne à l'appelant ; entre en section critique

leave_region:

MOVE LOCK,#0	stocke un 0 dans lock
RET	retourne à l'appelant

Monitor Defined

- ***Monitor***: a lock + condition variables
- ***Lock***: provides mutual exclusion to shared data
- ***Condition variable***: provides a queue for waiting threads inside a critical section

Locks

- A lock provides two operations:
 - Lock::Acquire()
 - Wait until the lock is free, then grab it
 - Lock::Release()
 - Unlock, wake up anyone waiting to Acquire
 - A lock is initially free

More on Locks

- Always acquire a lock before accessing a shared data structure
- Always release the lock after finishing with the shared data structure

An Example of Using Locks

```
AddToQueue() {  
    Lock.Acquire();  
    // put 1 item to the queue  
    Lock.Release();  
}
```

```
RemoveFromQueue() {  
    Lock.Acquire();  
    // if something on the queue  
    //    remove 1 item form the queue  
    Lock.Release();  
    return item;  
}
```

Condition Variables

- We need additional mechanisms to wait inside locked regions
- However, holding the lock while waiting prevents other threads from entering the locked region
- Condition variables make it possible to sleep inside a critical section
 - Atomically release the lock & go to sleep

Condition Variables

- Each condition variable
 - Consists of a queue of threads
 - Provides three operations
 - Wait();
 - Atomically release the lock and go to sleep
 - Reacquire lock on return
 - Signal();
 - Wake up one waiting thread, if any
 - Broadcast();
 - Wake up all waiting threads

An Example of Using Condition Variables

```
AddToQueue() {  
    lock.Acquire();  
    // put 1 item to the queue  
    condition.Signal(&lock);  
    lock.Release();  
}
```

```
RemoveFromQueue() {  
    lock.Acquire();  
    while nothing on queue  
        condition.Wait(&lock);  
    lock.Release();  
    return item;  
}
```

Les Sockets

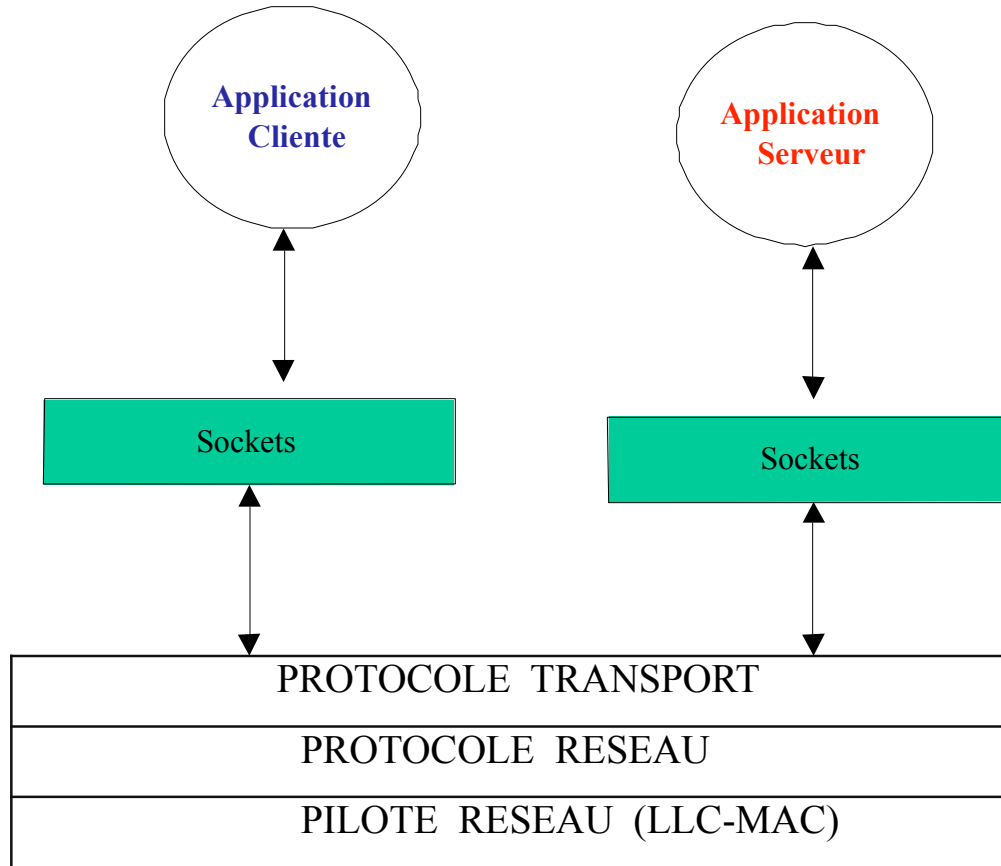
La version Berkeley 4.2 d'Unix a été la première à inclure TCP/IP dans le noyau du système d'exploitation

- et à proposer une interface de programmation de ces protocoles : les *sockets*

Les sockets sont

- une API (*Application Program Interface*), interface avec les couches réseau.
 - un point de communication par lequel un processus peut émettre ou recevoir des données
-

Les Sockets

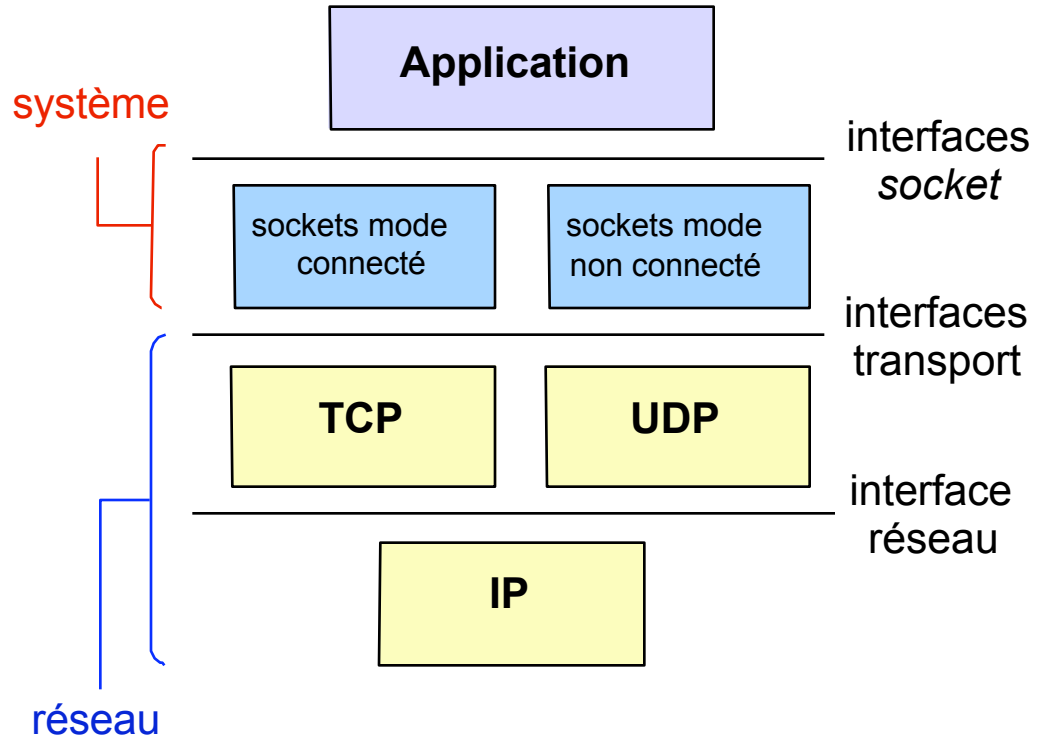


Place des sockets

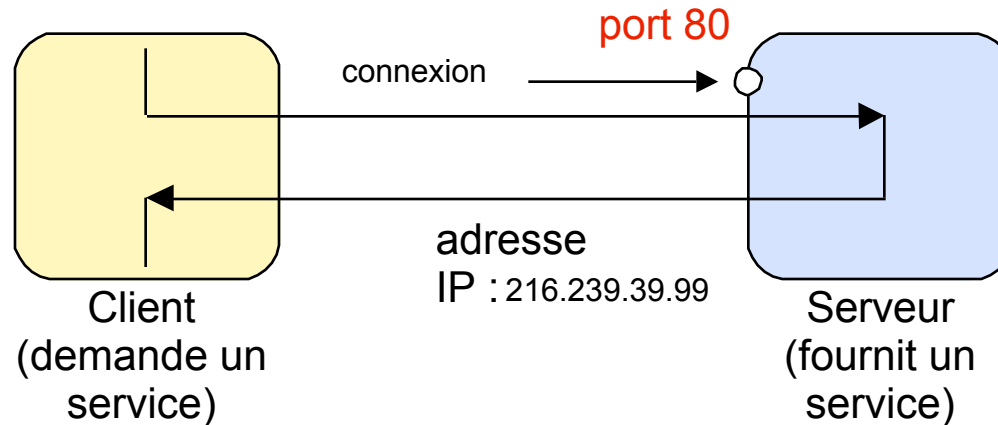
Les *sockets* fournissent une interface d'accès, à partir d'un hôte, aux interfaces de transport TCP et UDP

TCP (mode connecté) : une liaison est établie au préalable entre deux hôtes, et ensuite les messages (plus exactement des flots d'octets) sont échangés sur cette liaison

UDP (mode non connecté) : aucune liaison n'est établie. Les messages sont échangés individuellement



Le réseau vu de l'utilisateur



Le serveur (machine physique) peut comporter différents services;

- il faut préciser le service demandé au moyen d'un **numéro de port**, qui permet d'atteindre un processus particulier sur la machine serveur.
- Un numéro de port comprend 16 bits (0 à 65 535). Les numéros de 0 à 1023 sont réservés, par convention, à des services spécifiques.

Exemples :

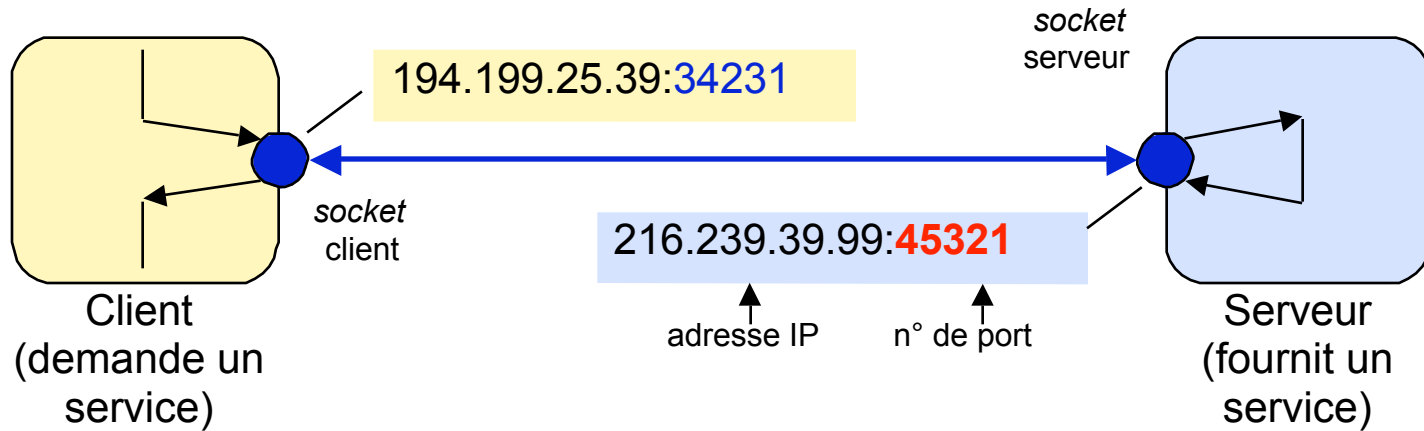
7 : echo

80 : serveur web

23 : telnet (connexion à distance)

25 : mail

Le réseau vu de l'utilisateur (3)



Pour établir une communication vers une machine distante, il faut :

- s'attribuer (ou laisser le soin au système d'attribuer) un numéro de port
- déterminer l'adresse Internet de la machine avec laquelle les échanges vont s'effectuer
- connaître ou déterminer le numéro de port du service distant.

Les Sockets

Pour créer une socket, on doit préciser :

- le domaine de travail : Unix (AF_UNIX), TCP/IP (AF_INET), etc.
 - le type de protocole de communication :
 - **SOCK_DGRAM** : envoi de messages sous forme de datagrammes (exemple UDP dans le domaine AF_INET)
 - **SOCK_STREAM** : envoi de flux d'octets (exemple TCP dans le domaine AF_INET)
 - **SOCK_RAW** : il permet (moyennant des droits système) l'accès à des protocoles de plus bas niveau, comme IP dans le domaine AF_INET.
-

Utilisation des Sockets en mode connecté

CLIENT

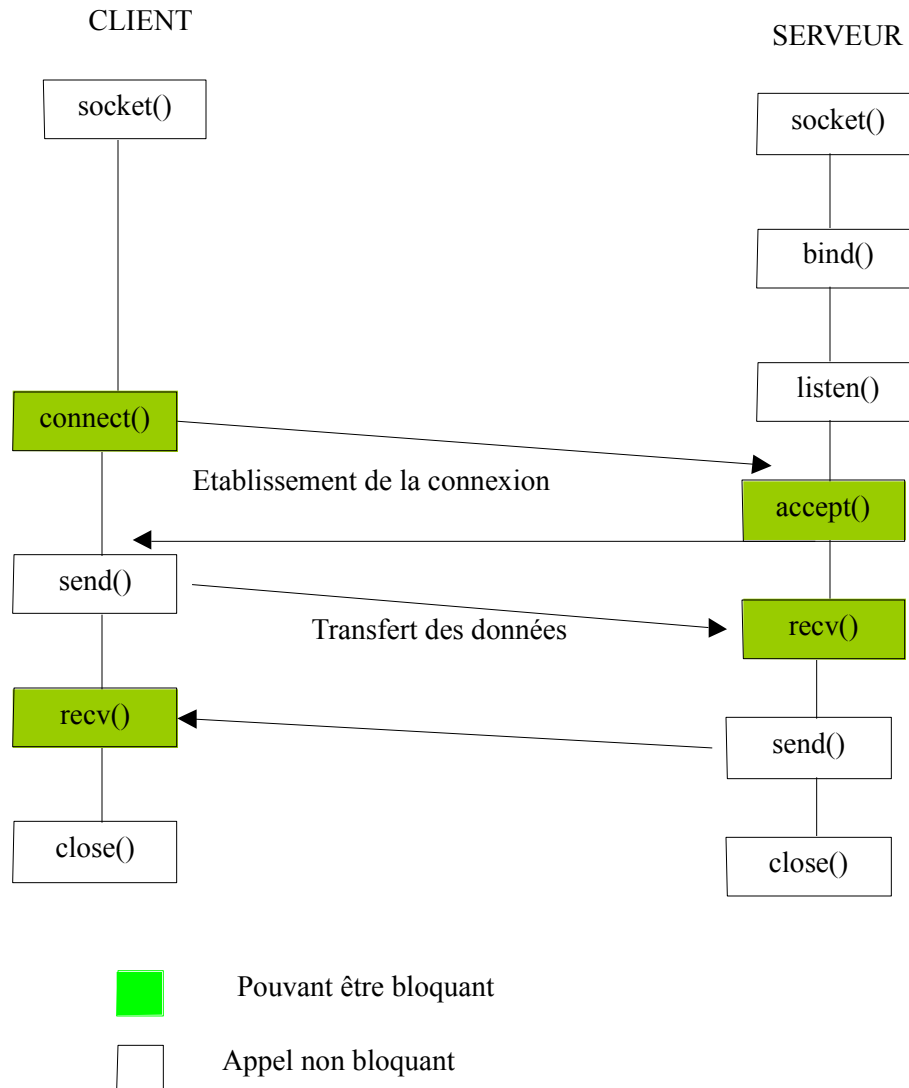
1. crée une socket
2. se connecte au serveur en donnant l'adresse socket distante (adresse Internet du serveur et numéro de port du service).
 - Cette condition attribue automatiquement un numéro de port local au client.
3. lit ou écrit sur la socket
4. ferme la socket.

Utilisation des Sockets en mode connecté/2

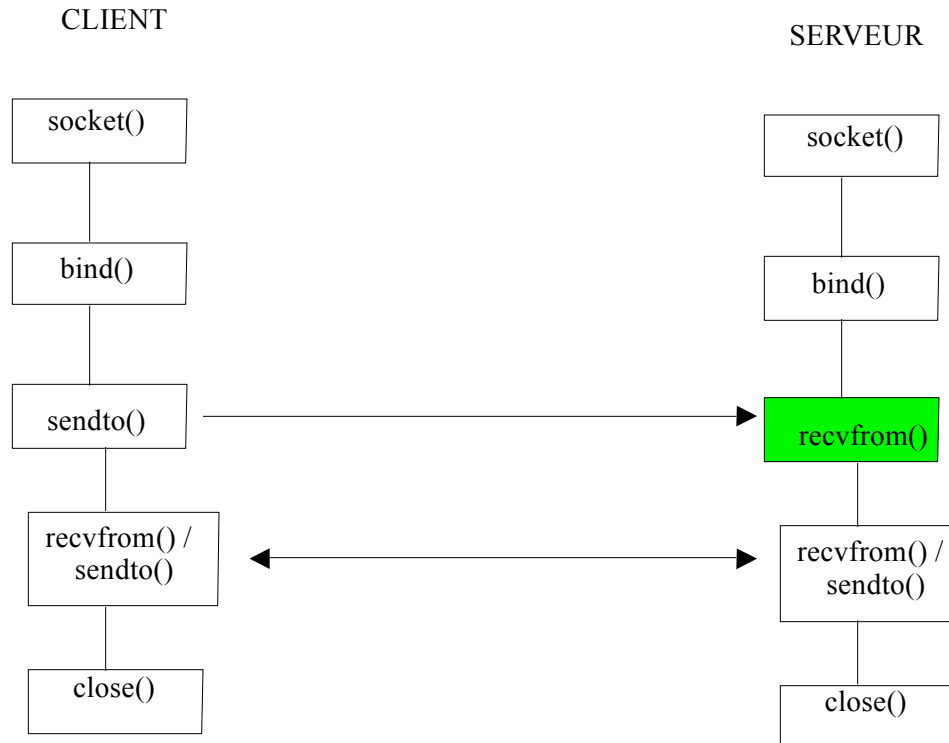
SERVEUR

1. crée une socket
2. associe une adresse socket (son adresse Internet et le numéro de port choisi) au service : “ binding ”
3. se met à l’écoute des connexions entrantes
4. Pour chaque connexion entrante :
 - a. accepte la connexion (une nouvelle socket est ainsi créée avec les mêmes caractéristiques que la socket d’origine)
 - b. lit ou écrit sur la nouvelle socket
 - c. ferme la nouvelle socket.

Le modèle client/serveur en mode TCP



Le modèle client/serveur en mode UDP



Pouvant être bloquant

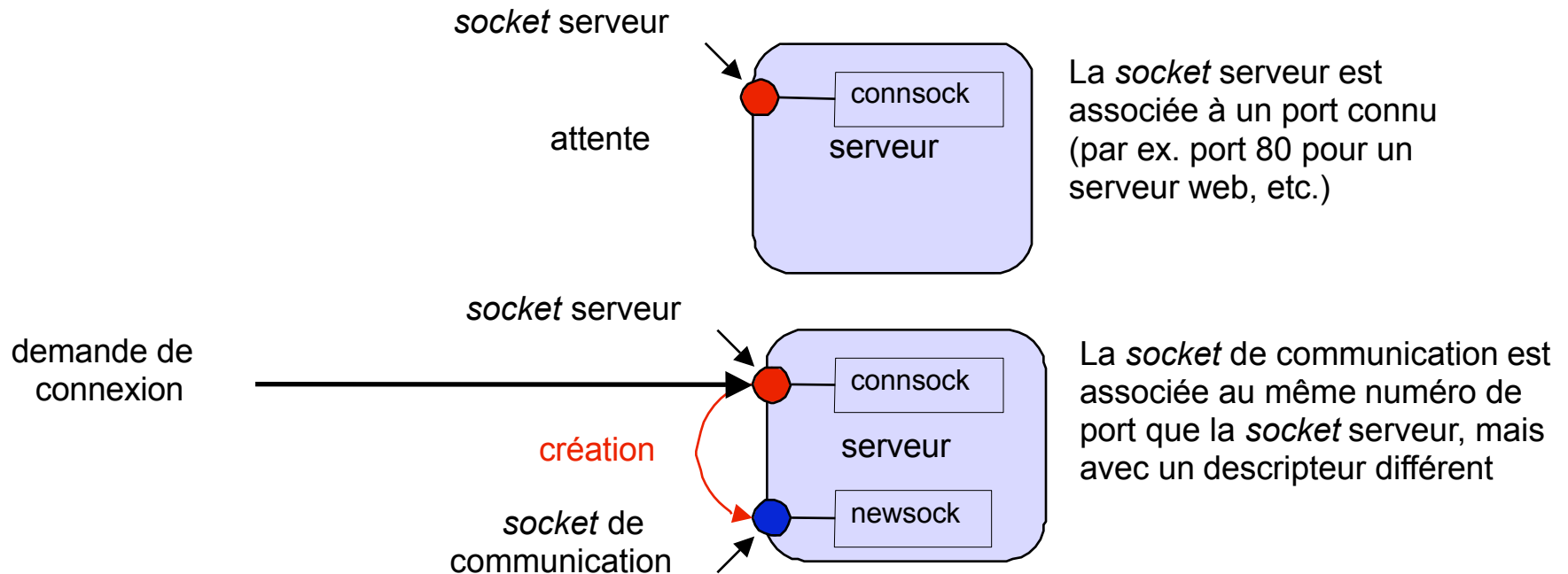


Appel non bloquant

Sockets côté serveur

Un serveur fournit un service à des clients. Il doit donc **attendre** une demande, puis la **traiter**.

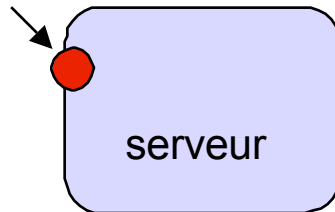
Les fonctions d'attente et de traitement sont séparées, pour permettre au serveur d'attendre de nouvelles demandes pendant qu'il traite des requêtes en cours



Sockets côté serveur

On procède en 4 étapes, décrites schématiquement ci-après (détails plus tard)

socket serveur



Étape 1 : créer une socket :

```
connsock = socket.socket(socket.AF_INET,  
                           socket.SOCK_STREAM, 0)
```

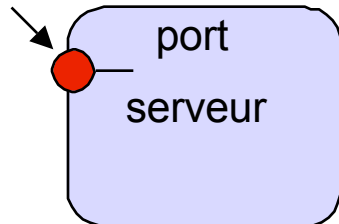
Internet

TCP

protocole,
sans usage ici

connsock est un objet socket

socket serveur

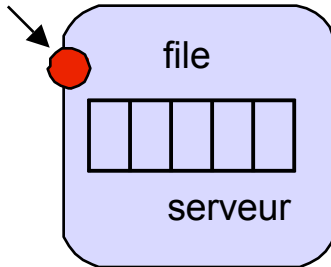


Étape 2 : associer la socket à un port:

Créer un adresse locale (couple {adresse,port})
serveraddr = (addr, 5678)
connsock.bind(serveraddr)

Sockets côté serveur (3)

socket serveur



Étape 3 : indiquer que c'est une socket serveur

```
QUEUE_SIZE = 5 # par  
exemple  
connsock.listen(QUEUE_SIZE)
```

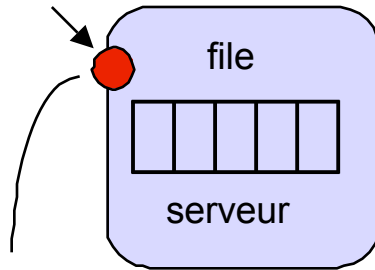
Taille max de la file d'attente des demandes

Une *socket* serveur est en attente de demandes de connexion.

- Si une demande arrive pendant qu'une autre est en cours de traitement, elle est placée dans une file d'attente.
- Si une demande arrive alors que la file est pleine, elle est rejetée (pourra être refaite plus tard) ; voir primitive connect plus loin.

Sockets côté serveur

socket serveur



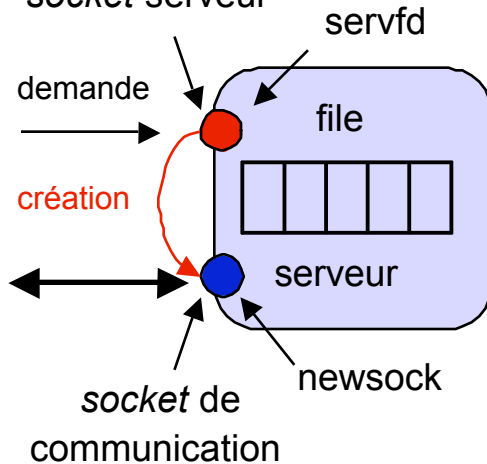
Étape 4a : se mettre en attente des demandes

```
newsock, ( addr, port ) = connsock. accept()
```

la primitive accept est **bloquante**

prête à accepter les demandes de connexion

socket serveur



Étape 4b : après acceptation d'une demande

```
newsock, ( addr, port ) = connsock. accept()
```

nouvel objet socket
Il sera utilisé pour envoyer ou
recevoir des données sur la
nouvelle connexion

adresse et port du client qui a
initié la nouvelle connexion.

Résumé des primitives pour les sockets côté serveur

```
import socket

sock = socket.socket(domain, type, protocol)
    # crée une socket client ou serveur, renvoie descripteur)

sock.bind((addr, port))
    # associe une socket à une adresse locale

sock.listen(maxqueue size)
    # déclare une socket comme serveur avec taille max queue

readsock = sock.accept()
    # met une socket serveur en attente de demandes de connexion
```

Il existe en outre une primitive comme select

Sockets côté client

On procède en 2 étapes, décrites schématiquement ci-après

On suppose que l'on connaît l'adresse d'un serveur et le numéro de port d'une *socket* serveur sur celui-ci (un processus serveur est en attente sur ce port)

Étape 1 : créer une *socket* :

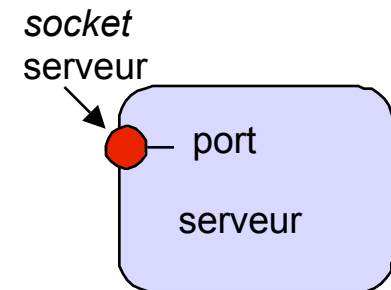
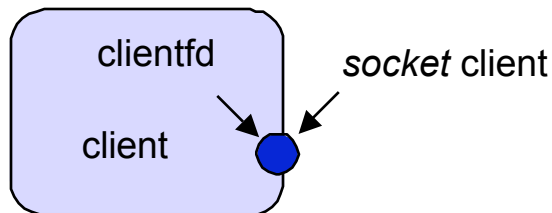
```
clisock = socket.socket(socket.AF_INET,  
                        socket.SOCK_STREAM, 0)
```

Internet

TCP

protocole,
sans usage ici

N.B. : opération
identique à la création
d'une *socket* serveur



Le serveur est en attente sur
la *socket* (accept)

Sockets côté client

Étape 2 : établir une connexion entre la socket client et le serveur

```
addr = (" www.efrei.fr" , 80)
```

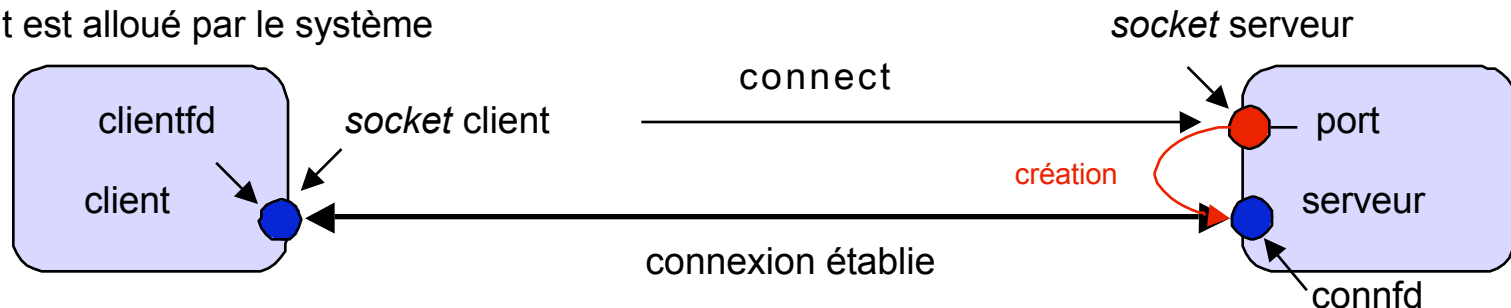
```
#remplir serveraddr avec adresse et n° port du serveur
```

```
clisock.connect(addr)
```

```
#renvoie 0 si succès, 1 si échec
```

connect envoie une demande de connexion vers la *socket* serveur

le numéro de port associé à la *socket* client est alloué par le système



Le client et le serveur peuvent maintenant dialoguer sur la connexion

Résumé des primitives pour les sockets côté client

```
import socket

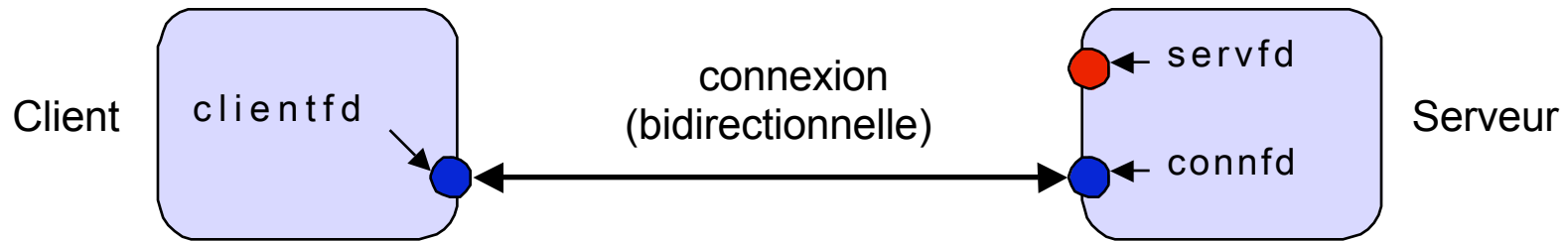
clisock = socket.socket(domain, type, protocol);
    # crée une socket client ou serveur, renvoie descripteur)

clisock.connect((addr, port))
    # envoie une demande de connexion à un serveur sur un
    # port donné
```

Échanges sur une connexion entre sockets

Une fois la connexion établie, le client et le serveur disposent chacun d'un objet socket vers l'extrémité correspondante de la connexion.

Cet objet fonctionne de façon un peu similaire à un descripteur de fichier : on peut l'utiliser pour les méthodes `recv` et `send` ; on le ferme avec `close`.



Un application client-serveur avec sockets (1)

Principes de la programmation d'une application avec sockets (les déclarations sont omises).

Côté serveur :

```
listensock = socket.socket(socket.AF_INET,socket.SOCK_STREAM,0)
listensock.bind((adrr,port))
listensock.listen(5)
# crée une socket serveur associée au numéro de port "port"
while True:
    connfd = Accept(listenfd, &clientaddr,&clientlen);
    #accepte la connexion d'un client

    # on peut communiquer avec ce client sur connfd
    server_body(connfd)    #le serveur proprement dit...
    # lit les requêtes et renvoie les réponses sur connfd
    # lorsque ce serveur se termine, on ferme la connexion
    connfd.close()
    # maintenant on va accepter la prochaine connexion
```

Un application client-serveur avec sockets (2)

Principes de la programmation d'une application avec *sockets* (les déclarations sont omises).

Côté client :

```
clisock = socket.socket(socket.AF_INET,socket.SOCK_STREAM,0)

clisock.connect( ("www.efrei.fr", port))
# ouvre une connexion vers le serveur
client_prog(clisock)
# envoyer requêtes et recevoir réponses avec clisock
# en utilisant recv et send

# à la fin des échanges, fermer le descripteur
clisock.close()
```

Un application client-serveur avec sockets (3)

Pour exécuter l'application :

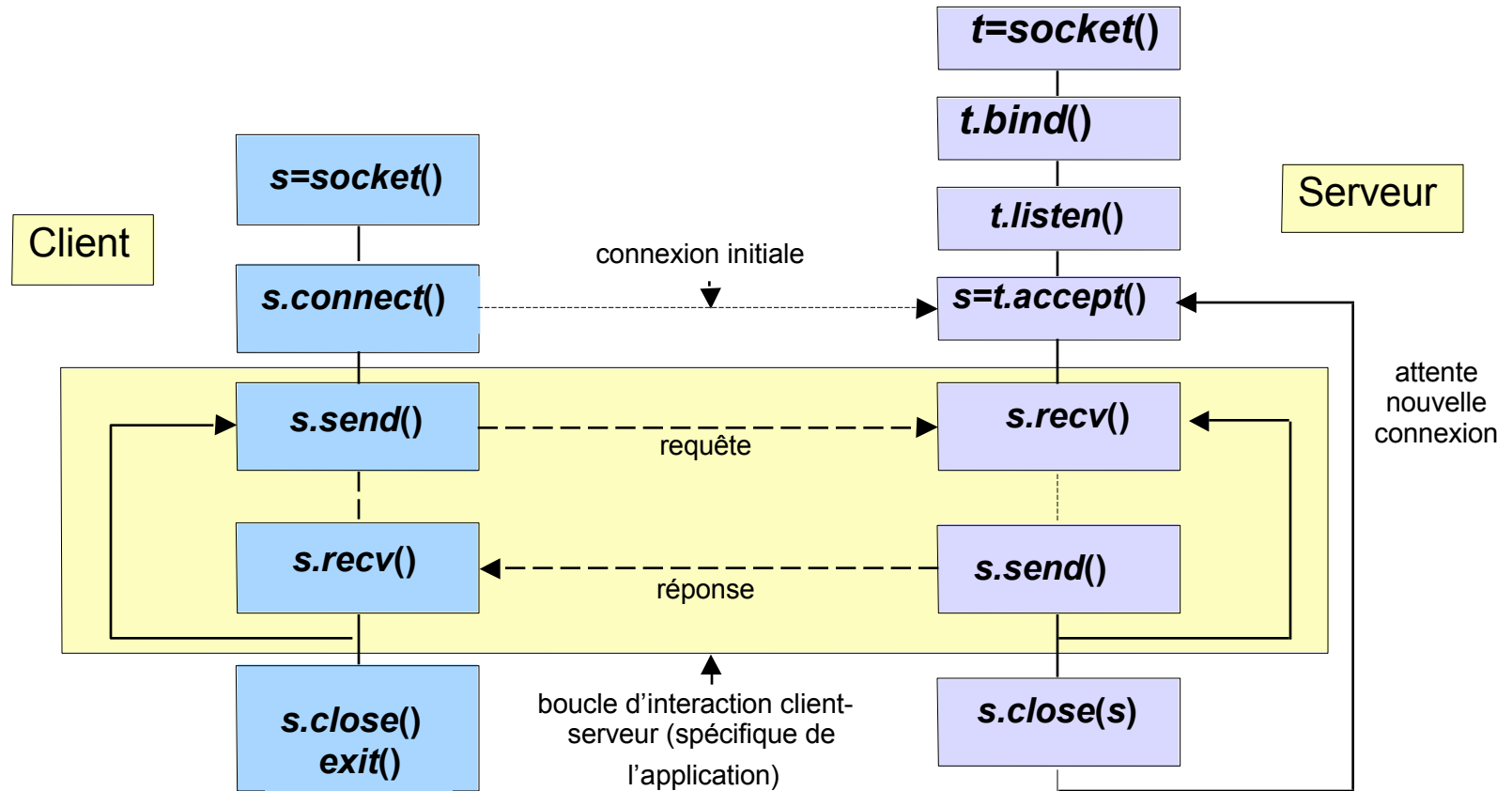
Lancer le programme serveur sur une machine, en indiquant un numéro de port (>1023 , les numéros ≤ 1023 sont réservés) ; de préférence en travail de fond

Lancer le programme client sur une autre machine (ou dans un autre processus de la même machine), en spécifiant adresse du serveur et numéro de port

N.B. On n'a pas prévu d'arrêter le serveur (il faut tuer le processus qui l'exécute). Dans une application réelle, il faut prévoir une commande pour arrêter proprement le serveur

Client-serveur en mode itératif

Les programmes précédents réalisent un serveur en **mode itératif** : un seul client est servi à la fois. Schéma :



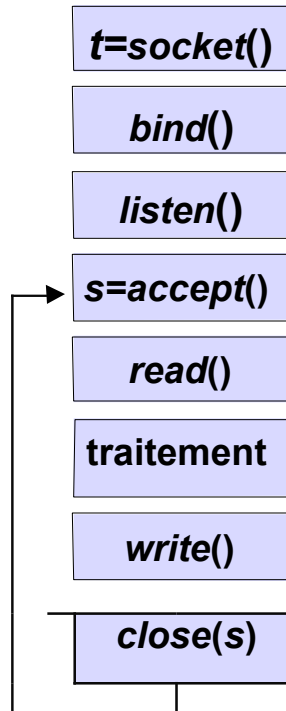
Client-serveur en mode concurrent (1)

Pour réaliser un serveur en **mode concurrent**, une solution consiste à **créer un nouveau processus** pour servir chaque demande de connexion, le programme principal du serveur ne faisant que la boucle d'attente sur les demandes de connexion.

Donc il y a un processus principal (appelé *veilleur*) qui attend sur `accept()`. Lorsqu'il reçoit une demande de connexion, il crée un processus fils (appelé *exécutant*) qui va interagir avec le client. Le veilleur revient se mettre en attente sur `accept()`. Plusieurs exécutants peuvent exister simultanément.

Client-serveur en mode concurrent (2)

Serveur itératif



Serveur concurrent

