

# OPERATING SYSTEMS

*Internals and Design Principles*



WILLIAM STALLINGS



Pearson

Ninth Edition

# Docenti



- Docente: Riccardo Lazzeretti
  - Email: [lazzeretti@diag.uniroma1.it](mailto:lazzeretti@diag.uniroma1.it)
  - Sito web: <http://www.diag.uniroma1.it/lazzeretti>
  - Ricevimento: su appuntamento via email
- Tutor: Gabriele Proietti Mattia
  - Email: [proiettimattia@diag.uniroma1.it](mailto:proiettimattia@diag.uniroma1.it)



# Il corso

- Sito web: <http://www.diag.uniroma1.it/sc2>
- Pagina classroom [classroom.google.com](https://classroom.google.com)
  - codice **ev26sbw**
- Lezioni teoriche
  - Lunedì 17:00 - 19:00
  - Mercoledì 11:00 – 13:00
  - Aula 106 Marco Polo
- Esercitazione in Laboratorio (laboratorio Via Tiburtina 205)
  - Martedì 15:00 - 19:00
  - Aula 15
- Link per lezioni online: <https://uniroma1.zoom.us/j/3583342777>

The screenshot shows the Google Classroom interface. At the top, there's a navigation bar with three horizontal dots, the text "Google Classroom", and a circular icon with the number "1". Below the navigation bar are two buttons: "Iscriviti al corso" (with the number "2") and "Crea corso". The main area is a card titled "Codice corso" with the sub-instruction "Chiedi il codice del corso all'insegnante e inseriscilo qui.". A text input field contains the code "ev26sbw", which is highlighted with a red border and the number "3".

# Temi principali del corso

- Processi, thread, concorrenza
- Il sistema operativo
- Reti di Calcolatori
- Inter-process communication
- Sistemi distribuiti
- Sicurezza informatica



# Materiale didattico

- Slides e materiale aggiuntivo scaricabili dalla pagina classroom
- Esercitazioni in lab scaricabili da classroom
- Libri di testo (se volete approfondire):
  - W. Stallings: "Operating Systems: Internals and Design Principles" (ninth edition), Pearson
  - G. Coulouris, J. Dollimore, T. Kindberg, G. Blair: "Distributed Systems: Concepts and Design" (fifth edition), Pearson
  - W. Stallings: "Cryptography and Network Security: Principles and Practice" (seventh edition), Pearson

# Esame SC2

- Modalità di esame: Prova unica al calcolatore al fine di verificare
  - Conoscenza della teoria
    - *Domande di teoria*
    - *Esercizi di logica*
  - Capacità di programmazione
    - *Sviluppo codice C su programmazione concorrente, sincronizzazione e comunicazione inter-processo*
  - **Viene attribuito un voto ad entrambe le parti in 33esimi**
  - **Voto finale: media dei due (arrotondata per eccesso)**
  - **È necessario raggiungere almeno 16 in ciascuna delle due parti**
  - **Codice che non risulta compilabile non verrà corretto**
  - **L'esame si svolge in laboratorio**
    - per motivazioni legate al covid può essere svolto in modalità telematica, ma prevede una prova orale aggiuntiva

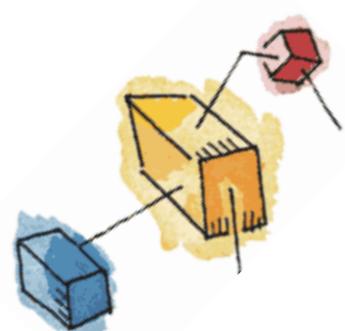
# Processes and Threads

The present slides are mainly adapted from «*Operating Systems: Internals and Design Principles*» 6/E by William Stallings (Chapter 4). Some materials are obtained from the POSIX threads Programming tutorial by Blaise Barney.

*Sistemi di Calcolo 2*

*Instructor: Riccardo Lazzeretti*

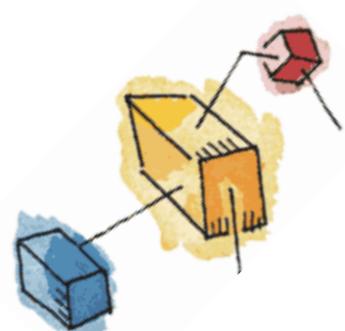
*Special thanks to: Daniele Cono D'Elia, Leonardo Aniello, Roberto Baldoni*



# Roadmap

- 
- Processes: fork(), wait()
  - Threads: resource ownership and execution
  - Case study:
    - Pthreads → *libreria multi-threads*
  - Symmetric multiprocessing (SMP)

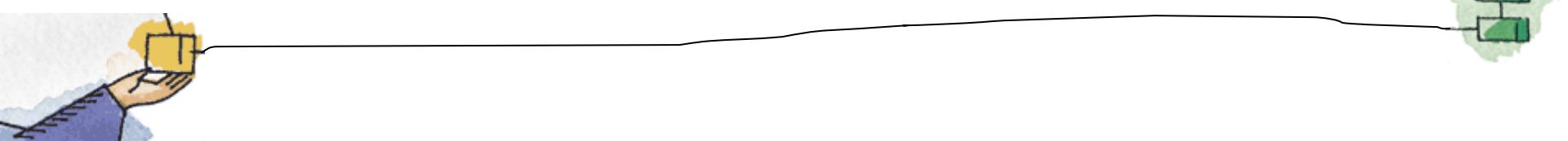


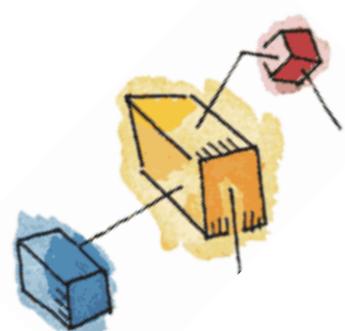


# Role of Processes

- Most requirements that an OS must meet can be expressed w.r.t. processes:
  - Interleaved execution (*esecuzione interrotta*)
  - Resource allocation and policies
  - User creation of processes and inter-process communication

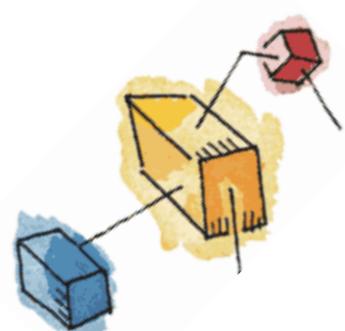
→ metodi per creare processi e la loro comunicazione





# Process Elements

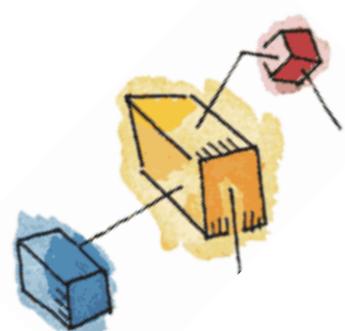
- (COMPOJTO)
- A process is comprised of:
    - Program code (possibly shared)
    - A set of data
    - A number of attributes describing the state of the process *during execution*



# Process Elements

- While the process is running it has a number of elements including
  - Identifier (numero del processo)
  - State (running, idle, terminato, ...)
  - Priority (priorità su altri processi)
  - Program counter (la posizione dell'istruzione da eseguire)
  - Memory pointers
  - Context data (il valore che c'è all'interno dei registri)
  - I/O status information
  - Accounting information





# Process Control Block

- Contains the process elements
- Created and managed by the operating system (OS)
- Allows support for multiple processes

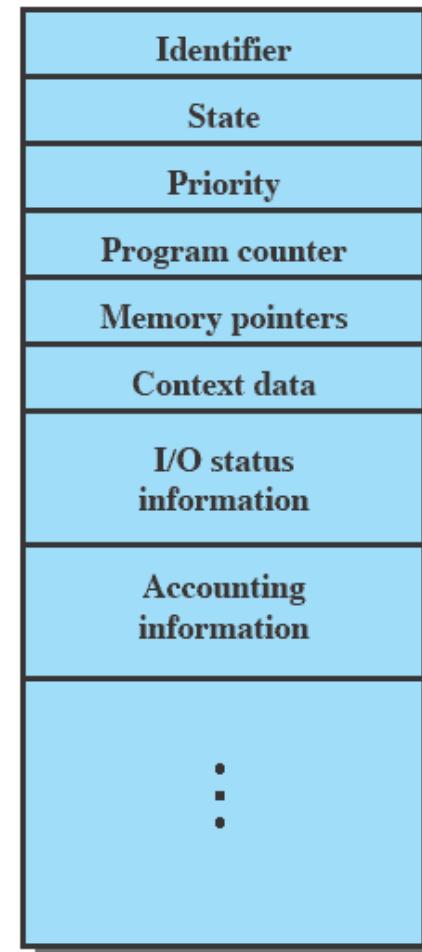
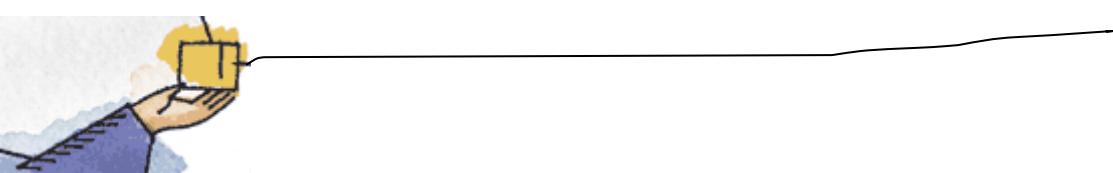


Figure 3.1 Simplified Process Control Block



# Unix system calls

## Creating new Processes

---

`fork( ) - wait( ) - exit( )`

*Credits: Mirela Damian, Allan Gottlieb*

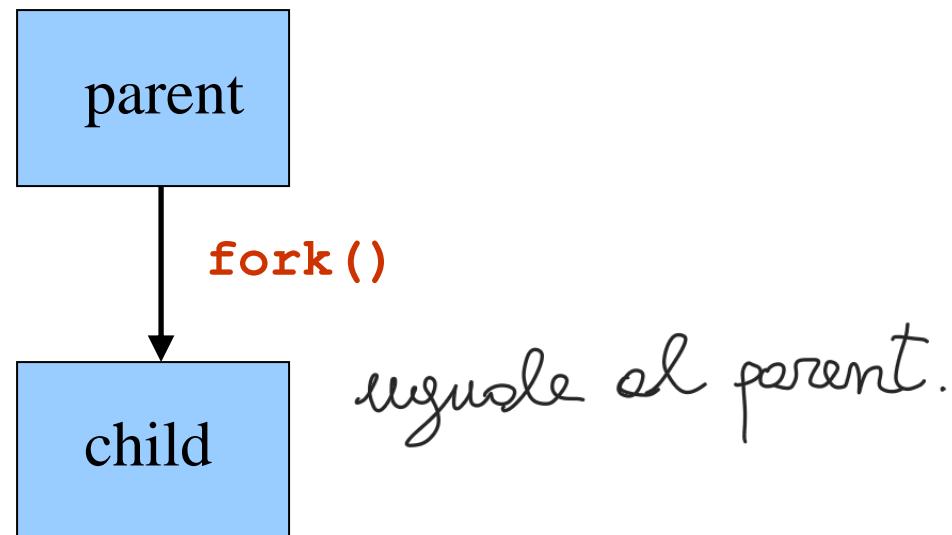
*Class notes: <https://cs.nyu.edu/~gottlieb/courses/os202/class-notes.html>*

# How To Create New Processes?

---

## Underlying mechanism

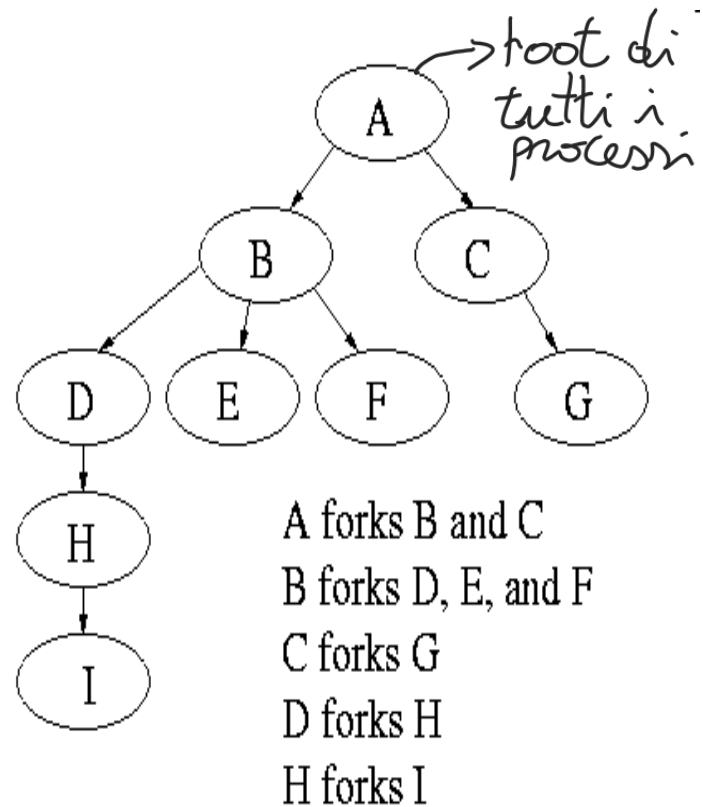
- A process runs **fork** to create a child process
- Parent and children execute concurrently
- Child process is a duplicate of the parent process



# Process Creation

---

- After a **fork**, both parent and child keep running, and each can fork off other processes.
- A **process tree** results. The root of the tree is a special process created by the OS during startup.
- A process can *choose* to wait for children to terminate. For example, if C issued a **wait()** system call, it would block until G finished.



# Bootstrapping

---

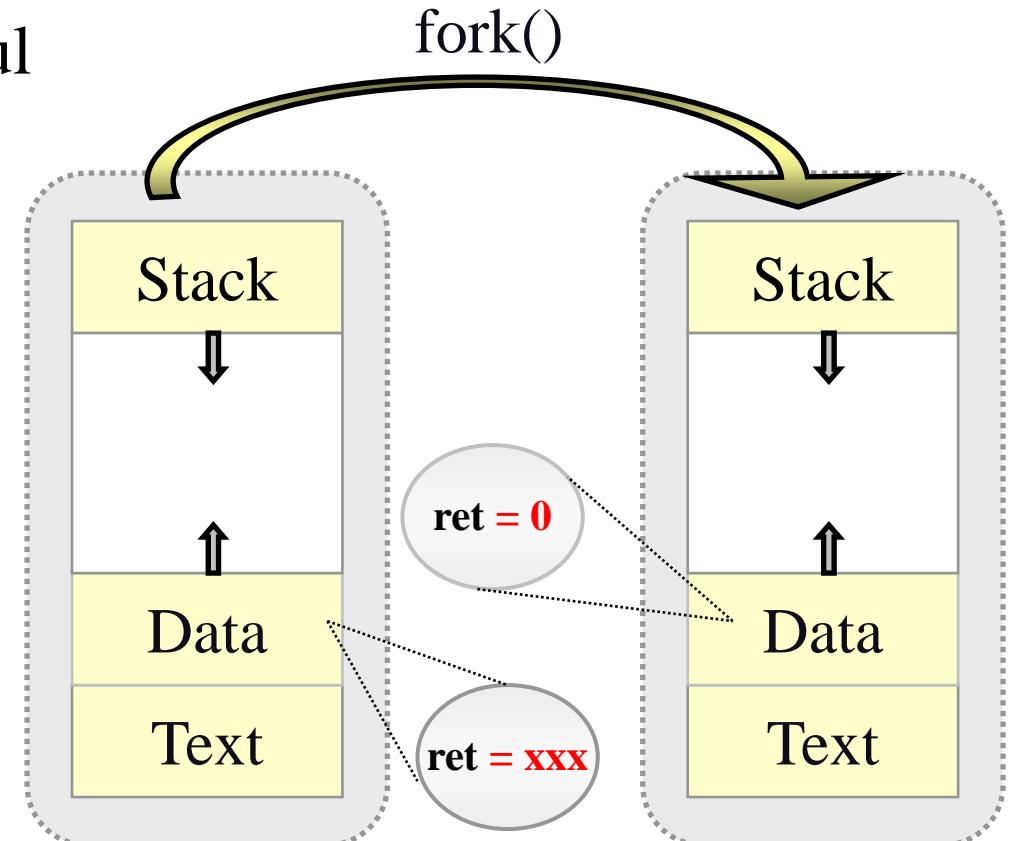
- When a computer is switched on or reset, there must be an initial program that gets the system running
- This is the bootstrap program
  - Initialize CPU registers, device controllers, memory
  - Load the OS into memory
  - Start the OS running
- OS starts the first process (such as “init”)
- OS waits for some event to occur
  - Hardware interrupts or software interrupts (traps)

*▼ padre di tutti  
i processi.*

# Fork System Call

- Current process split into 2 processes: parent, child
- Returns -1 if unsuccessful
- Returns 0 in the child
- Returns the child's identifier in the parent

→ One is child  
int id = fork()  
→ ≠ One is parent

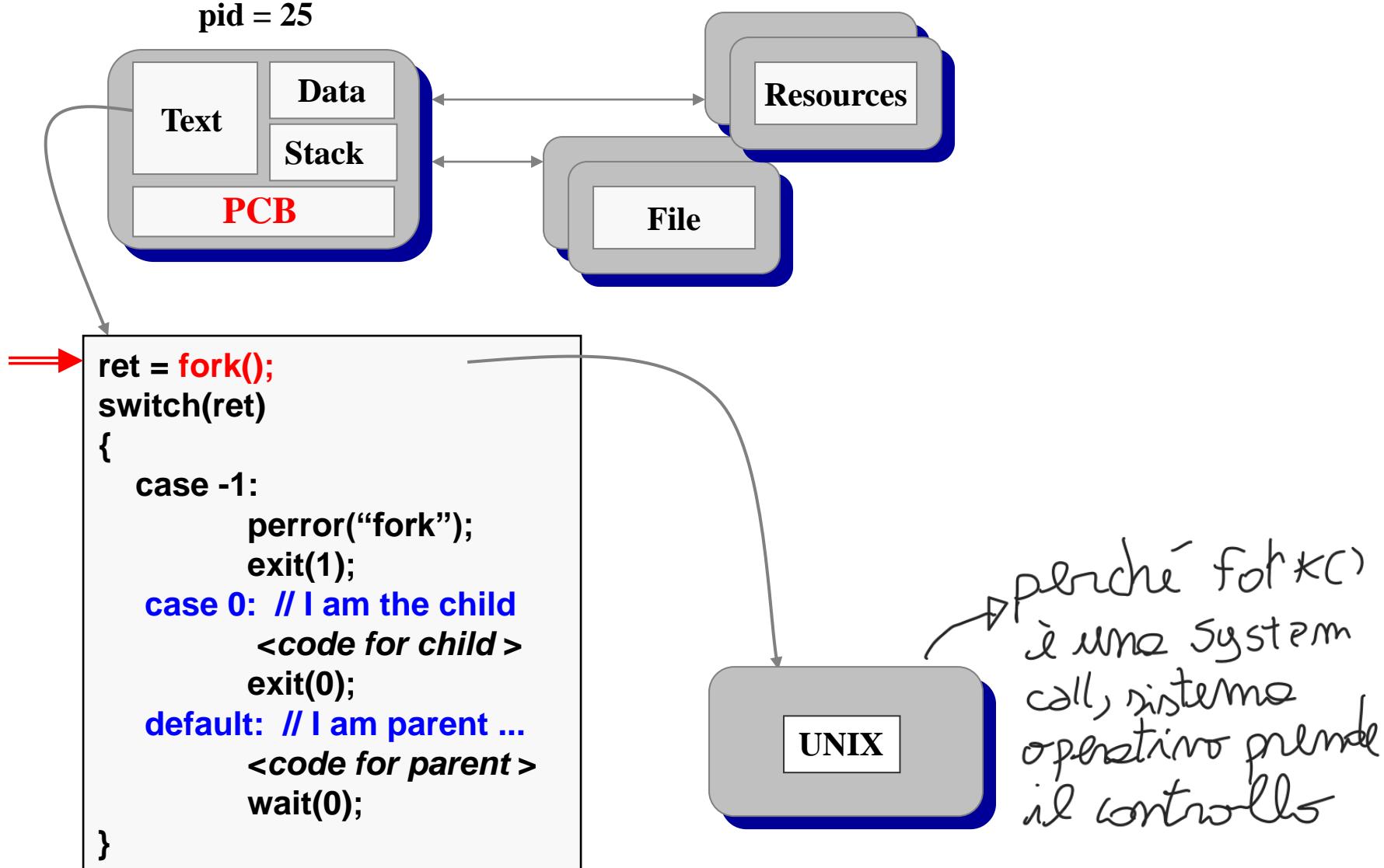


# Fork System Call

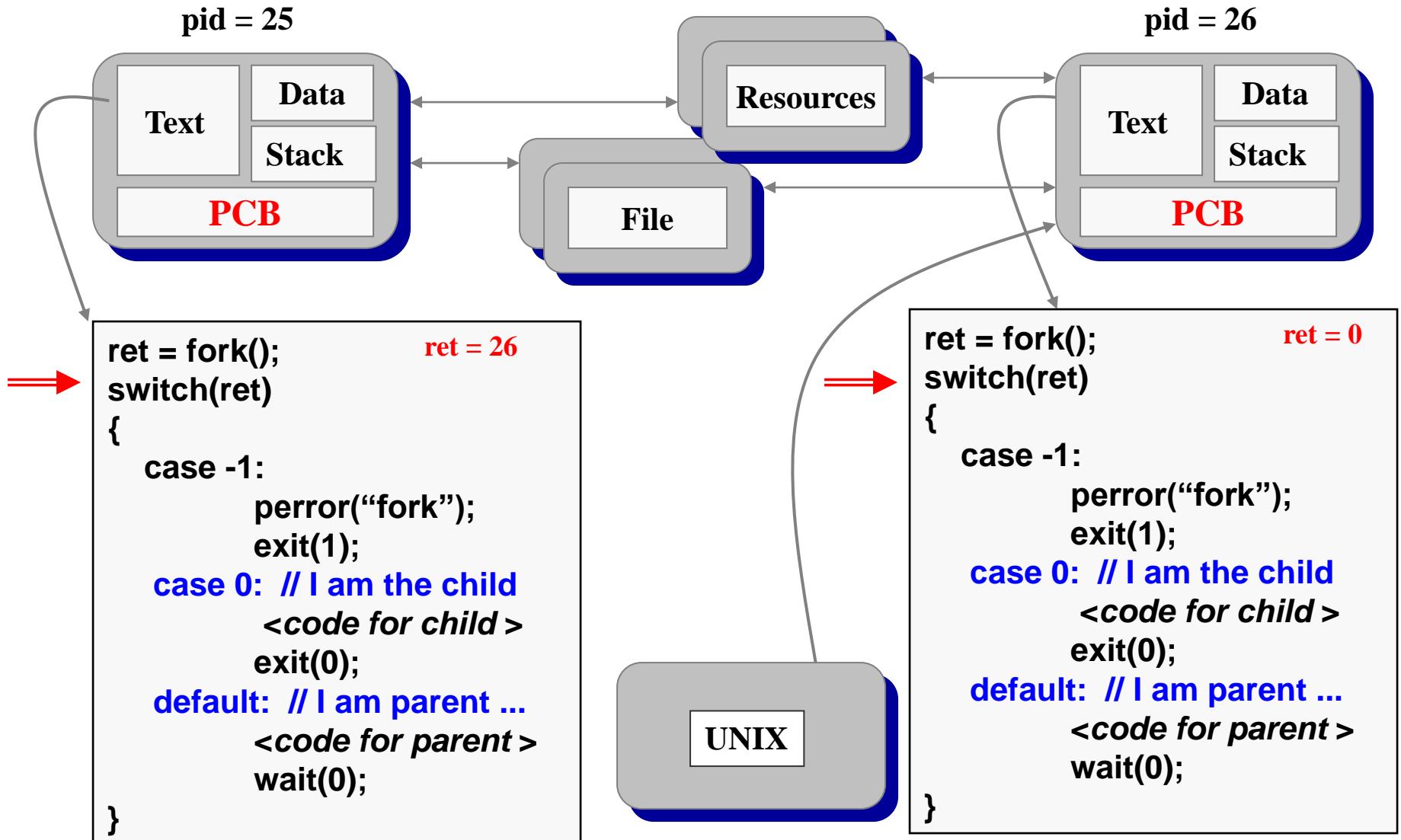
---

- The child process inherits from parent
  - identical copy of memory ( $\text{child} \approx \text{parent}$ )
  - CPU registers
  - all files that have been opened by the parent
- Execution proceeds **concurrently** with the instruction following the fork system call *tipicamente*  $\text{if}(id == 0) \{ \text{codice child} \}$   
 $\text{else} \{ \text{codice parent} \}$
- The execution context (PCB) for the child process is a copy of the parent's context at the time of the call

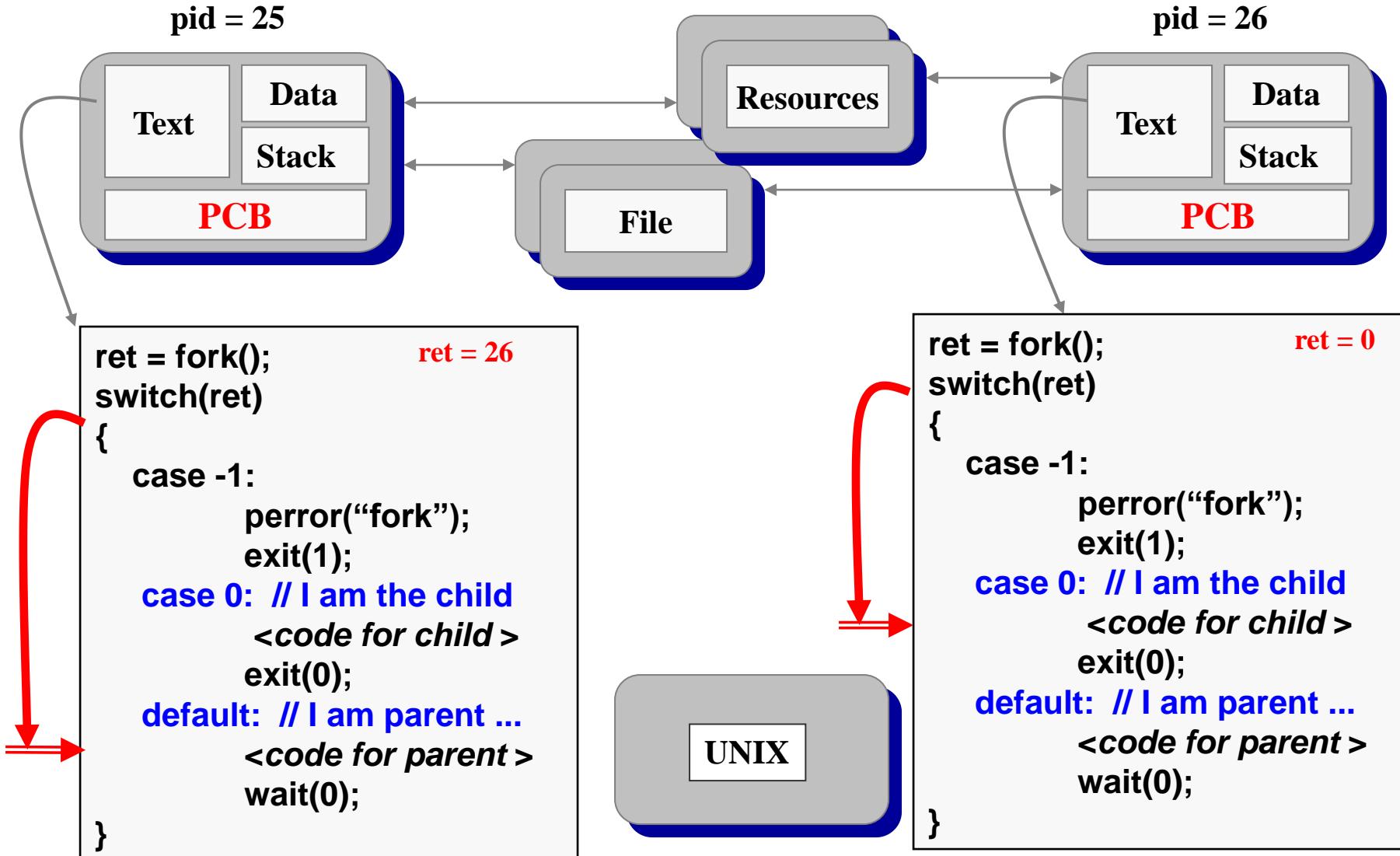
# How fork Works (1)



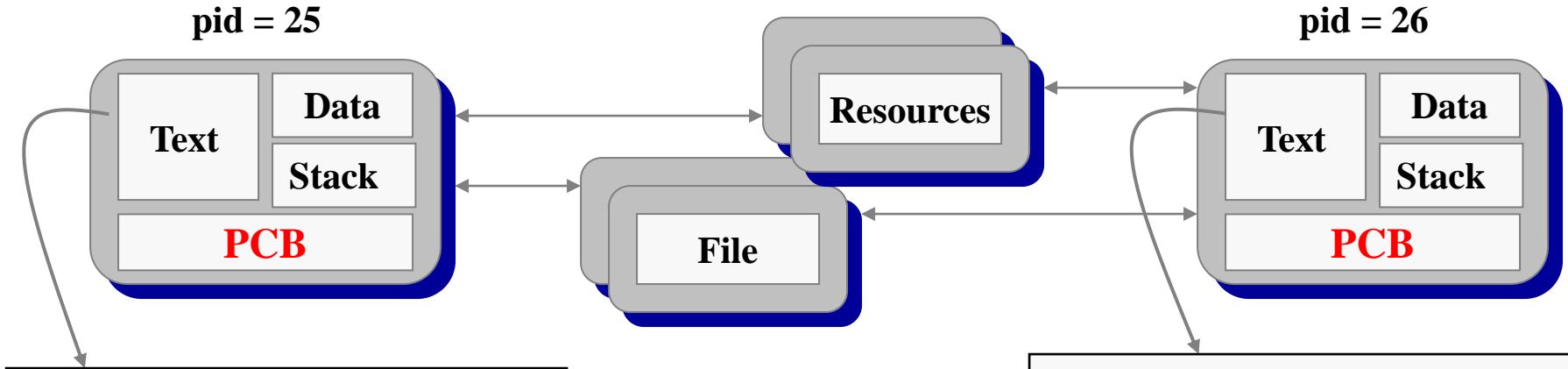
# How fork Works (2)



# How fork Works (3)

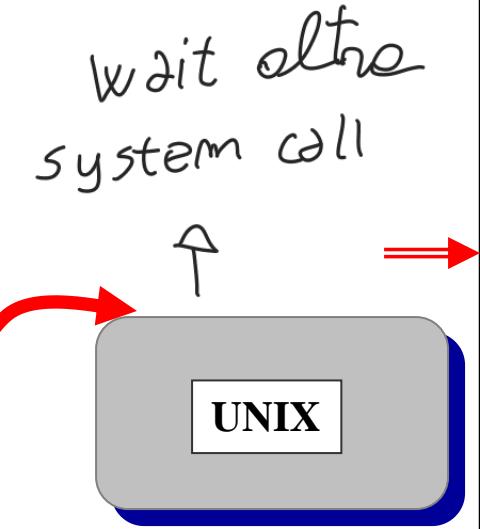


# How fork Works (4)

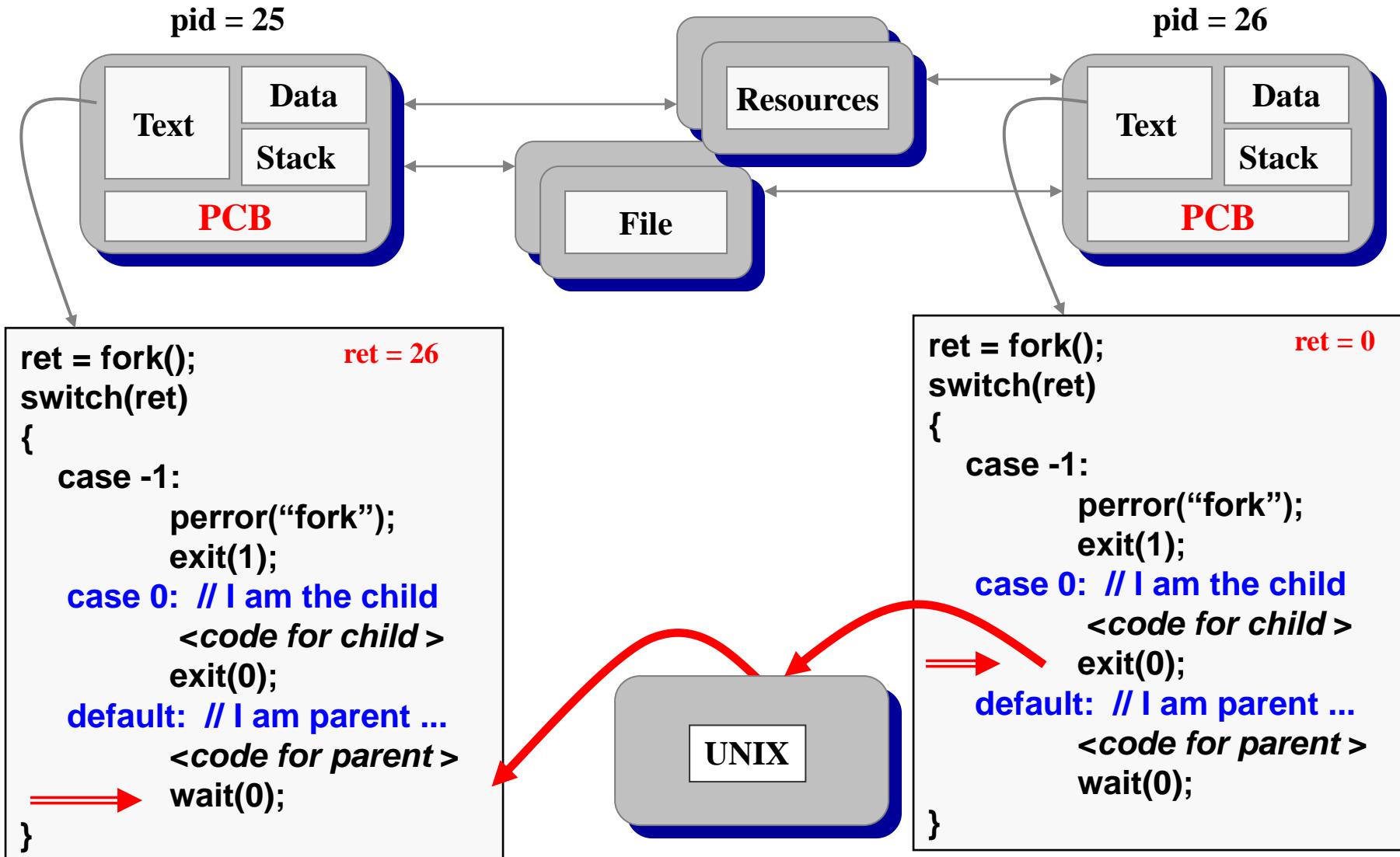


```
ret = fork();
switch(ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0: // I am the child
        <code for child>
        exit(0);
    default: // I am parent ...
        <code for parent>
        wait(0);
}
```

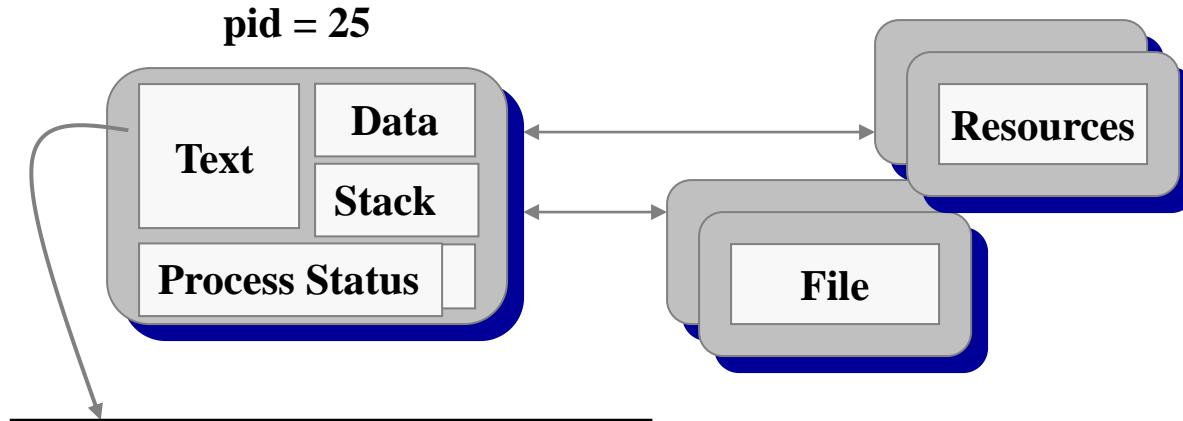
```
ret = fork();
switch(ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0: // I am the child
        <code for child>
        exit(0);
    default: // I am parent ...
        <code for parent>
        wait(0);
}
```



# How fork Works (5)



# How fork Works (6)



```
ret = fork();           ret = 26
switch(ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0: // I am the child
        <code for child>
        exit(0);
    default: // I am parent ...
        <code for parent>
        wait(0);
    < ... >
    ======>
```



# Execution of a program

---

- If the fork is used to create a new process that must execute a program we use the exec() command
- When executed, the Operating System replaces the current process image (text, data, stack) with a new process image
- The new program must have a main()
- There is no return

In this course we assume that the child is executing a code defined in the parent program

```
ret = fork();
switch(ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0: // I am the child
        exec*( .... )
    default: // I am parent ...
        <code for parent>
        wait(0);
        < ... >
```

# Orderly Termination: exit()

---

- To finish execution, a child may call **exit(status)**
- This system call:
  - Saves result = argument of exit (*status*)
  - Executes all functions specified with **atexit(fun)** and **on\_exit(fun)**
    - ↳ fa parte della stdlib
  - Streams are downloaded with **fflush()**
  - Closes all open files, connections
    - (not the ones shared with other processes)
      - ↳ se padre e figlio hanno aperto un file, va chiuso per entrambi.
  - Call **\_exit(status)**

# Orderly Termination: `_exit()`

---

- To finish execution, a child may call `_exit(status)`
- This system call:
  - Saves result = argument of exit (`status`)
  - Deallocates memory
  - If the process has running childs, they are assigned to init
  - Checks if parent is alive
  - If parent is alive, holds the result value until the parent requests it (with `wait`); in this case, the child process does not really die, but it enters a zombie/defunct state
  - If parent is not alive, the child terminates (dies)

# Waiting for the Child to Finish

---

- Parent may want to wait for children to finish
  - Example: a shell waiting for operations to complete
- Waiting for any some child to terminate: **wait()**
  - Blocks until some child terminates (*qualsiasi figlio*)
  - Returns the process ID of the child process
  - Or returns -1 if no children exist (i.e., already exited)
- Waiting for a specific child to terminate: **waitpid()**
  - Blocks till a child with particular process ID terminates

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

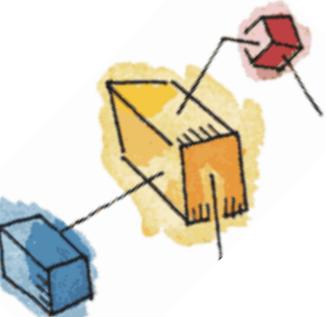
# Processes with Python

SALTARE!

- Process creation is managed by operating system
- Python must be able to ask the OS to do it
  - `import os`
- Commands are similar
  - `pid = os.fork()`
  - `os.wait()`
  - `os.waitpid()`
  - `os.exit()`
  - `os._exit()`

```
import os

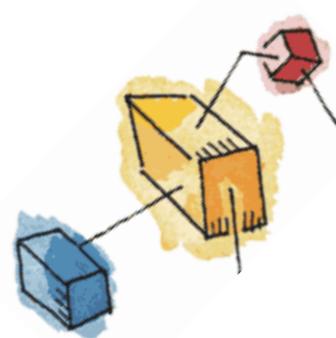
ret = os.fork()
if ret==0:
    # I am the child
    <code for child>
    os._exit(0)
else:
    # I am parent ...
    <code for parent>
    os.wait(0);
    < ... >
```



# Roadmap

- Processes: fork(), wait()
- • Threads: resource ownership and execution
- Case study:
  - PThreads
- Symmetric multiprocessing (SMP)



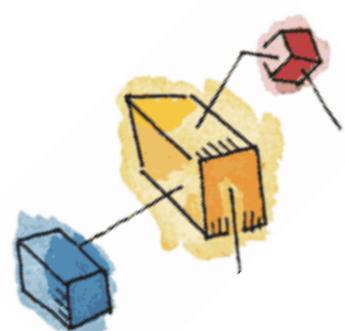


# Processes and Threads

↳ è un flusso di esecuzione

- A process has two characteristics:
  - **Scheduling/execution** - follows an execution path that may be interleaved with other processes (*sequenza di esecuzione*)
  - **Resource ownership** - includes a virtual address space to hold the process image (*entità a cui vengono assegnate delle risorse*)
- These two characteristics are treated independently by the operating system



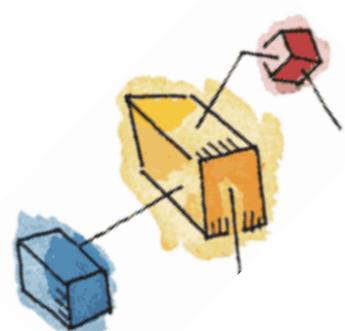


# Processes and Threads

- The unit of dispatching is referred to as a **thread** or lightweight process
- The unit of resource ownership is referred to as a process or **task**

La sequenza di esecuzione ci si riferisce a **THREAD**





# Multithreading

- The ability of an OS to support multiple, concurrent paths of execution within a single process.

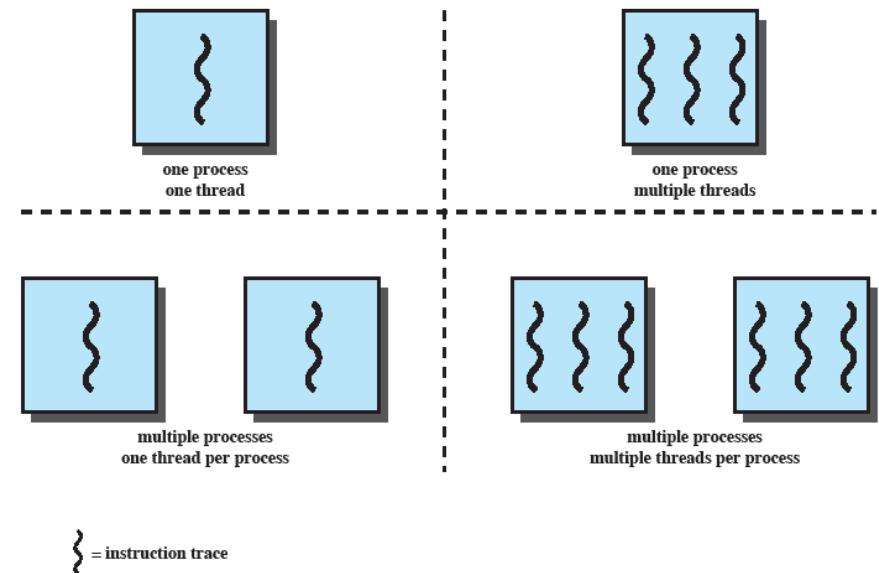
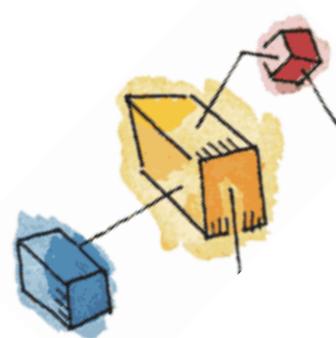


Figure 4.1 Threads and Processes [ANDE97]



# Single Thread Approaches

- MS-DOS supports a single user process and a single thread
- Some UNIX support multiple user processes but only support one thread per process

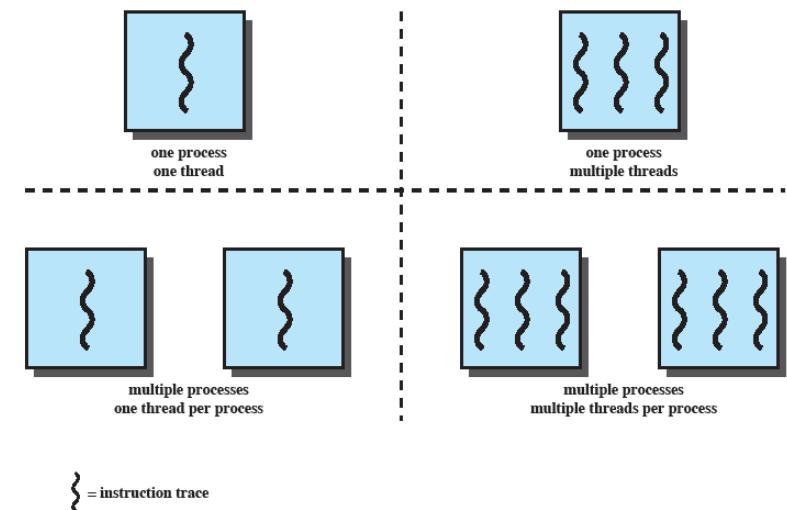
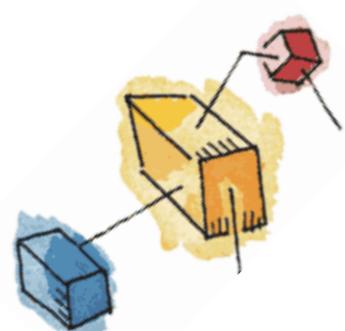
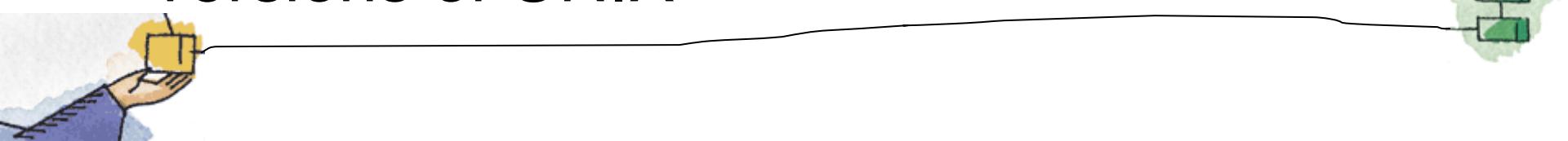


Figure 4.1 Threads and Processes [ANDE97]





# Multithreading

- Often a Java run-time environment is a single process with multiple threads
  - Multiple processes **and** threads are found in Windows, Solaris, and many modern versions of UNIX
- 

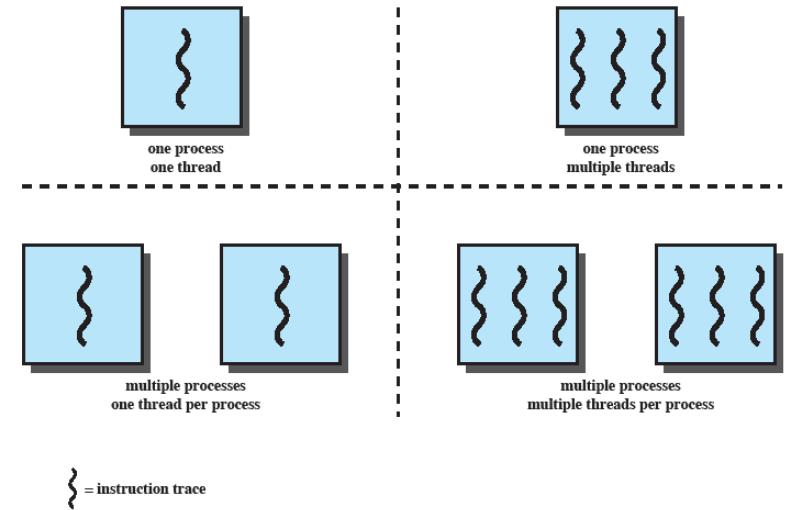
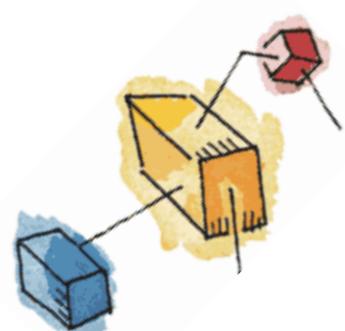


Figure 4.1 Threads and Processes [ANDE97]





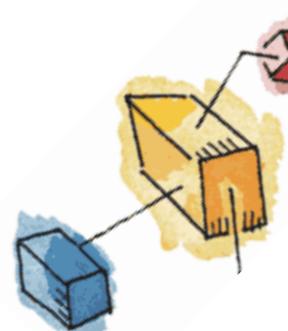
# Processes in Multithreaded OS

- A virtual address space which holds the process image
- Protected access to
  - Processors
  - Other processes
  - Files
  - I/O resources

PROCESSI : oggetti  
che usano memoria

THREAD : hanno una  
memoria condivisa.





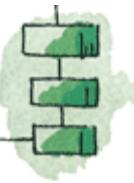
# One or More Threads in Process

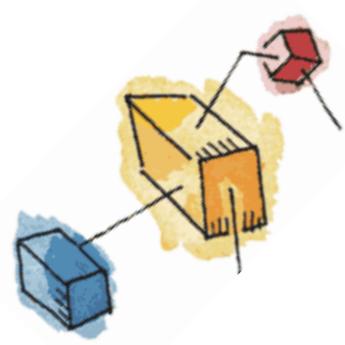
- Each thread has
  - An execution state (running, ready, etc.)
  - Saved thread context when not running
  - An execution stack (come le funzioni)
  - Some per-thread static storage for local variables
  - Access to the memory and resources of its process (all threads of a process share this)

simile a quelli  
dei processi

→ REGISTRI  
DEL THREAD

↳ memorie globali accessibile  
a tutti i THREAD!

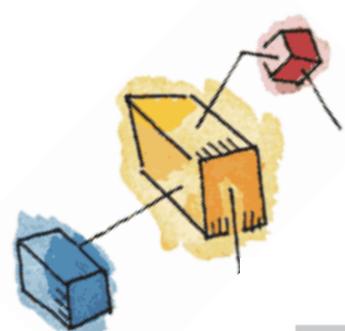




# One view...

- One way to view a thread is as an independent program counter operating *within* a process





# Threads vs. processes

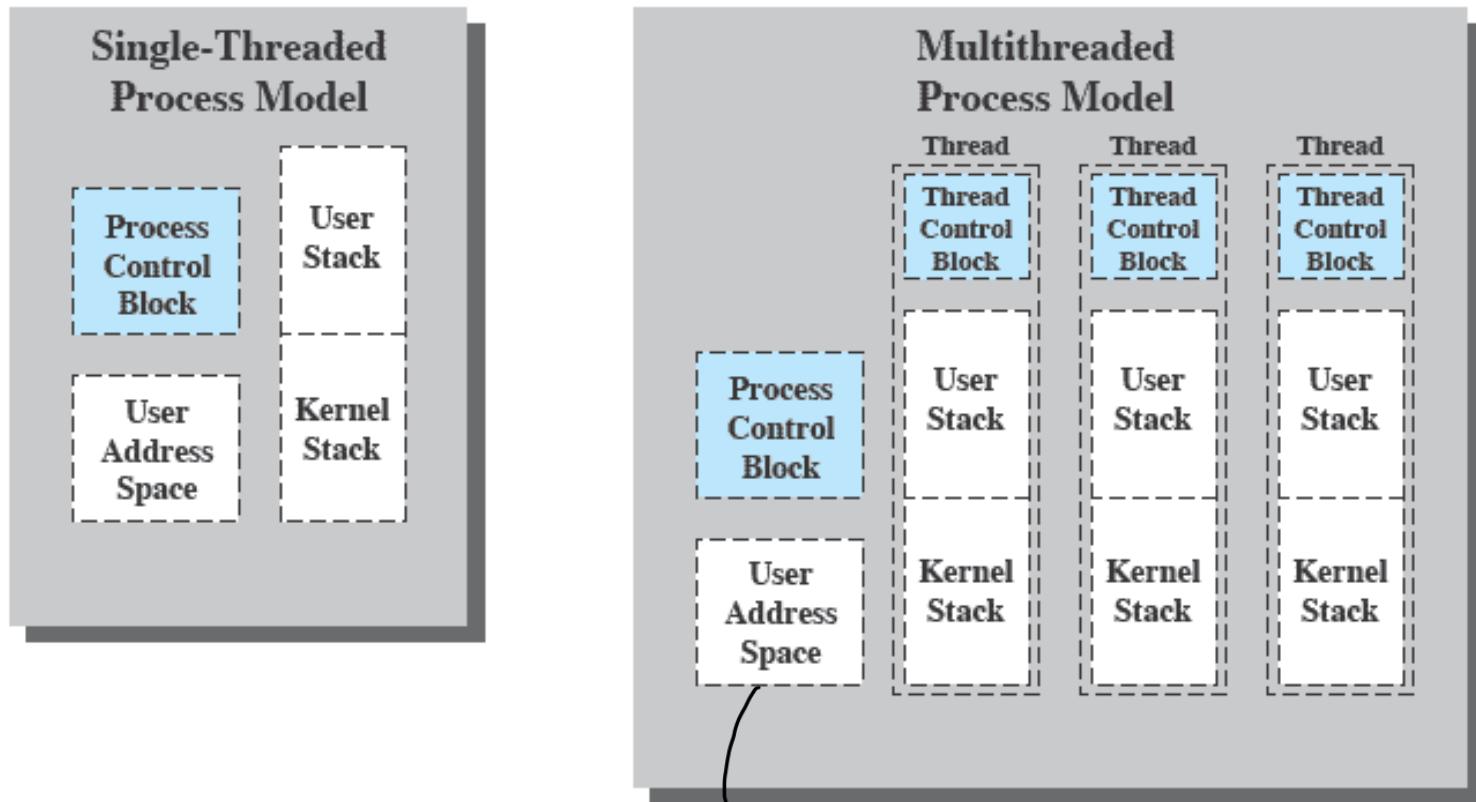
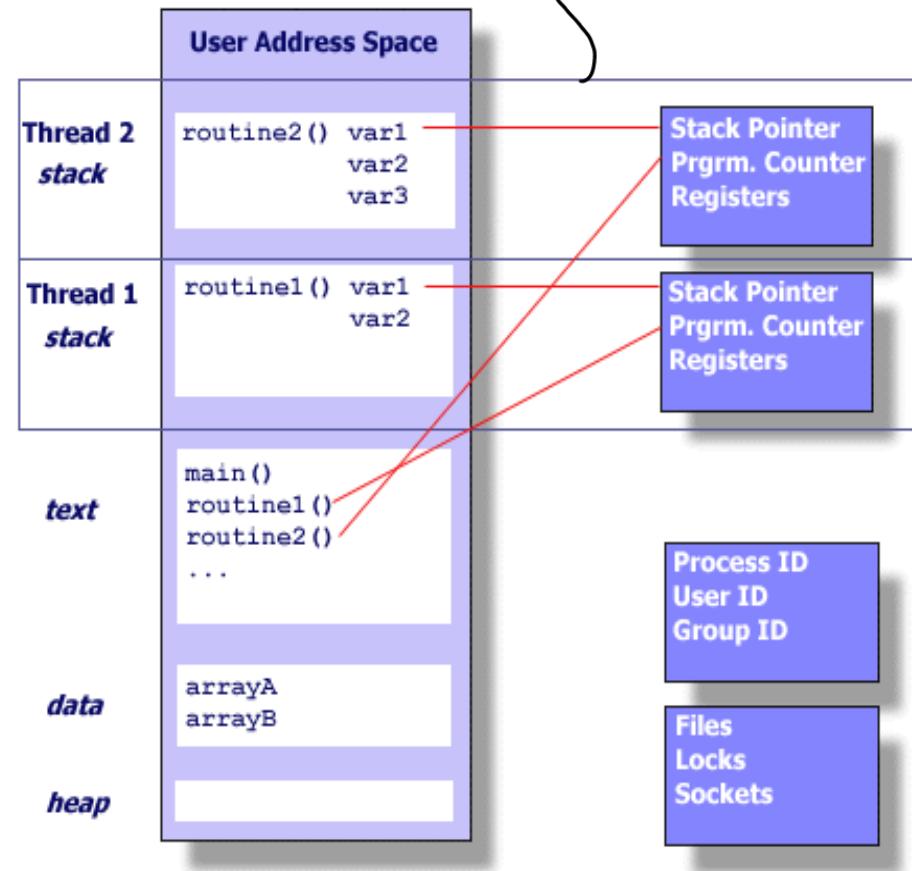
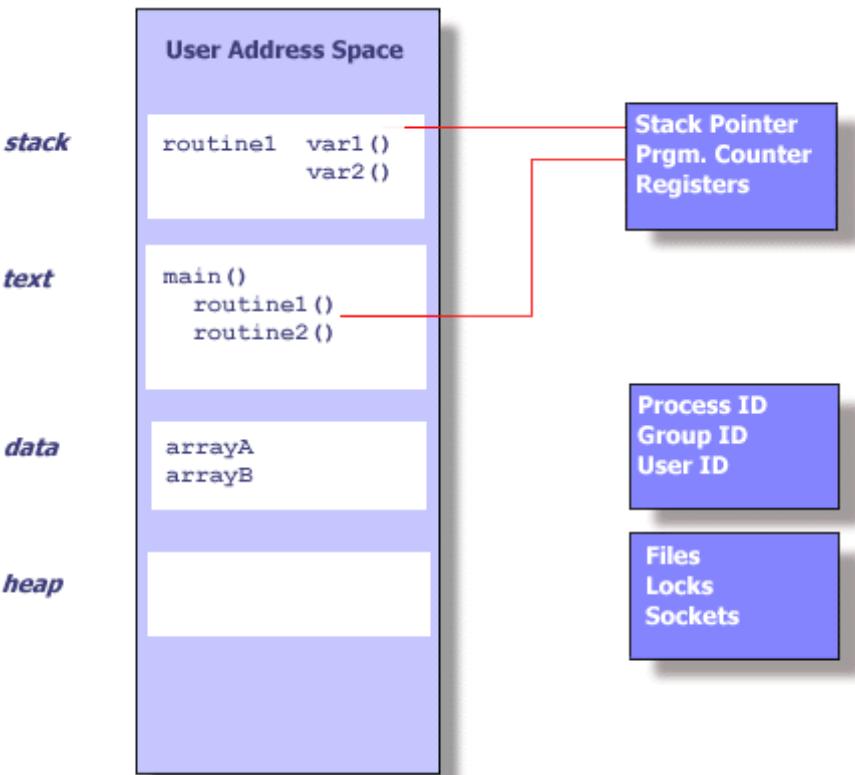
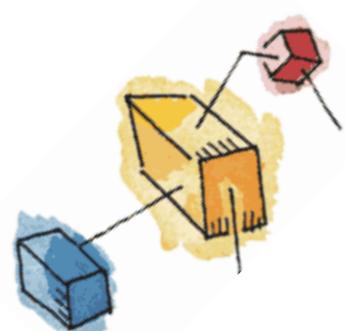


Figure 4.2 Single Threaded and Multithreaded Process Models

# Unix Process vs thread

stack local  
oh agni thread

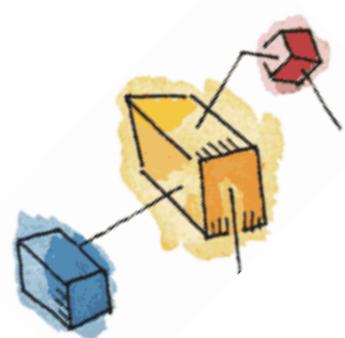




# Benefits of Threads

- Takes less time to create or terminate a new thread than a process
- Switching between two threads takes less time than switching processes
- Threads can communicate with each other
  - without invoking the kernel

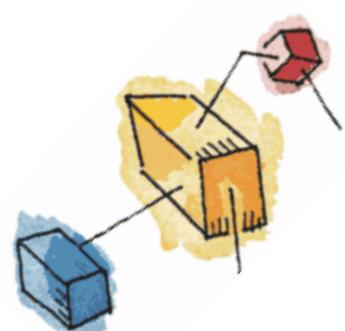




# Thread use in a Single-User System

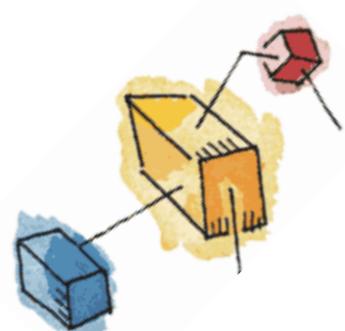
- Foreground and background work
- Asynchronous processing
- Speed of execution
  - e.g., execution advances while a thread waits for I/O
- Modular program structure





# Threads

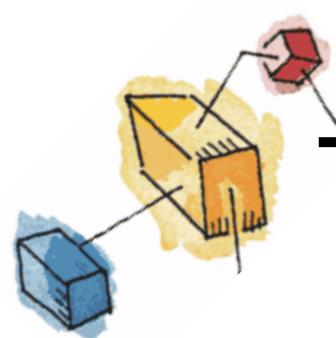
- Several actions can affect all of the threads in a process
  - OS must manage these at the process level  
*(non hanno direttamente con i thread)*
- Examples:
  - Suspending a process involves suspending all threads of the process (*same address space!*)
  - Termination of a process terminates all threads within the process



# Activities similar to Processes

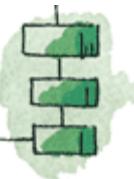
- Threads have execution states and may synchronize with one another
  - Similar to processes
- We look at these two aspects of thread functionality in turn
  - States
  - Synchronisation

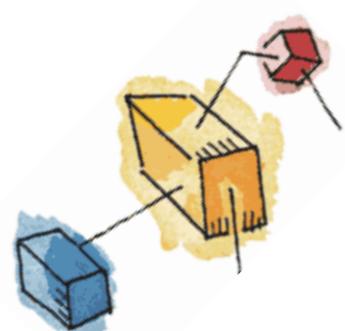




# Thread Execution States

- States associated to threads:
  - Running, ready, blocked (membri dei processi)
- To change the thread state
  - Spawn (another thread)
  - Block
    - Issue: can blocking a thread result in blocking some other thread, or even the whole process?
  - Unblock
  - Finish (thread)
    - Deallocate register context and stacks

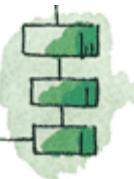


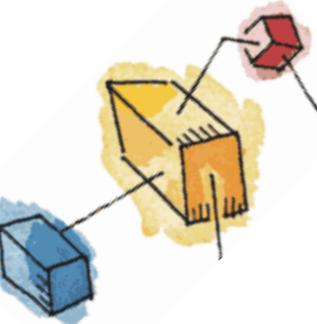


# Example: Remote Procedure Call

- Consider:
  - A program that performs two remote procedure calls (RPCs)
  - to two different hosts
  - to obtain a combined result.

RPC : funzioni che risiedono da tutt'oltre  
parte (Es. : in un server)

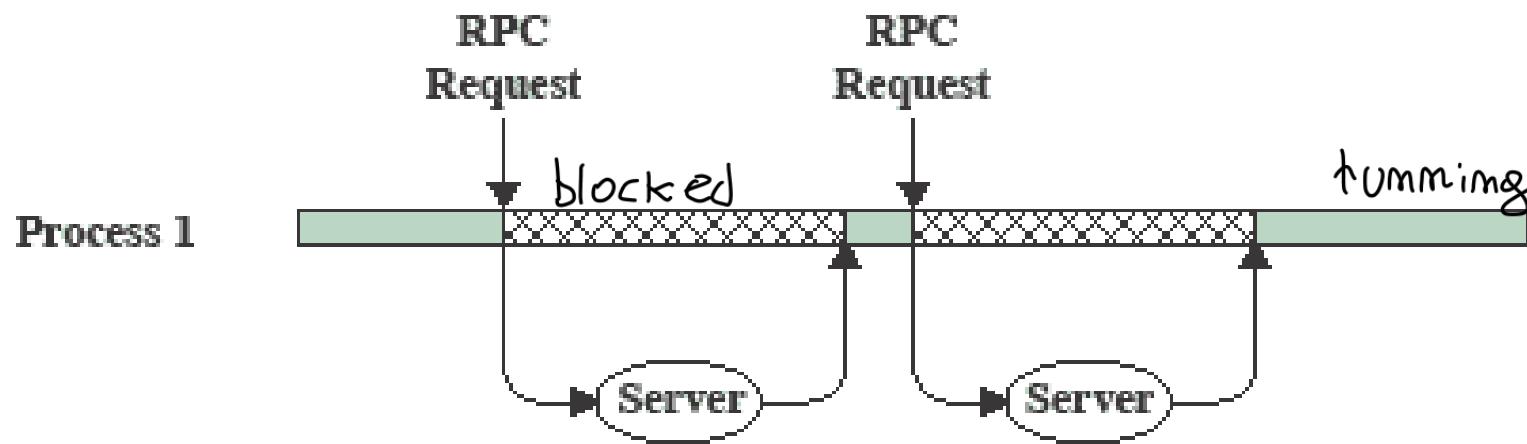




# RPC

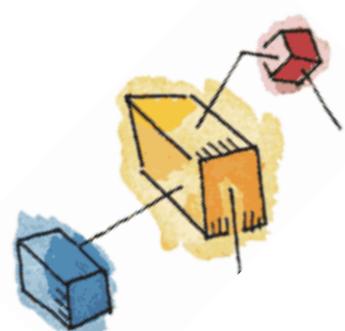
## Using Single Thread

Time →

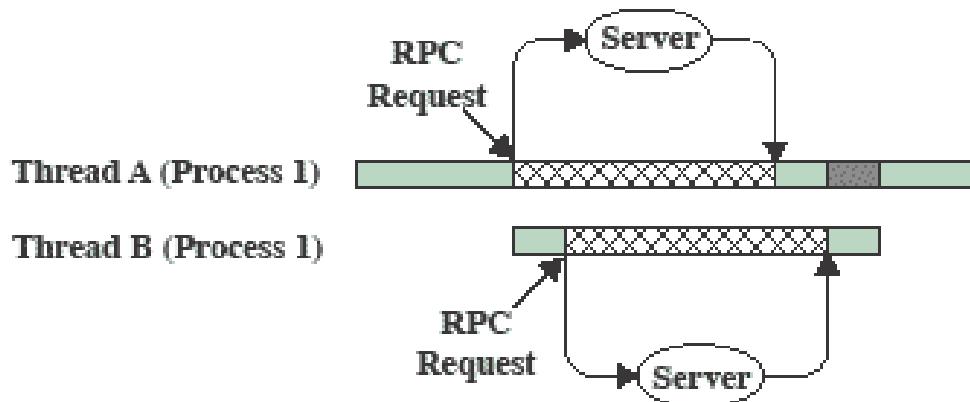


(a) RPC Using Single Thread





# RPC Using One Thread per Server



→ potrebbe  
essere anche  
eseguita prima A

(b) RPC Using One Thread per Server (on a uniprocessor)

██████ Blocked, waiting for response to RPC

█████ Blocked, waiting for processor, which is in use by Thread B

███ Running

N.B.: si supponendo di avere un solo processore



# Multithreading on a Uniprocessor

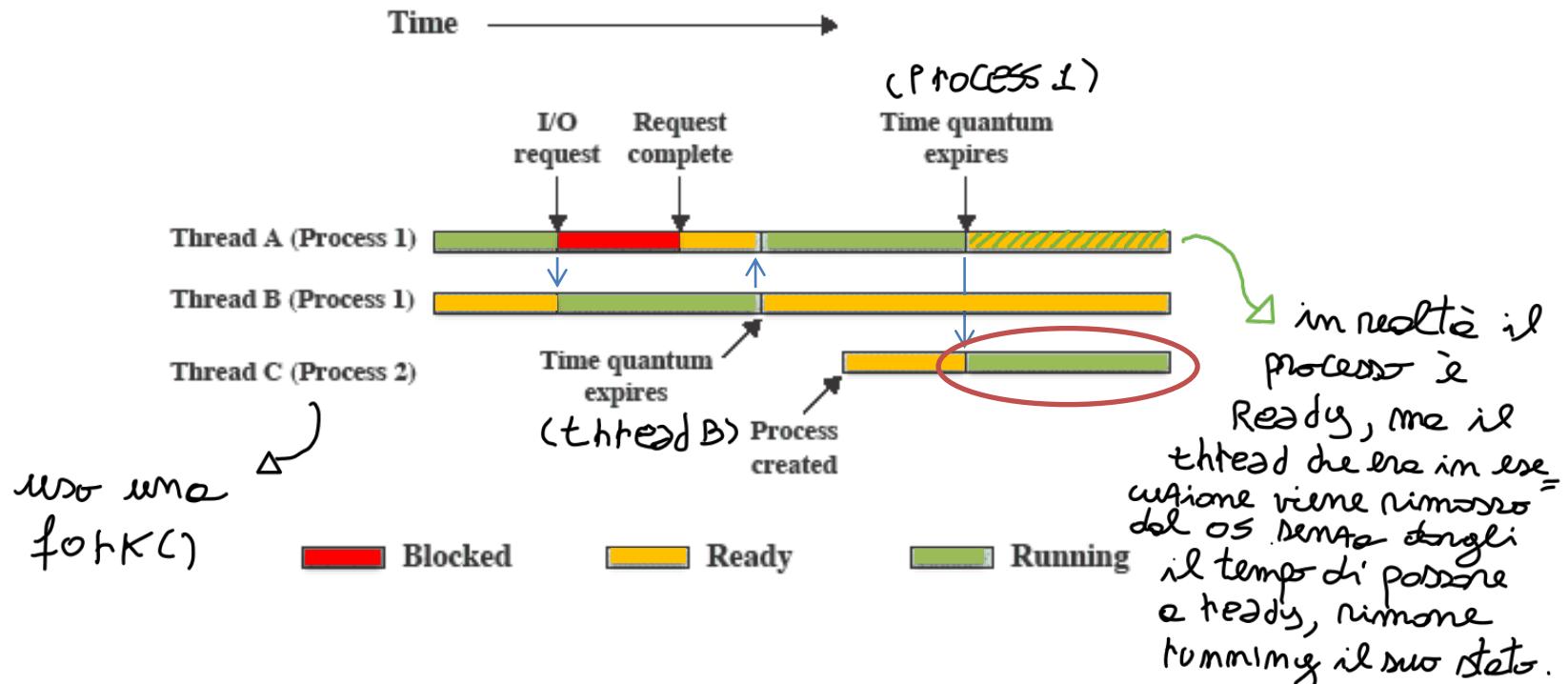
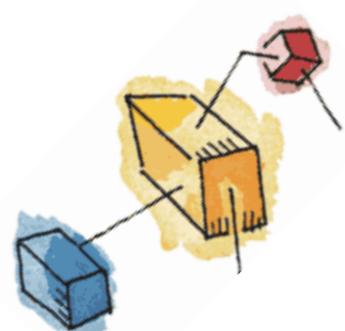


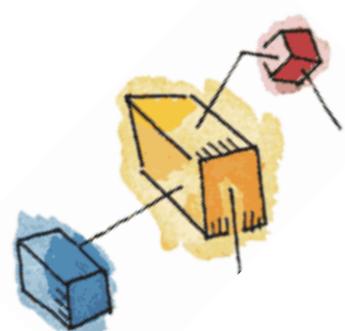
Figure 4.4 Multithreading Example on a Uniprocessor



# Categories of Thread Implementation

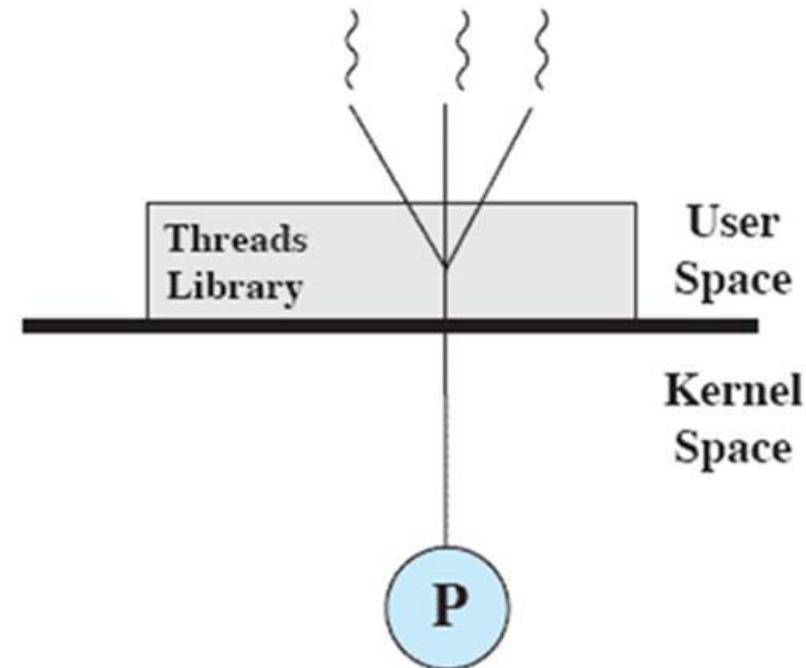
- User Level Thread (ULT) gestite a livello d'utente, non chiamate mai il Kernel (veloci)
- Kernel level Thread (KLT) also called:
  - kernel-supported threads
  - lightweight processes

→ potrebbe usare più core per i thread (come per i processi), non solamente uno come per ULT.



# User-Level Threads

- All thread management is done by the application
- The kernel is not aware of the existence of threads



(a) Pure user-level



# Relationships between ULT Threads and Process States

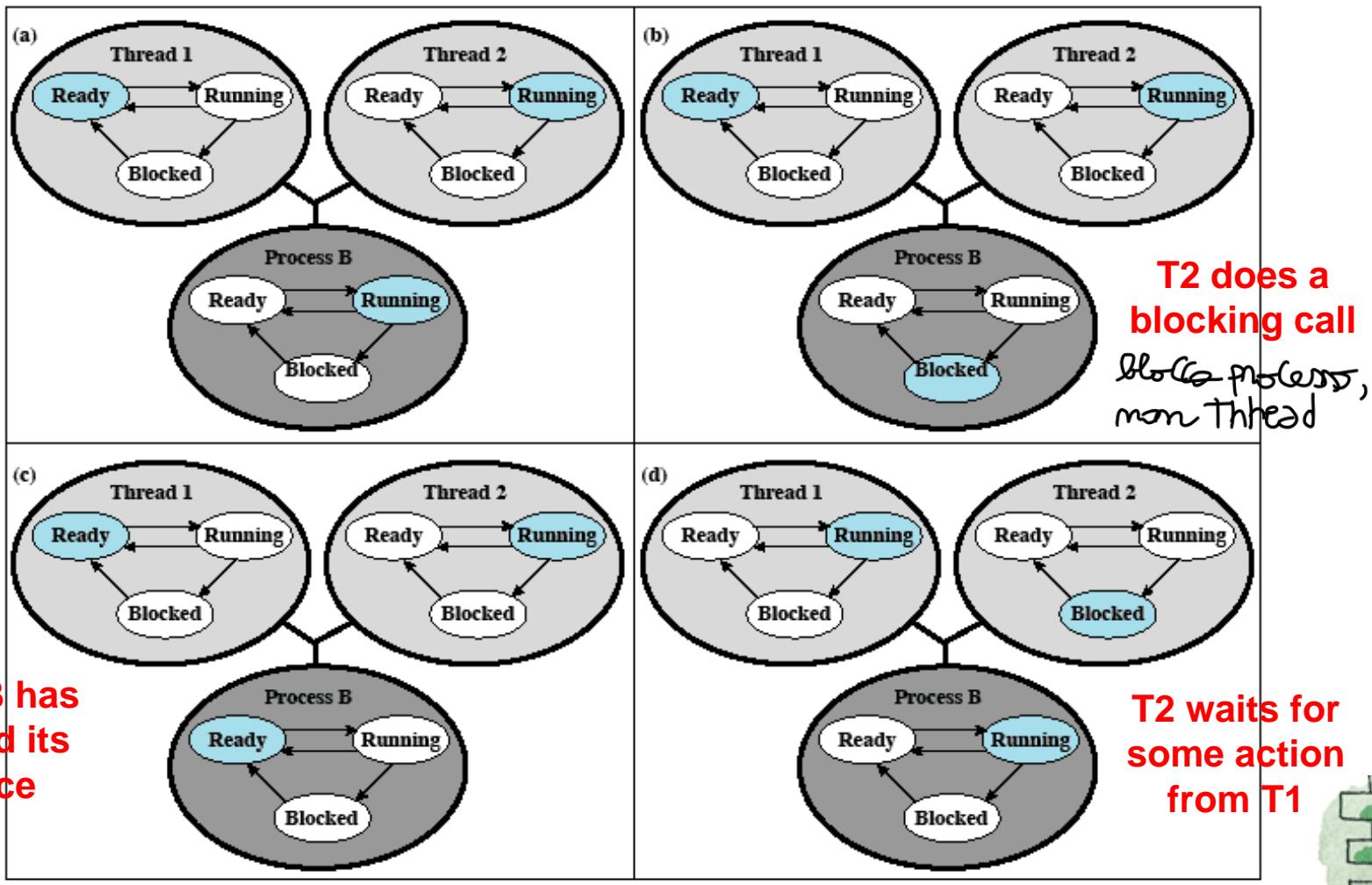
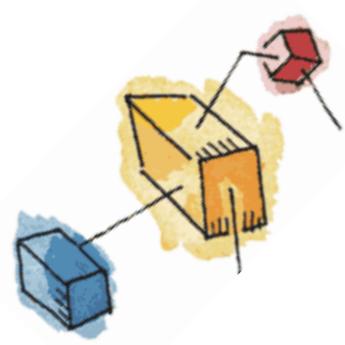
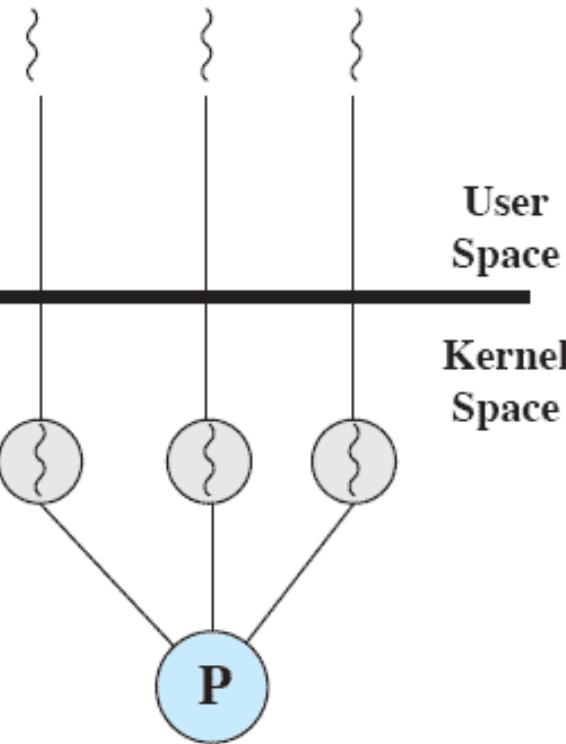


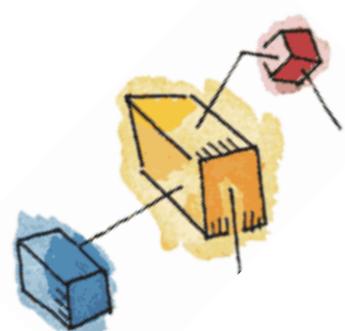
Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States



# Kernel-Level Threads



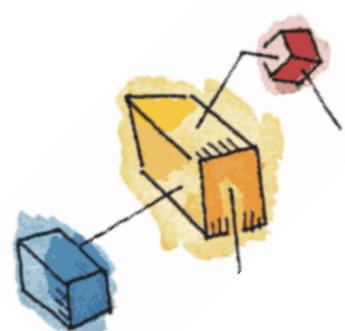
- Kernel maintains context information for the process and the threads
    - No thread management done by application
  - Scheduling is done on a thread basis
  - Windows is an example of this approach
- 



# Advantages of ULT

- Application-specific thread scheduling (i.e., independent of kernel)
- Thread switch does not require kernel privilege/switch to kernel mode
- ULTs run on any OS: implementation is done through a thread library at user level

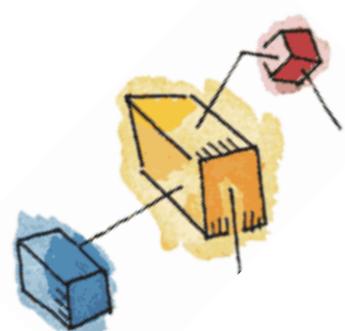




# Disadvantages of ULT

- A blocking systems call executed by a thread blocks *all threads* of the process
- Pure ULTs does not take full advantage of multiprocessors/multicores architectures

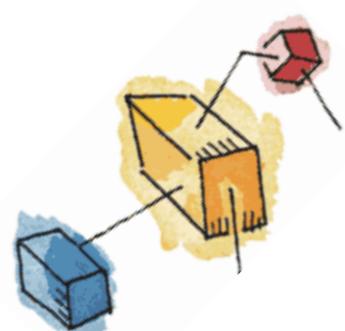




# Advantages of KLT

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines themselves can be multithreaded

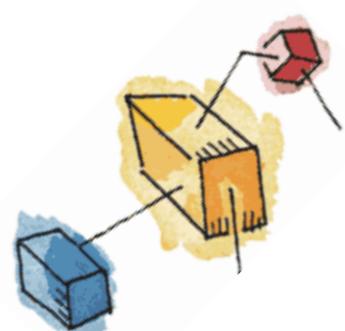




# Disadvantage of KLT

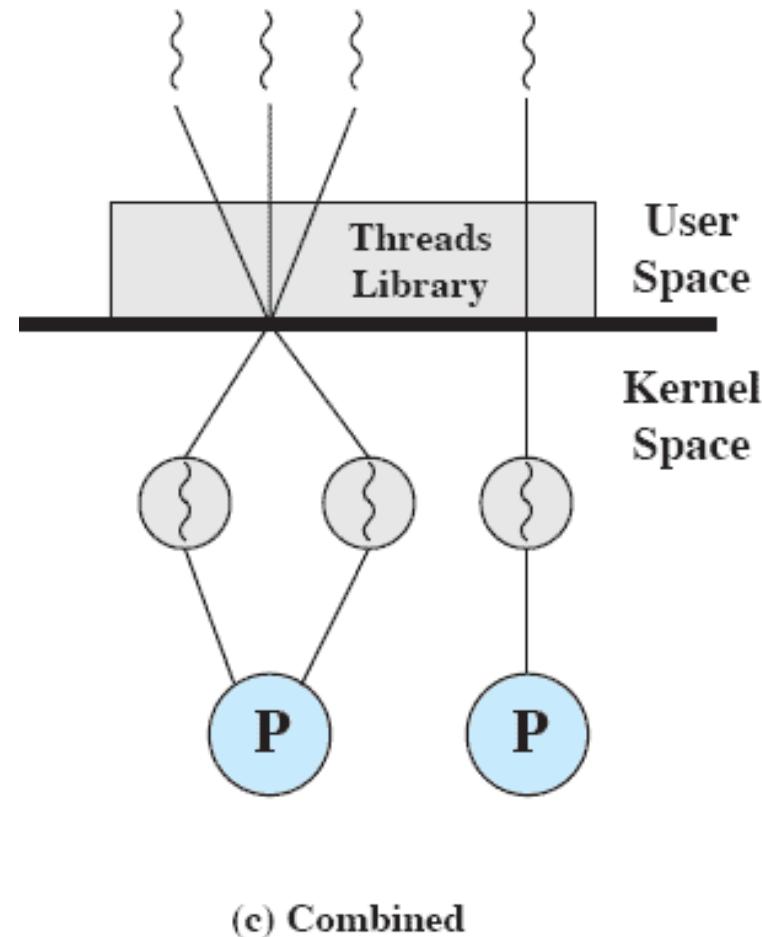
- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

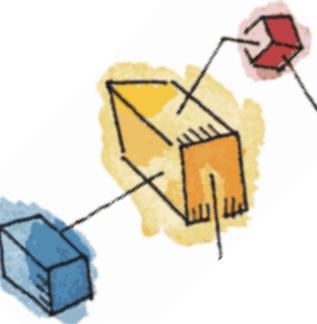




# Combined Approaches

- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads done within the application
- $u$  ULTs are mapped onto  $k$  KLTs ( $k=u$  in Solaris)

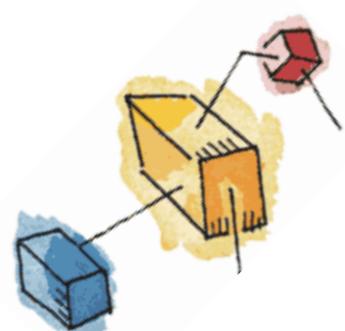




# Threads & Processes: Possible Arrangements

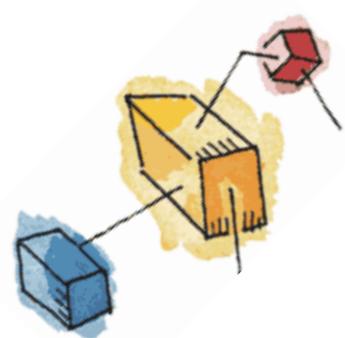
Table 4.2 Relationship Between Threads and Processes

| Threads:Processes | Description  | Example Systems                                |
|-------------------|--|--|
| 1:1               | Each thread of execution is a unique process with its own address space and resources.   | Traditional UNIX implementations               |
| M:1               | A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process. | Windows NT, Solaris, Linux, OS/2, OS/390, MACH |
| 1:M               | A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.        | Ra (Clouds), Emerald                           |
| M:N               | Combines attributes of M:1 and 1:M cases.  | TRIX   |



# Roadmap

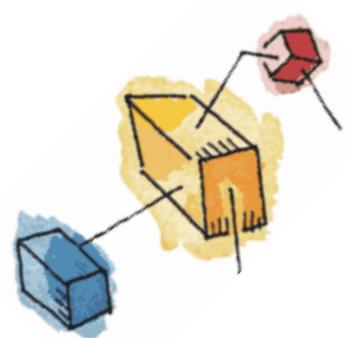
- Processes: fork (), wait()
- Threads: Resource ownership and execution
- • Case study:
  - PThreads
  - Symmetric multiprocessing (SMP).



# POSIX Threads (PThreads)

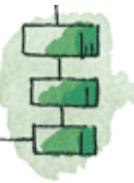
- For UNIX systems, implementations of threads that adhere to the IEEE POSIX 1003.1c standard are Pthreads.
- Pthreads are C language programming types defined in the `pthread.h` header/include file.

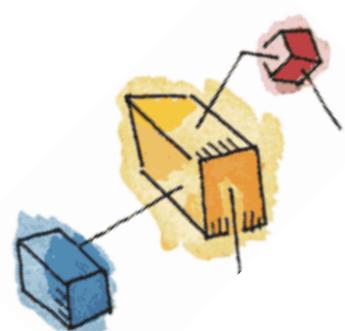




# Why Use Pthreads

- The primary motivation behind Pthreads is improving program performance
- Can be created with much less OS overhead
- Need fewer system resources to run
- Timing comparison (next slide)
  - forking processes vs `pthread_create()`
  - timings reflect 50K process/thread creations (unit: s)



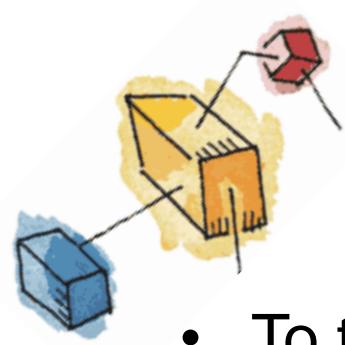


# Threads vs Forks

| Platform                                   | fork() |      |      | pthread_create() |      |     |
|--|--------|------|------|------------------|------|-----|
|  | real   | user | sys  | real             | user | sys |
| Intel 2.6 GHz Xeon E5-2670 (16 cores/node) | 8.1    | 0.1  | 2.9  | 0.9              | 0.2  | 0.3 |
| Intel 2.8 GHz Xeon 5660 (12 cores/node)    | 4.4    | 0.4  | 4.3  | 0.7              | 0.2  | 0.5 |
| AMD 2.3 GHz Opteron (16 cores/node)        | 12.5   | 1.0  | 12.5 | 1.2              | 0.2  | 1.3 |
| AMD 2.4 GHz Opteron (8 cores/node)         | 17.6   | 2.2  | 15.7 | 1.4              | 0.3  | 1.3 |
| IBM 4.0 GHz POWER6 (8 cpus/node)           | 9.5    | 0.6  | 8.8  | 1.6              | 0.1  | 0.4 |
| IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)    | 64.2   | 30.7 | 27.6 | 1.7              | 0.6  | 1.1 |
| IBM 1.5 GHz POWER4 (8 cpus/node)           | 104.5  | 48.6 | 47.2 | 2.1              | 1.0  | 1.5 |
| INTEL 2.4 GHz Xeon (2 cpus/node)           | 54.9   | 1.5  | 20.8 | 1.6              | 0.7  | 0.9 |
| INTEL 1.4 GHz Itanium2 (4 cpus/node)       | 54.5   | 1.1  | 22.2 | 2.0              | 1.2  | 0.6 |

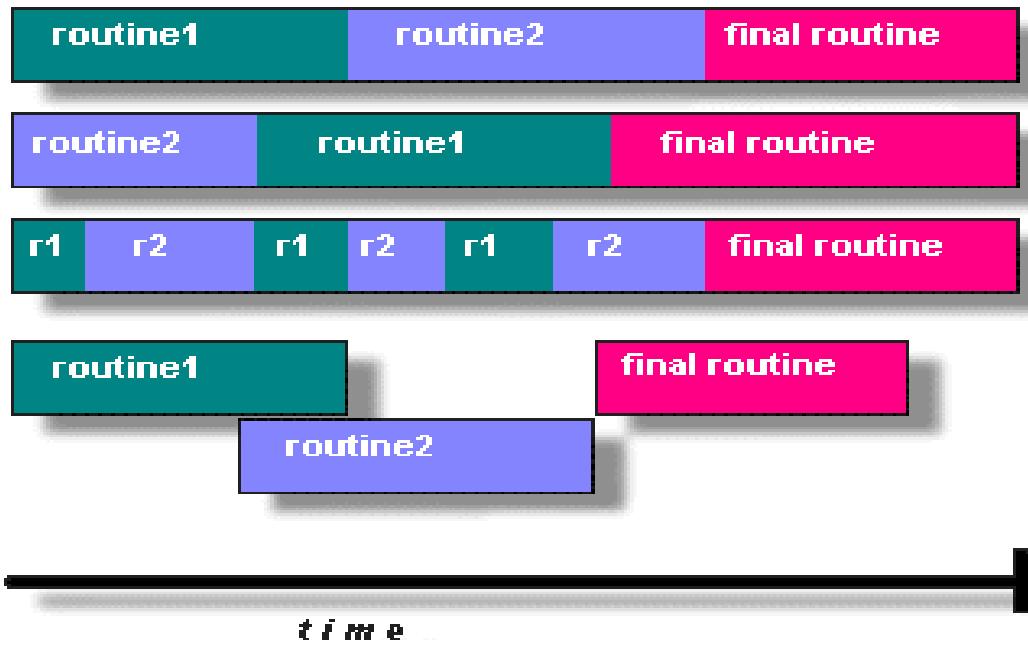
Runtime in seconds to execute 50000 operations

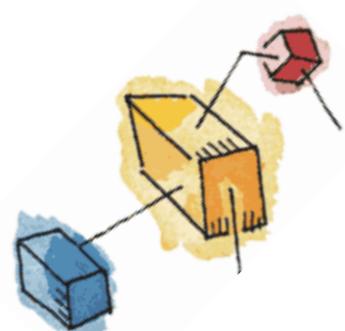




# Designing Threaded Programs as in Parallel Programming

- To take advantage of Pthreads, a program should be organized into discrete, independent tasks that can execute concurrently
  - E.g., if routine1 and routine2 can be interchanged, interleaved and/or overlapped in real time, they are candidates for threading.

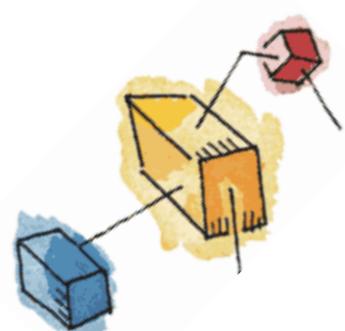




# Models for Threaded Programs

- **Manager/worker** normalmente usato (manager)
  - A *manager* thread assigns work to other threads, the *workers*. Manager handles input and hands out the work to the other tasks
- Pipeline
  - A task is broken into a series of suboperations, each handled in series, but concurrently, by a different thread  
(Esempio server (n ro clienti))



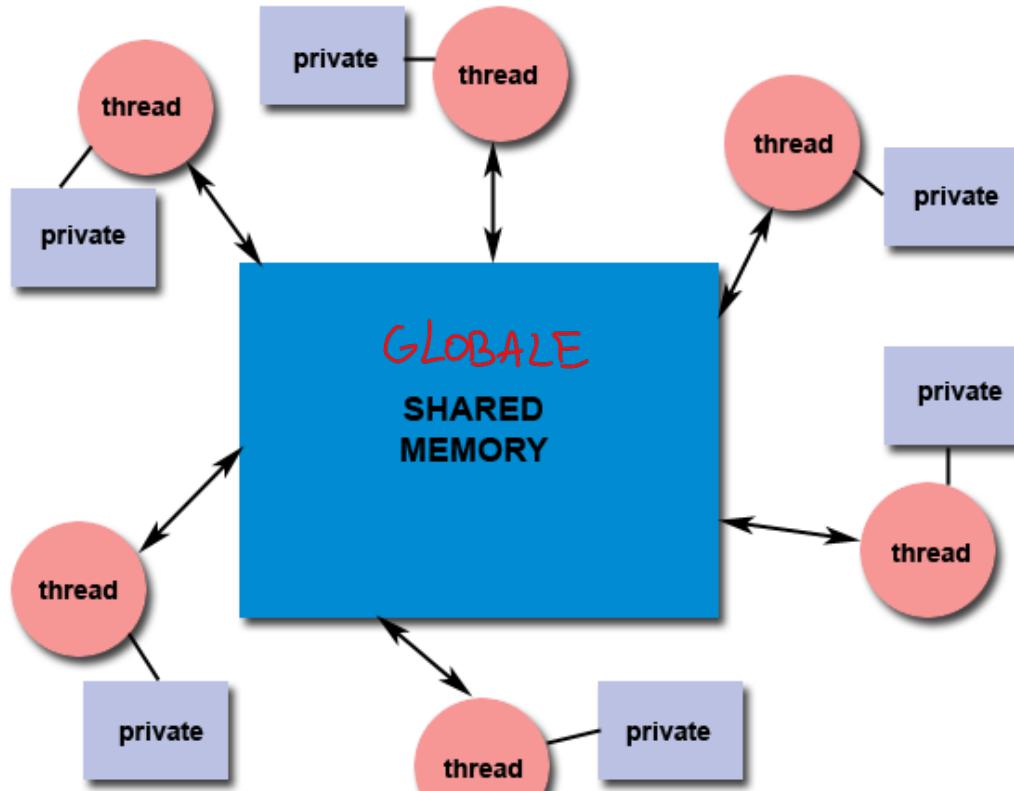


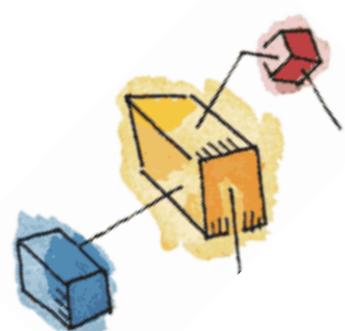
# Shared-memory Model

- All threads have access to the *same* global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access to (i.e., protecting) globally shared data



# Shared-memory Model



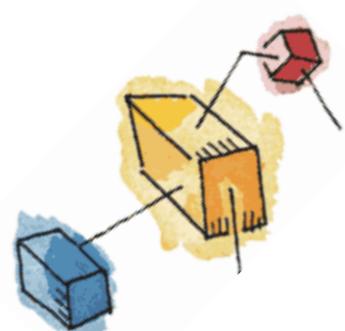


# Thread Safety

- A code is thread-safe when multiple threads can execute it simultaneously without unintended interactions
  - (without *clobbering* shared data)
  - (without creating ***race conditions***)

↳ due thread lavorano sulla  
stesse memorie

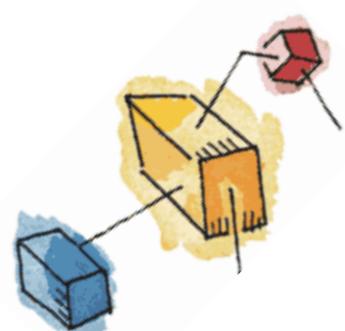




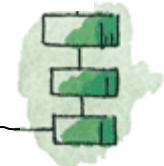
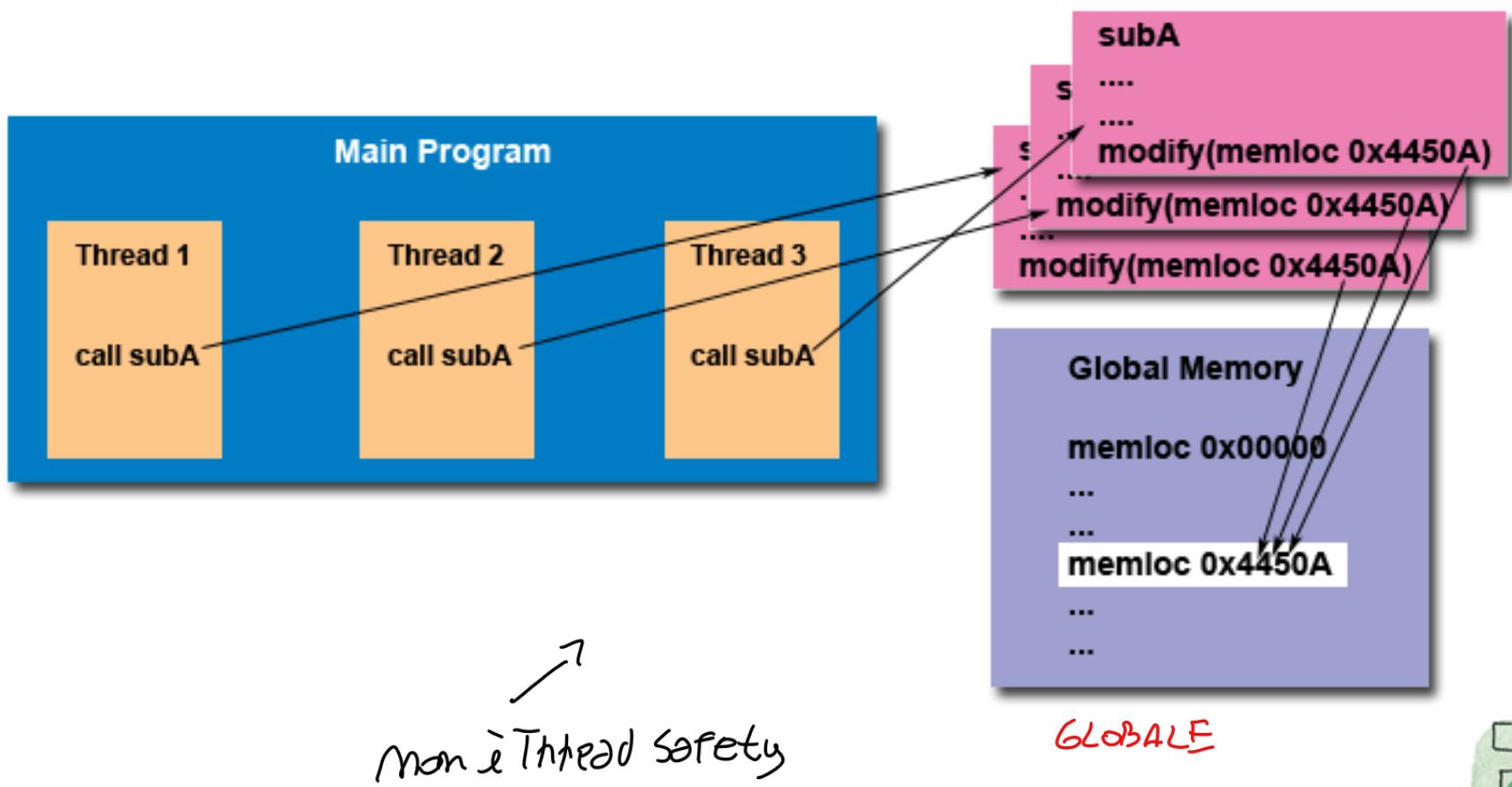
# Thread Safety

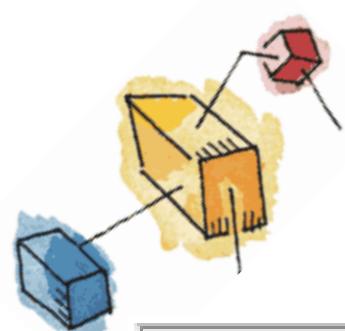
- Example: an application creates several threads, each of which makes a call to the same library routine:
  - The library routine accesses/modifies a global structure or location in memory
  - As each thread calls this routine, it is possible that they may try to modify this structure/location at the same time
  - If the routine does not employ some sort of *synchronization mechanism* to prevent data corruption, then it is not thread-safe





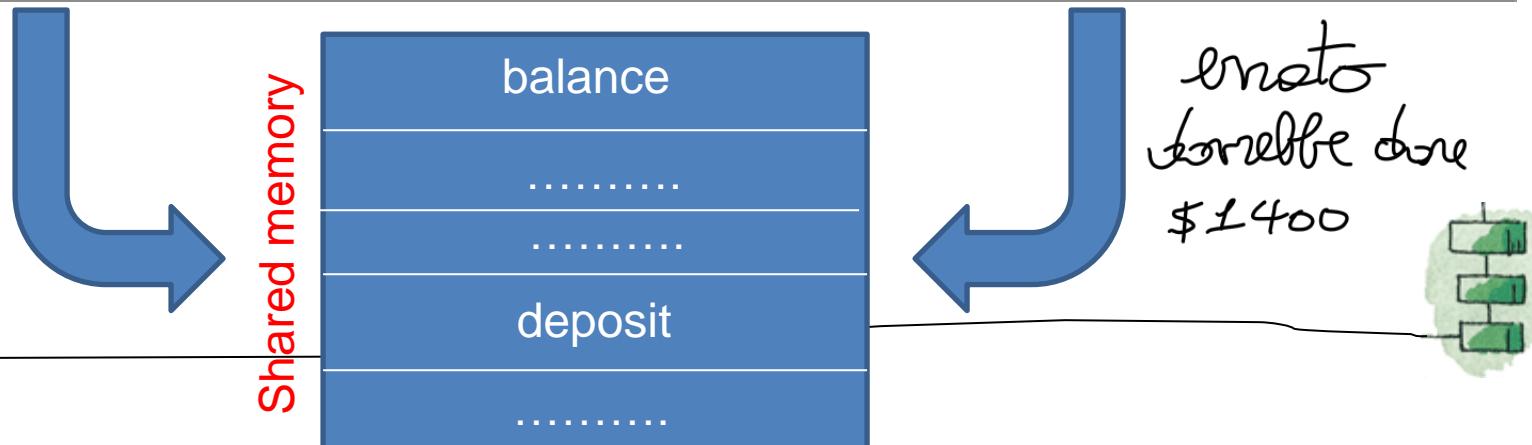
# Thread Safety

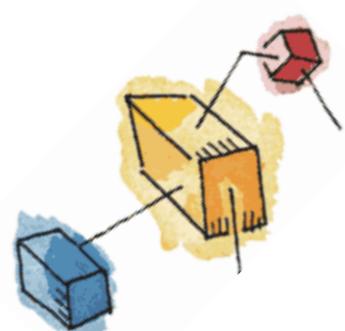




# Thread Safety

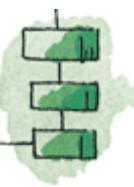
| Thread 1                              | Thread 2                    | Balance |
|---------------------------------------|-----------------------------|---------|
| Read balance: \$1000                  |                             | \$1000  |
|                                       | Read balance: \$1000        | \$1000  |
|                                       | Deposit \$200               | \$1000  |
| Deposit \$200 (was \$1000 now 1200 ↑) |                             | \$1000  |
| Update balance \$1000+\$200           |                             | \$1200  |
|                                       | Update balance \$1000+\$200 | \$1200  |

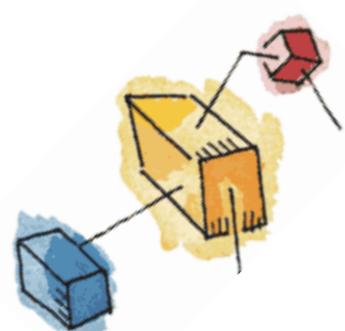




# Pthreads: Creating Threads

- A program's main() method comprises a single, default thread.
- `pthread_create()` creates a new thread and makes it executable
  - The maximum number of threads that a process can create is implementation dependent
  - Once created, threads are *peers*, and can create other threads as well

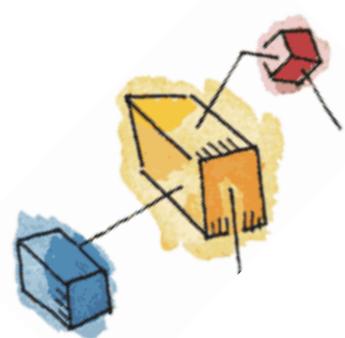




# Pthreads: Terminating Threads

- Several ways to terminate a thread, e.g.:
  - The thread is complete, i.e., the function it started with reaches a return statement
  - `pthread_exit()` is called once a thread has completed its work and it is no longer required to exist
  - `pthread_cancel()` from another thread
  - `exit()` is called (*affects the entire program!*)
  - The main terminates without executing `pthread_exit()` [caveat: `pthread_detach()`]





# Pthread: Terminating Threads (cont)

- If the main thread finishes before any other thread does, the other threads will continue executing if `pthread_exit()` was used to terminate the main, or if `pthread_detach()` was used on them
- `pthread_exit()` doesn't free resources (e.g., any file opened inside the thread will stay open), so bear cleanup in mind!



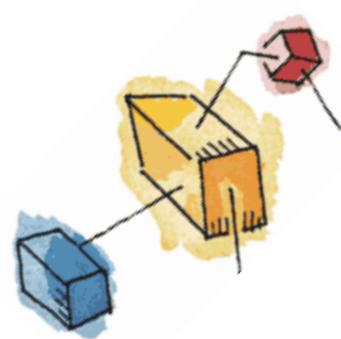
# Pthread Example (1/2)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 5

void* printHello(void *arg) {
    int threadID = *(int*) arg;
    printf("Hey! It's me, thread #%d!\n",
           threadID); free(arg);
    pthread_exit(NULL);
}
```

# Pthread Example (2/2)

```
int main (int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int ret, t;
    for (t=0; t<NUM_THREADS; t++) {
        printf("In main: creating thread %d\n", t);
        int *arg = malloc(sizeof(int));
        *arg = t;
        ret = pthread_create(&threads[t], NULL,
                            printHello, (void*)arg);
        if (ret != 0) {
            printf("ERROR: code %d\n", ret); exit(-1);
        }
    }
    pthread_exit(NULL);
}
```



# One Possible Execution

In main: creating thread 0

In main: creating thread 1

Hey! It's me, thread #0!

In main: creating thread 2

Hey! It's me, thread **#2**!

Hey! It's me, thread #1!

In main: creating thread 3

In main: creating thread 4

Hey! It's me, thread #3!

Hey! It's me, thread #4!



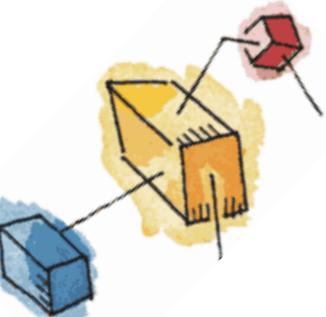
# Example: Multiple Threads

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

void *hello (void *arg) {
    printf("Hello Thread\n");
}

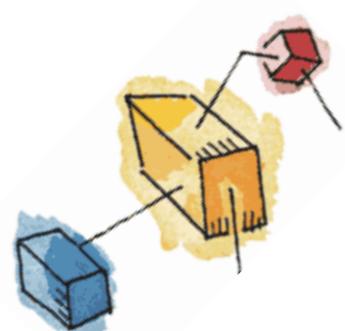
main() {
    pthread_t tid[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], NULL, hello, NULL);

    for (int i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}
```

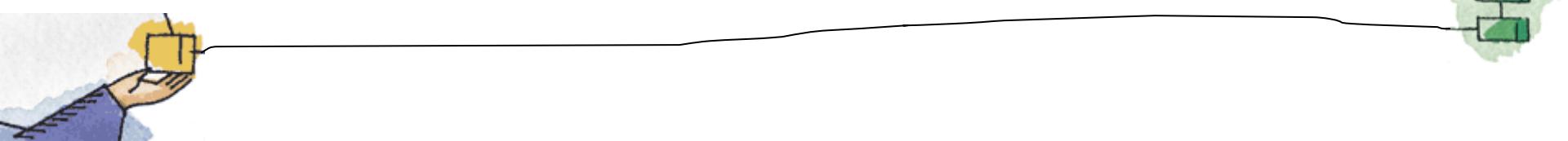


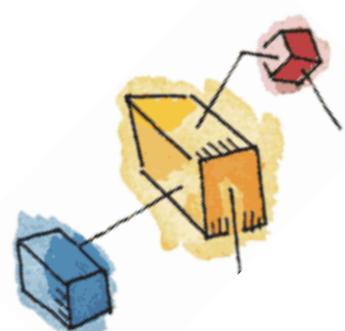
# Roadmap

- Processes: fork(), wait()
  - Threads: resource ownership and execution
  - Case study:
    - Pthreads
  - Symmetric multiprocessing (SMP)
- 



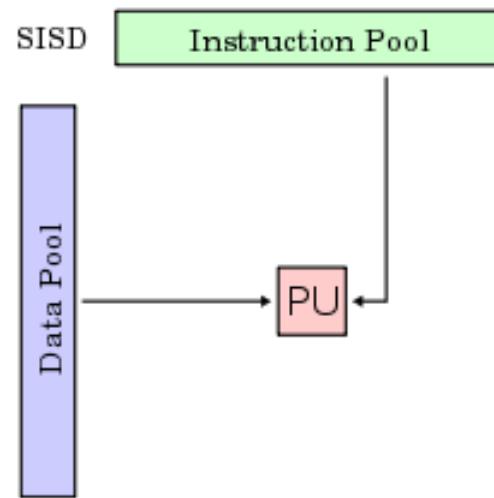
# Traditional View

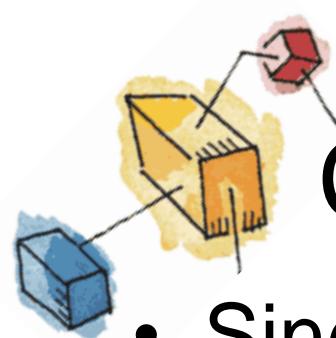
- Traditionally, the computer has been viewed as a sequential machine
    - A processor executes instructions one at a time in sequence
    - Each instruction is a sequence of operations
  - Some popular approaches to parallelism
    - Symmetric MultiProcessors (SMPs)
    - Clusters
- 



# Categories of Computer Systems (Flynn's Taxonomy)

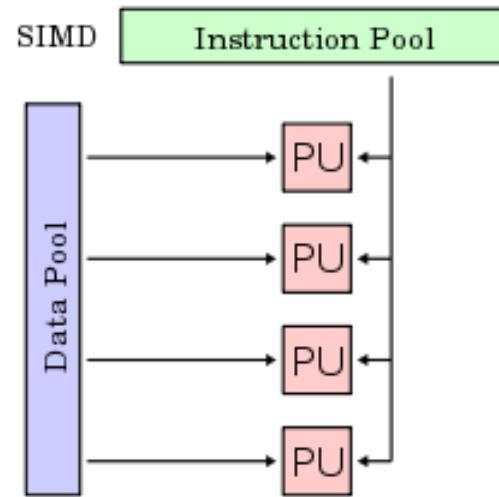
- Single Instruction Single Data (SISD)
  - Single processor executes a single instruction stream to operate on data stored in a single memory





# Categories of Computer Systems

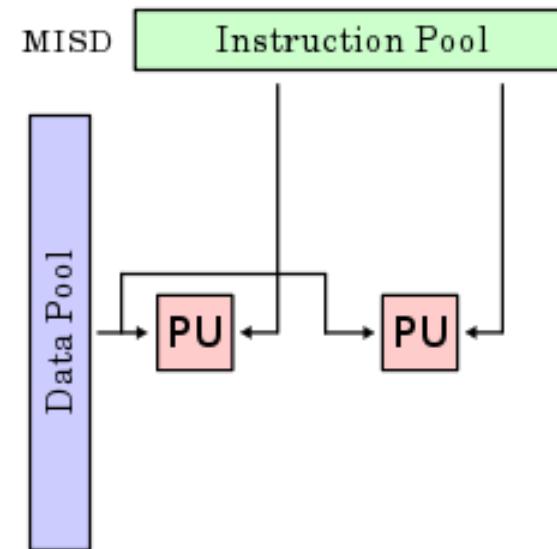
- Single Instruction Multiple Data (SIMD)
  - Each instruction is executed on a different set of data by the different processors

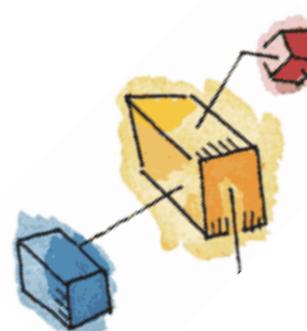




# Categories of Computer Systems

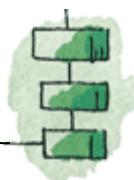
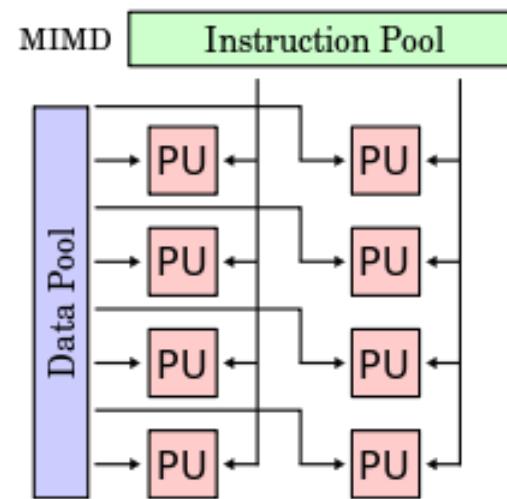
- Multiple Instruction Single Data (MISD) stream
  - A sequence of data is transmitted to a set of processors, each executing a different instruction sequence

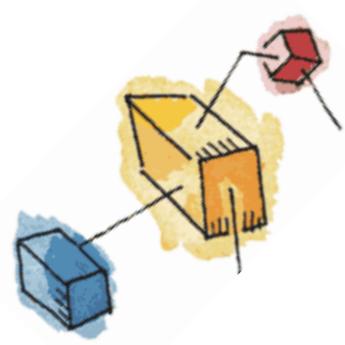




# Categories of Computer Systems

- Multiple Instruction Multiple Data (MIMD)
  - A set of processors simultaneously execute different instruction sequences on different data sets





# Parallel Processor Architectures

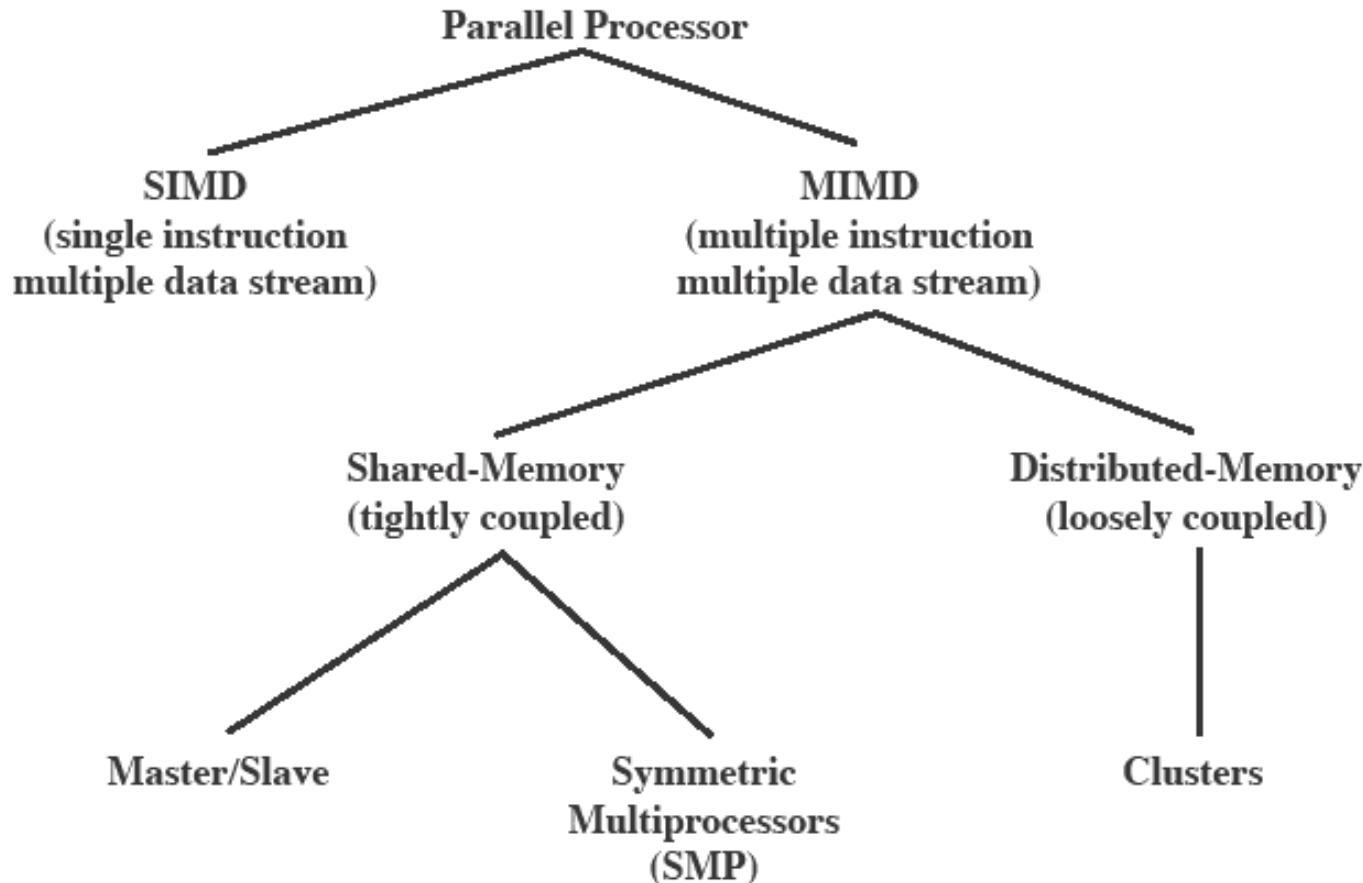
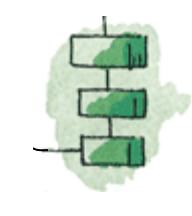


Figure 4.8 Parallel Processor Architectures



# SISTEMA MULTI PROCESSORE

## Typical SMP organization

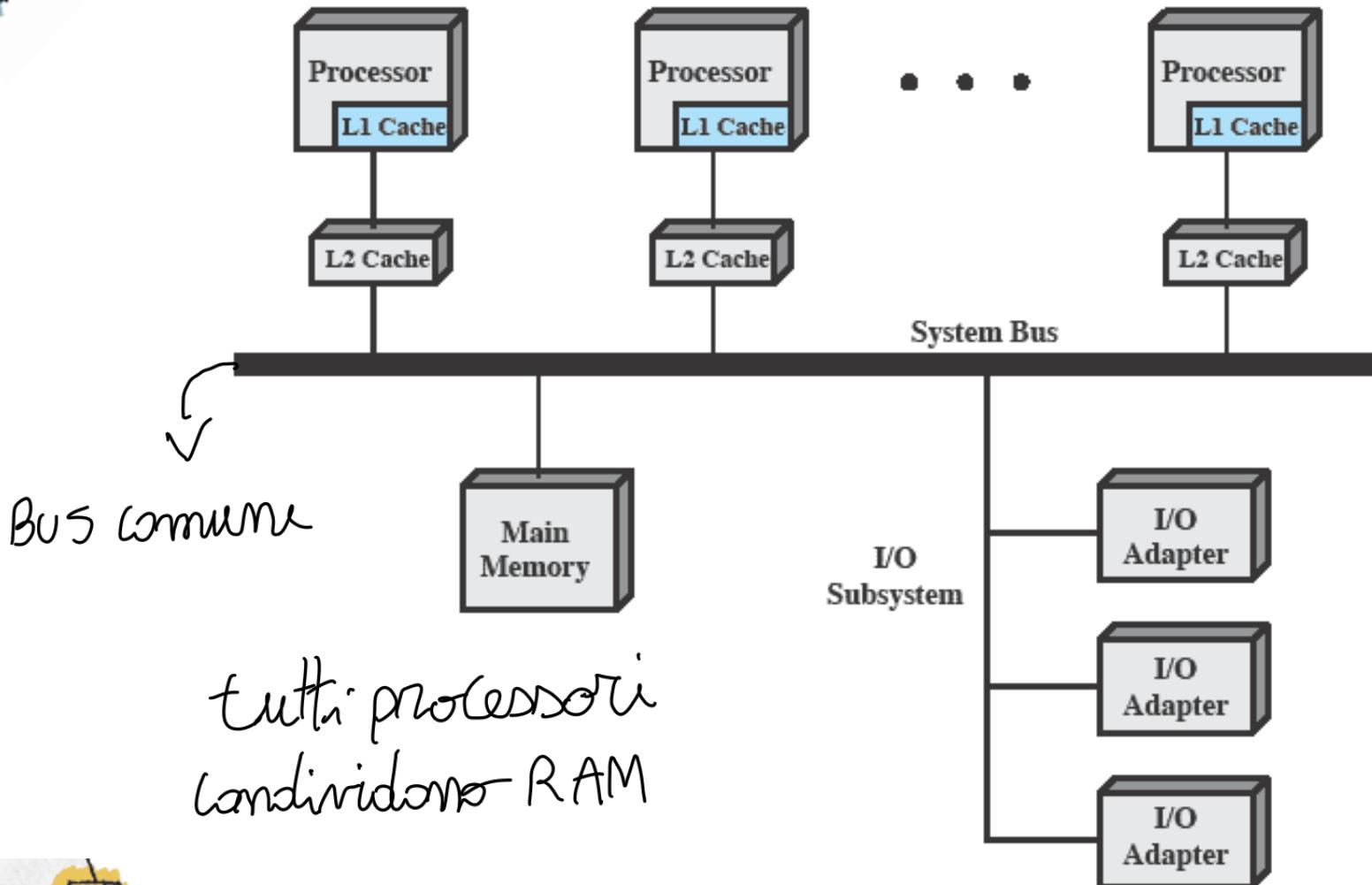
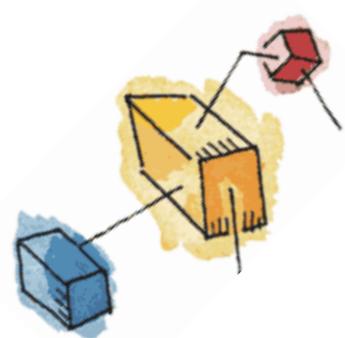


Figure 4.9 Symmetric Multiprocessor Organization



# Multiprocessor OS Design Considerations

- The key design issues include
  - Simultaneous concurrent processes or threads
  - Scheduling (Code dei processi running, blocked...)
  - Synchronization
  - Memory Management
  - Reliability and Fault Tolerance



odo uno deve accedere  
↑  
mutua esclusione e sincronizzazione

# Concurrency: mutual exclusion and synchronization

Slides are mainly taken from «*Operating Systems: Internals and Design Principles*», 8/E William Stallings (Chapter 5).

*Sistemi di Calcolo 2*

*Instructor: Riccardo Lazzeretti*

*Special thanks to: Daniele Cono D'Elia, Leonardo Aniello, Roberto Baldoni*

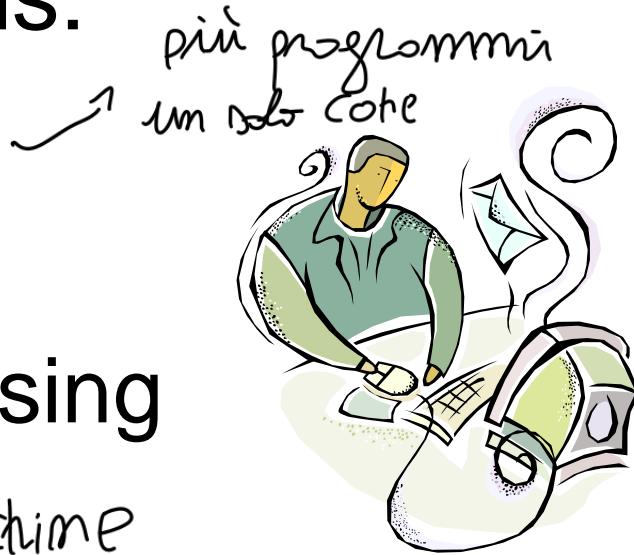
# Multiple Processes

- Operating System design is concerned with the management of processes and threads:
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing

più rate

più programmi

• Multiprogramming



↳ più macchine

# Concurrency

## Arises in Three Different Contexts:

si sono molte divisioni  
nel programma in diverse TASK

### Multiple Applications

invented to allow processing time to be shared among active applications

### Structured Applications

extension of modular design and structured programming

### Operating System Structure

OS themselves implemented as a set of processes or threads

go-nomisms con concreto

### Some Key Terms Related to Concurrency

OPERAZIONE ATOMICA: una o più istruzioni che ha le caratteristiche di essere indivisibile, se inviate insieme alla fine

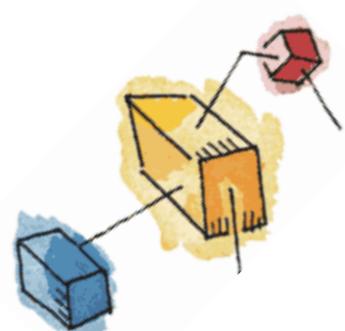
|  |   |
|--|---|
| <b>Atomic operation</b>  | A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes |
| <b>Critical Section</b><br>↳ Comparte risorse condivise                      | A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.  |
| <b>Mutual Exclusion</b><br>↳ vogliamo vivere una sezione critica, ma atomica | The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources (può essere interrotta, non come Atomic)   |
| <b>Race Condition</b><br>BISOGNO DI SINCRONIZZAZIONE                         | A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.  |

*due o più processi incapaci di eseguire, per colpa di un altro  
processo*

## Some Key Terms Related to Concurrency

|  |   |
|--|---|
| <b>Deadlock</b>                                      | A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.                           |
| <b>Livelock</b><br><i>→ e volte ci si più uscire</i> | A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work |
| <b>Starvation</b>                                    | A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.                  |

*un processo può proseguire, ma non lo fa, per colpa dello scheduling.*



# Multiprogramming Concerns

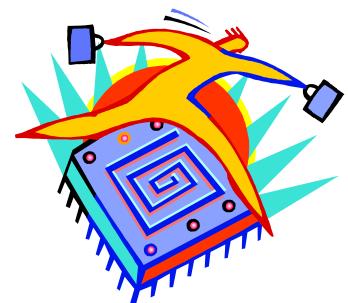
- **Output of process must be independent of the speed of execution of other concurrent processes**



# Principles of Concurrency

- *casi alterni blocchi di codice su un processo* → Interleaving and overlapping → *più processi che lavorano in contemporanea*

- can be viewed as examples of concurrent processing
- both present the same problems
- [Uniprocessor] The relative speed of execution of processes cannot be predicted; depends on:
  - activities of other processes
  - the way the OS handles interrupts
  - scheduling policies of the OS



# Difficulties of Concurrency

- Sharing of global resources
- Difficult for the OS to manage the allocation of resources optimally
- Difficult to locate programming errors as results are not deterministic and reproducible

↓  
Debug  
difficult

system operati  
the problem  
a gesticulo



# Race Condition

- Occurs when:
  - multiple processes or threads read and write data items
  - AND the final result depends on the order of execution
- The “loser” of the race is the process that updates last and will determine the *final value* of the variable



# Operating System Concerns

- Design and management issues raised by the existence of concurrency:
  - The OS must:



be able to keep track of various processes



allocate and de-allocate resources for each active process



protect the data and physical resources of each process against interference by other processes



ensure that the processes and outputs are independent of the processing speed

**Table 5.2** Process Interaction

| Degree of Awareness   | Relationship                 | Influence that One Process Has on the Other  | Potential Control Problems  |
|---|------------------------------|--|---|
| Processes unaware of each other ( <i>PROCESSI NON CONSAPEVOLI DI ALTRI PROCESSI</i> )                                 | Competition                  | <ul style="list-style-type: none"> <li>Results of one process independent of the action of others</li> <li>Timing of process may be affected</li> </ul>            | <ul style="list-style-type: none"> <li>Mutual exclusion</li> <li>Deadlock (renewable resource)</li> <li>Starvation</li> </ul>                         |
| Processes indirectly aware of each other (e.g., shared object)<br><i>INDIRETTAMENTE, con qualche ph condizionale.</i> | Cooperation by sharing       | <ul style="list-style-type: none"> <li>Results of one process may depend on information obtained from others</li> <li>Timing of process may be affected</li> </ul> | <ul style="list-style-type: none"> <li>Mutual exclusion</li> <li>Deadlock (renewable resource)</li> <li>Starvation</li> <li>Data coherence</li> </ul> |
| Processes directly aware of each other (have communication primitives available to them)                              | Cooperation by communication | <ul style="list-style-type: none"> <li>Results of one process may depend on information obtained from others</li> <li>Timing of process may be affected</li> </ul> | <ul style="list-style-type: none"> <li>Deadlock (consumable resource)</li> <li>Starvation</li> </ul>  |

# Resource Competition

- Concurrent processes come into conflict when they are competing for use of the same resource
  - for example: I/O devices, memory, processor time, clock

In the case of competing processes, three control problems must be faced:



- the need for mutual exclusion
- deadlock
- starvation

# Illustration of Mutual Exclusion

PROCESS 1 \*/

```
void P1
{
    while (true) {
        /* preceding code */;
        entercritical (Ra);
        /* critical section */;
        exitcritical (Ra);
        /* following code */;
    }
}
```

/\* PROCESS 2 \*/

```
void P2
{
    while (true) {
        /* preceding code */;
        entercritical (Ra);
        /* critical section */;
        exitcritical (Ra);
        /* following code */;
    }
}
```

/\* PROCESS n \*/

```
void Pn
{
    while (true) {
        /* preceding code */;
        entercritical (Ra);
        /* critical section */;
        exitcritical (Ra);
        /* following code */;
    }
}
```

• • •

# Requirements for Mutual Exclusion

- Must be **enforced**
- A process that halts in its noncritical section must do so without interfering with other processes
- No deadlock or starvation
- A process must not be denied access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a *finite time* only

# 1.

# Mutual Exclusion: Hardware Support

## ■ **Interrupt Disabling**

- uniprocessor system
- disabling interrupts guarantees mutual exclusion

## ■ **Disadvantages:**

- the efficiency of execution could be noticeably degraded
- this approach will not work in a multiprocessor architecture

2.

# Mutual Exclusion: Hardware Support

- Compare&Swap Instruction
  - also called a “compare and exchange instruction”
  - a **compare** is made between a memory value and a test value
  - if the values are the same a **swap** with the new value occurs in memory
  - carried out atomically
  - Available in x86, IA64, sparc, IBM architectures

Non viene interrotto



# Mutual Exclusion: Hardware Support

```
int compare_and_swap(int* reg, int oldval, int newval)
{
    ATOMIC();
    int old_reg_val = *reg;
    if (old_reg_val == oldval)
        *reg = newval;                }
    END_ATOMIC();
    return old_reg_val;
}
```



# Hardware Support for Mutual Exclusion

```
int compare_and_swap(int* reg, int oldval, int newval)
{
    ATOMIC();
    int old_reg_val = *reg;
    if (old_reg_val == oldval)
        *reg = newval;
    END_ATOMIC();
    return old_reg_val;
}
```

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true) {
        while (compare_and_swap(&bolt, 0, 1) == 1)
            /* do nothing */;
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . . , P(n));
}
```

(a) Compare and swap instruction

# Mutual Exclusion: Hardware Support

- Exchange instruction
  - exchange the content of a register with that of a memory location
  - Implemented in Pentium and Itanium (IA64) architectures



# Mutual Exclusion: Hardware Support

- Exchange instruction

```
void exchange (int *register, int *memory)
```

```
{
```

```
    int temp;
```

```
    temp = *memory;
```

```
    *memory = *register;
```

```
    *register = temp;
```

```
}
```



# Hardware Support for Mutual Exclusion

- Exchange instruction

```
void exchange (int *register, int *memory)
```

```
{  
    int temp;  
    temp = *memory;  
    *memory = *register;  
    *register = temp;  
}
```

garantisce

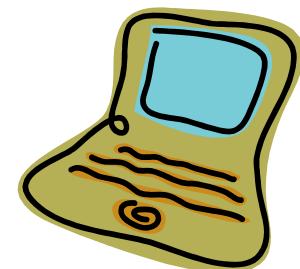
MUTUA ESCLUSIONE

```
/* program mutualexclusion */  
int const n = /* number of processes */;  
int bolt;  
void P(int i)  
{  
    while (true) {  
        int keyi = 1;  
        do exchange (&keyi, &bolt) while (keyi != 0);  
        /* critical section */;  
        bolt = 0;  
        /* remainder */;  
    }  
}  
void main()  
{  
    bolt = 0;  
    parbegin (P(1), P(2), . . . , P(n));  
}
```

(b) Exchange instruction

# Special Machine Instruction: Advantages

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- Simple and easy to verify
- It can be used to support multiple critical sections; each critical section can be defined by its own variable



# Special Machine Instruction: Disadvantages

- **Busy-waiting** is employed: while a process is waiting for access to a critical section it continues to consume processor time
- Starvation is possible when a process leaves a critical section and more than one process is waiting
- Deadlock is possible (*e.g., because of process priority*)
- Not available in any architecture
  - *Developer must be aware of the architecture*

# Common Concurrency Mechanisms

→ BASATO SU UN CONTATORE, COME DEDIS 3 OPERAZIONI

|  |  |
|--|--|
| <b>Semaphore</b><br><b>SEMAFORO</b><br><br>✓ | A construct based on an integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: <i>initialize</i> , <i>decrement</i> , and <i>increment</i> . The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process |
| <b>Binary Semaphore</b><br><br>✓             | A semaphore that takes on only the values 0 and 1  |
| <b>Mutex</b><br><br>✓                        | Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1)   |
| <b>Condition Variable</b><br><br>✓           | A data type that is used to block a process or thread until a particular condition is true   |

# Common Concurrency Mechanisms

|  |  |
|--|--|
| <b>Monitor</b><br>SEMAFORO<br>A D OGGETTI.<br><br>↓      | A programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type. <b>The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time.</b> The access procedures are critical sections. A monitor may have a queue of processes that are waiting to access it. |
| <b>Event Flags</b><br>utilizzati dai<br>DRIVER<br><br>✗? | A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).                                  |
| <b>Mailboxes / Messages</b> ✓                            | <b>A means for two processes to exchange information and that may be used for synchronization.</b>   |
| <b>Spinlocks</b> ✓<br>busy waiting !                     | Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability. (PORZIONI DI CODICE BREVE) ✗   |

1.

# Semaphore

A variable that has  
an integer value  
upon which only  
three operations  
are defined:

- There is no way to inspect or manipulate semaphores other than these three operations

- 1) May be initialized to a nonnegative integer value
- 2) The semWait operation decrements the value
- 3) The semSignal operation increments the value

↳ semaphore

# Consequences

There is no way to know before a process decrements a semaphore whether it will block or not

1.

There is no way, on a uniprocessor system, to know which process will continue immediately after a ~~semWait~~ when two processes are running concurrently

semSignal

2.

You don't know whether another process is waiting so the number of unblocked processes may be zero or one

3.

## Figure 5.3 – A Definition of Semaphore Primitives

contatore

```
struct semaphore {  
    int count;           ↗  
    queueType queue;    ↗ code ok  
};                      ↗ process  
void semWait(semaphore s) blocks  
{  
    s.count--;  
    if (s.count < 0) {  
        /* place this process in s.queue */;  
        /* block this process */;  
    }  
}  
void semSignal(semaphore s) ~> manda signal  
{  
    s.count++;  
    if (s.count <= 0) {  
        /* remove a process P from s.queue */;  
        /* place process P on ready list */;  
    }  
}
```

OPERAZIONI  
ATOMICHE:  
WAIT  
&  
SIGNAL

~> manda signal  
quando ha finito

## Figure 5.4 - A Definition of Binary Semaphore Primitives

```
struct binary_semaphore {
    enum {zero, one} value;
    queueType queue;
};

void semWaitB(binary_semaphore s)
{
    if (s.value == one)
        s.value = zero;
    else {
        /* place this process in s.queue */;
        /* block this process */;
    }
}

void semSignalB(semaphore s)
{
    if (s.queue is empty())
        s.value = one;
    else {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

# Strong/Weak Semaphores

A **queue** is used to hold processes waiting on the semaphore

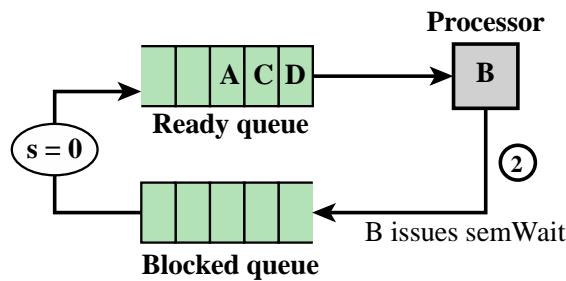
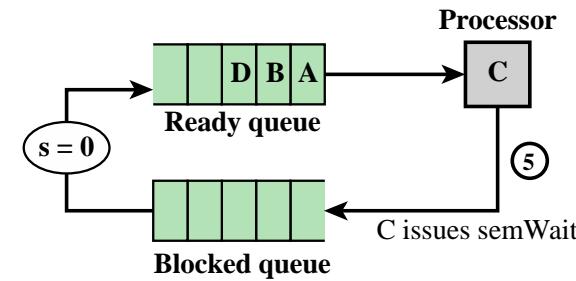
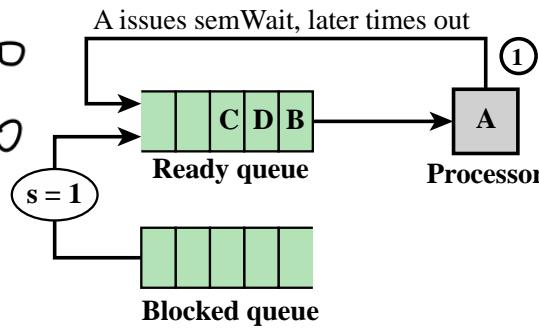
## *Strong Semaphores*

- the process that has been blocked the longest is released from the queue first (FIFO) *(NO STARVATION)*

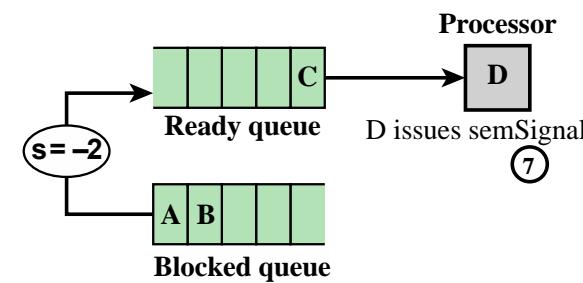
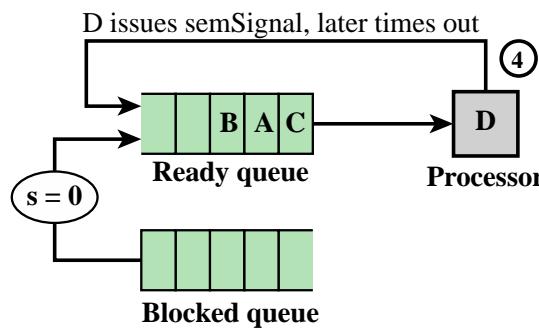
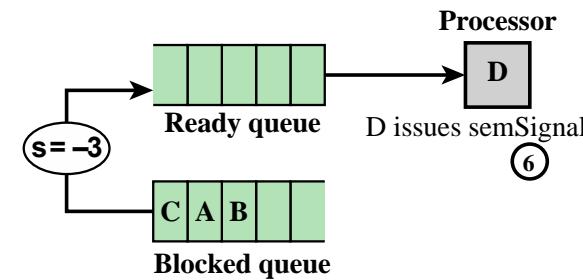
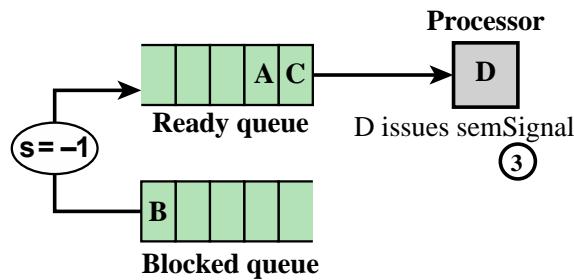
## *Weak Semaphores*

- the order in which processes are removed from the queue is not specified *(qui non posso garantire NO STARVATION)*

# ESEMPIO SEMAFORO

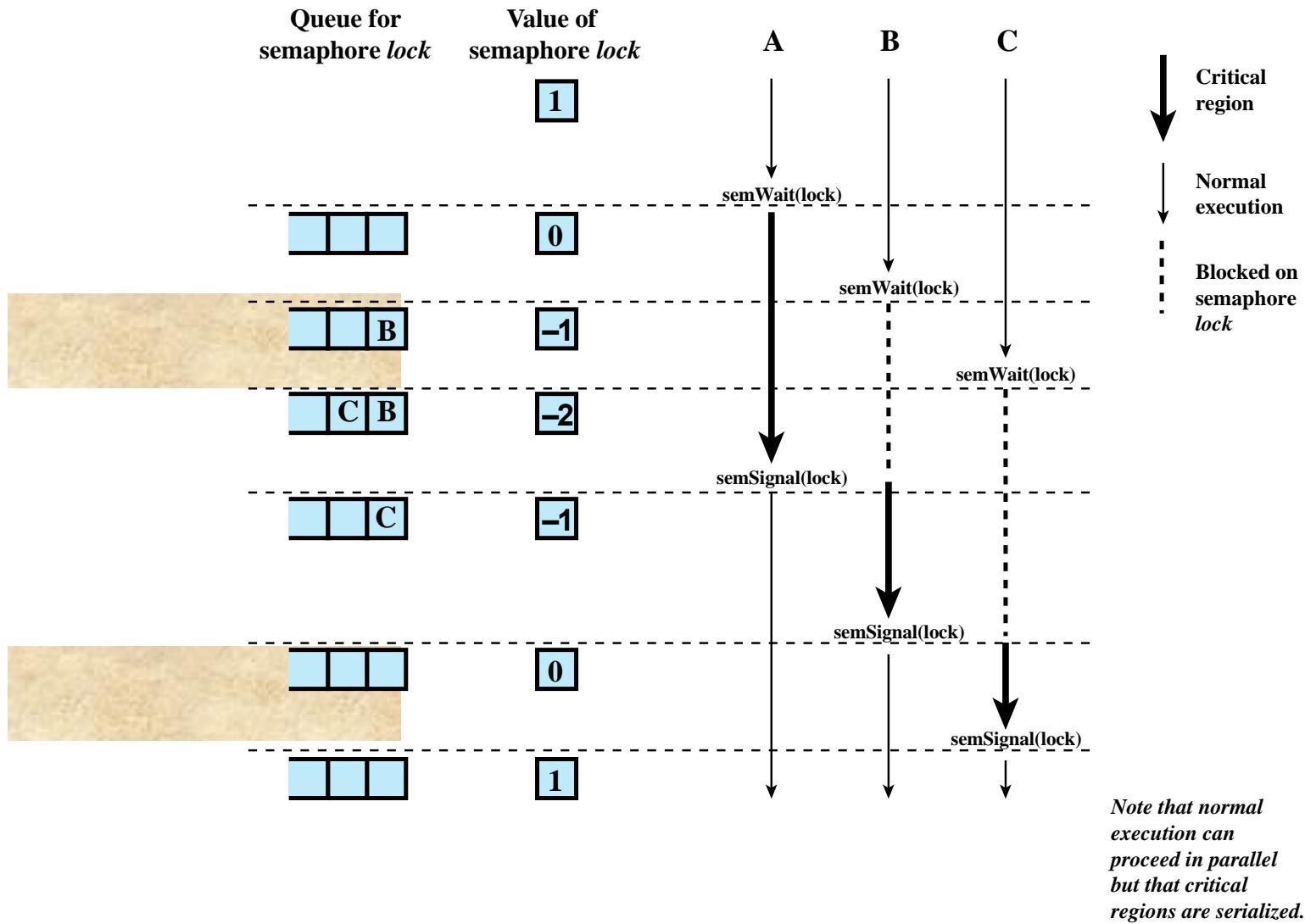


•  
•  
•



# Mutual Exclusion Using Semaphores

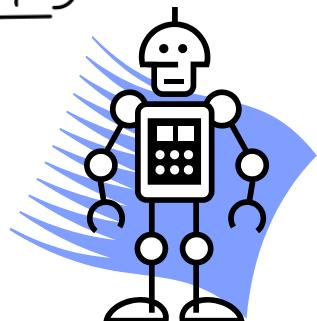
```
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
    while (true) {
        semWait(s);
        /* critical section */;
        semSignal(s);
        /* remainder */;
    }
}
void main()
{
    parbegin (P(1), P(2), . . . , P(n));
}
```



**Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore**

# Implementation of Semaphores

- Imperative that the `semWait` and `semSignal` operations be implemented as atomic primitives
  - Manipulating a semaphore is a mutual exclusion problem
- ✗ Can be implemented in hardware or firmware
  - Software schemes such as Dekker's or Peterson's algorithms can be used CDOPO
  - Use one of the hardware-supported schemes for mutual exclusion



# Two Possible Implementations of Semaphores

```
semWait(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue*/;
        /* block this process (must also set s.flag to 0) */;
    }
    s.flag = 0;
}

semSignal(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    s.flag = 0;
}
```

(a) Compare and Swap Instruction

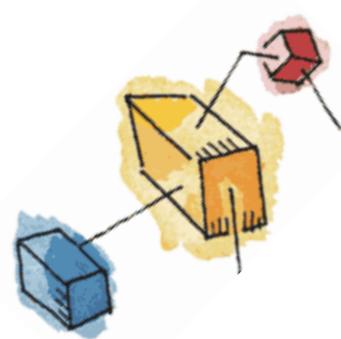
# Figure 5.14

## Two Possible Implementations of Semaphores

```
semWait(s)
{
    inhibit interrupts;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process and allow interrupts */;
    }
    else
        allow interrupts;
}

semSignal(s)
{
    inhibit interrupts;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    allow interrupts;
}
```

(b) Interrupts

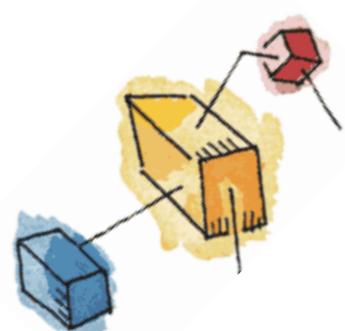


→ sono semafri **WEAK**

# Posix semaphores (c)

- `int sem_init(sem_t * sem, int pshared, unsigned value);` //initialization
- `int sem_wait(sem_t *sem);` //wait
- `int sem_post(sem_t *sem);` //signal = **SEM-Signal**
- `int sem_destroy(sem_t *sem);` //destruction
- **Parameters:**
  - `sem`: the semaphore
  - `pshared`: 0 if semaphore is shared among threads, 1 if shared among processes
  - `value`: the starting value (how many resources we can share)
- **return -1 in case of error, 0 otherwise**
  - Set `errno` to allow identification of the error type





# Example: semaphore

```
...
#include <semaphore.h>

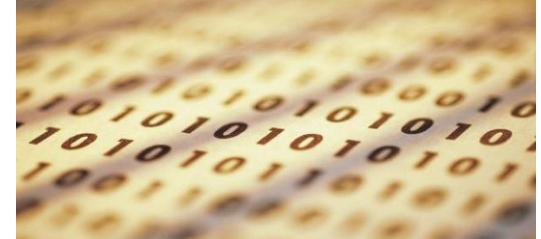
sem_t sem;

void* thread_fun(void* arg){
    sem_wait(&sem);
    //critical section
    sem_post(&sem);
}

int main (){
    sem_init(&sem,0,1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&sem);
    return 0;
}
```



# Monitors



- Programming language construct that provides equivalent functionality to that of semaphores and is easier to control
- Implemented in a number of programming languages
  - including Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, and Java
- Has also been implemented as a program library
- Software module consisting of one or more procedures, an initialization sequence, and local data

# Monitor Characteristics

Solo un processo alla volta nel Monitor

Local data variables are accessible only by the monitor's procedures and not by any external procedure



Process enters monitor by invoking one of its procedures

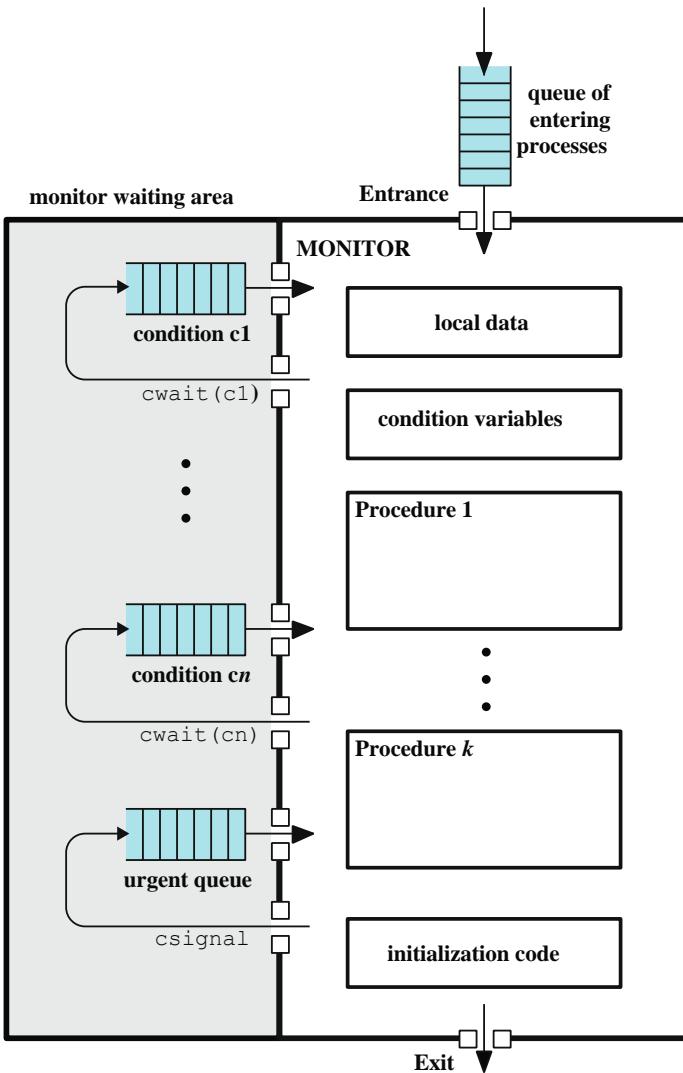


Only one process may be executing in the monitor at a time

# Synchronization

- Achieved by the use of **condition variables** that are contained within the monitor and accessible only within the monitor
  - Condition variables are operated on by two functions:
    - » `cwait(c)`: suspend execution of the calling process on condition c
    - » `csignal(c)`: resume execution of some process blocked after a `cwait` on the same condition





→ nelle procedure ci sarà una CWait su una certa condizione prima della sezione critica e alla fine una CSignal

Figure 5.15 Structure of a Monitor

# Message Passing

- When processes interact with one another, two fundamental requirements must be satisfied:

synchronization

communication

- to enforce mutual exclusion

- to exchange information

- Message Passing is one approach to providing both of these functions
  - works with distributed systems and shared memory multiprocessor and uniprocessor systems

# Message Passing



- The actual function is normally provided in the form of **a pair of primitives**:

**send (destination, message)**

**receive (source, message)**

- » A process sends information in the form of a message to another process designated by a destination
- » A process receives information by executing the receive primitive, indicating the source and the message

# Design Characteristics of Message Systems for Interprocess Communication and Synchronization

## Synchronization

Send  
blocking  
nonblocking → Controlla se il messaggio è correttamente inviato  
Receive  
blocking  
nonblocking  
test for arrival (Poco usato)

## Addressing (INDIRIZZAMENTO)

### Direct

- send
- receive
  - explicit
  - implicit

### Indirect

- static
- dynamic
- ownership

controlla se il messaggio è correttamente inviato

(Poco usato)

## Format (FORMATO)

- Content
- Length
  - fixed → FISSA
  - variable → VARIABILE

## Queueing Discipline

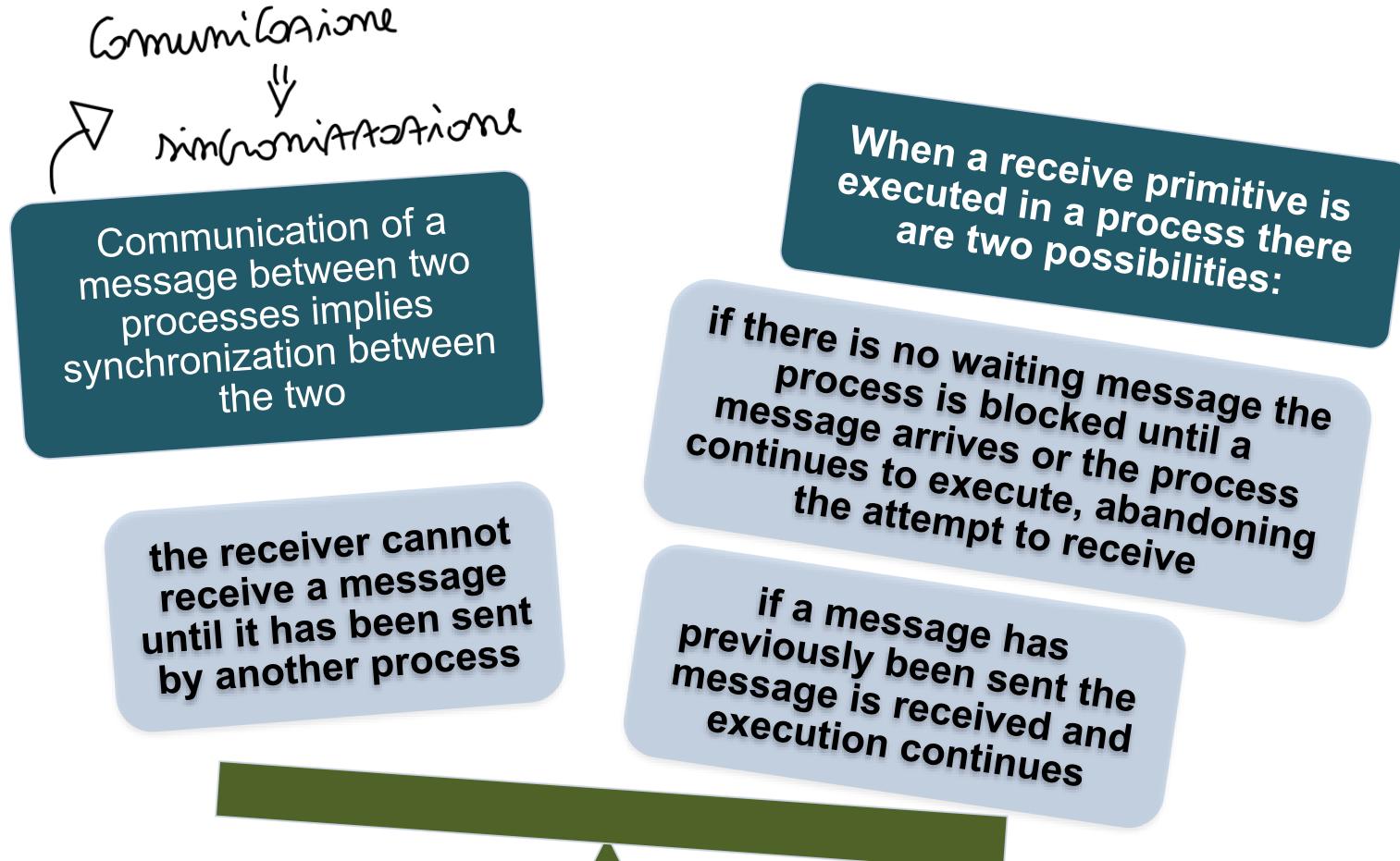
- FIFO
- Priority

se molti messaggi ho problemi di code

↳ chi possiede il messaggio

Mandante è conoscitore, solo messaggio

# Synchronization



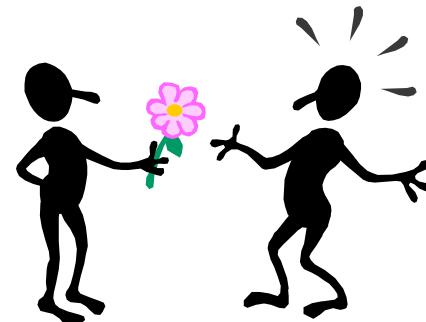
# Blocking Send, Blocking Receive

send

receive

BLOCCO FORTE

- Both sender and receiver are blocked until the message is delivered
- Sometimes referred to as a *rendezvous*
- Allows for tight synchronization between processes



# Nonblocking Send

## Nonblocking send, blocking receive

 SEND  
 RECEIVE

- sender continues on but receiver is blocked until the requested message arrives
- most useful combination (molto usata)
- sends one or more messages to a variety of destinations as quickly as possible
- example -- a service process that exists to provide a service or resource to other processes

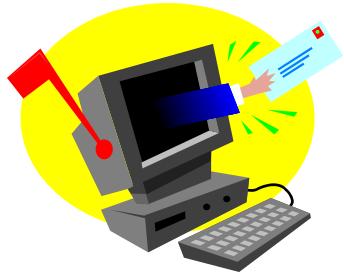
## Nonblocking send, nonblocking receive

- neither party is required to wait



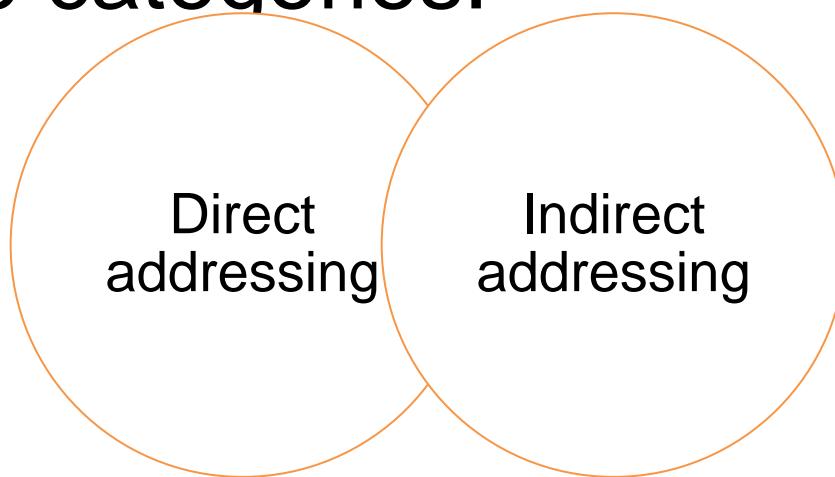
(meno comune)

 SEND  
 RECEIVE



# Addressing

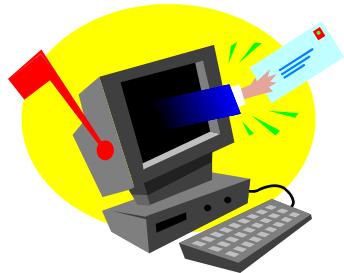
- ★ Schemes for specifying processes in send and receive primitives fall into two categories:





# Direct Addressing

- Send primitive includes a specific identifier of the destination process
- Receive primitive can be handled in one of two ways:
  - require that the process explicitly designate a sending process
    - effective for cooperating concurrent processes
  - implicit addressing
    - source parameter of the receive primitive possesses a value returned when the receive operation has been performed



# Indirect Addressing

Queues = mailboxes

Messages are sent to a shared data structure consisting of queues that can temporarily hold messages

Queues are referred to as *mailboxes*

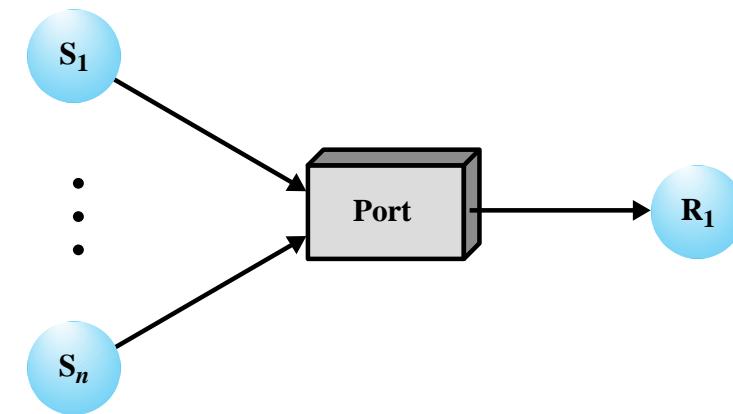
Allows for greater flexibility in the use of messages

One process sends a message to the mailbox and the other process picks up the message from the mailbox

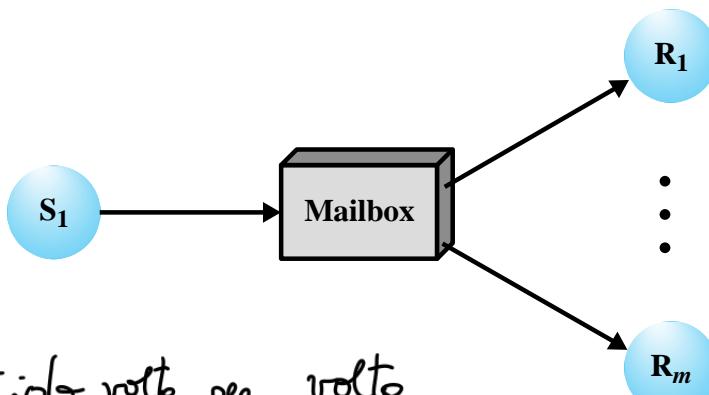
Il processo ricevente potrebbe anche non essere inizializzato, potrà leggere la mailbox in un momento differente, senza causare errore di comunicazione.



(a) One to one

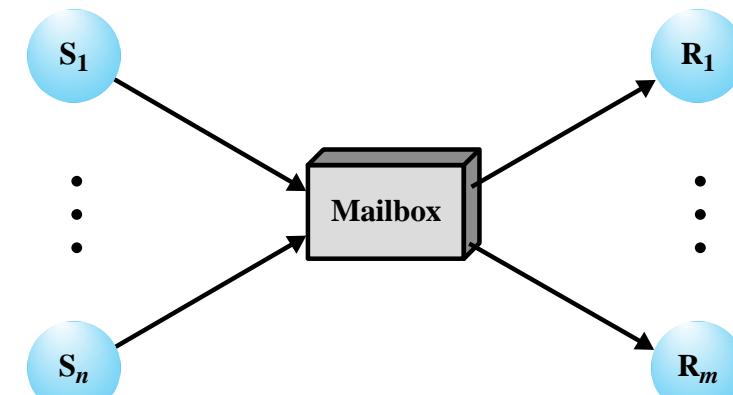


(b) Many to one



(c) One to many

1) decido volte per volte  
e chi inviare



(d) Many to many

2) invio a  
uno qualcuno

3) Broadcast

Figure 5.18 Indirect Process Communication

# MESSAGGIO

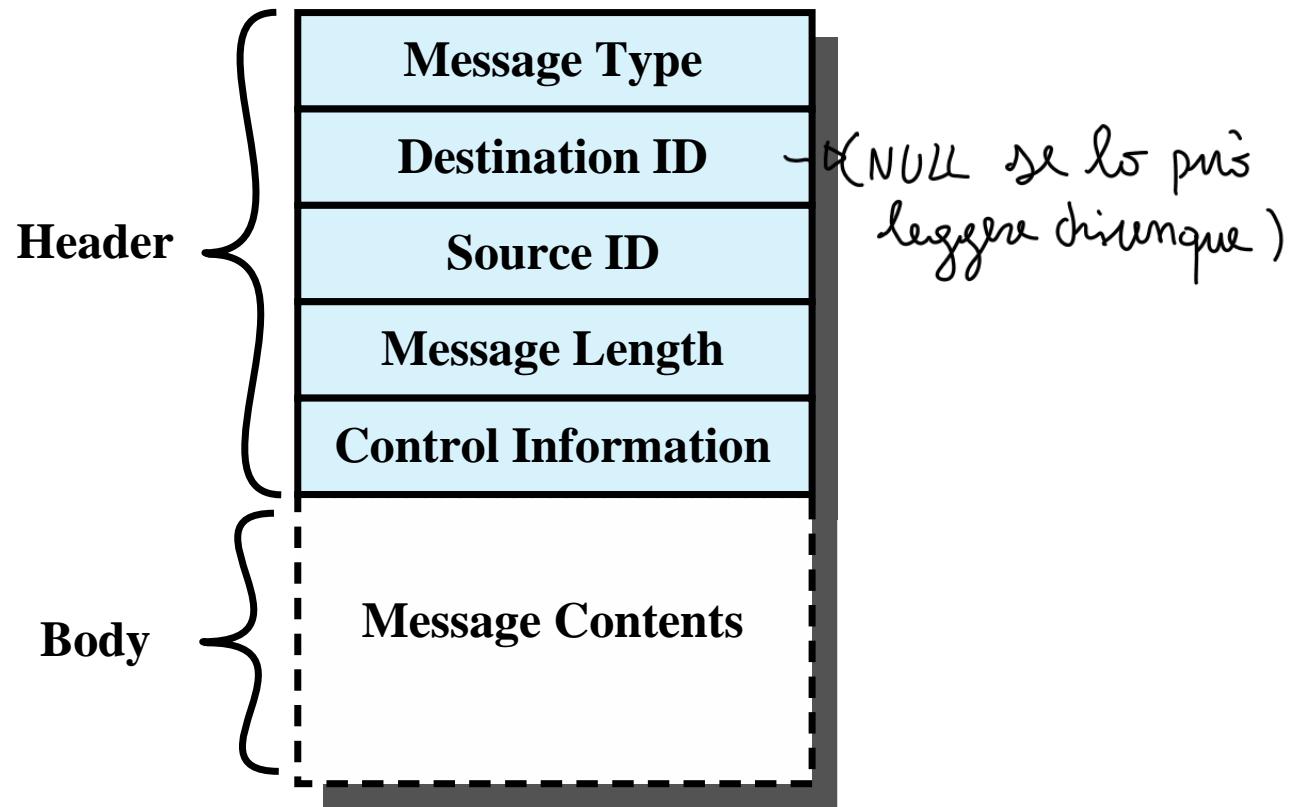


Figure 5.19 General Message Format

MUTUA ESCLUSIONE CON MESSAGGI  $\Rightarrow$  può capitare STARVATION!

```
/* program mutualexclusion */
const int n = /* number of processes */;
void P(int i)
{
    message msg;
    while (true) {
        receive (box, msg);
        /* critical section */
        send (box, msg);
        /* remainder */
    }
}
void main()
{
    create_mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . . , P(n));
}
```

una struttura fornita  
da Posix.

finché è vuota la mailbox,  
la sezione critica non  
\*/; viene svolta.

Send  
Receive  $\Rightarrow$  ATOMICHE  
||

non vengono  
interrutte.

Figure 5.20  
Mutual Exclusion Using Messages  
(Demonio many to many)

# Producer/Consumer Problem

General Statement:

one or more producers are generating data and placing these in a buffer

one or more consumer are taking items out of the buffer one at a time

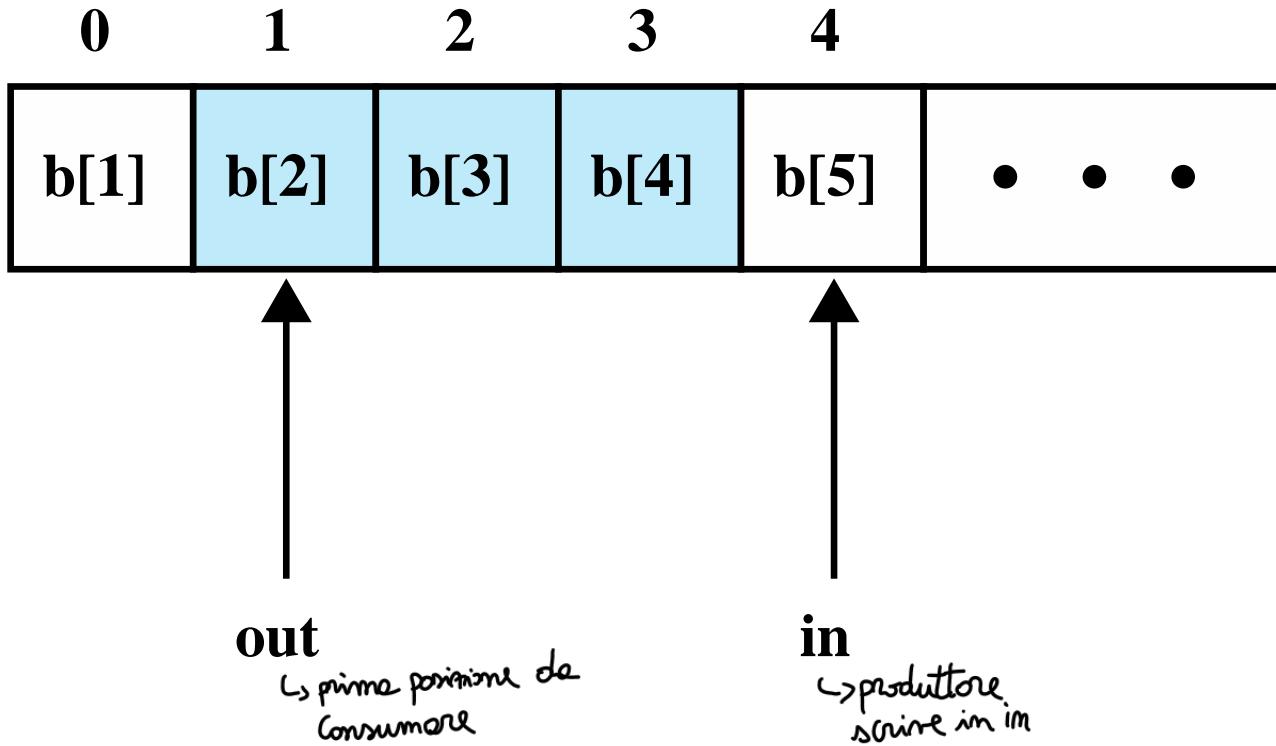
only one producer or consumer may access the buffer at any one time

*2> mentre uno scrive, nessuno può leggere*

The Problem:

ensure that the producer can't add data into full buffer and consumer can't remove data from an empty buffer

buffet ideale, è infinito



Note: shaded area indicates portion of buffer that is occupied

**Figure 5.8 Infinite Buffer for the Producer/Consumer Problem**

# Infinite-Buffer P/C - An incorrect solution

```
/* program producerconsumer */  
int n = 0;                                → mi dice se  
binary_semaphore s = 1, delay = 0;
```

```
void producer()  
{  
    while (true) {  
        produce();  
        semWaitB(s);  
        append(); → sezione critica  
        n++;  
        if (n==1) semSignalB(delay);  
        semSignalB(s);  
    } c'è qualcosa  
da consumare
```

# Infinite-Buffer P/C - An incorrect solution

```
/* program producerconsumer */  
int n = 0;  
binary_semaphore s = 1, delay = 0;
```

```
void consumer()  
{  
    semWaitB(delay); // aspetta che c'è qualcosa  
    while (true) {  
        semWaitB(s); // da consumare, non  
        take(); // ha nulla  
        n--;  
        semSignalB(s);  
        consume();  
        if (n==0) semWaitB(delay);  
    }  
}
```

NO n potrebbe

essere  
completo

↳ Il buffer si è  
svuotato

# Consuming an item that does not exist...

| Producer                                 | Consumer        | s | n | Delay |
|--|-----------------|---|---|-------|
|  |                 | 1 | 0 | 0     |
| semWaitB(s)                              |                 | 0 | 0 | 0     |
| n++                                      |                 | 0 | 1 | 0     |
| <b>if (n==1)<br/>(semSignalB(delay))</b> |                 | 0 | 1 | 1     |
| semSignalB(s)                            |                 | 1 | 1 | 1     |
|  | semWaitB(delay) | 1 | 1 | 0     |
|  | semWaitB(s)     | 0 | 1 | 0     |
|  | n--             | 0 | 0 | 0     |
|  | semSignalB(s)   | 1 | 0 | 0     |

**Consumer is descheduled before it can test:  
if (n==0) semWaitB(delay)  
So the producer enters CS and increases n...**

# Consuming an item that does not exist...

|   |                                    |   |   |   |
|---|------------------------------------|---|---|---|
|   | semSignalB(s)                      | 1 | 0 | 0 |
| semWaitB(s)                             |                                    | 0 | 0 | 0 |
| n++                                     |                                    | 0 | 1 | 0 |
| <b>if</b> (n==1)<br>(semSignalB(delay)) |                                    | 0 | 1 | 1 |
| semSignalB(s)                           |                                    | 1 | 1 | 1 |
|   | <b>if</b> (n==0) (semWaitB(delay)) | 1 | 1 | 1 |

Later on we get **n == -1**: we are reading an invalid element!

```
// consumer
semWaitB(s)
take(), n--
semSignalB(s)
consume()
if (n==0) ...
```

|                                    |   |    |   |
|------------------------------------|---|----|---|
| semWaitB(s)                        | 0 | 1  | 1 |
| n--                                | 0 | 0  | 1 |
| semSignalB(s)                      | 1 | 0  | 1 |
| <b>if</b> (n==0) (semWaitB(delay)) | 1 | 0  | 0 |
| semWaitB(s)                        | 0 | 0  | 0 |
| n--                                | 0 | -1 | 0 |
| semSignalB(s)                      | 1 | -1 | 0 |

# Consuming an item that does not exist...

|    | Producer                                | Consumer                           | s | n  | Delay |
|----|---|------------------------------------|---|----|-------|
| 1  |   |                                    | 1 | 0  | 0     |
| 2  | semWaitB(s)                             |                                    | 0 | 0  | 0     |
| 3  | n++                                     |                                    | 0 | 1  | 0     |
| 4  | <b>if</b> (n==1)<br>(semSignalB(delay)) |                                    | 0 | 1  | 1     |
| 5  | semSignalB(s)                           |                                    | 1 | 1  | 1     |
| 6  |   | semWaitB(delay)                    | 1 | 1  | 0     |
| 7  |   | semWaitB(s)                        | 0 | 1  | 0     |
| 8  |   | n--                                | 0 | 0  | 0     |
| 9  |   | semSignalB(s)                      | 1 | 0  | 0     |
| 10 | semWaitB(s)                             |                                    | 0 | 0  | 0     |
| 11 | n++                                     |                                    | 0 | 1  | 0     |
| 12 | <b>if</b> (n==1)<br>(semSignalB(delay)) |                                    | 0 | 1  | 1     |
| 13 | semSignalB(s)                           |                                    | 1 | 1  | 1     |
| 14 |   | <b>if</b> (n==0) (semWaitB(delay)) | 1 | 1  | 1     |
| 15 |   | semWaitB(s)                        | 0 | 1  | 1     |
| 16 |   | n--                                | 0 | 0  | 1     |
| 17 |   | semSignalB(s)                      | 1 | 0  | 1     |
| 18 |   | <b>if</b> (n==0) (semWaitB(delay)) | 1 | 0  | 0     |
| 19 |   | semWaitB(s)                        | 0 | 0  | 0     |
| 20 |   | n--                                | 0 | -1 | 0     |
| 21 |   | semSignalB(s)                      | 1 | -1 | 0     |

# Infinite-Buffer P/C - Another incorrect solution

```
/* program producerconsumer */
int n = 0;
binary_semaphore s = 1, delay = 0;
```

```
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
```

# Infinite-Buffer P/C - Another incorrect solution

```
/* program producerconsumer */
int n = 0;
binary_semaphore s = 1, delay = 0;
```

```
void consumer()
{
    semWaitB(delay);
    while (true)
        semWaitB(s);
        take();
        n--;
        consume();
        if (n==0) semWaitB(delay);
        semSignalB(s);
}
```

**Deadlock**

mano produttore  
può entrare, ma non che  
più andare avanti il  
consumatore

# A possible fix (using binary semaphores only)

```
void consumer()
{
    int m; /* a local variable */
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        m = n;
        semSignalB(s);
        consume();
        if (m==0) semWaitB(delay);
    }
}
```

We make a local copy  $m = n$  of the variable, so we can read the correct value once we left the CS

CRITICAL  
SECTION

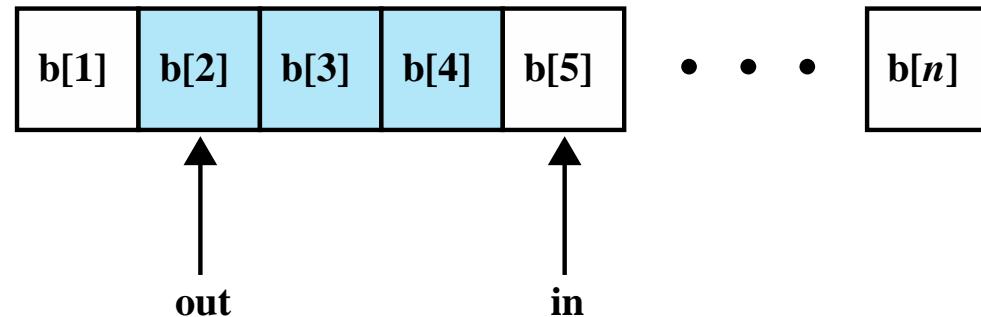
# Infinite-Buffer P/C – General semaphores

```
/* program producerconsumer */  
semaphore n = 0, s = 1;
```

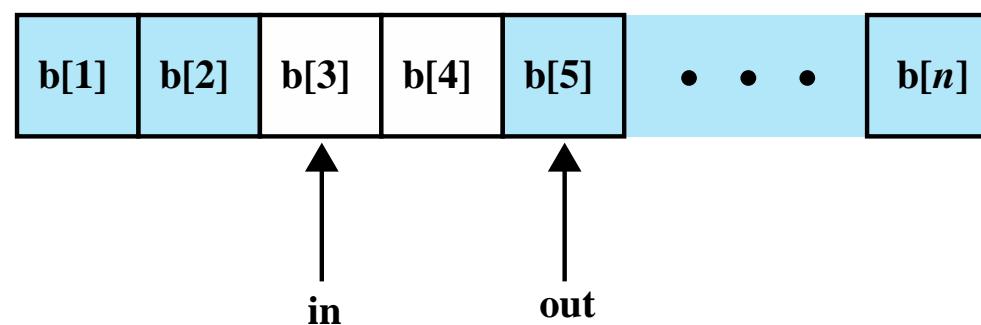
↳ semafori  
come prima

```
void producer()  
{  
    while (true) {  
        produce();  
        semWait(s);  
        append();  
        semSignal(s);  
        semSignal(n);  
    }  
}
```

```
void consumer()  
{  
    while (true) {  
        semWait(n);  
        semWait(s);  
        take();  
        semSignal(s);  
        consume();  
    }  
}
```



(a)



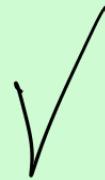
(b)

Figure 5.12 Finite Circular Buffer for the Producer/Consumer Problem

# Bounded-buffer Producer/Consumer

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
```

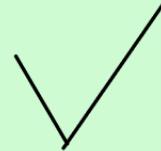
```
void producer()
{
    while (true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
```



# Bounded-buffer Producer/Consumer

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
```

```
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}
```



# Bounded-buffer Producer/Consumer

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
```

```
void producer()
{
    while (true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
```

```
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}
```

- ↳ What if I swap pairs of adjacent semWait or semSignal operations???
- ↳ What if I have only a producer or only a reader???
- ↳ SATURA IL BUFFER      ↳ NON ENTRA MAI

DEADLOCK

non succede niente

# Bounded-buffer P/C using Monitors

```
void producer()
{
    char x;
    while (true) {
        produce(x);
        append(x);
    }
}

void consumer()
{
    char x;
    while (true) {
        take(x);
        consume(x);
    }
}
```

## Monitors:

- local data accessible only by the monitor's procedures
- only one process may be executing in the monitor at a time
- programmer can decide when a procedure running in the monitor should stop or resume depending on a condition predicate

Thus, consume() and produce() do not need to know how take() and append() are implemented inside the monitor...

# Bounded-buffer P/C using Monitors

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];
int nextin, nextout;
int count;
cond notfull, notempty; /* condition variables for synchronization */

Init: nextin = nextout = count = 0;
                                /* space for N items */
                                /* buffer pointers */
                                /* number of items in buffer */
```

```
void append (char x)
{
    if (count == N) cwait(notfull);
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* one more item in buffer */
    csignal(notempty);
}
```

Wait if buffer is full, notify pending consumer process on exit (if any)

# Bounded-buffer P/C using Monitors

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];
int nextin, nextout;
int count;
cond notfull, notempty; /* condition variables for synchronization */

Init: nextin = nextout = count = 0;
                                /* space for N items */
                                /* buffer pointers */
                                /* number of items in buffer */
```

```
void take (char x)
{
    if (count == 0) cwait(notempty);
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    csignal(notfull);
}
```

Wait if buffer is empty, notify pending producer process on exit (if any)  
(By the way, this algorithm supports *multiple consumers* too...)

```

const int
    capacity = /* buffering capacity */ ;
    null =/* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
        receive (mayproduce, pmsg);
        pmsg = produce();
        send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
        receive (mayconsume, cmsg);
        consume (cmsg);
        send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++) send (mayproduce,
null);
    parbegin (producer, consumer);
}

```

**Figure 5.21**

## A Solution to the Bounded-Buffer Producer/Consu mer Problem Using Messages

Init. Mayproduce is full

Init. Mayconsume is empty

→ varianti di prime

# Readers/Writers Problem

- A data area is shared among many processes
  - some processes only read the data area (readers)
  - some processes only write to the data area (writers)
- Conditions that must be satisfied:
  1. any number of readers may simultaneously read the file
  2. only one writer at a time may write to the file
  3. if a writer is writing to the file, no reader may read it

# Readers/Writers Problem Using Semaphores

```
/* program readersandwriters */
int readcount = 0;
semaphore x = 1, wsem = 1;
    ↳ semaphore scritto
```

Requirements:

- any number of readers may read simultaneously
- only one writer at a time may write
- when a writer is writing, no reader is allowed to read

```
while (true) {
    semWait (wsem);
    WRITEUNIT();
    semSignal (wsem);
}
```

writer()

- acquire exclusive lock using binary semaphore wsem

# Readers/Writers Problem Using Semaphores

```
while (true) {  
    semWait (x);  
    readcount++;  
    if (readcount == 1) semWait (wsem);  
    semSignal (x);  
    READUNIT();  
    semWait (x);  
    readcount--;  
    if (readcount == 0) semSignal (wsem);  
    semSignal (x);  
}
```

## reader()

- binary semaphore x to safely update variable readcount
- when there is only one reader ( $x==1$ ) lock wsem (no writes!)
- once the read is performed, unlock wsem if no reader is active  
(i.e.,  $readcount==0$ ) => *Writer may come, extension needed!!*

# Concurrency: Deadlock and Starvation Problems, more on Mutual Exclusion

Most slides have been adapted from «*Operating Systems: Internals and Design Principles*», 7/E W. Stallings (Chapter 6 + Appendix A).

*Sistemi di Calcolo 2*

*Instructor: Riccardo Lazzaretti*

## A real-world example

*“When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”*

*Statute passed by the Kansas State Legislature,  
early in the 20th century.*



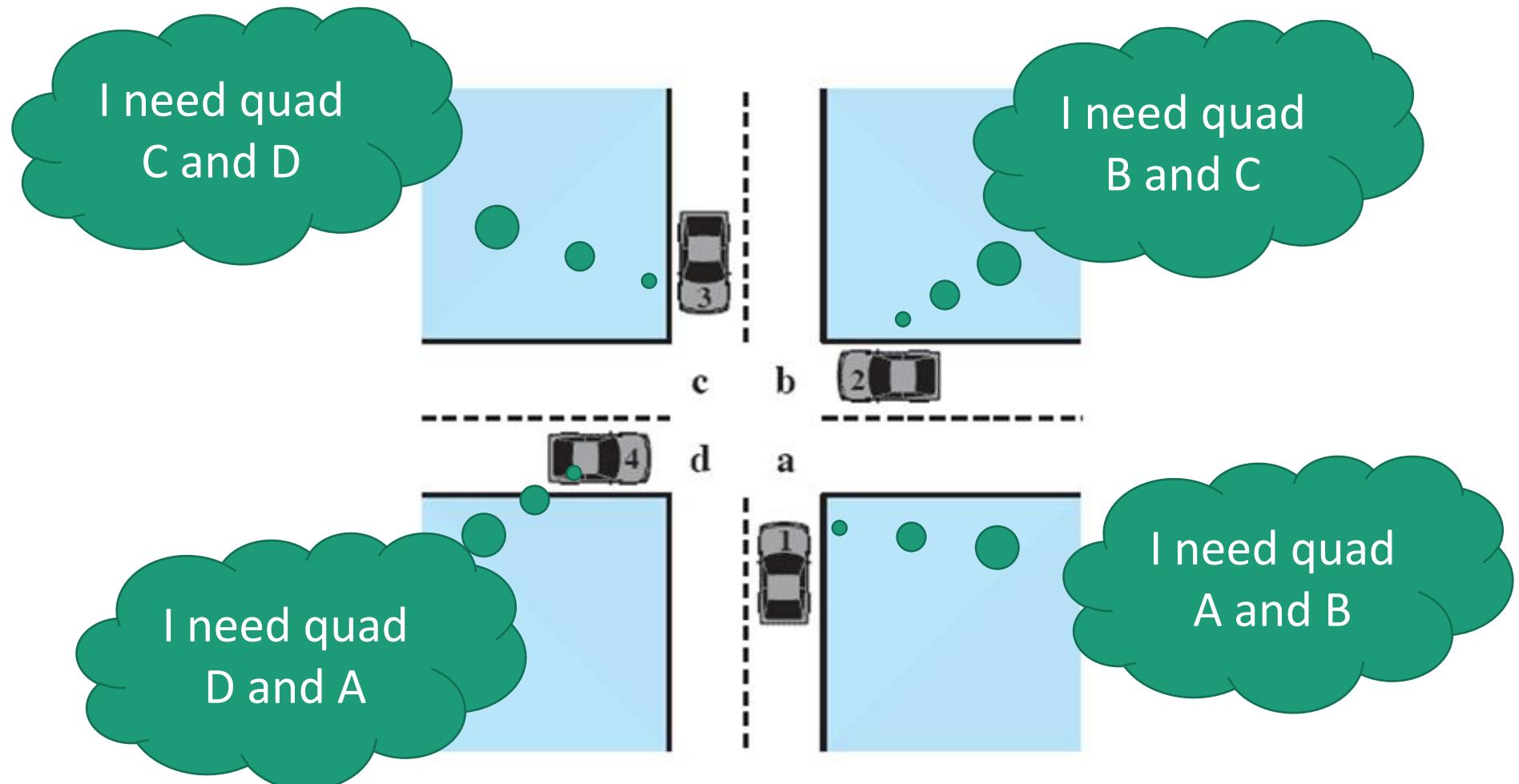
—*A TREASURY OF RAILROAD FOLKLORE*,  
B. A. Botkin and Alvin F. Harlow

# Deadlock

- The permanent blocking of a set of processes that either compete for system resources or communicate with each other
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
- Permanent
- No efficient solution

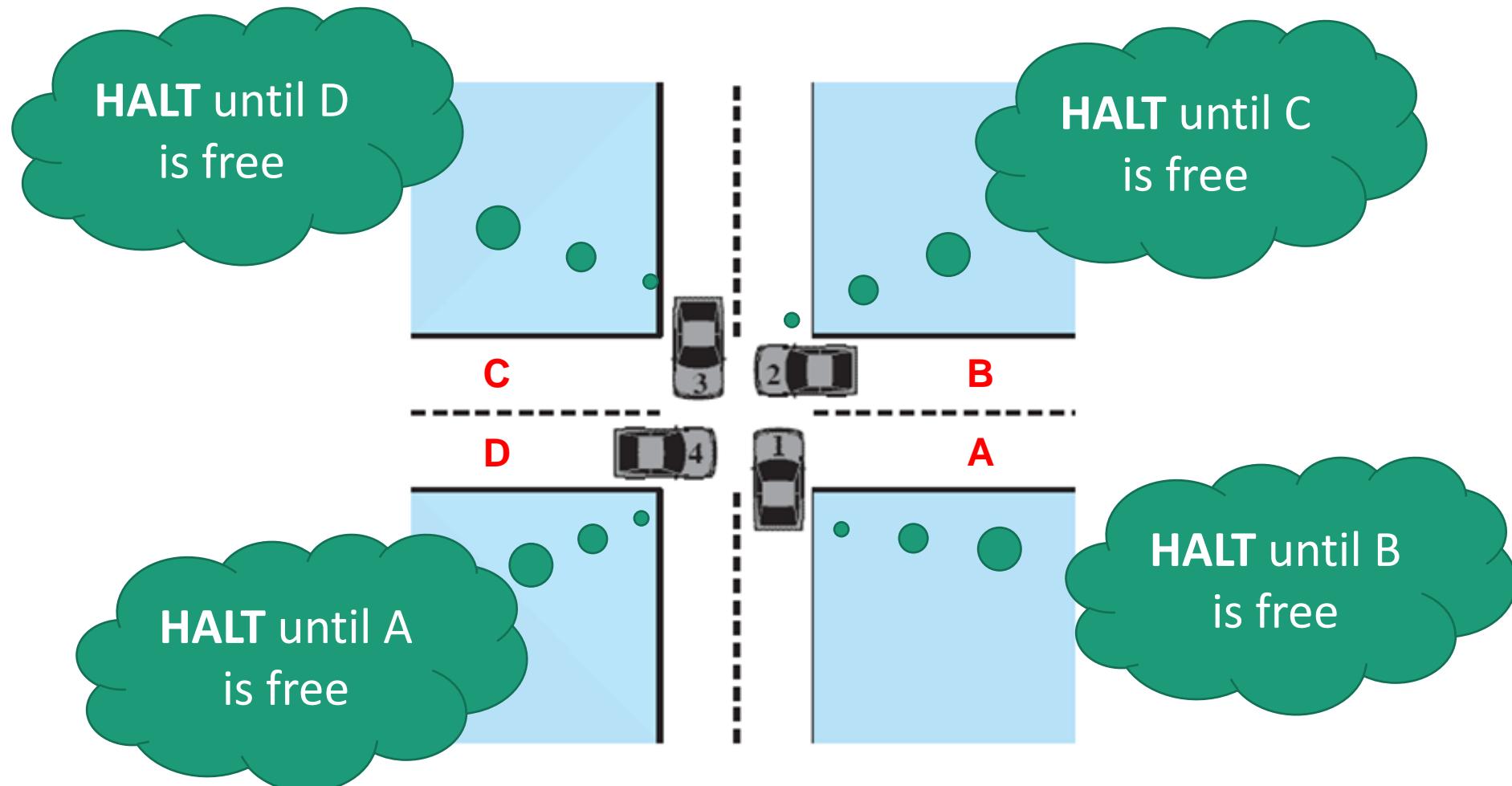


# Potential Deadlock



Se tornano indietro  
e avanti 2  
in LIVE LOCK

# Actual Deadlock



# Resource Categories

## Reusable resource

Can be safely used by only one process at a time, and it is not depleted by that use

*Examples:* processors, I/O channels, main and secondary memory, devices, and structures such as files, databases, and semaphores

## Consumable resource

One that can be created (produced) and destroyed (consumed)

*Examples:* interrupts, signals, messages, and information in I/O buffers

# Reusable Resources - Example

|                |                  | D è HARDISK                       | T è PENNA USB |
|----------------|------------------|-----------------------------------|---------------|
| Step           | Action           | Process P                         | Process Q     |
| p <sub>0</sub> | Request (D)      |                                   |               |
| p <sub>1</sub> | Lock (D)         |                                   |               |
| p <sub>2</sub> | Request (T)      |                                   |               |
| p <sub>3</sub> | Lock (T)         |                                   |               |
| p <sub>4</sub> | Perform function |                                   |               |
| p <sub>5</sub> | Unlock (D)       |                                   |               |
| p <sub>6</sub> | Unlock (T)       |                                   |               |
|                |                  | nimangons<br>blocket;<br>DEADLOCK |               |
| q <sub>0</sub> | Request (T)      |                                   |               |
| q <sub>1</sub> | Lock (T)         |                                   |               |
| q <sub>2</sub> | Request (D)      |                                   |               |
| q <sub>3</sub> | Lock (D)         |                                   |               |
| q <sub>4</sub> | Perform function |                                   |               |
| q <sub>5</sub> | Unlock (T)       |                                   |               |
| q <sub>6</sub> | Unlock (D)       |                                   |               |

Figure 6.4 Example of Two Processes Competing for Reusable Resources

## Example 2: Memory Request

- Space is available for allocation of 200Kbytes, and the following sequence of events occur:

P1

...

**Request 80 Kbytes;**

...

**Request 60 Kbytes;**

P2

...

**Request 70 Kbytes;**

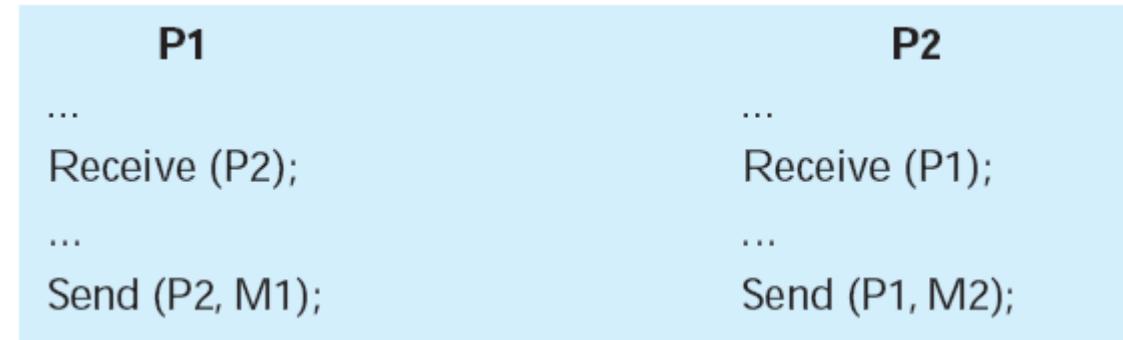
...

**Request 80 Kbytes;**

- Deadlock occurs if both processes progress to their second request

# Consumable Resources Deadlock

- Consider a pair of processes, in which each process attempts to receive a message from the other process and then sends a message to the other process:



- Deadlock occurs if the Receive is blocking

# Conditions for Deadlock

me non  
sufficient:

| Mutual Exclusion  | Hold-and-Wait  | No Pre-emption  | Circular Wait  |
|---|--|---|--|
| <ul style="list-style-type: none"><li>only one process may use a resource at a time</li></ul> <p>  <br/>se tutti possono accedervi non c'è DEADLOCK</p> | <ul style="list-style-type: none"><li>a process may hold allocated resources while awaiting assignment of others</li></ul> | <ul style="list-style-type: none"><li>no resource can be forcibly removed from a process holding it</li></ul> | <ul style="list-style-type: none"><li>a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain</li></ul> |

The first three conditions are necessary but not sufficient

qualcuno è  
in attesa di un altro

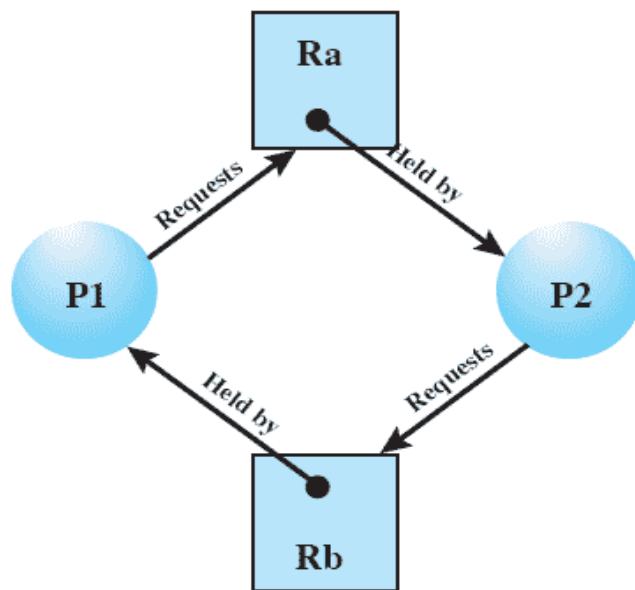
# Resource Allocation Graphs



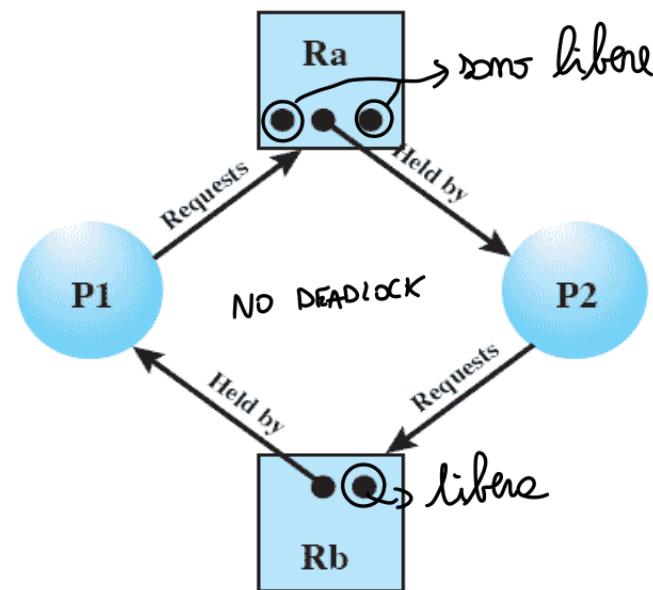
(a) Resource is requested



(b) Resource is held

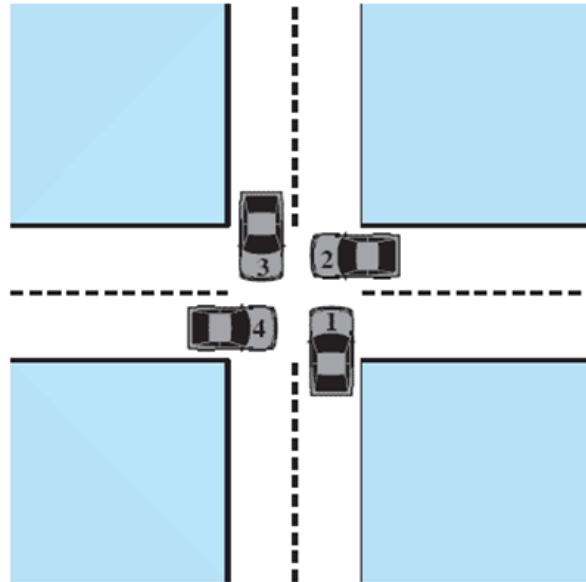


(c) Circular wait

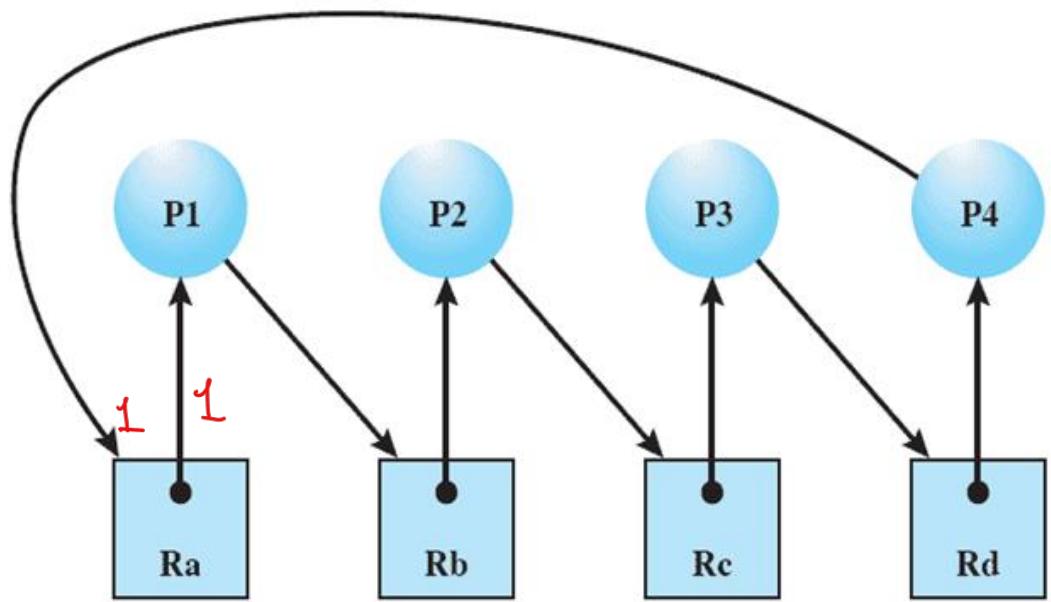


(d) No deadlock

# Resource Allocation Graphs



CIRCULAR WAITING  
↳ DEADLOCK



# Dealing with Deadlock

- Three general approaches exist for dealing with deadlock:

## Prevent Deadlock

- adopt a policy that eliminates one of the conditions

## Avoid Deadlock

- make the appropriate dynamic choices based on the current state of resource allocation

## Detect Deadlock

- attempt to detect the presence of deadlock and take action to recover

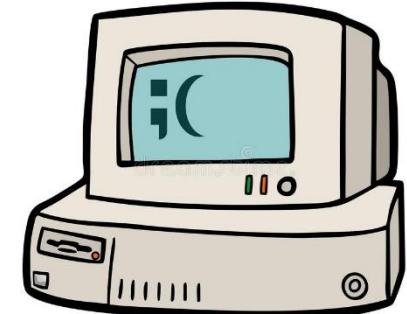
# Deadlock Prevention Strategy

- Design a system in such a way that the possibility of deadlock is excluded
- Two main methods:
  - Indirect
    - prevent the occurrence of one of the three necessary conditions
  - Direct
    - prevent the occurrence of a circular wait

# Deadlock Condition Prevention

- **Mutual Exclusion**

- if access to a resource requires mutual exclusion then it must be supported by the OS
- usually, cannot be disallowed
- we can relax it under some conditions
  - Ex: access to file must be granted in mutual exclusion
  - Read access can be allowed to multiple processes



- **Hold and Wait**

- require that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously
- Inefficient:
  - 1. Process can wait for a long time before all the resources are available
  - 2. Process can hold resources for a long time, even when not used
- Process could not know in advance which resources it will need

# Deadlock Condition Prevention

- **No Preemption**

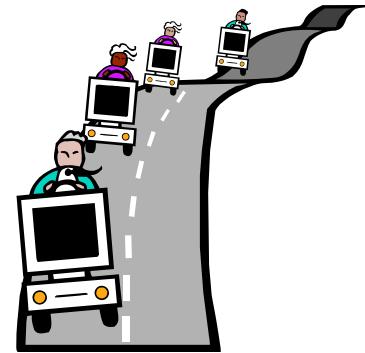
- if a process holding certain resources is denied a further request, that process must release its original resources and request them again; alternatively, OS may preempt the second process and require it to release its resources
- a process must have higher priority than the other one
- approach practical only for resources for which the state can be saved and later restored (e.g., CPU)

- **Circular Wait** EFFICACE MA COMPLESSO

- define a linear ordering of resource types
- if a resource of type R has been assigned to a process, the latter can only request resources whose types follow R in the order
- same problems of hold and wait

↳ se le devono chiedere con un certo ordine, potranno aspettare molti

# Deadlock Avoidance



- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Requires knowledge of future process requests !
- Allows more concurrency than deadlock prevention



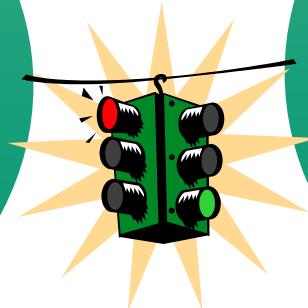
# Two Approaches to Deadlock Avoidance

## Deadlock Avoidance

non gli dir access  
1 alle risorse

### Resource Allocation Denial

- do not grant an incremental resource request to a process if this allocation might lead to deadlock



non lo initiliz  
1 propri, chetica

### Process Initiation Denial

- do not start a process if its demands might lead to deadlock



# Process Initiation Denial

RICHIESTE

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 6  |

Resource vector R

P1      R1      R2      R3 ] OCCUPANO  
P2      6      1      3      TUTTE LE  
P3      3      1      4      RISORSE  
P4      4      2      2      DISPONIBILI

Claim matrix C

process:

- OS allows the execution of a new process only if the sum of the claim resources of running process + claim resources of new process is lower than the available resources
- If P1 and P2 are running, P3 and P4 cannot be executed

- se P2 termina allora P3 può partire, P4 no per R2
- se anche P1 termina, P4 può partire
- se P1 termina e P2 no, P3 e P4 devono aspettare

# Determination of a Safe State

- A state is safe if a sequence of resource allocation exists such that does not result in deadlock
- State of a system consisting of four processes and three resources
- Allocations have been made to the four processes

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 3  | 2  | 2  |
| P2 | 6  | 1  | 3  |
| P3 | 3  | 1  | 4  |
| P4 | 4  | 2  | 2  |

Claim matrix C

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 1  | 0  | 0  |
| P2 | 6  | 1  | 2  |
| P3 | 2  | 1  | 1  |
| P4 | 0  | 0  | 2  |

Allocation matrix A

*P2 può terminare 1 R3 AVAILABLE*

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 2  | 2  | 2  |
| P2 | 0  | 0  | 1  |
| P3 | 1  | 0  | 3  |
| P4 | 4  | 2  | 0  |

*C - A*

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 6  |

Resource vector R

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 0  | 1  | 1  |

Available vector V

(a) Initial state

*Se ottiene uno più avvincente  
e terminazione allora è SAFE*

Amount of existing resources

Resources available after allocation

*↳ qualche risorsa  
dovrebbe essere  
disponibile*

# P2 Runs to Completion

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 3  | 2  | 2  |
| P2 | 0  | 0  | 0  |
| P3 | 3  | 1  | 4  |
| P4 | 4  | 2  | 2  |

Claim matrix C

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 1  | 0  | 0  |
| P2 | 0  | 0  | 0  |
| P3 | 2  | 1  | 1  |
| P4 | 0  | 0  | 2  |

Allocation matrix A

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 2  | 2  | 2  |
| P2 | 0  | 0  | 0  |
| P3 | 1  | 0  | 3  |
| P4 | 4  | 2  | 0  |

$C - A$

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 6  |

Resource vector R

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 6  | 2  | 3  |

Available vector V

(b) P2 runs to completion

# P1 Runs to Completion

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 0  | 0  | 0  |
| P2 | 0  | 0  | 0  |
| P3 | 3  | 1  | 4  |
| P4 | 4  | 2  | 2  |

Claim matrix C

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 0  | 0  | 0  |
| P2 | 0  | 0  | 0  |
| P3 | 2  | 1  | 1  |
| P4 | 0  | 0  | 2  |

Allocation matrix A

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 0  | 0  | 0  |
| P2 | 0  | 0  | 0  |
| P3 | 1  | 0  | 3  |
| P4 | 4  | 2  | 0  |

C - A

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 6  |

Resource vector R

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 7  | 2  | 3  |

Available vector V

(c) P1 runs to completion

# P3 Runs to Completion

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 0  | 0  | 0  |
| P2 | 0  | 0  | 0  |
| P3 | 0  | 0  | 0  |
| P4 | 4  | 2  | 2  |

Claim matrix C

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 0  | 0  | 0  |
| P2 | 0  | 0  | 0  |
| P3 | 0  | 0  | 0  |
| P4 | 0  | 0  | 2  |

Allocation matrix A

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 0  | 0  | 0  |
| P2 | 0  | 0  | 0  |
| P3 | 0  | 0  | 0  |
| P4 | 4  | 2  | 0  |

C - A

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 6  |

Resource vector R

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 4  |

Available vector V

(d) P3 runs to completion

Thus, the state defined originally is a safe state

STATE

SAFE

# Determination of an Unsafe State

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 3  | 2  | 2  |
| P2 | 6  | 1  | 3  |
| P3 | 3  | 1  | 4  |
| P4 | 4  | 2  | 2  |

Claim matrix C

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 1  | 0  | 0  |
| P2 | 5  | 1  | 1  |
| P3 | 2  | 1  | 1  |
| P4 | 0  | 0  | 2  |

Allocation matrix A

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 2  | 2  | 2  |
| P2 | 1  | 0  | 2  |
| P3 | 1  | 0  | 3  |
| P4 | 4  | 2  | 0  |

C - A

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 6  |

Resource vector R

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 1  | 1  | 2  |

Available vector V

(a) Initial state

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 3  | 2  | 2  |
| P2 | 6  | 1  | 3  |
| P3 | 3  | 1  | 4  |
| P4 | 4  | 2  | 2  |

Claim matrix C

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 2  | 0  | 1  |
| P2 | 5  | 1  | 1  |
| P3 | 2  | 1  | 1  |
| P4 | 0  | 0  | 2  |

Allocation matrix A

|    | R1 | R2 | R3 |
|----|----|----|----|
| P1 | 1  | 2  | 1  |
| P2 | 1  | 0  | 2  |
| P3 | 1  | 0  | 3  |
| P4 | 4  | 2  | 0  |

C - A

✓ ALLOC

STATE SAFE

X DENIED

↑

↑

↑

↑

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 9  | 3  | 6  |

Resource vector R

|  | R1 | R2 | R3 |
|--|----|----|----|
|  | 0  | 1  | 1  |

Available vector V

(b) P1 requests one unit each of R1 and R3

↳ prima di concedere il rientro rimuove gli avvenimenti

# Deadlock Avoidance Logic

```
struct state {
    int resource[m];
    int available[m];
    int claim[n][m];
    int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
    < error >;                                /* total request > claim */
else if (request [*] > available [*])
    < suspend process >; NON HO RISORSE
else {
    < define newstate by:
    alloc [i,*] = alloc [i,*] + request [*];
    available [*] = available [*] - request [*];
}
if (safe (newstate))
    < carry out allocation >;
else {
    < restore original state >;
    < suspend process >;
}
```

(b) resource alloc algorithm

## BANKER'S ALGORITHM

# Deadlock Avoidance Logic

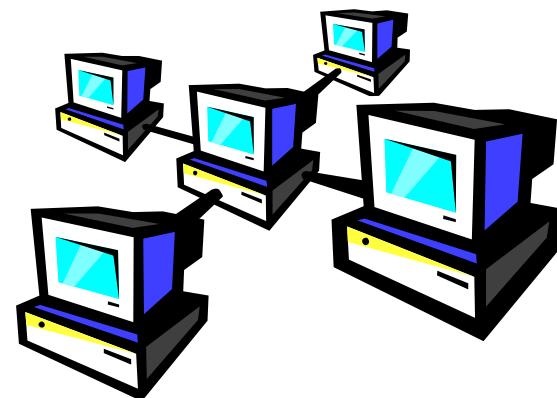
```
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
        <find a process Pk in rest such that
            claim [k,*] - alloc [k,*] <= currentavail;>
        if (found) {                                /* simulate execution of Pk */
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else possible = false;
    }
    return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)

Figure 6.9 Deadlock Avoidance Logic

# Deadlock Avoidance Advantages

- Process initiation denial is hardly optimal...
- ...while resource allocation denial has a few perks!
  - It is not necessary to preempt and roll back processes, as in deadlock detection (*we will discuss detection soon*)
  - It is less restrictive than deadlock prevention



# Deadlock Avoidance Restrictions

- Maximum resource requirement for each process must be stated in advance
- Processes under consideration must be independent and with no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources

entrombi pesonti



# Deadlock Strategies

Deadlock prevention strategies are very conservative

- limit access to resources by imposing restrictions on processes

Deadlock detection strategies do the opposite

- resource requests are granted whenever possible

# Deadlock Detection Algorithms

→ più usato

A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur.

Detection performed at each resource request:

- Pros: leads to early detection, algorithm is simple
- Cons: frequent checks consume considerable CPU time

↳ dispersione i check, riene  
fatti a intervalli regolari

# A Detection protocol

- We iteratively mark protocols that can be suspended or terminate
- If at the end all the protocols are marked we have no deadlock

|    | R1 | R2 | R3 | R4 | R5 |
|----|----|----|----|----|----|
| P1 | 0  | 1  | 0  | 0  | 1  |
| P2 | 0  | 0  | 1  | 0  | 1  |
| P3 | 0  | 0  | 0  | 0  | 1  |
| P4 | 1  | 0  | 1  | 0  | 1  |

Request matrix Q

|    | R1 | R2 | R3 | R4 | R5 |
|----|----|----|----|----|----|
| P1 | 1  | 0  | 1  | 1  | 0  |
| P2 | 1  | 1  | 0  | 0  | 0  |
| P3 | 0  | 0  | 0  | 1  | 0  |
| P4 | 0  | 0  | 0  | 0  | 0  |

Allocation matrix A

|  | R1 | R2 | R3 | R4 | R5 |
|--|----|----|----|----|----|
|  | 2  | 1  | 1  | 2  | 1  |

Resource vector

|  | R1 | R2 | R3 | R4 | R5 |
|--|----|----|----|----|----|
|  | 0  | 0  | 0  | 0  | 1  |

Allocation vector

1. P4 is marked because has no resources yet
2. P3 is marked because if we give resources it can terminate and release resources
3. Even if P3 releases resources, neither P1 nor P2 can terminate  
=> Deadlock!!!

# Deadline Detection - Recovery

Some approaches, in increasing order of sophistication:

- Abort all deadlocked processes (most common)
- Back up each deadlocked process to some previously defined checkpoint and restart all processes
- Successively abort deadlocked processes until deadlock no longer exists (*lin cada una sola volta*)
- Successively preempt resources until deadlock no longer exists (requires rollback mechanism)

# Deadlock Approaches

| Approach   | Resource Allocation Policy                                   | Different Schemes                         | Major Advantages  | Major Disadvantages   |
|------------|--|---|---|---|
| Prevention | Conservative; undercommits resources                         | Requesting all resources at once          | <ul style="list-style-type: none"> <li>• Works well for processes that perform a single burst of activity</li> <li>• No preemption necessary</li> </ul>                           | <ul style="list-style-type: none"> <li>• Inefficient</li> <li>• Delays process initiation</li> <li>• Future resource requirements must be known by processes</li> </ul> |
|            |  | Preemption                                | <ul style="list-style-type: none"> <li>• Convenient when applied to resources whose state can be saved and restored easily</li> </ul>   | <ul style="list-style-type: none"> <li>• Preempts more often than necessary</li> </ul>  |
|            |  | Resource ordering                         | <ul style="list-style-type: none"> <li>• Feasible to enforce via compile-time checks</li> <li>• Needs no run-time computation since problem is solved in system design</li> </ul> | <ul style="list-style-type: none"> <li>• Disallows incremental resource requests</li> </ul>   |
| Avoidance  | Midway between that of detection and prevention              | Manipulate to find at least one safe path | <ul style="list-style-type: none"> <li>• No preemption necessary</li> </ul>   | <ul style="list-style-type: none"> <li>• Future resource requirements must be known by OS</li> <li>• Processes can be blocked for long periods</li> </ul>               |
| Detection  | Very liberal; requested resources are granted where possible | Invoke periodically to test for deadlock  | <ul style="list-style-type: none"> <li>• Never delays process initiation</li> <li>• Facilitates online handling</li> </ul>  | <ul style="list-style-type: none"> <li>• Inherent preemption losses</li> </ul>  |

# Dining Philosophers Problem

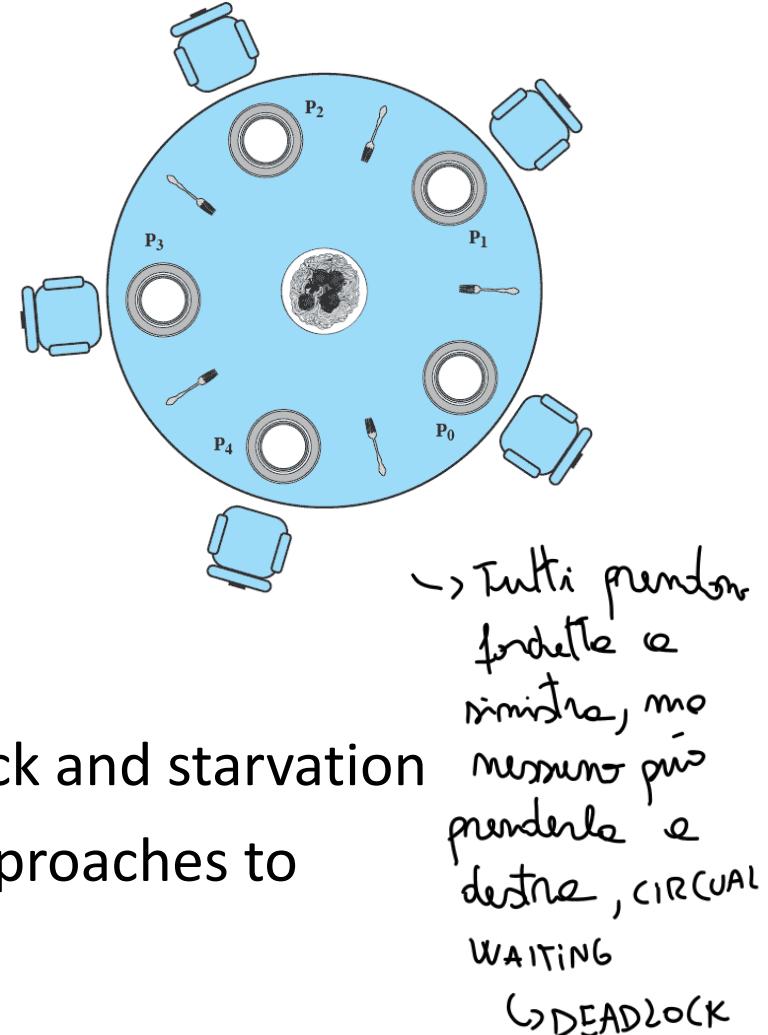
No two philosophers can use the same fork at the same time (mutual exclusion)

No philosopher must starve (avoid deadlock & starvation)

Usare FORCHETTA A SINISTRA E DESTRA

*Why so relevant?*

- illustrates basic problems in deadlock and starvation
- standard test case for evaluating approaches to synchronization



# Using N semaphores?

→ Semaforo  
+ forchette

```
semaphore fork [5] = {1};  
void philosopher (int i)  
{  
    while (true) {  
        think();  
        wait (fork[i]);  
        wait (fork [(i+1) mod 5]);  
        eat(); //ZONA CRITICA  
        signal(fork [(i+1) mod 5]);  
        signal(fork[i]);  
    }  
}
```

*Try to pick the  $i$ -th fork on the left, then  $(i+1)$ -th fork on the right: the code above will easily lead to deadlock*

# A Correct Solution . . .

```
semaphore fork[5] = {1};  
semaphore room = {4}; → Impedisce un filosofo  
void philosopher (int i) di sedersi se ci  
{ sono 4 posti occupati:  
    while (true) { (lavora sulle risorse)  
        think();  
        wait (room);  
        wait (fork[i]);  
        wait (fork [(i+1) mod 5]);  
        eat();  
        signal (fork [(i+1) mod 5]);  
        signal (fork[i]);  
        signal (room);  
    }  
}
```

# Software approaches to mutual exclusion

Mutual exclusion can be implemented at software level too

Scenario: processes communicate via a central memory on a single/multi-processor machine

Assumption: mutual exclusion at the memory access level

- read/write operations for the same location are serialized by some sort of memory arbiter (the order is unknown)
- No support in OS, hardware, or programming language

# Dekker algorithm

Dijkstra reported an algorithm for mutual exclusion designed by Dutch mathematical Dekker

2 processes! Solo due processi

We develop it by attempts

# First attempt Si me busy waiting

```
/* global */  
int turn = 0; → modifica turn è atomica  
  
// assignments valid for P0 (flip for P1)  
int me = 0, other = 1;  
while (true) {  
    /*NCS*/  
    while (turn != me) → aspetto se turno non è il mio.  
        /* busy wait */ ;  
    /* CS */  
    turn = other;  
}
```

## Second attempt NO

```
/* global */  
boolean flag[2] = {false, false};  
  
// assignments valid for P0 (flip for P1)  
int me = 0, other = 1;  
  
while (true) {  
    /*NCS*/  
    while (flag[other]) → De forms in contention  
        /* busy wait */ ;  
    flag[me] = true;  
    /* CS */  
    flag[me] = false;  
}
```

→ De forms in contention  
means, non  
ignoring mutual  
exclusion

# Third attempt NO DEADLOCK

```
// assignments valid for P0 (flip for P1)
int me = 0, other = 1;

while (true) {
    /*NCS*/
    flag[me] = true;           → garantisce mutua
    while (flag[other])        esclusione, me non
        /* busy wait */ ;     evita la DEADLOCK,
    /* CS */                   interrompi mettendo
    flag[me] = false;          true .
```

## Fourth attempt Ni potrebbe esserci LIVELOCK

```
// assignments valid for P0 (flip for P1)
int me = 0, other = 1;

while (true) {
    /*NCS*/
    flag[me] = true;
    while (flag[other]) {
        flag[me] = false;
        /* delay */           → garantisce mutua
        flag[me] = true;      esclusione, ma potrebbe
    }
    /* CS */
    flag[me] = false;
}
```

durante mutua esclusione, ma potrebbe accadere: LIVELOCK, potrebbero entrambi cambiare flag con lo stesso DELAY, se non perfettamente sincronizzati:

# Correct Solution: Dekker's Algorithm

```
int me = 0, other = 1; // P0 (flip for P1)

while (true) {
    /*NCS*/
    flag[me] = true;
    while (flag[other]) {
        if (turn == other) {
            flag[me] = false;
            while (turn == other) /* busy wait */ ;
            flag[me] = true;      ↳ hits LIVELOCK on
        }                      leturnazione, do
    }                      precedente e chiude
    /* CS */                de più temp.
    turn = other;
    flag[me] = false;
}
```

# Dekker's Algorithm, informally

```
flag[me] = true;  
while (flag[other]) {  
    if (turn==other) {  
        flag[me] = false;  
        while (turn==other);  
        flag[me] = true;  
    }  
}  
/* CS */  
turn = other;  
flag[me] = false;
```

*I want to enter*  
*If you want to enter*  
*and if it's your turn*  
*I don't want any more*  
*If it's your turn I'll wait*  
*I want to enter*

*You can enter next*  
*I don't want any more*

Credits: <https://cs.stackexchange.com/q/12621>

# Dijkstra (1965)

Dijkstra generalized Dekker's algorithm to deal with N entities

He also provided definitions for mutual exclusion (ME), no deadlock (ND), and no starvation (NS) properties

=> notice that NS implies ND

The controller does not interfere in interests

N statics

```
/* global storage */  
boolean interested[N] = {false, ..., false}  
boolean passed[N]     = {false, ..., false}  
int k = <any> TURNO    // k ∈ {0, 1, ..., N-1}  
  
/* local info */  
int i = <entity ID>   // i ∈ {0, 1, ..., N-1}
```

Potrebbe esserci STARVATION

# Dijkstra's Algorithm

```
while (true) {  
    /*NCS*/  
    1. interested[i] = true  
    2. while (k != i) {  
        3.     passed[i] = false  
        4.     if (!interested[k]) then k = i  
            }  
    5.     passed[i] = true  
    6.     for j in 1 ... N except i do  
        7.         if (passed[j]) then goto 2  
    8.         <critical section>  
    9.     passed[i] = false; interested[i] = false  
}
```

Non è atomico, potrebbe esserci  
race condition

retrava un  
eltra che i processi  
rinviano a 2.

entrambi  
tornano  
a 2., ma  
non con K=1

# A closer look

```
1. interested[i] = true
2. while (k != i) {
3.     passed[i] = false
4.     if (!interested[k]) then k = i
    }
5. passed[i] = true
6. for j in 1 ... N except i do
7.     if (passed[j]) then goto 2
```

Trying protocol

1: process i shows interest in entering CS

2-4: k “chooses” among processes that showed interest

5: phase one passed by process i

6-7: restart if more than one process passed phase one

INTERSECTION

# Example ( $N \geq 4$ , initially $k = 4$ )

|                 | P1              | P2              | P3 |
|-----------------|-----------------|-----------------|----|
| intr[1] = true  |                 |                 |    |
| while (k!=1)    |                 |                 |    |
| pass[1] = false |                 |                 |    |
|                 | intr[2] = true  |                 |    |
|                 | while (k!=2)    |                 |    |
|                 | pass[2] = false |                 |    |
|                 |                 | intr[3] = true  |    |
|                 |                 | while (k!=3)    |    |
|                 |                 | pass[3] = false |    |
| if (!intr[4])   |                 |                 |    |
|                 | if (!intr[4])   |                 |    |
|                 |                 | if (!intr[4])   |    |
| k = 1           |                 |                 |    |
|                 | k = 2           |                 |    |
|                 |                 | k = 3           |    |

} face  
condition

# Entering the Critical Section

Processes P1, P2, and P3 are now ready to execute lines 5-7

- only one of them will be able to access CS
- k=3 does not imply that P3 goes first: k is used only to solve conflicts, thus the scheduler may also let P1 or P2 in first!
- a conflict arises when two processes set pass[i]=true before the other has entered and then left the critical section

P3

```
pass[3] = true
for j in 1..N except 3 do
  if pass[j] goto 2 (skipped)
  <critical section>
  pass[3] = false
intr[3] = false
```

One possible conflict-free continuation: the scheduler lets P3 run with P1 and P2 still “stopped” at line 5

# P3 finished, conflict for P1 & P2?

| P1                                   | P2                                   |
|--------------------------------------|--------------------------------------|
| while (k!=1)                         |                                      |
| pass[1] = false                      |                                      |
|                                      | while (k!=2)                         |
|                                      | pass[2] = false                      |
| if (!intr[3]) <i>(true: P3 done)</i> |                                      |
|                                      | if (!intr[3]) <i>(true: P3 done)</i> |
| then k = 1                           |                                      |
| while (k!=1) <i>(skipped)</i>        |                                      |
| pass[1] = true                       |                                      |
|                                      | then k = 2                           |
|                                      | while (k!=2) <i>(skipped)</i>        |
| if (pass[2]) goto 2 <i>(taken!!)</i> |                                      |
|                                      | if (pass[1]) goto 2 <i>(taken!!)</i> |

When a conflict occurs, the algorithm goes back to line 2 and clears pass[i]... except for the process that set k last! (P2 here)

---

# Inter Process Communications

Most slides have been adapted from  
«The Linux Programming interface: A Linux and UNIX® System Programming Handbook», Michael Kerrisk

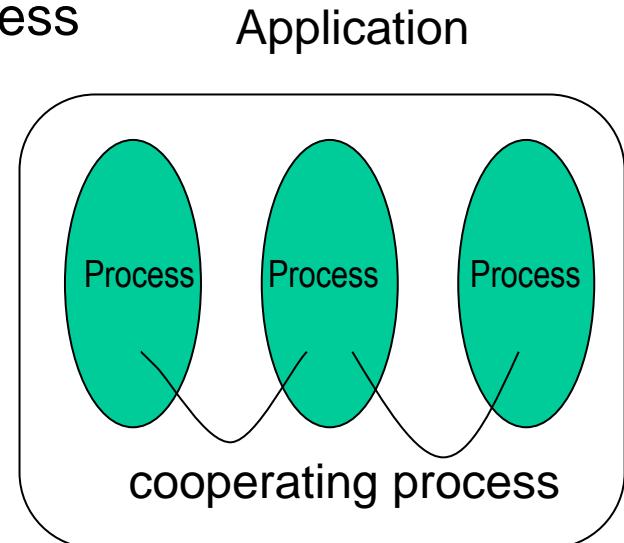
*Sistemi di Calcolo 2*  
*Riccardo Lazzaretti*

# Process Address Space

- A process can only access its address space
- Each process has its own address space
- Kernel can access everything

# Cooperating Processes and the need for Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
  - **Independent** process cannot affect or be affected by the execution of another process
  - **Cooperating** process can affect or be affected by the execution of another process
- Reasons for process cooperation
  - **Information sharing**
  - **Computation speed-up**
  - **Modularity** (application will be divided into modules/sub-tasks)
  - **Convenience** (may be better to work with multiple processes)

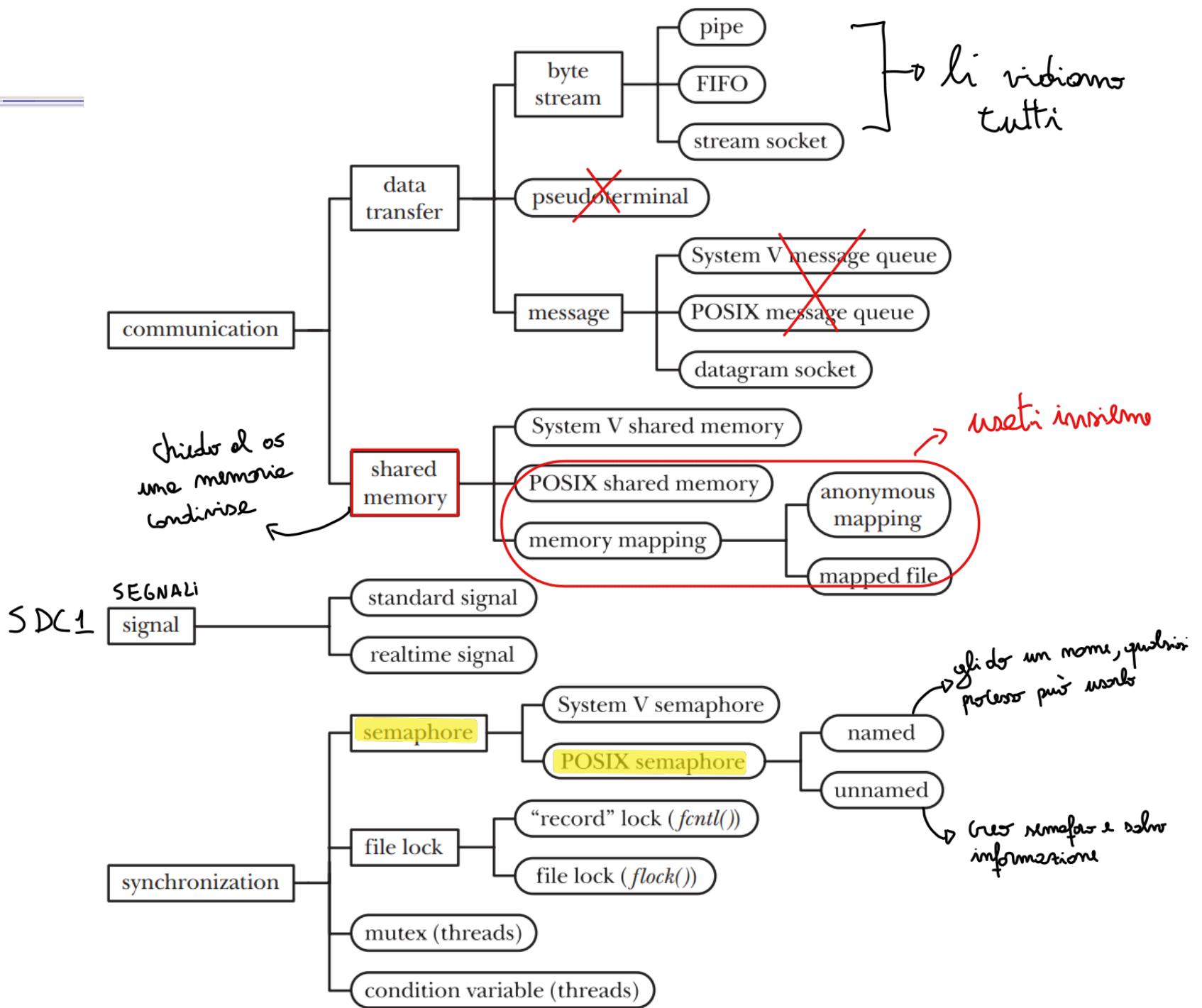


The overall application is designed to consist of cooperating processes

---

# Inter-process Communication (IPC)

Mechanism for processes to communicate and to synchronize their actions.

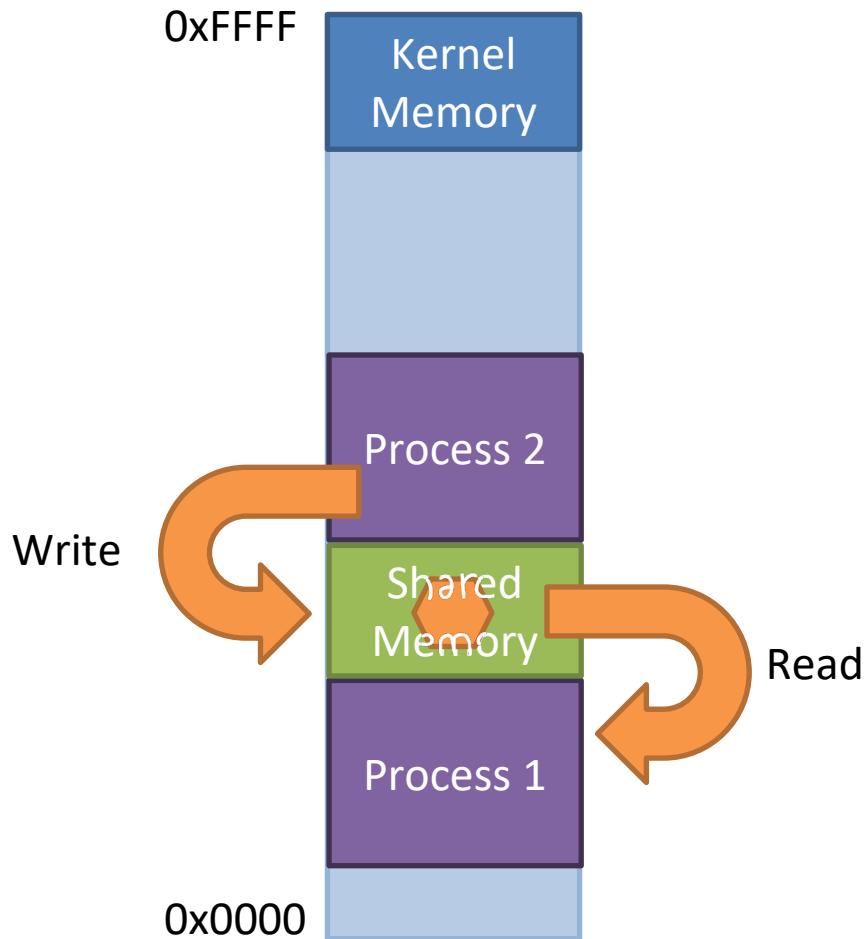


# IPC Mechanisms

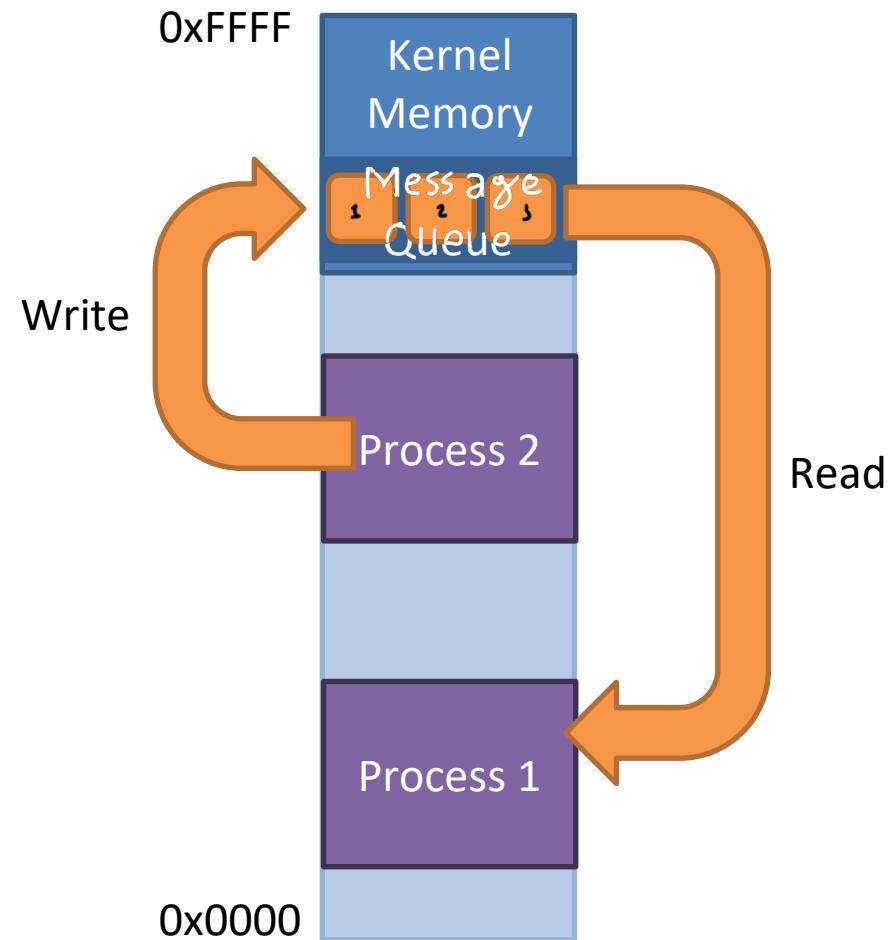
- Cooperating processes require a facility/mechanism for inter-process communication (IPC)
- There are two basic IPC models provided by most systems:
  - 1) Shared memory model  
processes use a shared memory to exchange data
  - 2) Message passing model  
processes send messages to each other through the kernel

# Communication models

## Shared Memory



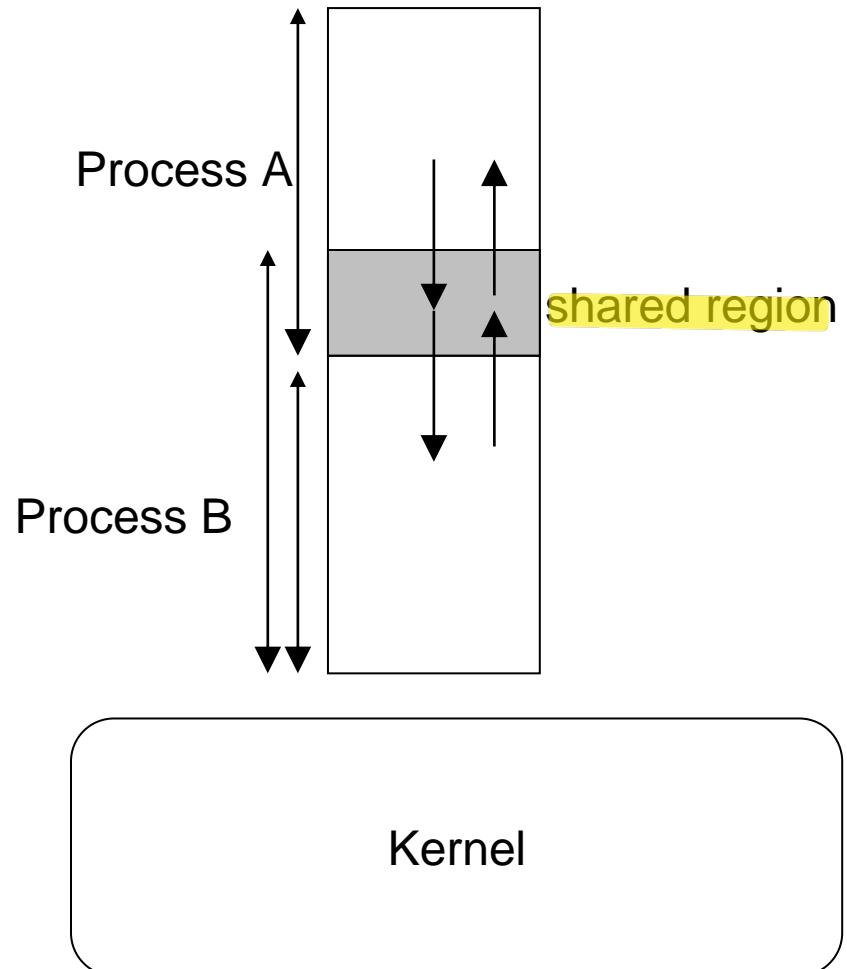
## Message Passing



# Shared Memory IPC Mechanism

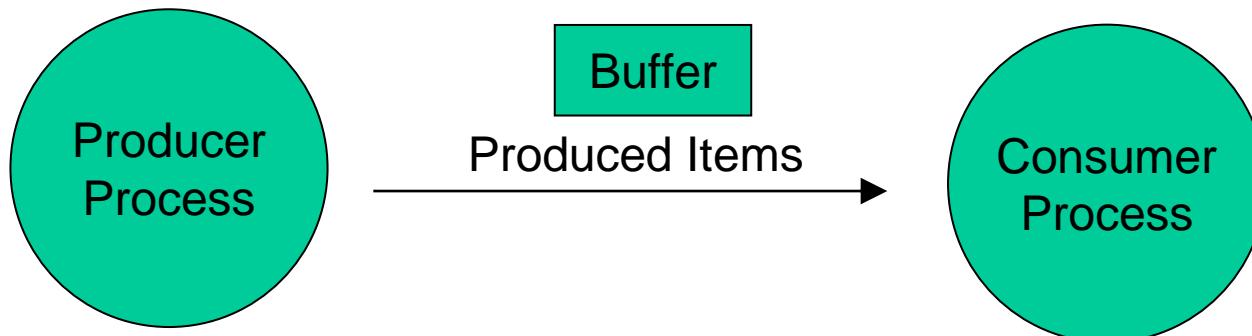
- A region of shared memory is established between (among) two or more processes.
  - via the help of the operating system kernel (i.e. **system calls**).
- Processes can **read and write** shared memory region (segment) directly as ordinary memory access (**pointer access**)
  - During this time, kernel is not involved.
  - Hence it is fast

(↳ deno degration  
imcommunication)



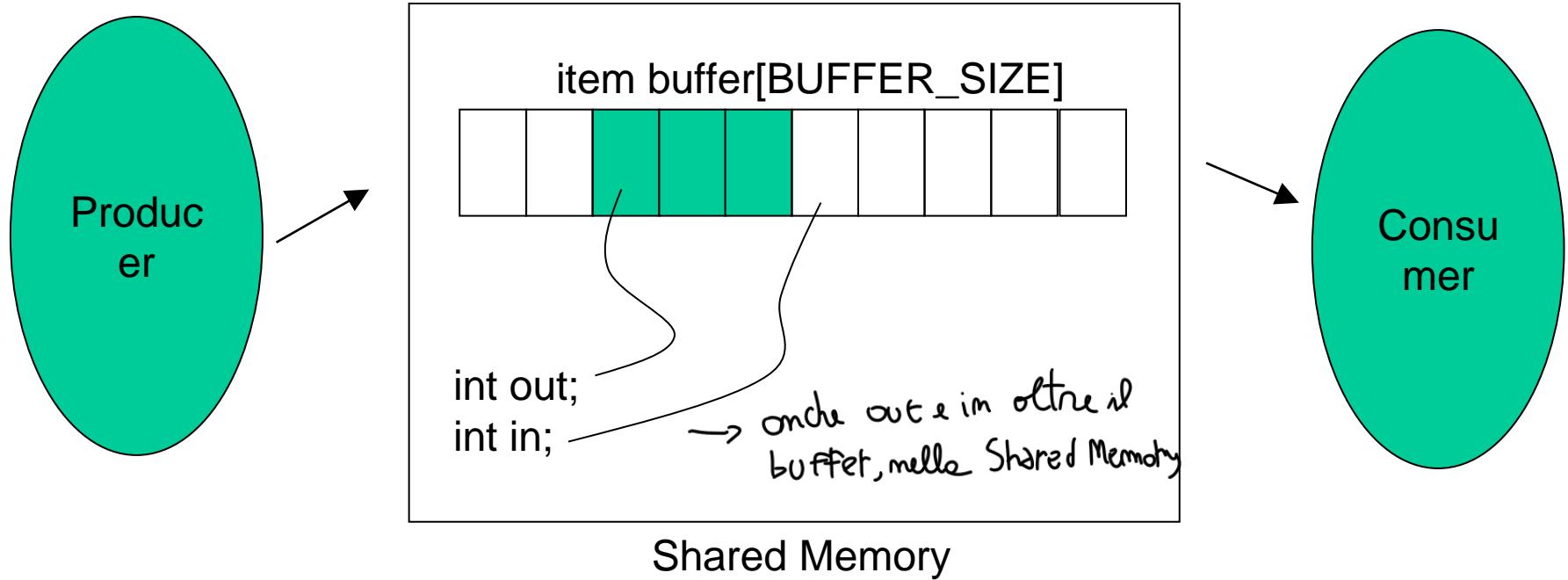
# Shared Memory IPC Mechanism

- To illustrate use of an IPC mechanism, a general model problem, we use the **producer-consumer problem with bounded buffer**



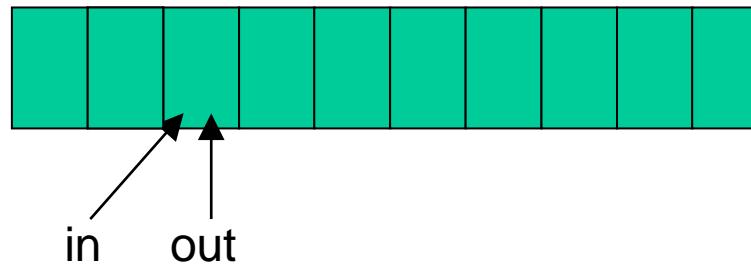
We can solve this problem via shared memory IPC mechanism

# Buffer State in Shared Memory



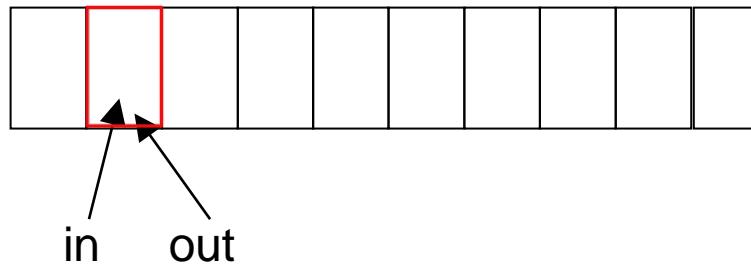
# Buffer State in Shared Memory

## Buffer Full



```
in == out; sem_empty.val == 0; sem_filled.val == BUFFER_SIZE
```

## Buffer Empty



```
in == out; sem_empty.val == BUFFER_SIZE; sem_filled.val == 0
```

# Bounded-Buffer - Producer and Consumer Code

```
while (true) {  
    /* Produce an item */  
    sem_wait(sem_empty); %named semaphores  
    sem_wait(sem_cs);  
    writeToBuffer (ITEM);  
    in = (in+1) % BUFFER_size;  
    sem_signal(sem_cs);  
    sem_signal(sem_filled);  
}
```

Producer

Buffer (an array)  
*in, out* integer variable

Shared Memory

```
while (true) {  
    sem_wait(sem_filled);  
    sem_wait(sem_cs);  
    readFromBuffer (ITEM);  
    out = (out +1) % BUFFER_size;  
    sem_signal(sem_cs);  
    sem_signal(sem_empty);  
    /* Consume the item */  
}
```

Consumer

# Posix Shared Memory API

- **shm\_open()** – create and/or open a shared memory page
  - Returns a file descriptor for the shared page
  - We can create a private shared memory
    - Only child processes can access the SHM too
- **ftruncate()** or **ftruncate()** – limit the size of the shared memory page
- **mmap()** – map the memory page into the processes address space *mappa e posso usá-la*
  - Now you can read/write the page using a pointer
- **close()** – close a file descriptor *(não pode mais ser usado)*
- **shm\_unlink()** – remove a shared page
  - Processes with open references may still access the page

```

/* Program to write some data in shared memory */
int main() {
    const int SIZE = 4096;

    const char * name = "MY_PAGE";
    const char * msg = "Hello World!";
    int shm_fd; (FILE DESCRIPTOR)
    char * ptr; (stringa di messaggi)
    ← primo bit delle shared memory)
    ↗ lettura e scrittura

    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    ftruncate(shm_fd, SIZE);
    ptr = (char *) mmap(0, SIZE, PROT_WRITE,
                           MAP_SHARED, shm_fd, 0); ↗ ci scrivo, ma
    sprintf(ptr, "%s", msg);
    close(shm_fd);
    return 0;
}

```

Non **um** (ike altrimenti  
chiuderebbe **shm\_fd**

```
/* Program to read some data from shared memory */
int main() {
    const int SIZE = 4096;

    const char * name = "MY_PAGE";
    int shm_fd;
    char * ptr;

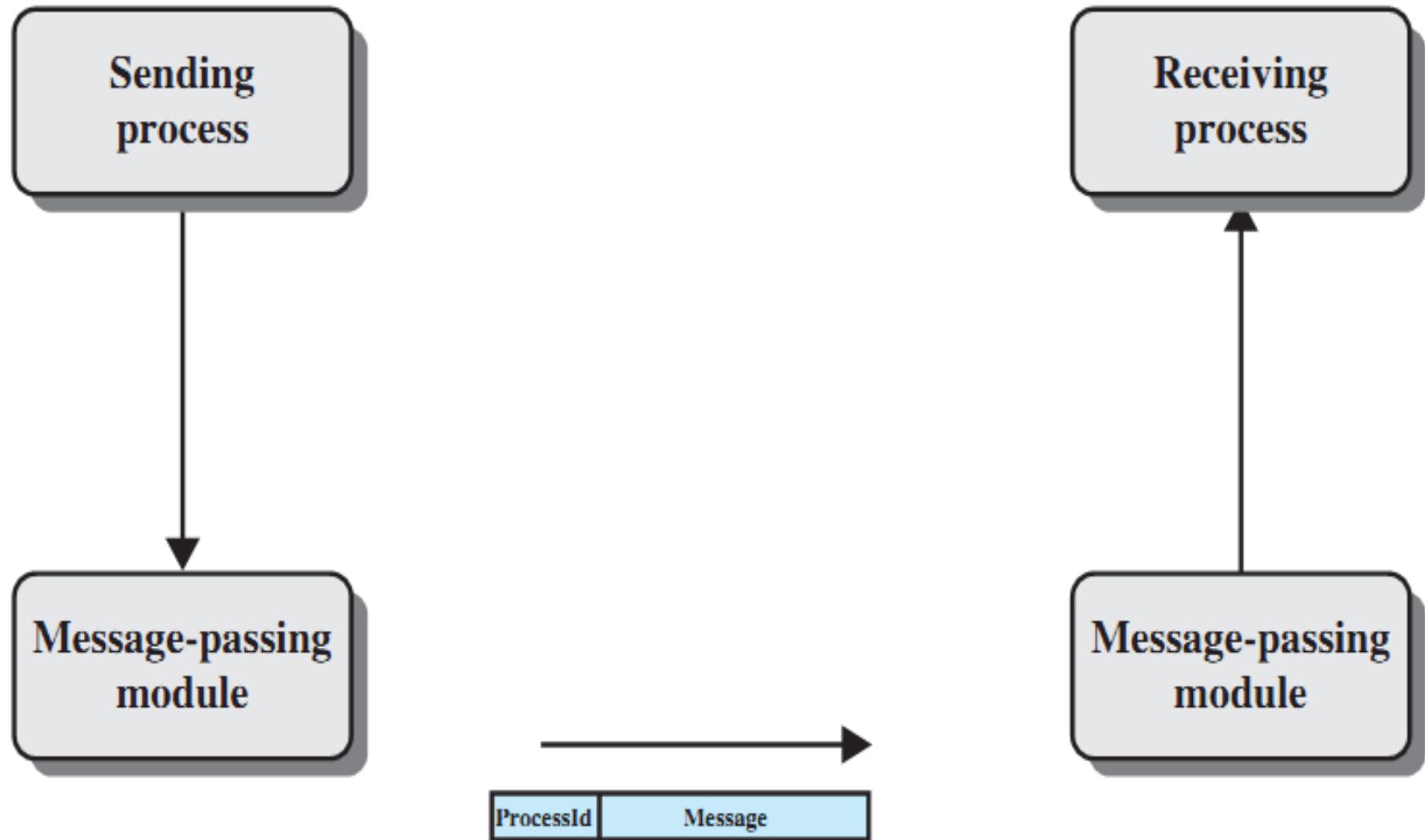
    shm_fd = shm_open(name, O_RDONLY, 0666);
    ptr = (char *) mmap(0, SIZE, PROT_READ,
                         MAP_SHARED, shm_fd, 0);
    printf("%"s\n", ptr); } non controlla errori,
    close(shm_fd); Bisogna fare attenti quando
    shm_unlink(shm_fd); si utilizza memoria.
    return 0;
}
```

# Message Passing IPC Mechanisms

- Message system – processes communicate with each other without resorting to shared variables.
- We have at least two primitives:
  - **send**(*destination, message*) or **send**(*message*)
  - **receive**(*source, message*) or **receive**(*message*)
- Message size is fixed or variable.

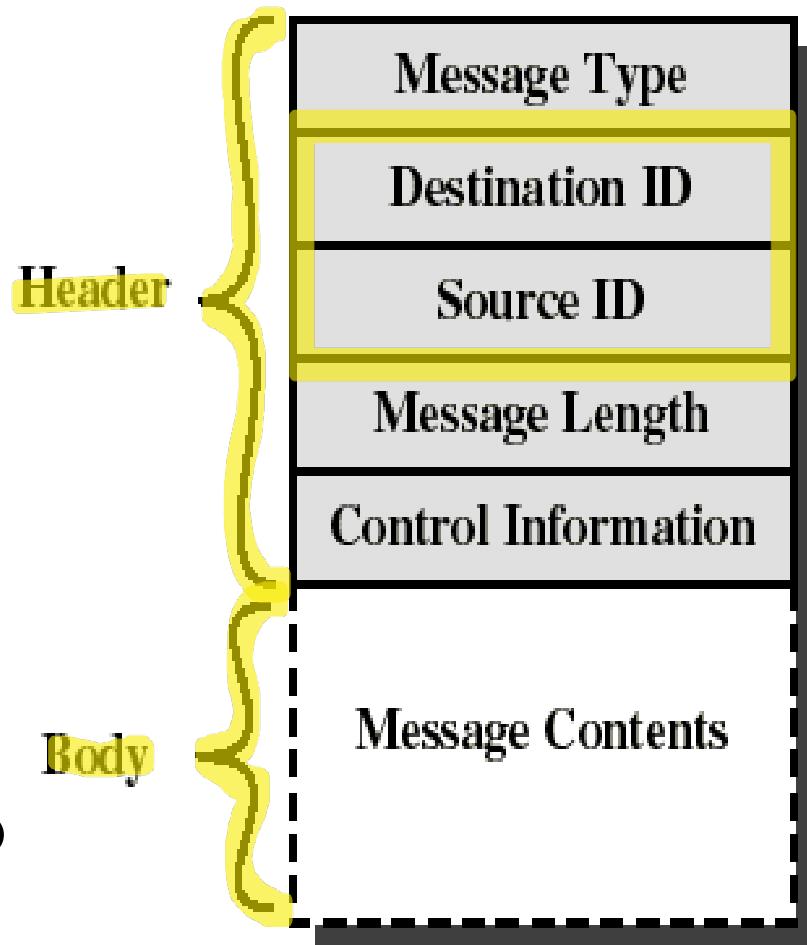


# Basic Message-passing Primitives



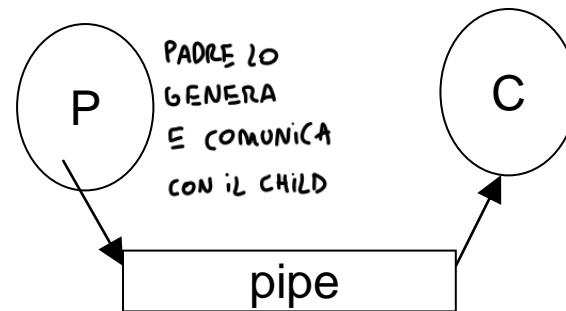
# Message format

- Consists of header and body of message.
- In Unix: no ID, only message type.
- Control info:
  - what to do if run out of buffer space.
  - sequence numbers.
  - priority.
- **Queuing discipline:** usually FIFO but can also include priorities.



# Message Passing: pipes

- Piped and Named-Pipes (FIFOs)
- In Unix/Linux:
  - A pipe enables one-way communication between a parent and child
  - It is easy to use.
  - When process terminates, pipe is removed automatically
  - `pipe()` system call



# Concetti di base sulle PIPE (1/2)

- permettono a più processi di comunicare come se stessero accedendo a dei file sequenziali
- il termine “pipe” significa tubo in Inglese, e la comunicazione avviene in modo monodirezionale
- una volta lette, le informazioni spariscono dalla PIPE e non possono più ripresentarsi
  - a meno che non vengano riscritte all'altra estremità del tubo
- a livello di OS, le PIPE non sono altro che buffer di dimensione più o meno grande (solitamente 4096 byte)
  - quindi è possibile che un processo venga bloccato se tenta di scrivere su una PIPE piena

## Concetti di base sulle PIPE (2/2)

- i processi che usano una PIPE devono essere “relazionati” come conseguenza di operazioni `fork()`
- le named PIPE (FIFO) permettono la comunicazione anche tra processi non relazionati
- in sistemi UNIX l’uso delle PIPE avviene attraverso la nozione di descrittore

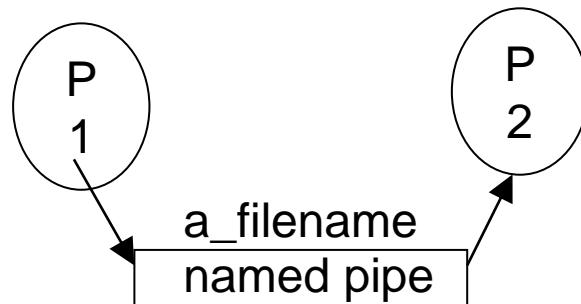
# IPC methods: Pipes



- File-like abstraction for sending data between processes
  - Can be read or written to, just like a file
  - Permissions controlled by the creating process
- Two types of pipes
  - Named pipe: any process can attach as long as it knows the name
    - Typically used for long lived IPC
  - Unnamed/anonymous pipe: only exists between a parent and its children
- Full or half-duplex
  - Can one or both ends of the pipe be read?
  - Can one or both ends of the pipe be written?

# IPC methods: named-pipes (FIFOs)

- A **named-pipe** is called FIFO.
- It has a name
- When processes terminate, it is not removed automatically
- No need for parent-child relationship
- birectional
- Any two process can create and use named pipes.



# PIPE nei Sistemi UNIX

```
int pipe(int fd[2])
```

→ ARRAY FILE DESCRIPTOR

**Descrizione** invoca la creazione di una PIPE

**Argomenti** **fd:** puntatore ad un buffer di due interi  
- **fd[0]** : descrittore di lettura dalla PIPE  
- **fd[1]:** descrittore di scrittura sulla PIPE

**Restituzione** -1 in caso di fallimento, 0 altrimenti *non torna la PIPE*

fd[0] è un canale aperto in lettura che consente ad un processo di leggere dati da una PIPE

fd[1] è un canale aperto in scrittura che consente ad un processo di immettere dati sulla PIPE

fd[0] e fd[1] possono essere usati come normali descrittori di file tramite le chiamate read() e write()

# Lettura e scrittura

bloccante  
↑

- Un processo che provi a leggere da una pipe vuota rimane bloccato finché non ci sono dati disponibili
- Un processo che provi a scrivere su una pipe piena rimane bloccato finché un altro processo non ha letto (e rimosso) abbastanza dati da permettere la scrittura
- Una pipe ha una dimensione massima equivalente a 16 pagine di memoria in linux
  - 65,536 byte in un sistema con una page size di 4096 byte
- E' possibile rendere la lettura e la scrittura non bloccanti (non in questo corso)
- E' possibile cambiare la dimensione della pipe

# Attenzione alle scritture!!!

---

- Una scrittura di  $n \leq \text{PIPE\_BUF}$  byte è atomica
  - $\text{PIPE\_BUF} = 4096$  byte
  - Si blocca se non ci sono  $n$  byte disponibili
- Una scrittura di  $n > \text{PIPE\_BUF}$  byte si potrebbe inframezzare con altre `write()`
  - Il processo rimane bloccato finché non sono stati scritti tutti gli  $n$  byte
- Nel caso di write non bloccanti
  - Se  $n \leq \text{PIPE\_BUF}$  ma non ci sono  $n$  byte disponibili `write()` fallisce e setta `errno`
  - Se  $n > \text{PIPE\_BUF}$  potrebbe risultare in una scrittura parziale (ma non in un errore)

---

# Avvertenze

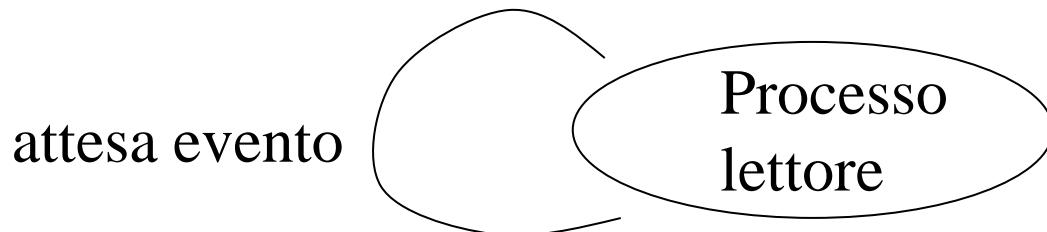
- le PIPE non sono dispositivi fisici, ma logici, pertanto viene spontaneo chiedersi come un processo sia in grado di vedere la fine di un “file” su una PIPE
- per convenzione, ciò avviene quando tutti i processi scrittori che condividevano il descrittore fd[1] lo hanno chiuso
- in questo caso la chiamata read() effettuata da un lettore restituisce zero come notifica dell'evento che tutti gli scrittori hanno terminato il loro lavoro
- allo stesso modo, un processo scrittore che tenti di scrivere sul descrittore fd[1] quando tutte le copie del descrittore fd[0] siano state chiuse (non ci sono lettori sulla PIPE), riceve il segnale SIGPIPE, altrimenti detto «Broken pipe»

# PIPE e deadlock

Per fare in modo che tutto funzioni correttamente e non si verifichino situazioni di deadlock, è necessario che tutti i processi chiudano i descrittori di PIPE che non gli servono, usando una normale close()

Si noti che ogni processo lettore che erediti la coppia (fd[0],fd[1]) deve chiudere la propria copia di fd[1] prima di mettersi a leggere da fd[0] dichiarando così di non essere uno scrittore

Se così non facesse, l'evento "tutti gli scrittori hanno terminato" non potrebbe mai avvenire se il lettore è impegnato a leggere, e si potrebbe avere un deadlock



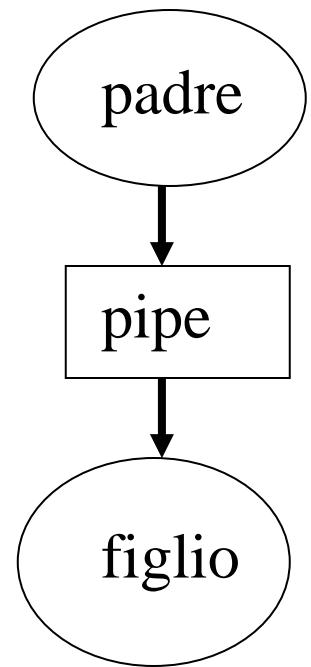
imp.

# Un esempio: trasferimento stringhe tramite PIPE

```
#include <stdio.h>
#define Errore_(x) { puts(x); exit(1); }

int main(int argc, char *argv[]) {
    char messaggio[30];    int pid, status, fd[2];
    ret = pipe(fd); /* crea una PIPE */
    if ( ret == -1 )
        Errore_("Errore nella creazione pipe");
    pid = fork(); /* crea un processo figlio */
    if ( pid == -1 ) Errore_("Errore nella fork");

    if ( pid == 0 ) { /* processo figlio: lettore */
        close(fd[1]); /* il lettore chiude fd[1] */
        while( read(fd[0], messaggio, 30) > 0 ) (QUANTI bit ho letto)
            printf("letto messaggio: %s", messaggio);
        close(fd[0]);
        _exit();
    }
}
```



# .....continua

```
/* processo padre: scrittore */
else {

    close(fd[0]);
    puts("digitare testo da trasferire (quit per
terminare):");

    do {
        fgets(messaggio,30,stdin);
        write(fd[1], messaggio, 30);
        printf("scritto messaggio: %s", messaggio);
    } while( strcmp(messaggio,"quit\n") != 0 );

    close(fd[1]);
    wait(&status);
}

}
```

# Named PIPE (FIFO) in Sistemi UNIX

---

```
int mkfifo(char *name, int mode)
```

---

**Descrizione** invoca la creazione di una FIFO

---

**Argomenti** 1) \*name: nome della FIFO da creare

2) mode: specifica i permessi di accesso alla FIFO

---

**Restituzione** -1 in caso di fallimento, 0 altrimenti

La rimozione di una FIFO dal file system avviene mediate la chiamata di sistema unlink()

# Avvertenze

---

- normalmente, l'apertura di una FIFO è bloccante, nel senso che il processo che tenti di aprirla in lettura (scrittura) viene bloccato fino a quando un altro processo non la apre in scrittura (lettura)
- se si vuole inibire questo comportamento è possibile aggiungere il flag `O_NONBLOCK` al valore del parametro `mode` passato alla system call `open()` su di una FIFO
- ogni FIFO deve avere sia un lettore che uno scrittore: se un processo tenta di scrivere su una FIFO che non ha un lettore esso riceve il “segnale `SIGPIPE`” da parte del sistema operativo

# Un esempio: client/server tramite FIFO (UNIDIREZIONALE)

imp:

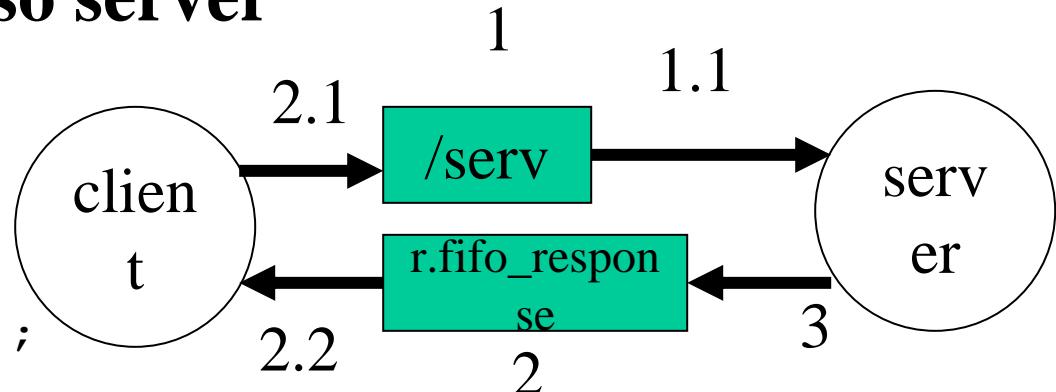
## Processo server

```
#include <stdio.h>
#include <fcntl.h>

typedef struct {
    long type;
    char fifo_response[20];
} request;
```

```
int main(int argc, char *argv[]) {
    char *response = "fatto";
    int pid, fd, fdc, ret;
    request r;

    ret = mkfifo("/serv", 0666);
    if (ret == -1) {
        printf("Errore nella chiamata mkfifo\n");
        exit(1);
}
```



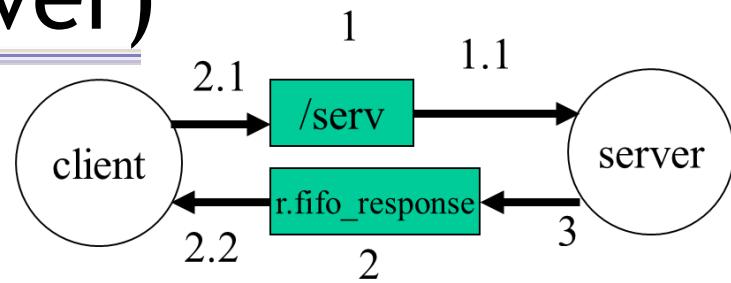
1

# ....continua (Processo Server)

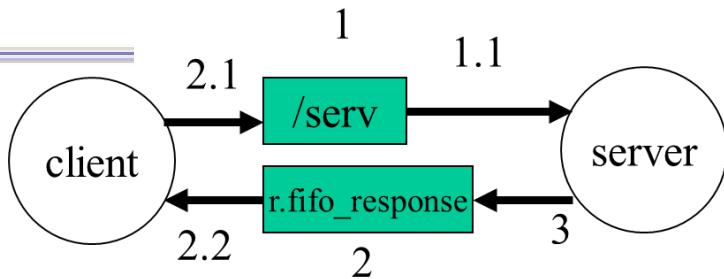
→ APERTURA FIFO

```
fd = open("/serv", O_RDONLY);
```

```
while(1) {
    ret = read(fd, &r, sizeof(request));
    if (ret != 0) { -1 & more
        printf("Richiesto un servizio (fifo di restituzione = %s)\n",
               r fifo response);
        /* switch sul tipo di servizio */
        sleep(10); /* emulazione di ritardo per il servizio */
        fdc = open(r fifo response, O_WRONLY);
        write(fdc, response, 20);
        close(fdc);
        exit(0);
    } /* end if (ret != 0) */
}
```



# Processo client



```
#include <stdio.h>
#include <fcntl.h>

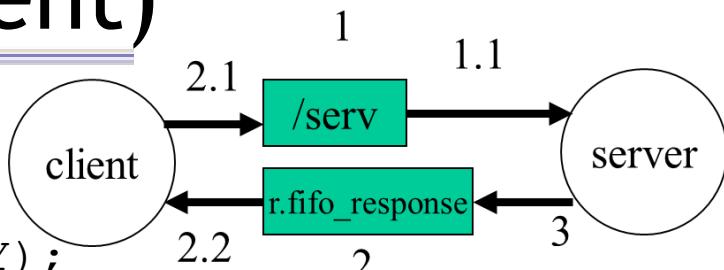
typedef struct { long type; char fifo_response[20]; } request;

int main(int argc, char *argv[]) {
    int pid, fd, fdc, ret;    request r;    char response[20];

    printf("Selezionare un carattere alfabetico minuscolo: ");
    scanf("%s",r.fifo_response);
    if (r.fifo_response[0] > 'z' || r.fifo_response[0] < 'a' ) {
        printf("carattere selezionato non valido, ricominciare operazione\n");
        exit(1);
    }
    r.fifo_response[1] = '\0';
    ret = mkfifo(r.fifo_response, 0666);
    if ( ret == -1 ) {
        printf("\n servente sovraccarico - riprovare \n");
        exit(1);
    }
```

2

# .....continua (Processo Client)



```
fd = open("/serv", O_WRONLY);
if ( fd == -1 ) {
    printf("\n servizio non disponibile \n");
    ret = unlink(r fifo_response);
    exit(1);
}
```

```
write(fd, &r, sizeof(request));
close(fd);
```

2.1

2.2

```
fdc = open(r fifo_response, O_RDONLY);
read(fdc, response, 20);
printf("risposta = %s\n", response);

close(fdc);
unlink(r fifo_response);
}
```

# Confronto tra pipe e FIFO

---

- L'apertura di una pipe comporta ottenere sia il descrittore per la lettura che quello per la scrittura
  - Bisogna poi chiudere quello che non viene usato
- L'apertura di una FIFO avviene in lettura o in scrittura
- Nella gestione si differenziano solo nella fase di apertura e chiusura
  - Lettura e scrittura avvengono nello stesso modo

# Le reti dei calcolatori

Slides are mainly taken from

- *W.R. Stevens “Unix Network Programming” Prentice Hall, 1999*
- *Peterson – Davie “Computer Networks: A system approach” Morgan Kaufmann 2000*
- *Andrew Tanenbaum and David Wetherall, “Computer Networks”*
- *William Stallings “Operating Systems: Internals and Design Principles”, 8/E (Chapter 5).*

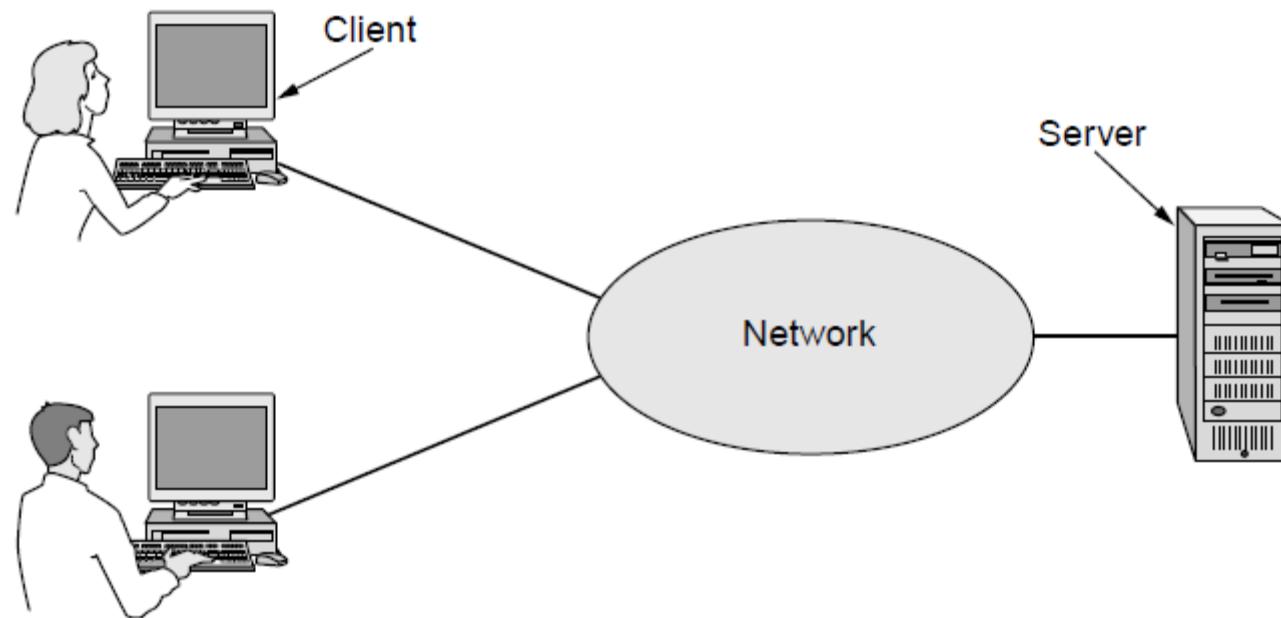
*Sistemi di Calcolo 2*

*Instructor: Riccardo Lazzeretti*

*Special thanks to: Daniele Cono D’Elia, Leonardo Aniello, Roberto Baldoni*

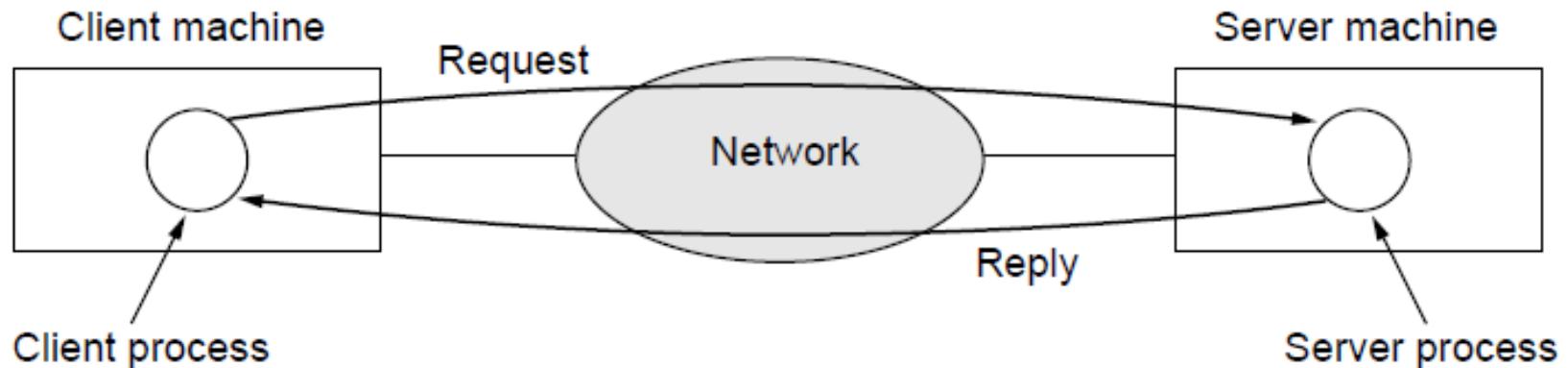
# Architettura di Internet

# Business Applications (1)



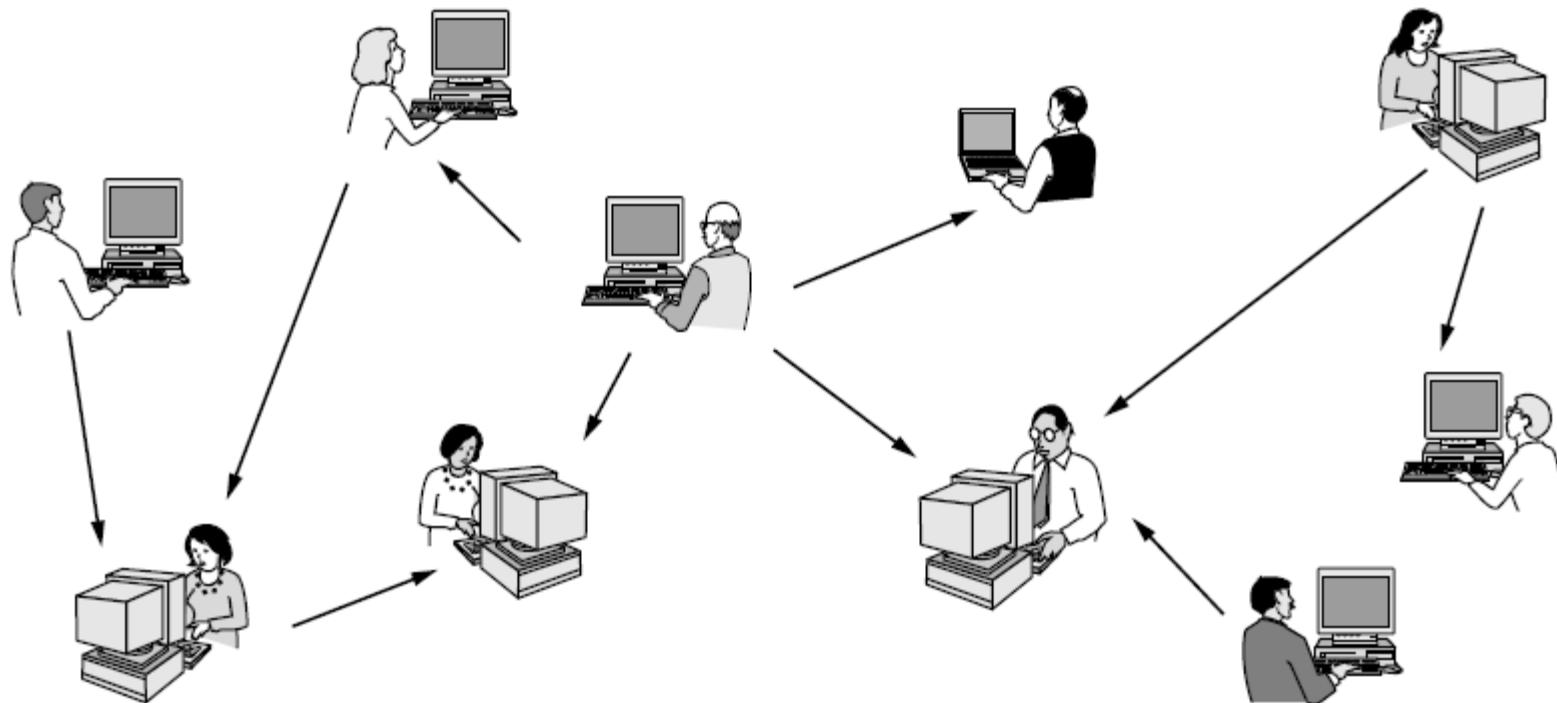
# Business Applications (2)

The client-server model involves requests and replies



# Home Applications

Ogni dispositivo  
è sia client che server

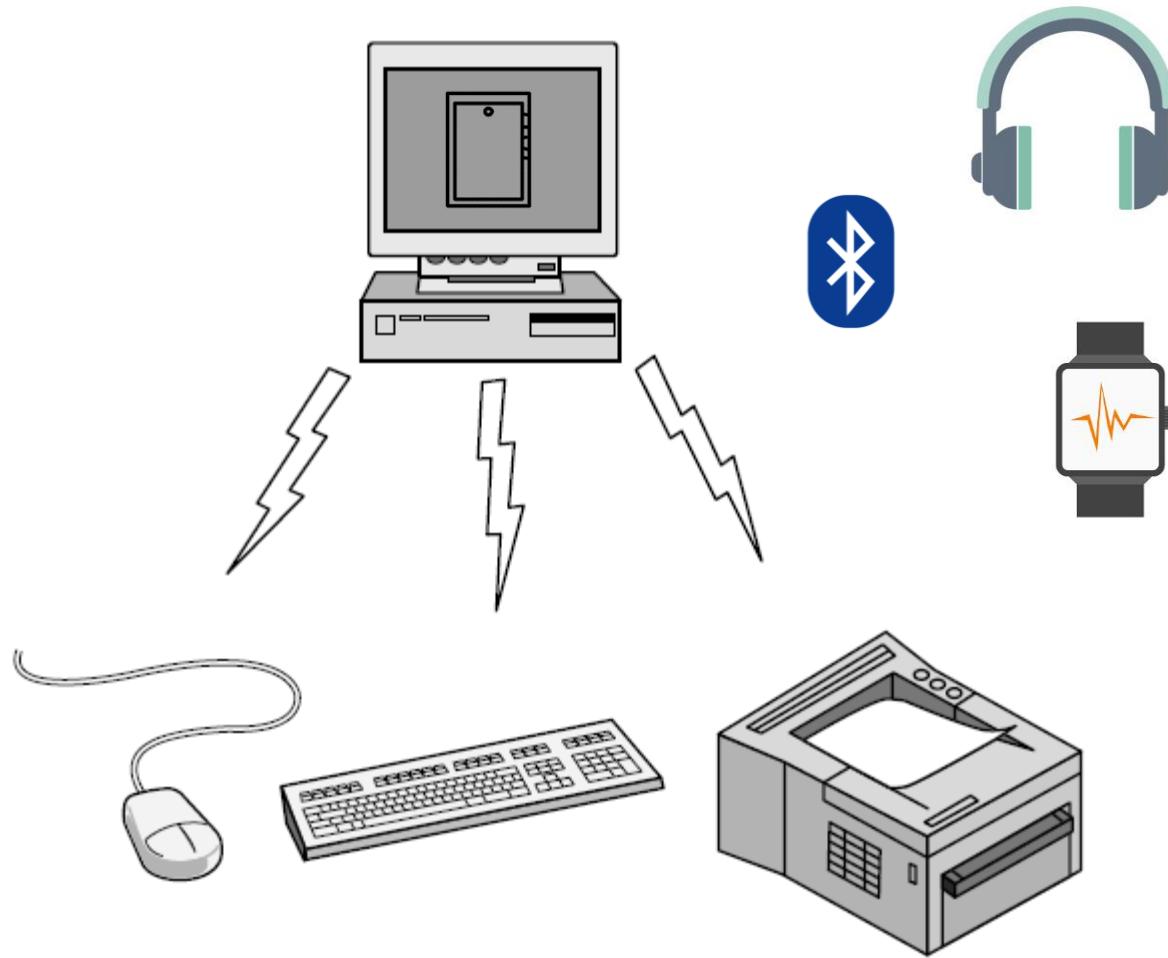


In a peer-to-peer system there are no fixed clients and servers.

# Network classification by scale

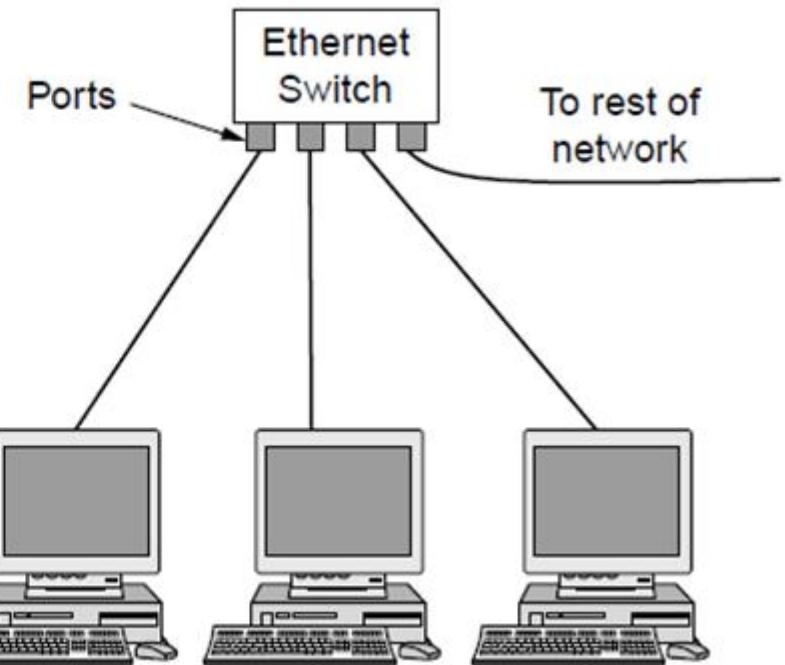
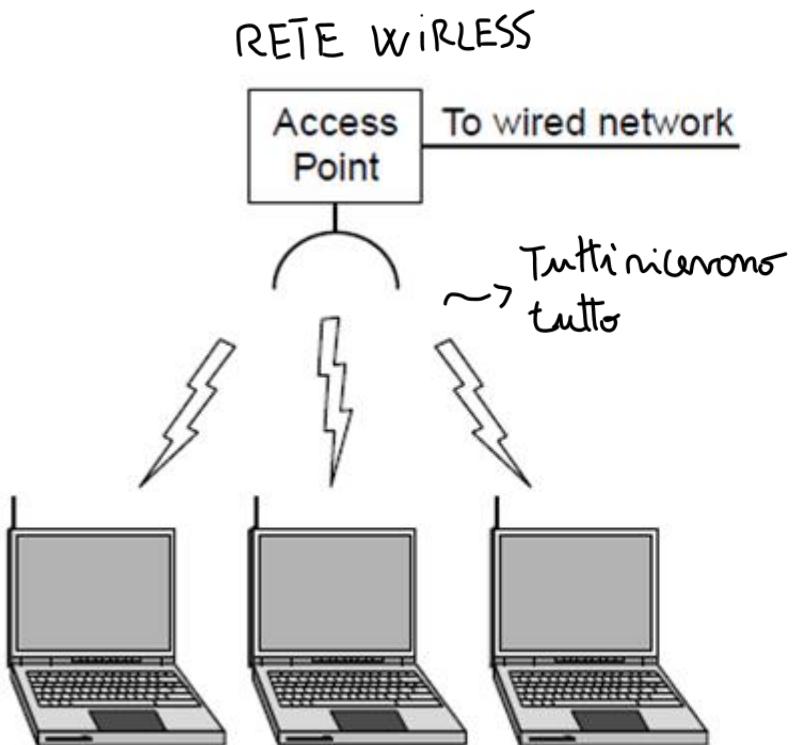
| Interprocessor distance<br><small>(Pochi metri)</small> | Processors located in same | Example                   |
|---|----------------------------|---------------------------|
| 1 m   | Square meter               | Personal area network     |
| 10 m  | Room                       | Local area network        |
| 100 m   | Building                   |                           |
| 1 km  | Campus                     |                           |
| 10 km   | City                       | Metropolitan area network |
| 100 km  | Country                    | Wide area network         |
| 1000 km   | Continent                  |                           |
| 10,000 km   | Planet                     | The Internet              |

# Personal Area Network



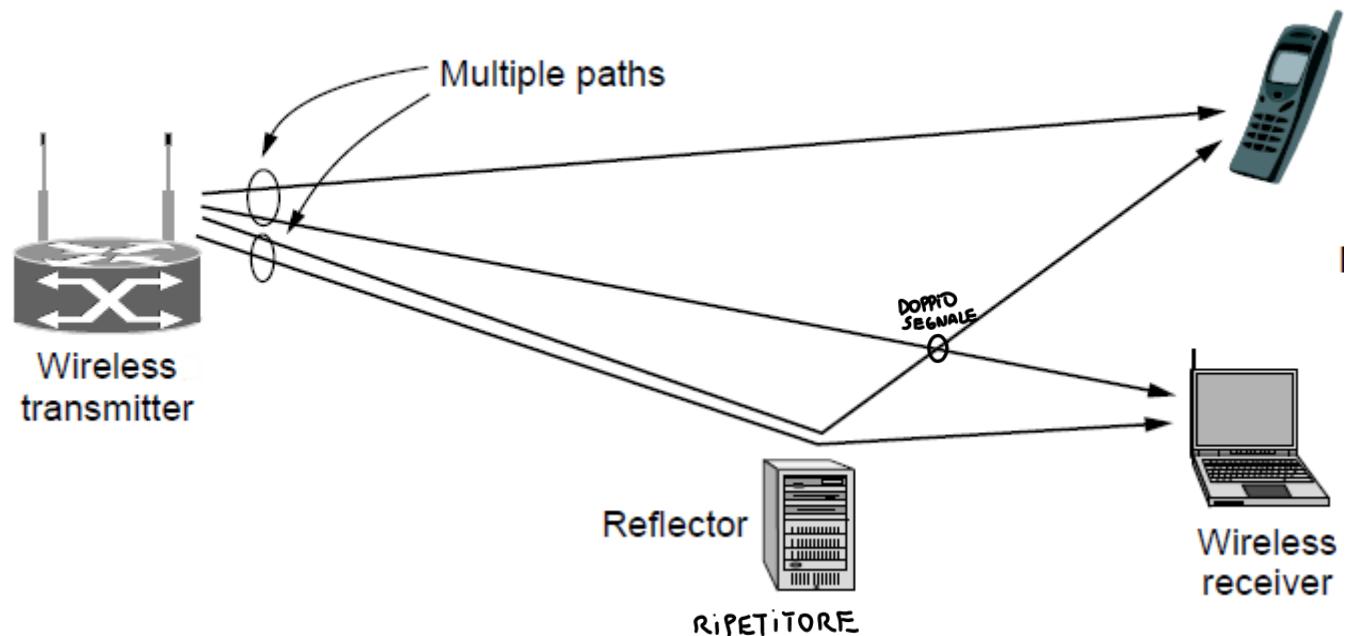
# Local Area Networks

Wireless and wired LANs. (a) 802.11. (b) Switched Ethernet.

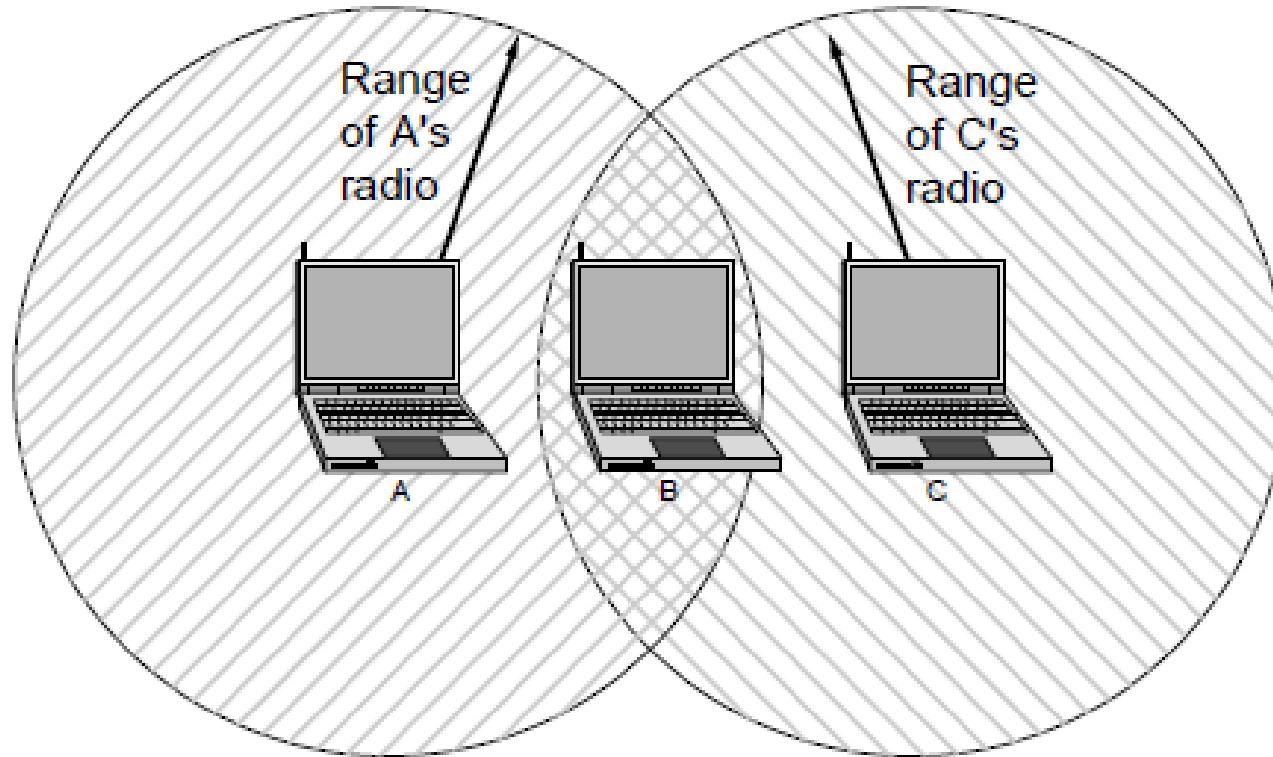


# Wireless LANs: 802.11 (2)

## Multipath fading

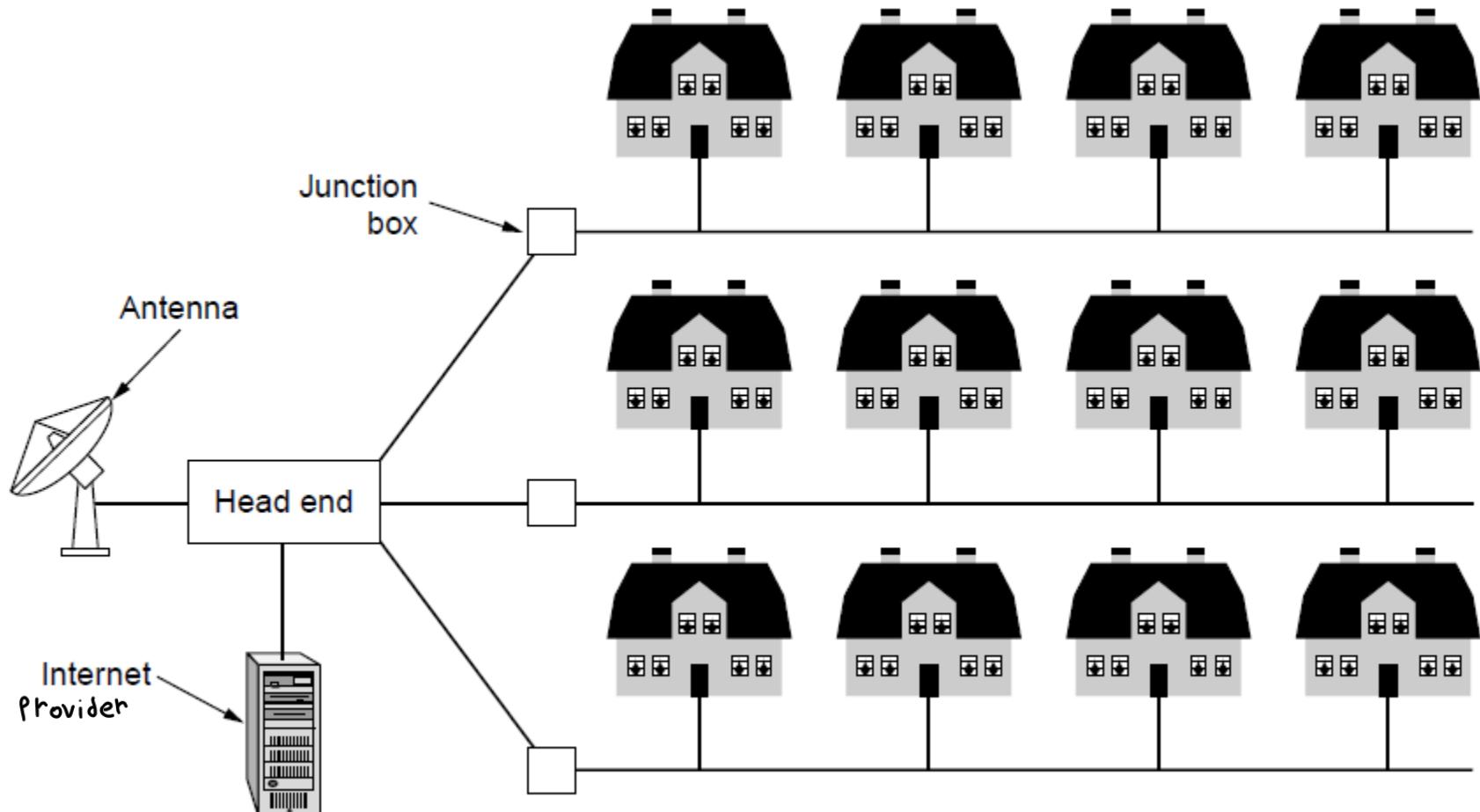


# Wireless LANs: 802.11 (3)



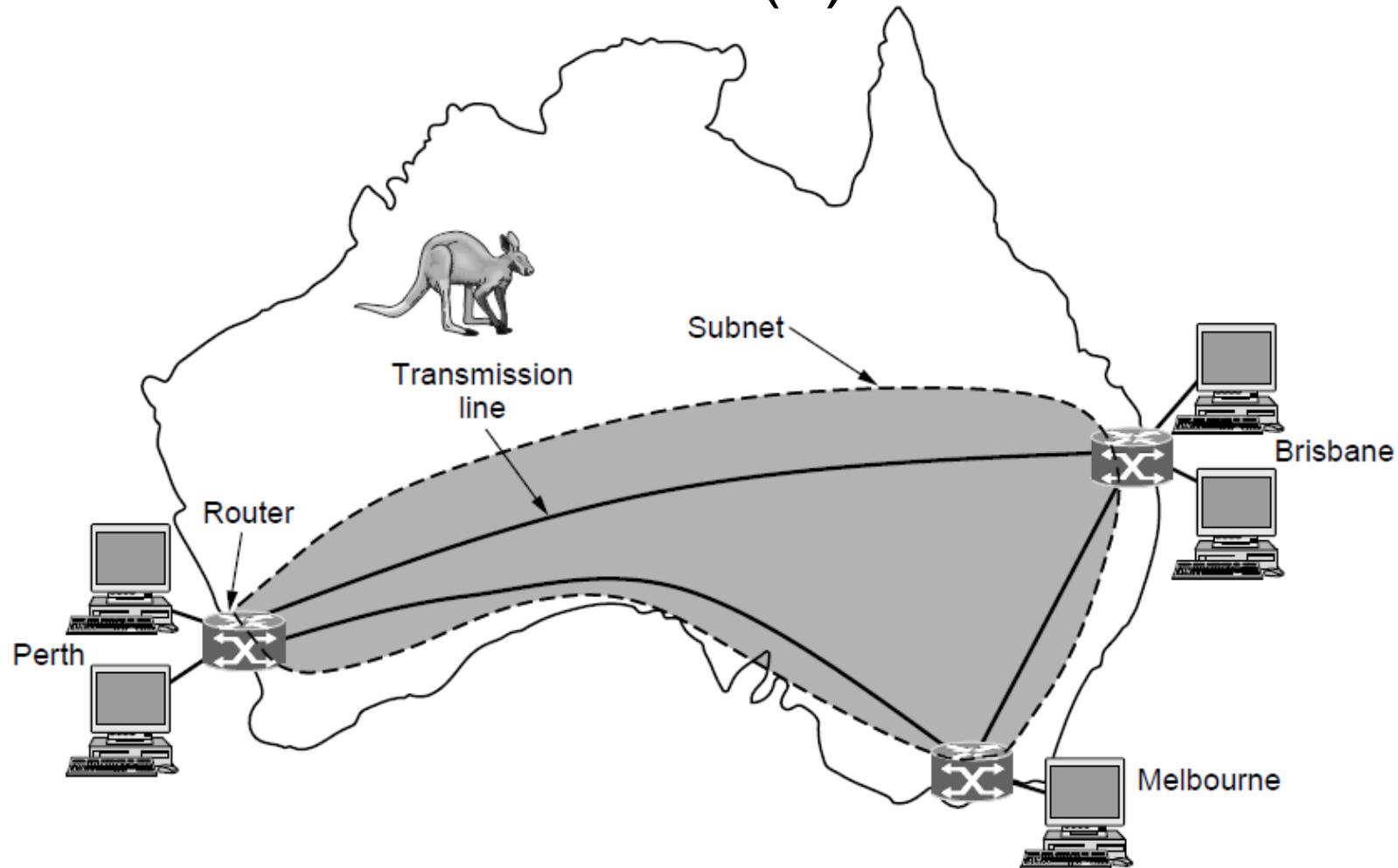
The range of a single radio may not cover the entire system.

# Metropolitan Area Networks



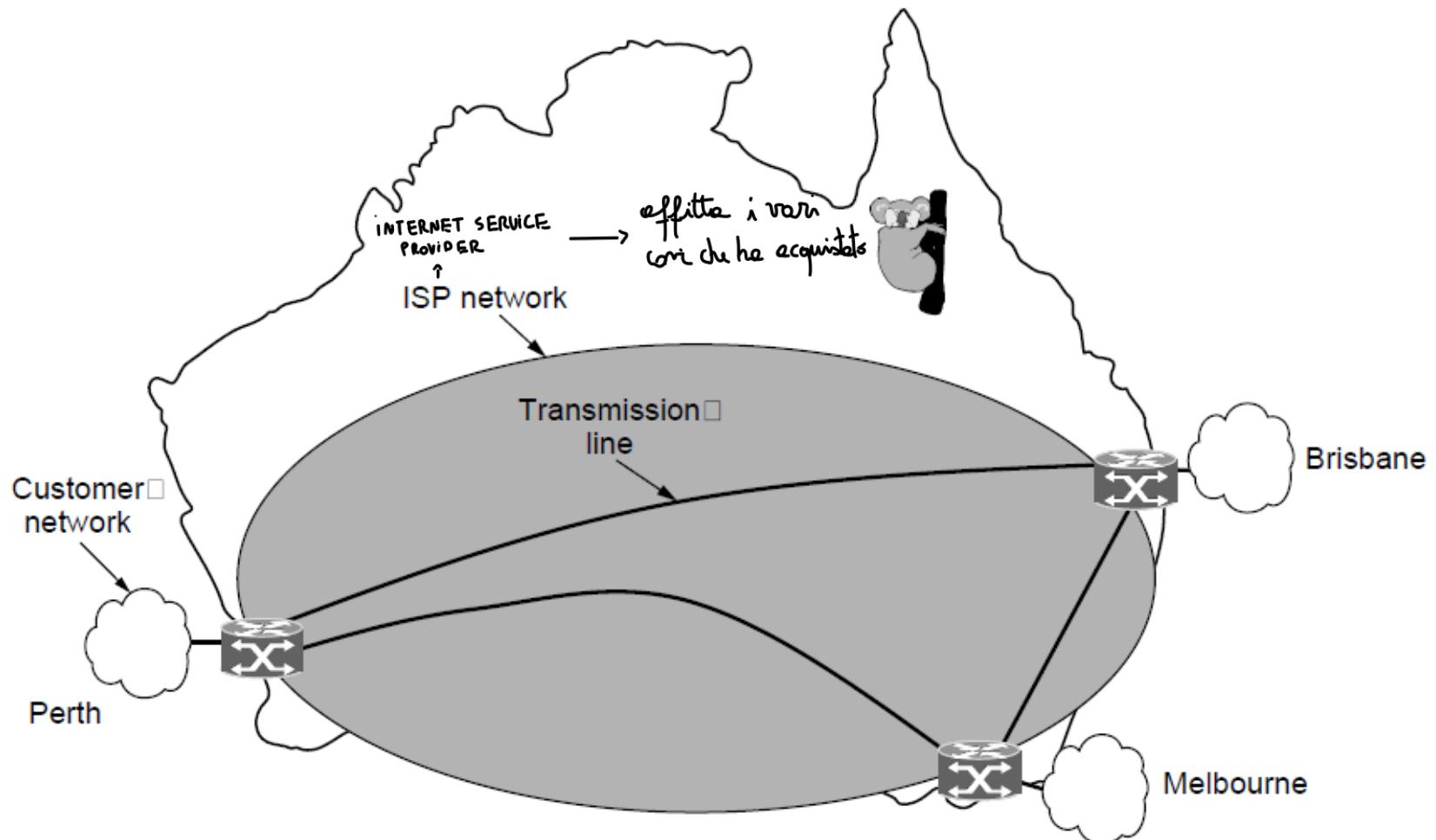
A metropolitan area network based on cable TV.

# Wide Area Networks (1)



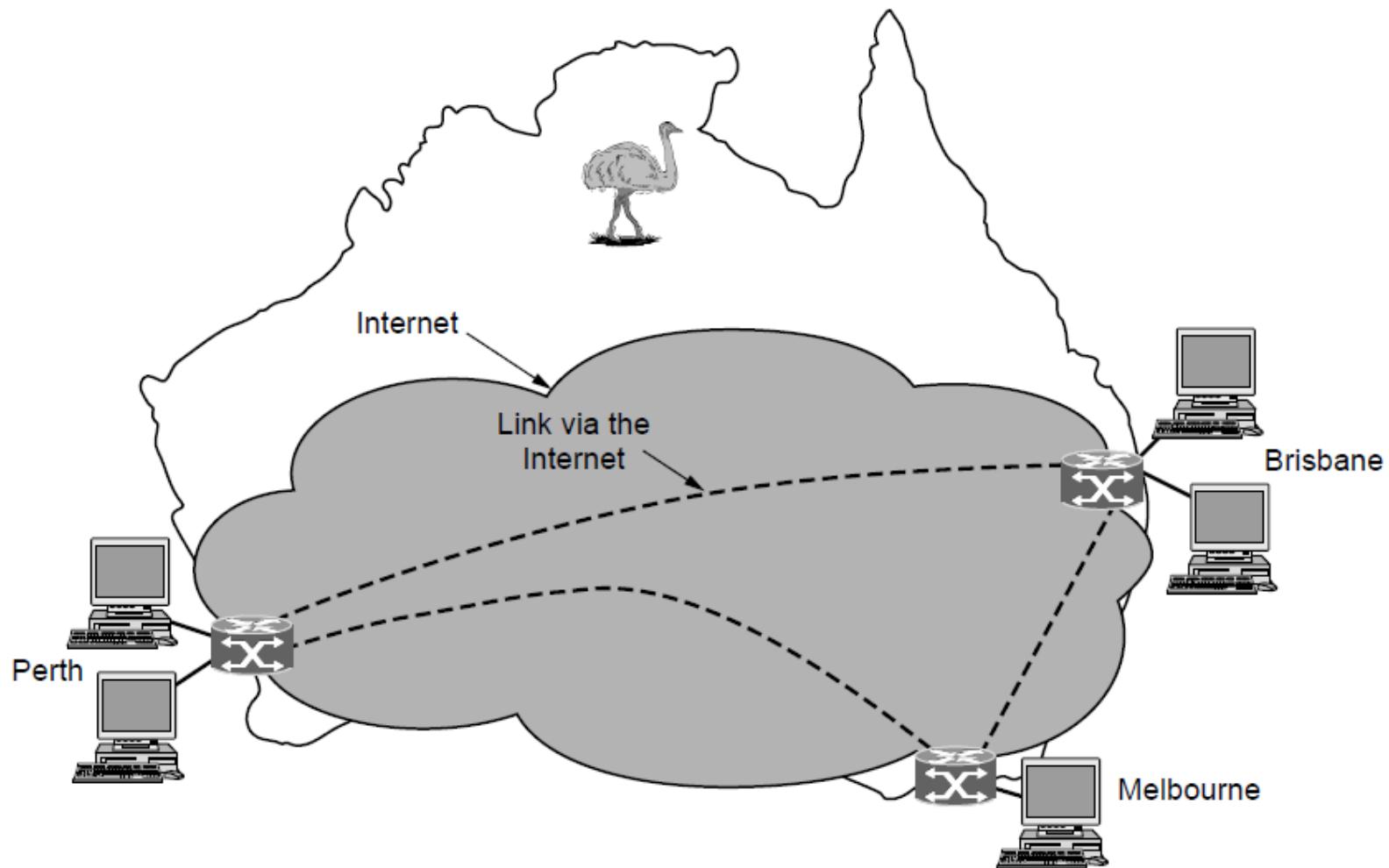
WAN that connects three branch offices in Australia

# Wide Area Networks (2)



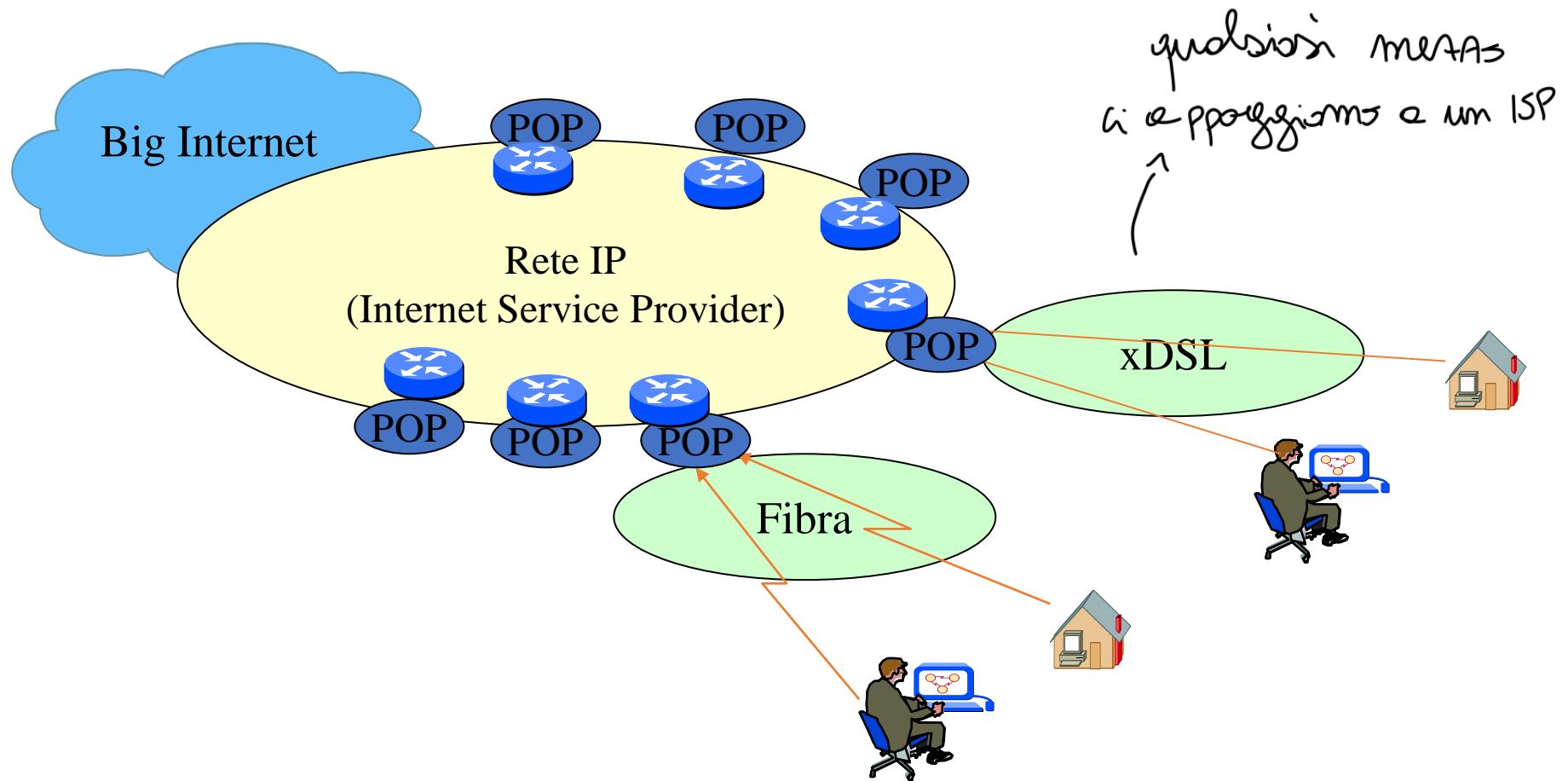
WAN using an ISP network.

# Wide Area Networks (3)

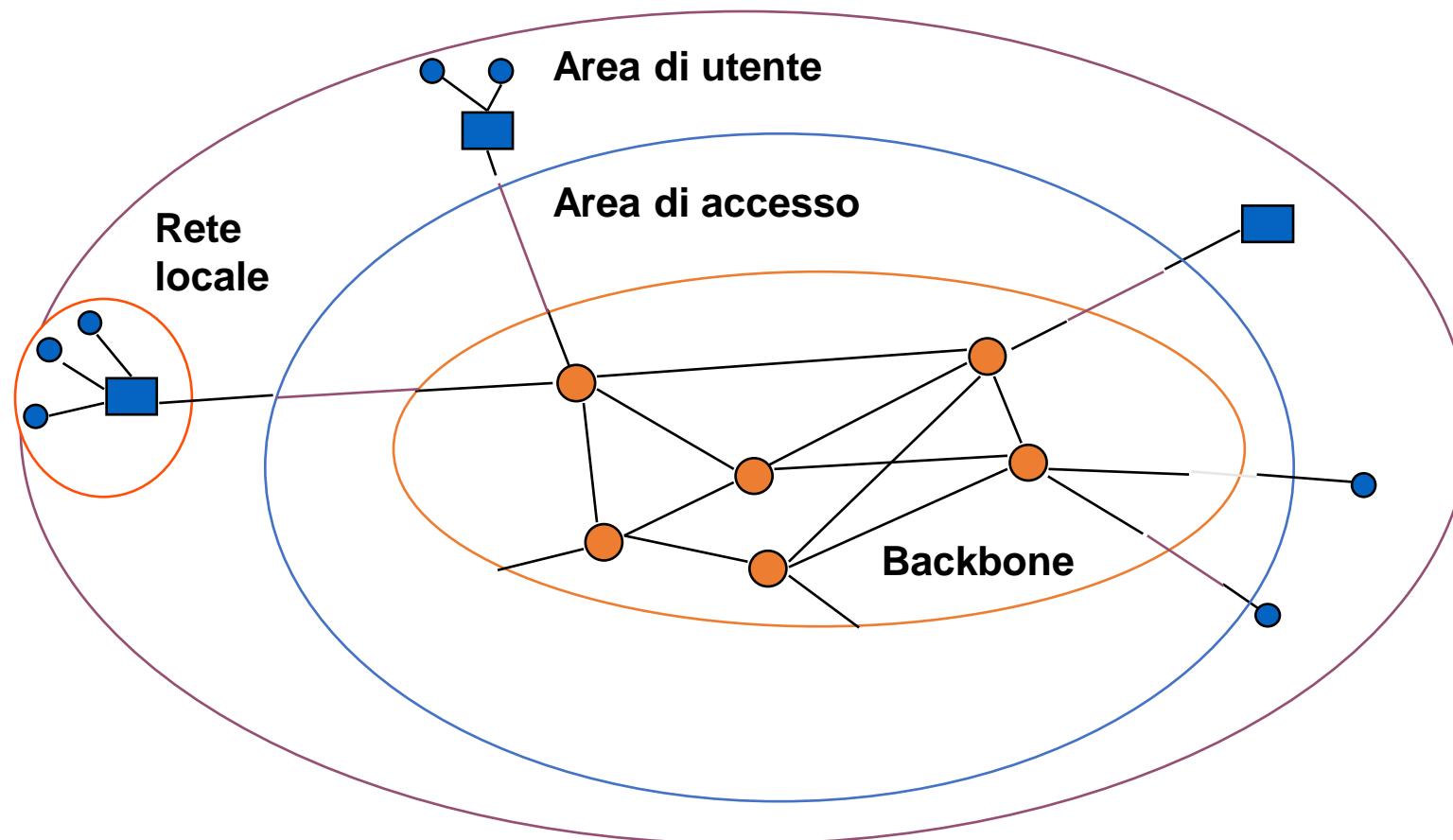


WAN using a virtual private network. (VPN)

# Internet Service Providers



# Rete geografica per trasmissione dati



- = terminale di utente
- = unità di accesso
- = nodo del sottosistema di comunicazione

# Global IP Backbone



PCCW Global PoP (IP)

PCCW Global IP Backbone

PCCW Global Extended-Net Coverage (IP)



# GARR NETWORK

(università europee  
tra di loro su una  
backbone)

- GARR Network PoPs

## OPTICAL FIBRE

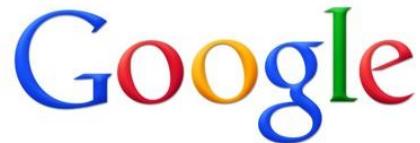
- operational
- planned

## PEERING

- research links

© GARR, MARCH 2017

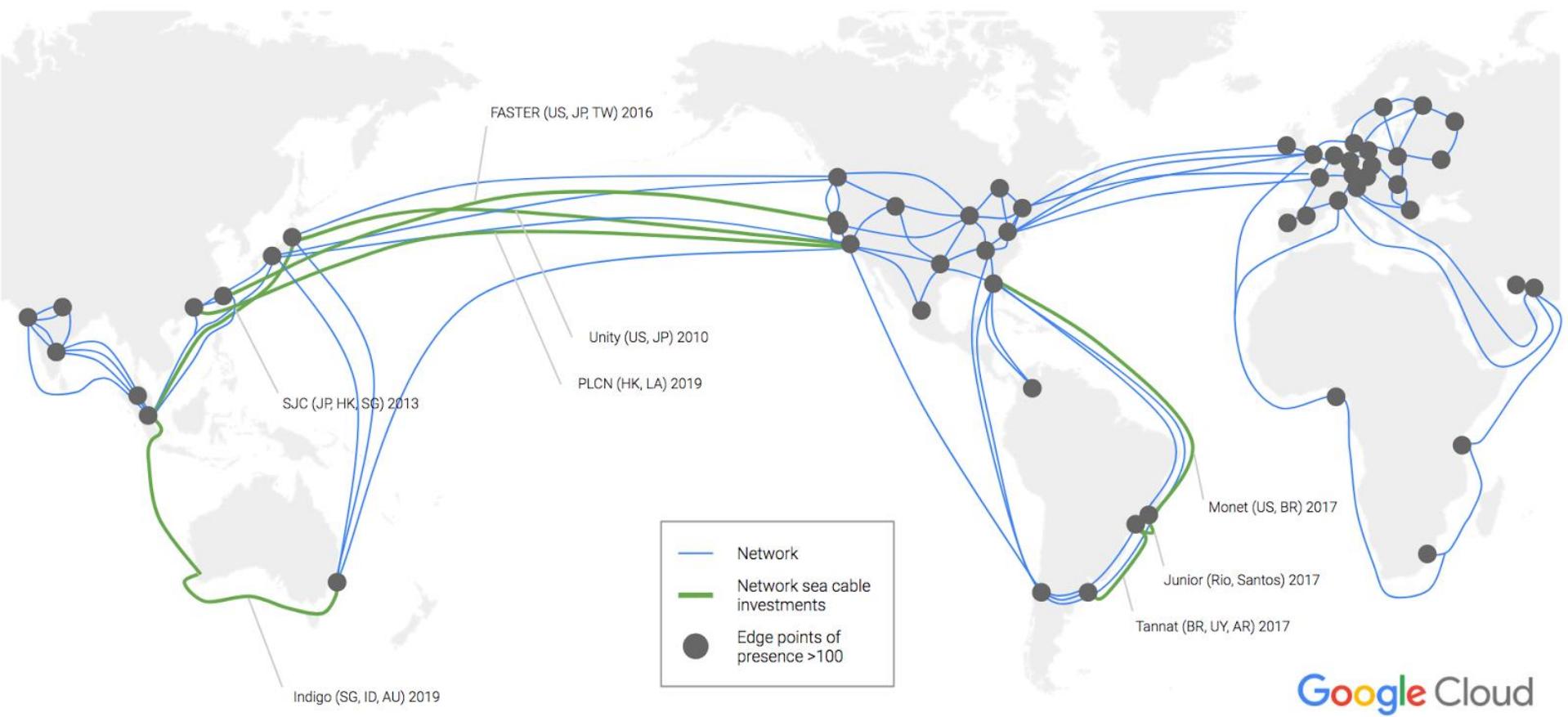




The company has essentially two huge networks: the one that connects users to Google services (Search, Gmail, YouTube, etc.) and another (internal) that connects Google data centers to each other.

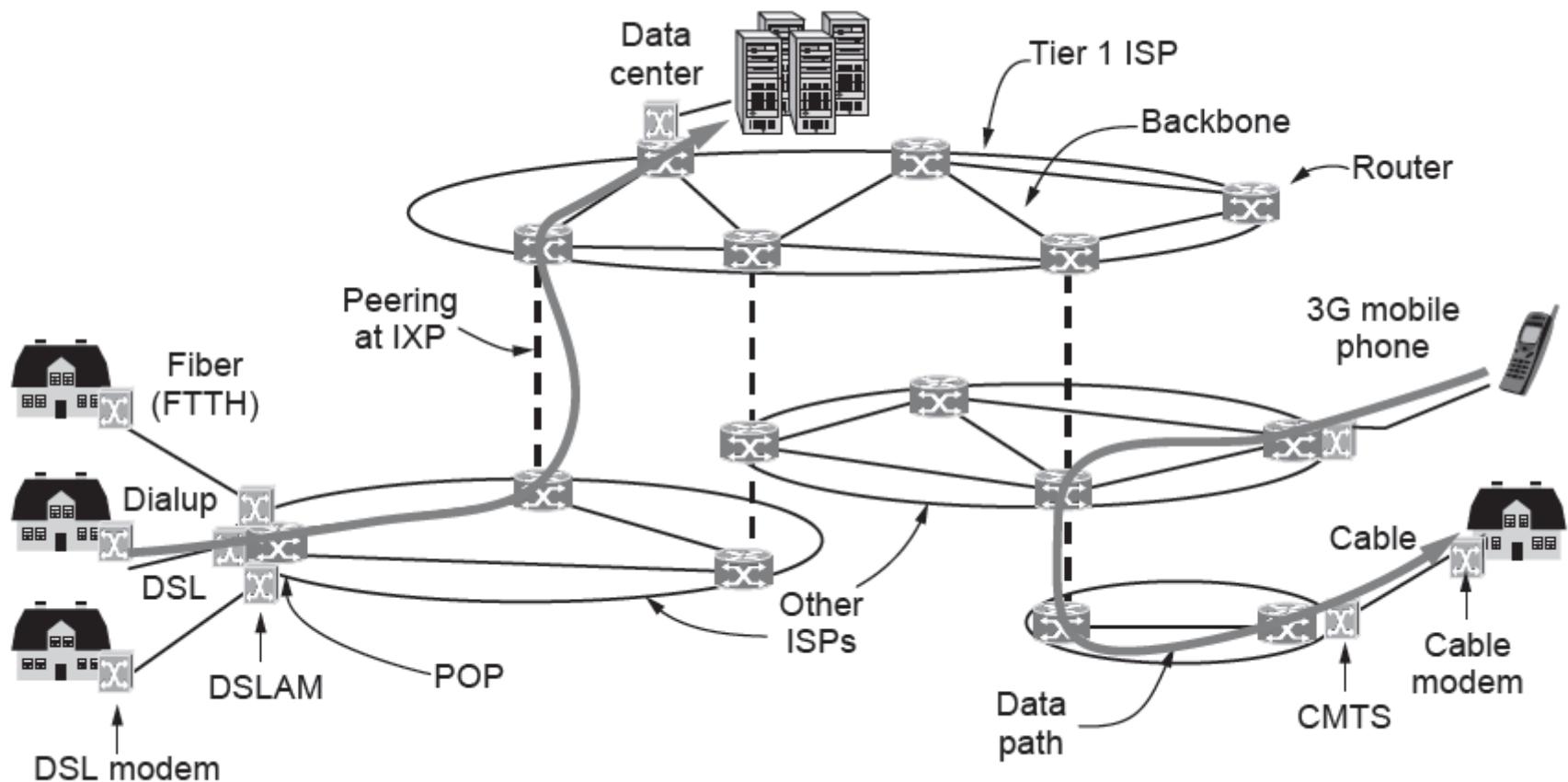
Google is in control of scheduling internal traffic (bursty), but it faces difficulties in traffic engineering.

Often Google has to move many petabytes of data (indexes of the entire web, millions of backup copies of user Gmail) from one place to another.



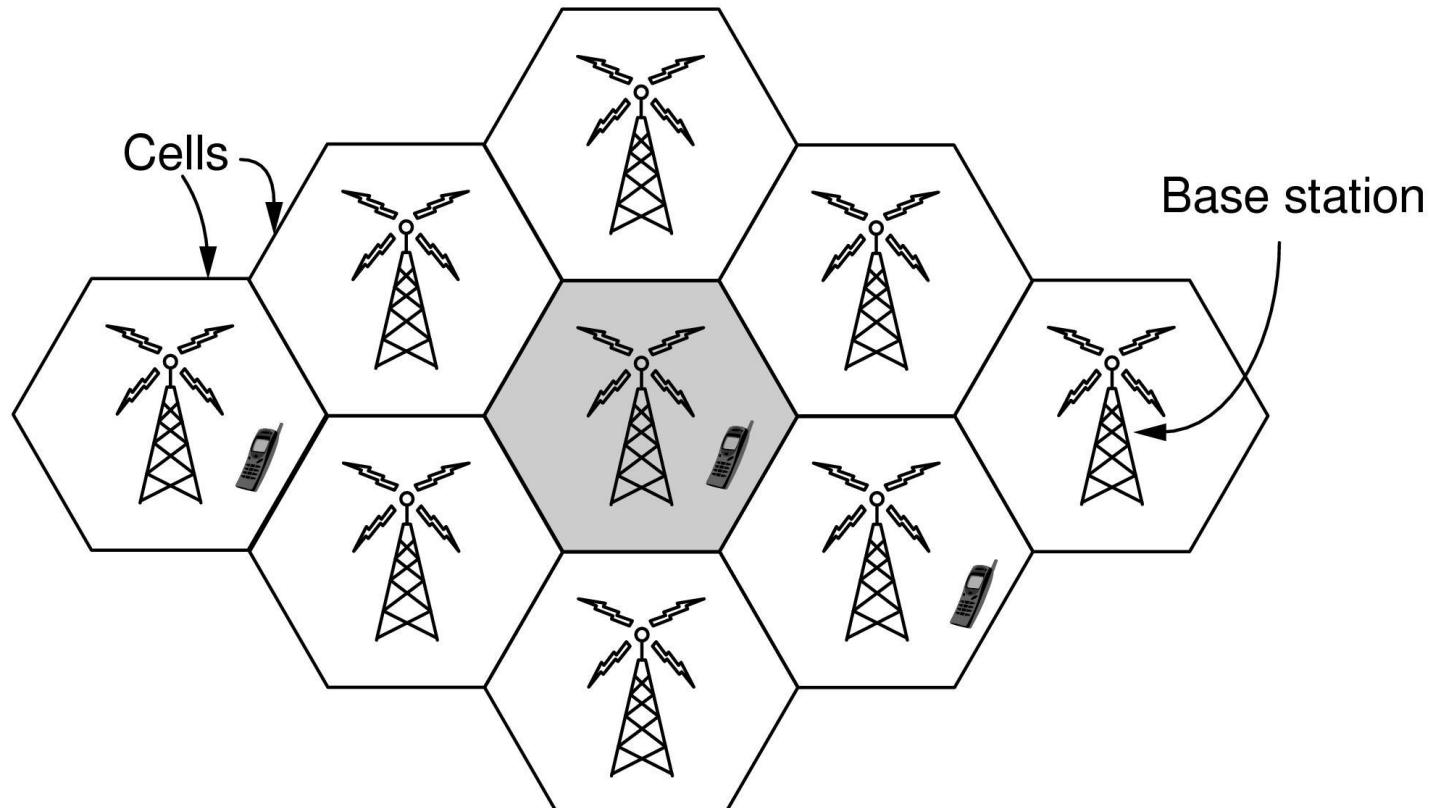
Google Cloud

# Architecture of the Internet



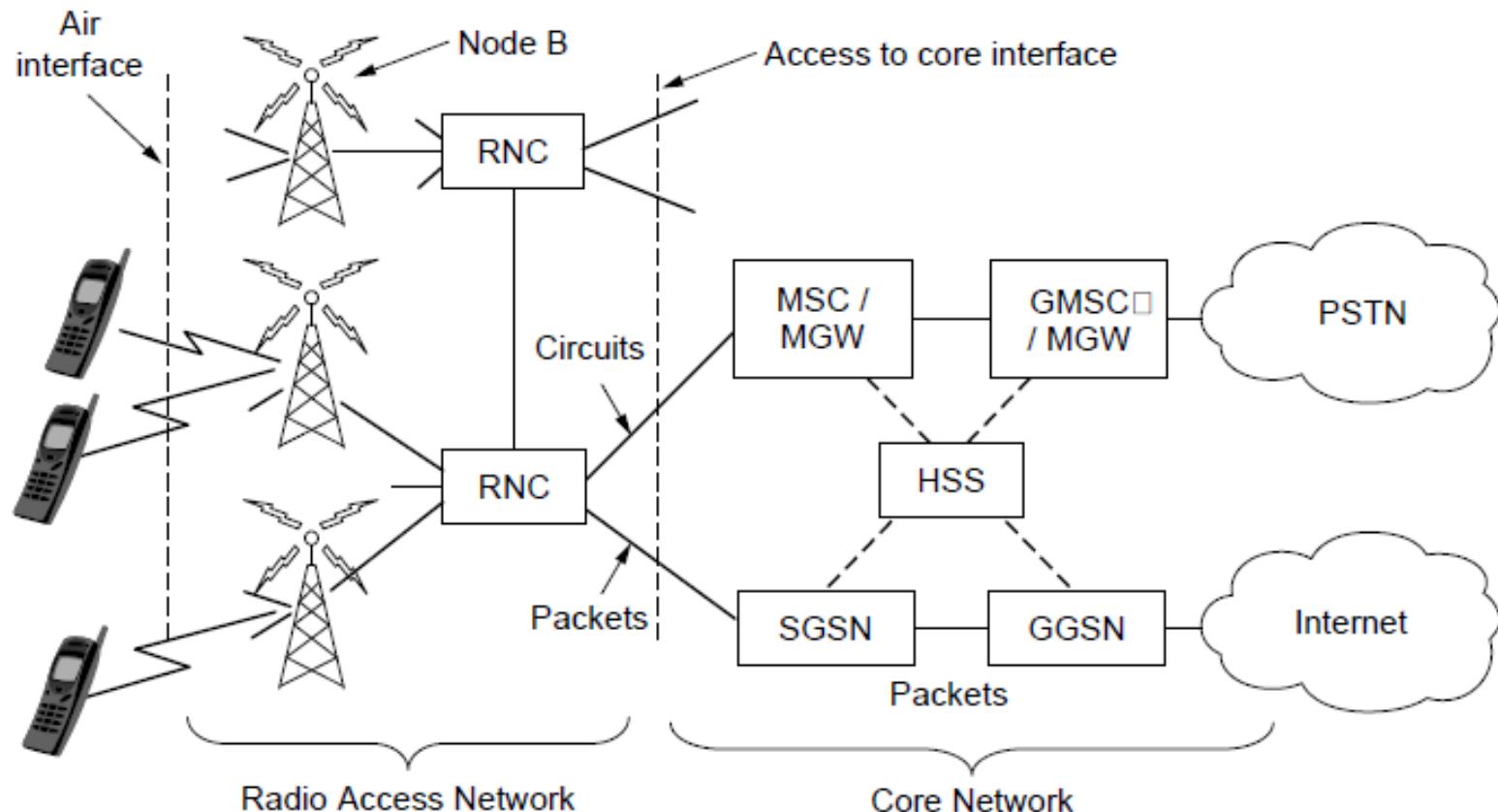
Overview of the Internet architecture

# Third-Generation Mobile Phone Networks (1)



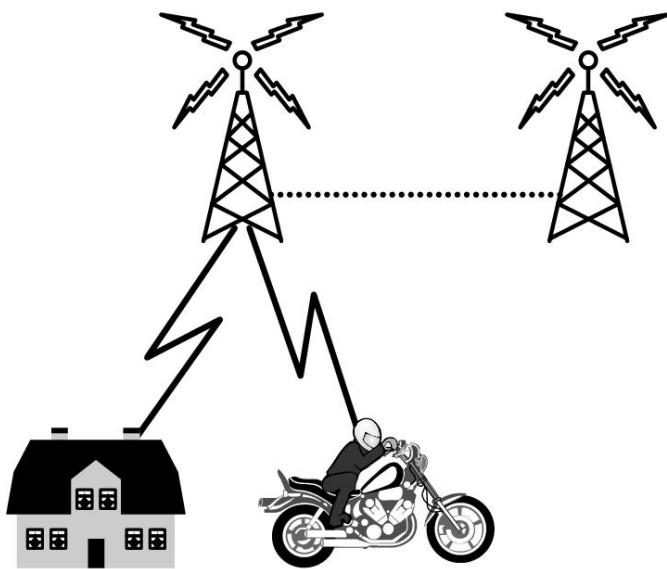
Cellular design of mobile phone networks

# Third-Generation Mobile Phone Networks (2)

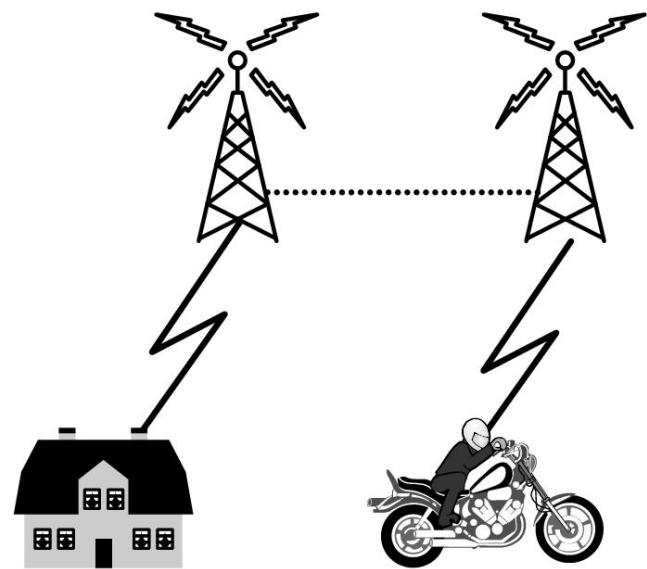


Architecture of the UMTS 3G mobile phone network.

# Third-Generation Mobile Phone Networks (3)



(a)

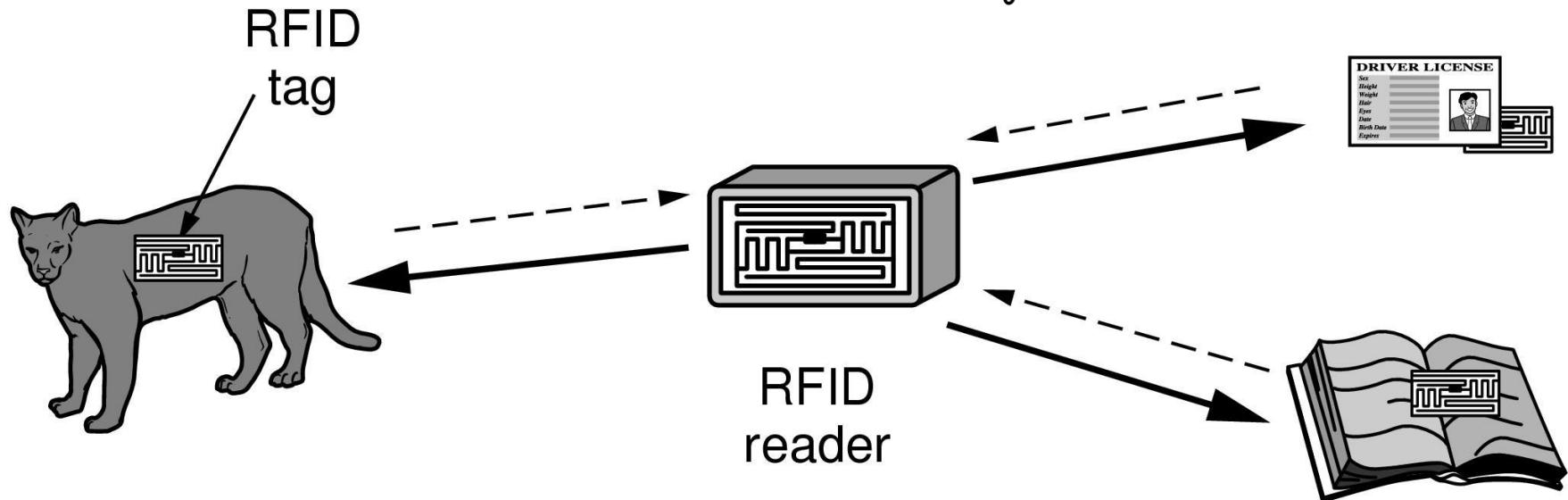


(b)

Mobile phone handover (a) before, (b) after.

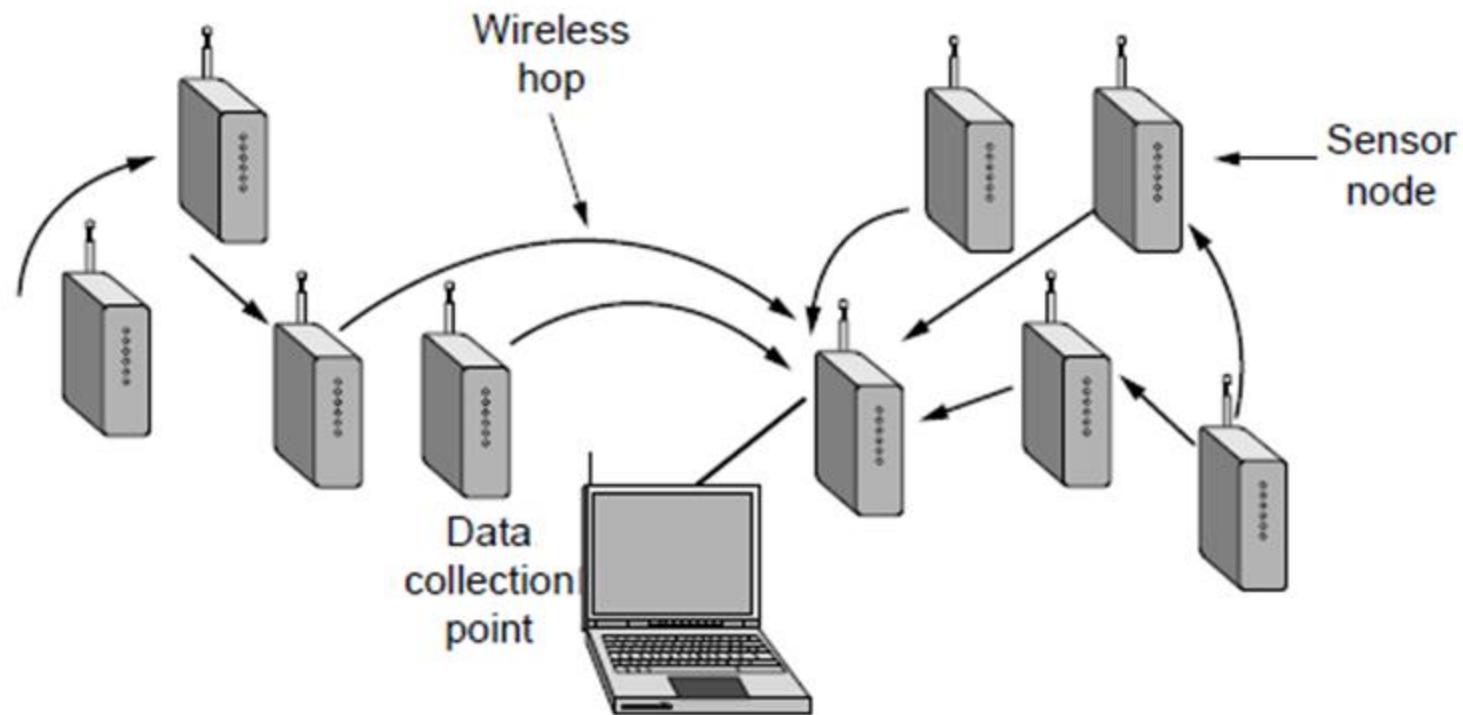
# RFID and Sensor Networks (1)

circuiti non alimentati che si  
attivano con un campo magnetico



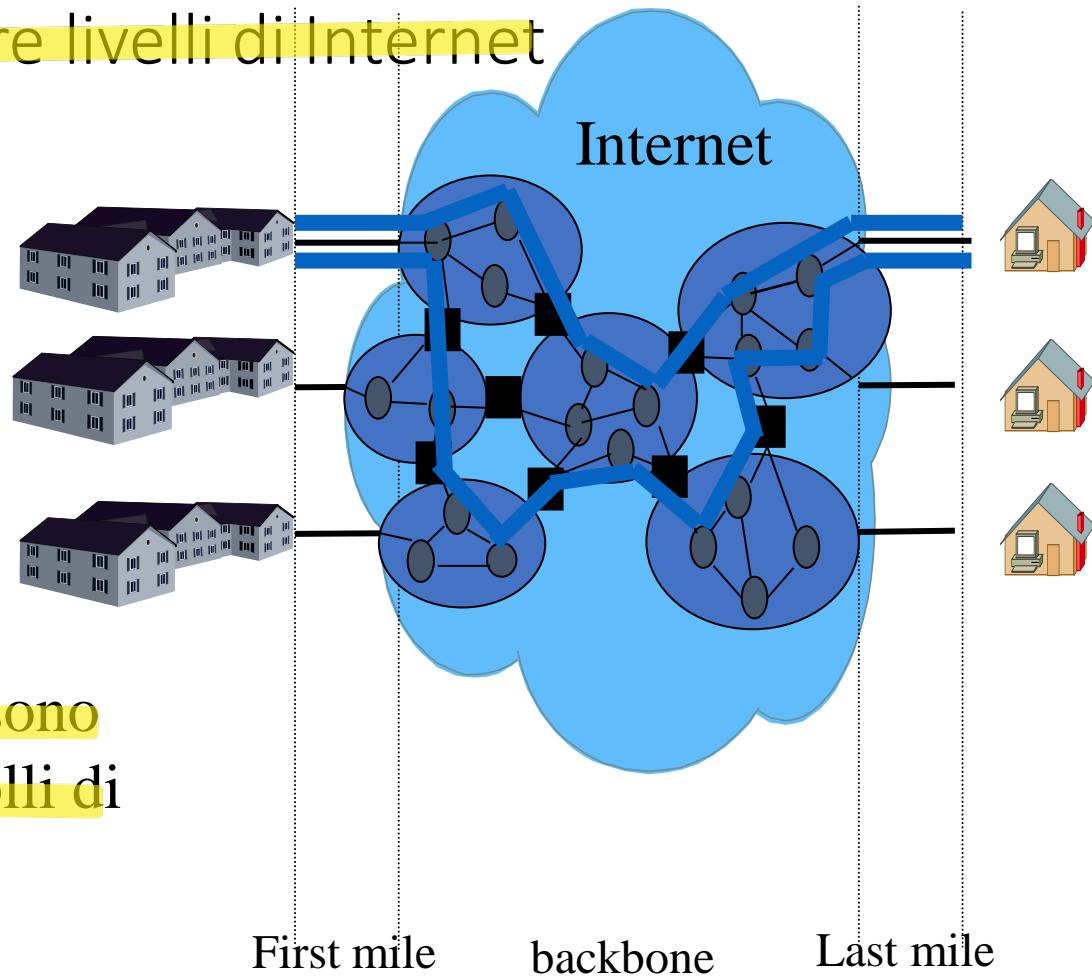
RFID used to network everyday objects.

# RFID and Sensor Networks (2)



Multihop topology of a sensor network

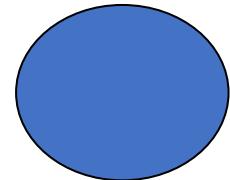
## Architettura a tre livelli di Internet



Le tre zone sono  
potenziali colli di  
bottiglia



Neutral/Network access point



ISP o NSP

# Architettura a tre livelli di Internet

Eliminare colli di bottiglia (soluzioni hardware)

- first mile, last mile -> aumentare la banda che connette al provider
- Backbone -> dipende dal miglioramento delle infrastrutture di rete dei singoli ISP (non controlabile dagli utenti finali)

Eliminare il collo di bottiglia di backbone (soluzione software)

- Content Delivery Networks.
- Caching di pagine vicino a dove risiede l'utente completamente trasparente all'utente (e.g. AKAMAI). In questo modo si spera che l'utente possa accedervi con larga banda

Nota: idea di soluzione simile a quella della gerarchia di caching delle memorie nei processori

## Akamai's Global Platform

### ■ Akamai's Internet Platform

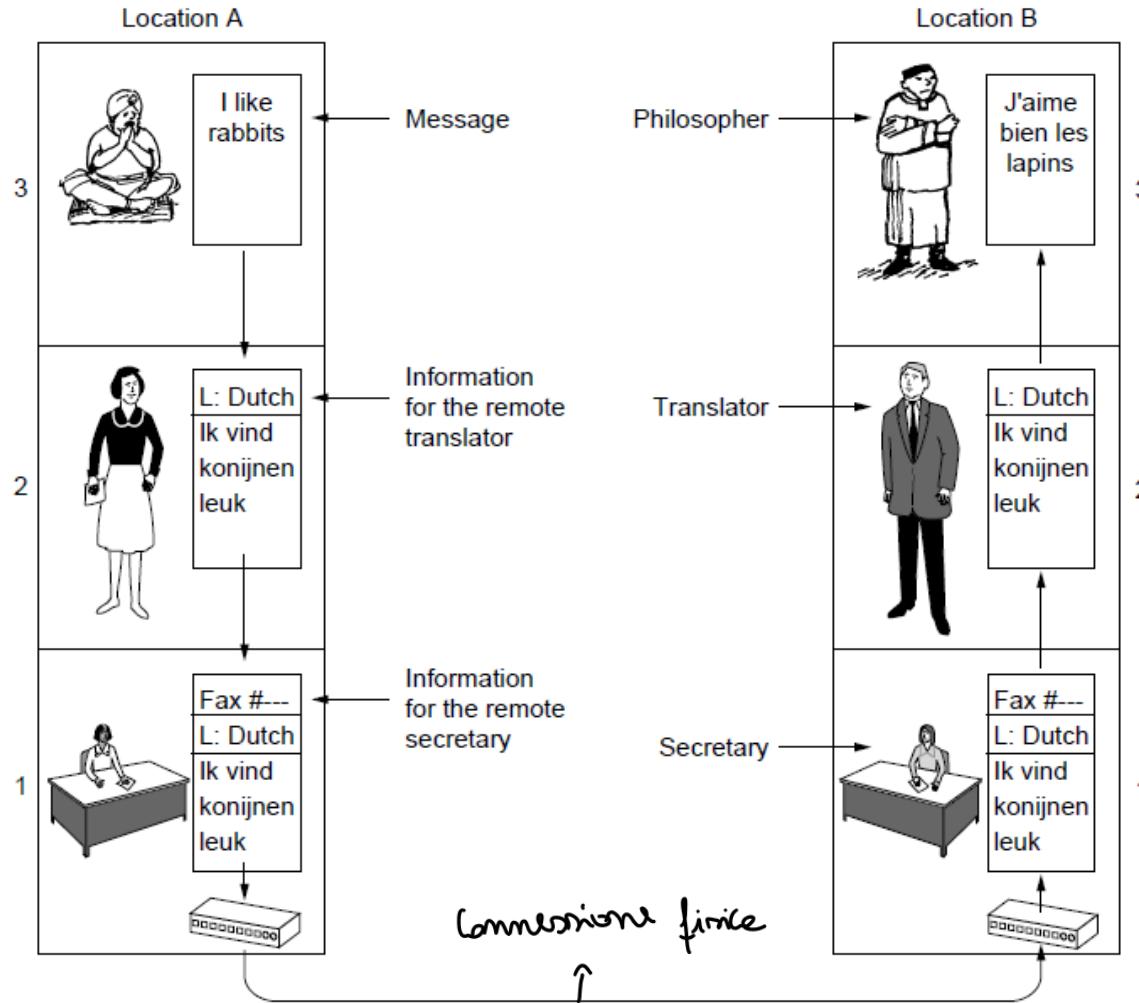
- 100,000+ servers
- 72 countries
- 1,500+ locations
- 1,000 networks



### ■ Ginormous Daily Traffic

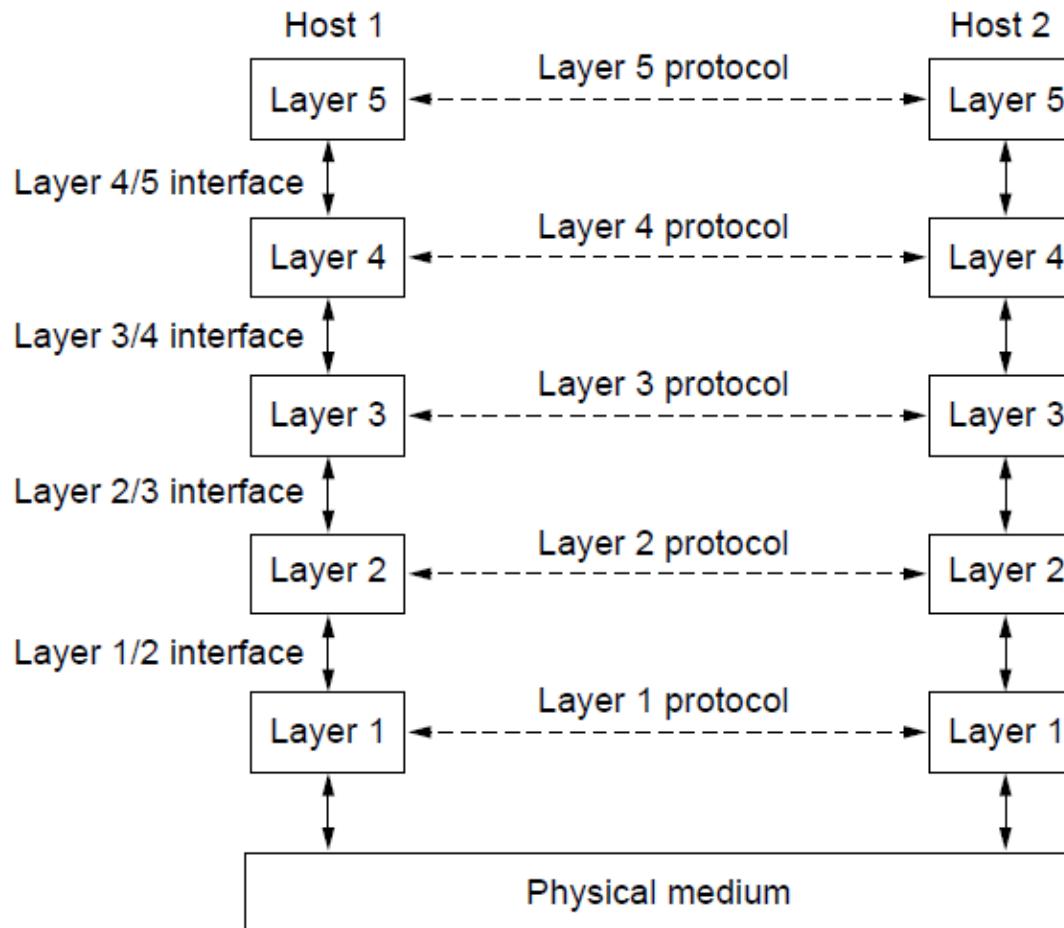
- Carries 15-30% of the world's web traffic on any given day
- More than 1 trillion requests
- More than 30 petabytes
- 10 million+ concurrent video streams

# Protocol Hierarchies (1)



The philosopher-translator-secretary architecture

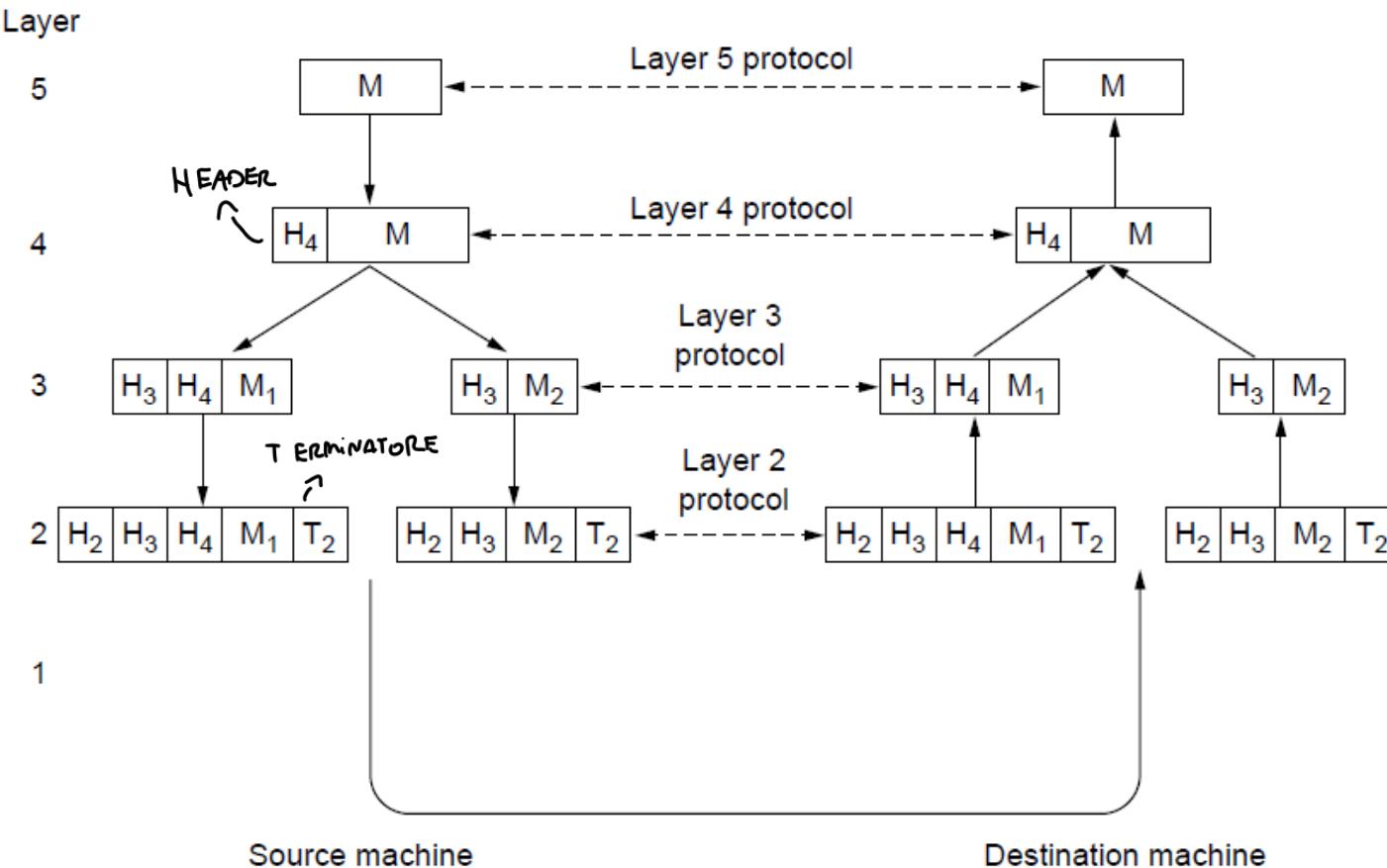
# Protocol Hierarchies (2)



Layers, protocols, and interfaces.

# Protocol Hierarchies (3)

→ su modello osi  
che avranno + livelli



**Example information flow supporting virtual communication in layer 5.**

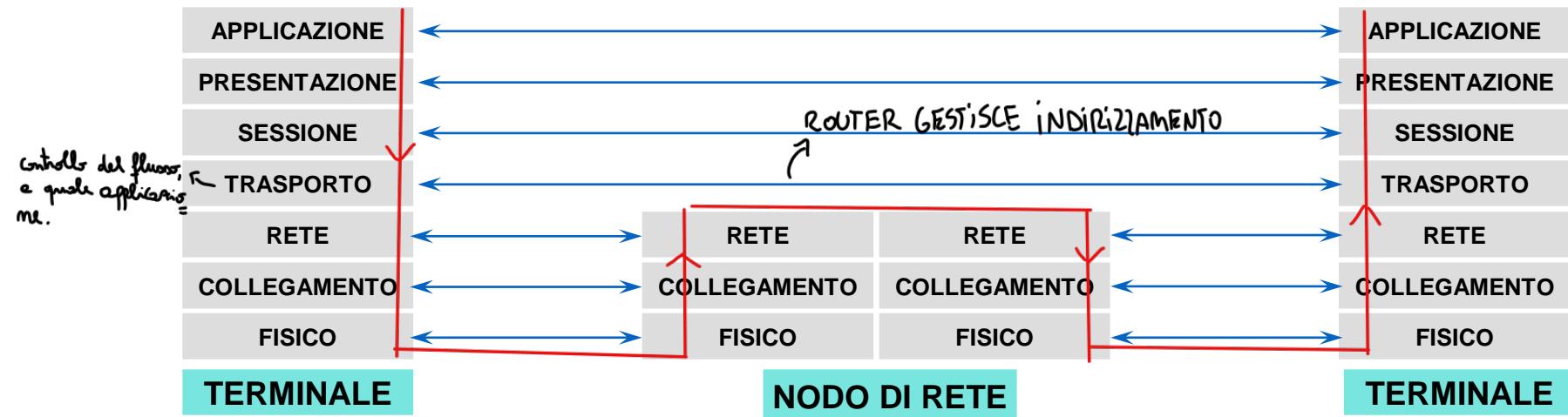
# The OSI Reference Model

## Principles for the seven layers

- Layers created for different abstractions
- Each layer performs well-defined function
- Function of layer chosen with definition of international standard protocols in mind
- Minimize information flow across interfaces between boundaries
- Number of layers optimum

# Il modello di comunicazione OSI

## ESEMPIO DI PROFILO DEI PROTOCOLLI PER IL PIANO UTENTE (commutazione di pacchetto)



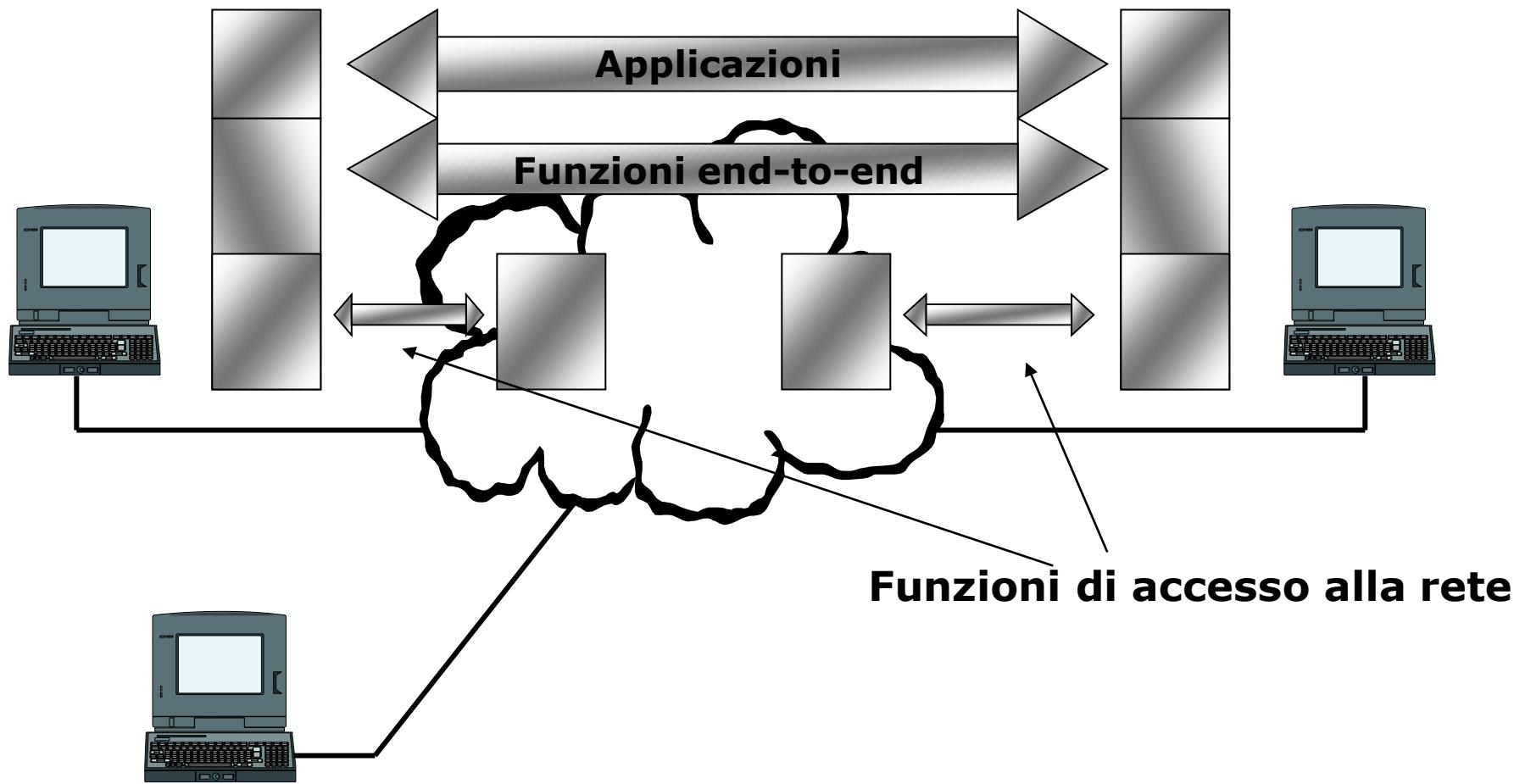
## ESEMPIO DI PROFILO DEI PROTOCOLLI PER IL PIANO UTENTE (commutazione di circuito)



# Critique of the OSI Model and Protocols

- Bad timing.  
→ homos impiega molto tempo
- Bad technology.  
→ ormai vecchie tecnologie
- Bad implementations.
- Bad politics.

# Rete geografica di calcolatori



# Struttura a tre livelli di una rete di calcolatori

Area Applicativa

Interoperabilità trasporto dell'informazione

TRASPORTO

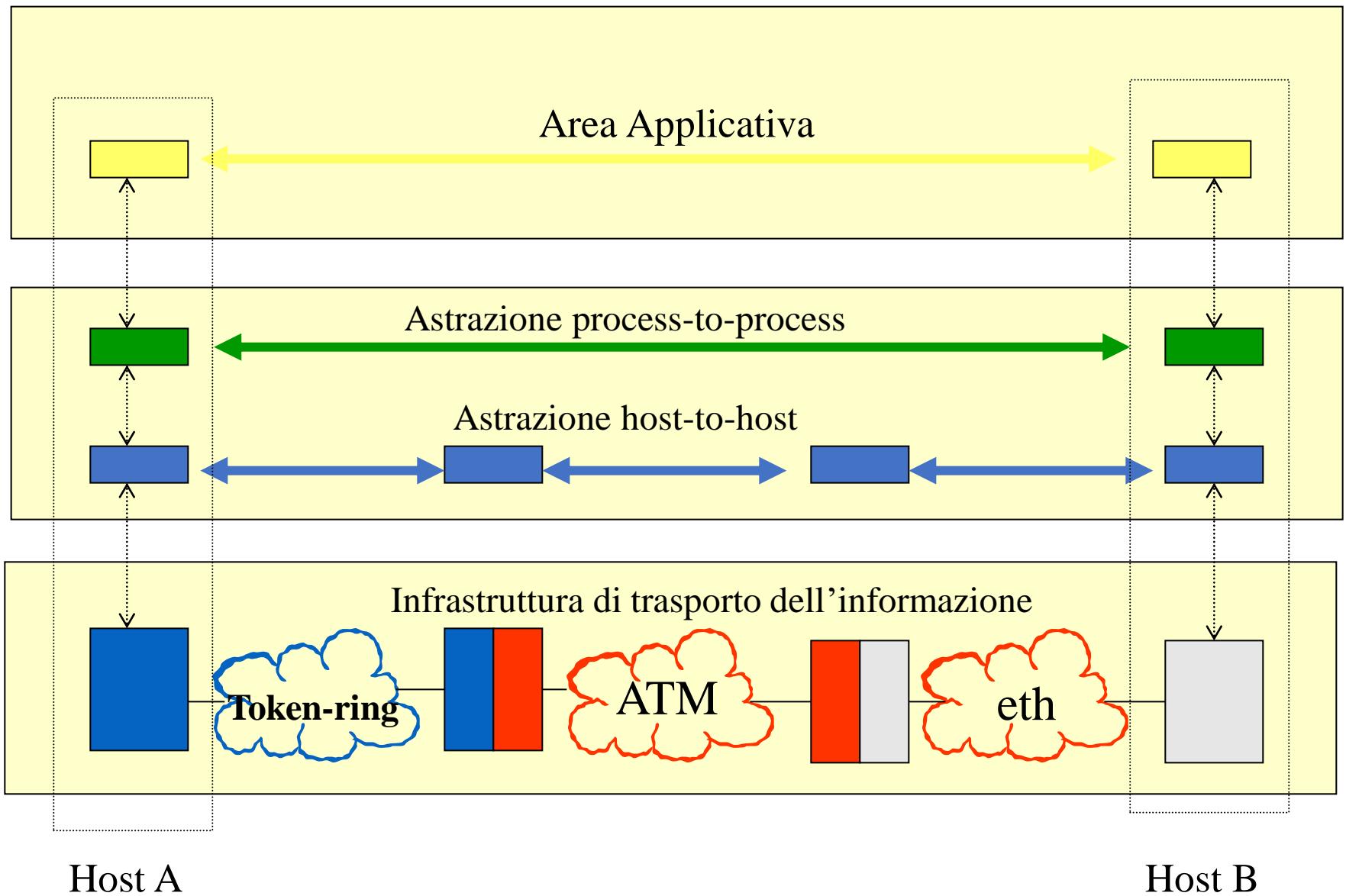
RETE

LINK

FISICO

Infrastruttura di trasporto dell'informazione

# Rete geografica di calcolatori



## Esempi di problematiche comuni: Indirizzamento

Area Applicativa

Interoperabilità trasporto  
dell'informazione

Infrastruttura di trasporto  
dell'informazione

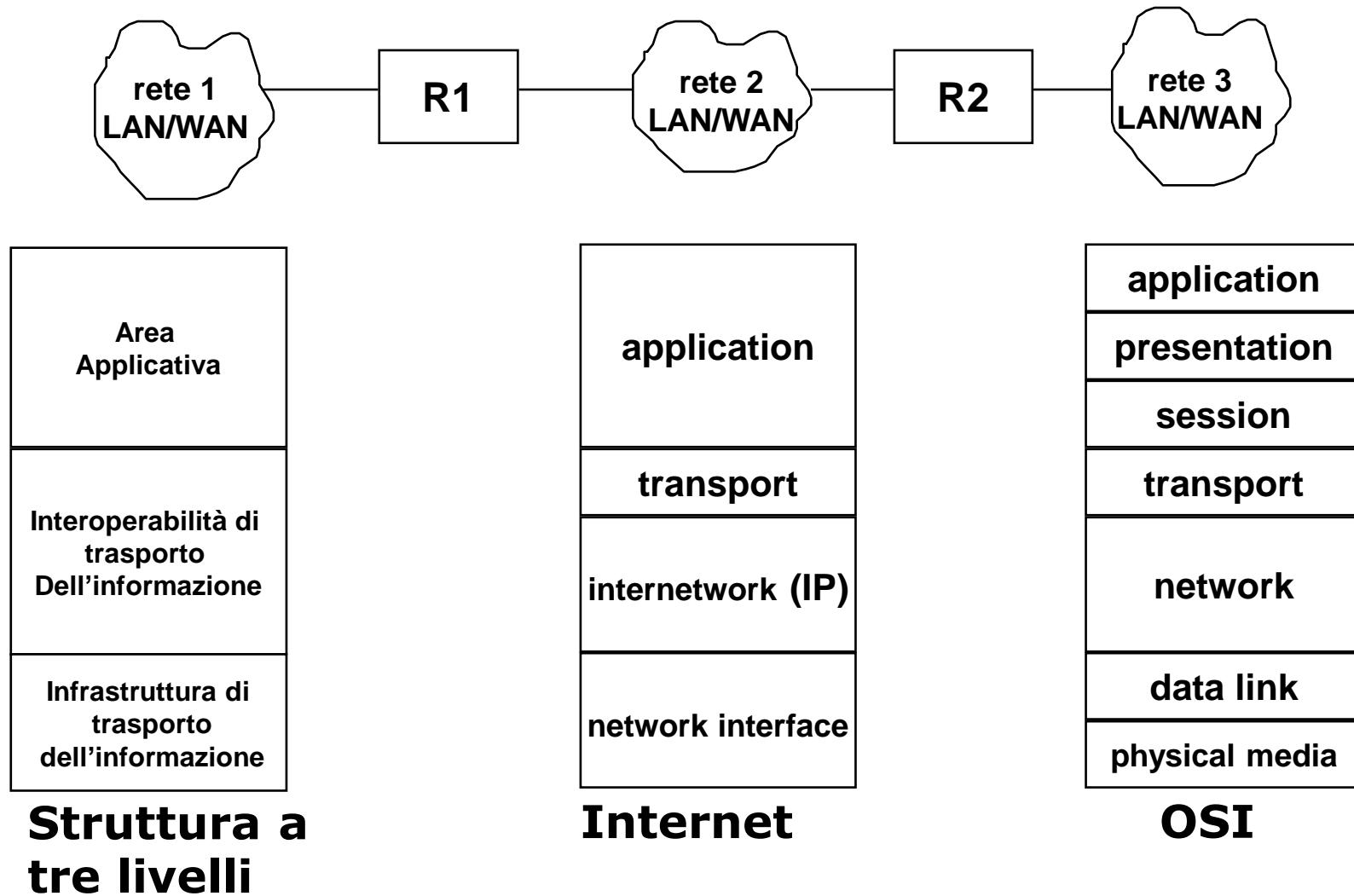
**Indirizzamento DNS “www.uniroma1.it”**

**Indirizzamento IP “151.100.16.1”**

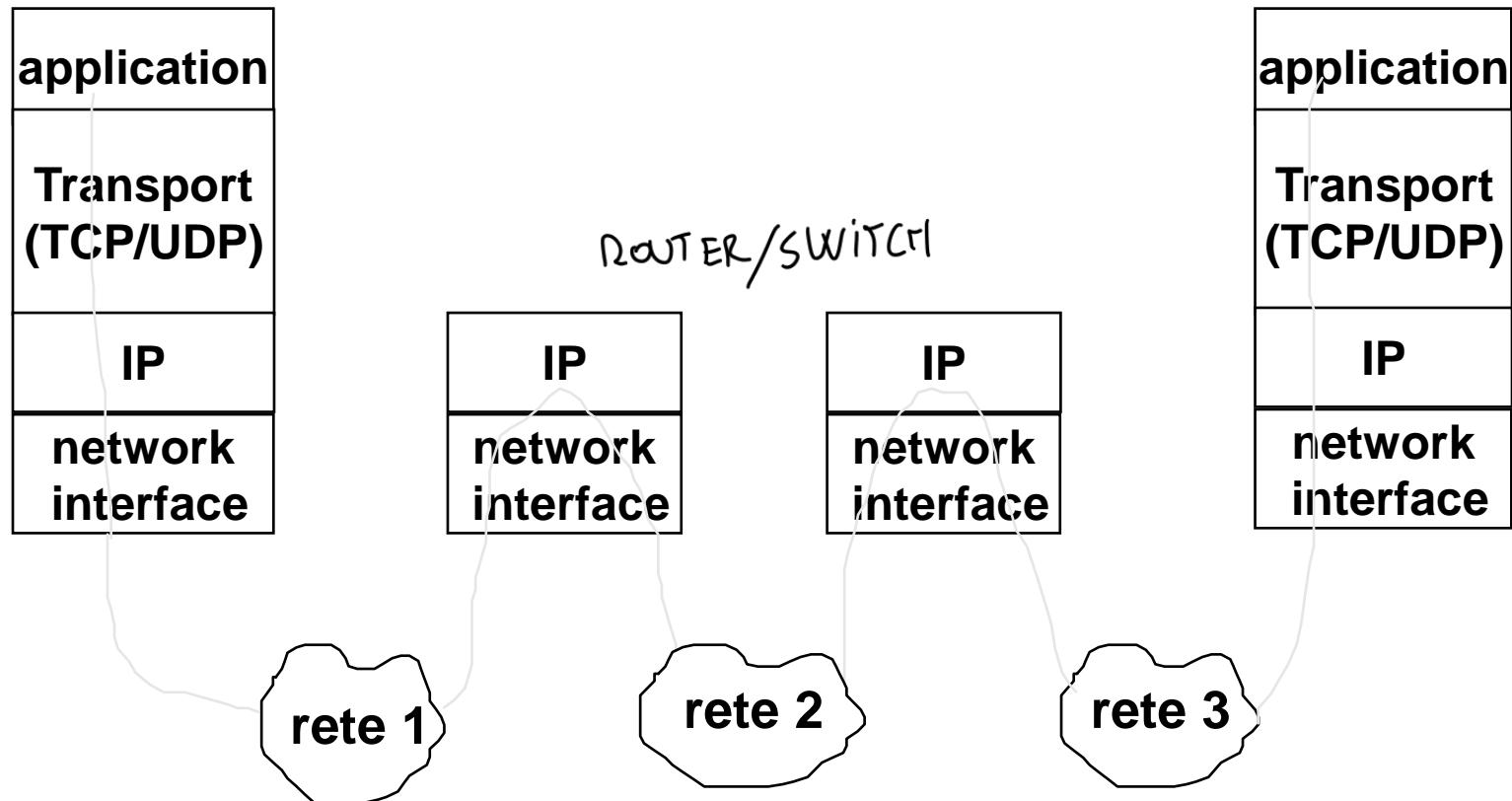
**Indirizzamento MAC “ABC123578ABB”**



# Interoperabilità Trasporto dell'informazione: Internet



# L'ARCHITETTURA TCP/IP E LA RETE INTERNET

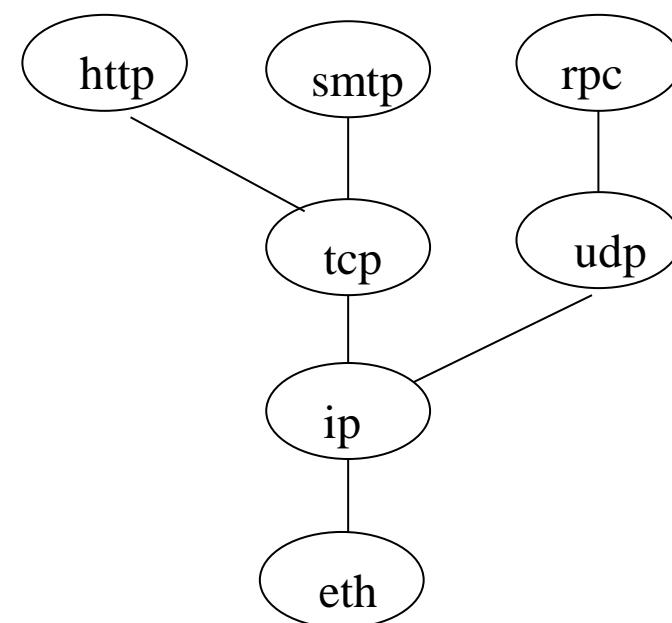


## Protocol Stack: esempi

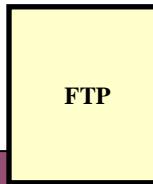
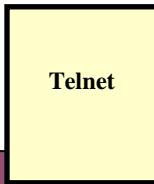
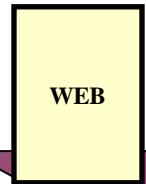
http= hyper text tranfer protocol

smtp= simple mail transfer protocol

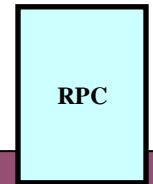
Rpc= remote procedure call



## Applicazioni di base



## Supporto per interoperabilità applicativa



Area delle applicazioni

garantisce  
trasporto e  
le sessione

Interoperabilità di trasporto dell'informazione

PUNTO DI RACCORDO  
TRA TRASPORTO  
E APPLICAZIONI

# TCP/IP

Process-to-process

Host-to-host

Infrastruttura di trasporto dell'informazione

802.1  
Bridging e Switching

802.3  
802.3u  
802.3z  
CSMA/CD

802.5  
TOKEN RING

FDDI  
ISO  
9314

802.11  
Wireless

X.25

Frame Relay

ATM

Reti Locali

Backbone

# Basi di TCP/IP

# Il protocollo IP

- IP è una grande coperta che nasconde ai protocolli sovrastanti tutte le disomogeneità della infrastruttura di trasporto dell'informazione
- Per far questo necessita di due funzionalità di base:  
  - Indirizzamento di rete (indirizzi omogenei a dispetto della rete fisica sottostante)
  - Instradamento dei pacchetti (Routing) (capacità di inviare pacchetti da un host ad un altro utilizzando gli indirizzi definiti al punto precedente) ~> trovare strada migliore per la destinazione.

## Proprietà di IP

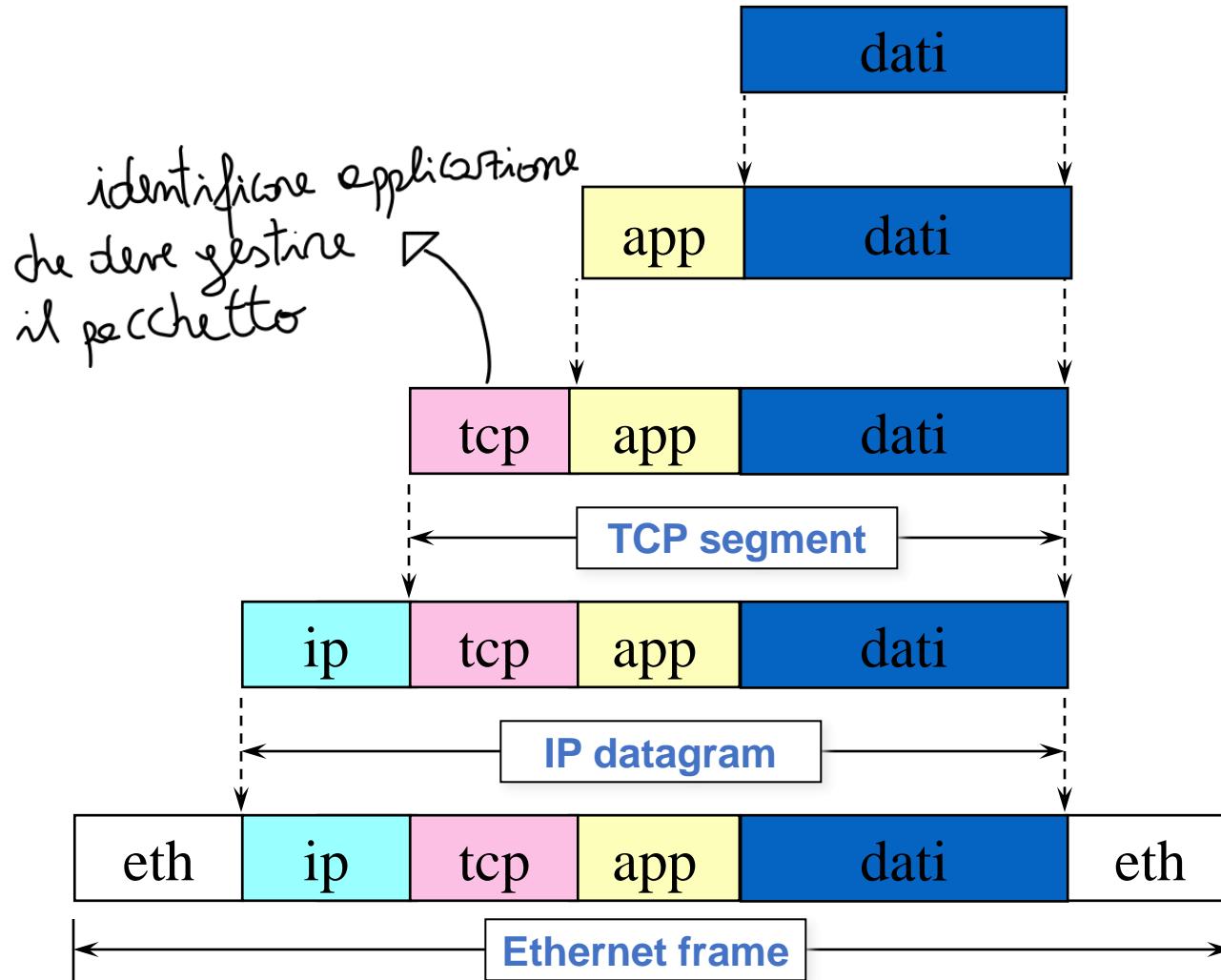
- Senza connessione (datagram based)
  - Consegnata Best effort
    - I pacchetti possono perdere
    - I pacchetti possono essere consegnati non in sequenza
    - I pacchetti possono essere duplicati
    - I pacchetti possono subire ritardi arbitrari
- NON ABBIAMO GARANZIE  
DI QUALITÀ A LIVELLO  
DI IP.

## Servizi di compatibilità con l'hardware sottostante

- Frammentazione e riassemblaggio
- Corrispondenza con gli indirizzi dei livelli sottostanti (ARP)

↳ risolvere problema  
MAC

# Il protocollo IP



# Il protocollo IP

*In Trasmissione, IP*

- riceve il segmento dati dal livello di trasporto

Segmento dati

- inserisce header e crea datagram

IP

Segmento dati

- applica l'algoritmo di routing

- invia i dati verso l'opportuna interfaccia di rete

*In Ricezione, IP*

- consegna il segmento al protocollo di trasporto individuato

Segmento dati

- se sono dati locali, individua il protocollo di trasporto, elimina l'intestazione

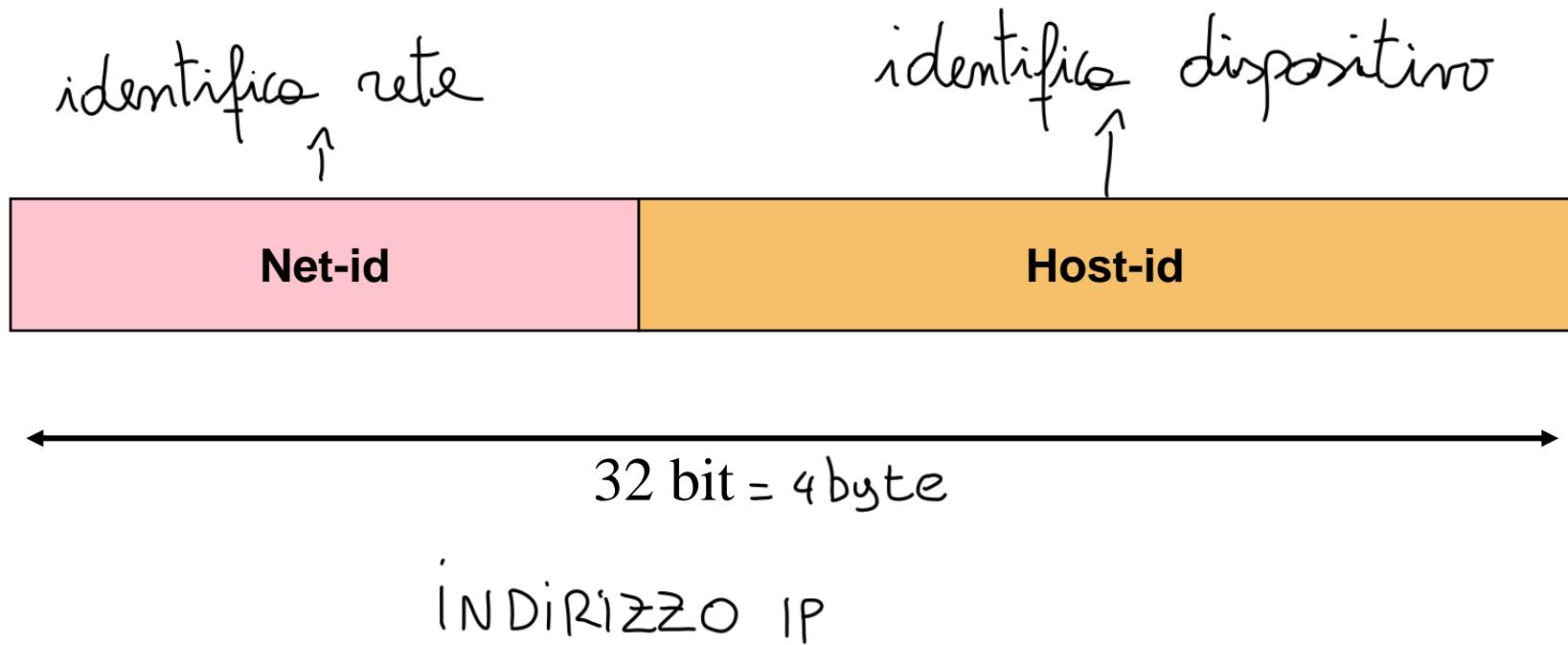
IP

Segmento dati

- verifica la validità del datagram e l'indirizzo IP

- riceve i dati dalla interfaccia di rete

# Indirizzamento



# Classi di indirizzi

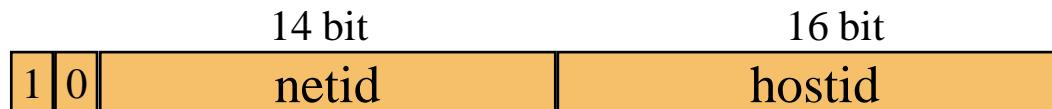
**Classe A** (POCHE RETI CON MILIONI DI HOST) **(0.0.0.0 - 127.255.255.255)**

**127.0.0.0 riservato**



**Classe B**

**(128.0.0.0 - 191.255.255.255)**



**Classe C**

**(192.0.0.0 - 223.255.255.255)**



**Classe D** (UNICAST)

**(224.0.0.0 - 239.255.255.255)**



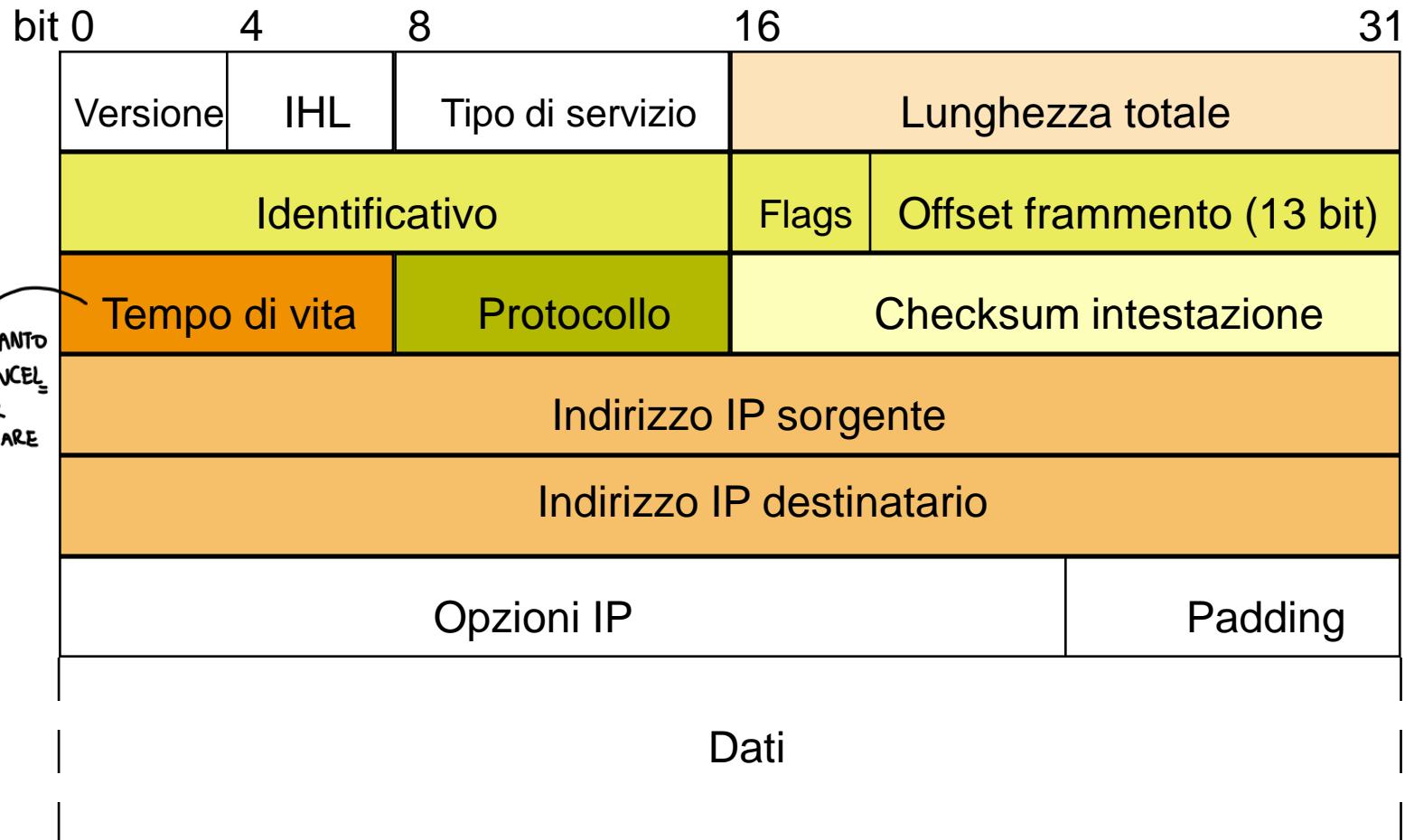
**Classe E** (PER POCHE PERSONE)

**(240.0.0.0 - 255.255.255.254)**

27 bit



# Il protocollo IP

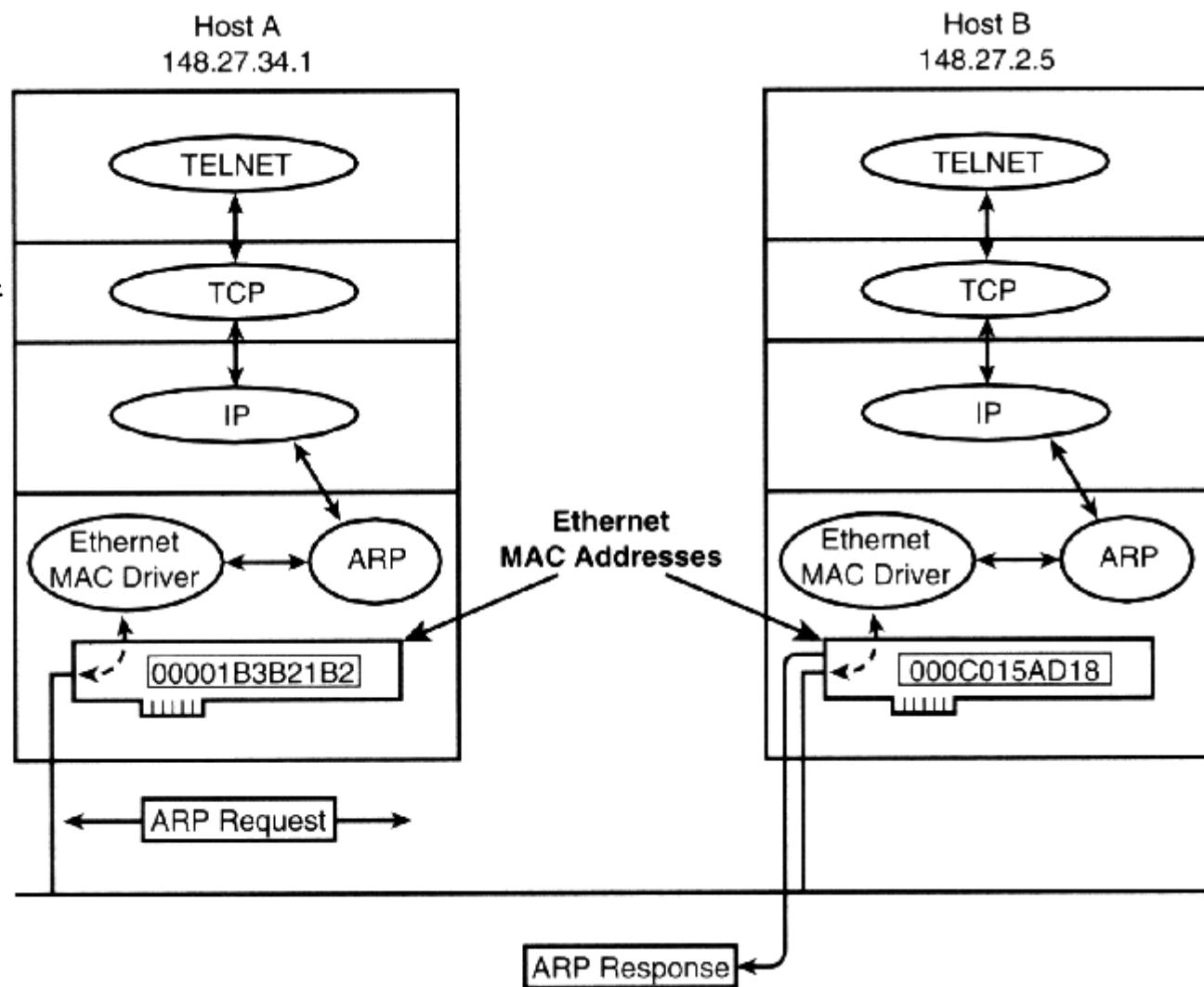


# Address Resolution Protocol: ARP

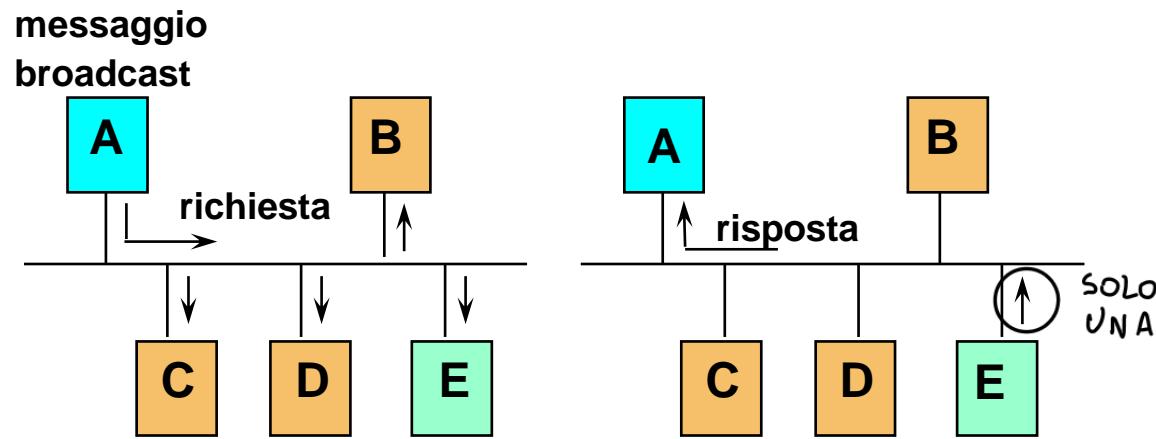
**FIGURE 4.9.**

*Resolution of an IP address into its MAC address using ARP.*

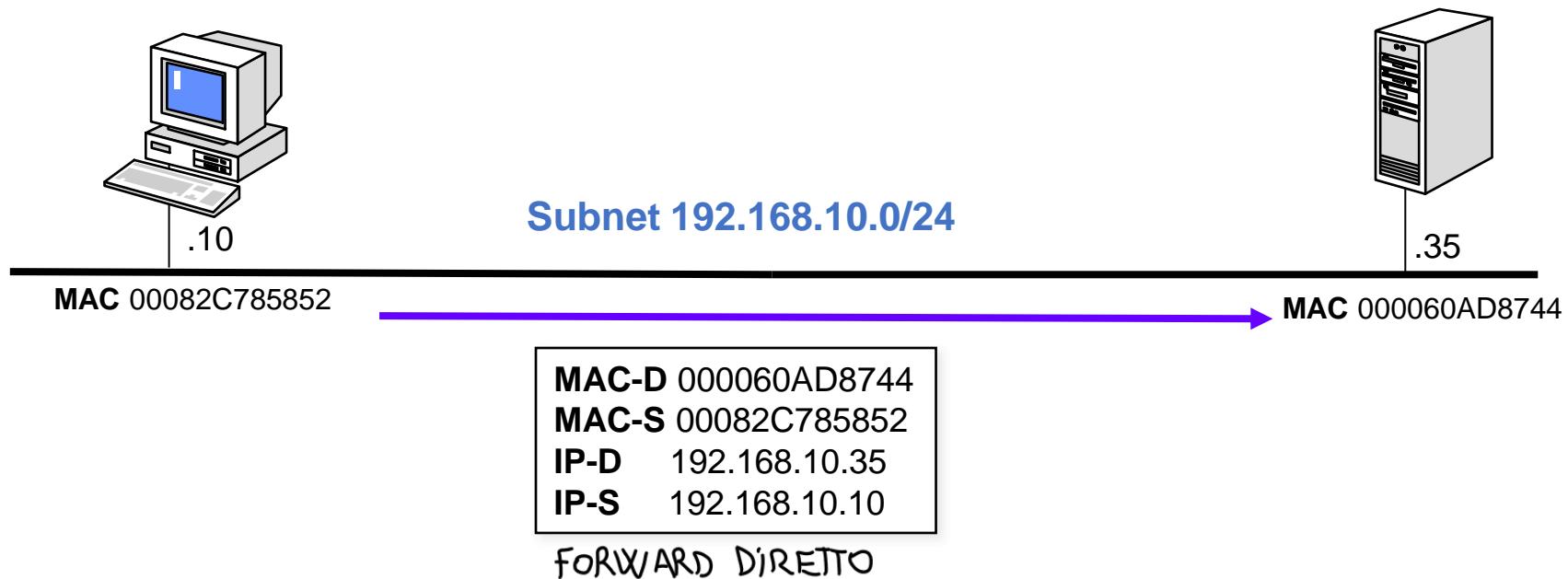
NON CONOSCO MAC DESTINAZIONE, USO ARP per individuare e chiedere l'indirizzo IP.



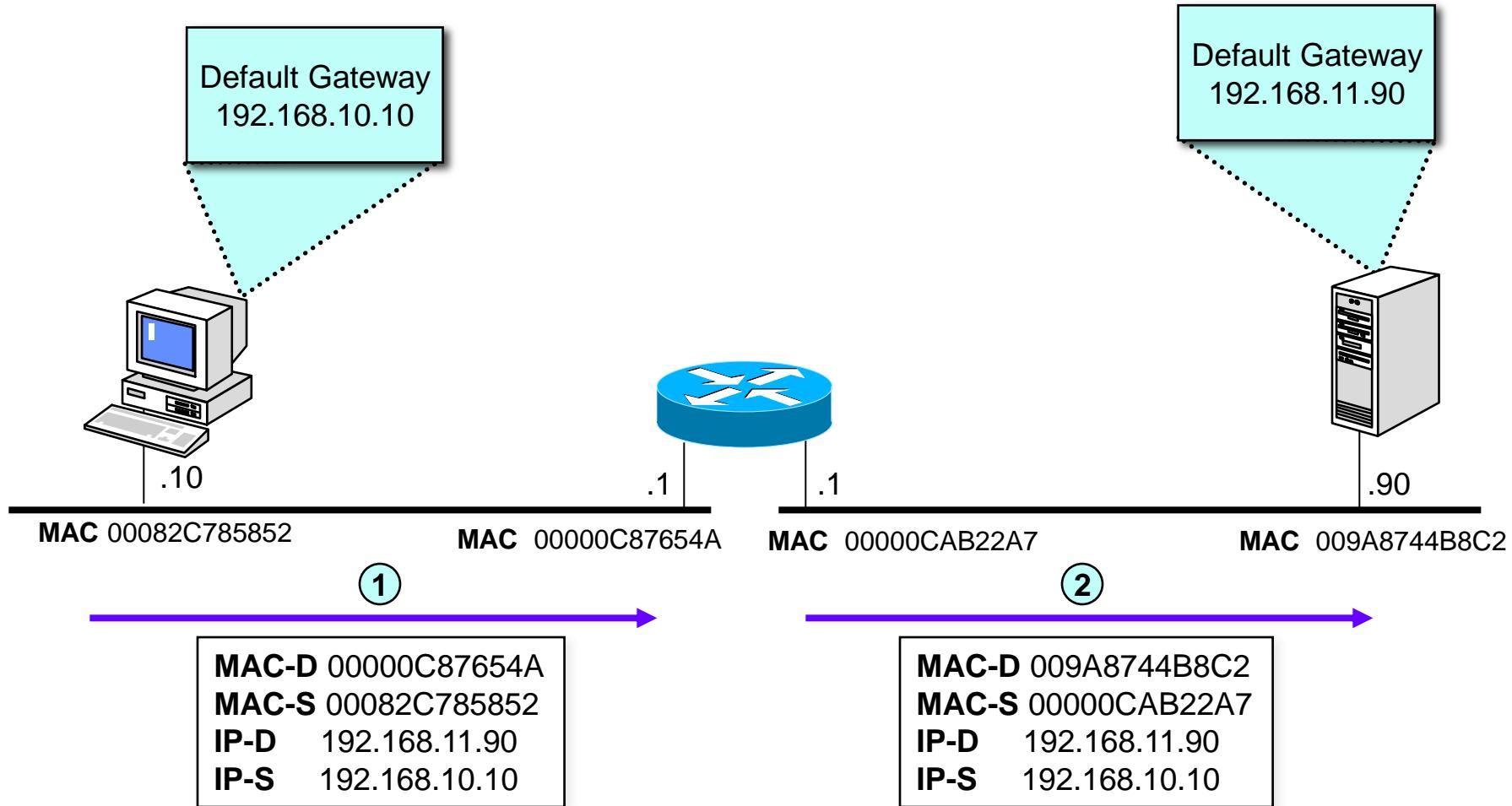
# Address Resolution Protocol: ARP



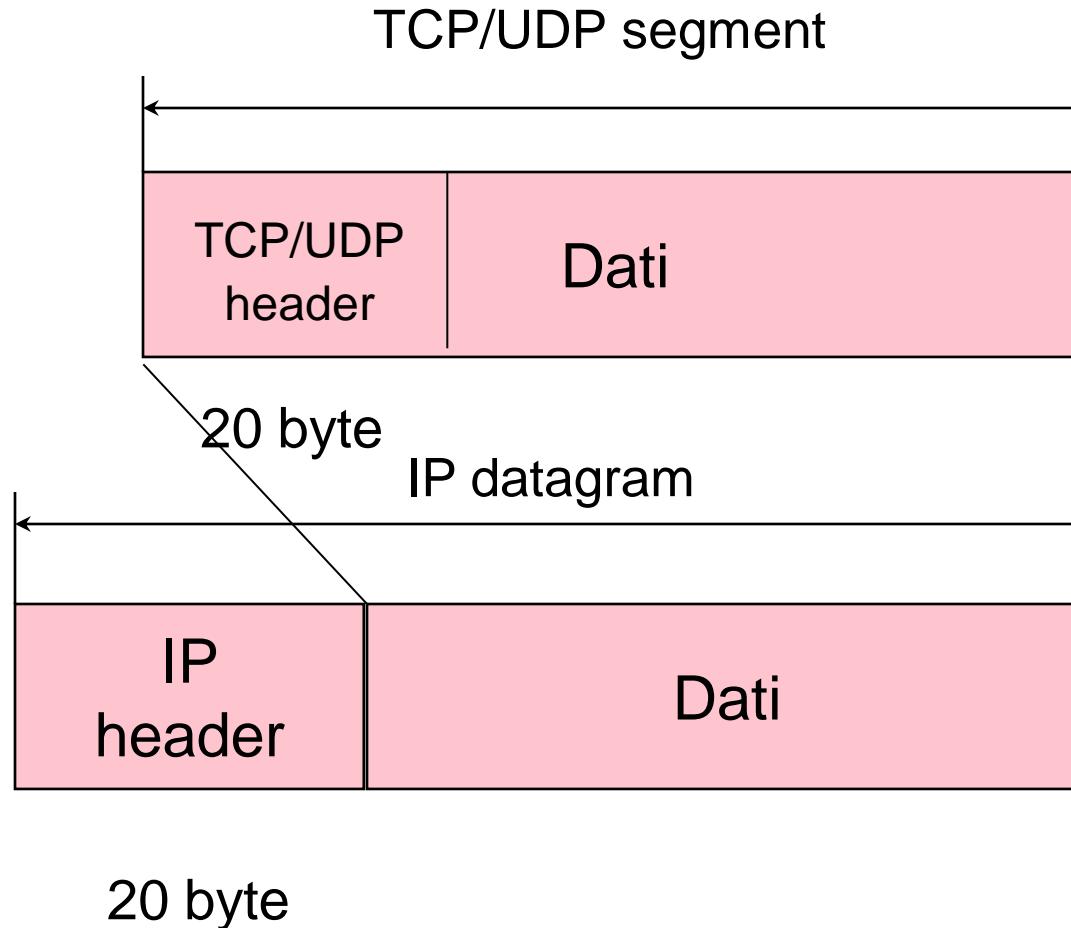
# Forwarding diretto: esempio



# Forwarding indiretto: esempio



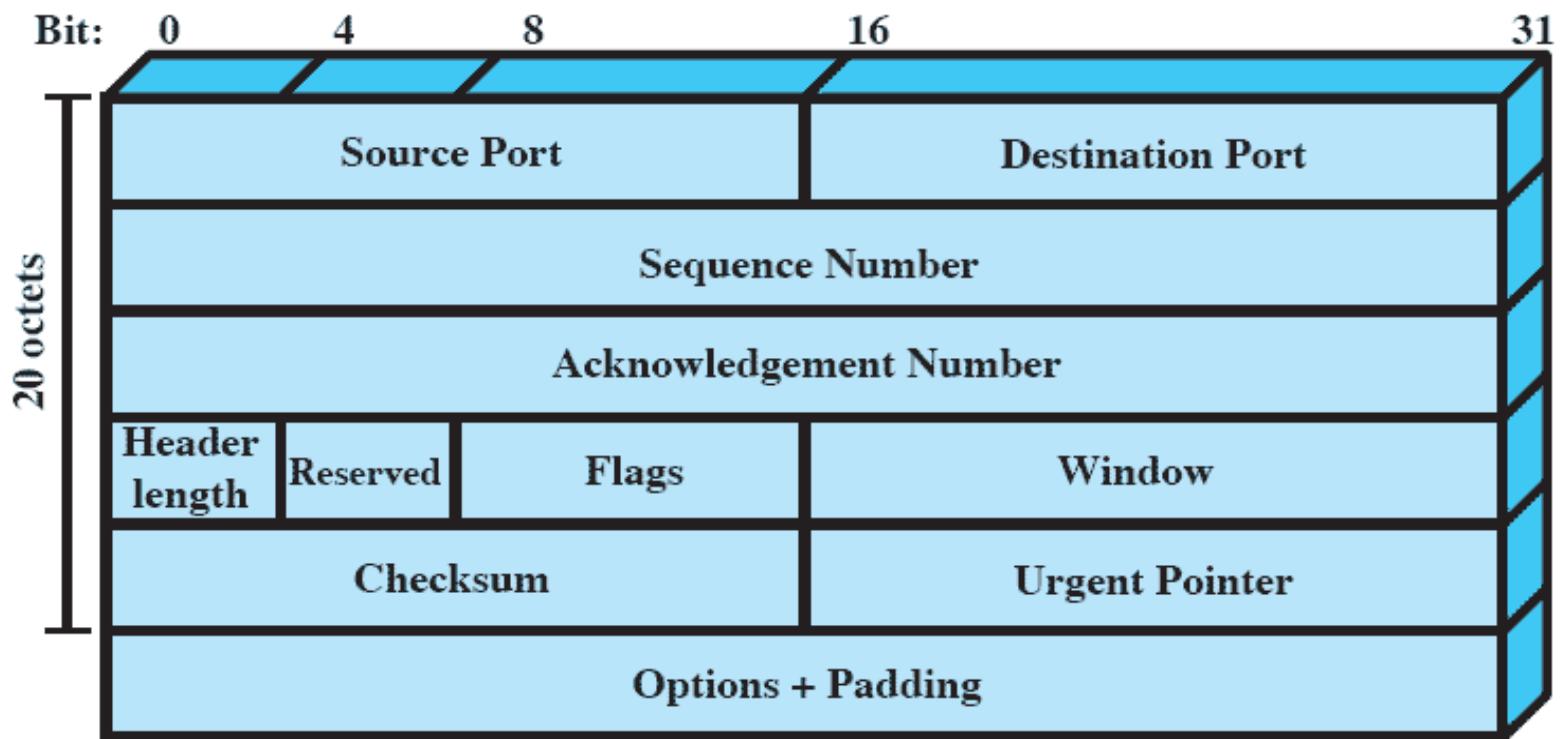
# Strato di Trasporto



# Strato di Trasporto



# TCP header

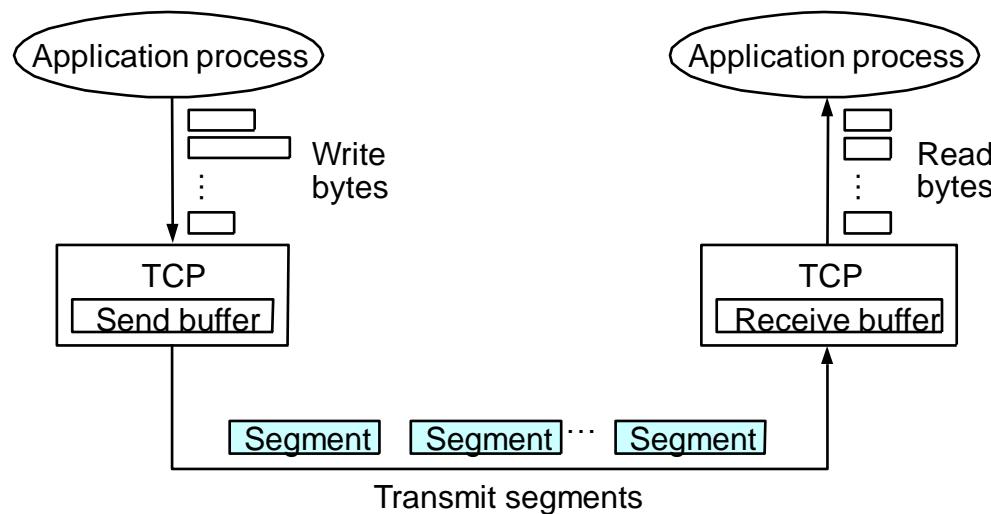


# TCP Overview

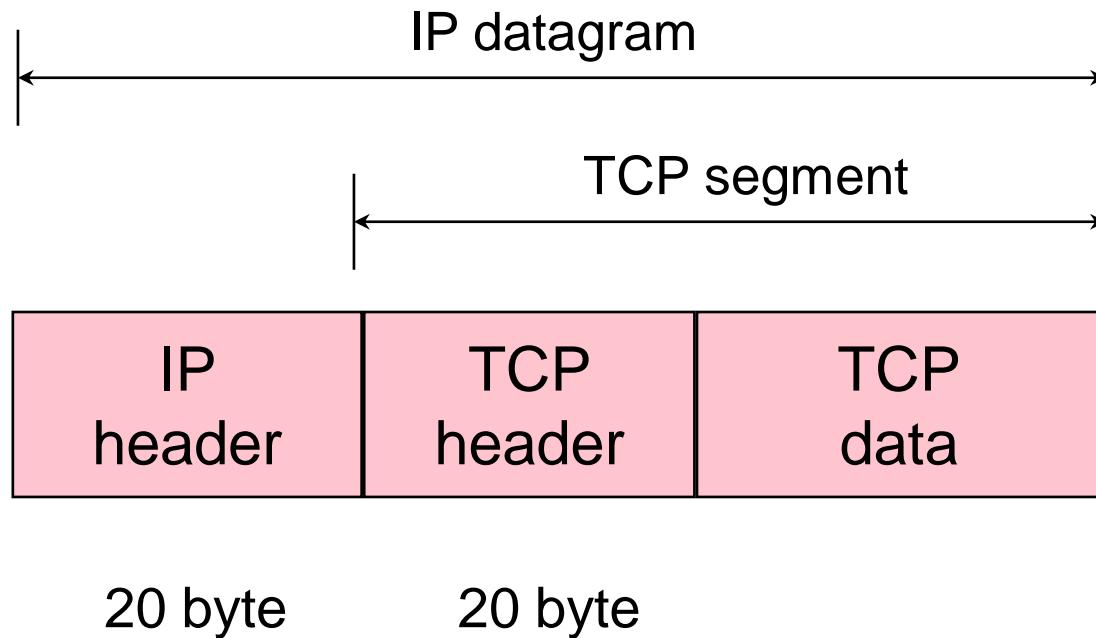
(OLTRE IL MESSAGGIO, INSTAURO UNA CONNESSIONE)

- Connection-oriented
- Byte-stream
  - app writes bytes
  - TCP sends *segments*
  - app reads bytes

- Full duplex
- Flow control: keep sender from overrunning receiver (NON SATURARE Ricev.)
- Congestion control: keep sender from overrunning network (NON INTASARE LA RETE)



# Strato di Trasporto: TCP



# Protocollo senza connessione

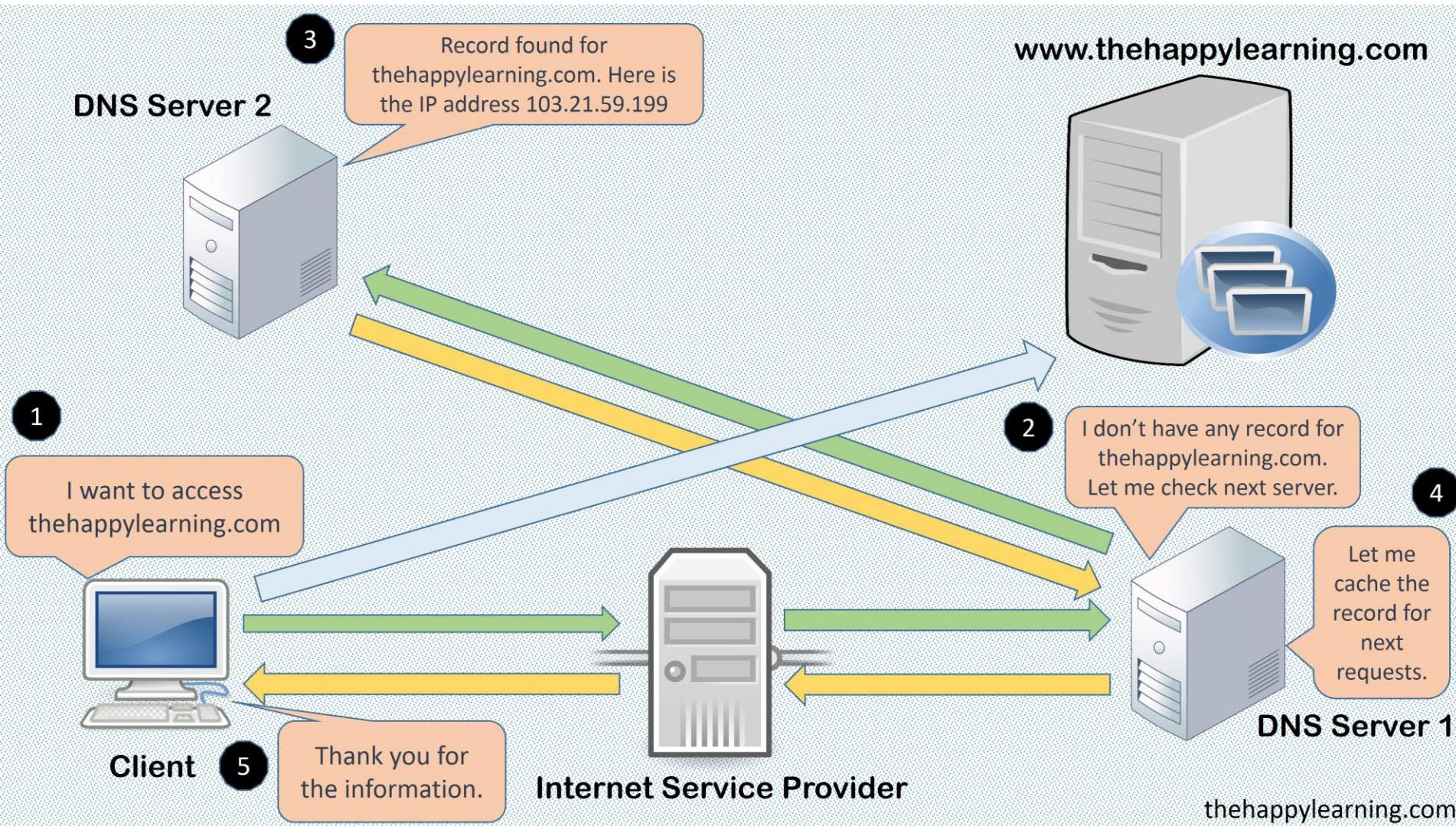
## UDP



- **minimum protocol mechanism**
  - **connectionless**
  - **no guarantees about delivery, preservation of sequence, nor protection against duplication**
  - **useful, e.g., for transaction-oriented applications**
  - **multicast support**

DNS per trovare IP di un sito

# Domain Name System (DNS)



# Concurrency again

*Sistemi di Calcolo 2*

*Instructor: Riccardo Lazzeretti*

Software solutions for  
syncronization

# Dijkstra's Algorithm

```
/* global storage */
boolean interested[N]      = {false, ..., false}
boolean passed[N]    = {false, ..., false}
int k = <any>                  // k ∈ {0, 1, ..., N-1}

/* local info */
int i = <entity ID> // i ∈ {0, 1, ..., N-1}
1. interested[i] = true
2. while (k != i) {
3.     passed[i] = false
4.     if (!interested[k]) then k = i
    }
5. passed[i] = true
6. for j in 1 ... N except i do
7.     if (passed[j]) then goto 2
8.     <critical section>
9.     passed[i] = false; interested[i] = false
```

# Dijkstra characteristics

- Mutual Exclusion
- No deadlock
- No starvation?
  - Not guaranteed
- Other problems:
  - Needs atomic read/write
  - Needs memory sharing for k

→ un processo potrebbe rimanere in attesa indefinitivamente

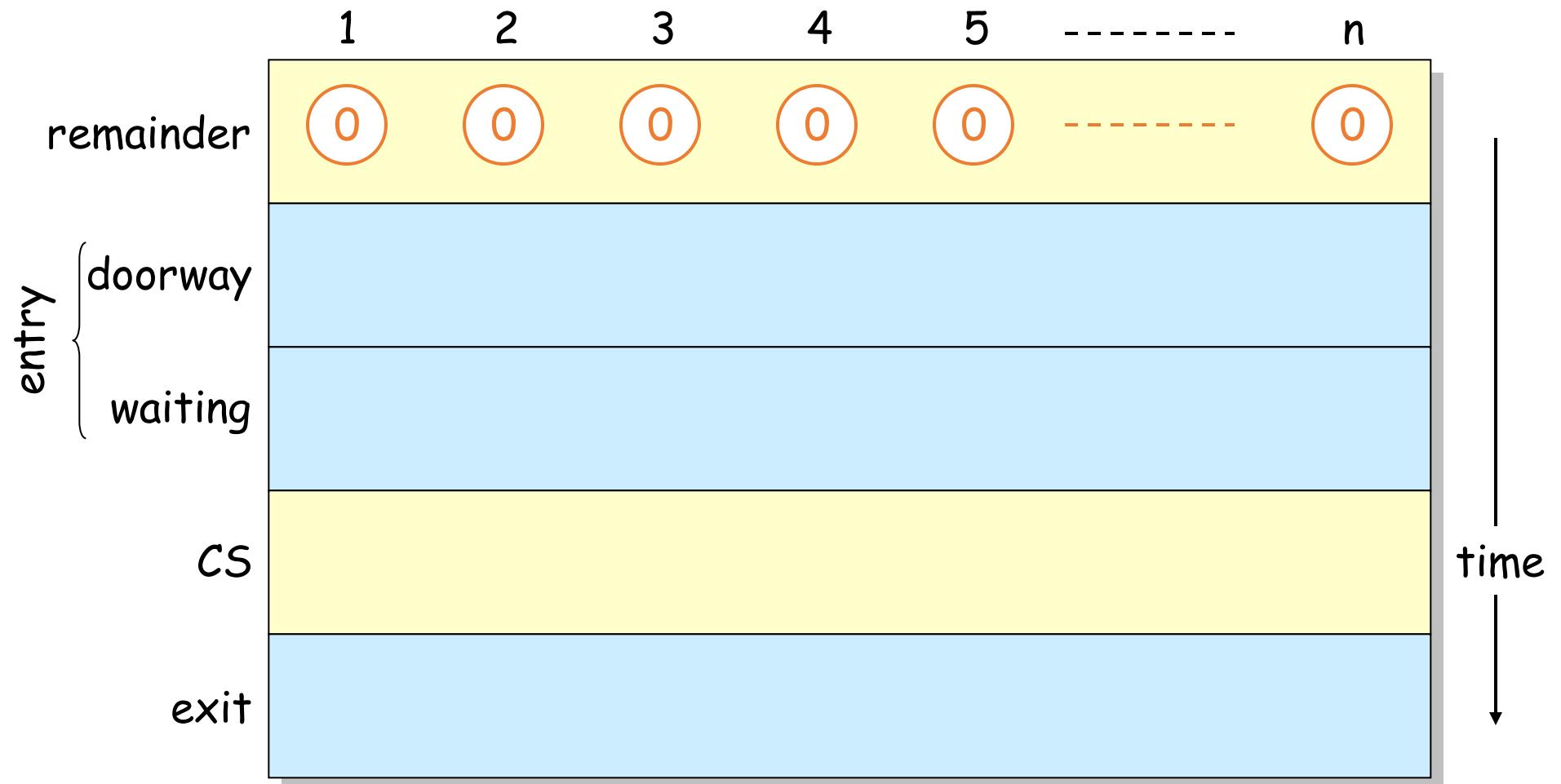
# Bakery Algorithm

## Lamport (1975)

- Concept:
  - Think of a popular store with a crowded counter
    - People take a ticket from a machine
    - If nobody is waiting, tickets don't matter
    - When several people are waiting, ticket order determines order in which they can make purchases

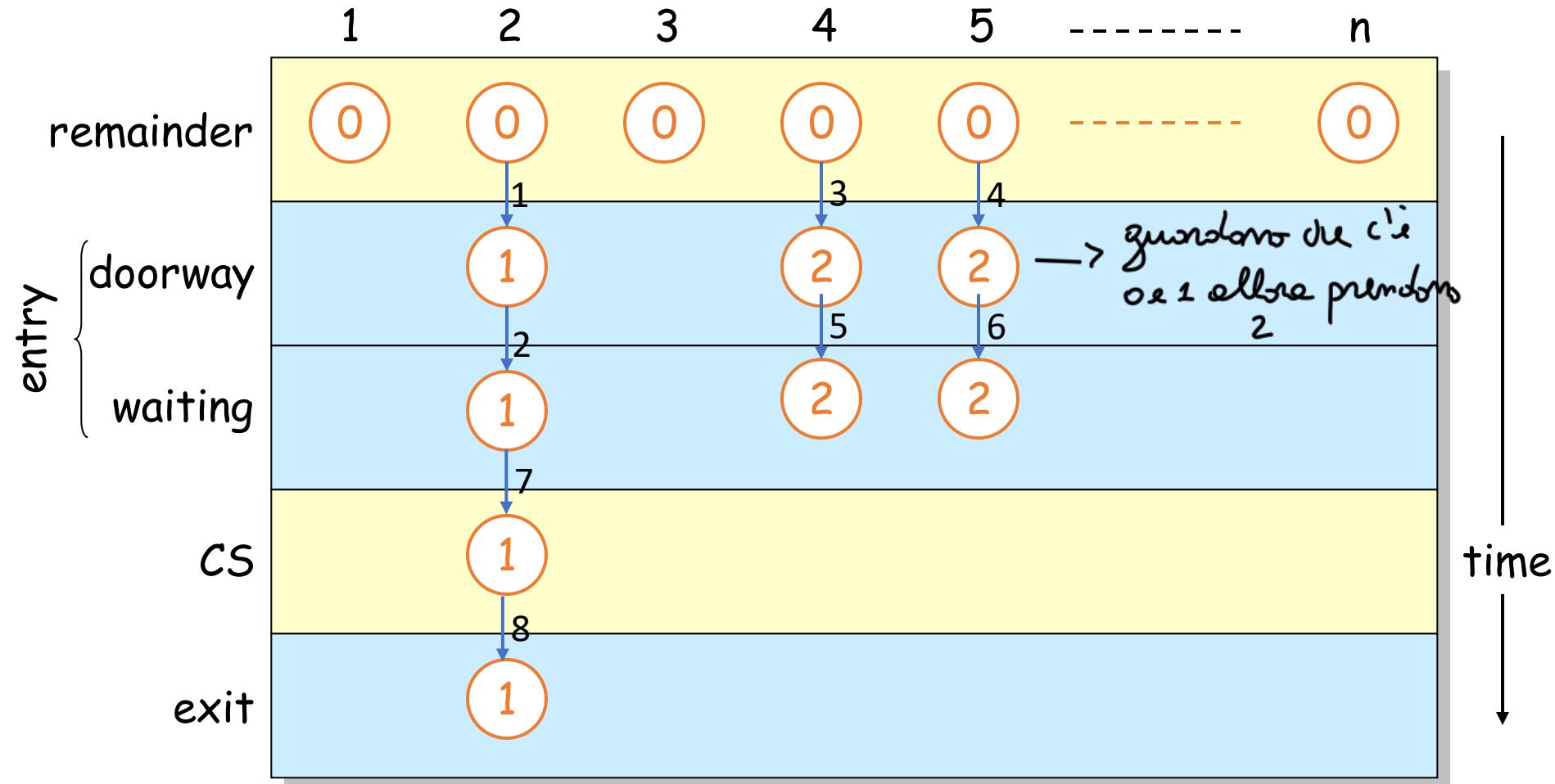
# The Bakery Algorithm

ognuno può leggere  
il proprio numero e gli  
altri.



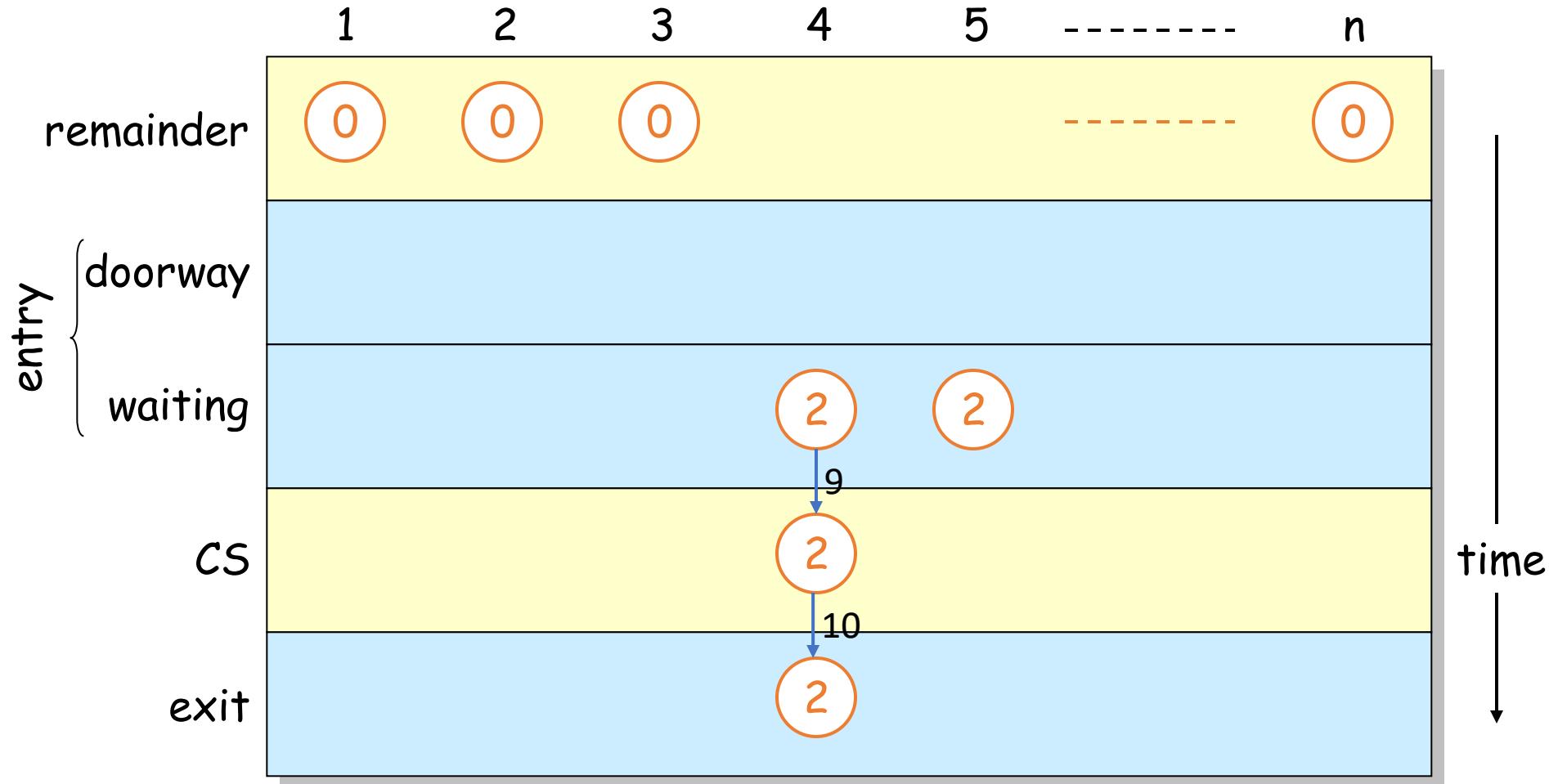
# The Bakery Algorithm

più processi possono avere lo stesso numero.

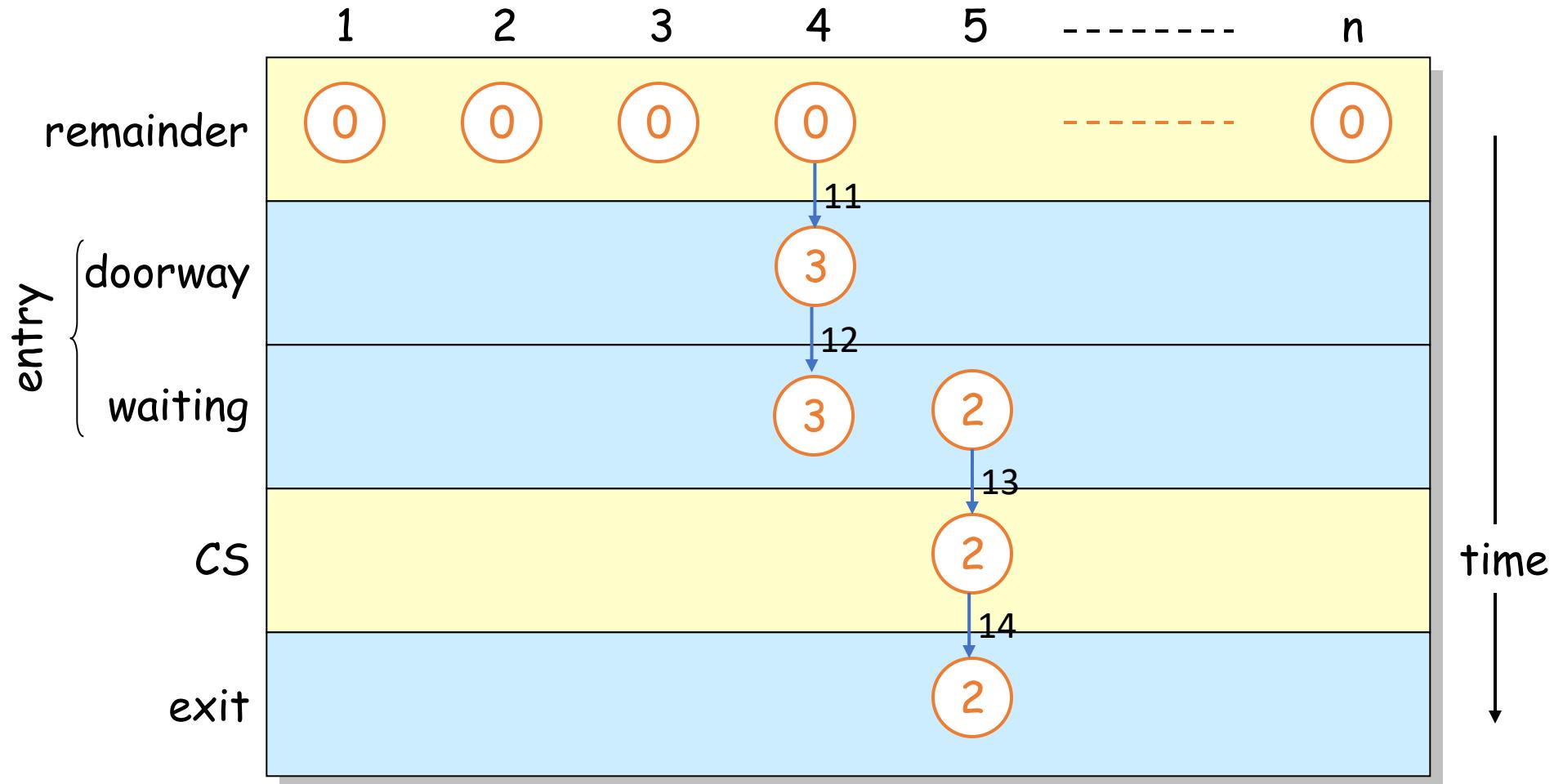


# The Bakery Algorithm

O non è interessante  
in CS



# The Bakery Algorithm



# Implementation 1

code of process  $i$ ,  $i \in \{1, \dots, n\}$

```
while (1){  
    /*NCS*/  
    number[i] = 1 + max {number[j] | (1 ≤ j ≤ N) except i} //Doorway  
    for j in 1 .. N except i {  
        while (number[j] != 0 && number[j] < number[i]);  
    }  
    /*CS*/  
    number[i] = 0; //Bakery  
}
```

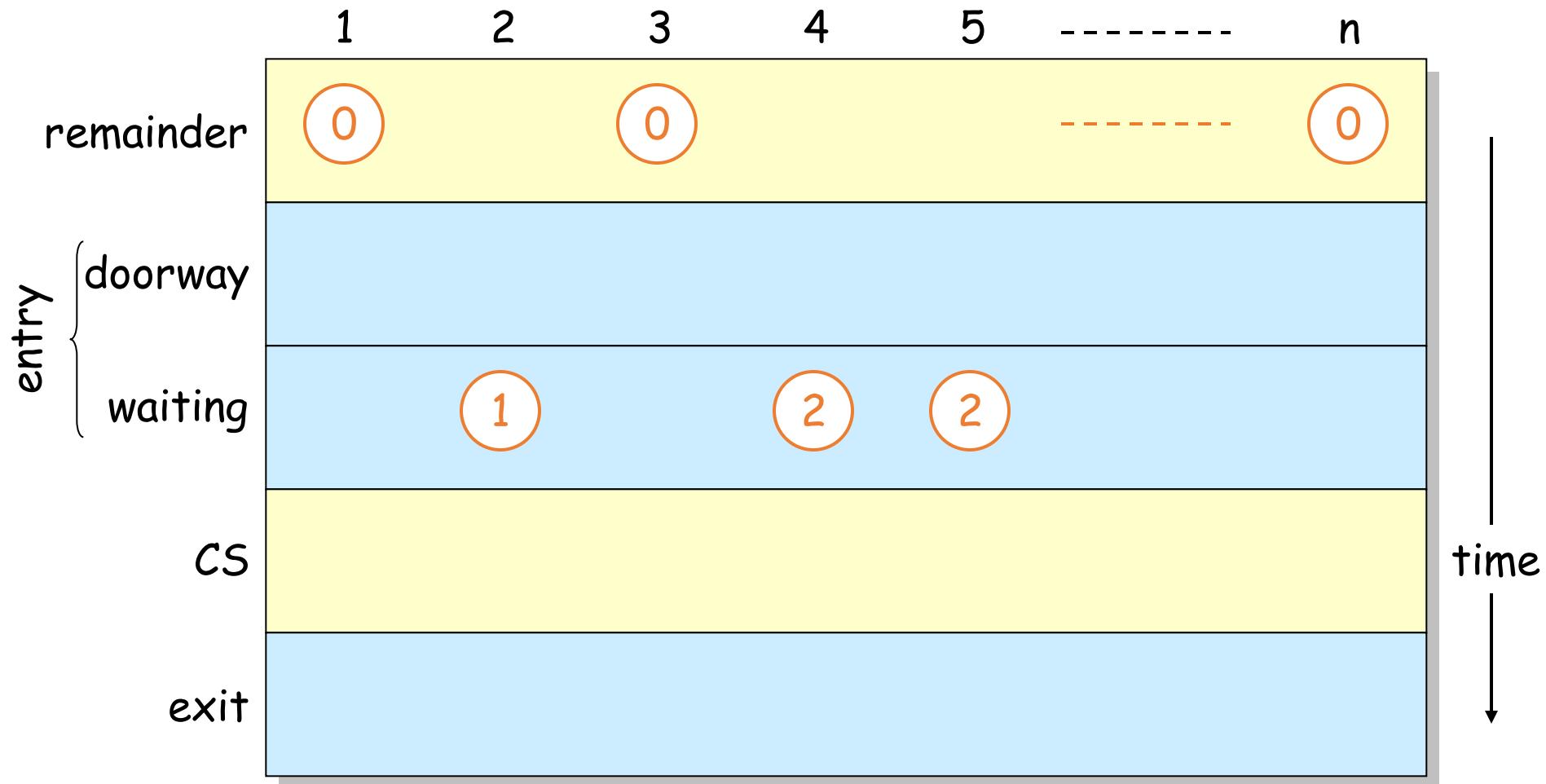
WAITING ROOM

|        |   |   |   |   |       |   |         |
|--------|---|---|---|---|-------|---|---------|
|        | 1 | 2 | 3 | 4 | ----- | n |         |
| number | 0 | 0 | 0 | 0 | 0     | 0 | integer |

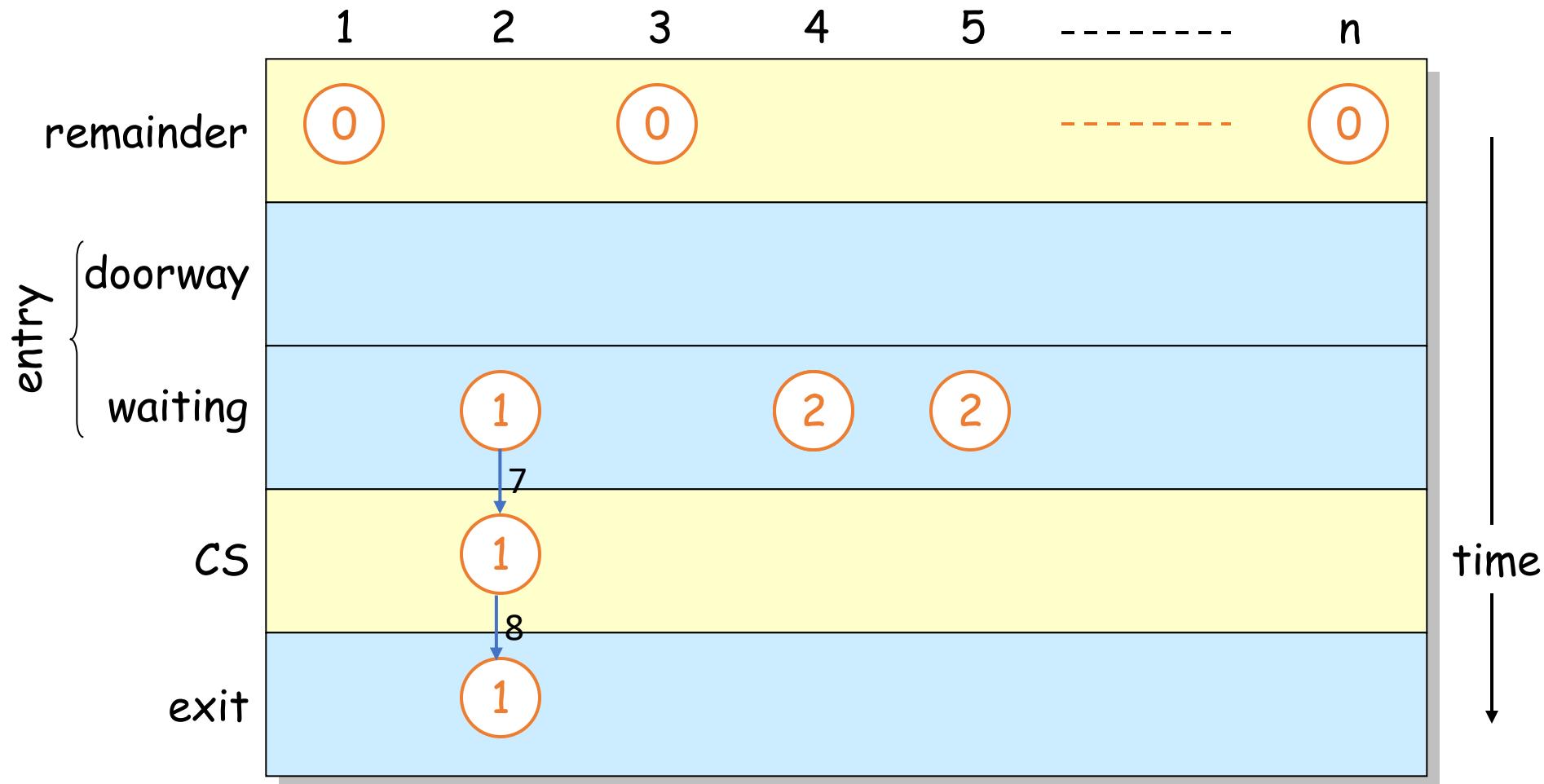
Answer: does not satisfy mutual exclusion

può succedere che due processi calcolino lo stesso numero

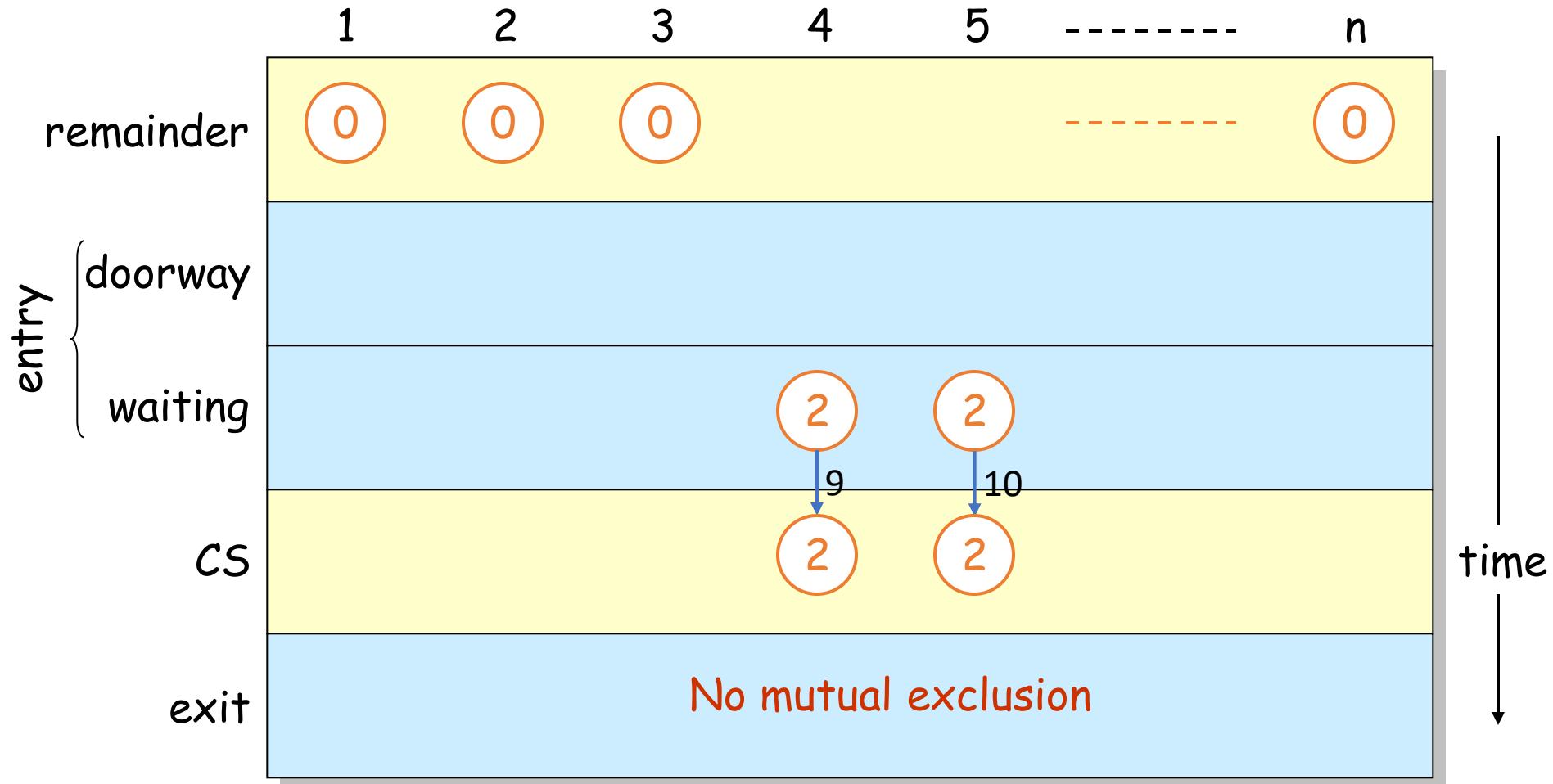
## Implementation 1: deadlock



## Implementation 1: deadlock



## Implementation 1: deadlock



# Implementation 1

code of process  $i$ ,  $i \in \{1, \dots, n\}$

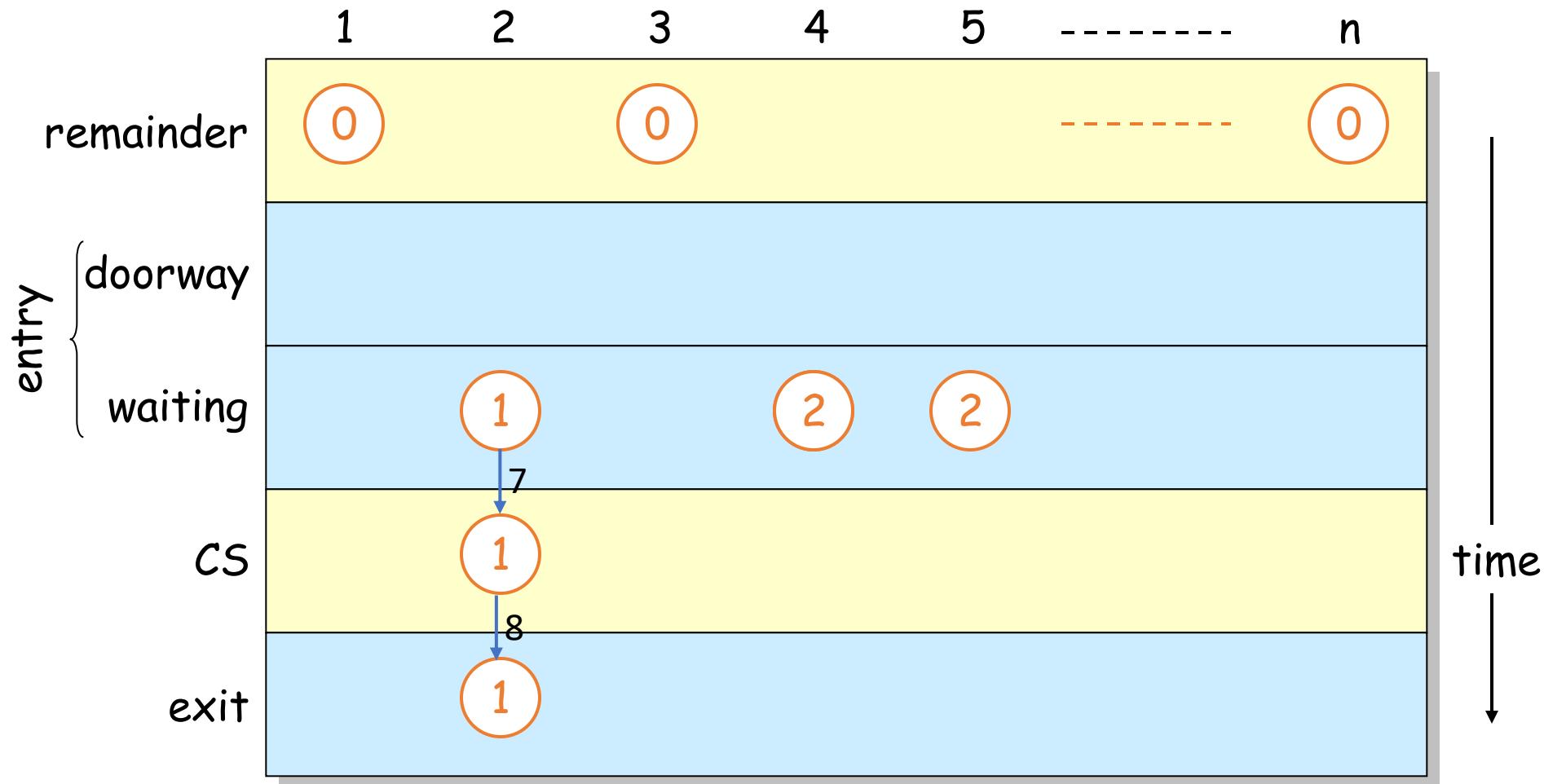
```
while (1){  
    /*NCS*/  
    number[i] = 1 + max {number[j] | (1 ≤ j ≤ N) except i}  
    for j in 1 .. N except i {  
        while (number[j] != 0 && number[j] < number[i]);  
    }  
    /*CS*/  
    number[i] = 0;  
}
```

What if we replace  $<$  with  $\leq$ ?

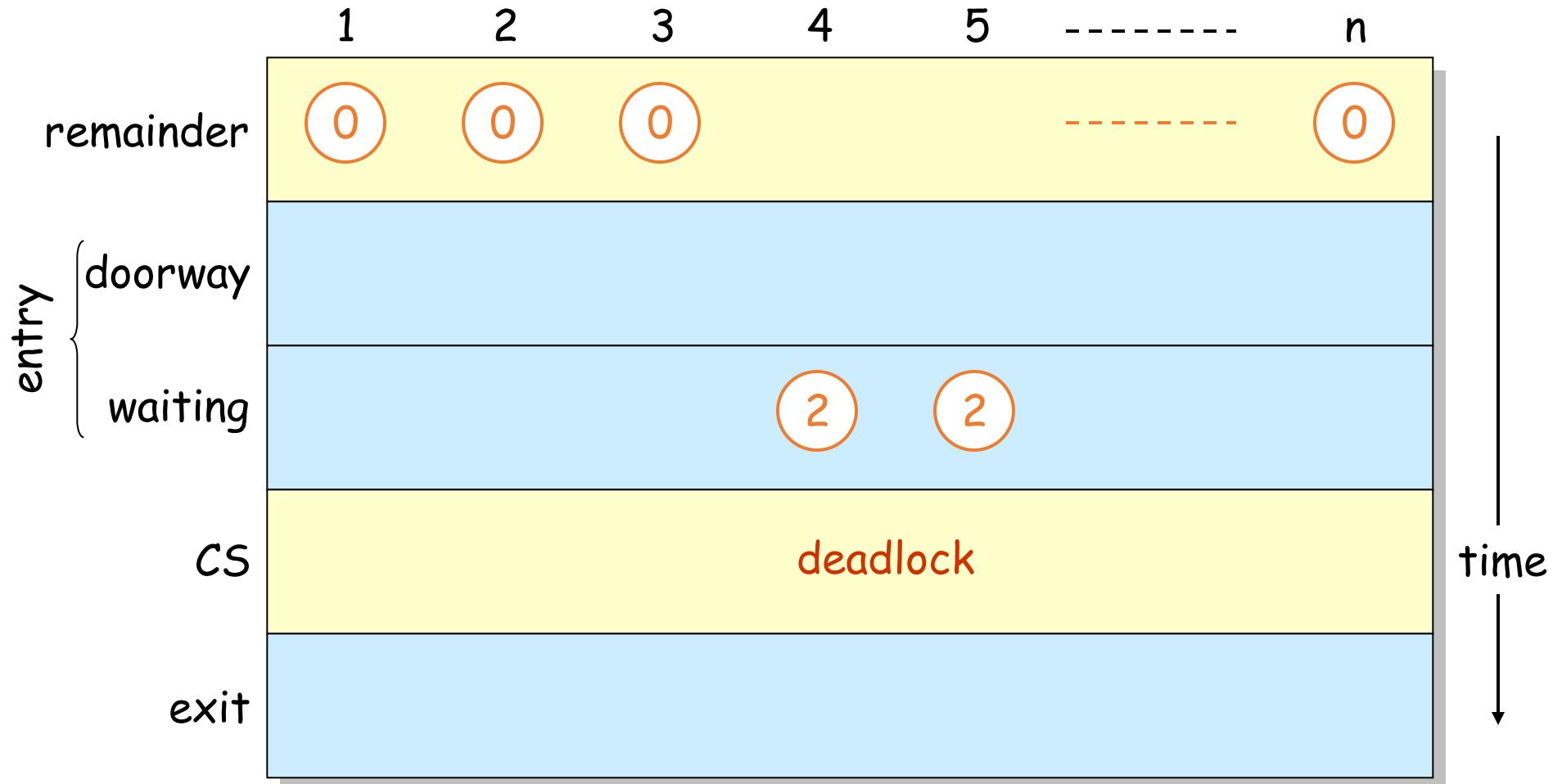
|        |   |   |   |   |       |   |         |
|--------|---|---|---|---|-------|---|---------|
|        | 1 | 2 | 3 | 4 | ----- | n |         |
| number | 0 | 0 | 0 | 0 | 0     | 0 | integer |

Answer: No! can deadlock → i due processi con numero maggiore  
potrebbero bloccarsi

## Implementation 1: deadlock



## Implementation 1: deadlock



## Implementation 2

NO DEADLOCK

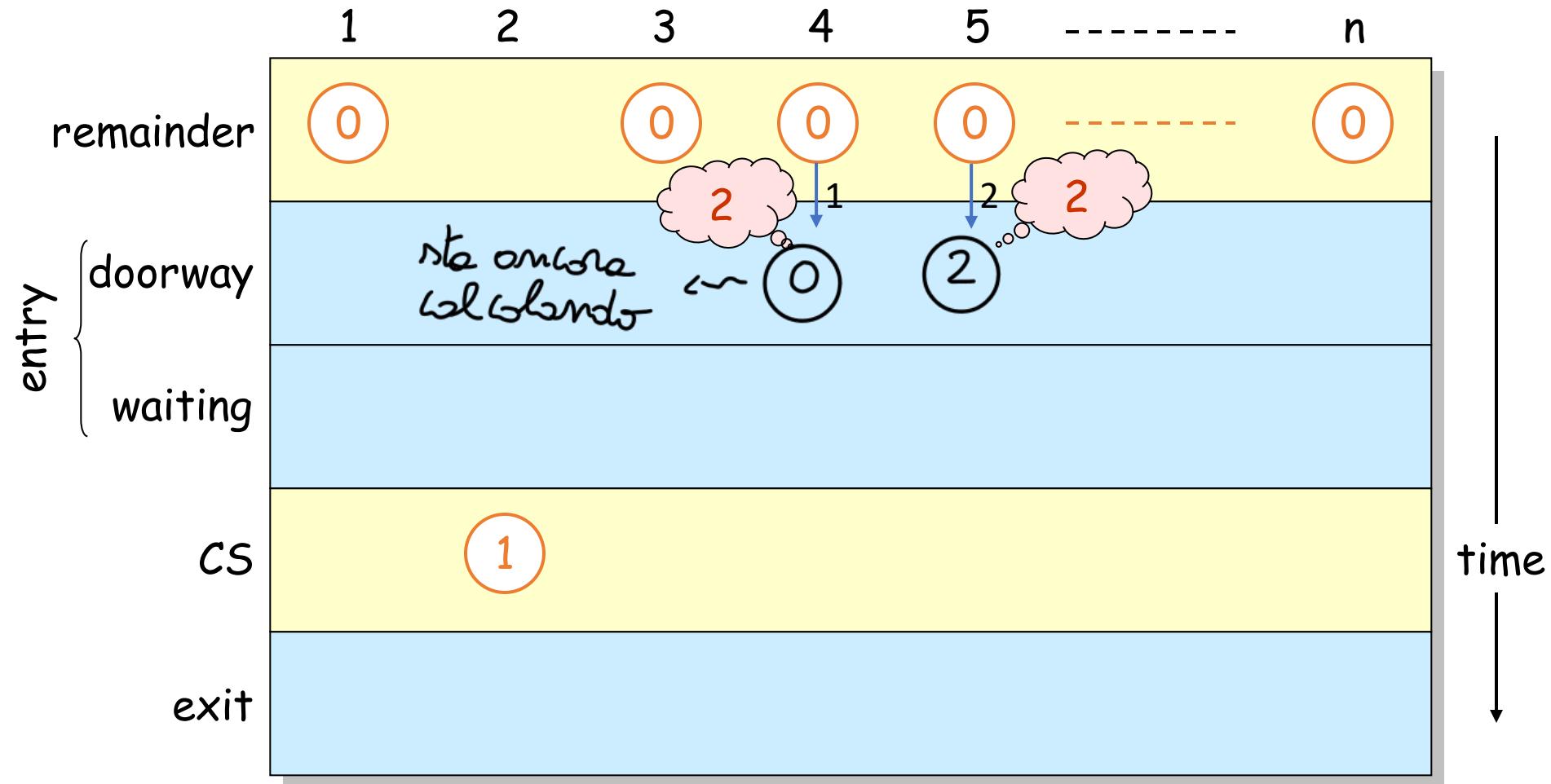
code of process  $i$ ,  $i \in \{1, \dots, n\}$

```
while (1){  
    /*NCS*/  
    number[i] = 1 + max {number[j] | (1 ≤ j ≤ N) except i}  
    for j in 1 .. N except i {  
        while (number[j] != 0 && (number[j],j) < (number[i],i));  
    }  
    /*CS*/  
    number[i] = 0;  
}  
  
// lexicographical order: (B,j) < (A,i) means (B < A || (B == A && j < i))
```

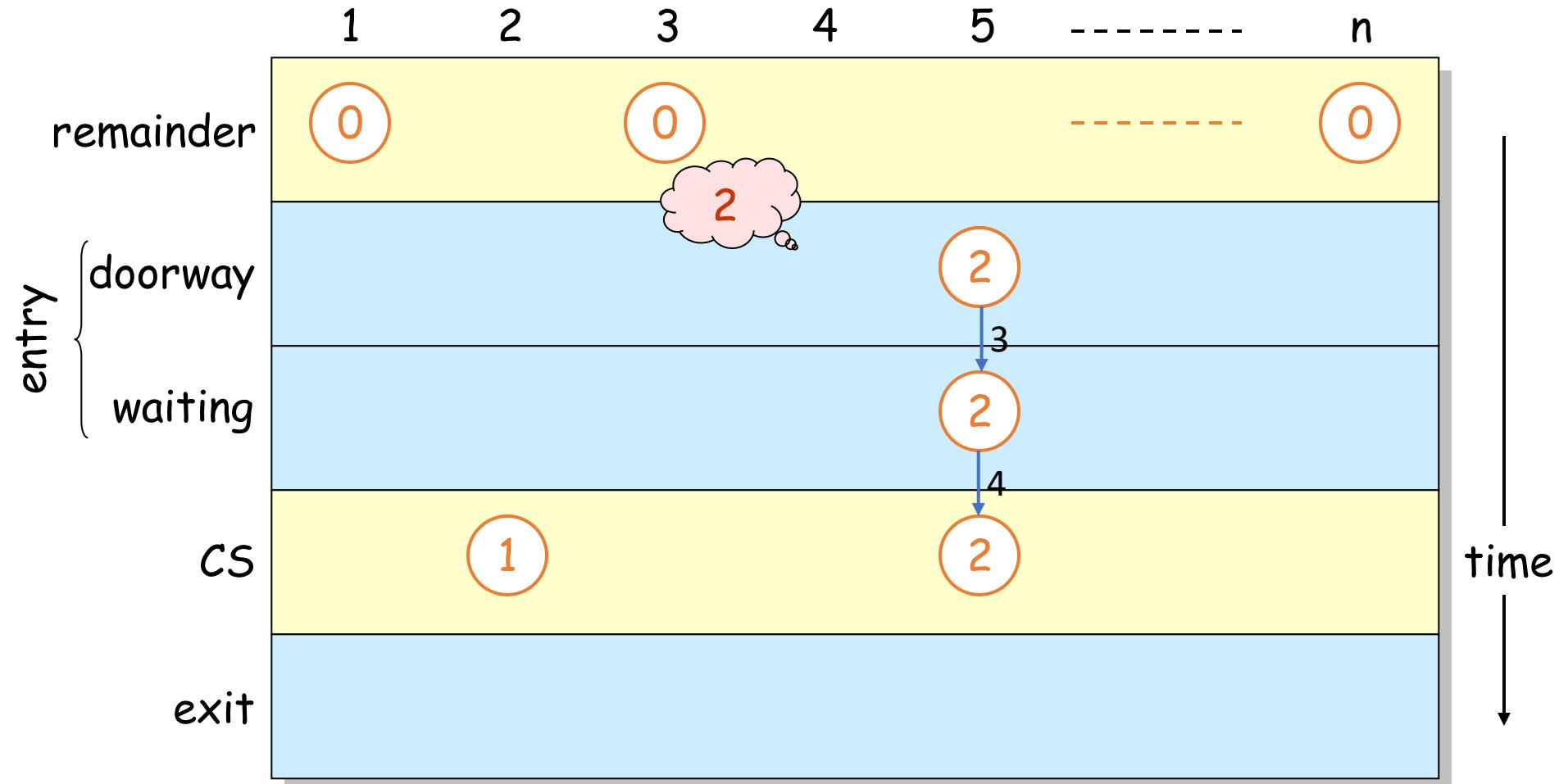
|        |   |   |   |   |       |   |         |
|--------|---|---|---|---|-------|---|---------|
|        | 1 | 2 | 3 | 4 | ----- | n |         |
| number | 0 | 0 | 0 | 0 | 0     | 0 | integer |

Answer: does not satisfy mutual exclusion

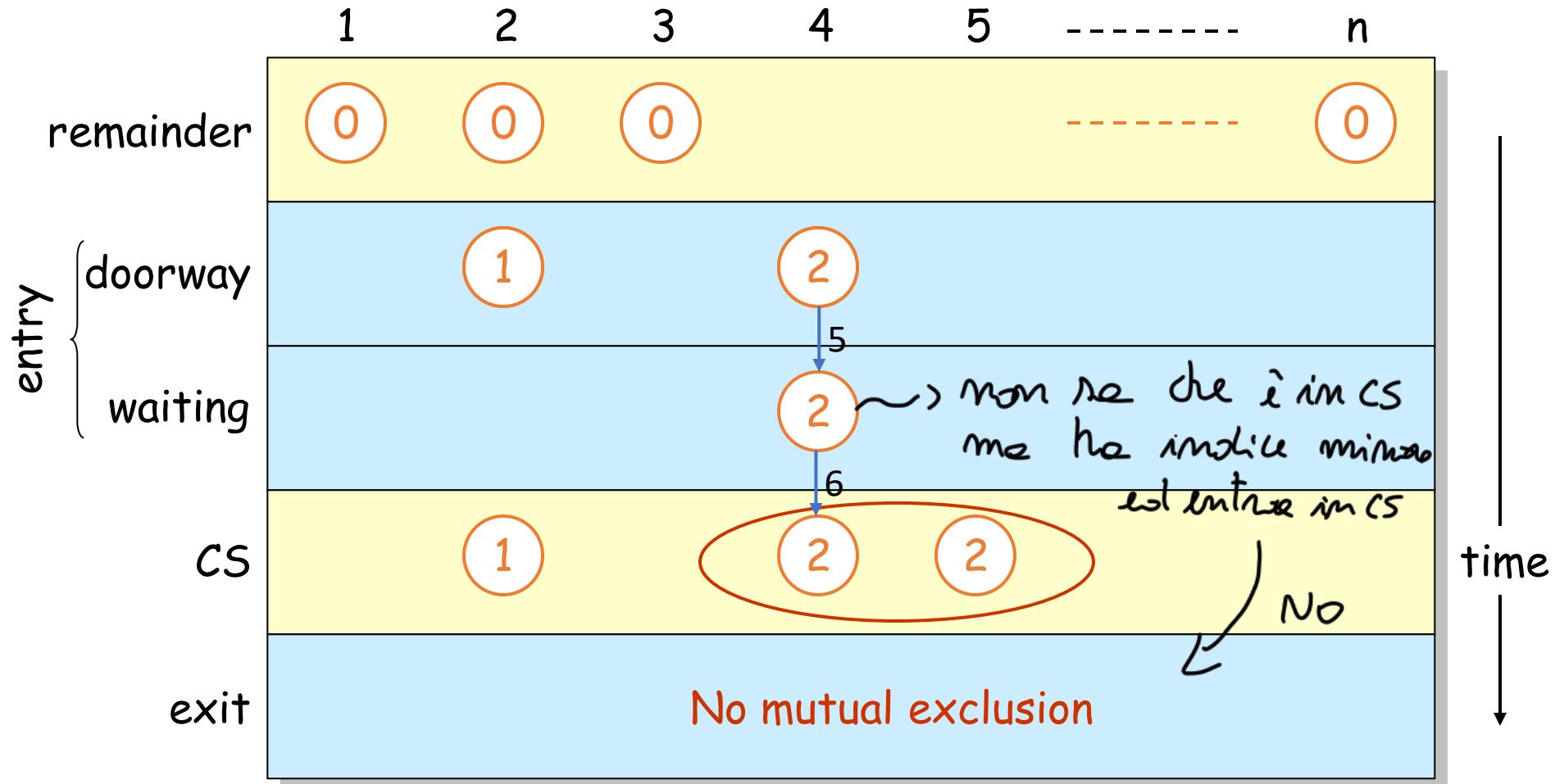
## Implementation 2: no mutual exclusion



## Implementation 2: no mutual exclusion



## Implementation 2: no mutual exclusion



# The Bakery Algorithm

code of process  $i$ ,  $i \in \{1, \dots, n\}$

```
while (1){  
    /*NCS*/  
    choosing[i] = true;  
    number[i] = 1 + max {number[j] | (1 ≤ j ≤ N) except i}  
    choosing[i] = false;  
  
    for j in 1 .. N except i {  
        while (choosing[j] == true);  
        while (number[j] != 0 && (number[j],j) < (number[i],i));  
    }  
    /*CS*/  
    number[i] = 0;  
}
```

|          | 1     | 2     | 3     | 4     | ----- | n     |         |
|----------|-------|-------|-------|-------|-------|-------|---------|
| choosing | false | false | false | false | false | false | bits    |
| number   | 0     | 0     | 0     | 0     | 0     | 0     | integer |

# Computing the Maximum *Calcolo del massimo*

code of process  $i$ ,  $i \in \{1, \dots, n\}$

Correct implementation

```
number[i] = 1 + max {number[j] | (1 ≤ j ≤ N)}
```

```
local1 = 0;  
for local2 in 1 .. N {  
    local3 = number[local2];  
    if (local1 < local3)  
        local1 = local3;  
}  
number[i] = 1 + local1
```

→ Usiamo solo  
variabili locali

|        | 1 | 2 | 3 | 4 | ----- | n |         |
|--------|---|---|---|---|-------|---|---------|
| number | 0 | 0 | 0 | 0 | 0     | 0 | integer |

# The Bakery Algorithm with bounded numbers

code of process  $i$ ,  $i \in \{1, \dots, n\}$

```
while (1){           ↗ re raggiunto MAXIMUM  
    /*NCS*/          richiesto il valore, finora  
    while(number[i] == 0){ che MAXIMUM ha fatto,  
        choosing[i] = true; tutti aspettano.  
        number[i] = (1 + max {number[j] | (1 ≤ j ≤ N) except i}) % MAXIMUM  
        choosing[i] = false;  
    }  
    for j in 1 .. N except i {  
        while (choosing[j] == true);  
        while (number[j] != 0 && (number[j],j) < (number[i],i));  
    }  
    /*CS*/  
    number[i] = 0;  
}
```

numbers more  
no represents  
bile.

↑

# Bakery algorithm characteristics

- Processes communicate by writing/reading shared variables (as Dijkstra)
- Read/write are not atomic operations
  - Reader can read while writer is writing
  - None receives any notification
- Any shared variable is owned by a process that can write it, others can read it
- No process can perform two concurrent writings
- Execution times are not correlated

# The Bakery Algorithm in client/server app.

code of process  $i$ ,  $i \in \{1, \dots, n\}$

```
while (1){ //client thread
    /*NCS*/
    choosing = true; //doorway
    for j in 1 .. N except i {
        send(Pj, num);
        receive(Pj, v);
        num = max(num, v);
    }
    num = num+1;
    choosing = false;
    for j in 1 .. N except i { //bakery
        do{
            send(Pj, choosing);
            receive(Pj, v);
        }while (v == true);
        do{
            send(Pj, v);
            receive(Pj, v);
        }while (v != 0 && (v,j) < (num,i));
    }
    /*CS*/
    num = 0;
}
```

```
//global variable
//inizialization:
int num = 0;
boolean choosing = false;
// and process ip/ports
```

```
while (1){ //server thread
    receive(Pj, message);
    if (message is a number)
        send(Pj, num);
    else
        send(Pj, choosing);
}
```

Assumptions:

- Finite response time
- Reliable communication channels

# Network and Sockets

Slides are mainly taken from «*Operating Systems: Internals and Design Principles*», 7/E William Stallings (Chapter 17).

*Sistemi di Calcolo 2*

*Instructor: Riccardo Lazzeretti*

*Special thanks to: Daniele Cono D'Elia, Leonardo Aniello,  
Roberto Baldoni*

# **RICHIAMI DI RETE**

# Need for a Protocol Architecture

- The procedures involved in exchanging data between devices can be complex
- There must be a data path between the two computers, either directly or via a communication network
- Typical tasks performed are:
  - source system must either activate the direct data communication path or inform the communication network of the identity of the desired destination system
  - source system must ascertain that the destination system is prepared to receive data
  - file transfer application on the source system must ascertain that the file management program on the destination system is prepared to accept and store the file for this particular user
  - if the file formats or data representations used on the two systems are incompatible, one or the other system must perform a format translation function

# Computer Communications

- The exchange of information between computers for the purpose of cooperative action is

**computer  
communications**

- When two or more computers are interconnected via a communication network, the set of computer stations is referred to as a

**computer  
network**

- In discussing computer communications and computer networks, two concepts are paramount:

**1) protocols  
2) computer  
communications  
architecture  
(or  
protocol architecture)**

# Protocol

- A set of rules governing the exchange of data between two entities
  - used for communication between entities in different systems
  - Entity: anything capable of sending or receiving information
  - System: physically distinct object that contains one or more entities
  - To communicate successfully, two entities must “speak the same language”
- Key elements of a protocol are:

**Syntax**

- includes such things as data format and signal levels

**Semantics**

- includes control information for coordination and error handling

**Timing**

- includes speed matching and sequencing

# Protocol Architecture

There must be a high degree of cooperation between the two computer systems

Rather than implementing everything as a single module, tasks are broken up into subtasks

## Example

The file transfer module contains all the logic that is unique to the file transfer application

The communications service module has to assure that the two computer systems are active and ready for data transfer, and for keeping track of the data that are being exchanged to assure delivery

The logic for actually dealing with the network is put into a separate network access module

The structured set of modules that implements the communications function is referred to as a ***protocol architecture***

# File Transfer Architecture

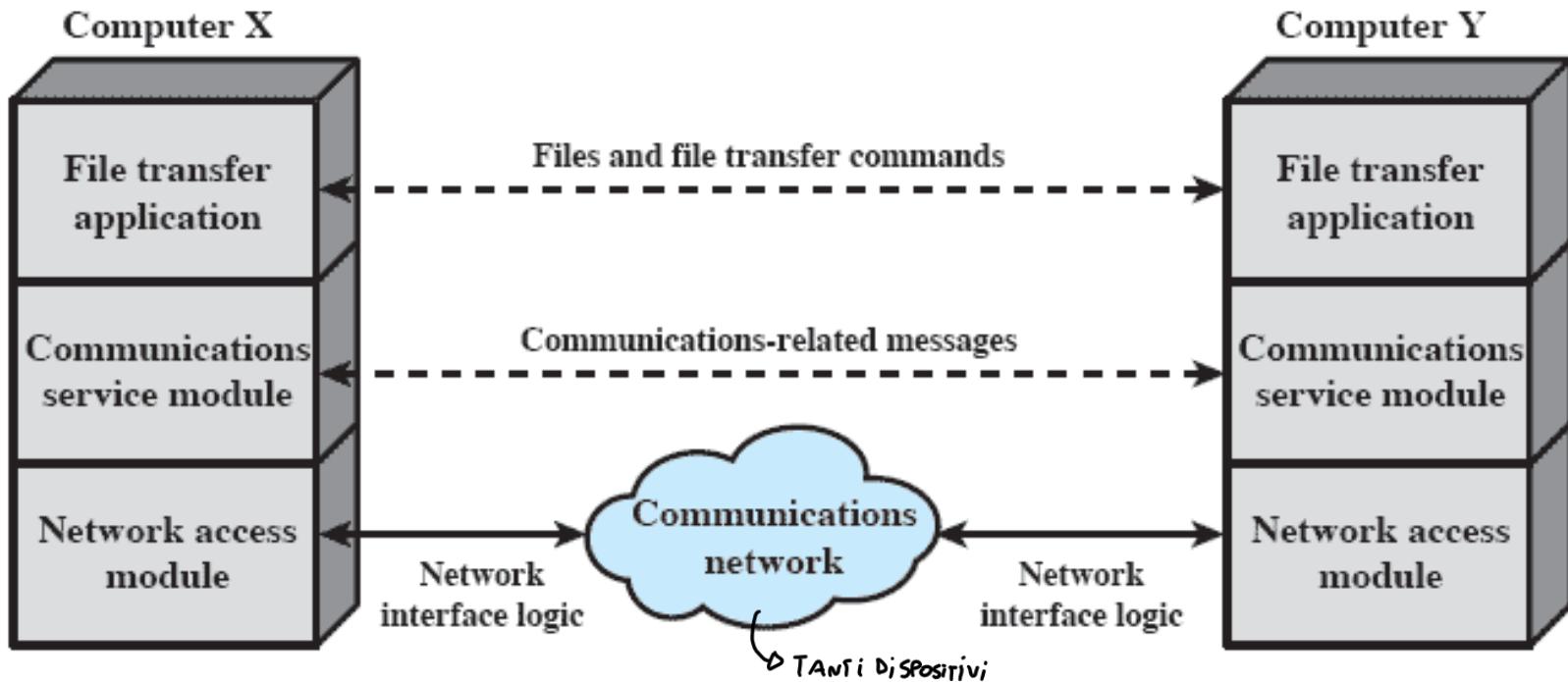


Figure 17.1 A Simplified Architecture for File Transfer

# TCP/IP Protocol Architecture

- A result of protocol research and development conducted on the experimental packet-switched network, ARPANET
- Referred to as the TCP/IP protocol suite
- The communication task for TCP/IP can be organized into five independent layers:
  - application layer
  - host-to-host, or transport layer
  - internet layer
  - network access layer
  - physical layer

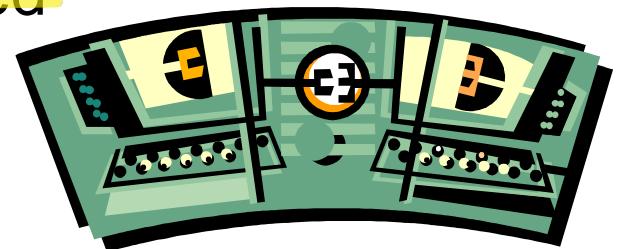
# Physical Layer and Network Access Layer

## Physical layer

- covers the physical interface between a data transmission device and a transmission medium or network
- this layer is concerned with:
  - specifying the characteristics of the transmission medium
  - the nature of the signals
  - the data rate

## Network access layer

- concerned with the exchange of data between an end system and the network to which it is attached
- the specific software used at this layer depends on the type of network to be used



# Internet Layer and Transport Layer

## Internet layer

- function is to allow data to traverse multiple interconnected networks
- the *Internet Protocol (IP)* is used at this layer to provide the routing function across multiple networks
  - implemented not only in the end systems but also in routers
    - » a *router* is a processor that connects two networks
    - » primary function is to relay data from one network to the other



## Transport layer

- a common layer shared by all applications and contains the mechanisms for providing reliability when data is exchanged
- the *transmission control protocol (TCP)* is the most commonly used protocol to provide this function

# Application Layer

- contains the logic needed to support the various user applications
- for each different type of application, a separate module is needed that is peculiar to that application



# TCP/IP Concepts

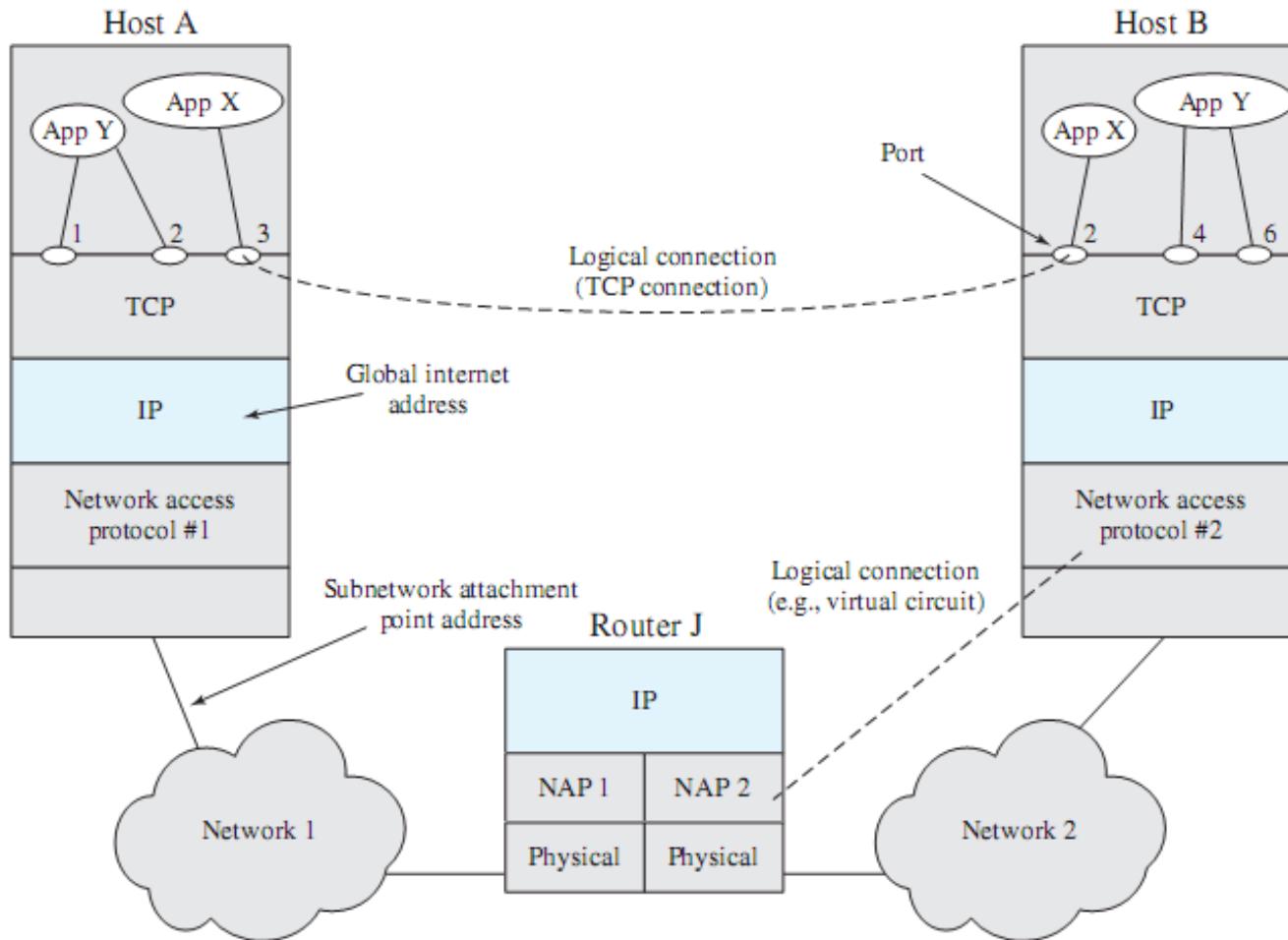
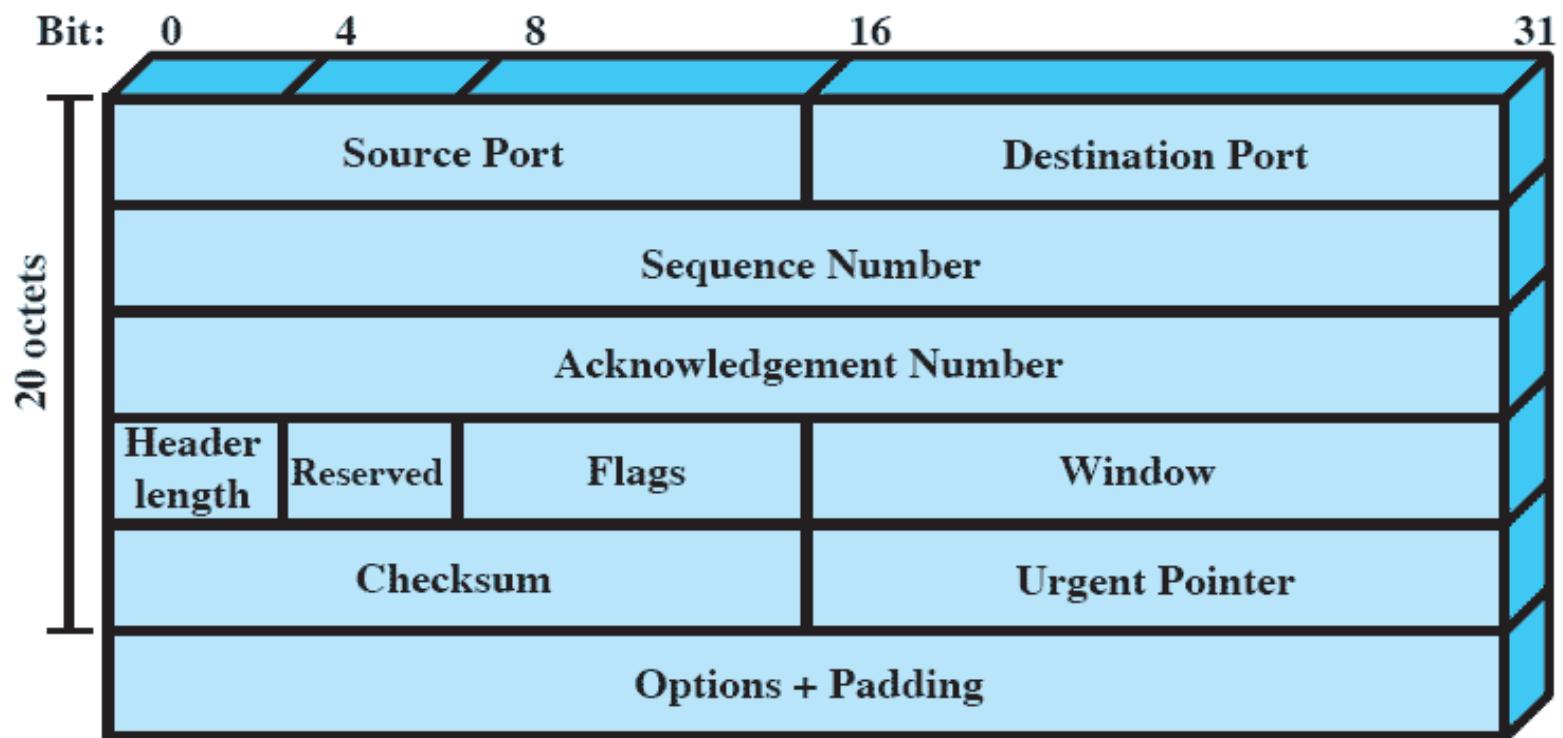


Figure 17.4 TCP/IP Concepts

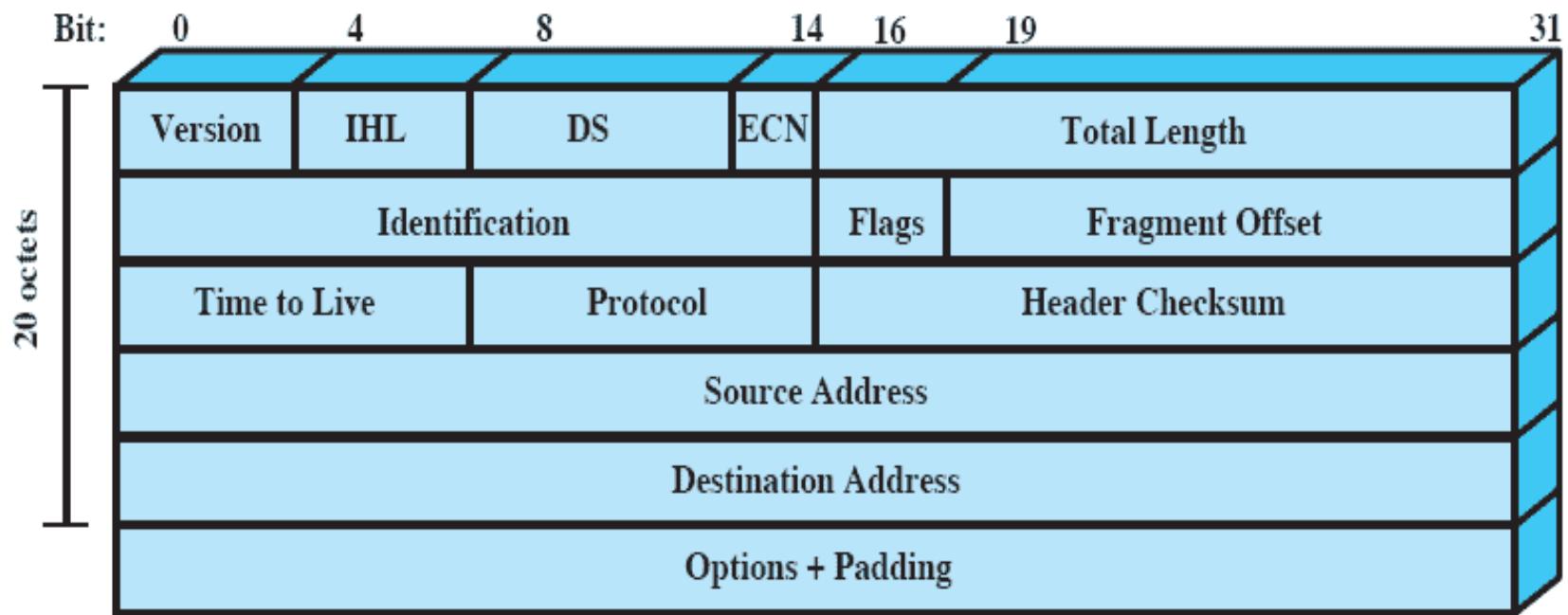
# Operation of TCP/IP

- Every entity in the overall system must have a unique address
    - two levels of addressing are needed
1. Each host on a network must have a unique global internet address
    - » this address is used by IP for routing and delivery
  2. Each application within a host must have an address that is unique within the host
    - » this allows the host-to-host protocol (TCP) to deliver data to the proper process
    - » these addresses are known as *ports*

# TCP header



# IP header (IPv4)



# TCP/IP Applications

- A number of applications have been standardized to operate on top of TCP
- Examples:
  - **Simple Mail Transfer Protocol (SMTP)**
    - provides a basic electronic mail facility
    - features include mailing lists, return receipts, and forwarding
  - **File Transfer Protocol (FTP)**
    - used to send files from one system to another under user command
    - both text and binary files are accommodated
    - provides features for controlling user access
  - **Secure Shell (SSH)**
    - provides a secure remote logon capability, which enables a user at a terminal or personal computer to log on to a remote computer and function as if directly connected to that computer
    - supports file transfer between the local host and a remote server
    - SSH traffic is carried on a TCP connection



# UDP



- minimum protocol mechanism
  - **connectionless**
  - no guarantees about delivery, preservation of sequence, nor protection against duplication
  - useful, e.g., for transaction-oriented applications
  - multicast support

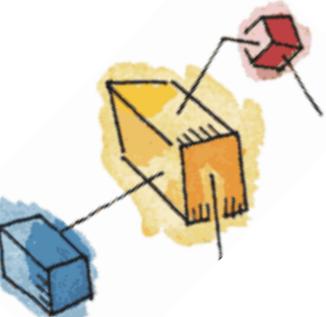
# SOCKETS

→ ciò che identifica il mio dispositivo

→ Protocollo creato per poter utilizzare la comunicazione tra due dispositivi

# Sockets

- Concept was developed in the 1980s in the UNIX environment as the Berkeley Sockets Interface (BSI)
- Enables communication between a client and server process
- May be either connection oriented or connectionless
- Can be considered an endpoint in communication
- The BSI transport layer interface is the de-facto standard API for developing networking applications
- Windows sockets (WinSock) are based on the Berkeley specification



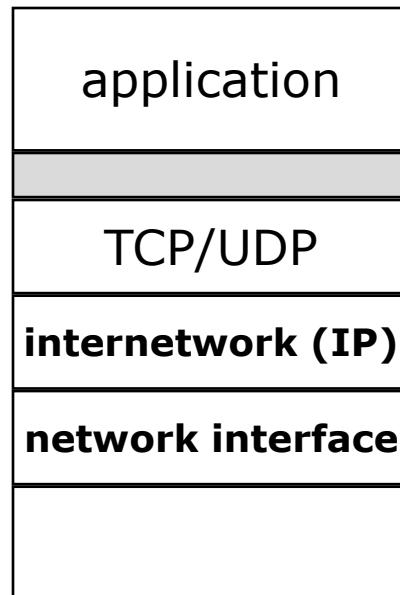
# Sockets

Application details

User process

Kernel

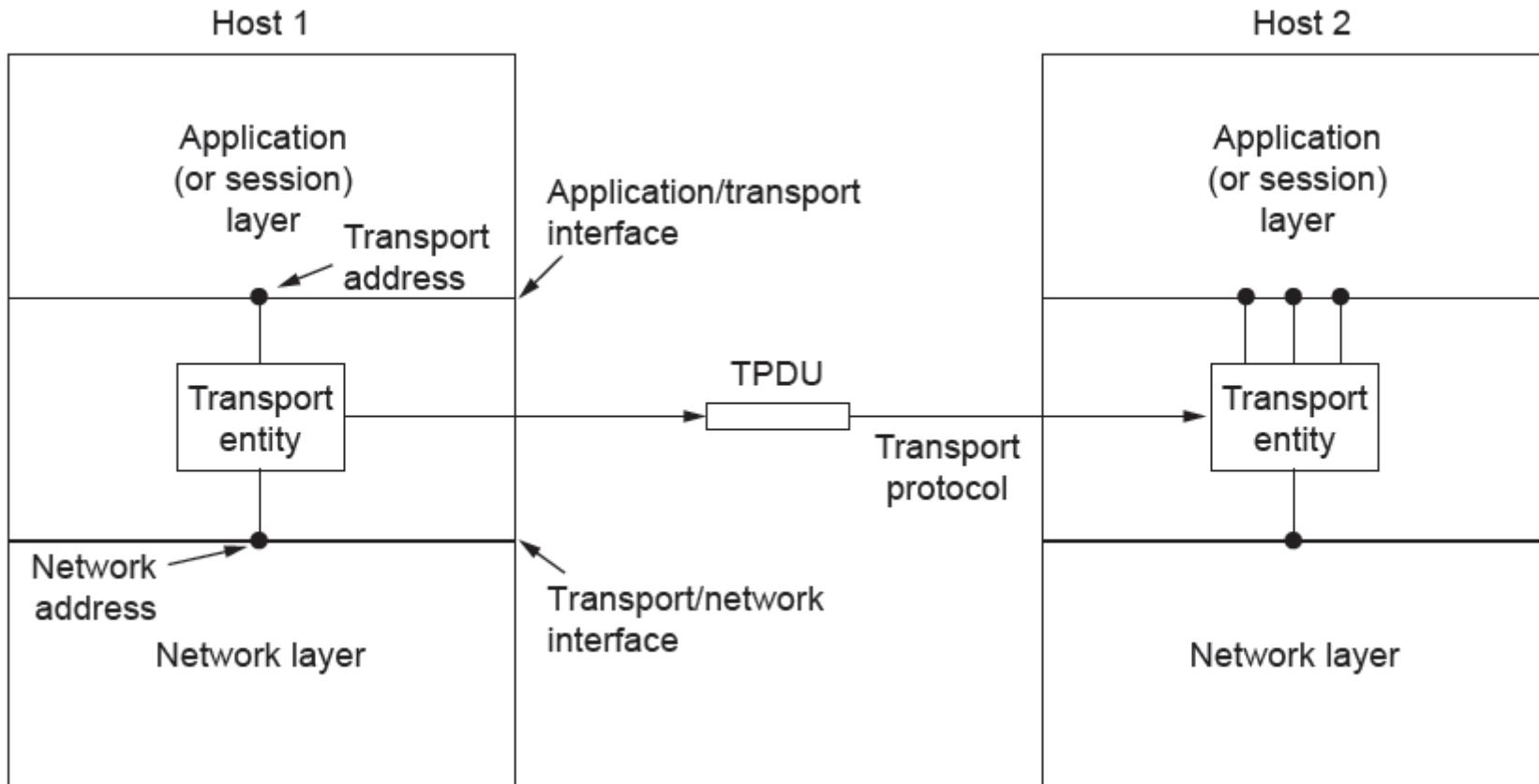
Communication details



**Interfaccia socket**  
**TLI (Transport Layer Interface)**



# Services Provided to the Upper Layers



The network, transport, and application layers

# The Socket

- ***IP addresses*** identify the respective host systems
- The concatenation of a port value and an IP address forms a ***socket***, which is unique throughout the Internet

## Stream sockets

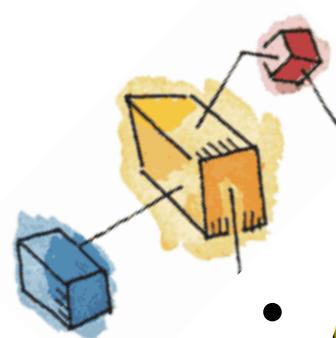
- makes use of TCP
- provides a connection-oriented reliable data transfer

## Datagram sockets

- make use of UDP
- delivery is not guaranteed, nor is order necessarily preserved

## Raw sockets

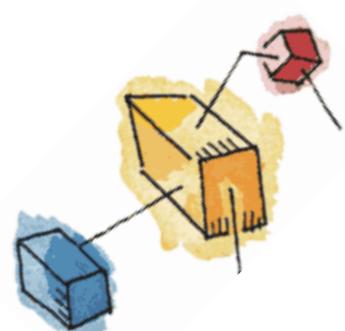
- allow direct access to lower-layer protocols



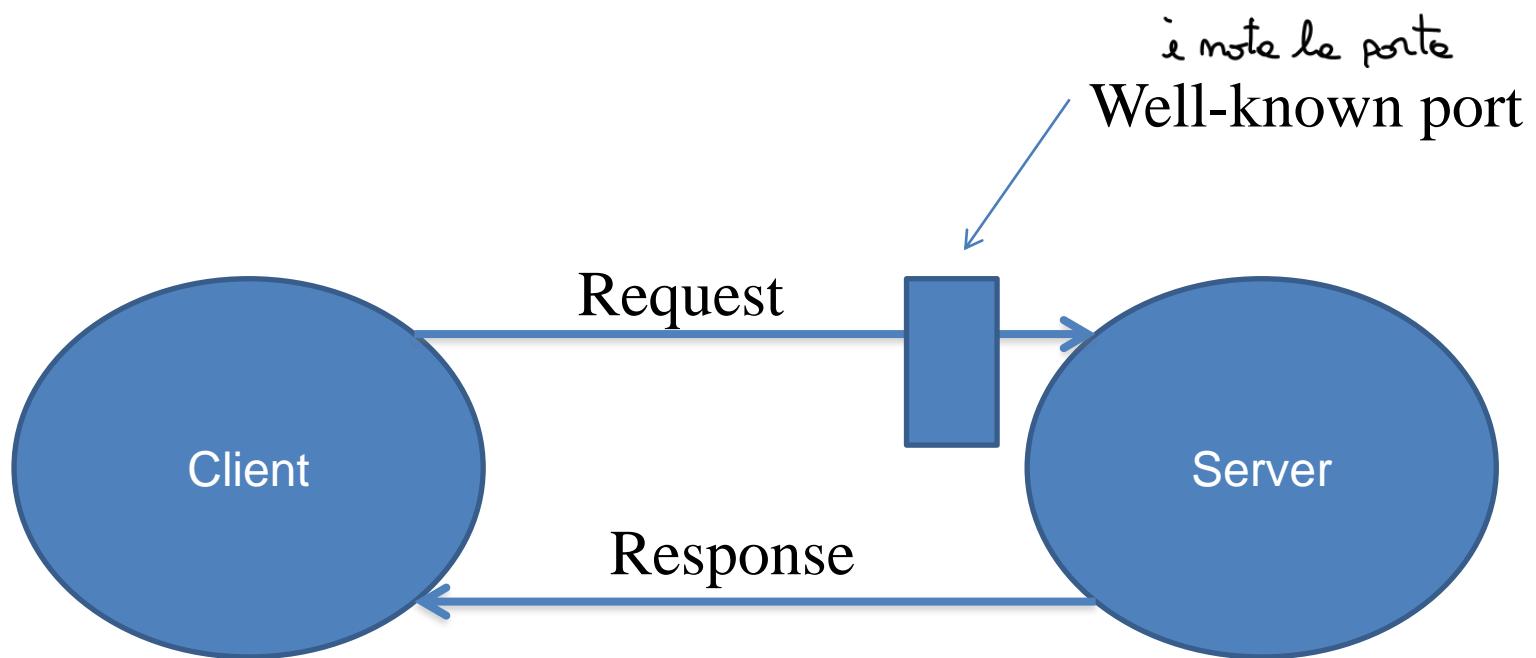
# Socket Basics

- An end-point for a IP network connection
  - what the application layer “plugs into”
  - programmer cares about API
- End point determined by two things:
  - Host address: IP address is *Network Layer*
  - Port number: is *Transport Layer*
- Two end-points determine a connection: socket pair
  - ex: 206.62.226.35,p21 + 198.69.10.2,p1500
  - ex: 206.62.226.35,p21 + 198.69.10.2,p1499

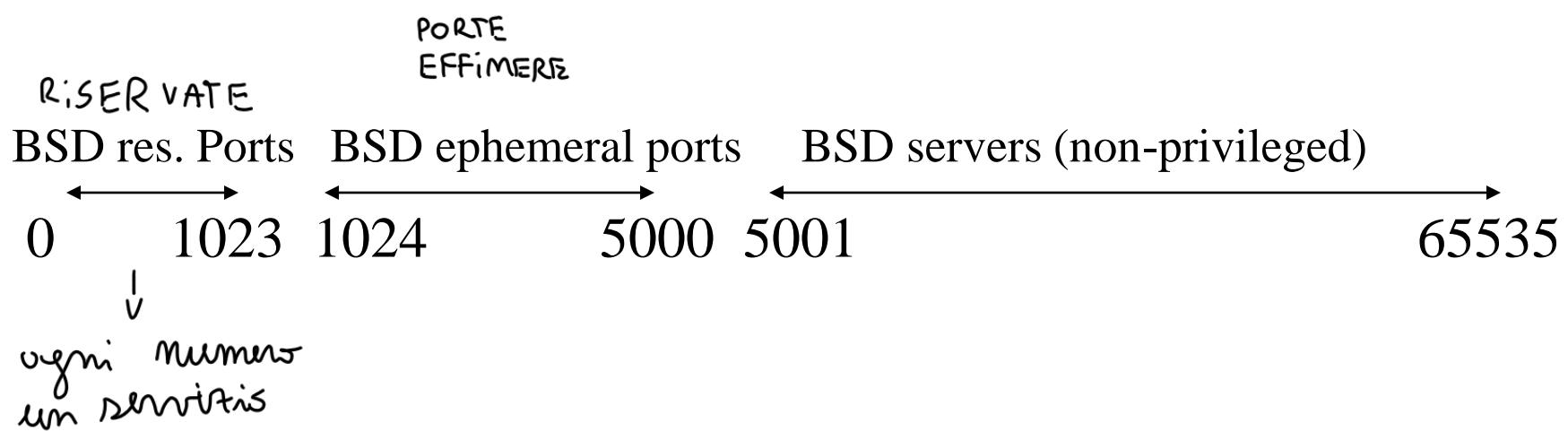




# Client-Server Approach

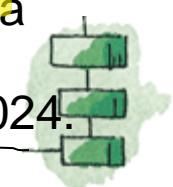


# Ports

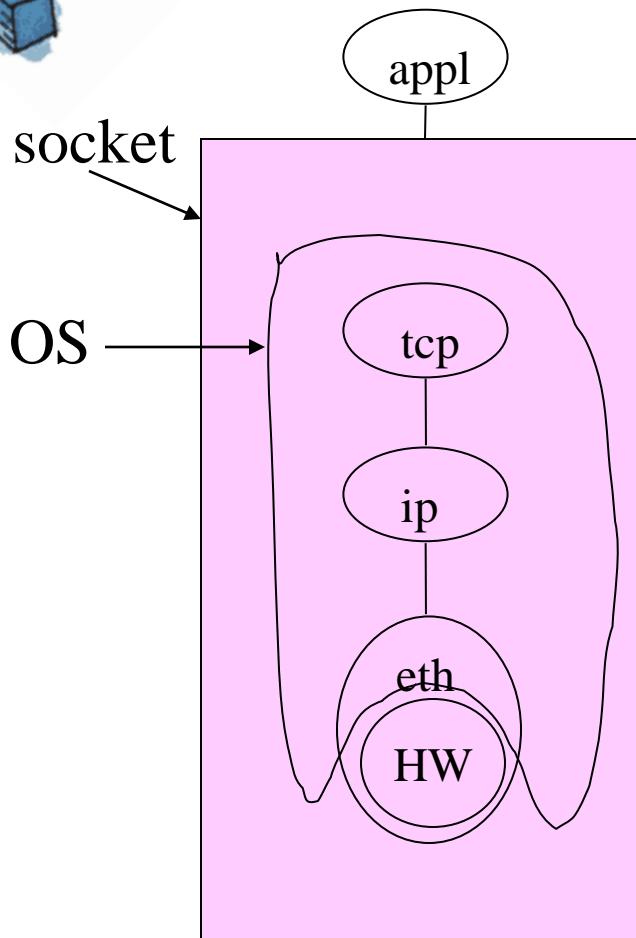




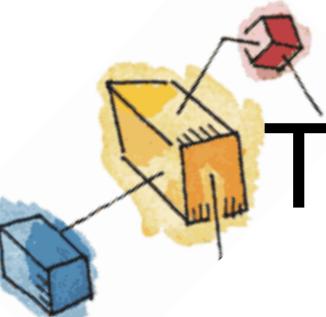
# Ports

- Numbers (vary in BSD, Solaris, Linux):
    - 0-1023 “reserved”, must be root
    - 1024 - 5000 “ephemeral” (short-lived ports assigned automatically by the OS to clients)
    - however, many systems allow > 3977 ephemeral ports due to the number of increasing handled by a single PC. (Sun Solaris provides 30,000 in the last portion of BSD non-privileged)
  - Well-known, reserved services /etc/services:
    - ftp 21/tcp
    - telnet 23/tcp
    - finger 79/tcp
    - snmp 161/udp
  - Several client program needs to be a server at the same time (rlogin, rsh) as a part of the client-server authentication. These clients call the library function `rresvport` to create a socket and to assign an unused port in the range 512-1024.
- 
- 

# Sockets and the OS



- User sees  
“descriptor”, integer  
index
  - like: FILE \*, or file  
index from open ()
  - returned by  
socket () call (more  
later)

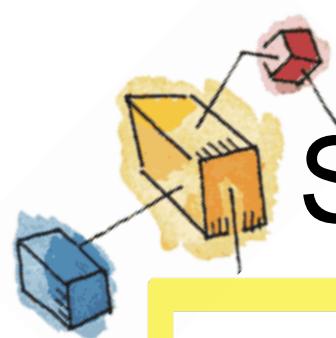


# Transport Service Primitives

| Primitive  | Packet sent        | Meaning                                    |
|------------|--------------------|--|
| LISTEN     | (none)             | Block until some process tries to connect  |
| CONNECT    | CONNECTION REQ.    | Actively attempt to establish a connection |
| SEND       | DATA               | Send information                           |
| RECEIVE    | (none)             | Block until a DATA packet arrives          |
| DISCONNECT | DISCONNECTION REQ. | This side wants to release the connection  |



Primitives for a simple transport service

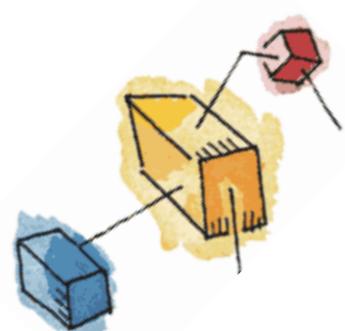


# Socket Address Structure

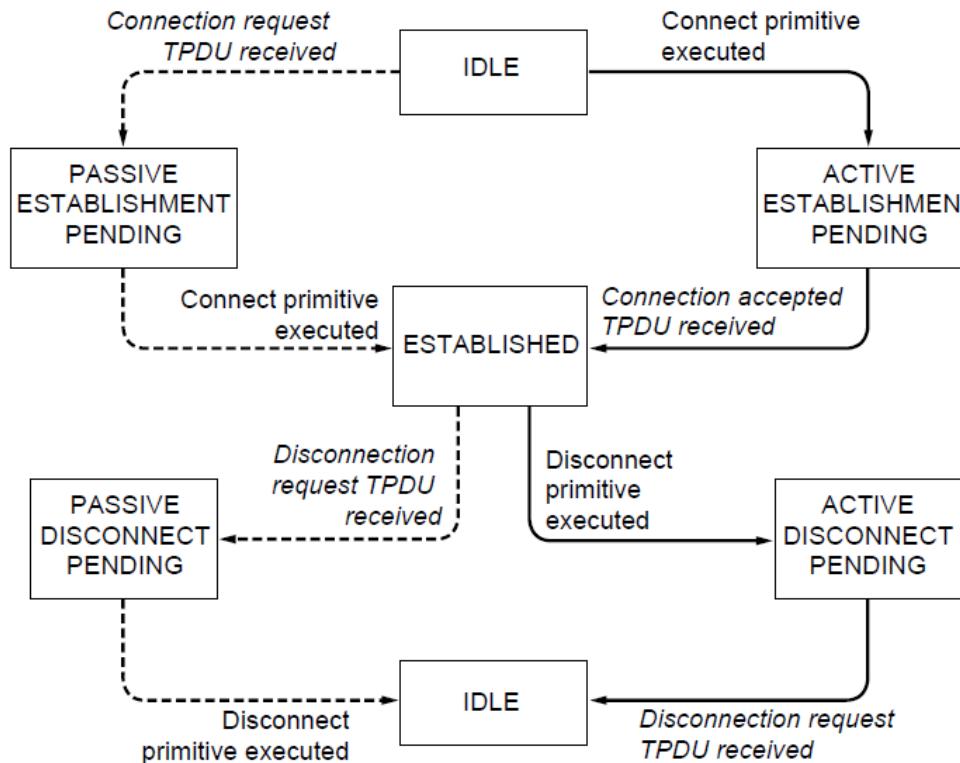
```
struct in_addr {  
    in_addr_t s_addr;             /* 32-bit IPv4 addresses */  
};  
  
struct sockaddr_in {  
    uint8_t      sin_len;        /* length of structure */  
    sa_family_t  sin_family;     /* AF_INET */  
    in_port_t    sin_port;       /* TCP/UDP Port num */  
    struct in_addr sin_addr;     /* IPv4 address (above) */  
    char sin_zero[8];           /* unused */  
}
```

- **Zero-initialize (e.g., bzero, **memset**) before using sockaddr**

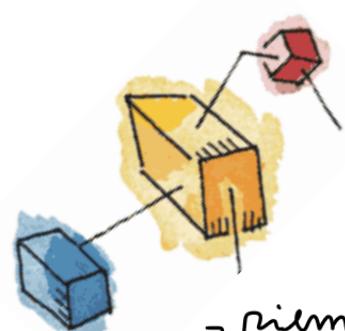




# Berkeley Sockets (1)



A state diagram for a simple connection management scheme. Transitions labeled in italic are caused by packet arrivals. The solid lines show the client's state sequence. The dashed lines show the server's state sequence.



# Berkeley Sockets (2)

→ riempire qui l'mpi di prima

| Primitive | Meaning   |
|-----------|---|
| SOCKET    | Create a new communication end point                        |
| BIND      | Associate a local address with a socket                     |
| LISTEN    | Announce willingness to accept connections; give queue size |
| ACCEPT    | Passively establish an incoming connection                  |
| CONNECT   | Actively attempt to establish a connection                  |
| SEND      | Send some data over the connection                          |
| RECEIVE   | Receive some data from the connection                       |
| CLOSE     | Release the connection                                      |



The socket primitives for TCP

# Socket Connection

Client:

- For a stream socket, once the socket is created, a connection must be set up to a remote socket
- One side functions as a client and requests a connection to the other side, which acts as a server

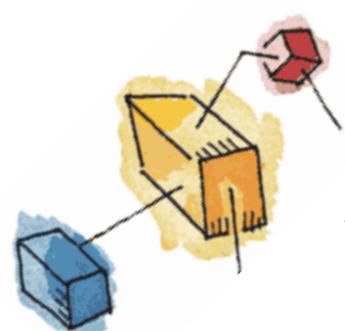
issues a connect()  
that specifies both a  
local socket and the  
address of a remote  
socket

once a connection is  
set up, getpeername()  
can be used to find  
out who is on the  
other end of the  
connected stream  
socket

Server side of a  
connection setup  
requires two steps:

a server application  
issues a listen (),  
indicating that the  
given socket is ready  
to accept incoming  
connections

each incoming  
connection is placed  
in this queue until a  
matching accept () is  
issued by the server  
side



# Addresses and Sockets

- Structure to hold address information
- Functions pass address from user to OS
  - bind ()
  - connect () (*TCP only*)
  - sendto () (*UDP only*)
- Functions pass address from OS to user
  - accept () (*TCP only*)
  - recvfrom () (*UDP only*)

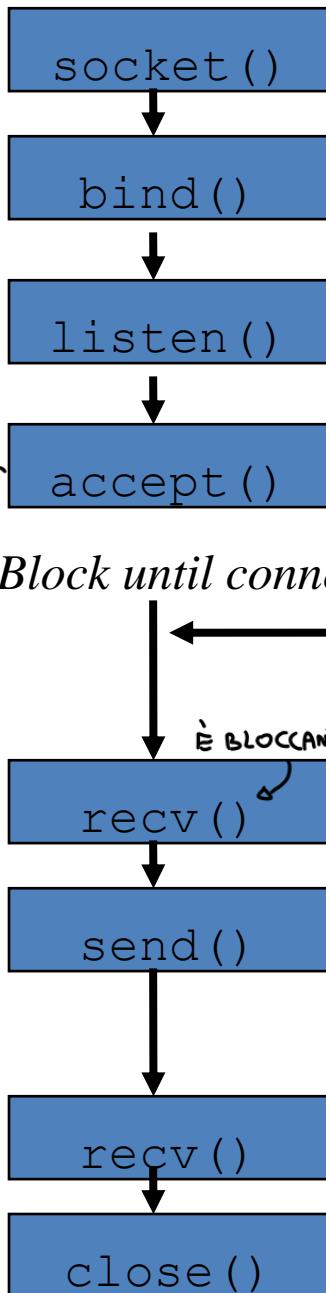


# TCP Client-Server

**Server**

si mette in ascolto.

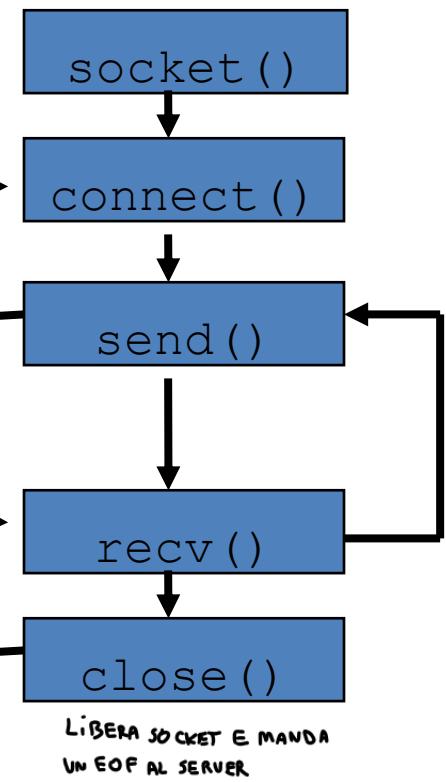
PUÒ ACCETTARE  
PIÙ CLIENT SU  
QUESTA PORTA



“well-known”  
port

TORNA UN’ALTRA PORTA  
≠ DA PORTA D’ASCOLTO

**Client**



“Handshake”

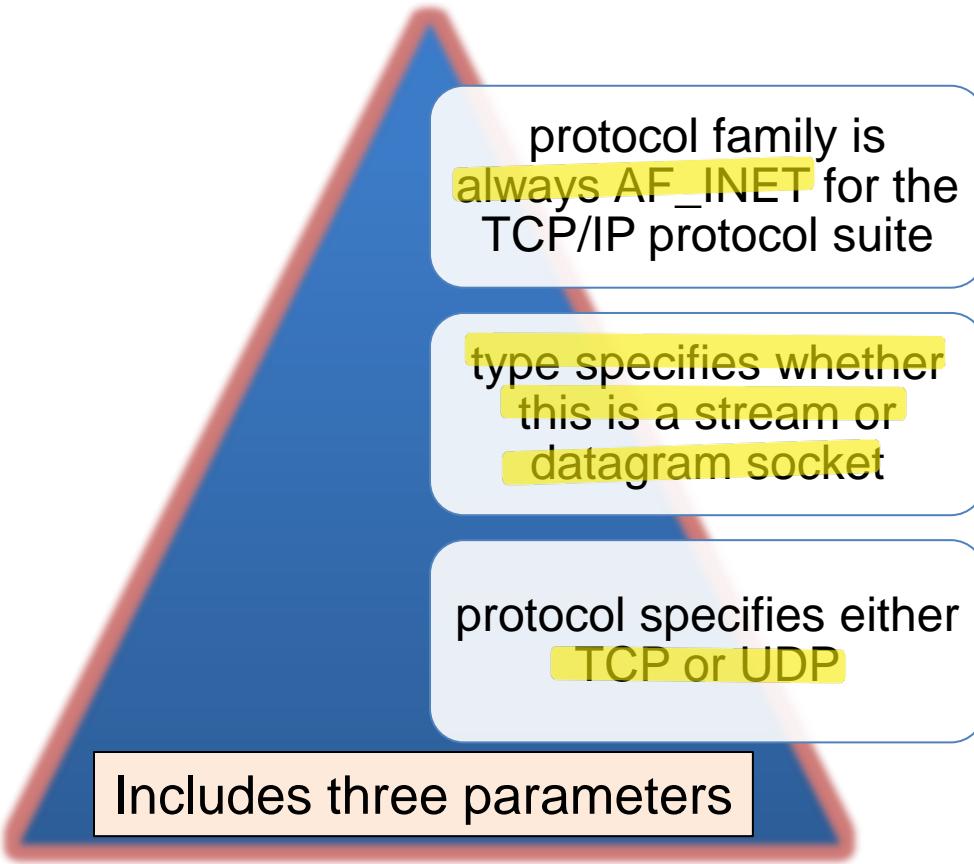
Data (request)

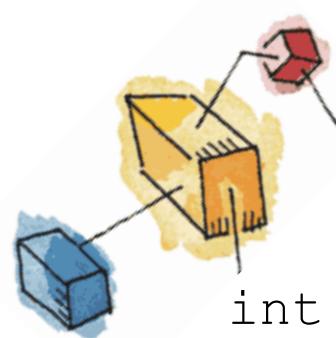
Data (reply)

End-of-File

LIBERA SOCKET E MANDA  
UN EOF AL SERVER

# Socket Setup: Socket()

- The **first** step in using Sockets is to **create a new socket using the socket() API**
  - **Returns an integer that identifies the socket**
    - similar to a UNIX file descriptor
- 
- protocol family is always AF\_INET for the TCP/IP protocol suite
- type specifies whether this is a stream or datagram socket
- protocol specifies either TCP or UDP
- Includes three parameters

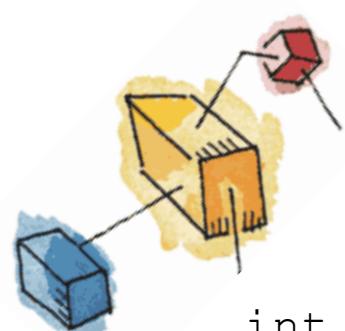


# socket ()

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

Create a socket, giving access to transport layer service

- *family* is one of
    - AF\_INET (IPv4), AF\_INET6 (IPv6), AF\_LOCAL (local Unix),
    - AF\_ROUTE (access to routing tables), AF\_KEY (new, for encryption)
  - *type* is one of
    - SOCK\_STREAM (TCP), SOCK\_DGRAM (UDP)
    - SOCK\_RAW (for special IP packets, PING, etc. Must be root)
  - *protocol* is 0 (used for some raw socket options)
  - upon success returns socket descriptor
    - Integer, like file descriptor
    - Return -1 if failure
- 
- 



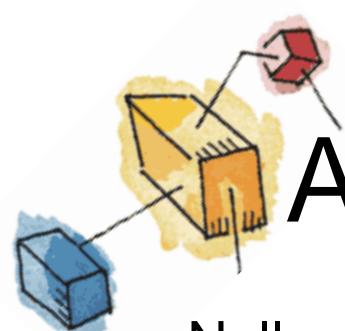
# bind()

```
int bind(int sockfd, const struct sockaddr *myaddr,  
        socklen_t addrlen);
```

Assign a local protocol address (“name”) to a socket.

- *sockfd* is socket descriptor from `socket()`
- *myaddr* is a pointer to address struct with:
  - *port number* and *IP address*
  - if port is 0, then host OS will pick *ephemeral port*
- *addrlen* is length of structure
- returns 0 if ok, -1 on error
  - EADDRINUSE (“Address already in use”)

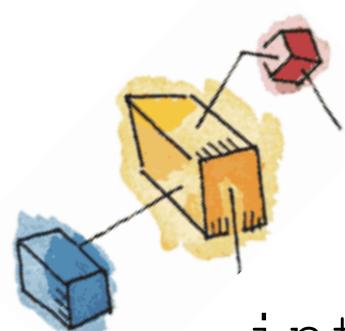




# Argomenti Valore-risultato

- Nelle chiamate che passano la struttura indirizzo da processo utente a OS (esempio bind) viene passato un puntatore alla struttura ed un intero che rappresenta il **sizeof** della struttura (oltre ovviamente ad altri parametri). In questo modo l'OS kernel sa esattamente quanti byte deve copiare nella sua memoria.
- Nelle chiamate che passano la struttura indirizzo dall'OS kernel al processo utente (esempio accept) vengono passati nella system call eseguita dall'utente un puntatore alla struttura ed un puntatore ad un intero in cui è stata inserita la **dimensione della struttura** indirizzo. In questo caso, sulla chiamata della system call, il kernel dell'OS sa la dimensione della struttura quando la va a riempire e non ne oltrepassa i limiti. Quando la system call ritorna inserisce nell'intero la dimensione di quanti byte ha scritto nella struttura.
- Questo modo di procedere è utile in system call come la select e la getsockopt che vedrete durante le esercitazioni.





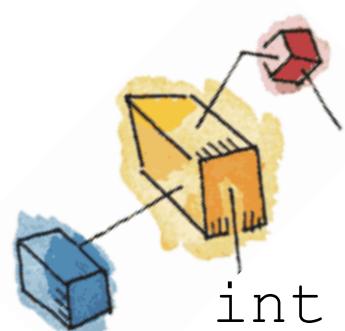
# listen()

```
int listen(int sockfd, int backlog);
```

Change socket state for TCP server.

- *sockfd* is socket descriptor from `socket()`
- *backlog* is maximum number of *incomplete connections*
  - historically 5
  - implementation might use it just as a hint





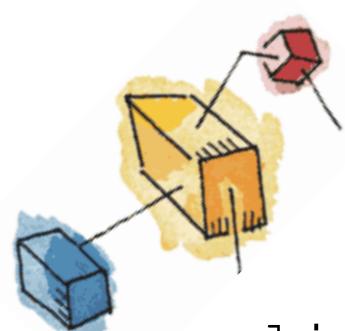
# accept ()

```
int accept(int sockfd,  
          struct sockaddr *cliaddr,  
          socklen_t *addrlen);
```

**Return next completed connection.** INDIRIZZO CLIENT

- **sockfd** is socket descriptor from `socket ()`
- **cliaddr** and **addrlen** return protocol address from client
- **returns new descriptor, created by OS**
- On error, -1 is returned, and `errno` is set appropriately.
- if used with `fork ()`, can create concurrent server.  
If used with `pthread_create ()` can create a multithread server





THREAD + EFFICIENZA  
PROCESS + AFFIDABILITÀ

# Accept() + fork()

```
lisfd = socket(....);  
bind(lisfd,....);  
listen(lisfd, 5);  
while(1) {  
    connfd = accept(lisfd,.....);  
    if (pid = fork() == 0) {  
        close(lisfd);  
        ↙ È IL FIGLIO  
        doit(connfd);  
        close(connfd);  
        _exit(0);  
    }  
    close(connfd);  
}
```

CHIUDO SUBITO: non miglior interazione.

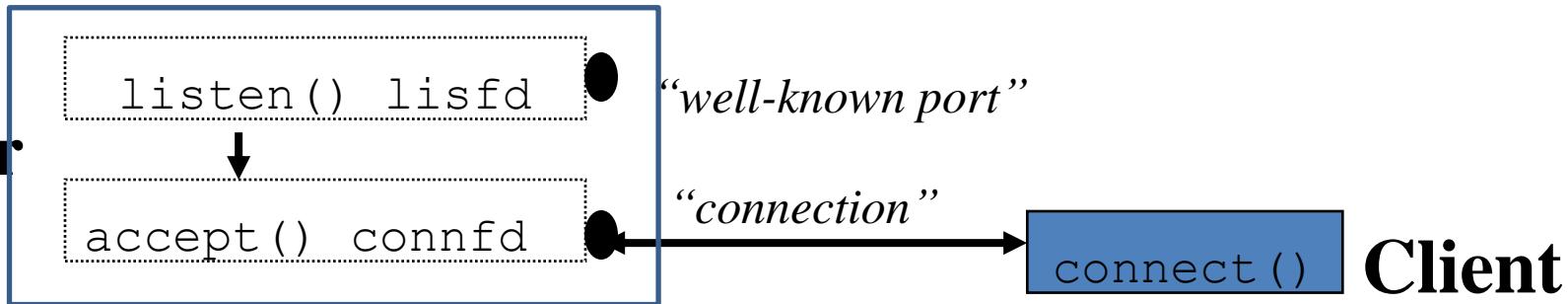
DUE SOCKET: ~~connfd, lisfd~~

CHIUDO SOCKET  
EREDITATA CON  
LA FORK()

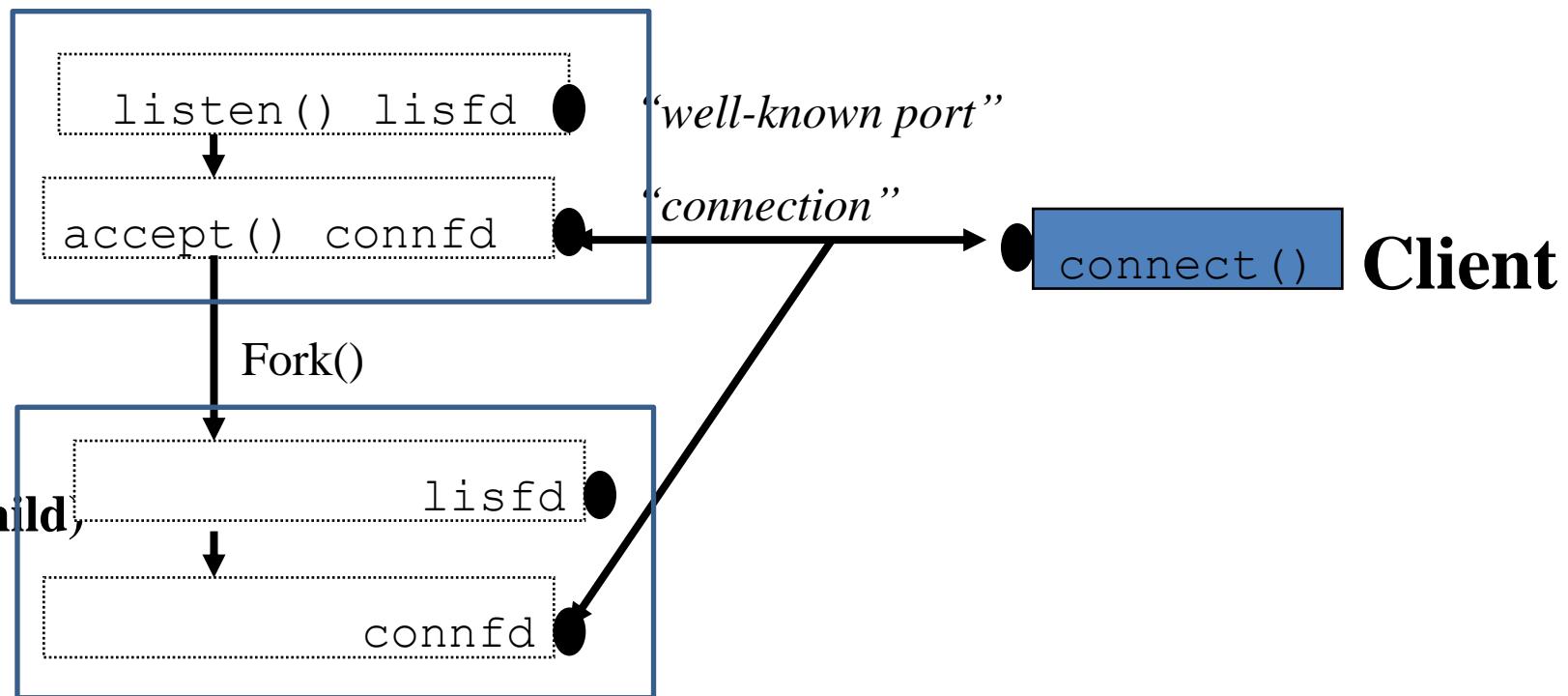


# Status Client-Server after Fork returns

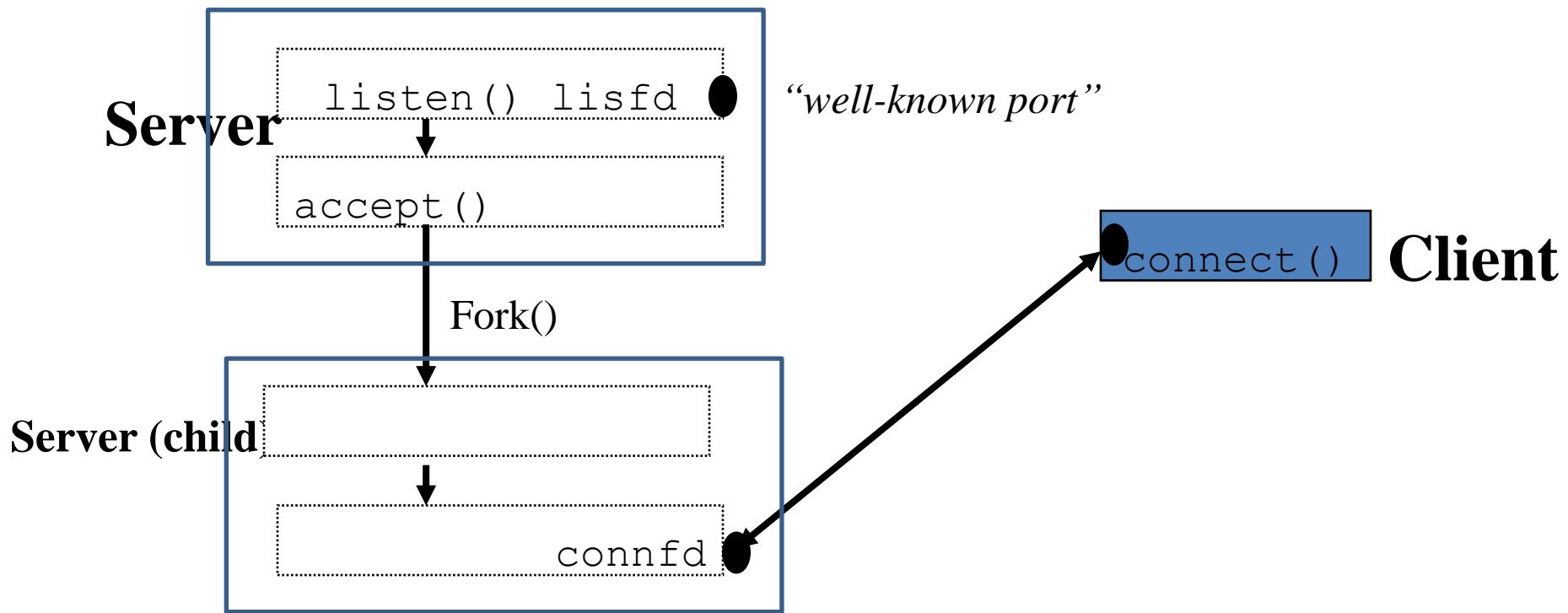
Server

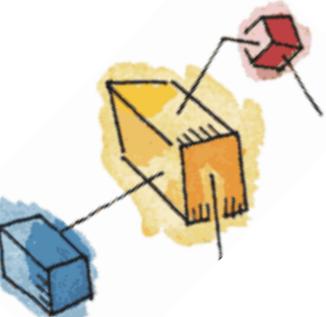


Server



# Status Client-Server after closing sockets





# close()

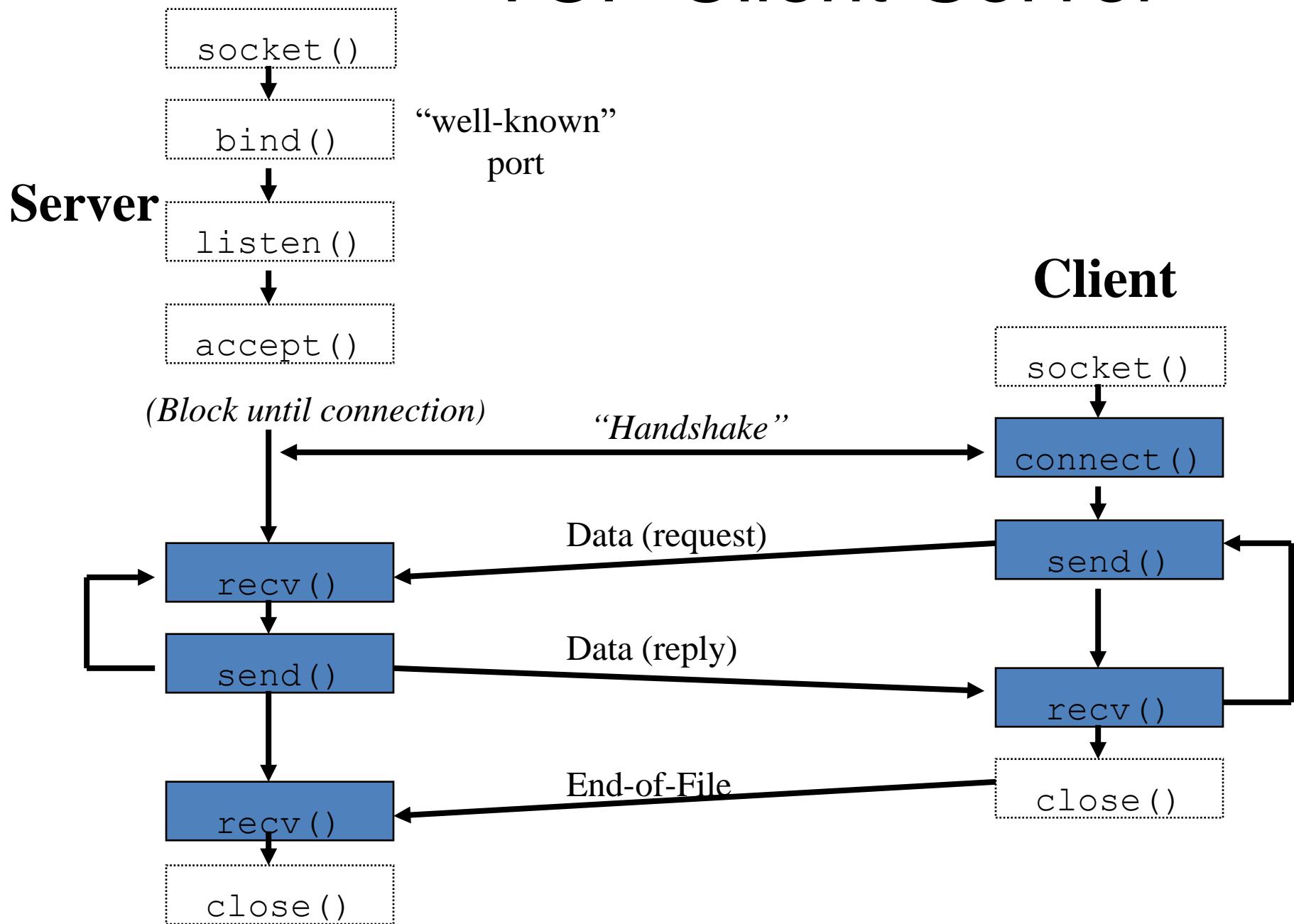
```
int close(int sockfd);
```

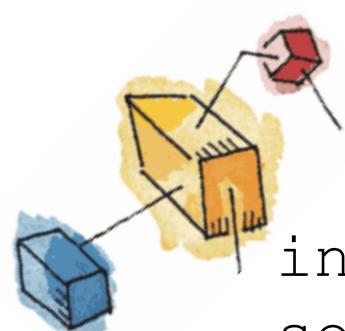
Close socket for use.

- *sockfd* is socket descriptor from `socket()`
- closes socket for reading/writing
  - returns (doesn't block)
  - attempts to send any unsent data
  - socket option: `SO_LINGER` timeout
    - block until data sent
    - or discard any remaining data
  - Returns -1 if error



# TCP Client-Server





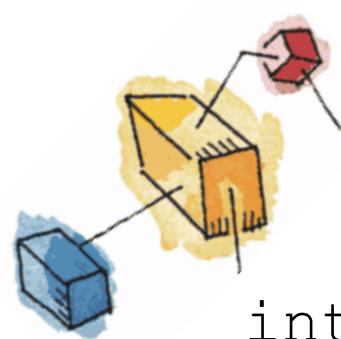
# connect ()

```
int connect(int sockfd, const struct  
sockaddr *servaddr, socklen_t addrlen);
```

**Connect to server.**

- **sockfd** is socket descriptor from `socket()`
- **servaddr** is a pointer to a structure with:
  - port number and IP address
  - must be specified (unlike `bind()`)
- **addrlen** is length of structure
- **Client doesn't need `bind()`**
  - OS will pick ephemeral port
- returns socket descriptor if ok, -1 on error





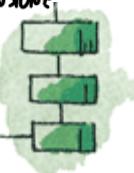
# Sending and Receiving

```
int recv(int sockfd, void *buff, size_t  
numBytes, int flags);
```

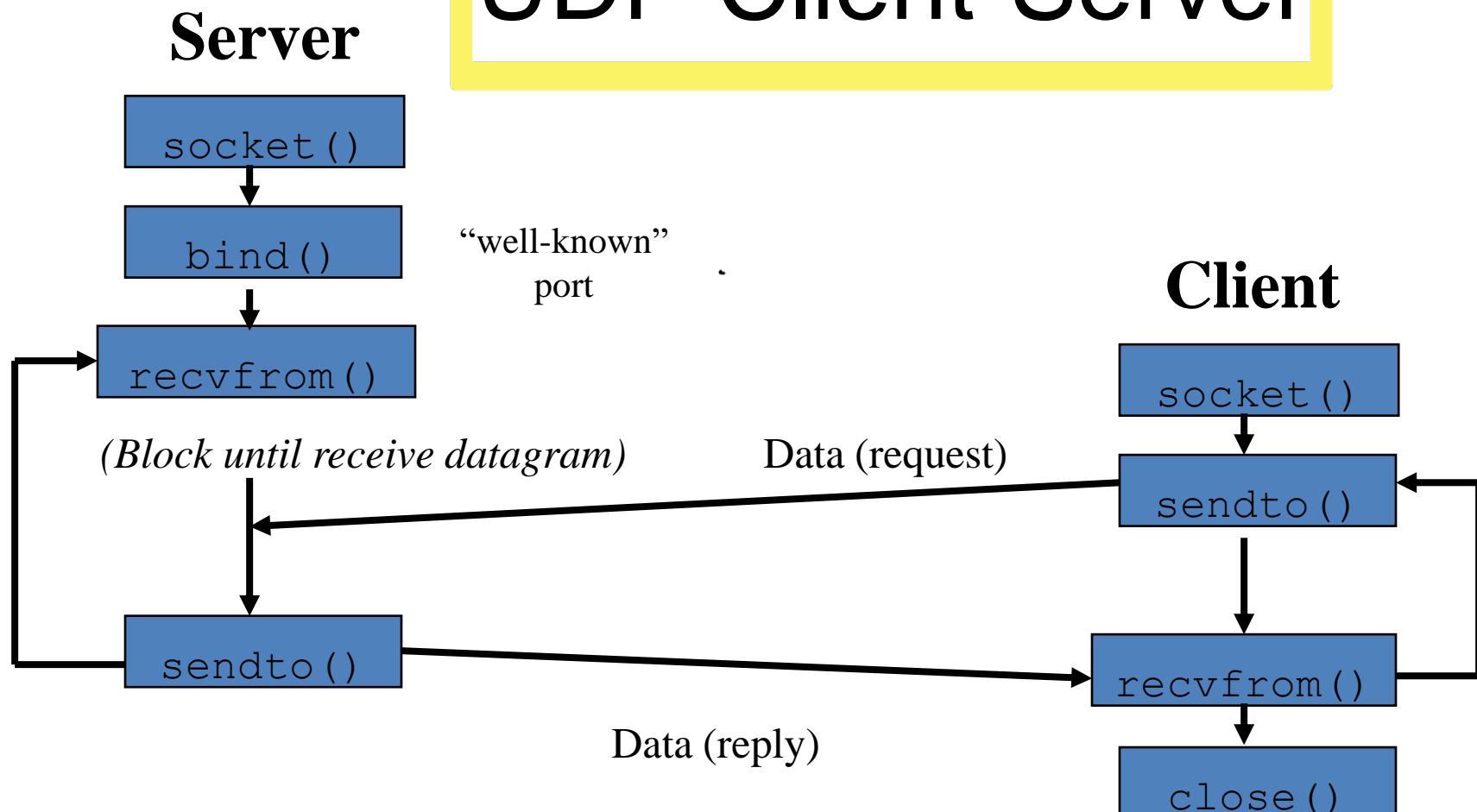
```
int send(int sockfd, void *buff, size_t  
numBytes, int flags);
```

POSSONO ESSERE  
INTERROTTE DA  
SEGNALI

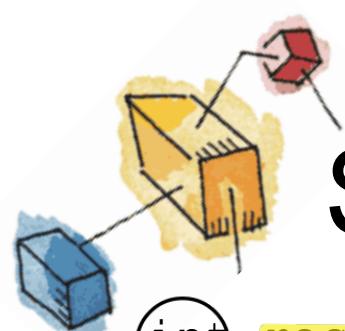
- Same as `read()` and `write()` but for `flags`
  - `MSG_DONTWAIT` (this send non-blocking)
  - `MSG_OOB` (out of band data, 1 byte sent ahead) AUMENTA PRIORITÀ
  - `MSG_PEEK` (look, but don't remove)
  - `MSG_WAITALL` (don't give me less than max) SOLO MESSAGGI CON DIMENSIONE SPECIFICATA
  - `MSG_DONTROUTE` (bypass routing table)



# UDP Client-Server



- No “handshake”
- No simultaneous close
- No `fork()` for concurrent servers!



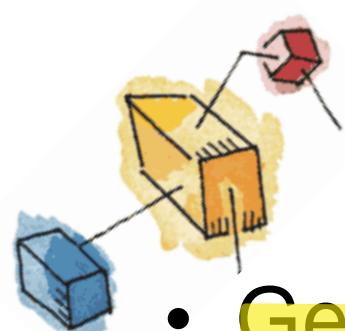
# Sending and Receiving

```
int recvfrom(int sockfd, void *buff, size_t numBytes, int  
m:byte  
numnun flags, struct sockaddr *from, socklen_t *addrLen);
```

```
int sendto(int sockfd, void *buff, size_t numBytes, int  
flags, const struct sockaddr *to, socklen_t addrLen);
```

- Same as `recv()` and `send()` but for `addr`
  - `recvfrom` fills in address of where packet came from
  - `sendto` requires address of where sending packet to

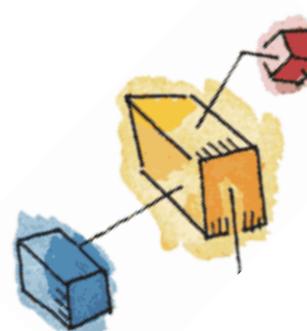




# gethostname()

- Get the name of the host the program is running on.
  - `int gethostname(char *hostname, int bufferLength)`
    - Upon return, `hostname` holds the name of the host
    - `bufferLength` provides a limit on the number of bytes that `gethostname()` can write to `hostname`.



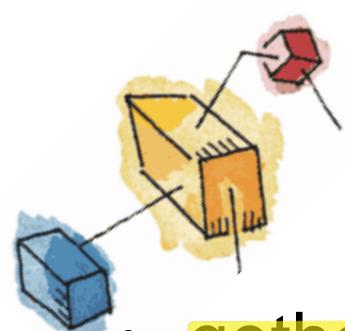


## Internet Address Library Routines (inet\_addr() and inet\_ntoa())

- `unsigned long inet_addr(char *address);`
  - converts address in dotted form to a 32-bit numeric value in network byte order
    - (e.g., “128.173.41.41”) → 32 bit
- `char* inet_ntoa(struct in_addr address)`
  - struct `in_addr`
    - `address.s_addr` is the long int representation

↳ DATO STRUTTURA CON i 32 bit , ride  
l'address come stringa

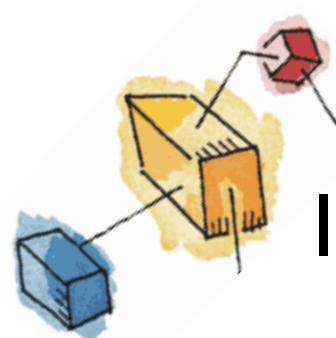




# Domain Name Library Routine

- **gethostbyname()**: Given a host name (such as acavax.lynchburg.edu) get host information.
  - `struct hostent* getbyhostname(const char *hostname)`
    - `char* h_name; // official name of host`
    - `char** h_aliases; // alias list`
    - `short h_addrtype; // address family (e.g., AF_INET)`
    - `short h_length; // length of address (4 for AF_INET)`
    - `char** h_addr_list; // list of addresses (null pointer terminated)`

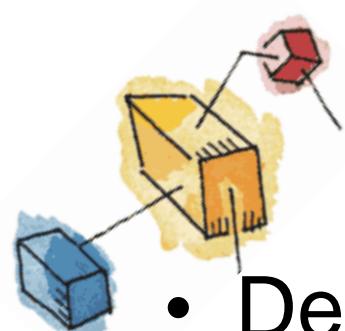




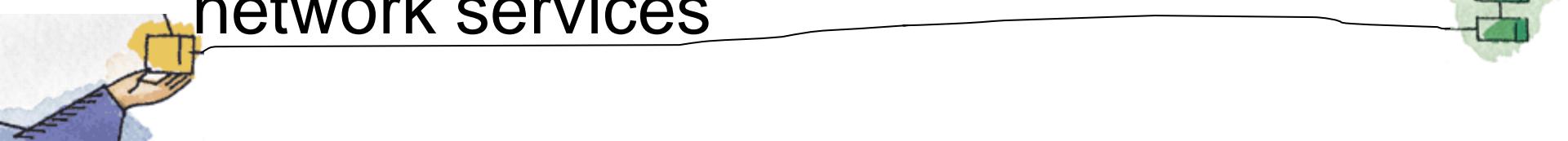
# Internet Address Library Routines

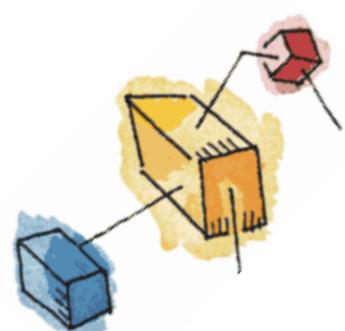
- Get the name of the host given its address
  - `struct hostent* gethostbyaddr(  
const void *addr, int len, int type)`
    - `*addr` is a pointer to a struct of a type depending on the address type (es: `struct in_addr`)
    - `len` is 4 if `type` is `AF_INET`
    - `type` is the address family (e.g., `AF_INET`)





# WinSock

- Derived from Berkeley Sockets (Unix)
    - includes many enhancements for programming in the windows environment
  - Open interface for network programming under Microsoft Windows
    - API freely available
    - Multiple vendors supply WinSock
    - Source and binary compatibility
  - Collection of function calls that provide network services
- 



# Differences Between Berkeley Sockets and WinSock

## Berkeley

## WinSock

bzero()

memset()

close()

closesocket()

read()

not required

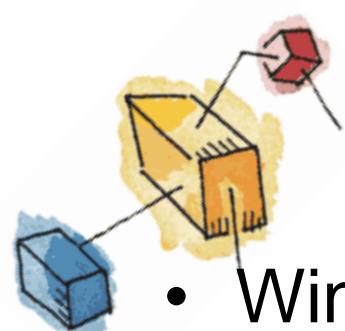
write()

not required

INPUT-OUTPUT-CONTROL ioctl()

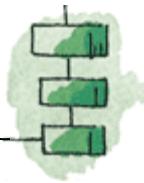
↳ specificare parametri  
specifici di una SOCKET

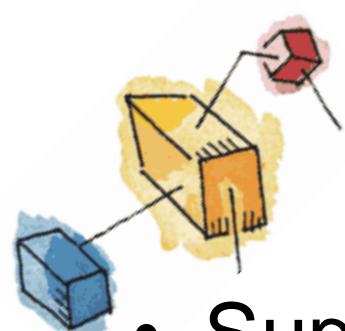
ioctlsocket()



# Additional Features of WinSock 1.1

- WinSock supports three different modes
  - Blocking mode
    - socket functions don't return until their jobs are done
    - same as Berkeley sockets
  - Non-blocking mode
    - Calls such as accept() don't block, but simply return a status
  - Asynchronous mode
    - Uses Windows messages
      - FD\_ACCEPT - connection pending
      - FD\_CONNECT - connection established
      - etc.

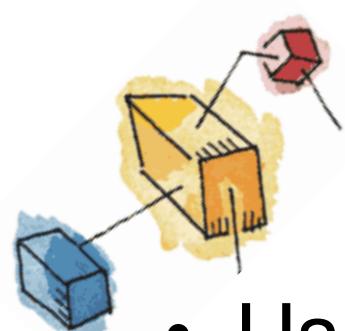




# WinSock 2

- Supports protocol suites other than TCP/IP
  - DecNet
  - IPX/SPX
  - OSI
- Supports network-protocol independent applications
- Backward compatible with WinSock 1.1





# WinSock 2 (Continued)

- Uses different files
  - winsock2.h
  - different DLL (WS2\_32.DLL)
- API changes
  - accept() becomes WSAAccept()
  - connect() becomes WSACConnect()
  - inet\_addr() becomes  
WSAAddressToString()
  - etc.



# Stream Sockets in Java

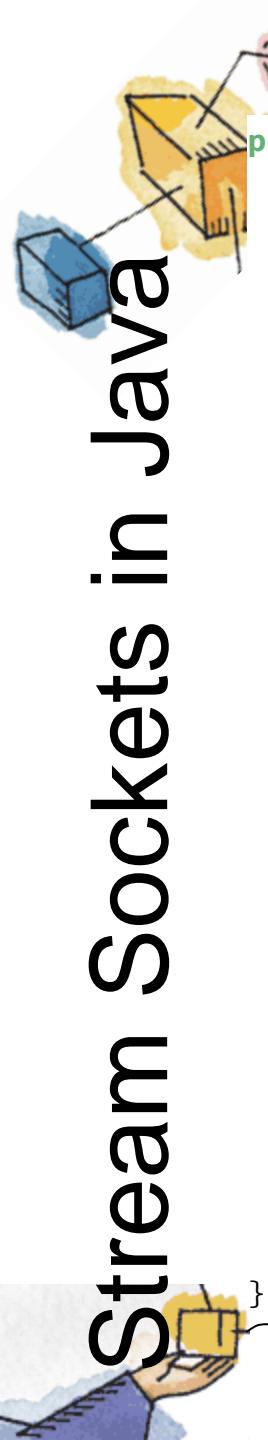
```
public class GreetServer {  
    private ServerSocket serverSocket;  
    private Socket clientSocket;  
    private PrintWriter out;  
    private BufferedReader in;  
  
    public void start(int port) {  
        serverSocket = new ServerSocket(port);  
        clientSocket = serverSocket.accept();  
        out = new PrintWriter(clientSocket.getOutputStream(), true);  
        in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));  
        String greeting = in.readLine();  
        if ("hello server".equals(greeting)) {  
            out.println("hello client");  
        }  
        else {  
            out.println("unrecognised greeting");  
        }  
    }  
  
    public void stop() {  
        in.close();  
        out.close();  
        clientSocket.close();  
        serverSocket.close();  
    }  
    public static void main(String[] args) {  
        GreetServer server=new GreetServer();  
        server.start(6666);  
        server.stop();  
    }  
}
```

Obviously a well done server must include

- Process/thread generation
- Continuos listening
- Repeated communication

# Stream Sockets in Java

```
public class GreetClient {  
    private Socket clientSocket;  
    private PrintWriter out;  
    private BufferedReader in;  
  
    public void startConnection(String ip, int port) {  
        clientSocket = new Socket(ip, port);  
        out = new PrintWriter(clientSocket.getOutputStream(), true);  
        in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));  
    }  
  
    public String sendMessage(String msg) {  
        out.println(msg);  
        String resp = in.readLine();  
        return resp;  
    }  
  
    public void stopConnection() {  
        in.close();  
        out.close();  
        clientSocket.close();  
    }  
  
    public static void main(String[] args) {  
        GreetClient client = new GreetClient();  
        client.startConnection("127.0.0.1", 6666);  
        String response = client.sendMessage("hello server");  
        assertEquals("hello client", response);  
        client.stopConnection()  
    }  
}
```



# UDP Sockets in Java

```
class UDPServer {
    public static void main(String args[]) throws Exception
    {

        DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] receiveData = new byte[1024];
        byte[] sendData = new byte[1024];

        while(true){

            DatagramPacket receivePacket =
                new DatagramPacket(receiveData, receiveData.length);
            serverSocket.receive(receivePacket);

            String sentence = new String(receivePacket.getData());
            InetAddress IPAddress = receivePacket.getAddress();
            int port = receivePacket.getPort();

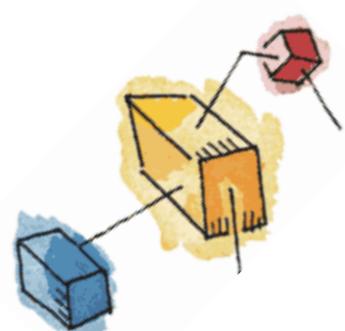
            String capitalizedSentence = sentence.toUpperCase();
            sendData = capitalizedSentence.getBytes();

            DatagramPacket sendPacket =
                new DatagramPacket(sendData, sendData.length, IPAddress, port);

            serverSocket.send(sendPacket);
        }
    }
}
```

# Stream Sockets in Java

```
class UDPClient {  
    public static void main(String args[]) throws Exception  
{  
  
    BufferedReader inFromUser =  
        new BufferedReader(new InputStreamReader(System.in));  
  
    DatagramSocket clientSocket = new DatagramSocket();  
  
    InetAddress IPAddress = InetAddress.getByName("hostname");  
  
    byte[] sendData = new byte[1024];  
    byte[] receiveData = new byte[1024];  
  
    String sentence = inFromUser.readLine();  
    sendData = sentence.getBytes();  
    DatagramPacket sendPacket =  
        new DatagramPacket(sendData, sendData.length, IPAddress, 9876);  
  
    clientSocket.send(sendPacket);  
  
    DatagramPacket receivePacket =  
        new DatagramPacket(receiveData, receiveData.length);  
  
    clientSocket.receive(receivePacket);  
  
    String modifiedSentence = new String(receivePacket.getData());  
  
    System.out.println("FROM SERVER:" + modifiedSentence);  
    clientSocket.close();  
}  
}
```



# Stream Sockets in Python

```
#!/usr/bin/env python3
# server

import socket

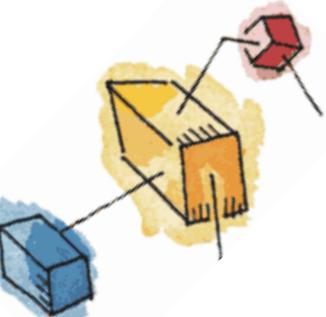
HOST = '127.0.0.1'    # Standard loopback interface address (localhost)
PORT = 65432           # Port to listen on (non-privileged ports are > 1023)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen()
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data:
            break
        conn.sendall(data)
s.close()
```

Obviously a well done server must include

- Process/thread generation
- Continuos listening
- Repeated communication





# Stream Sockets in Python

```
#!/usr/bin/env python3
# client

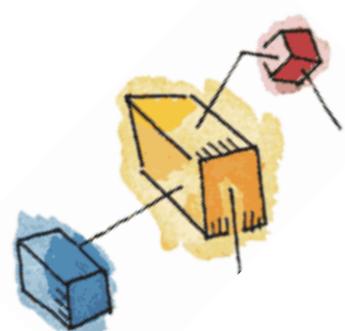
import socket

HOST = '127.0.0.1'    # The server's hostname or IP address
PORT = 65432           # The port used by the server

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.sendall(b'Hello, world')
data = s.recv(1024)

print('Received', repr(data))
s.close()
```





# UDP Sockets in Python

```
#!/usr/bin/env python3
# server

import socket

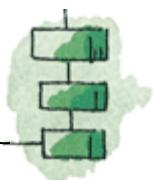
UDP_IP = "127.0.0.1"
UDP_PORT = 5005

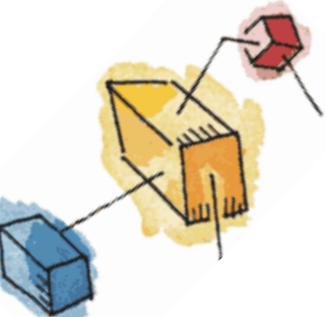
sock = socket.socket(socket.AF_INET, # Internet
                     socket.SOCK_DGRAM) # UDP
sock.bind((UDP_IP, UDP_PORT))

while True:
    data, addr = sock.recvfrom(1024) # buffer size is 1024 bytes
    print "received message:", data
    sock.sendto(data, addr)
```

Obviously a well done server must include

- Process/thread generation
- Continuos listening
- Repeated communication





# UDP Sockets in Python

```
#!/usr/bin/env python3
# client

import socket

UDP_IP = "127.0.0.1"
UDP_PORT = 5005
MESSAGE = "Hello, World!"

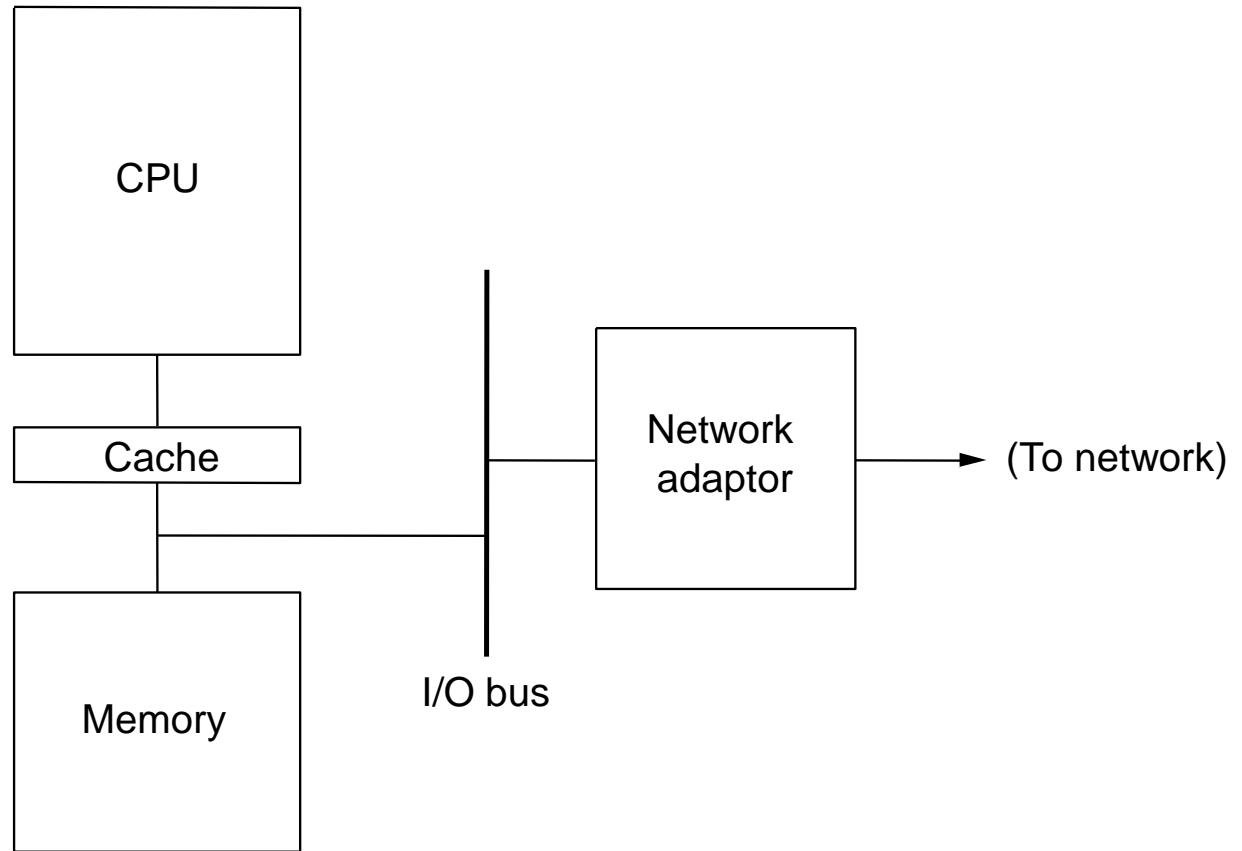
print "UDP target IP:", UDP_IP
print "UDP target port:", UDP_PORT
print "message:", MESSAGE

sock = socket.socket(socket.AF_INET, # Internet
                     socket.SOCK_DGRAM) # UDP
sock.sendto(MESSAGE, (UDP_IP, UDP_PORT))
data, addr = sock.recvfrom(1024)
sock.close()
```



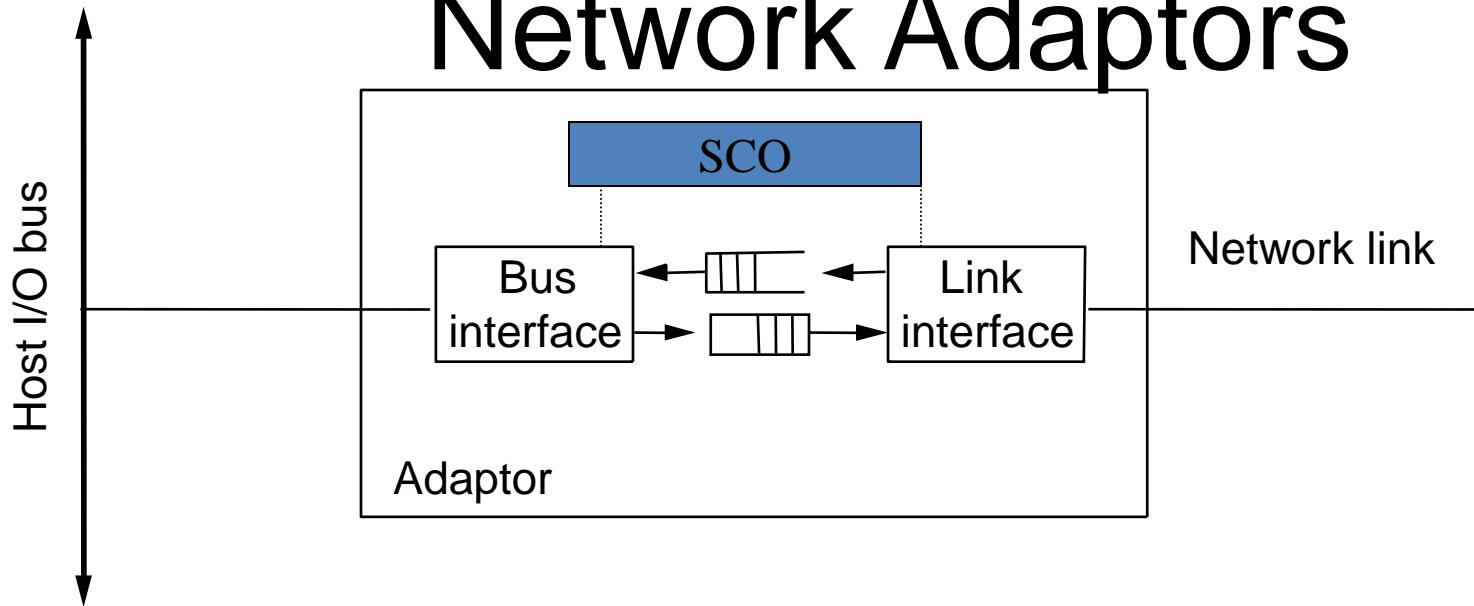
# Network Adaptors

# Network Adaptors



Interfaccia tra Host e Rete

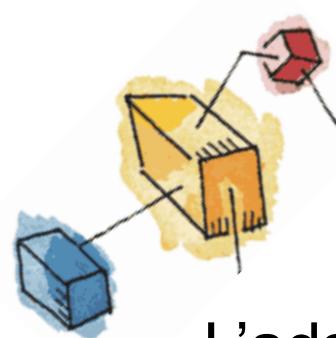
# Network Adaptors



La scheda è costituita da due parti separate che interagiscono attraverso una FIFO che maschera l'asincronia tra la rete e il bus interno

- La prima parte interagisce con la CPU della scheda
- La seconda parte interagisce con la rete (implementando il livello fisico e di collegamento)

Tutto il sistema è comandato da una SCO (sottosistema di controllo della scheda)



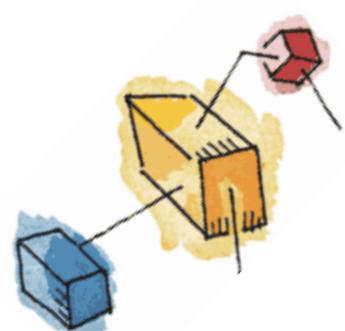
# Vista dall'host

- L'adaptor esporta verso la CPU uno o più registri macchina (Control Status Register)
- CSR è il mezzo di comunicazione tra la SCO della scheda e la CPU

*Esempio di scheda di rete*

```
/* CSR
 * leggenda: RO - read only; RC - Read/Clear (writing 1 clear, writing 0 has no effect);
 * W1 write-1-only (writing 1 sets, writing 0 has no effect)
 * RW - read/write; RW1 - Read-Write-1-only
 */
#define LE_ERR 0X8000      risetto un bit
.....
#define LE_RINT 0X0400    /* RC richiesta di interruzione per ricevere un pacchetto */
#define LE_TINT 0X0200    /* RC pacchetto trasmesso */
.....
#define LE_INEA 0X0040    /* RW abilitazione all'emissione di un interrupt da parte
                         dell'adaptor verso la CPU */
#define LE_TDMD 0X0008    /* W1 richiesta di trasmissione di un pacchetto dal device
                         driver verso l'adaptor */
.....
```





# Vista dall'host

L'host può controllare cosa accade in CSR in **due modi**

- **Busy waiting** (la CPU esegue un test continuo di CSR fino a che CSR non si modifica indicando la nuova operazione da eseguire. Ragionevole solo per calcolatori che non devono fare altro che attendere e trasmettere pacchetti, ad esempio i router)
- **Interrupt** (l'adaptor invia un interrupt all'host, il quale fa partire un *interrupt handler* che va a leggere CSR per capire l'operazione da fare)



# Trasferimento dati da adaptor a memoria (e viceversa)

- Direct Memory Access (Più comune) PARTE DELLA RAM assegnate alla NET. INT.
  - Nessun coinvolgimento della CPU nello scambio dati
  - Il SO assegna un'area di memoria all'host
  - Frame inviati immediatamente alla memoria di lavoro dell'host
  - Pochi bytes di memoria necessari sulla scheda
- Programmed I/O
  - Lo scambio dati tra memoria e adaptor passa per la CPU
  - Impone di bufferizzare almeno un frame sull'adaptor
  - La memoria deve essere di tipo dual port
    - Il processore e l'adaptor possono sia leggere che scrivere in questa porta

# DMA: Buffer Descriptor list (BD)

La memoria dove allocare i frames è organizzata attraverso una *buffer descriptor list*

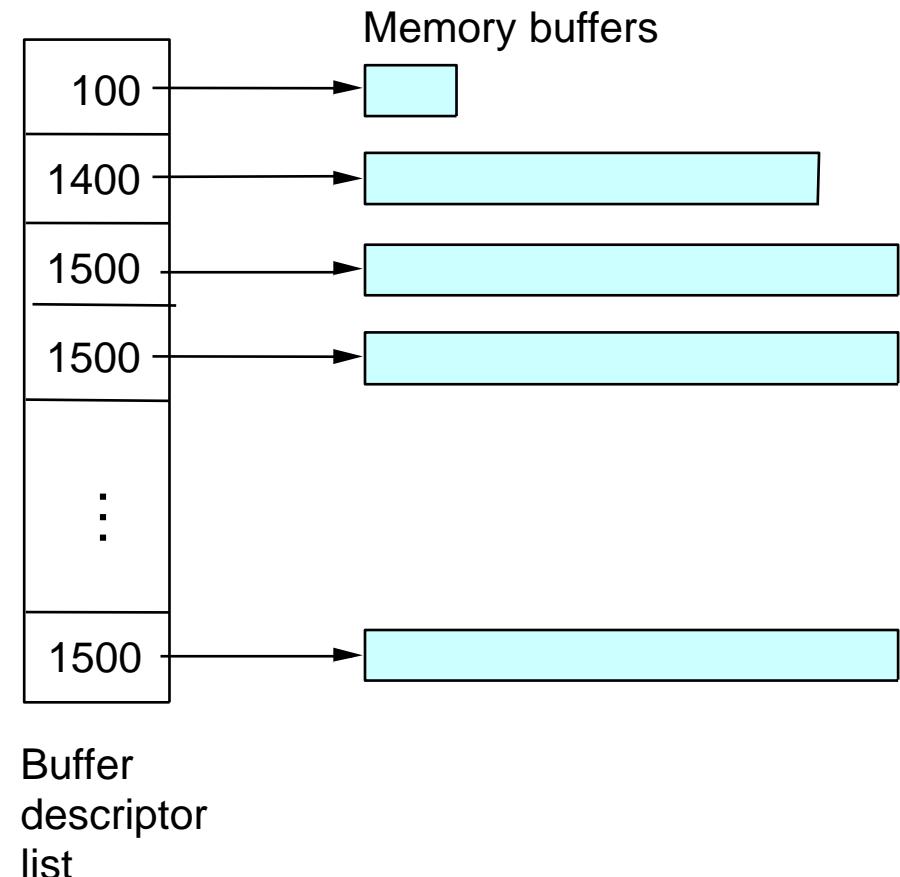
1 in scrittura

1 in lettura

(PRODUTTORE - CONSUMATORE)

Un vettore di puntatori ad aree di memoria (buffers) dove è descritta la quantità di memoria disponibile in quell'area

In ethernet vengono tipicamente preallocati 64 buffers da 1500 bytes



# Buffer Descriptor list

Tecnica usata per frame che arrivano dall'adaptor:

*scatter read / gather write*

- frame distinti sono allocati in buffer distinti
- un frame può essere allocato su più buffer (se più grande del buffer)
  - In ethernet non necessario

# Viaggio di un messaggio all'interno dell'SO

Quando un messaggio viene inviato da un utente in un certo socket

1. Il SO copia il messaggio dal buffer della memoria utente in una zona di BD
2. Tale messaggio viene processato da tutti i livelli protocollari (esempio TCP, IP, device driver) che provvedono ad inserire gli opportuni header e ad aggiornare gli opportuni puntatori presenti nel BD in modo da poter sempre ricostruire il messaggio
3. Quando il messaggio ha completato l'attraversamento del protocol stack, viene avvertita la SCO dell'adaptor dal device driver attraverso il set dei bit del CSR (LE\_TDMD e LE\_INEA). Il primo invita la SCO ad inviare il messaggio sulla linea. Il secondo abilita la SCO ad inviare una interruzione
4. La SCO dell'adaptor invia il messaggio sulla linea
5. Una volta terminata la trasmissione, la SCO notifica il termine alla CPU attraverso il set del bit (LE\_TINT) del CSR e scatena una interruzione
6. Tale interruzione avvia un interrupt handler che prende atto della trasmissione, resetta gli opportuni bit (LE\_TINT e LE\_INEA) e libera le opportune risorse (operazione semsignal su xmit\_queue)

# Device Drivers

Il device driver è una collezione di routine (inizializzare l'adaptor, invio di un frame sul link etc.) di OS che serve per “ancorare” il SO all’hardware sottostante specifico dell’adaptor

Esempio routine di richiesta di invio di un messaggio sul link

```
#define csr ((u_int) 0xffff3579 /*CSR address*/
Transmit(Msg *msg)
{
    descriptor *d;
    semwait(xmit_queue);      /* abilita non piu' di 64 accessi al BD*/
    d=next_desc();
    prepare_desc(d,msg);
    semwait(mutex);           /* abilità a non piu' di un processo (dei potenziali 64)
                                 alla volta la trasmissione verso l'adaptor */
    disable_interrupts();      /* il processo in trasmissione si protegge da eventuali
                                 interruzioni dall'adaptor */
    csr= LE_TDMD | LE_INEA;   /* una volta preparato il messaggio invita la SCO dell'adaptor a
                                 trasmetterlo e la abilita la SCO ad emettere una interruzione
                                 una volta terminata la trasmissione */
    enable_interrupts();       /* riabilita le interruzioni */
    semsignal(mutex);         /* sblocca il semaforo per abilitare un altro processo a
                                 trasmettere */
}
```

“next\_desc()” ritorna il prossimo buffer descriptor disponibile nel buffer descriptor list  
“prepare\_desc(d,msg)” il messaggio msg nel buffer d in un formato comprensibile dall’adaptor



# Interrupt Handler

- Disabilita le interruzioni
- Legge il CSR per capire che cosa deve fare: tre possibilità
  1. C'è stato un errore
  2. Una trasmissione è stata completata
  3. Un frame è stato ricevuto
- Noi siamo nel caso 2
  - LE\_TINT viene messo a zero (RC bit)
  - Ammette un nuovo processo nella BD poiché un frame è stato trasmesso
  - Abilita le interruzioni

```
lance_interrupt_handler()
{
    disable_interrupts();

    /* some error occurred */
    if (csr & LE_ERR)
    {
        print_error(csr);
        /* clear error bits */
        csr = LE_BABL | LE_CERR | LE_MISS | LE_MERR | LE_INEA;
        enable_interrupts();
        return();
    }

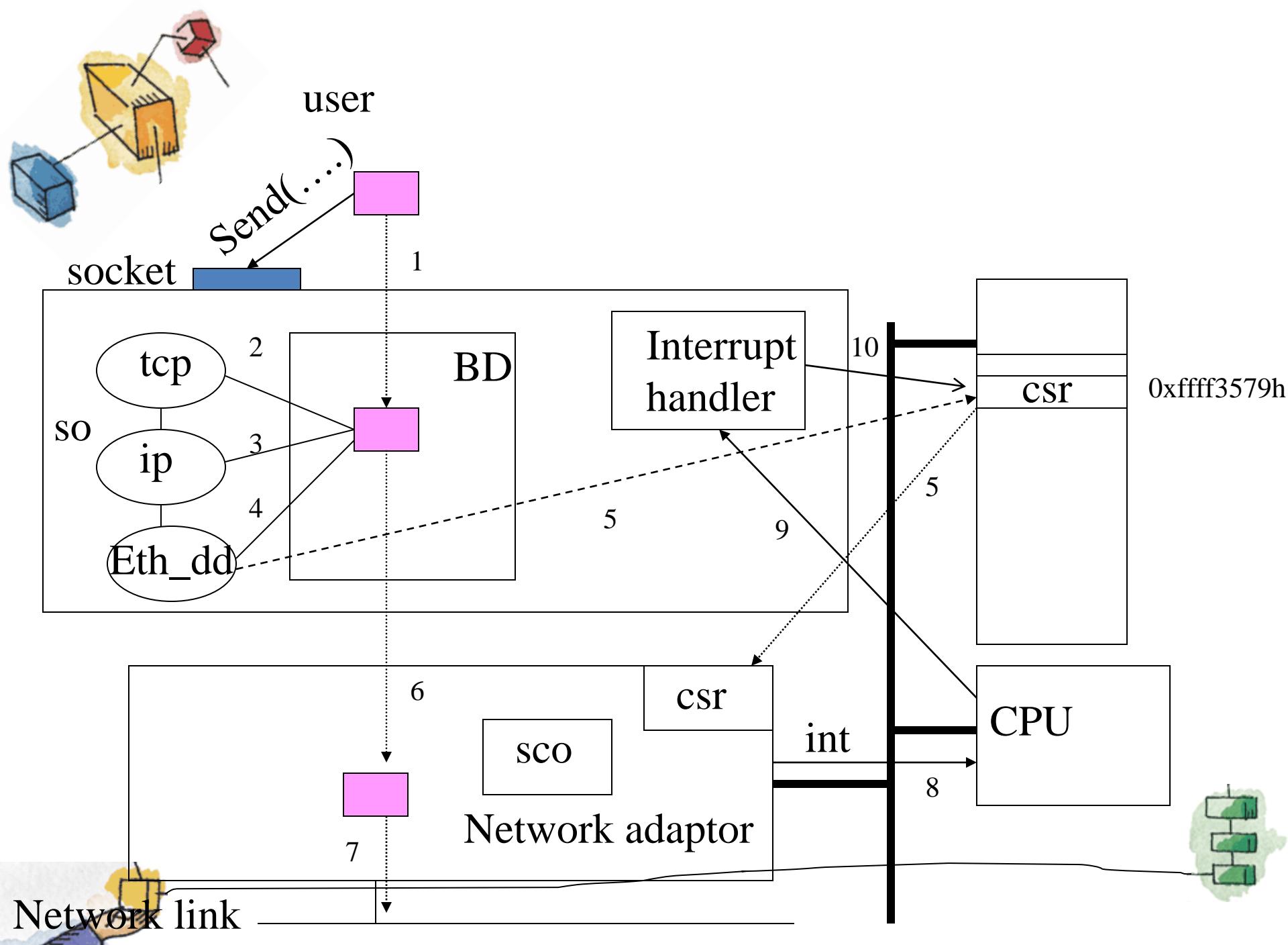
    /* transmit interrupt */
    if (csr & LE_TINT)
    {
        /* clear interrupt */
        csr = LE_TINT | LE_INEA;

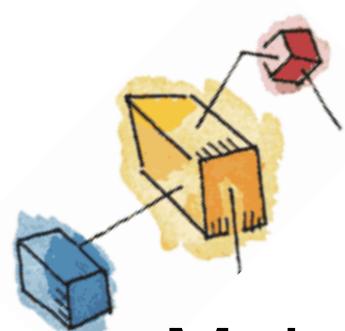
        /* signal blocked senders */
        semSignal(xmit_queue);

        enable_interrupts();
        return(0);
    }

    /* receive interrupt */
    if (csr & LE_RINT)
    {
        /* clear interrupt */
        csr = LE_RINT | LE_INEA;

        /* process received frame */
        lance_receive();
        enable_interrupts();
        return();
    }
}
```





# References

## Main references

- W. Stallings, «Operating Systems», Chapter 17, 6th edition (freely available at <https://app.box.com/s/mbh0v0f6nx>)
- L. Peterson & B. Davie, «Computer Networks: A systems approach», 3rd edition, pp. 137-144

## Further references

- A. Tanenbaum & D. Wetherall, «Computer Networks»
- W.R. Stevens, «Unix Network Programming»



# Distributed Systems

Most slides have been adapted from  
«Distributed systems: concepts and design», 3/E Coulouris et al.  
(Chapter 1+2)  
«*Operating Systems: Internals and Design Principles*», 7/E W. Stallings  
(Chapter 16)

*Sistemi di Calcolo 2*  
*Riccardo Lazzeretti*

# **INTRODUCTION TO DISTRIBUTED SYSTEMS**

# Distributed system definition

- A distributed system is a set of spatially separated entities, each of them with a certain computational power, that are able to communicate and to coordinate among themselves for reaching a common goal and that appears to its users as a single coherent system

Anche zoom è un sistema distribuito, tante piccole entità che comunicano tra di loro

# Introduction to distributed systems

- Why do we develop distributed systems?
  - availability of powerful yet cheap microprocessors (PCs, workstations), continuing advances in communication technology,
- What is a distributed system?
- A distributed system is a collection of independent computers that appear to the users of the system as a single system.
- Examples:
  - Network of workstations
  - Distributed manufacturing system (e.g., automated assembly line)
  - Network of branch office computers

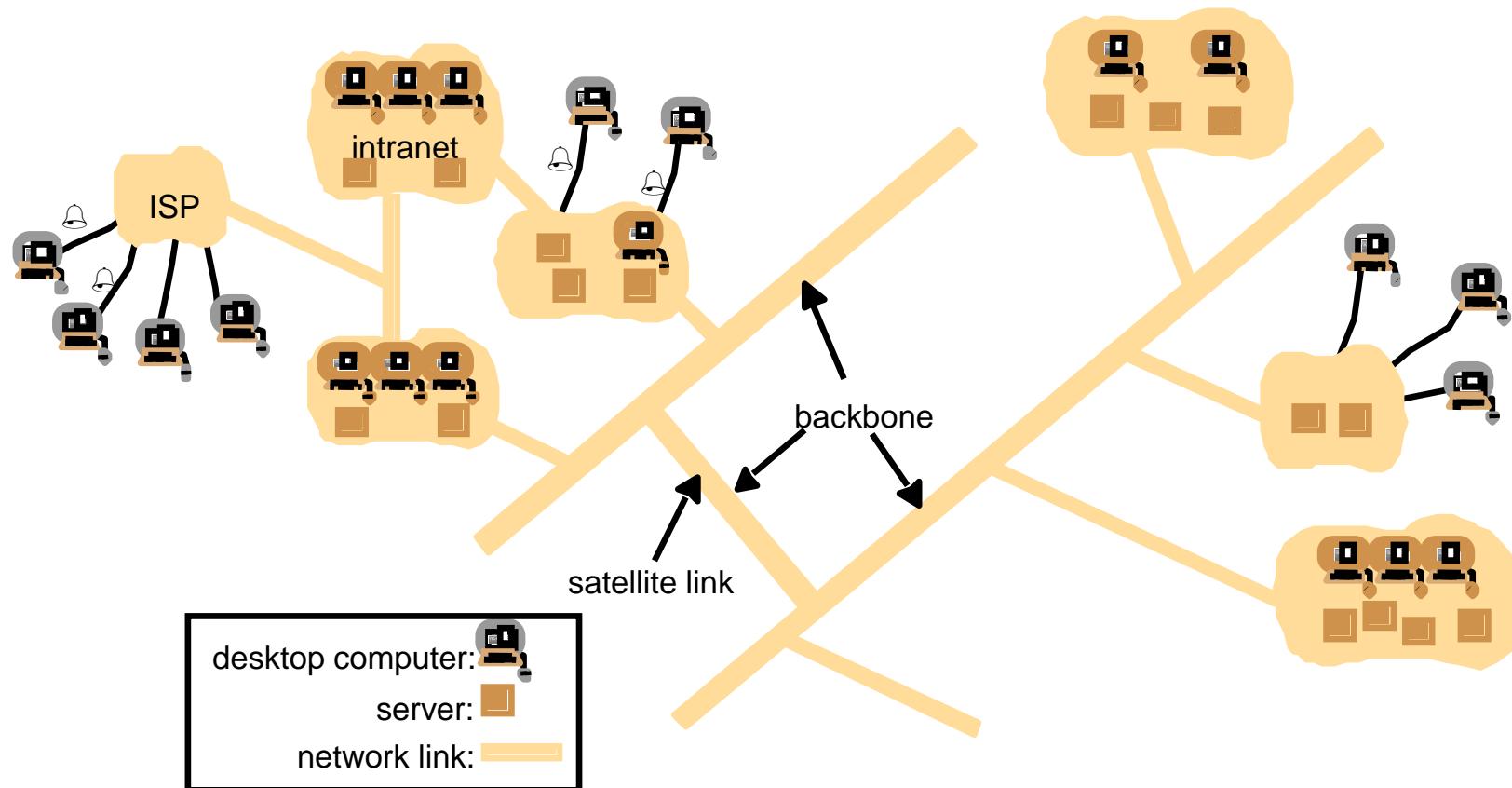
# Selected application domains and associated networked applications

|  |   |
|--|---|
| <i>Finance and commerce</i>                  | eCommerce e.g. Amazon and eBay, PayPal, online banking and trading                              |
| <i>The information society</i>               | Web information and search engines, ebooks, Wikipedia; social networking: Facebook and MySpace. |
| <i>Creative industries and entertainment</i> | online gaming, music and film in the home, user-generated content, e.g. YouTube, Flickr         |
| <i>Healthcare</i>                            | health informatics, on online patient records, monitoring patients                              |
| <i>Education</i>                             | e-learning, virtual learning environments; distance learning                                    |
| <i>Transport and logistics</i>               | GPS in route finding systems, map services: Google Maps, Google Earth                           |
| <i>Science</i>                               | The Grid as an enabling technology for collaboration between scientists                         |
| <i>Environmental management</i>              | sensor technology to monitor earthquakes, floods or tsunamis                                    |

# Examples of Distributed Systems

- Local Area Network and Intranet
- Database Management System
- Automatic Teller Machine Network (**BANCOMAT**)
- Internet/World-Wide Web
- Pervasive Systems and Ubiquitous Computing
  - Service Oriented Architecture
    - ↳ TESSERE CONTACTLESS,  
GOOGLE GLASS, ...
- Virtual networks
- Peer-to-peer (P2P)
- Cloud Computing
- Big Data Computing

# Internet



# World-Wide-Web

Uppsala universitet - Windows Internet Explorer  
http://www.uu.se

File Edit View Favorites Tools Help

Google Search Bookmarks Check Translate

Windows Live Bing What's New Profile Mail Photos Calendar MSN Share

Startsida Utbildning Forskning Samverkan Om UU Bibliotek Kontakt

In English Karta Personal Enhet  
Lyssna Fritext

RSS Sök

UPPSALA UNIVERSITET

Välkommen till Uppsala universitet!



Rektor Anders Hallberg:  
*"Vid Uppsala universitet arbetar vi målmedvetet och långsiktigt för att alltid kunna erbjuda de bästa förutsättningarna för vår utbildning och forskning."* Välkommen till Uppsala!

Genvägar

- » Alumn
- » Doktorand
- » Fakulteter och enheter
- » Internt för anställda
- » Lediga anställningar
- » Pressinformation
- » Student
- » Stöd Uppsala universitet

Aktuellt

Om Influenta A (H1N1)  
Här kan du vaccinera dig

Expedition Antarktis



Forskningsexpedition till Antarktis om influensavirus och antibiotikaresistens. Följ forskningsresan på professor Björn Olsens blogg.

Nyheter

Ännu fler miljoner i anslag till Hans Ellegren  
FN:s Karen AbuZayd årets Dag  
Hammarskjöldföreläsare

Blåsenhus



Blåsenhus - Uppsala universitets nyaste campusområde för nya

# Massively multiplayer online games (MMOGs)

- Offer an immersive experience whereby very large numbers of users interact through the Internet with a persistent virtual world
- Include:
  - Complex playing arenas
  - Social systems
  - Financial systems
- **Require:**
  - Fast responses
  - Real time propagation of events
  - Consistent view of the entire world

# Advantages of Distributed Systems over Centralized Systems

- **Economics:** a collection of microprocessors offer a better price/performance than mainframes. Low price/performance ratio: cost effective way to increase computing power.
- **Speed:** a distributed system may have more total computing power than a mainframe. Ex. 10,000 CPU chips, each running at 50 MIPS. Not possible to build 500,000 MIPS single processor since it would require 0.002 nsec instruction cycle. Enhanced performance through load distributing.
- **Inherent distribution:** Some applications are inherently distributed. Ex. a supermarket chain.
- **Reliability:** If one machine crashes, the system as a whole can still survive. Higher availability and improved reliability.
- **Incremental growth:** Computing power can be added in small increments. Modular expandability
- **Another deriving force:** the existence of large number of personal computers and IoTs, the need for people to collaborate and share information.

# Advantages of Distributed Systems over Independent PCs

- **Data sharing:** allow many users to access to a common data base
- **Resource Sharing:** expensive peripherals like color printers
- **Communication:** enhance human-to-human communication, e.g., email, chat
- **Flexibility:** spread the workload over the available machines

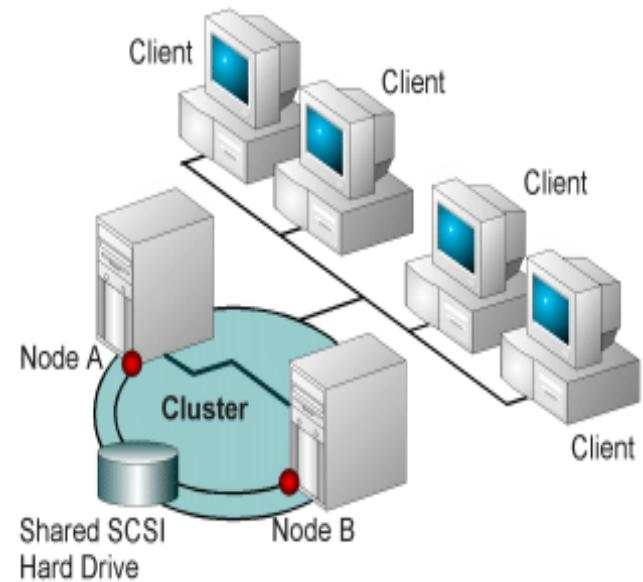
# Disadvantages of Distributed Systems

- **Software:** difficult to develop software for distributed systems
- **Network:** saturation, lossy transmissions
- **Security:** easy access also applies to secrete data

# Primary Goal: sharing data/resources

## Problems

- Synchronization
- Coordination



# Coordination

take into account the following condition that deviates from centralized systems:

1. Temporal and spatial concurrency
2. No global Clock
3. Failures
4. Unpredictable latencies

These limitations restrict the set of coordination problems we can solve in a distributed setting

# Design Issues of Distributed Systems

- Heterogeneity
- Openness
- Security
- Scalability
- Reliability
- Concurrency
- Flexibility
- Performance
- Transparency

# Heterogeneity

- networks
- computer hardware
- operating systems
- programming languages
- implementations by different developers
- mobile code

# Opennes

- Dyistributed services may be usable
- Their interfaces must be published
- Open distributed systems are based on the provision of
  - a uniform communication mechanism
  - published interfaces for access to shared resources.
- Open distributed systems can be constructed from heterogeneous hardware and software
  - the conformance of each component to the published standard must be carefully tested

# Security

- confidentiality
  - protection against disclosure to unauthorized individuals
- integrity
  - protection against alteration or corruption
- availability
  - protection against interference with the means to access the resources

# Scalability

- Systems grow with time or become obsolete.
  - Non linear growth
- Challenges:
  - Controlling the cost of physical resources
  - Controlling the performance loss
  - Preventing software resources running out (come in esaurimento IP)
  - Avoiding performance bottlenecks (colli di bottiglia)
- Examples of bottlenecks
  - Centralized components: a single mail server
  - Centralized tables: a single URL address book
  - Centralized algorithms: routing based on complete information

# Reliability

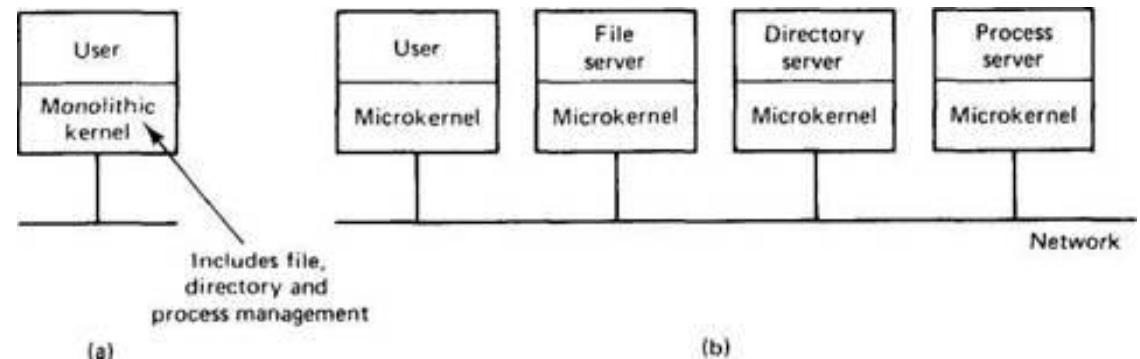
- Distributed system should be more reliable than single system.
- Example: 3 machines with .95 probability of being up.  $1-.05^3$  probability of being up.
- Fault tolerance
  - Detecting failures
  - Masking failures
  - Tolerating failures
  - Fault recovery
  - Redundancy

# Concurrency

- Do I really need to explain this one?
- Yes, I do
- Spatial concurrency
- Time concurrency

# Flexibility

- Make it easier to change
- Monolithic Kernel: systems calls are trapped and executed by the kernel. All system calls are served by the kernel, e.g., UNIX.
- + Microkernel: provides minimal services.
  - 1) InterProcess Communication
  - 2) some memory management
  - 3) some low-level process management and scheduling
  - 4) low-level I/O



# Performance

- Without gain on this, why bother with distributed systems.
- **Performance loss due to communication delays:**
  - fine-grain parallelism: high degree of interaction
  - + coarse-grain parallelism
- **Performance loss due to making the system fault tolerant.**

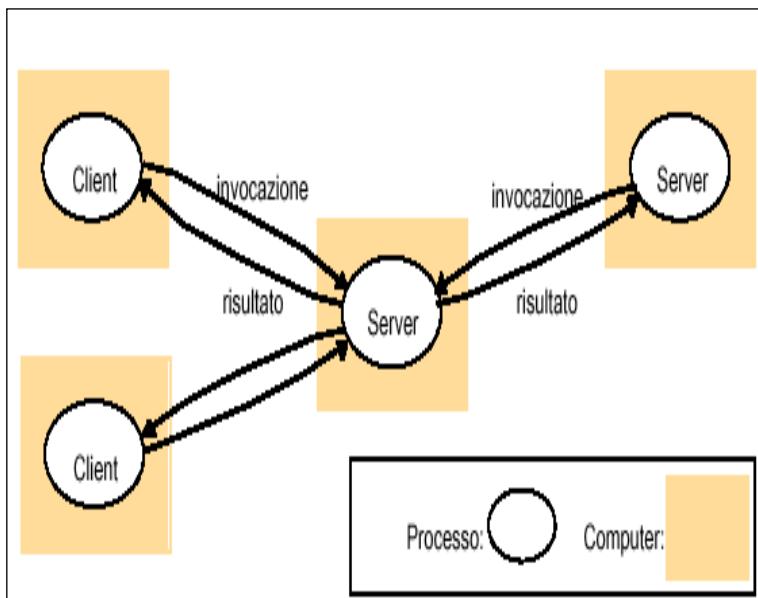
# Transparency

- How to achieve the single-system image, i.e.,  
how to make a collection of computers  
appears as a single computer.
  - **Access Transparency:** local and remote resources are accessed using the same operations
  - **Location Transparency:** users cannot tell where hardware and software resources such as CPUs, printers, files, data bases are located.
  - **Concurrency Transparency:** The users are not aware of the existence of other users. Need to allow multiple users to concurrently access the same resource. Lock and unlock for mutual exclusion.
  - **Replication Transparency:** OS can make additional copies of files and resources without users noticing.

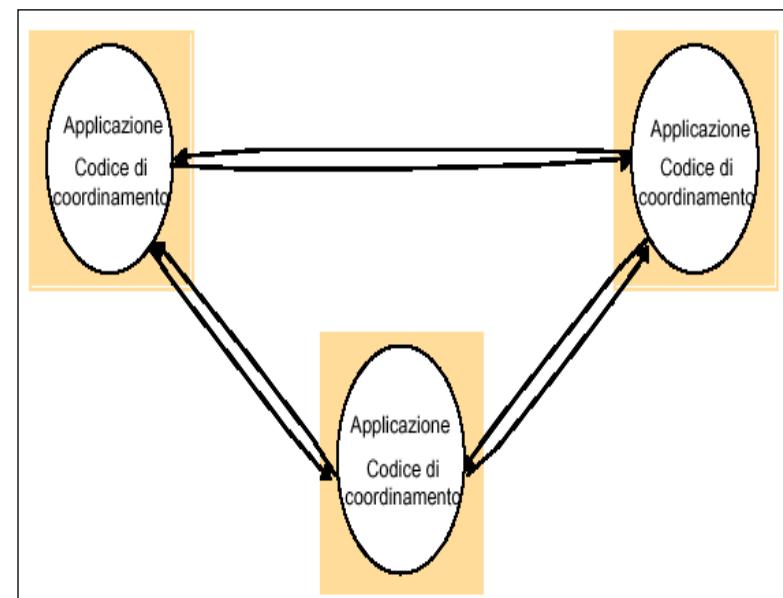
# Types of transparency

- **Failure transparency:** enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components
- **Migration Transparency:** resources and clients must be free to move from one location to another without affecting the operations or requiring their names changed.  
E.g., /usr/lee, /central/usr/lee
- **Performance transparency** allows the system to be reconfigured to improve performance as loads vary.
- **Scaling transparency** allows the system and applications to expand in scale without change to the system structure or the application algorithms.
- **Parallelism Transparency:** Automatic use of parallelism without having to program explicitly. The holy grail for distributed and parallel system designers.
- **Users do not always want complete transparency**

# Interaction Models

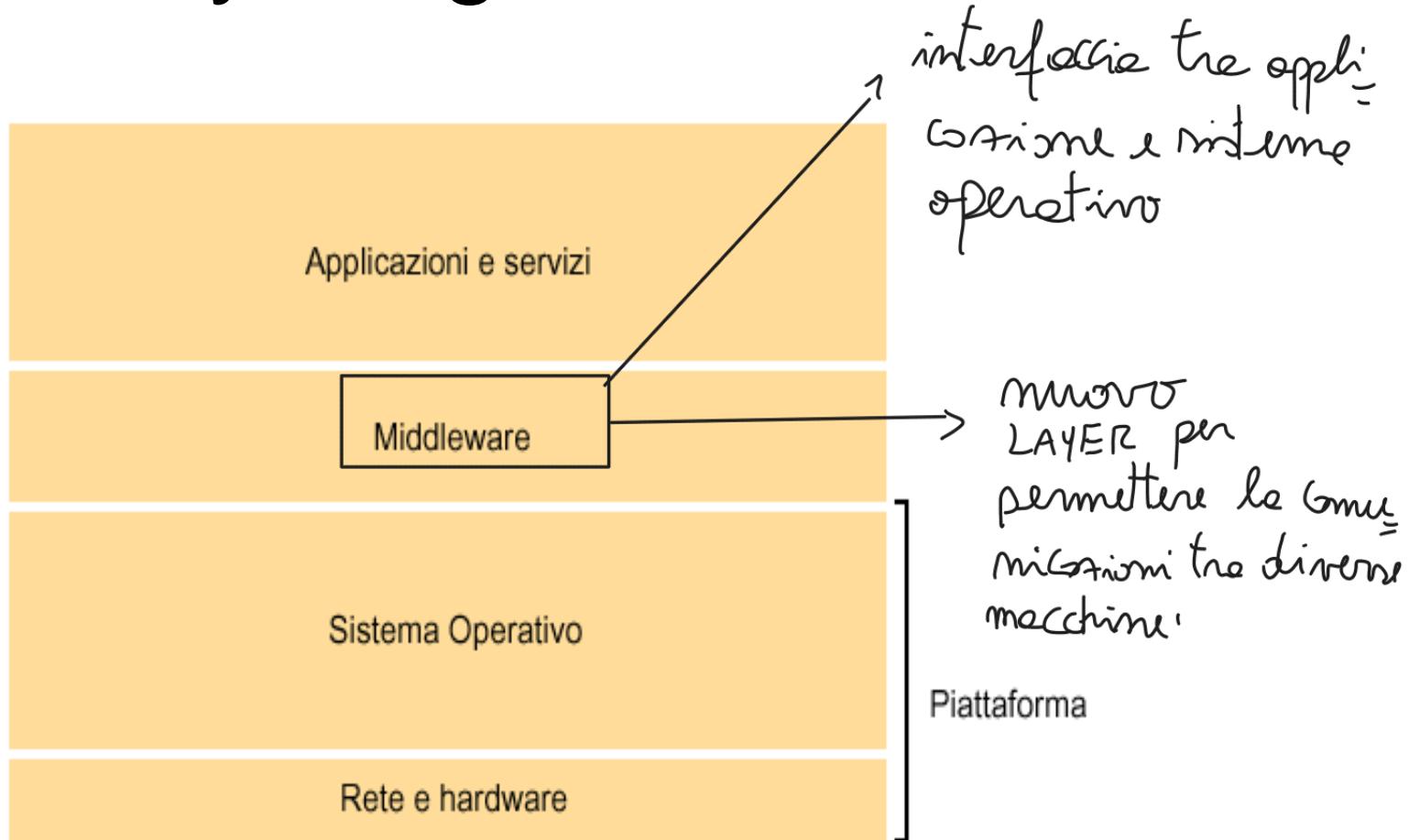


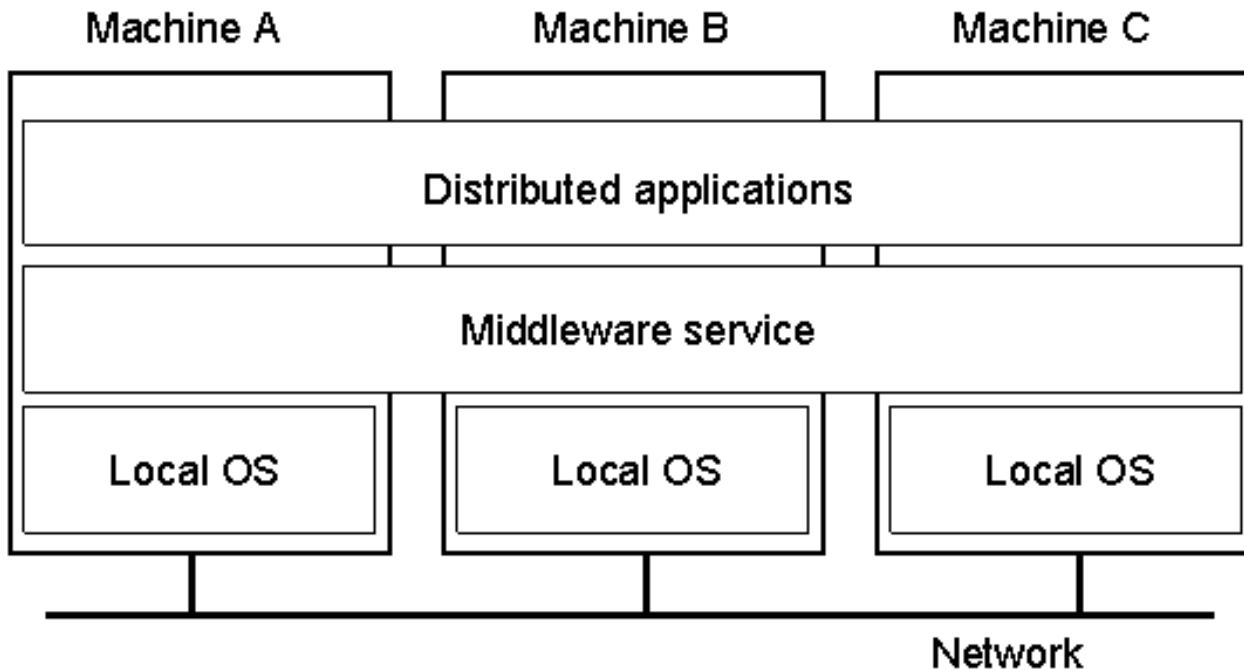
client/server



peer-to-peer

# Layering hw and sw





A distributed system organized as middleware.

Note that the middleware layer extends over multiple machines.

# Middleware problems

point struct

- **Heterogeneity:** OS, clock speeds, **data representation**, memory, architecture HW
- **Local Asynchrony:** load on a node, different HW, Interrupts
- **Lack of global knowledge:** knowledge propagates through messages whose propagation time will be much slower than the time taken by the execution of an internal event
- **Network Asynchrony:** propagation times of message can be unpredictable
- **Failures of nodes or network partitions**
- **Lack of a global order of events**
- **Consistency vs Availability vs Network Partitions**

This limits the set of problems that can be solved through deterministic algorithms on some distributed systems

# Middleware

- To achieve the true benefits of the distributed system, developers must have a set of tools that provide a uniform means and style of access to system resources across all platforms
- This would enable programmers to build applications that look and feel the same
- Enable programmers to use the same method to access data regardless of the location of that data
- The way to meet this requirement is by the use of standard programming interfaces and protocols that sit between the application (above) and communications software and operating system (below)

# Logical View of Middleware

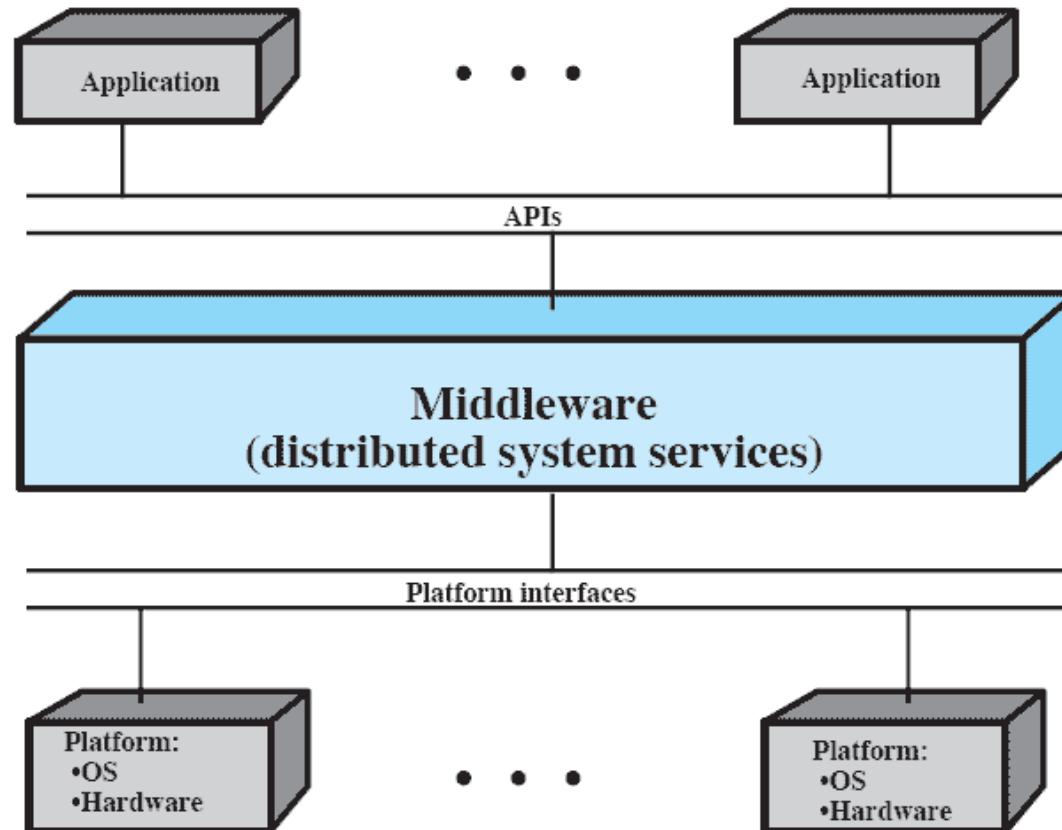


Figure 16.9 Logical View of Middleware

# **CLIENT/SERVER COMPUTING**

# Client/Server Computing

- Client machines are generally single-user PCs or workstations that provide a highly user-friendly interface to the end user
- Each server provides a set of shared services to the clients
- The server enables many clients to share access to the same database and enables the use of a high-performance computer system to manage the database

# Generic Client/Server Environment

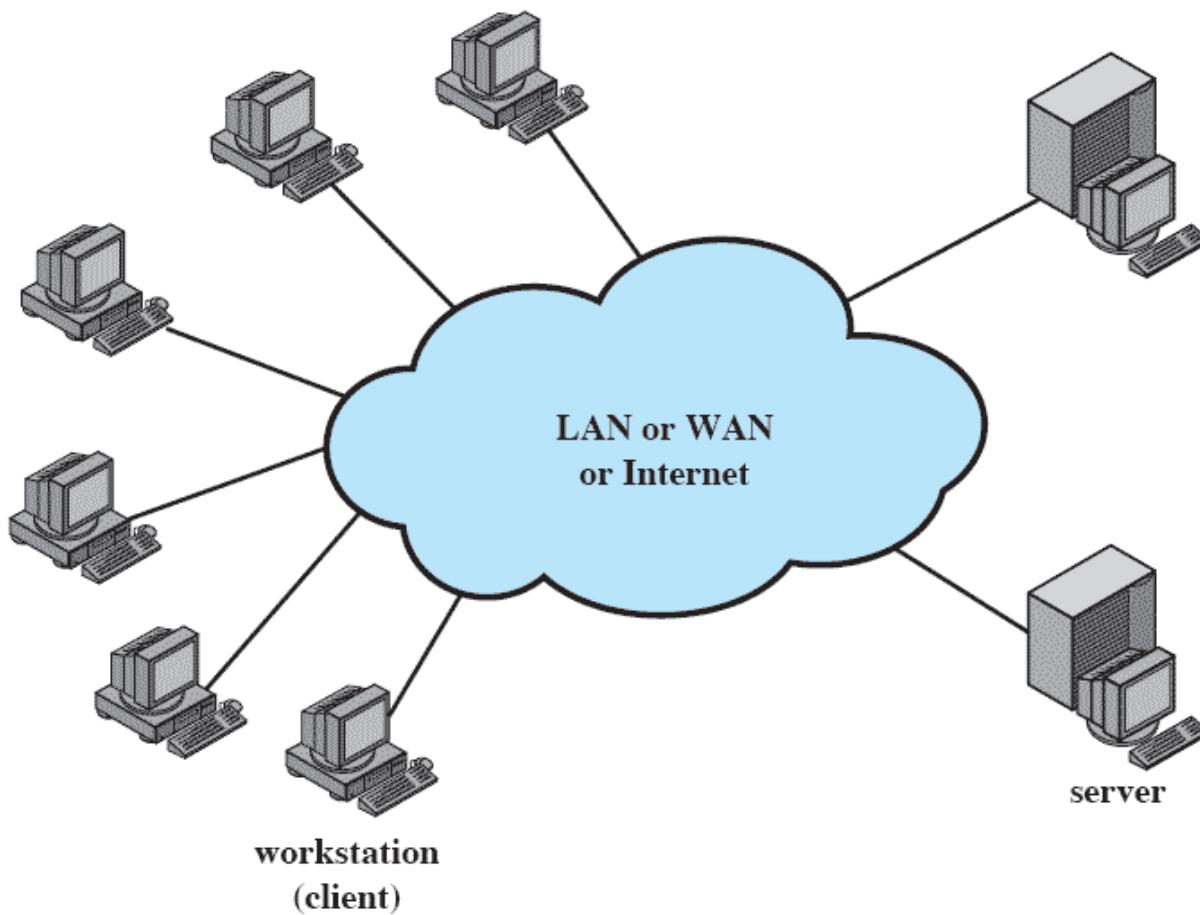


Figure 16.1 Generic Client/Server Environment

# Client/Server Characteristics

- A client/server configuration differs from other types of distributed processing:
  - there is a heavy reliance on bringing user-friendly applications to the user on his or her own system
  - there is an emphasis on centralizing corporate databases and many network management and utility functions
  - there is a commitment, both by user organizations and vendors, to open and modular systems
  - networking is fundamental to the operation

# Client/Server Applications

- Bulk of applications software executes on the server
- Application logic is located at the client
- Presentation services in the client

# Client/Server Applications

- The key feature of a client/server architecture is the allocation of application-level tasks between clients and servers
- Hardware and the operating systems of client and server may differ
- These lower-level differences are irrelevant as long as a client and server share the same communications protocols and support the same applications

# Client/Server Applications

- It is the communications software that enables client and server to interoperate
  - » principal example is TCP/IP
- Actual functions performed by the application can be split up between client and server in a way that optimizes the use of resources
- The design of the user interface on the client machine is critical
  - » there is heavy emphasis on providing a graphical user interface (GUI) that is easy to use, easy to learn, yet powerful and flexible

# Generic Client/Server Architecture

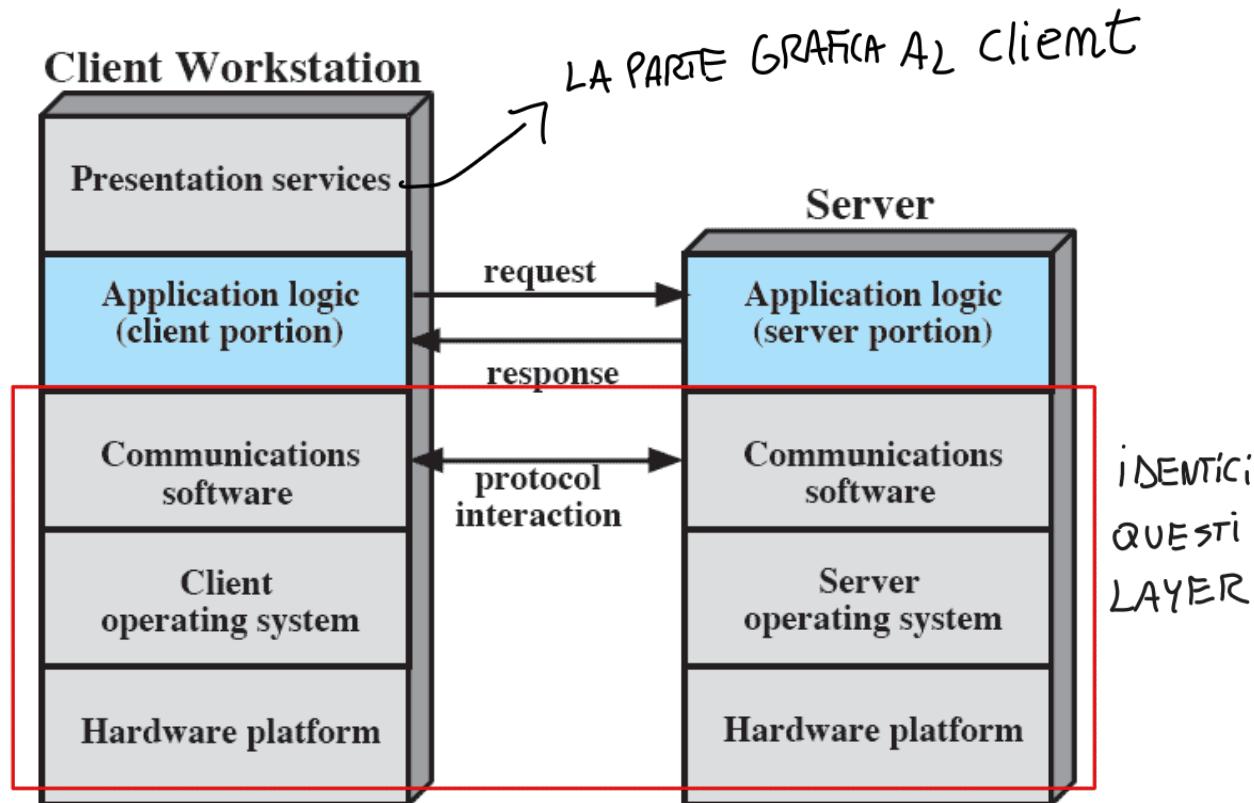


Figure 16.2 Generic Client/Server Architecture

# Distributed Application Components

- **Business-oriented applications** (payroll, order entry, customer tracking, inventory control, etc.) contain four general components:
  - **Presentation logic:** user interface.
  - **I/O processing logic:** data validation.
  - **Business processing logic:** business rules and calculations.
  - **Data storage logic:** constraints such as primary keys, referential integrity, and actual data retrieval.

While almost all applications contain those four general components, for any given application those components need not be part of the same program, resident on the same computer, written in the same language, nor written by the same group of programmers.

# Questions in application component development

- **Decisions to make:**
  - What language should a component be written in?
  - What hardware resource should a component reside upon?
- **Information needed to make the decision:**
  - How often will the component change?
    - Language changes.
    - Platform changes.
    - Business changes.
  - Who is responsible for maintaining the component?
  - How long is the application supposed to last?

# Role of Middleware in Client/Server Architecture

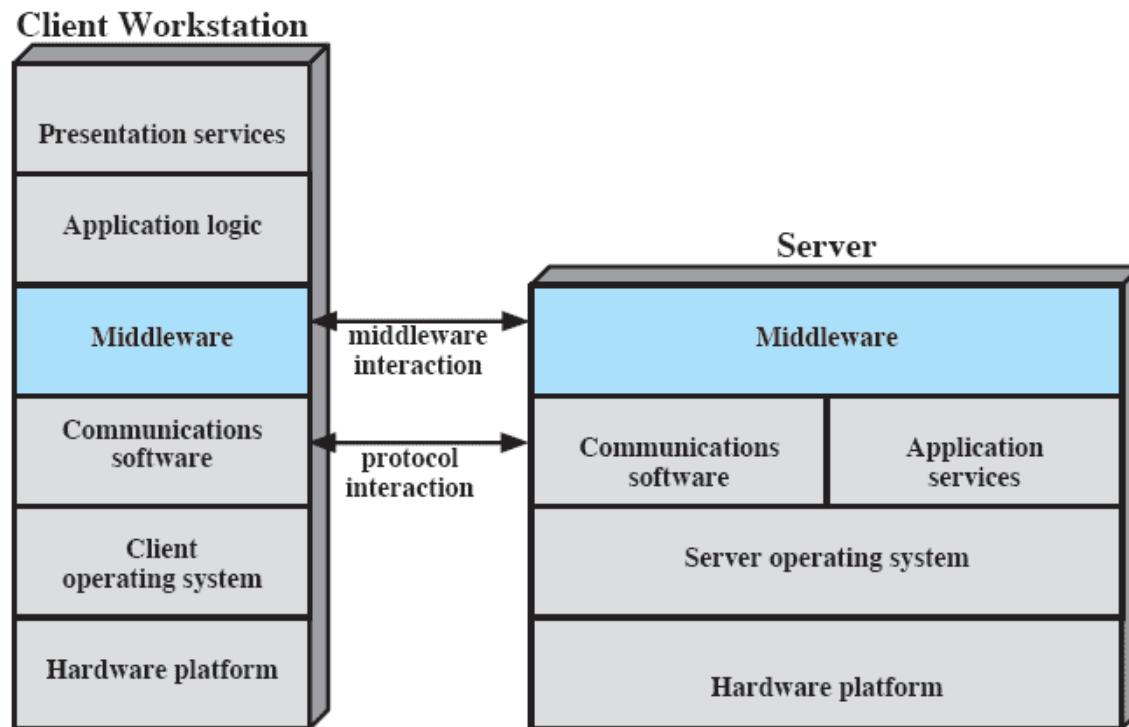


Figure 16.8 The Role of Middleware in Client/Server Architecture

# Classes of Client/Server Applications

Host-based processing

Dove faccio  
le operazioni?

↑  
Server-based processing

Four general classes are:

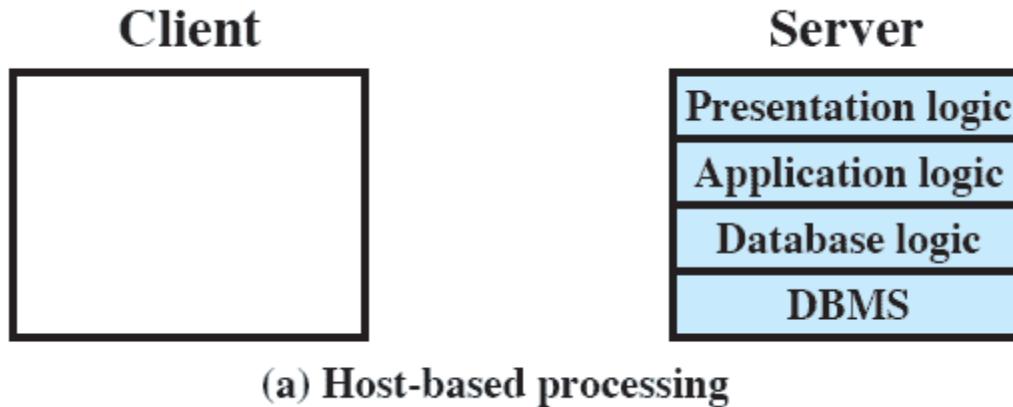
Cooperative processing



Client-based processing

# Classes of Client/Server Applications

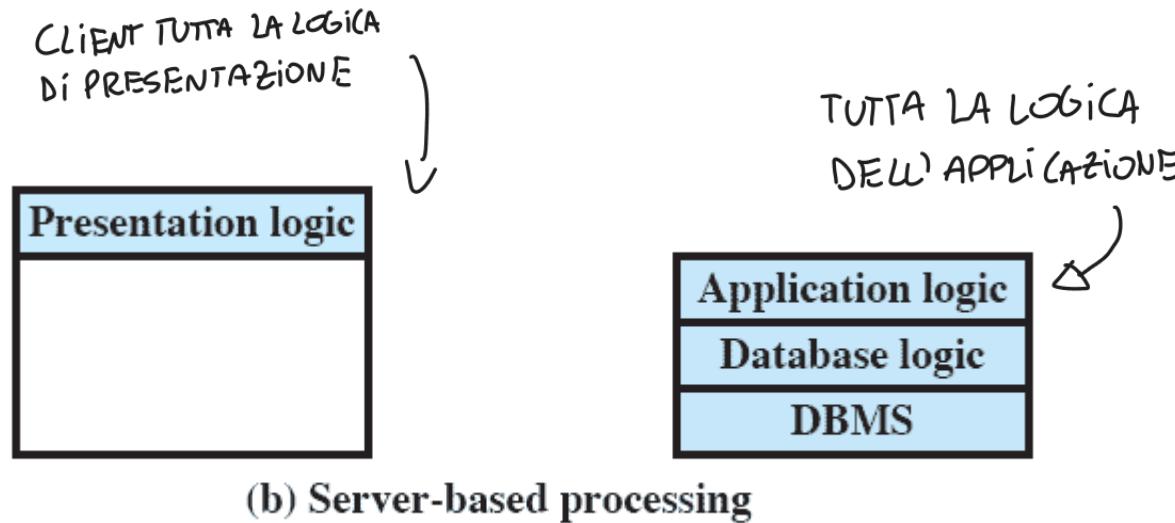
- Host-based processing
  - Not true client/server computing
  - Traditional mainframe environment



TUTTO AVviENE LATO SERVER ANCHE LOGICA  
DI PRESENTAZIONE, CLIENT METTE UNA MINIMA INTERFACCIA

# Classes of Client/Server Applications

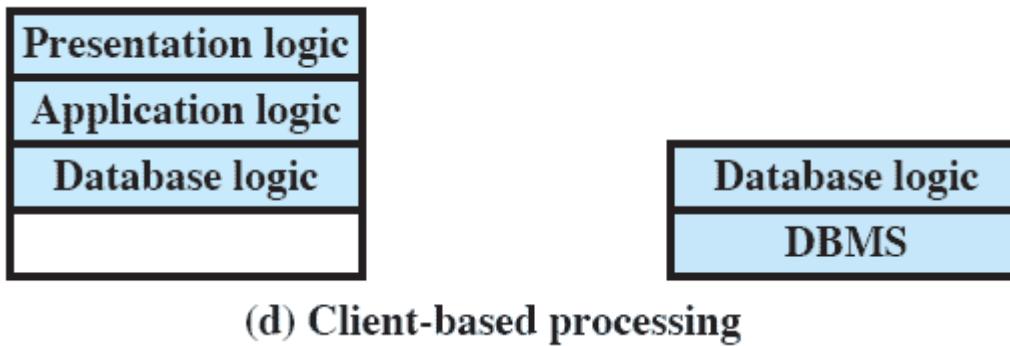
- **Server-based processing**
  - Server does all the processing
  - Client provides a graphical user interface



ES. BROWSER...

# Classes of Client/Server Applications

- **Client-based processing**
  - All application processing done at the client
  - Data validation routines and other database logic functions are done at the server



# Classes of Client/Server Applications

- Cooperative processing
  - Application processing is performed in an optimized fashion
  - Complex to set up and maintain
  - Fat vs Thin clients



(c) Cooperative processing

# Three-tier Client/Server Architecture

- Application software distributed among three types of machines
  - User machine
    - Thin client
  - Middle-tier server
    - Gateway
    - Convert protocols
    - Merge/integrate results from different data sources
  - Backend server

# Three-tier Client/Server Architecture

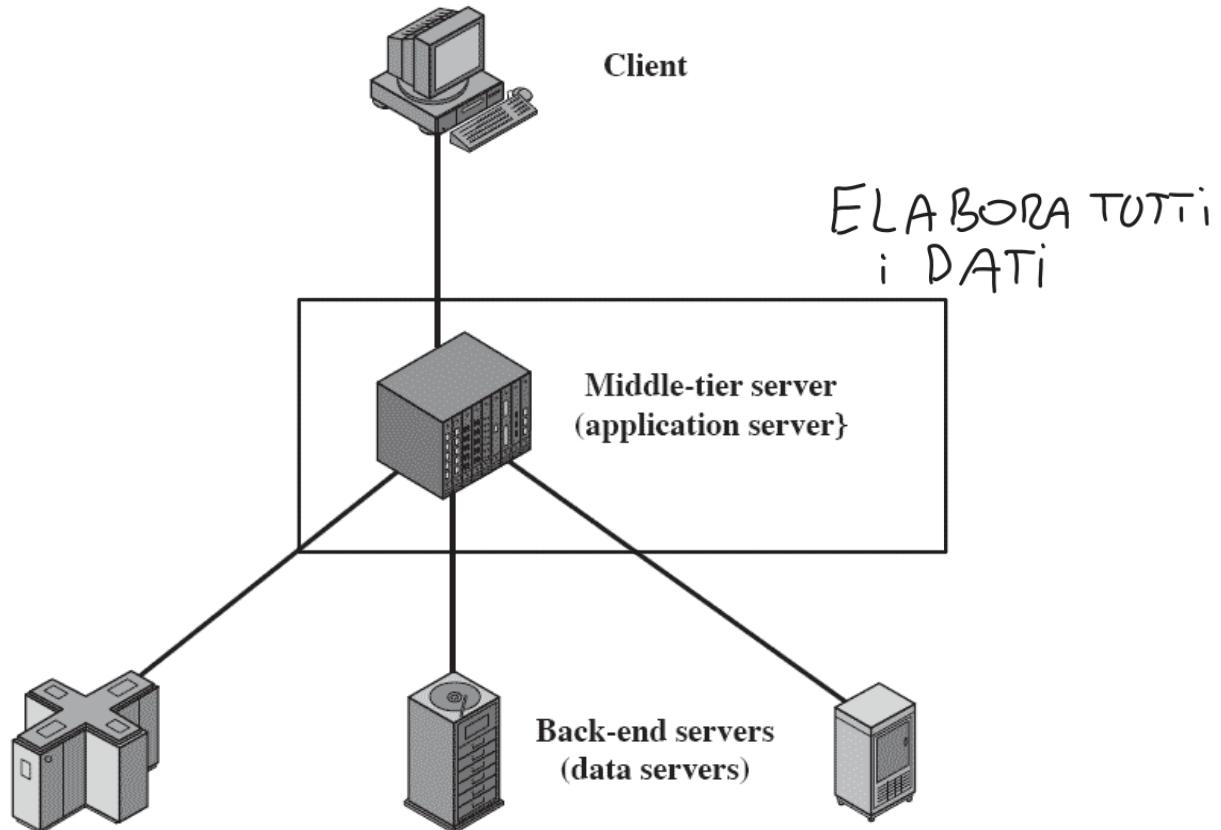


Figure 16.6 Three-tier Client/Server Architecture

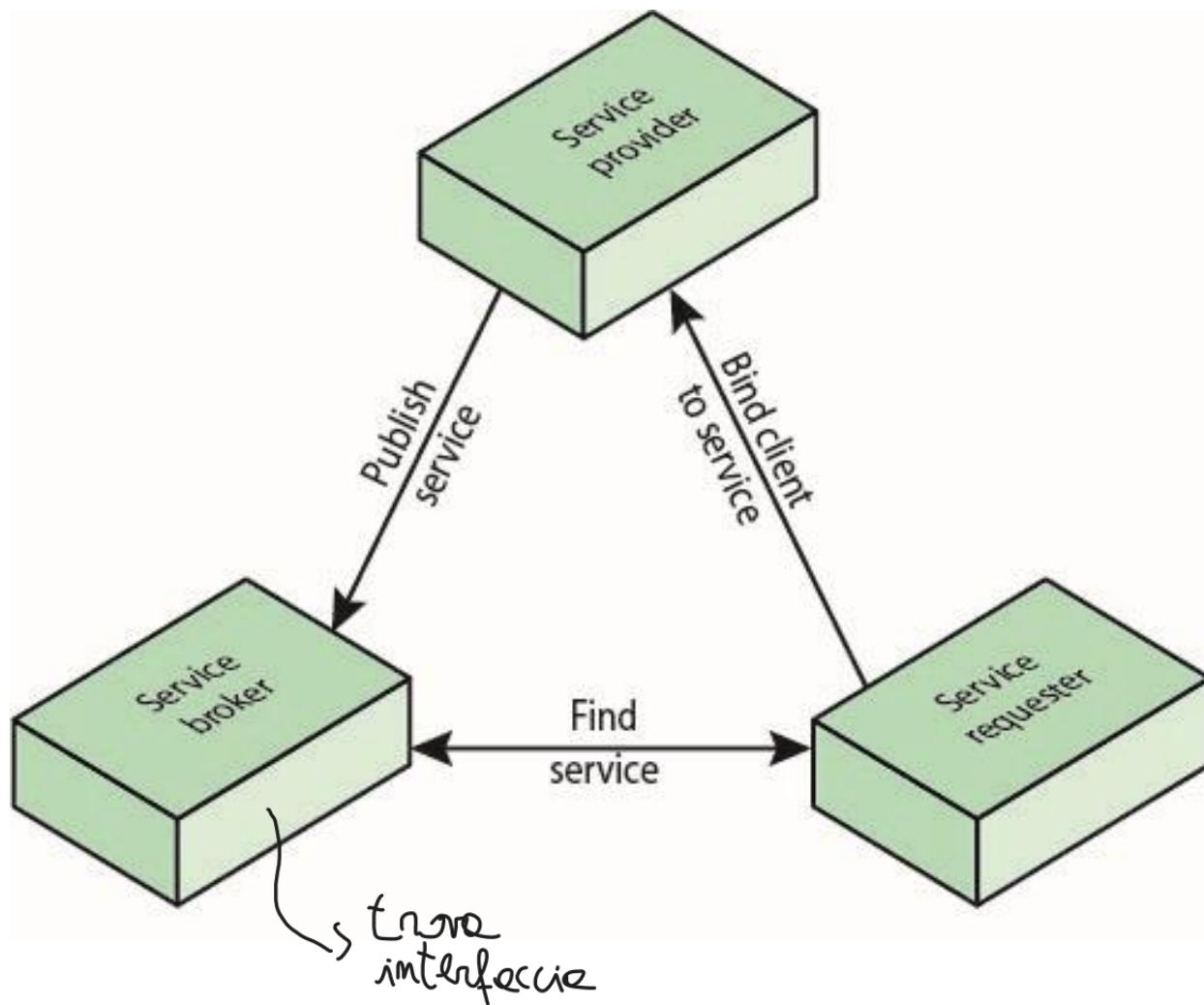


# Service-Oriented Architecture (SOA)

- A form of client/server architecture used in enterprise systems
- Organizes business functions into a modular structure rather than as monolithic applications for each department
  - as a result, common functions can be used by different departments internally and by external business partners as well
- **Consists of a set of services and a set of client applications that use these services**
- **Standardized interfaces are used to enable service modules to communicate with one another and to enable client applications to communicate with service modules**
  - most popular interface is **XML** (Extensible Markup Language) over **HTTP** (Hypertext Transfer Protocol), known as **Web services**

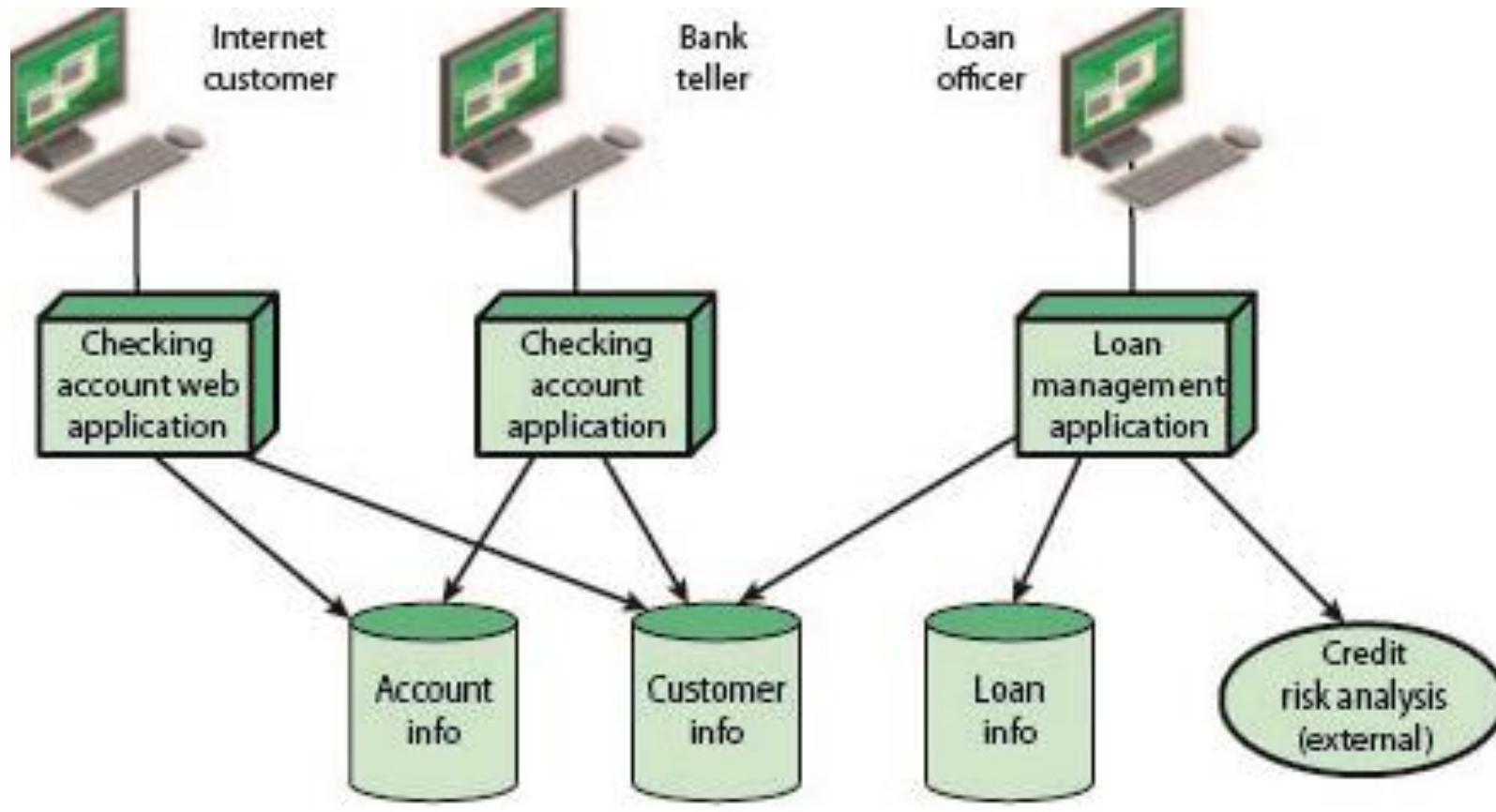
# SOA Model

Potranno essere anche di più ognuno di questi elementi



CONTO CORRENTE

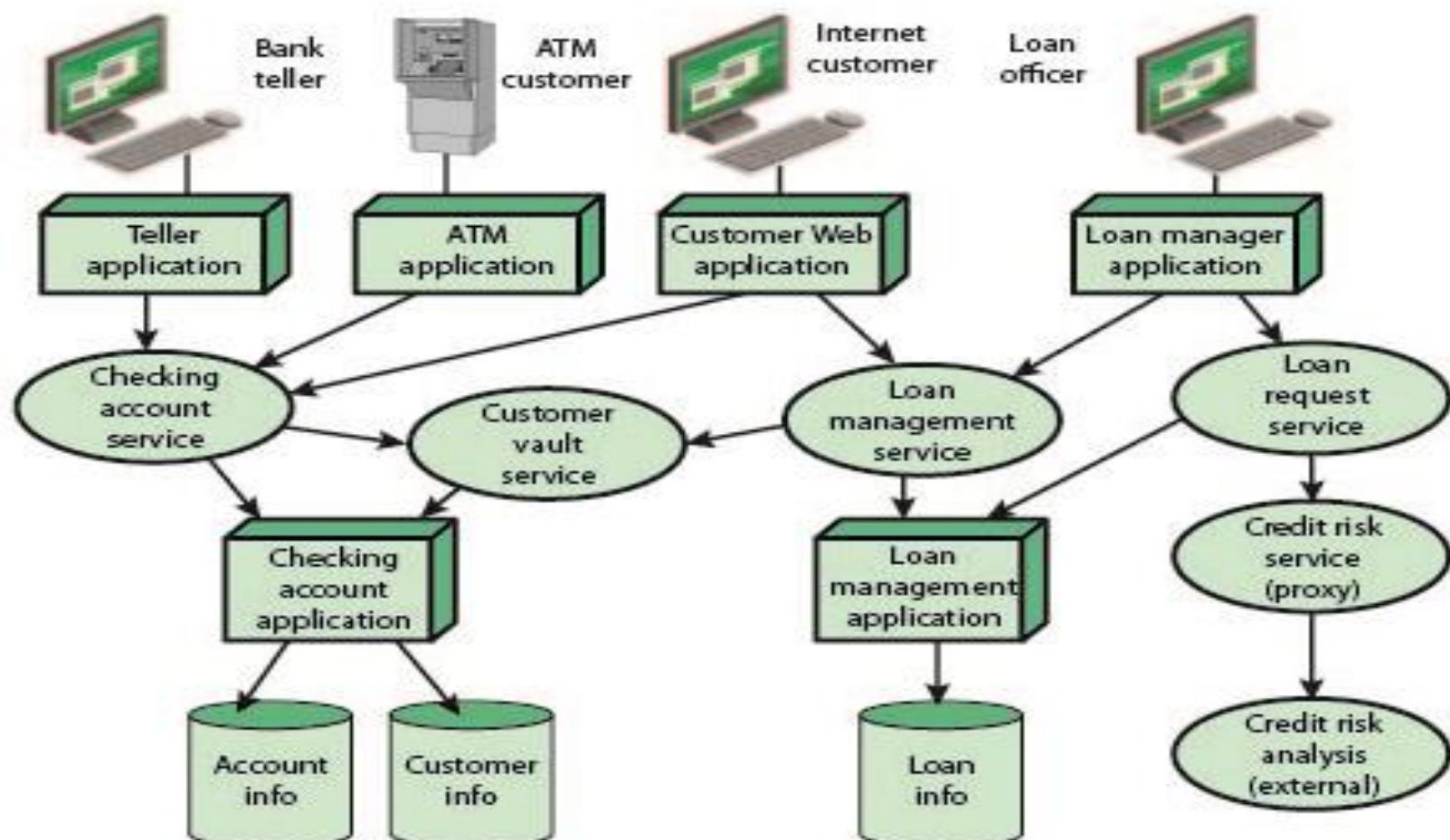
# Example Use of SOA



(a) Typical application structure

Il SOA può componenti per più applicazioni, offre e richiede servizi

# Example Use of SOA

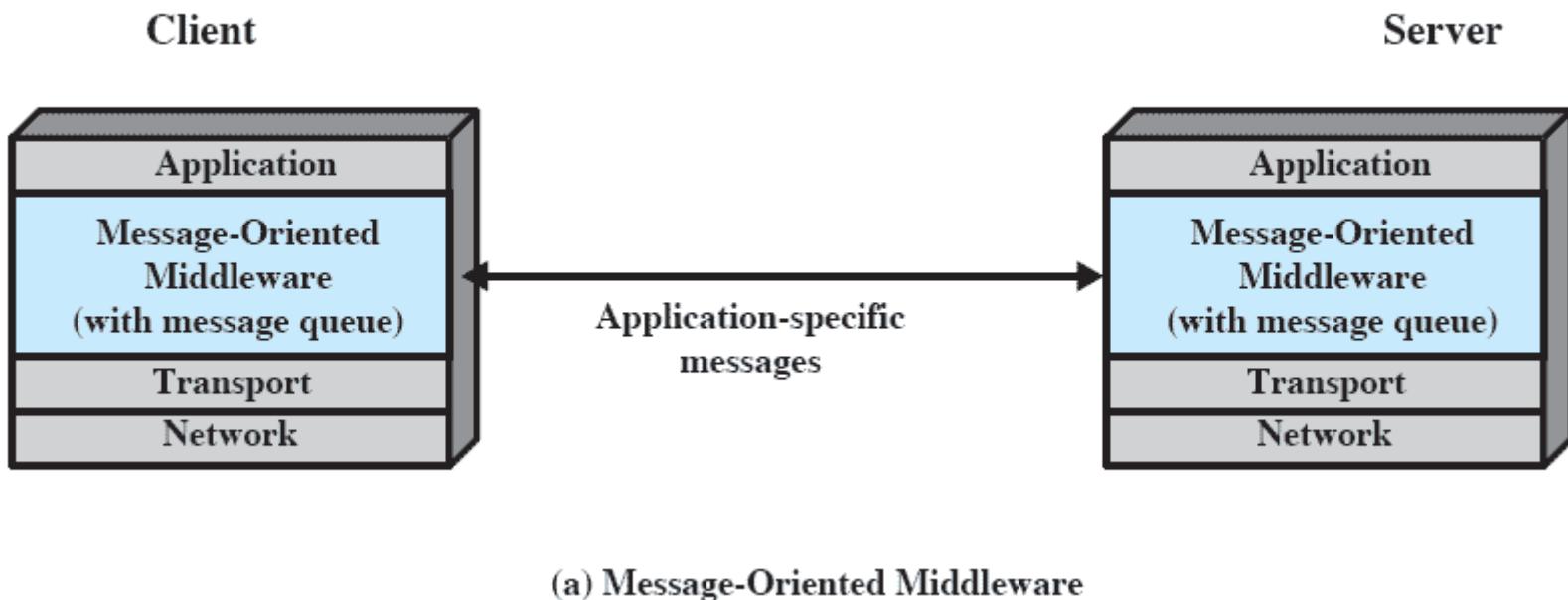


(b) An architecture reflecting SOA principles



# MESSAGE PASSING

# Distributed Message Passing



# Basic Message-Passing Primitives

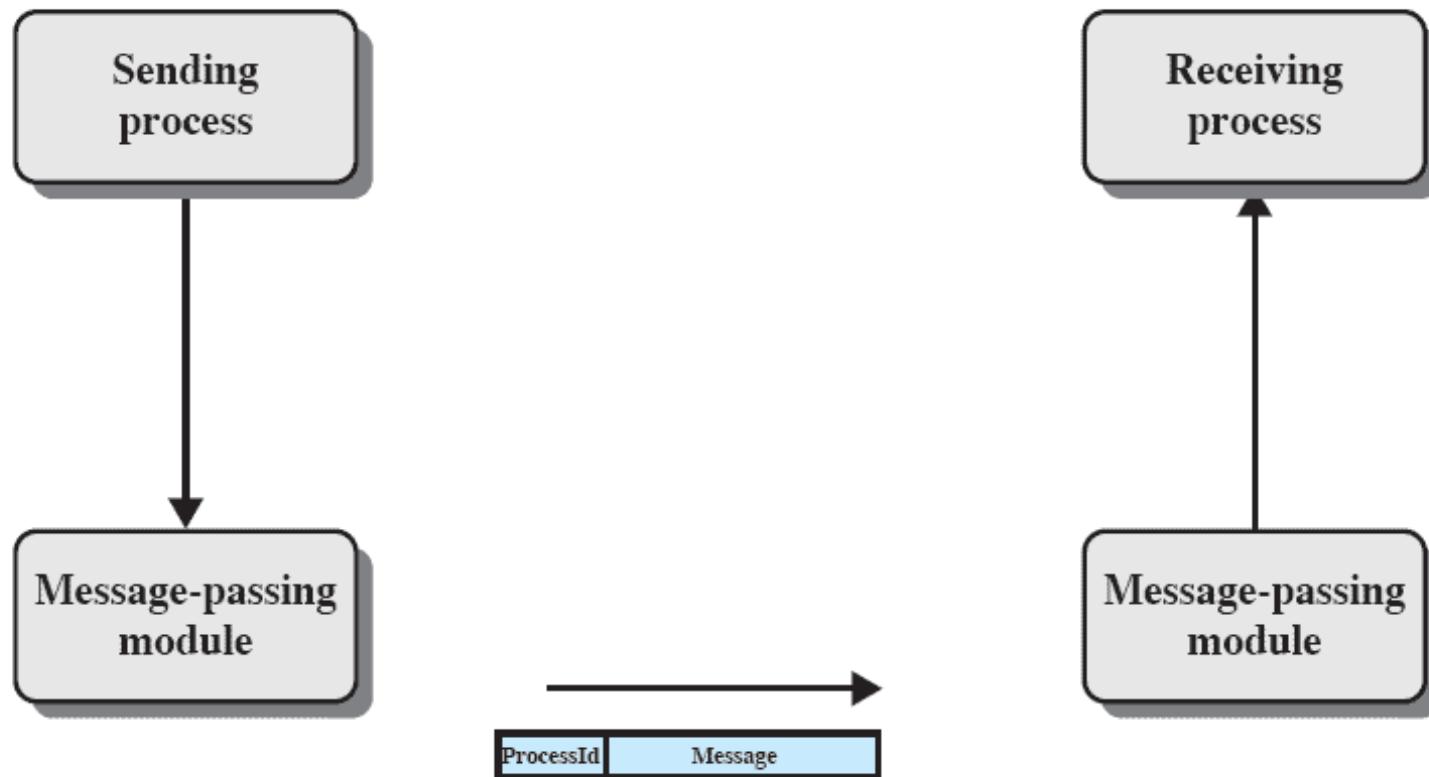


Figure 16.13 Basic Message-Passing Primitives

# REMOTE PROCEDURE CALLS

(API)



# Remote Procedure Calls

- Allow programs on different machines to interact using simple procedure call/return semantics
- Used for access to remote services
- Widely accepted and common method for encapsulating communication in a distributed system

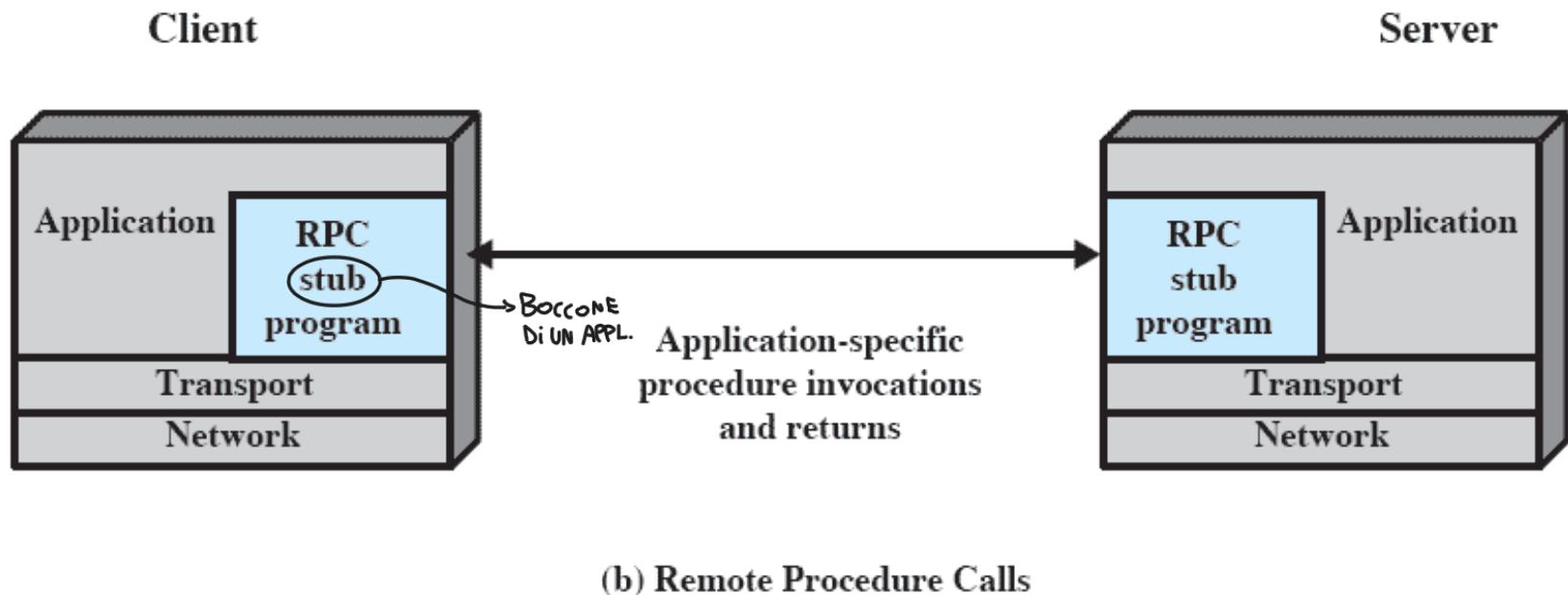
# RPC is standard technology

*Tutti possono usare*

## Advantages of standardization

- the communication code for an application can be generated automatically
- client and server modules can be moved among computers and OSs with little modification and recoding

# Remote Procedure Call Architecture



# Remote Procedure Call Mechanism

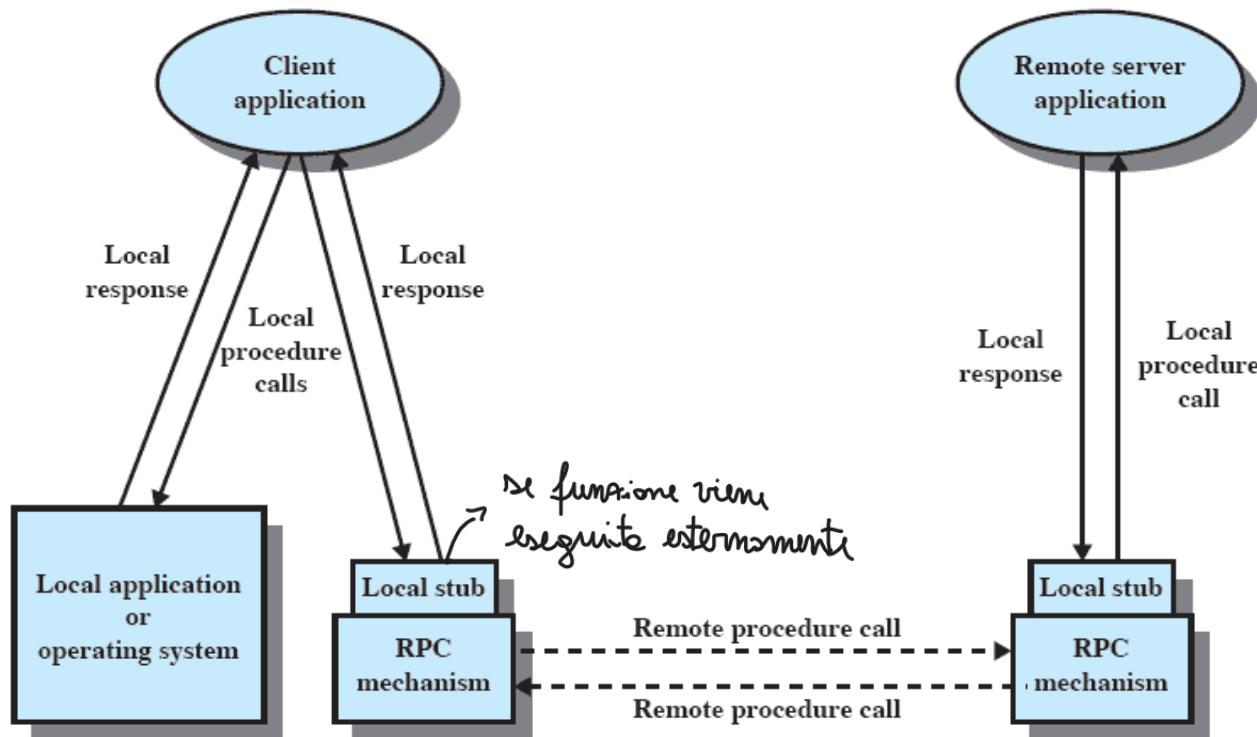


Figure 16.14 Remote Procedure Call Mechanism

# Parameter Passing/ Parameter Representation

Vengono usate come funzioni locali:

- Passing a parameter by **value** is easy
- Passing by **reference** is more difficult
  - a unique system wide pointer is necessary
  - the overhead for this capability may not be worth the effort
- The representation/format of the parameter and message may be difficult if the programming languages differ between client and server

# Client/Server Binding

A binding is formed when two applications have made a logical connection and are prepared to exchange commands and data

## Nonpersistent Binding

- Nonpersistent binding means that a logical connection is established between the two processes at the time of the remote procedure call and that as soon as the values are returned, the connection is dismantled
- The overhead involved in establishing connections makes nonpersistent binding inappropriate for remote procedures that are called frequently by the same caller

# Client/Server Binding

## Persistent Binding

- A connection that is set up for a remote procedure call is sustained after the procedure return
- The connection can then be used for future remote procedure calls
- If a specified period of time passes with no activity on the connection, then the connection is terminated
- For applications that make many repeated calls to remote procedures, persistent binding maintains the logical connection and allows a sequence of calls and returns to use the same connection

# Synchronous versus Asynchronous

## Synchronous RPC

Come una chiamata  
a funzione, blocca fino al **return**

- behaves much like a subroutine call
- behavior is predictable
- however, it fails to exploit fully the parallelism inherent in distributed applications
- this limits the kind of interaction the distributed application can have, resulting in lower performance

# Synchronous versus Asynchronous

## Asynchronous RPC

- does not block the caller
- replies can be received as and when they are needed
- allow client execution to proceed locally in parallel with server invocation

# RPC Call Semantics

Normal RPC functioning may get disrupted

- call or response message is lost
- caller node crashes and is restarted
- callee node crashes and is restarted

The call semantics determines how often the remote procedure may be executed under fault conditions

# At least once

- This semantics guarantees that the call is executed one or more times but does not specify which results are returned to the caller.
- Very little overhead & easy to implement
  - using timeout based retransmission without considering orphan calls (i.e., calls on server machines that have crashed)
  - the client machine continues to send call requests to the server machine until it gets an acknowledgement. If one or more acknowledgements are lost, the server may execute the call multiple times
- Works only for idempotent operations

STESSE OPERAZIONI = STESSO RISULTATO

SERVER elabora solo le prime richieste che arrivano.

# At most once

- This semantics guarantees that the RPC call is executed at most once
  - either it does not execute at all or it executes exactly once depending on whether the server machine goes down
- Unlike the previous semantics, this semantics require the detection of duplicate packets, but works for non-idempotent operations.

# Exactly once

- The RPC system guarantees the *local* call semantics assuming that a server machine that crashes will eventually restart.
- It keeps track of orphan calls and allows them to later be adopted by a new server
- Requires a very complex implementation!

# **CLUSTERS**

sistemi olistribuiti, dove i server  
sono costituiti da "cluster" di macchine

# Clusters

- Alternative to symmetric multiprocessing (SMP) as an approach to providing high performance and high availability
- Group of interconnected, whole computers working together as a unified computing resource that can create the illusion of being one machine
- ***Whole computer*** means a system that can run on its own, apart from the cluster
- Each computer in a cluster is called a ***node***

# Benefits of Clusters

## Absolute scalability

it is possible to create large clusters that far surpass the power of even the largest stand-alone machines

## Incremental scalability

configured in such a way that it is possible to add new systems to the cluster in small increments

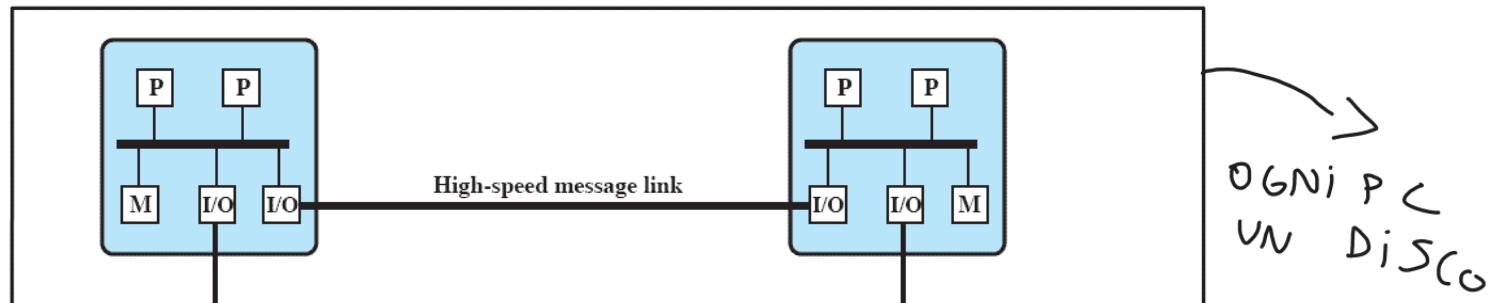
## High availability

failure of one node is not critical to system

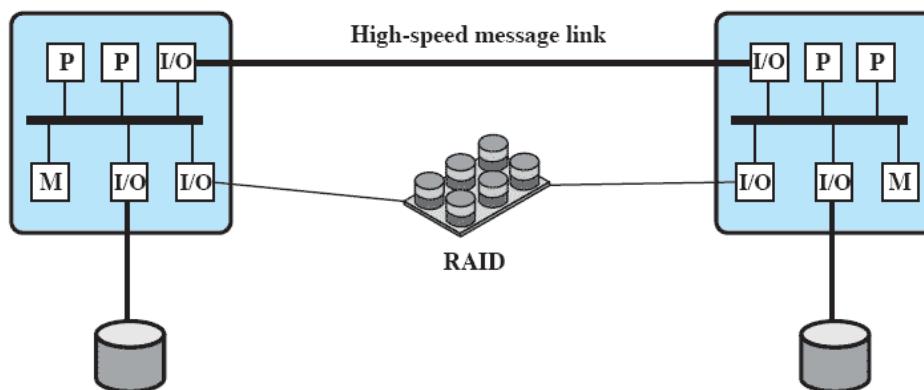
## Superior price/performance

by using commodity building blocks, it is possible to put together a cluster at a much lower cost than a single large machine

# Cluster Configurations



(a) Standby server with no shared disk



(b) Shared disk

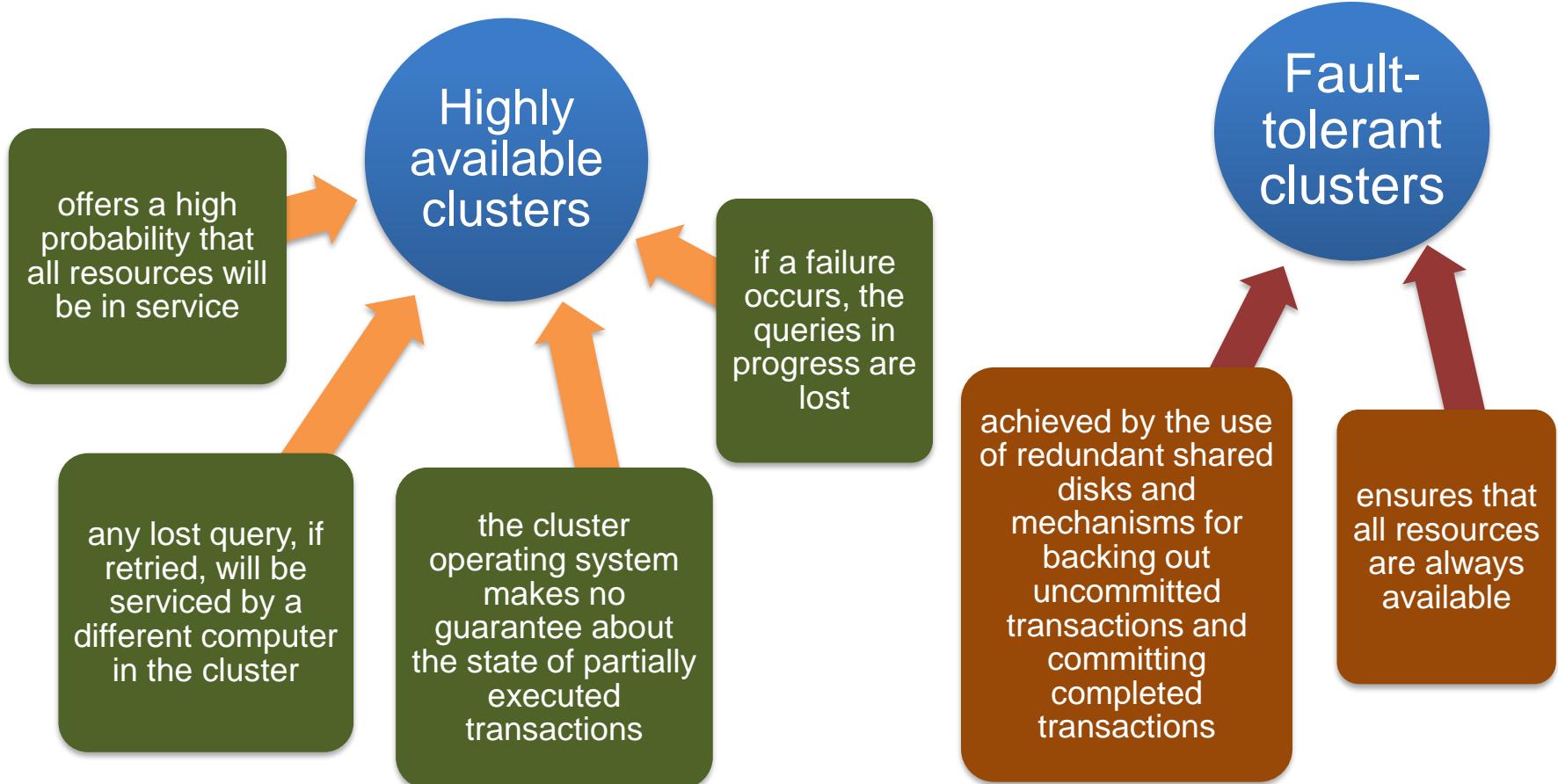
# Clustering Methods: Benefits and Limitations

POSSIAMO CLASSIFICARE ALCUNI PC COME SECONDARI, RSERVE DI QUELLI PRIMARI

| Clustering Method  | Description  | Benefits  | Limitations  |
|--|--|---|--|
| <b>Passive Standby</b><br>RIMANGONO INUTILIZZATI I PC SECONARI | A secondary server takes over in case of primary server failure.   | Easy to implement.  | High cost because the secondary server is unavailable for other processing tasks.          |
| <b>Active Secondary</b>  | The secondary server is also used for processing tasks.  | Reduced cost because secondary servers can be used for processing.                | Increased complexity.  |
| Separate Servers   | Separate servers have their own disks. Data is continuously copied from primary to secondary server.                                     | High availability.  | High network and server overhead due to copying operations.                                |
| Servers Connected to Disks                                     | Servers are cabled to the same disks, but each server owns its disks. If one server fails, its disks are taken over by the other server. | Reduced network and server overhead due to elimination of copying operations.     | Usually requires disk mirroring or RAID technology to compensate for risk of disk failure. |
| Servers Share Disks  | Multiple servers simultaneously share access to disks.   | Low network and server overhead. Reduced risk of downtime caused by disk failure. | Requires lock manager software. Usually used with disk mirroring or RAID technology.       |

# OS Design Issues: Failure Management

Two approaches can be taken to deal with failures:



# OS Design Issues: Failure Management

- The function of switching an application and data resources over from a failed system to an alternative system in the cluster is referred to as ***failover***
- The restoration of applications and data resources to the original system once it has been fixed is referred to as ***fallback***
- Fallback can be automated but this is desirable only if the problem is truly fixed and unlikely to recur
- Automatic fallback can cause subsequently failed resources to bounce back and forth between computers, resulting in performance and recovery problems

# Load Balancing

- A cluster requires an effective capability for balancing the load among available computers
- This includes the requirement that the cluster be incrementally scalable
- When a new computer is added to the cluster, the load-balancing facility should automatically include this computer in scheduling applications
- Middleware must recognize that services can appear on different members of the cluster and may migrate from one member to another

# Parallelizing Computation

## Parallelizing compiler

- determines, at compile time, which parts of an application can be executed in parallel
- performance depends on the nature of the problem and how well the compiler is designed

## Parallelized application

- the programmer designs the application to run on a cluster and uses message passing to move data, as required, between cluster nodes
- this places a high burden on the programmer but may be the best approach for exploiting clusters for some applications

## Parametric computing

- this approach can be used if an application is an algorithm or program that must be executed a large number of times, each time with a different set of starting conditions or parameters
- for this approach to be effective, parametric processing tools are needed to organize, run, and manage the jobs in an orderly manner

# Cluster Computer Architecture

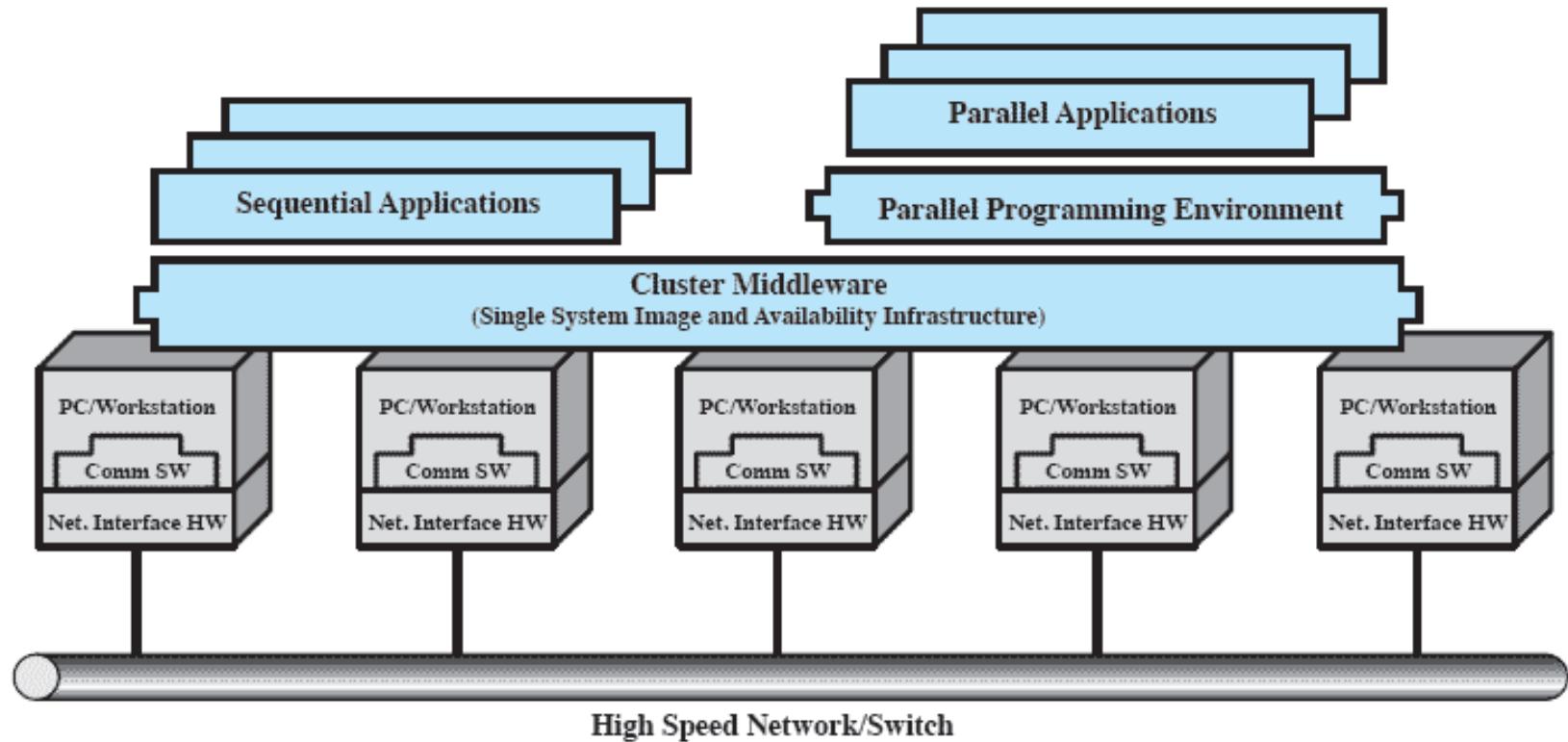


Figure 16.14 Cluster Computer Architecture [BUY99a]

# Internet of Things

# What is the IOT?

■ Don't know / not sure / no idea

■ Everything / items connected

■ Security of using the internet

■ User guide to the internet / things involving the internet

■ Other

■ To find/ do things / everything on the internet

■ Shopping

■ Library / directory

■ Every day items connected to the internet and can send / receive data

■ Smart homes

■ Search engine

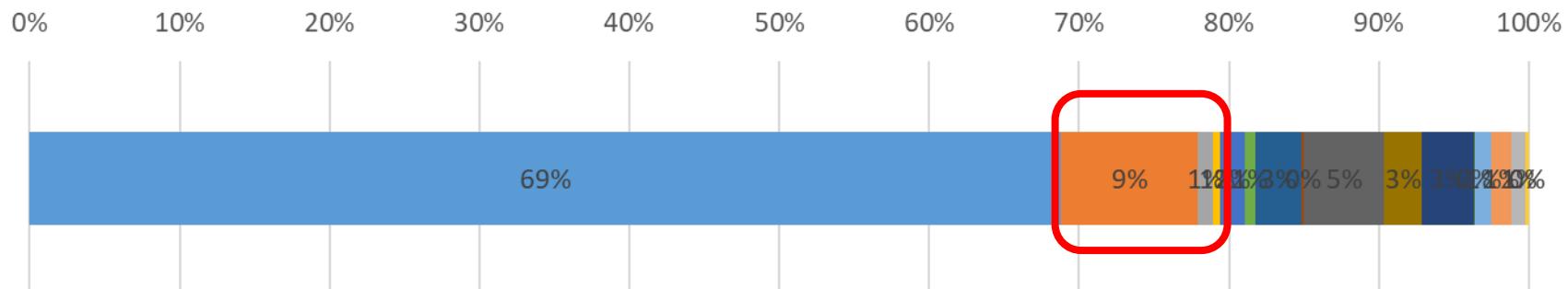
■ App

■ Things on the internet / internet of everything

■ Using the internet

■ Internet/ internet on time/ contents of internet

■ Data information



Taking in a couple of other descriptions which just about make sense, we hit around 11%.

# A whole IoT of headlines

■ Don't know / not sure / no idea

■ Smart homes

■ Security of using the internet

■ User guide to the internet / things involving the internet

■ Other

■ To find/ do things / everything on the internet

■ Shopping

■ Library / directory

■

■ Every day items connected to the internet and can send / receive data

■ Everything / items connected

■ Search engine

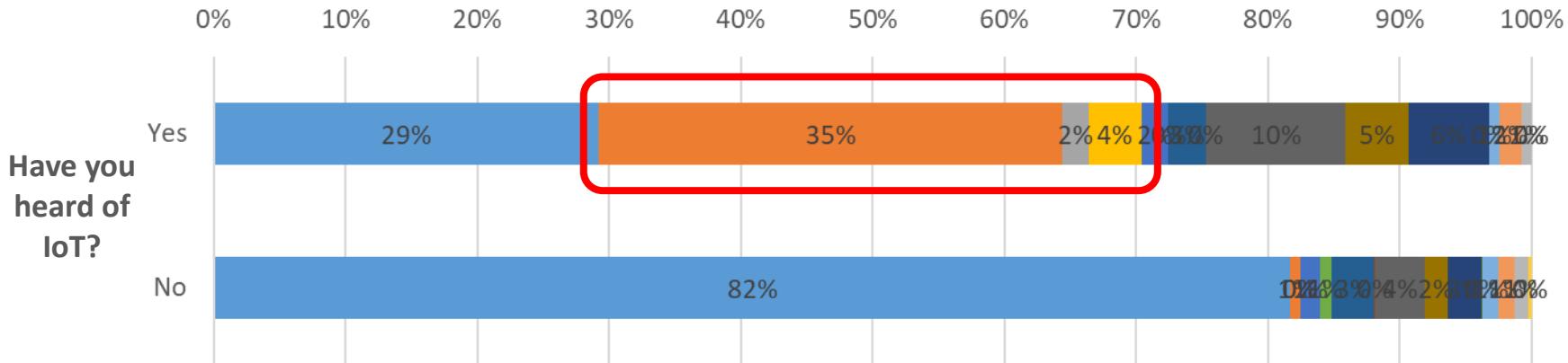
■ App

■ Things on the internet / internet of everything

■ Using the internet

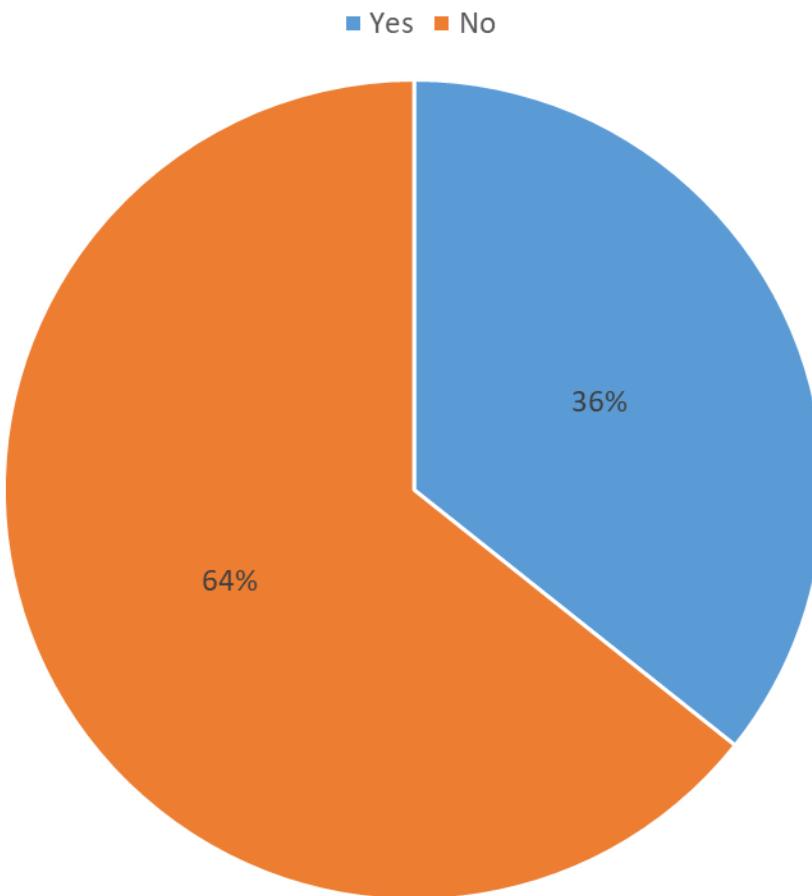
■ Internet/ internet on time/ contents of internet

■ Data information



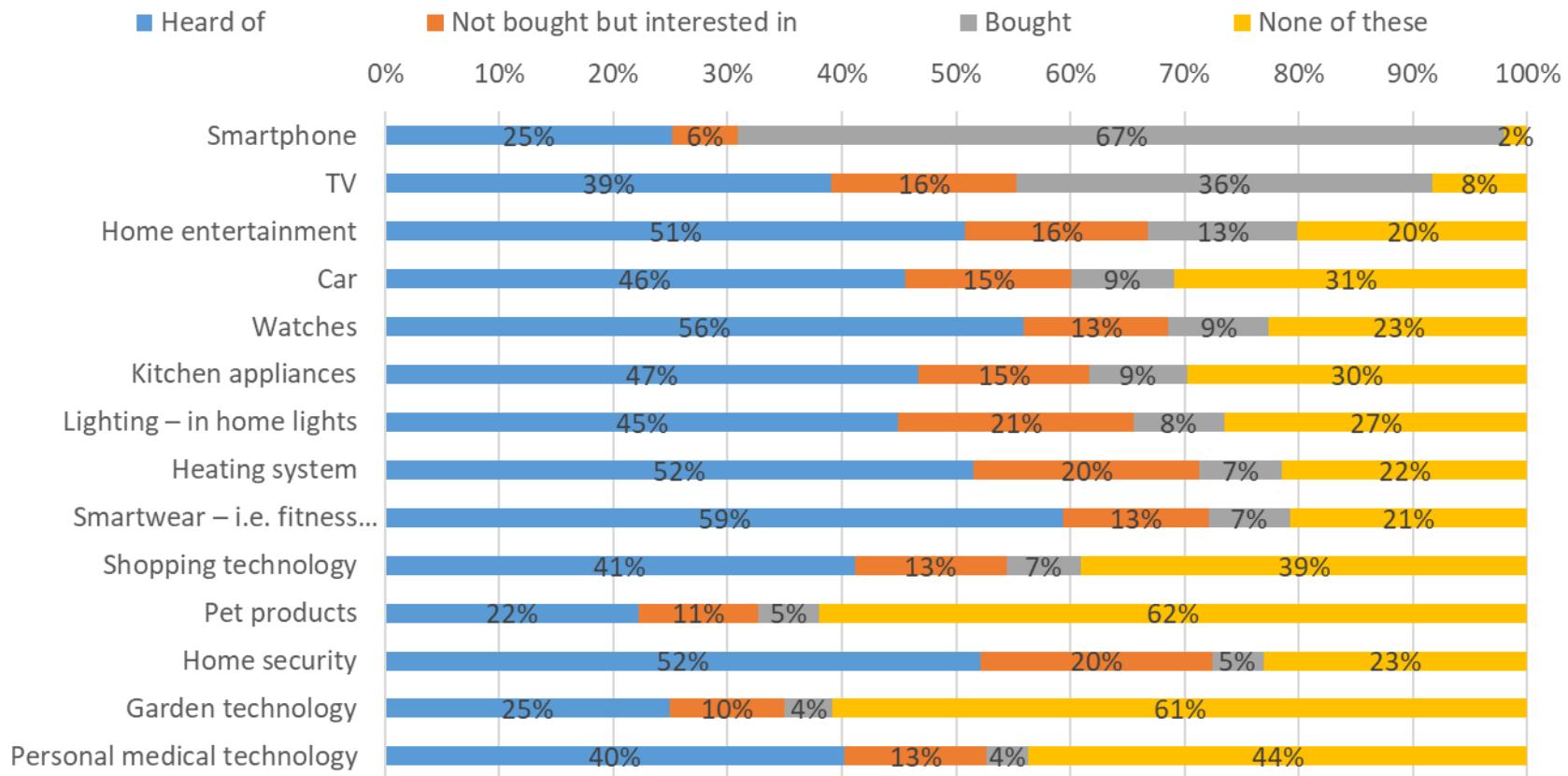
- Whilst tech writers are easing off the subject, there is a whole lot of consumer noise around the subject.
- Even those who claim to have already heard of IoT are likely to either get it wrong, or be blinded by the question.

# Have you bought any items that are ‘smart’?



N = 1000

# Smart phones no longer smart?



But there is plenty of headroom in the market.

# What is IoT?

- Latest development in the long and continuing revolution of computing and communications
- The term refers to:
  - expanding interconnection of smart devices
  - ranging from appliances to tiny sensors
  - Interconnection of billions of industrial and personal objects, usually through cloud systems
- What devices are in the IoT:
  - Personal computers
  - Smartphones
  - ....
  - But above all embedded devices
    - Low-bandwidth,
    - low-repetition data capture
    - low-bandwidth data-usage

# History of IoT

The concept of the Internet of Things first became popular in 1999, through the Auto-ID Center at MIT and related market-analysis publications.

Radio-frequency identification (RFID) was seen as a prerequisite for the IoT at that point.

If all objects and people in daily life were equipped with identifiers, computers could manage and inventory them.

Besides using RFID, the tagging of things may be achieved through such technologies as Near Field Communication, barcodes, QR codes, bluetooth, and digital watermarking.

# Evolution

## 1. Information technology (IT):

- PCs, servers, routers, firewalls, and so on
- bought as IT devices by enterprise IT people, primarily using wired connectivity.

## 2. Operational technology (OT):

- Machines/appliances with embedded IT built by non-IT companies, such as medical machinery, SCADA (supervisory control and data acquisition), process control, and kiosks
- bought as appliances by enterprise OT people, primarily using wired connectivity.

## 3. Personal technology:

- Smartphones, tablets, and eBook readers bought as IT devices by consumers (employees)
- exclusively using wireless connectivity and often multiple forms of wireless connectivity.

## 4. Sensor/actuator technology:

- Single-purpose devices bought by consumers, IT, and OT people
- exclusively using wireless connectivity, generally of a single form, as part of larger systems

# Components

- **Sensor**

- A sensor measures some parameter of a physical, chemical, or biological entity and delivers an electronic signal proportional to the observed characteristic, either in the form of an analog voltage level or a digital signal.
- In both cases, the sensor output is typically input to a microcontroller or other management element.

- **Actuator**

- An actuator receives an electronic signal from a controller and responds by interacting with its environment to produce an effect on some parameter of a physical, chemical, or biological entity.

- **Microcontroller**

- The “smart” in a smart device is provided by a deeply embedded microcontroller.

# Components

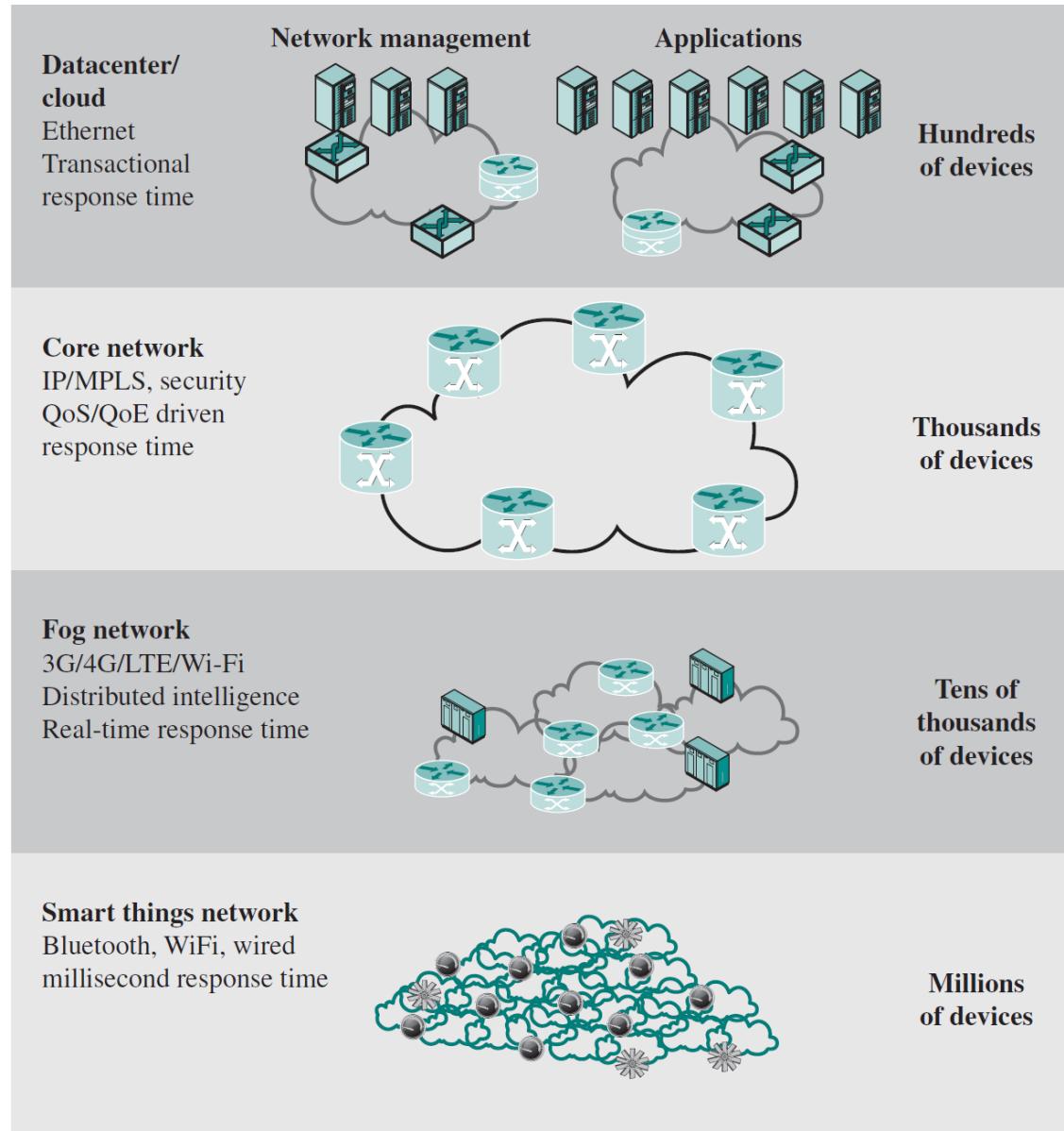
- **Transceiver**
  - A transceiver contains the electronics needed to transmit and receive data.
  - Most IoT devices contain a wireless transceiver, capable of communication using Wi-Fi, ZigBee, or some other wireless scheme.
- **Radio-Frequency Identification (RFID)**
  - Technology that uses radio waves to identify items, is increasingly becoming an enabling technology for IoT.
  - The main elements of an RFID system are tags and readers.
  - RFID tags are small programmable devices used for object, animal, and human tracking.
  - They come in a variety of shapes, sizes, functionalities, and costs.

# The Structure of IoT

The IoT can be viewed as a gigantic network consisting of networks of devices and computers connected through a series of intermediate technologies where numerous technologies like RFIDs, wireless connections may act as enablers of this connectivity.

- **Tagging Things** : Real-time item traceability and addressability by **RFIDs**.
- **Feeling Things** : **Sensors** act as primary devices to collect data from the environment.
- **Shrinking Things** : Miniaturization and **Nanotechnology** has provoked the ability of smaller things to interact and connect within the “things” or “smart devices.”
- **Thinking Things** : **Embedded intelligence** in devices through sensors has formed the network connection to the Internet. It can make the “things” realizing the intelligent control.

# IoT in the context of complete enterprise network



# IoT operating systems

- Constrained devices
  - very limited resources including limited RAM (kbytes) and ROM,
  - low-power requirements (batteries),
  - no memory management unit,
  - limited processor performance.
- We need ad-hoc operating systems
  - TinyOS
  - RIOT (open source)
  - μClinux

# IoT OS requirements

- **Small memory footprint**
  - Few available memory
  - We need to place both OS and applications
  - need for libraries optimized in terms of both size and performance, and space-efficient data structures.
- **Support for heterogeneous hardware**
  - For the largest systems, such as servers, PCs, and laptops, the Intel x86 processor architecture dominates.
  - For smaller systems, such as smartphones and a number of classes of IoT devices, the ARM architecture dominates.
  - Constrained devices are based on various microcontroller architectures and families, especially 8-bit and 16-bit processors.
  - A wide variety of communications technologies are also deployed on constrained devices.

# IoT OS requirements

- **Network connectivity**
  - IEEE 802.15.4 [low-rate wireless personal area network (WPAN)]
    - ZigBee
    - Bluetooth Low Energy (BLE)
  - 6LoWPAN (IPv6 over Low-power Wireless Personal Area Networks)
  - CoAP (Constrained Application Protocol)
  - RPL (Routing Protocol for Low power and Lossy Networks)
  - LoRaWAN (low-power Long-Range network technology)
- **Energy efficiency**
  - Need to work for years with a single battery
  - making the processor as energy efficient as possible
  - Wireless transmission schemes that minimize energy consumption
  - OS must provide energy consumption facilities
    - Provide them to higher level
    - Use them

# IoT OS requirements

- **Real-time capabilities**

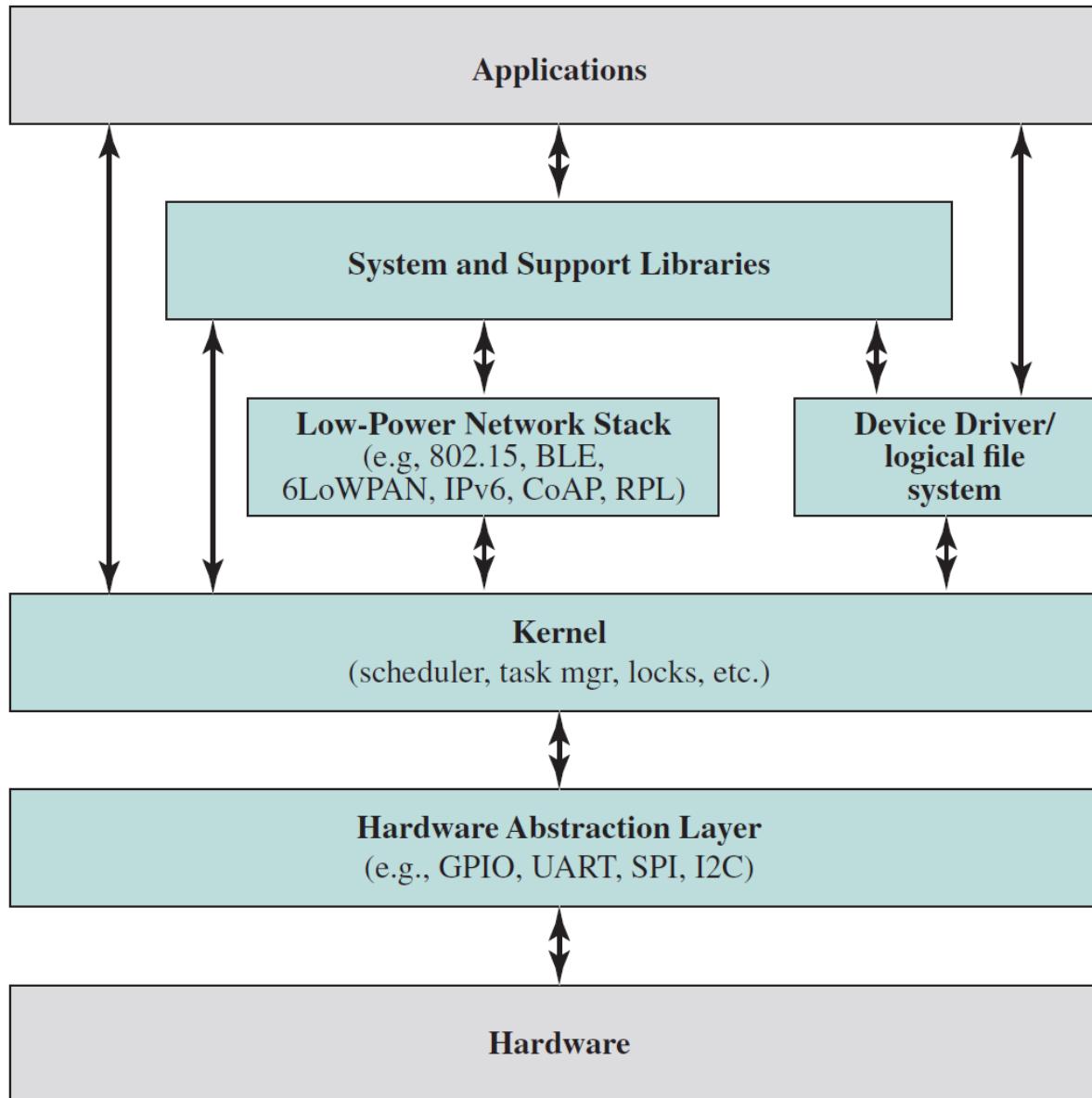
- Several devices must support real-time applications
  - Surveillance
  - Smart cars
- OS must fulfill timely execution requirements

- **Security**

- At application layer
  - Authorization, Authentication
  - Data confidentiality
  - Anti-virus
  - Privacy
- At network layer
  - Authorization, Authentication
  - Data confidentiality
  - Data integrity protection
- At device layer
  - Authorization, Authentication
  - Device integrity validation
  - Access control



# Typical structure for IoT OS



# Characteristics of famous IoT OS

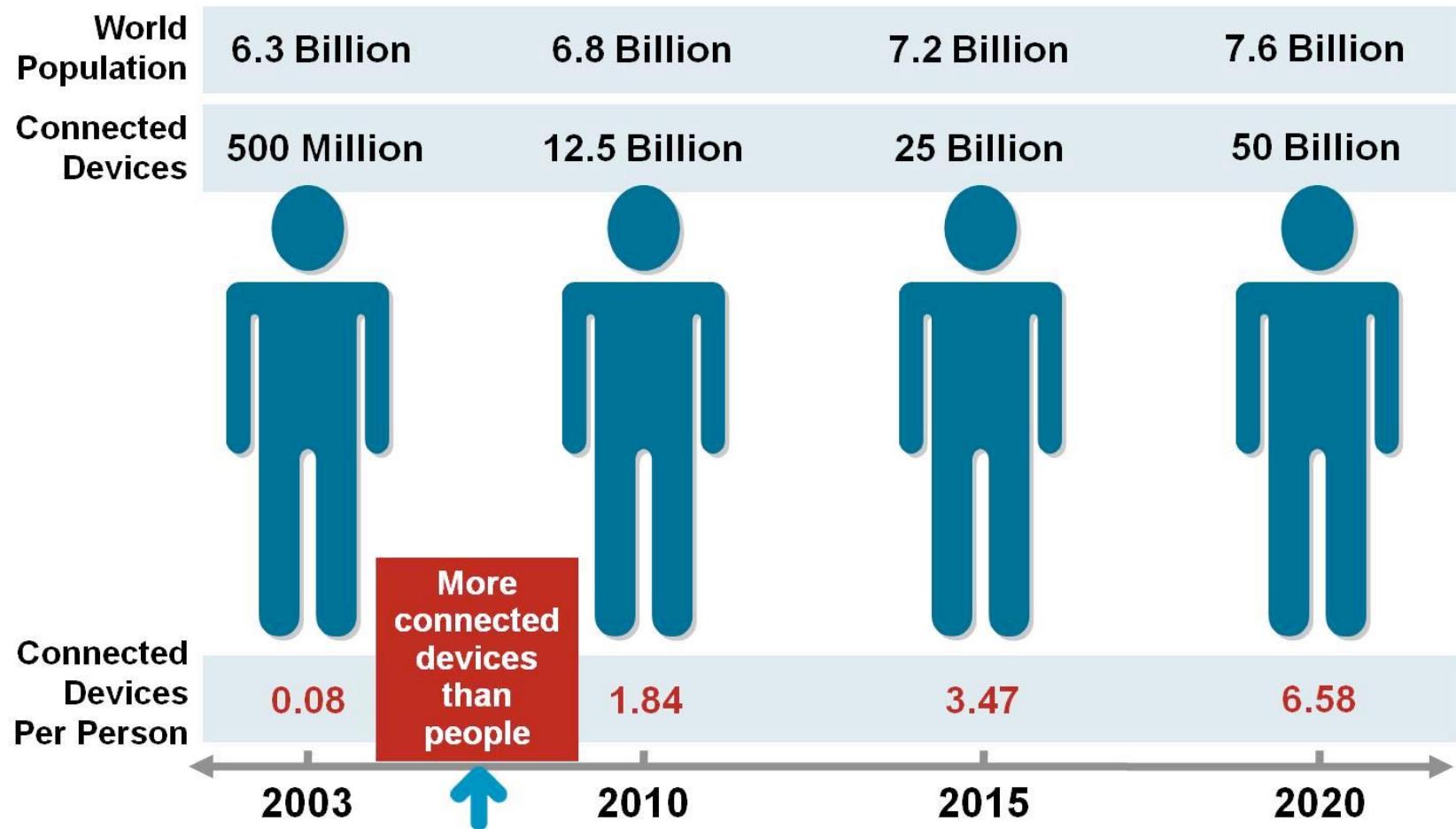
|                              | <b><math>\mu</math>Clinux</b> | <b>TinyOS</b> | <b>RIOT</b> |
|------------------------------|-------------------------------|---------------|-------------|
| Minimum RAM                  | < 32 MB                       | < 1 kB        | ~1.5 kB     |
| Minimum ROM                  | < 2 MB                        | < 4 kB        | ~5 kB       |
| C Support                    | ✓                             | ✗             | ✓           |
| C++ Support                  | ✓                             | ✗             | ✓           |
| Multithreading               | ✓                             | ○             | ✓           |
| Microcontrollers without MMU | ✓                             | ✓             | ✓           |
| Modularity                   | ○                             | ✗             | ✓           |
| Real time                    | ○                             | ✗             | ✓           |

✓ = full support

○ = partial support

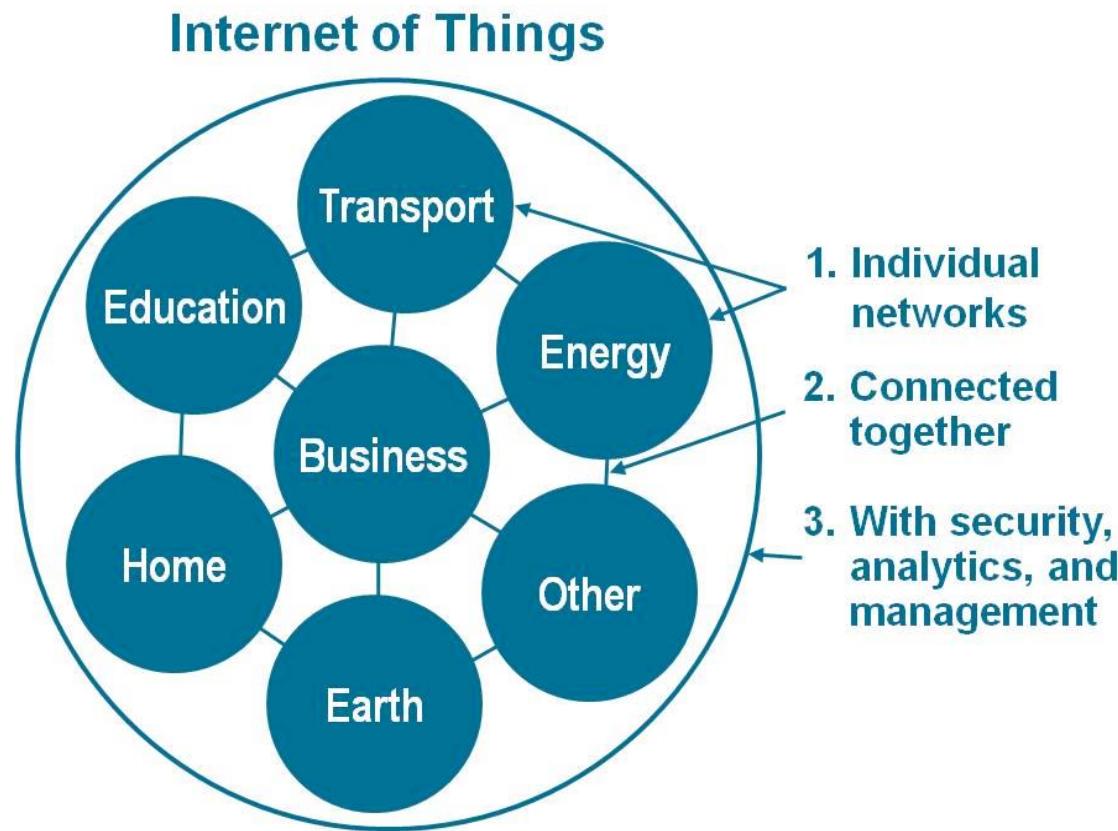
✗ = no support

# Current Status & Future Prospect of IoT



***"Change is the only thing permanent in this world"***

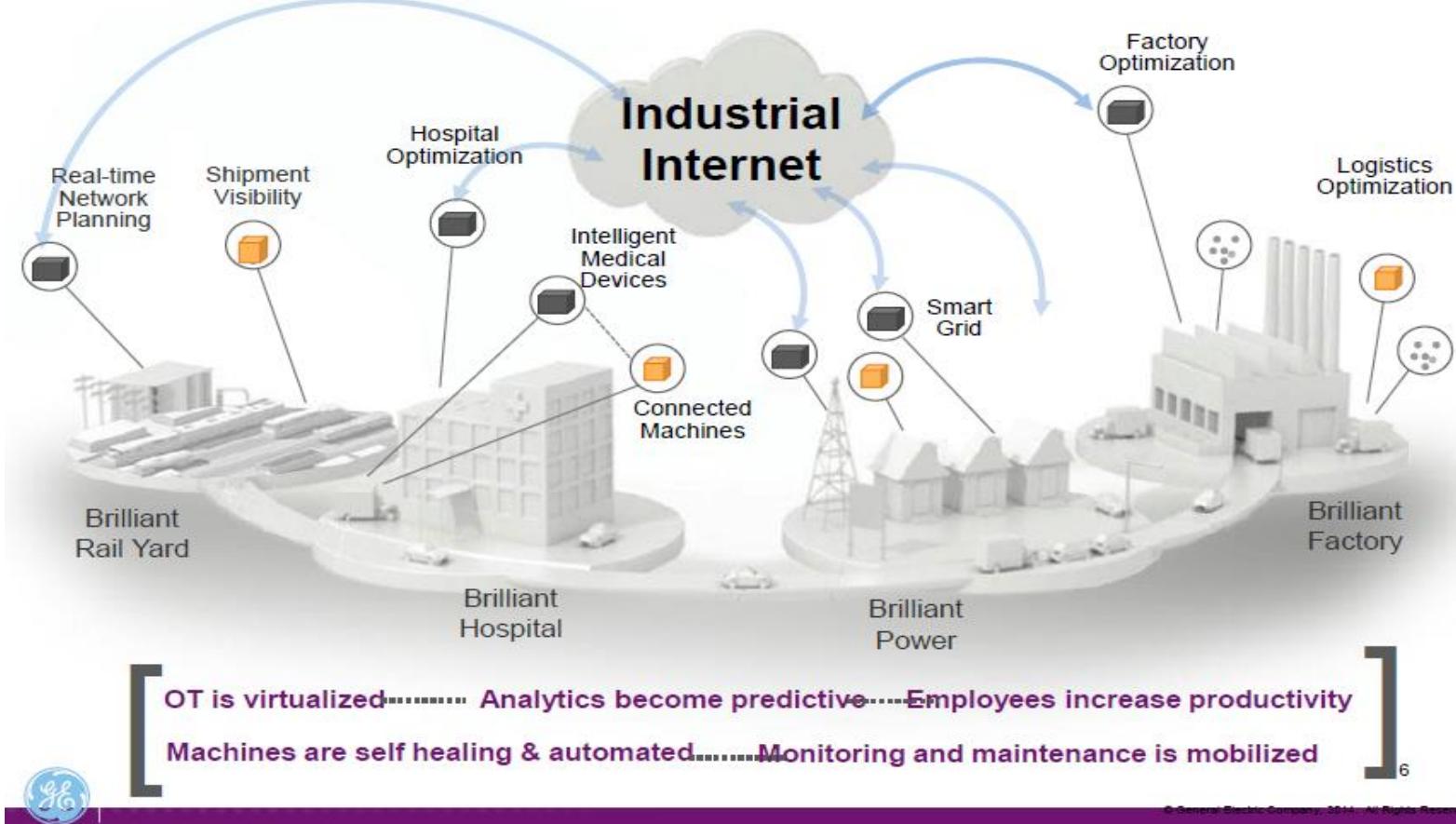
# IoT as a Network of Networks:



These networks connected with added security, analytics, and management capabilities. This will allow IoT to become even more powerful in what it can help people achieve.

# The Future of IoT

What happens when 50B Machines become connected?

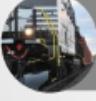


*"The Sky's not the limit. It's only the beginning with IoT."*

# The Potential of IoT

## Value of Industrial Internet is huge

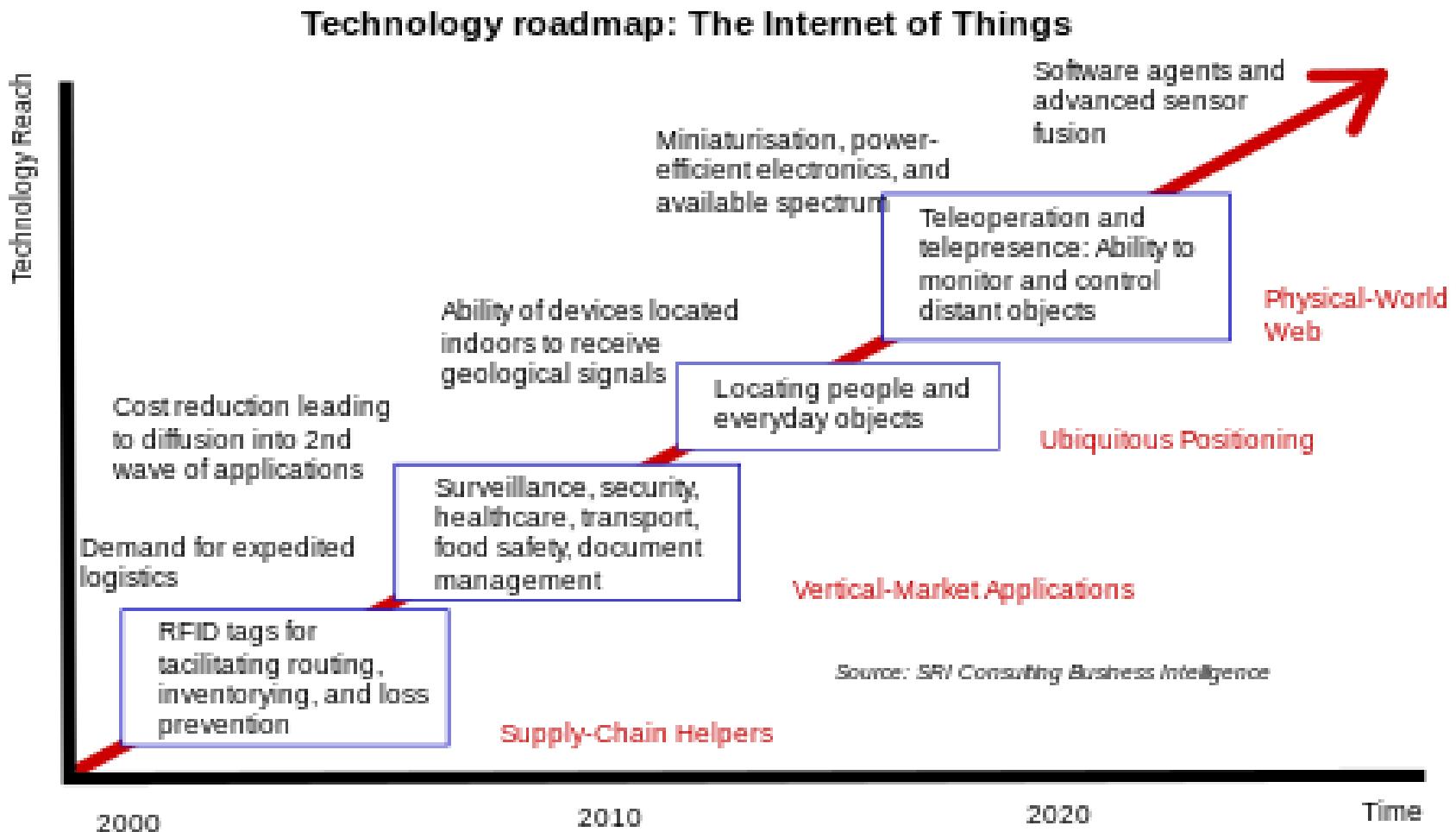
Connected machines and data could eliminate up to \$150 billion in waste across industries

| Industry  | Segment                     | Type of savings                      | Estimated value over 15 years<br>(Billion nominal US dollars) |
|---|-----------------------------|--------------------------------------|---|
|  Aviation      | Commercial                  | 1% fuel savings                      | \$30B   |
|  Power         | Gas-fired generation        | 1% fuel savings                      | \$66B   |
|  Healthcare    | System-wide                 | 1% reduction in system inefficiency  | \$63B   |
|  Rail         | Freight                     | 1% reduction in system inefficiency  | \$27B   |
|  Oil and Gas | Exploration and development | 1% reduction in capital expenditures | \$90B   |

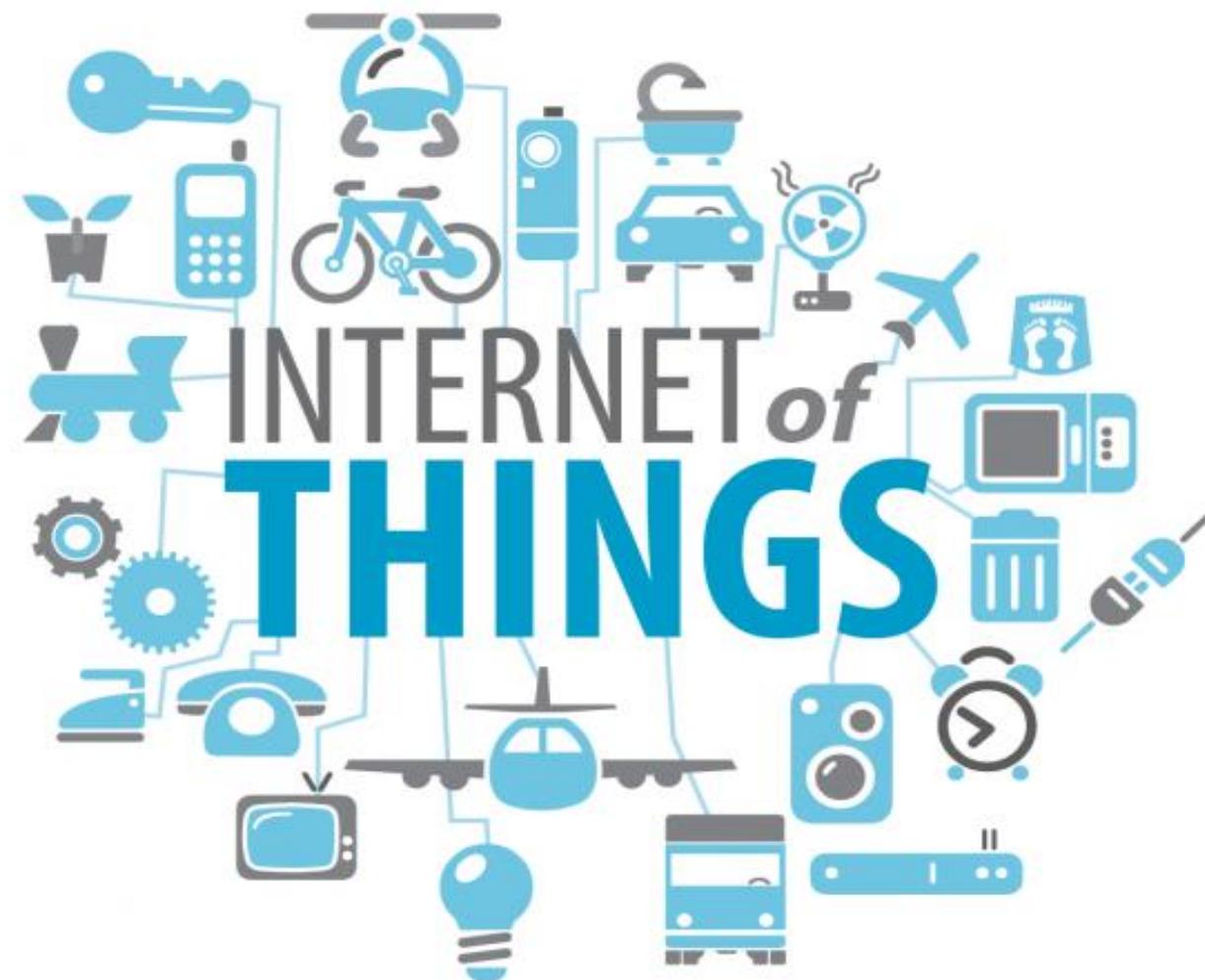
Note: Illustrative examples based on potential one percent savings applied across specific global industry sectors. Source: GE estimates

GE's estimates on potential of just ONE percent savings applied using IoT across global industry sectors.

# Technology roadmap of IoT



# Applications of IoT



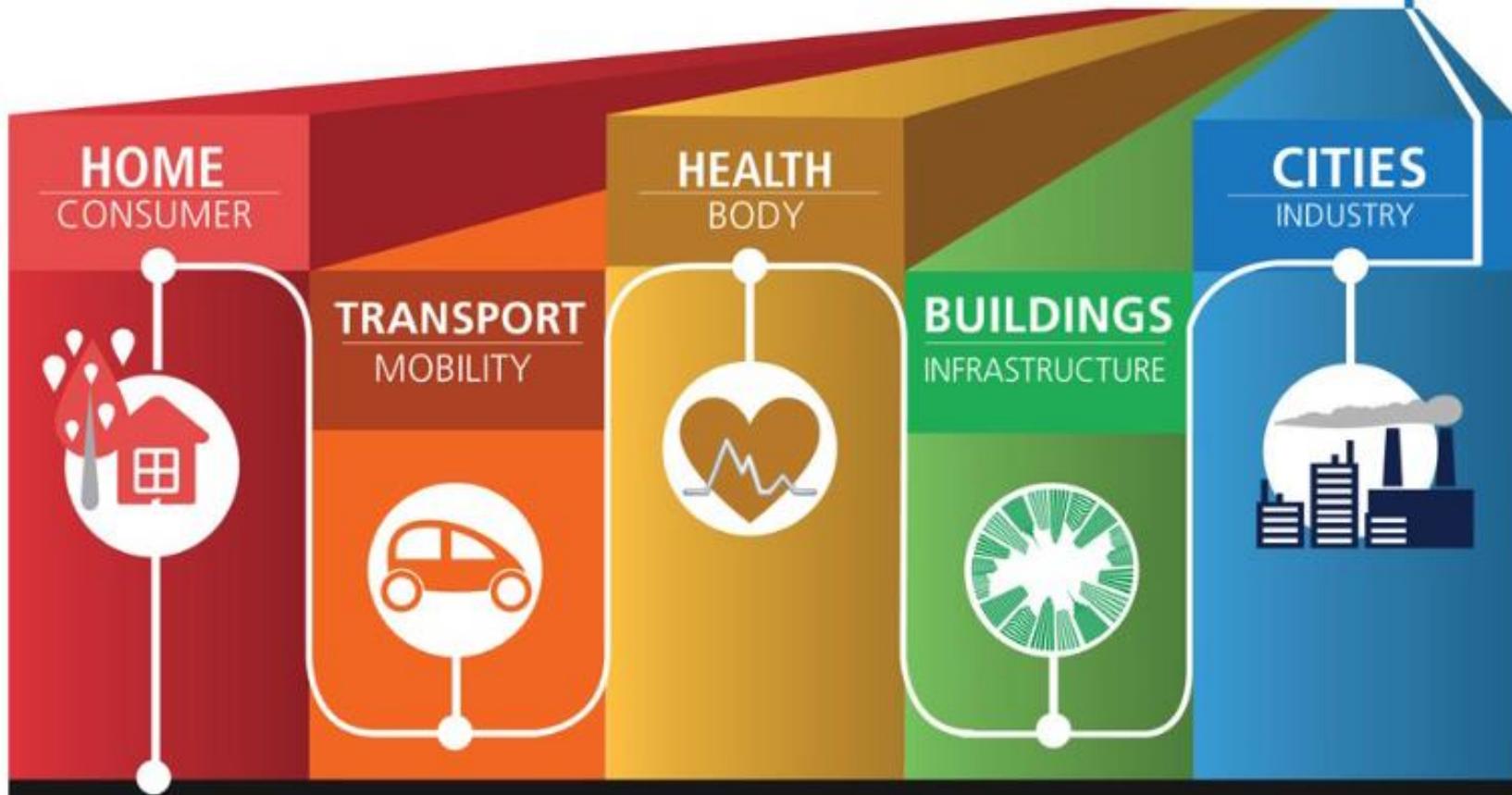
*"The Ultimate Goal of IOT is to Automate Human Life."*

# Few Applications of IoT

- ✓ Building and Home automation
- ✓ Manufacturing
- ✓ Medical and Healthcare systems
- ✓ Media
- ✓ Environmental monitoring
- ✓ Infrastructure management
- ✓ Energy management
- ✓ Transportation
- ✓ Better quality of life for elderly
- ✓ ... .... ..

***You name it, and you will have it in IoT!***

# TO ➔ DIVERSE APPLICATIONS



*Light bulbs  
Security  
Pet Feeding  
Irrigation Controller  
Smoke Alarm  
Refrigerator  
Infotainment  
Washer / Dryer  
Stove  
Energy Monitoring*

*Traffic routing  
Telematics  
Package Monitoring  
Smart Parking  
Insurance Adjustments  
Supply Chain  
Shipping  
Public Transport  
Airlines  
Trains*

*Patient Care  
Elderly Monitoring  
Remote Diagnostic  
Equipment Monitoring  
Hospital Hygiene  
Bio Wearables  
Food sensors*

*HVAC  
Security  
Lighting  
Electrical  
Transit  
Emergency Alerts  
Structural Integrity  
Occupancy  
Energy Credits*

*Electrical Distribution  
Maintenance  
Surveillance  
Signage  
Utilities / Smart Grid  
Emergency Services  
Waste Management*

# Smart Parking

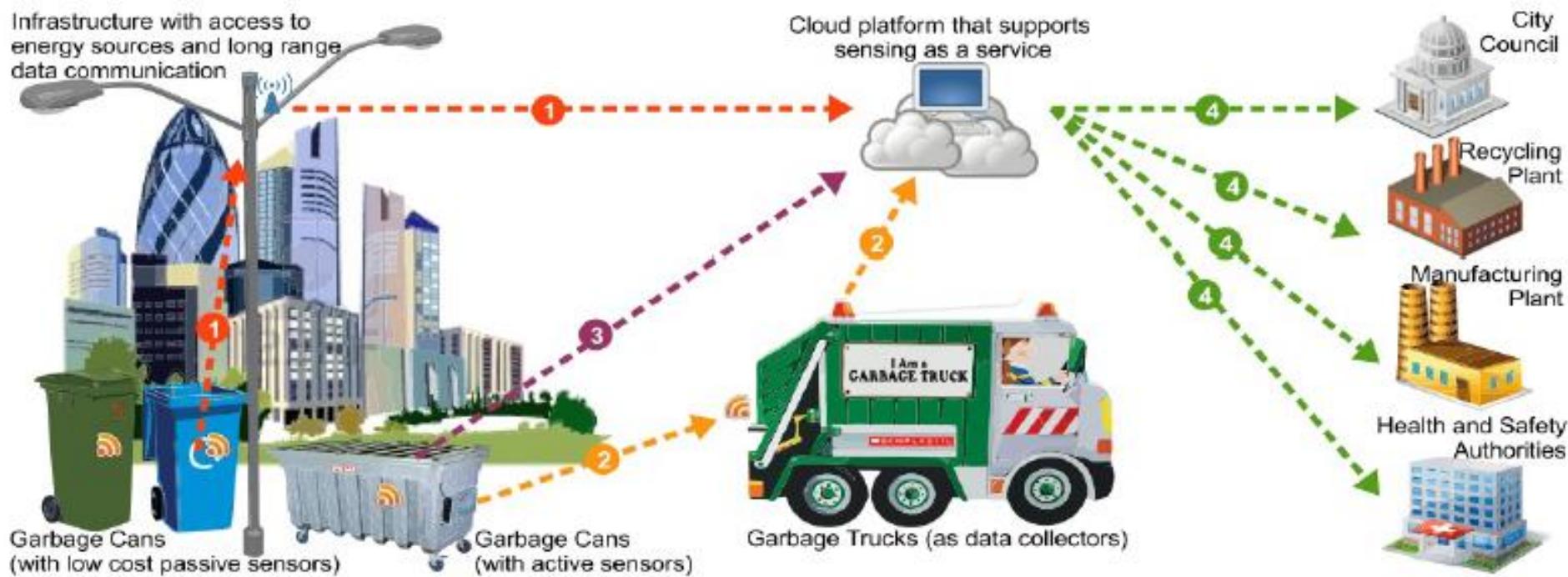
Create **USD 41Billion** by providing visibility into the availability of parking spaces across the city.



Residents can identify and reserve the closest available space, traffic wardens can identify non-compliant usage, and municipalities can introduce demand-based pricing.

[Source: <http://www.telecomreseller.com/2014/01/11/cisco-study-says-iot-can-create-savings/>]

# Efficient Waste Management in Smart Cities Supported by the Sensing-as-a-Service



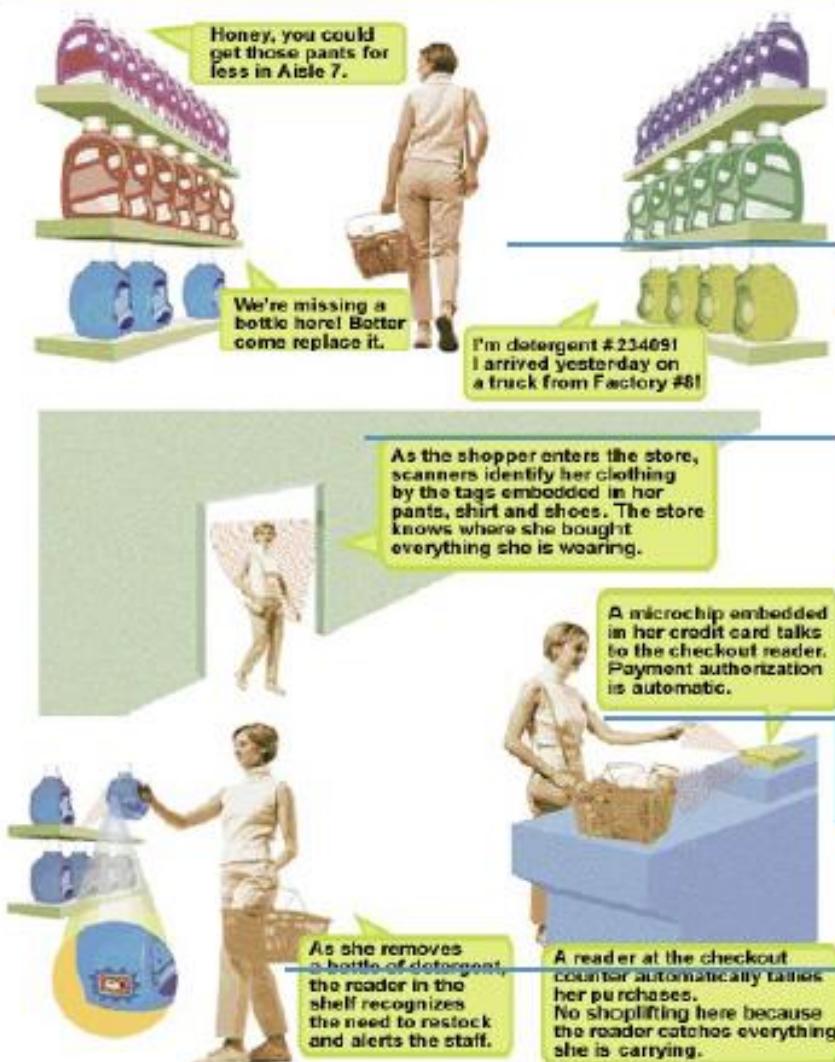
[Source: "Sensing as a Service Model for Smart Cities Supported by Internet of Things", Charith Perera et. al., Transactions on Emerging Telecommunications Technology, 2014]

# Sensors in even the holy cow!



In the world of IoT, even the cows will be connected and monitored. Sensors are implanted in the ears of cattle. This allows farmers to monitor cows' health and track their movements, ensuring a healthier, more plentiful supply of milk and meat for people to consume. On average, each cow generates about 200 MB of information per year.

# IOT Application Scenario - Shopping

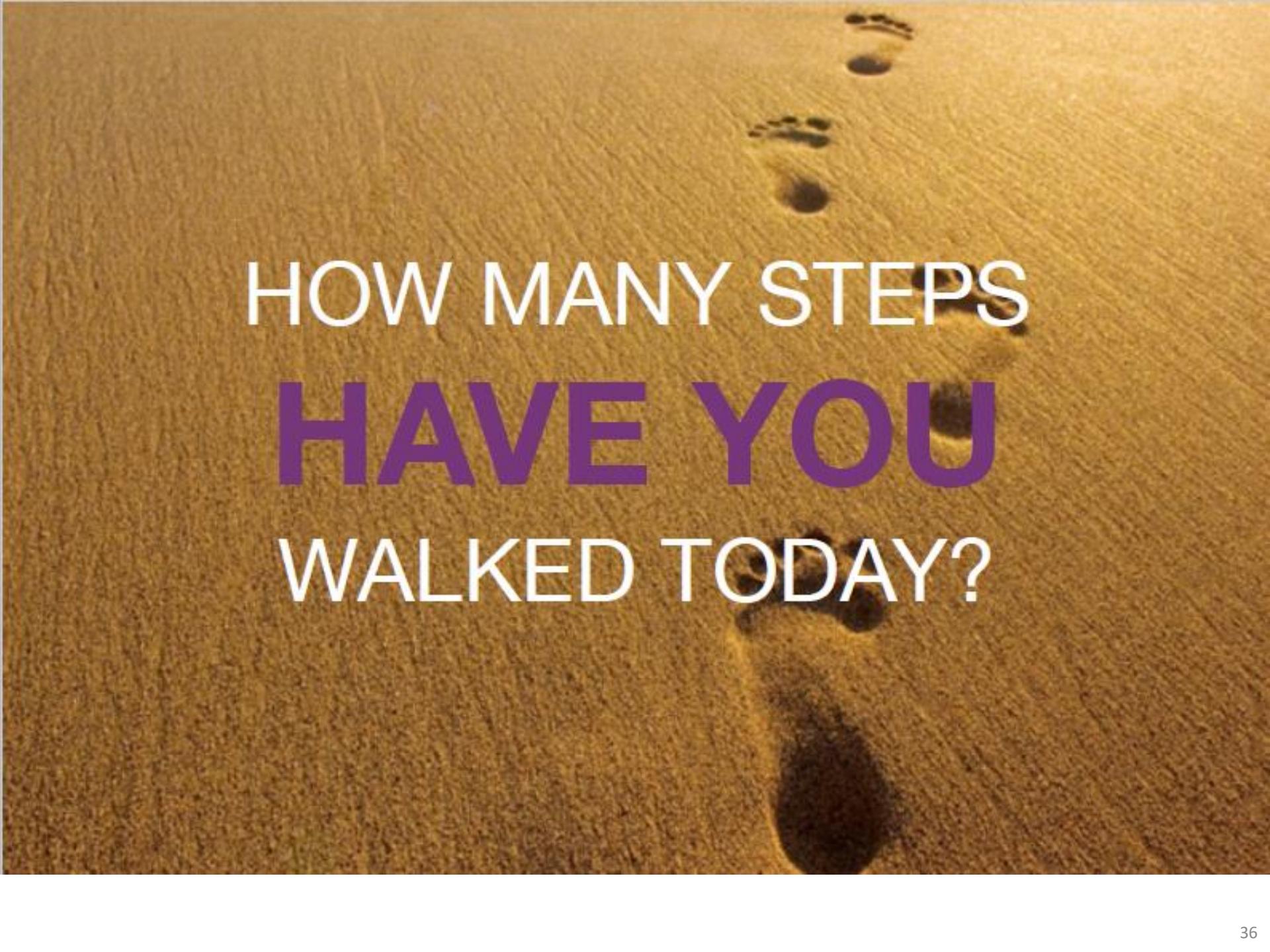


(2) When shopping in the market, the goods will introduce themselves.

(1) When entering the doors, scanners will identify the tags on her clothing.

(4) When paying for the goods, the microchip of the credit card will communicate with checkout reader.

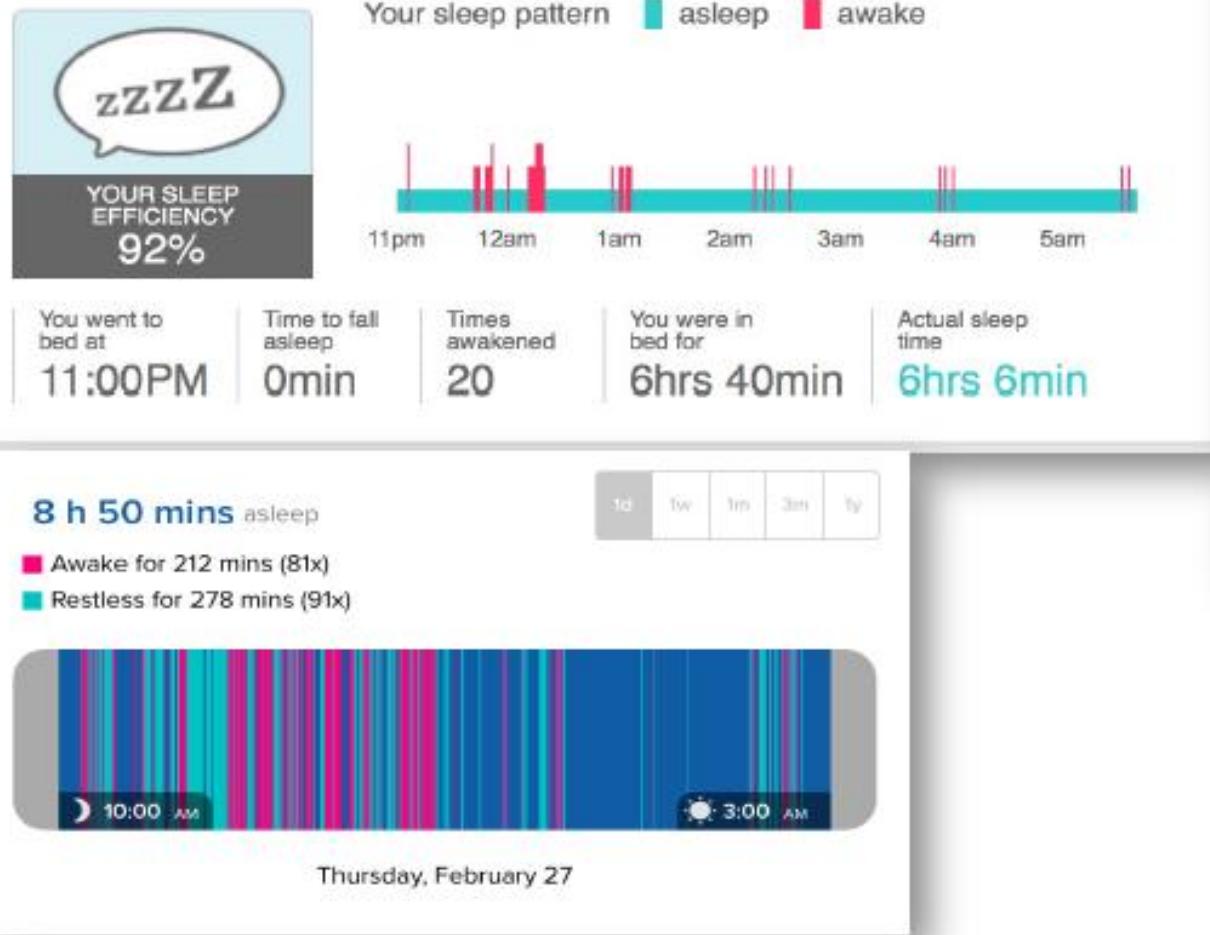
(3) When moving the goods, the reader will tell the staff to put a new one.



HOW MANY STEPS  
**HAVE YOU**  
WALKED TODAY?

# How Well Do I Sleep?

## Sleep



## Sleep Stats

Time asleep over the past 30 days in hours



Times awoken over the past 30 days



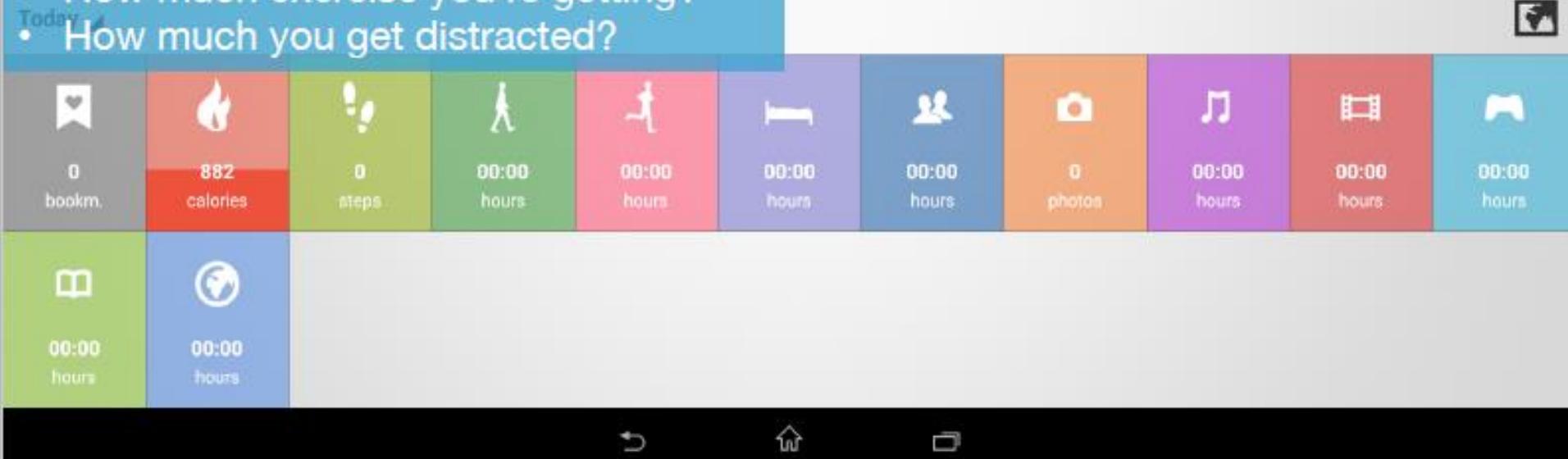
fitbit flex  
Wireless Activity + Sleep Wristband



# I Want To Know More About Myself

- Where you're going?
- Who you've interacted with?
- How long you've spoken to friends?
- The affinity of connections?
- How long it takes to get to work?
- The tone of your messages
- The amount you text, tweet or update?
- How much exercise you're getting?
- How much you get distracted?

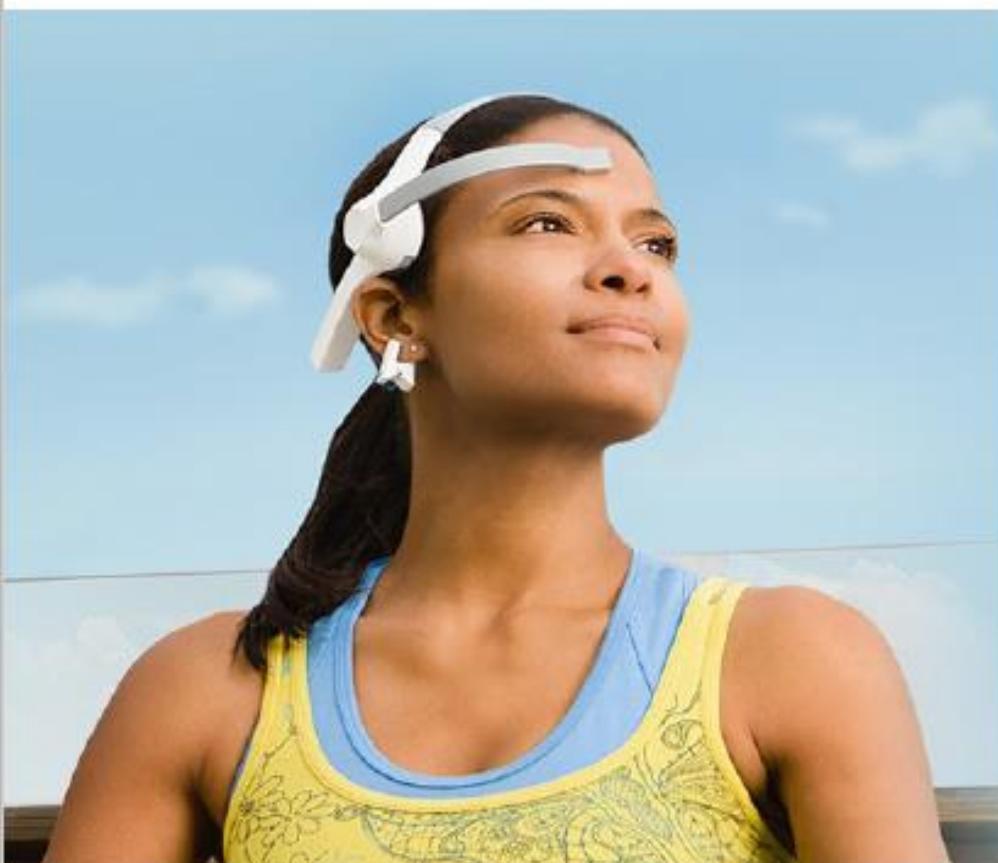
Today



Can Internet of Things (IOT) Help Us To Know More About Ourselves?

***IoT helps you in LIFE LOGGING***

# Thought Controlled Computing



The flagship product, MindWave, is a headset that can log into your computer using just your thoughts. Researchers recently used the EEG headset to develop a toy car that can be driven forward with thought.

NeuroSky's smart sensors can also track your heart rate and other bodily metrics and can be embedded in the next generation of wearable devices.

*"We make it possible for millions of consumers to capture and quantify critical health and wellness data,"* Yang (CEO of Softbank) said. Softbank is the funder.

[Source: <http://venturebeat.com/2013/11/04/next-step-for-wearables-neurosky-brings-its-smart-sensors-to-health-fitness/> ]

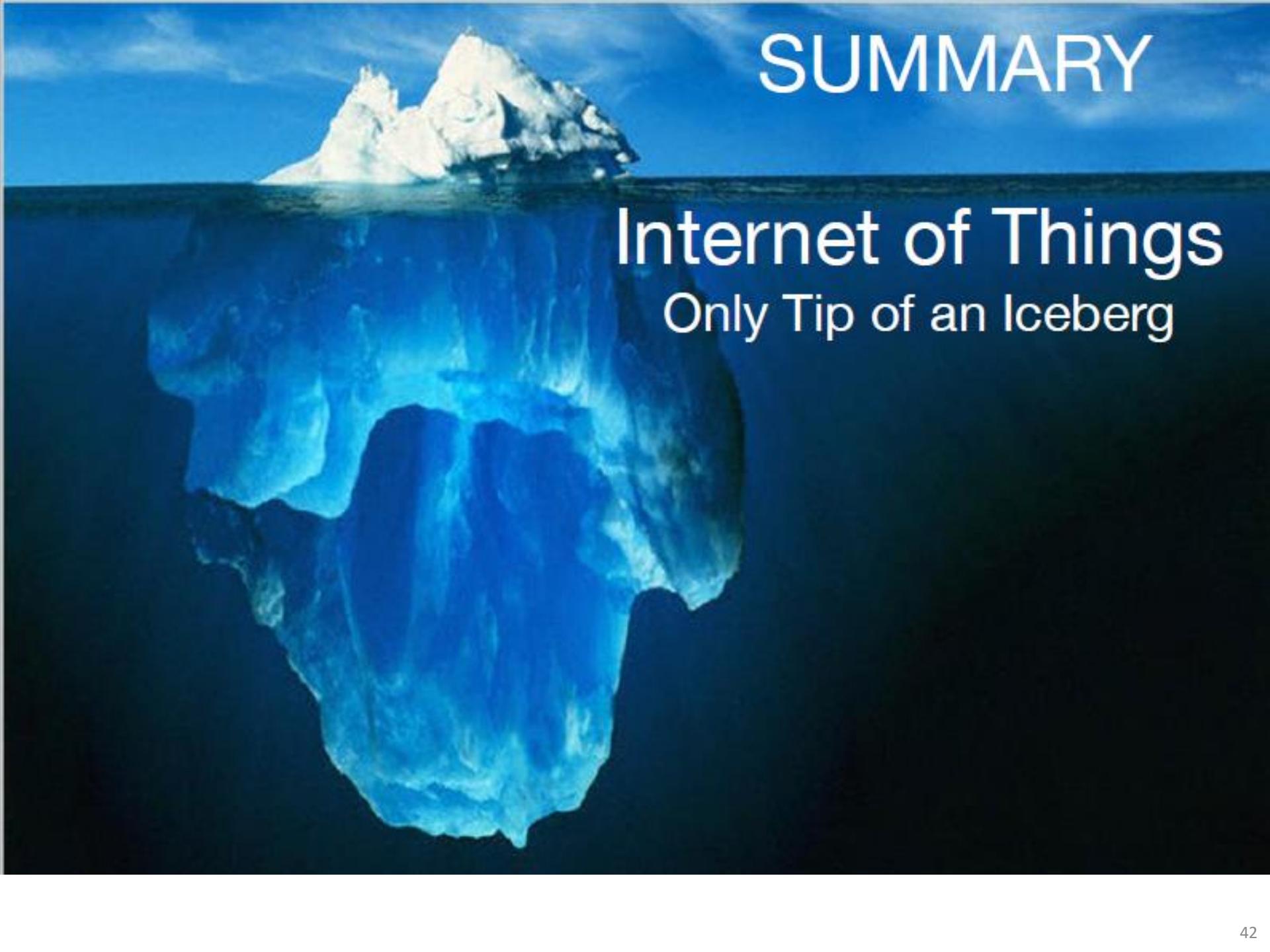
“Big Data is not magic. It doesn’t matter how much data you have if you can’t make sense of it.”



# Criticisms and Controversies of IoT

Scholars and social observers and pessimists have doubts about the promises of the ubiquitous computing revolution, in the areas as:

- Privacy
- Security
- Autonomy and Control
- Social control
- Political manipulation
- Design
- Environmental impact
- Influences human moral decision making

The background image shows a massive iceberg floating in the ocean. Only a small portion of the iceberg is visible above the water's surface, while the vast majority of it remains hidden beneath the waves.

# SUMMARY

## Internet of Things

Only Tip of an Iceberg

# Fog Computing

Introduzione, tecnologie e sfide



Dipartimento di Ingegneria  
Informatica, Automatica e  
Gestionale "A. Ruberti"

# Indice

## 1. Introduzione

Dal Cloud al Fog Computing

## 2. Modello Concettuale

Definizione del Fog Computing secondo il NIST

## 3. Tecnologie

Hardware e software tipici dello strato Fog

## 4. Applicazioni

Scenari applicativi del fog computing

## 5. Conclusioni

Riepilogo finale

1

# Introduzione

# Cloud Computing

## Modello Classico



# Cloud Computing

Servizi Noti



Cloud Gaming



A/V

Lato consumer



IBM Cloud



Google Drive



Dropbox

Storage



zoom



Social



Lato enterprise  
(Cloud Providers)

# Cloud Computing

## Locations



# Cloud Computing

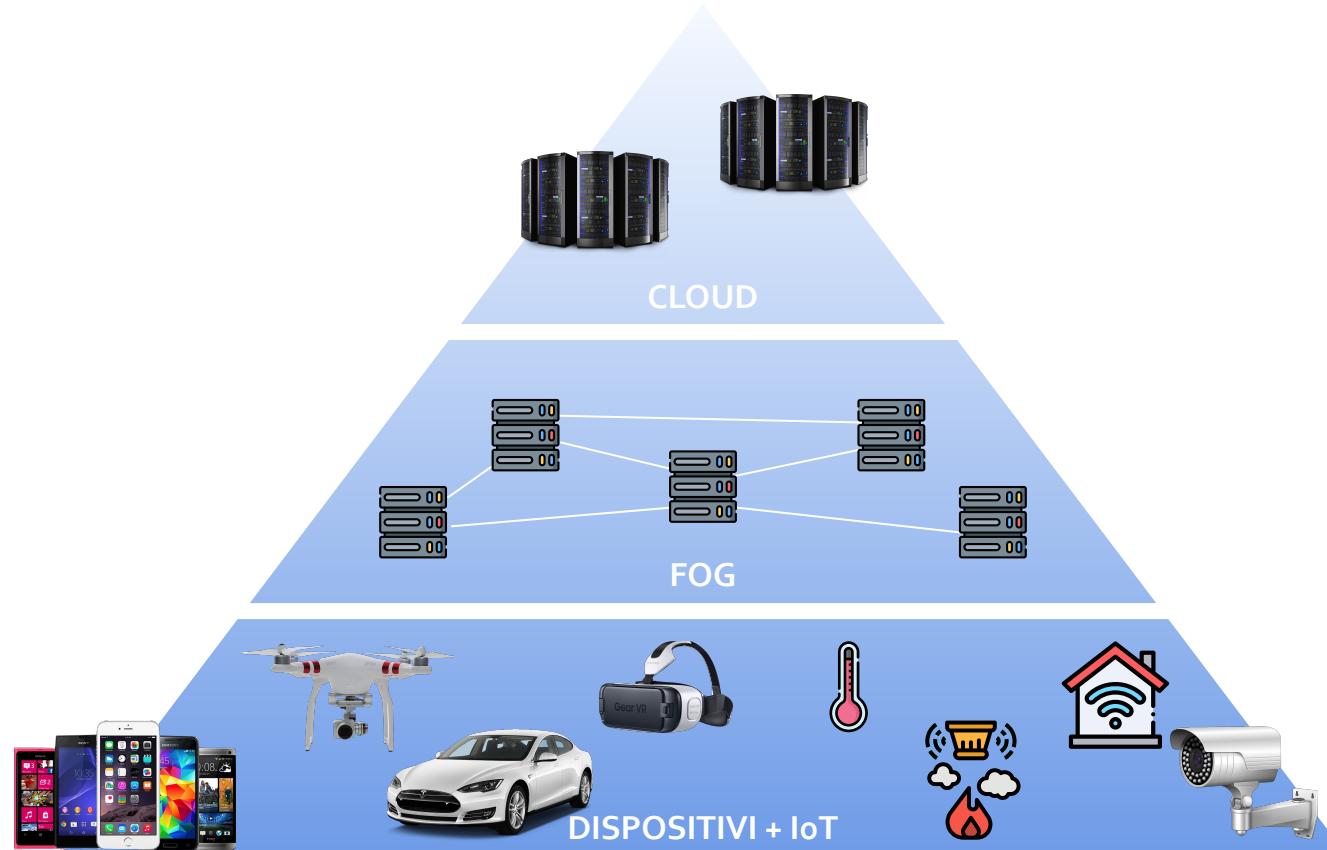
## Classe di applicazioni

Il cloud computing soddisfa i requisiti per un determinato tipo di applicazioni, in particolare, applicazioni in cui:

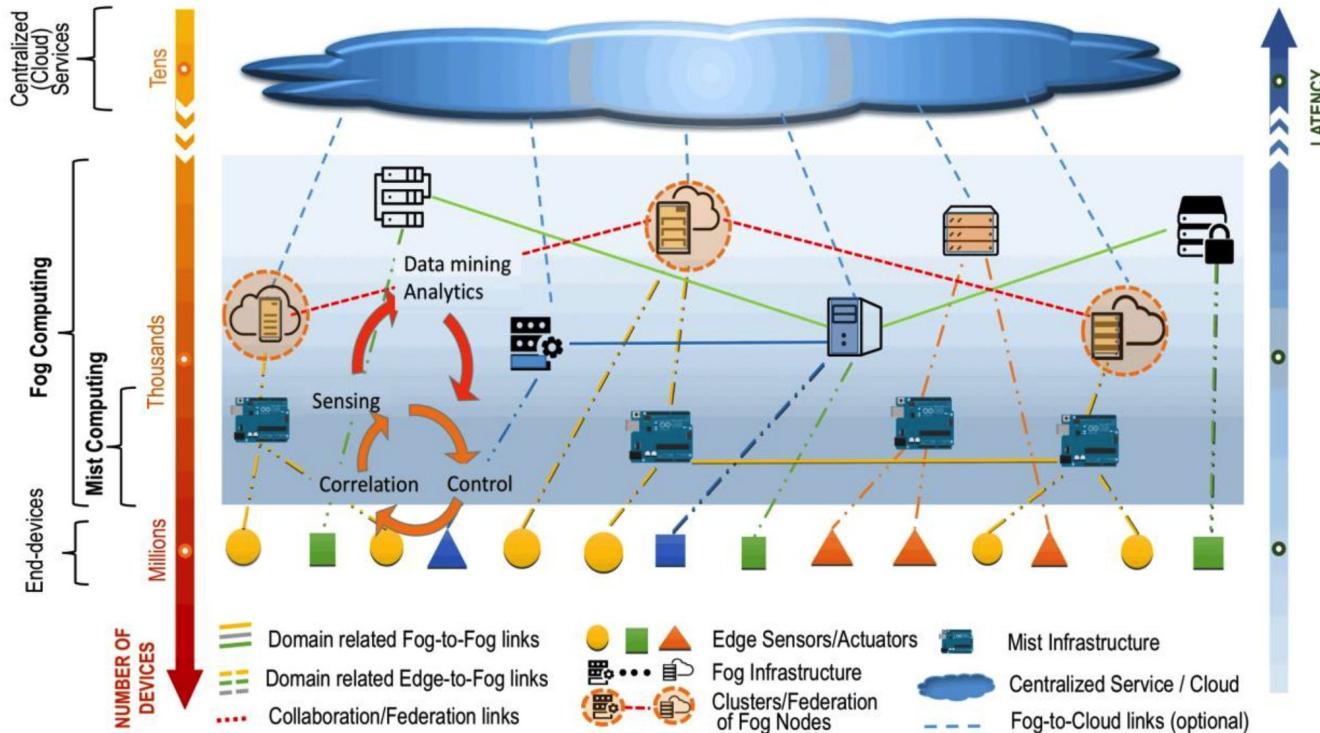
- la bassa **latenza** di ping (< 5ms) non è un requisito determinante
- la **distribuzione** del calcolo è centralizzata
- non vi è necessità di ***location awareness*** (ovvero le singole unità di calcolo non hanno percezione della location in cui si trovano)
- vi è necessità di **grande potenza di calcolo**

Esempio, un'applicazione real-time 30fps deve soddisfare al massimo di una latenza per frame di  $1/30 = 33\text{ms}$ , 60fps invece di circa **16ms**.

# Fog Computing



# Fog Computing



Michaela Iorga, Larry Feldman, Robert Barton, Michael J Martin, Ned Goren, and Charif Mahmoudi. Fog computing conceptual model, NIST, 2018. DOI: [10.6028/NIST.SP.500-325](https://doi.org/10.6028/NIST.SP.500-325)

# Modello Concettuale

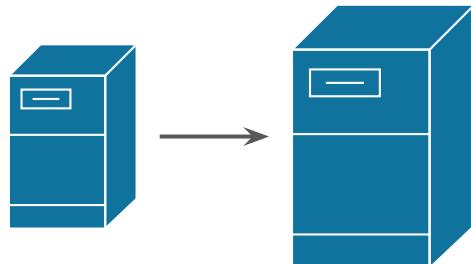
Da "Michaela Iorga, Larry Feldman, Robert Barton, Michael J Martin, Ned Goren, and Charif Mahmoudi. Fog computing conceptual model, NIST, 2018. DOI: [10.6028/NIST.SP.500-325](https://doi.org/10.6028/NIST.SP.500-325)"

# (Scaling)

Verticale e orizzontale

## Verticale

Incrementare la potenza di un singolo nodo in termini di CPU, RAM.



Singolo punto di “failure”

## Orizzontale

Aggiungere più istanze uguali al server



Necessità di bilanciamento del carico (load balancing) per distribuire il lavoro, overhead di rete

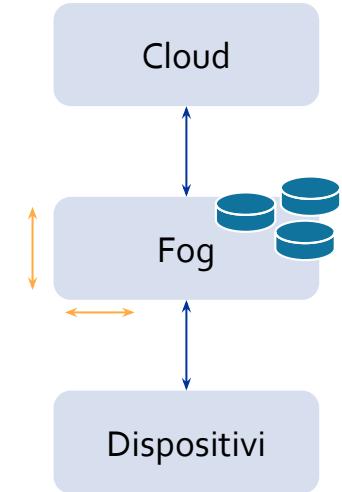
# Fog Computing

"Fog computing is a **layered model** for enabling ubiquitous access to a **shared continuum of scalable computing resources**.

The model facilitates the deployment of distributed, latency-aware applications and services, and consists of **fog nodes** (physical or virtual), residing **between smart end-devices and centralized (cloud) services**.

The fog nodes are **context aware** and support a common data management and communication system. They can be organized in clusters - either **vertically** (to support isolation), **horizontally** (to support federation), or relative to fog nodes' latency-distance to the smart end-devices.

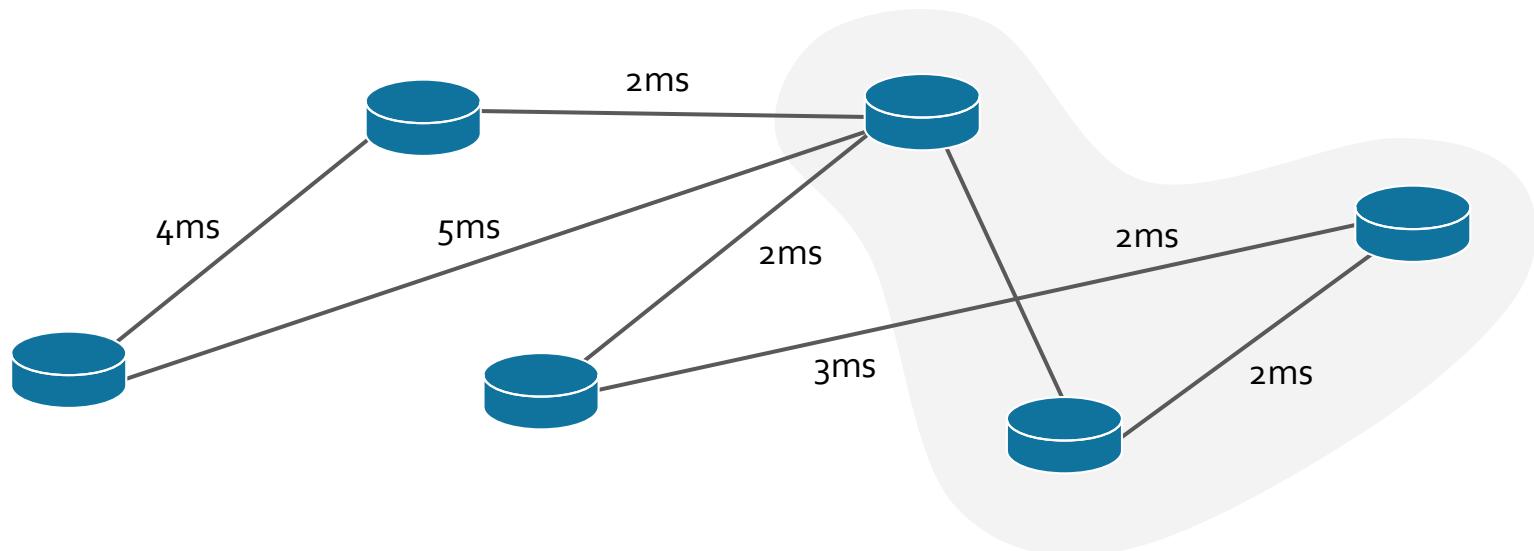
Fog computing **minimizes the request-response time** from/to supported applications, and provides, for the end-devices, local computing resources and, when needed, network connectivity to centralized services."



Continuum di calcolo  
■ Offloading ■ Scaling

# Fog Computing

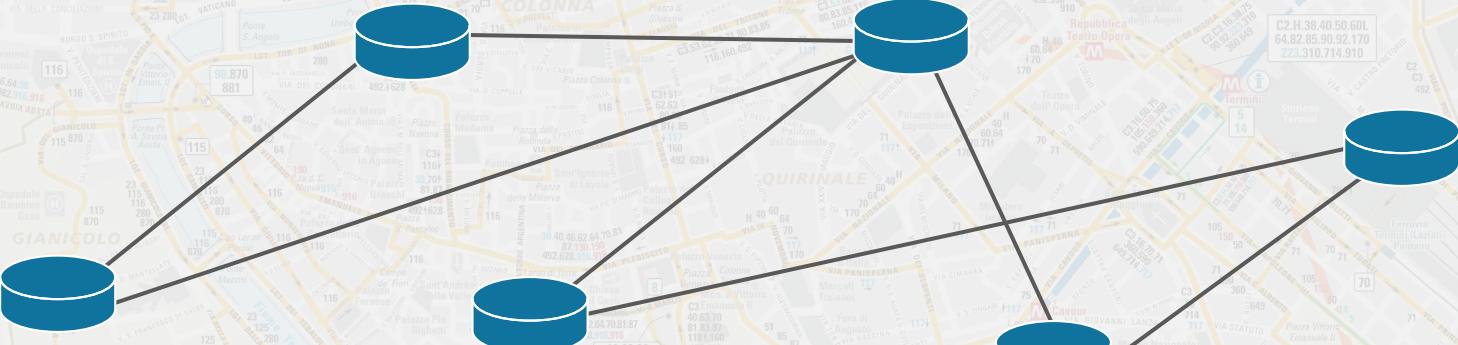
Caratteristiche essenziali · (1) Location awareness



**Contextual location awareness, and low latency.** Il fog computing offre la minor latenza possibile grazie al fatto che i nodi conoscono la posizione logica (o anche fisica) all'interno della rete e di solito questa è prossima ai dispositivi (client)

# Fog Computing

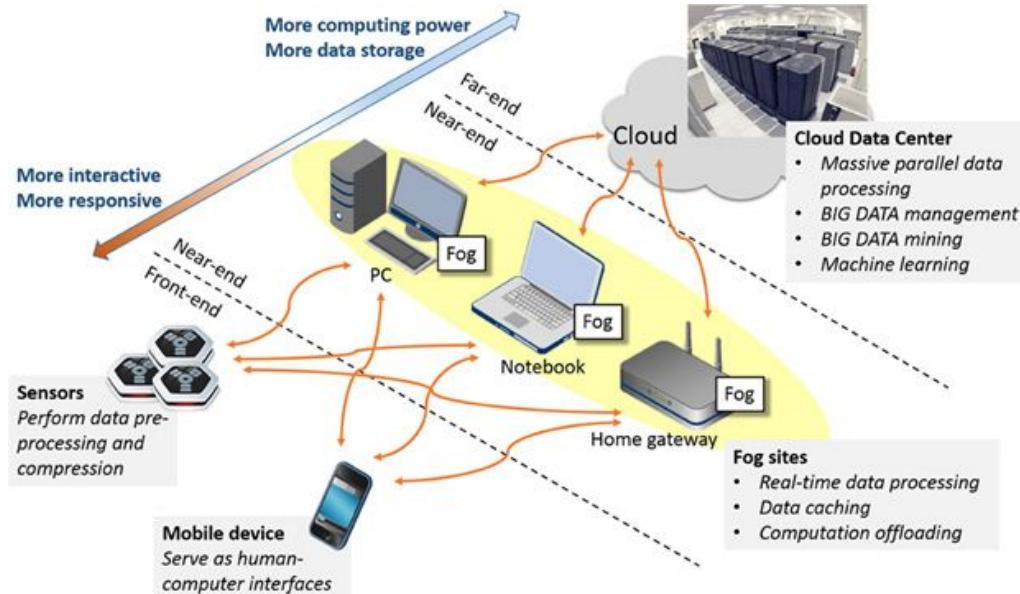
Caratteristiche essenziali · (2) Geographical Distribution



**Geographical distribution.** Differentemente dal cloud centralizzato, i servizi offerti dal fog computing sono distribuiti su un'area geografica specifica e identificabile (es. streaming ad alta qualità verso autoveicoli grazie ai nodi distribuiti lungo strade e autostrade)

# Fog Computing

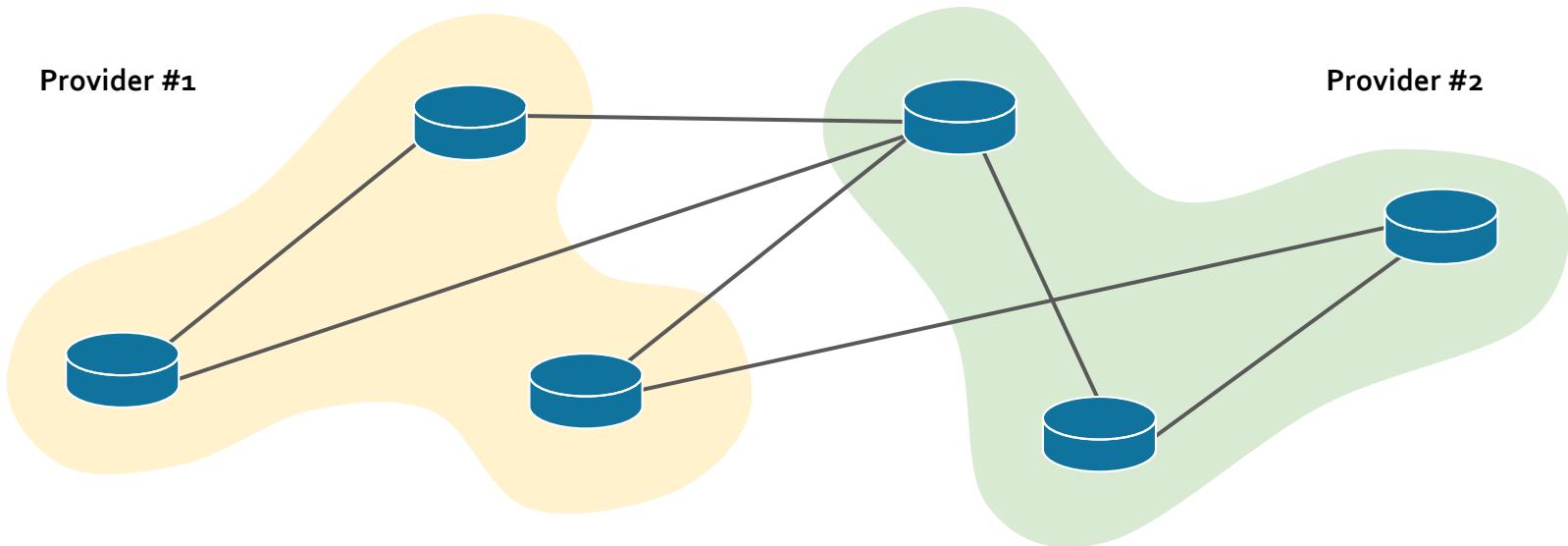
## Caratteristiche essenziali · (3) Heterogeneity



**Heterogeneity.** I dati processati dal fog computing provengono da qualsiasi tipo di rete o sensore. Non solo, qualsiasi dispositivo dotato di potenza di calcolo può diventare un nodo fog.

# Fog Computing

Caratteristiche essenziali · (4) Interoperability and federation

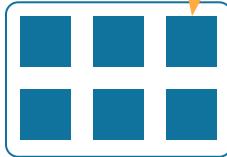
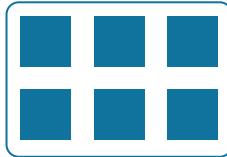
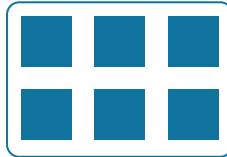
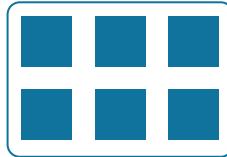


**Interoperability and federation.** Servizi come il real-time streaming, per esempio, richiedono la cooperazione tra provider differenti. Il fog computing deve essere in grado di interoperare tra provider e domini differenti.

# Fog Computing

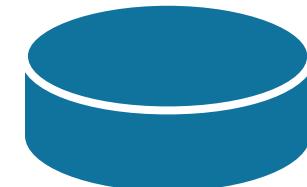
Caratteristiche essenziali · (5) Real-time interactions

Batch Processing



Job

Batch



Fog Node

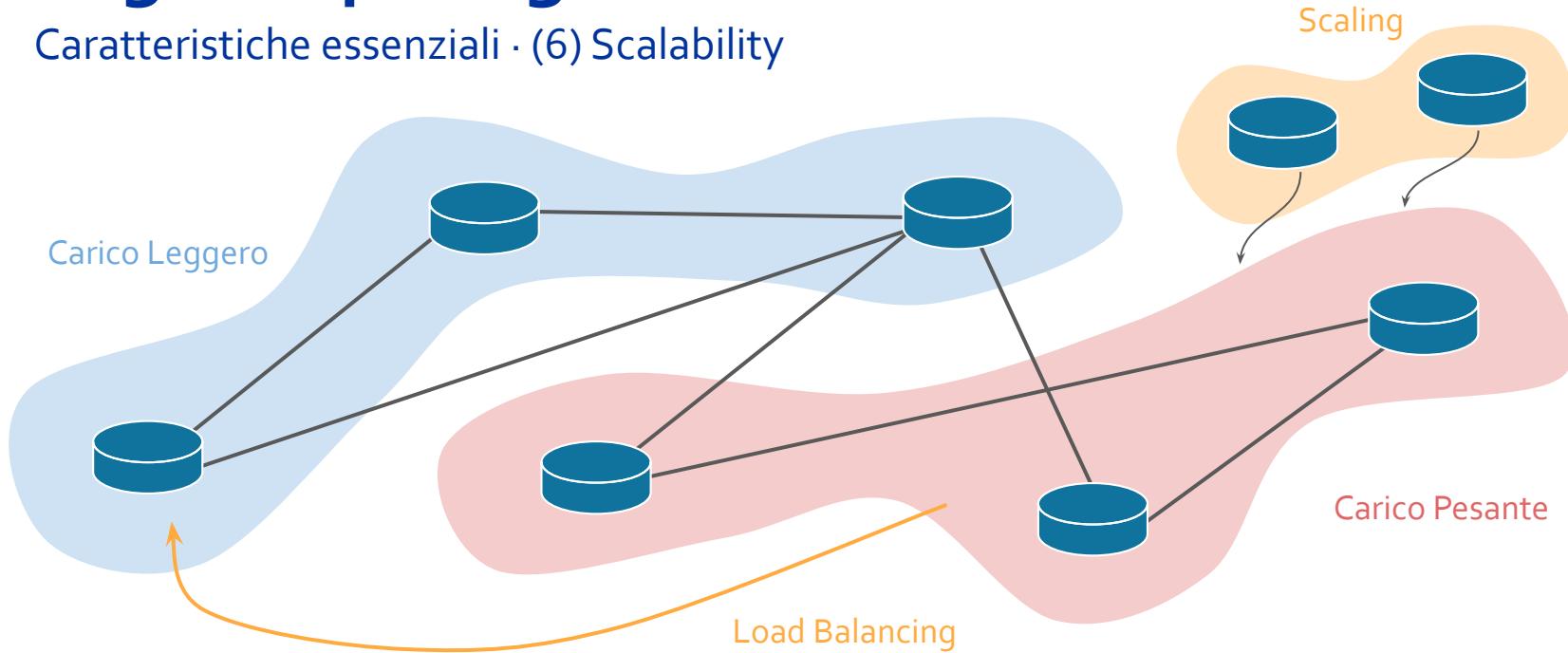
Stream processing



**Real-time interactions.** Nel fog i dati vengono processati a stream e non a batch.

# Fog Computing

Caratteristiche essenziali · (6) Scalability



**Scalability and agility of federated, fog-node clusters.** Il fog computing ha una natura adattiva, quindi esso è scalabile (elastic computing) e supporta cambiamenti dinamici del traffico o di rete.

3

# Tecnologie

# Hardware

L'architettura Fog può essere realizzata ovviamente anche con l'hardware tipico del cloud, come i **server** nei datacenter (48/128 core, 32-240GB RAM) con l'unica eccezione che questi vengono dislocati geograficamente.

Tuttavia, data la possibilità tecnologica di concentrare grande potenza di calcolo in piccolo spazio (es. smartphone moderni) consente di costruire un'architettura fog anche senza ricorrere a server ultra-performanti, in particolare si possono utilizzare per esempio:

- **PC consumer**, 4/8/12/24 core, 4-32GB RAM, GPU per calcoli di machine learning  
Pro: grande potenza, Contro: alti consumi
- **PC single board**, 4/8 core, 2-8GB RAM, architettura ARM a basso consumo  
Pro: bassi consumi, Contro: potenza non equiparabile a quella dei PC consumers

Ovviamente la scelta dipende dall'applicazione e dal contesto ma recentemente i PC single board hanno raggiunto una potenza non indifferente e possono essere utilizzati in ambienti critici dove per esempio il consumo di energia è determinante (alimentazione da batterie, pannelli solari). Inoltre si può pensare anche ad un'architettura **gerarchica**.

# Hardware

## PC Single Board · Raspberry Pi

Raspberry Pi è un computer a tutti gli effetti formato carta di credito. Caratteristiche essenziali (<https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications>):

- Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
- 2GB, 4GB or 8GB LPDDR4-3200 SDRAM (depending on model)
- 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE
- Gigabit Ethernet
- 2 USB 3.0 ports; 2 USB 2.0 ports.
- Raspberry Pi standard 40 pin GPIO header
- 2 x micro-HDMI ports (up to 4kp60 supported)
- 2-lane MIPI DSI display port, 2-lane MIPI CSI camera port
- 4-pole stereo audio and composite video port
- H.265 (4kp60 decode), H264 (1080p60 decode, 1080p30 encode)
- Micro-SD card slot for loading operating system and data storage
- 5V DC via USB-C connector (minimum 3A\*), or via GPIO (minimum 3A\*)
- Power over Ethernet (PoE) enabled (requires separate PoE HAT)
- Prezzo: 30 - 80€

\* A good quality 2.5A power supply can be used if downstream USB peripherals consume less than 500mA in total.

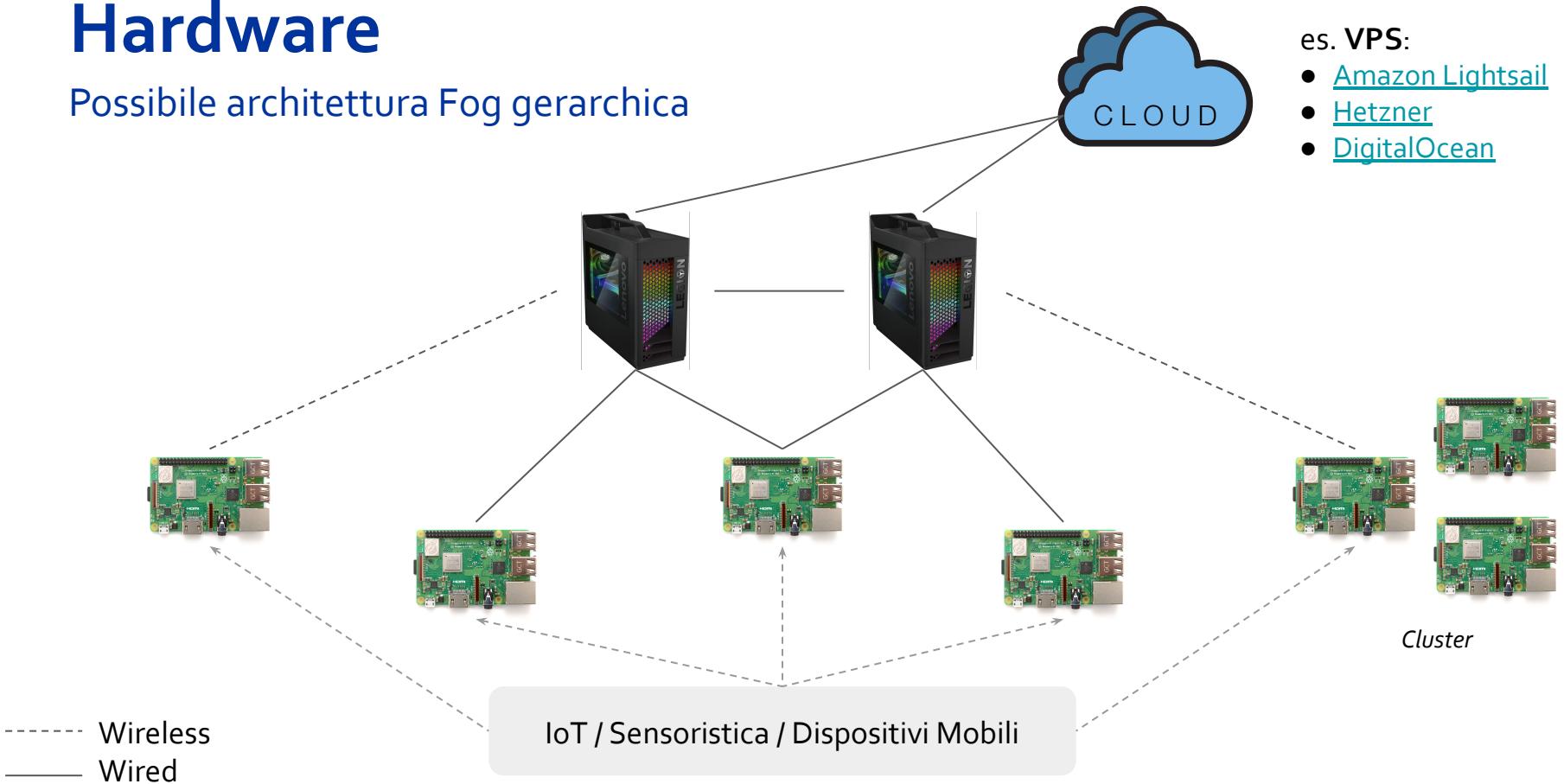


Cluster di 6 Raspberry Pi 4

**Alternative:** nVidia Jetson, ODROID, Banana Pi, Asus Tinkerboard, Orange Pi, ...  
<https://www.slant.co/topics/1629/~best-single-board-computers>

# Hardware

Possibile architettura Fog gerarchica



# Software

## Protocolli

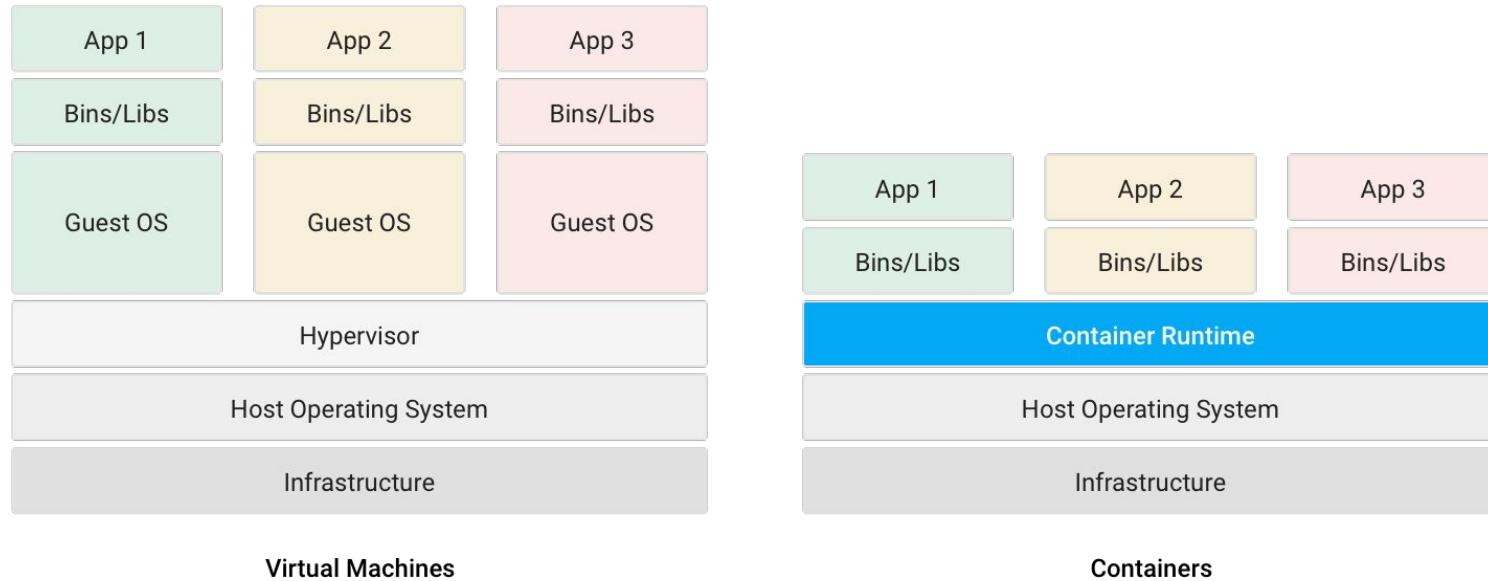
La comunicazione tra nodi fog o con l'IoT può essere **cablata** e quindi sfruttare il protocollo Ethernet 1/10Gbps, fibra ottica (GPON, EPON) oppure **wireless** e quindi sfruttare i protocolli WiFi 802.11n/ac/ax, Bluetooth 5 (anche BLE) o LoRaWAN per l'IoT. In particolare il protocollo **802.11ax** anche chiamato WiFi6 permette di ridurre drasticamente la latenza di comunicazione rispetto a 802.11ac (WiFi5) e raggiunge una velocità teorica di circa 10Gbps.

Il protocollo **HTTP** è invece utilizzato di norma per il trasferimento dei dati, esso permette di implementare le cosiddette REST API utilizzando le operazioni HTTP di GET, POST, PUT, DELETE verso url. La variante **HTTPS**, ad oggi preferita permette di criptare il traffico utilizzando un certificato (anche gratuito) rilasciato da un'autorità garante riconosciuta. Le REST API costituiscono la principale strategia di comunicazione client/server (es. Fog -> Cloud).

# Software

## Containers

La distribuzione delle applicazioni è oggi facilitata con l'utilizzo dei **container**. Un container è nella pratica un processo che vive in un ambiente isolato, ovvero un'immagine in sola lettura che porta con sé tutti file di cui l'applicazione ha bisogno.

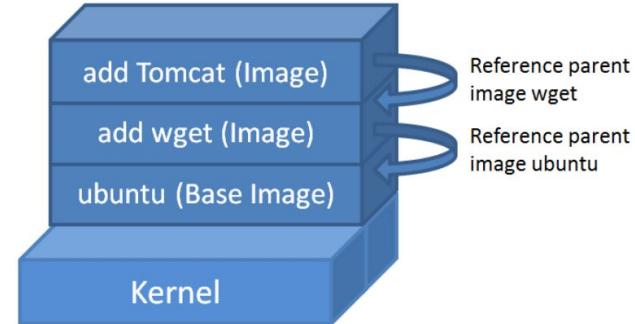


# Software

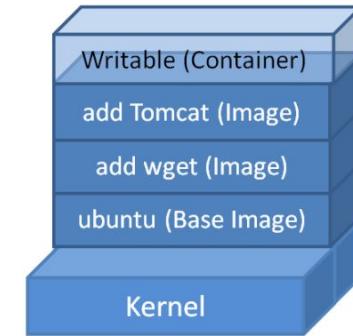
## Docker

Uno dei software più famosi e semplici da utilizzare per gestire i container è **Docker**. Docker fornisce un'interfaccia da terminale semplice e intuitiva per la creazione, l'avvio e la distribuzione dei container. I tre concetti fondamentali sono:

- **Docker Images.** Le immagini docker sono filesystem non scrivibili di natura **stratificata**.
- **Docker Containers.** I container docker sono processi avviati da un'immagine Docker.
- **Docker Registries.** Server che permette di fare push/pull di immagini docker. Il più noto è <https://hub.docker.com>



Natura stratificata delle immagini Docker

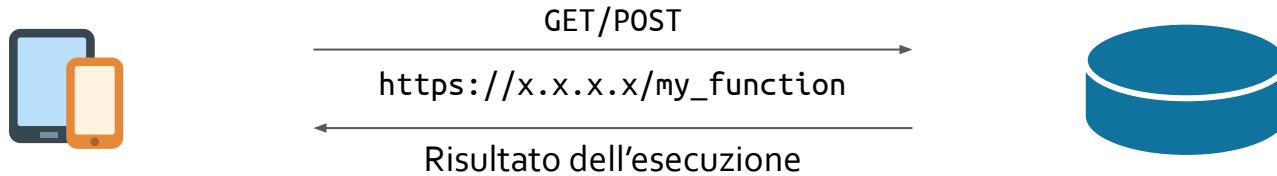


L'avvio di un container crea un nuovo strato scrivibile sullo stack

# Software

## Serverless e Function-as-a-Service (FaaS) - OpenFaaS

Il Function-as-a-Service è un modello di servizio in cui si associa una chiamata **HTTP** a un url specifico all'esecuzione di una funzione. La funzione può essere installata all'interno di un container e questo è un grande vantaggio perché ne consente una migrazione tra nodi quasi istantanea.



Il modello è anche offerto dai cloud providers (es. [AWS Lambda](#)) ma si può anche installare nei propri server utilizzando un framework open source chiamato [OpenFaaS](#). Questo tipo di servizio è anche detto **serverless** perché in contrasto con i server monolitici, con questo paradigma possiamo frammentare le singole API in tanti container differenti non avendo necessità di un singolo server always-on. Nel cloud infatti il modello di tariffazione delle FaaS è per tempo di esecuzione delle singole funzioni, invece per le istanze cloud per tempo di accensione dell'istanza (VM).

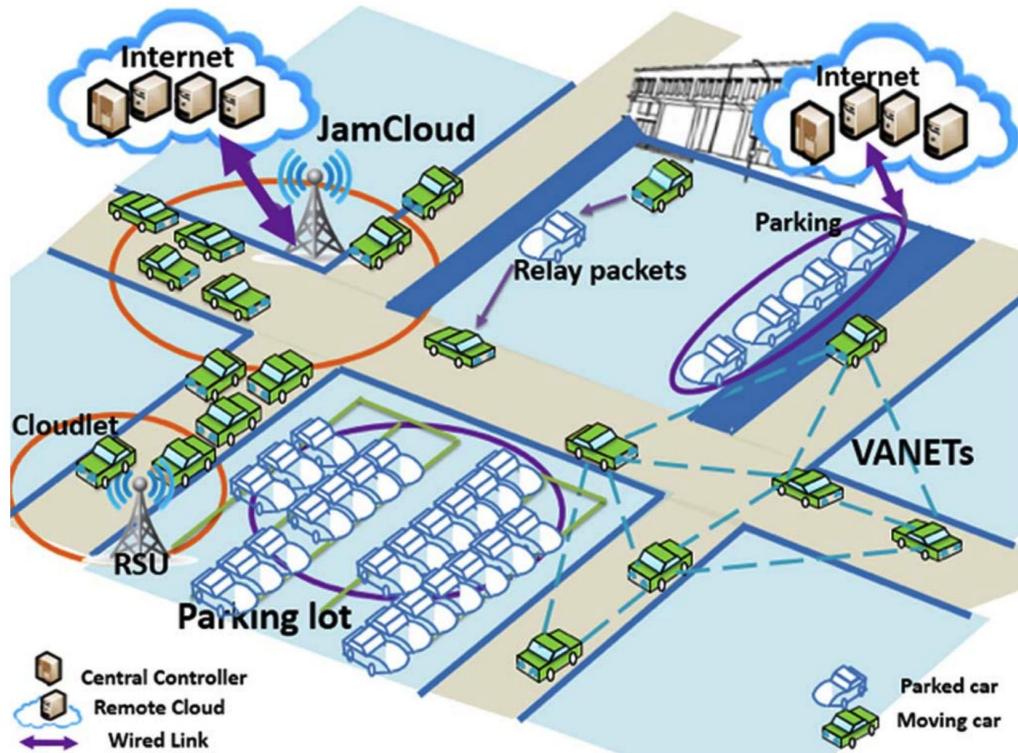
4

# Applicazioni & Sfide

# Smart Vehicles e Smart Cities

Un'applicazione del Fog Computing riguarda il supporto ai veicoli smart.

I nodi fog possono essere disseminati nelle città (**Smart Cities**) e possono offrire potenza computazionale ai dispositivi e non solo. I veicoli a guida autonoma potrebbero dialogare per gestire il traffico e aiutare l'algoritmo di guida segnalando pedoni, stato dei semafori. Gli stessi veicoli costituiscono il cosiddetto Vehicle Fog Computing (VFC).



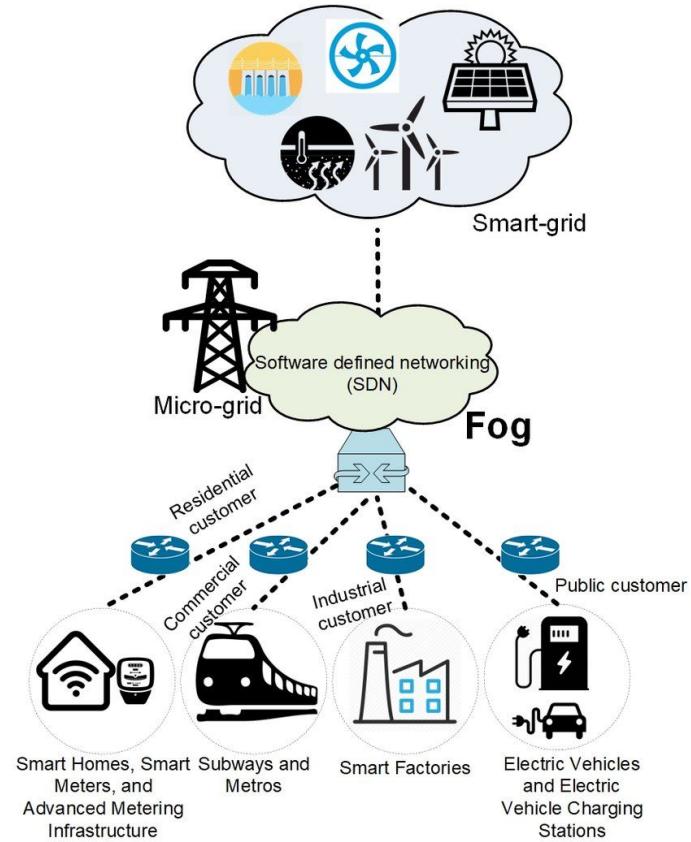
X. Hou, Y. Li, M. Chen, D. Wu, D. Jin and S. Chen, "Vehicular Fog Computing: A Viewpoint of Vehicles as the Infrastructures," in IEEE Transactions on Vehicular Technology, vol. 65, no. 6, pp. 3860-3873, June 2016, doi: 10.1109/TVT.2016.2532863.

# Smart Grid

La rete di distribuzione elettrica (in inglese Grid) è un campo applicativo che beneficia in modo particolare dal fog computing.

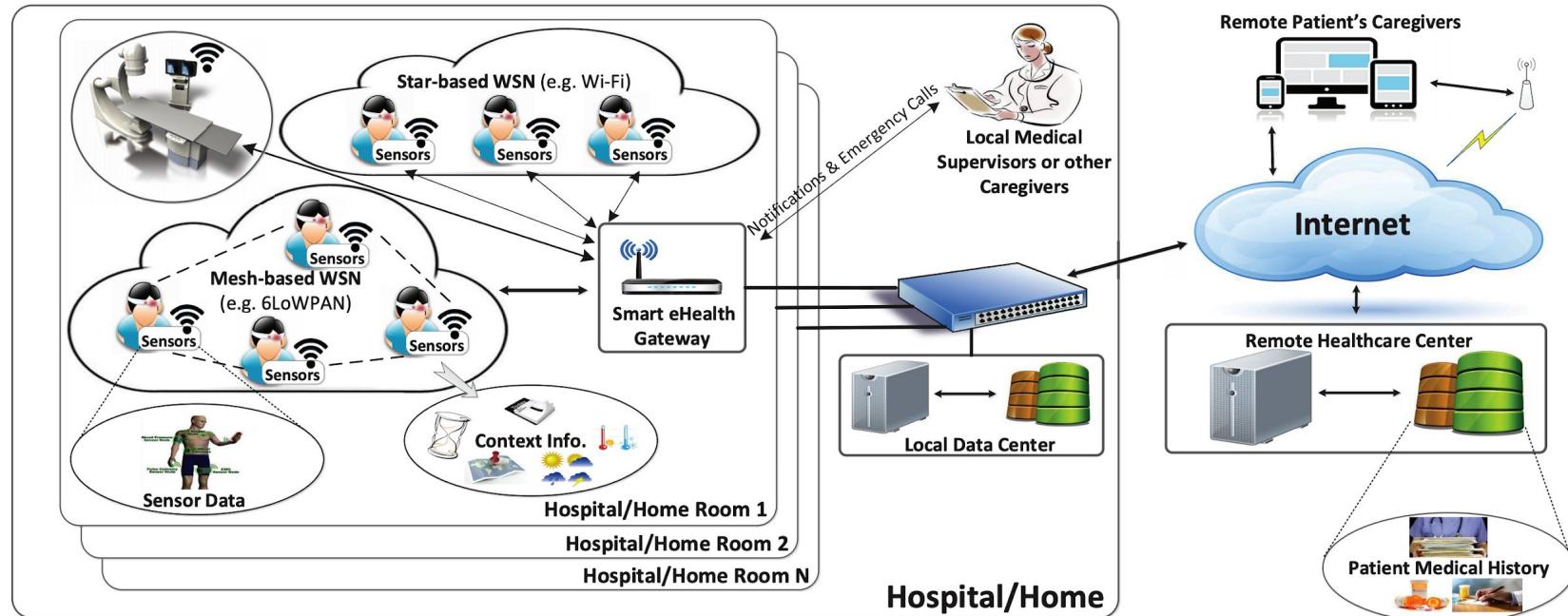
Le singole turbine di generazione di energia infatti devono ruotare alla stessa velocità per garantire un funzionamento corretto, e inoltre questa velocità dipende dal carico sulla rete.

Il caricamento di auto elettriche per esempio (che può richiedere anche 30-40kW) è un carico piuttosto pesante che deve essere gestito. Aggiungere dispositivi di calcolo distribuite lungo la rete può ottimizzare la rete di distribuzione garantendo un monitoring migliore per esempio, o anche per gestire il controllo delle turbine.



# Health Data Management

Data la sensibilità dei dati health, una soluzione è quella di utilizzare il fog per un processamento locale.



A. Rahmani et al., "Smart e-Health Gateway: Bringing intelligence to Internet-of-Things based ubiquitous healthcare systems," 2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC), Las Vegas, NV, 2015, pp. 826-834, doi: 10.1109/CCNC.2015.7158084.

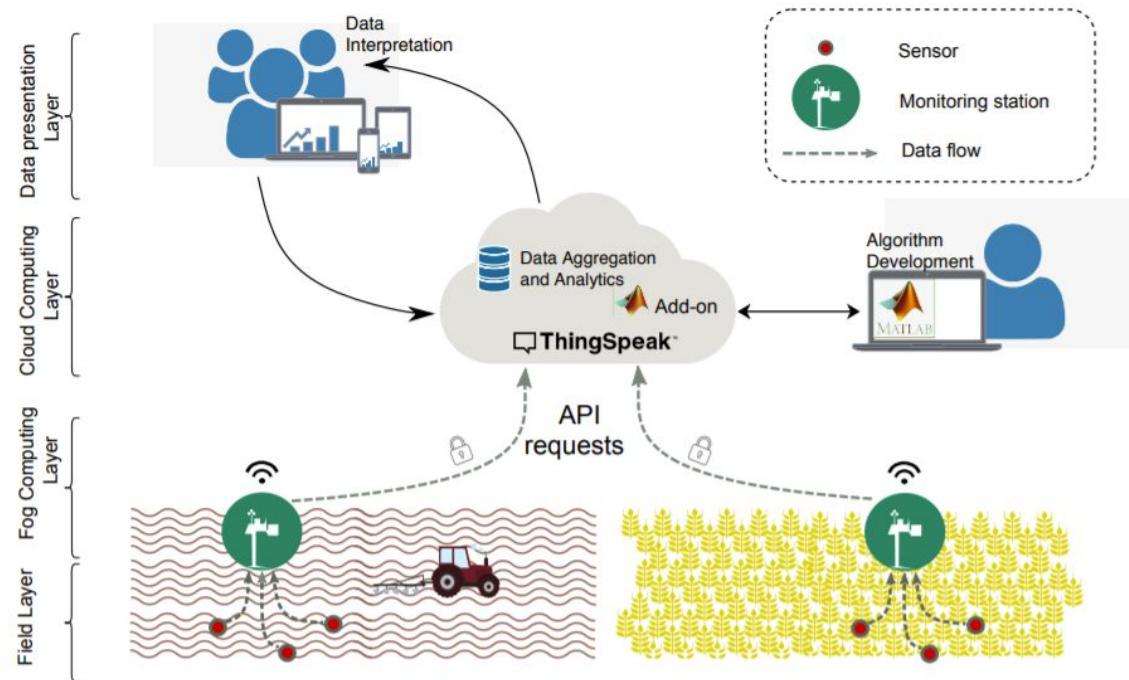
# Shared AR/VR

Vista la bassa latenza un'applicazione per il fog computing è la realizzazione di esperienze di augmented o virtual reality condivise. Il processamento a 30/60fps non è realizzabile con le latenze offerte dal cloud.



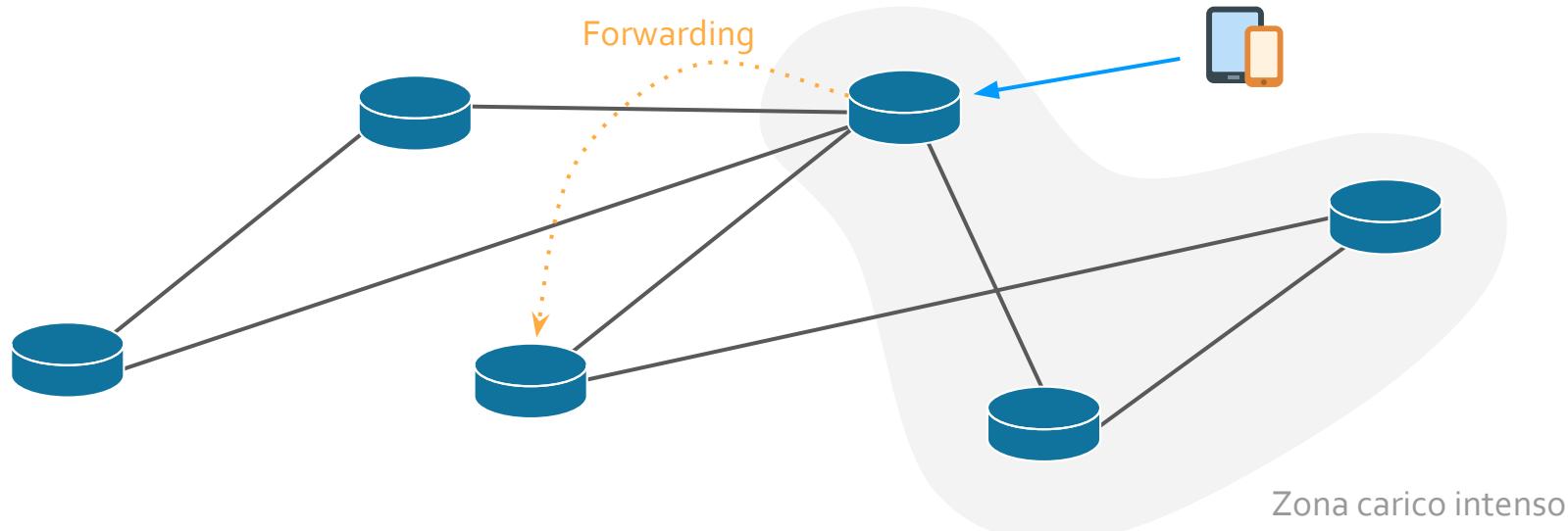
# Agricoltura di precisione

Nell'agricoltura di precisione il Fog Computing può essere usato per l'analisi locale dei parametri e conseguente inoltro al cloud per ulteriore elaborazione e aggregazione.



# P2P Fog Computing

Una sfida particolarmente rilevante per il fog computing riguarda lo studio di soluzioni che consentano ai nodi fog di cooperare, per esempio per bilanciare il carico. In breve, se alcuni nodi fog sono più carichi di altri essi possono redirezionare parte del loro carico su altri nodi che sono meno carichi. Questo implica diversi problemi, dalla scelta della policy di redirezionamento ai vari overhead di comunicazione tra i nodi.



5

# Conclusioni

# Riepilogo

- Il **Fog Computing** è un modello di calcolo distribuito complementare al Cloud. Esso permette di implementare applicazioni che necessitano di latenze molto basse o che comunque non soddisfano i requisiti del Cloud Computing.
- Le 5 **caratteristiche fondamentali** del fog computing sono: *location awareness, geographical distribution, heterogeneity, stream processing e interoperability*.
- Un layer fog computing può essere implementato anche con dispositivi non propri dei datacenter, quali per esempio **PC single board**, che hanno una potenza di calcolo modesta ma a consumi ridotti.
- A livello software i **container** sono un strategia molto efficace per distribuire le applicazioni, soprattutto in ambito fog computing.
- Numerose sono le **applicazioni** del fog computing (e dei suoi sottoprodotto: Edge Computing, Mist Computing, ecc.). Tra le più note sono troviamo smart cities, smart grids, AR/VR e applicazioni health.

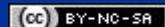
# Fog Computing

Introduzione, tecnologie e sfide

PRESENTATION & TALK

**Gabriele Proietti Mattia**

Dottorando in Engineering in Computer Science



[proiettimattia@diag.uniroma1.it](mailto:proiettimattia@diag.uniroma1.it) · [gpm.name](http://gpm.name)

**DIAG**

# The Road Ahead Future Internet: From IP to ICN

**Enkeleda Bardhi**

Department of Computer, Control and Management Engineering  
Sapienza University of Rome

*bardhi@diag.uniroma1.it*

November 15, 2021



# Outline

1 Context

2 Information Centric Networking

3 Is ICN mature enough? IP-ICN coexistence

4 Secure Transition



# Context (1/3)

## Global mobile data traffic 2017-2022<sup>1</sup>

- A factor 8 increase of mobile data traffic for Asia Pacific
- A factor 11 increase for Middle East and Africa
- A factor 5 increase for the rest of regions

|       | Asia Pacific | Middle East and Africa | Central and Eastern Europe | North America | Western Europe | Latin America |
|-------|--------------|------------------------|----------------------------|---------------|----------------|---------------|
| 2017  | 5.88         | 1.22                   | 1.38                       | 1.26          | 1.02           | 0.75          |
| 2018  | 10.35        | 2.05                   | 2.15                       | 1.8           | 1.47           | 1.18          |
| 2019* | 15.91        | 3.25                   | 3.12                       | 2.5           | 2.06           | 1.72          |
| 2020* | 22.81        | 5.01                   | 4.32                       | 3.41          | 2.81           | 2.42          |
| 2021* | 31.81        | 7.56                   | 5.83                       | 4.48          | 3.8            | 3.31          |
| 2022* | 43.17        | 11.17                  | 7.75                       | 5.85          | 5.12           | 4.44          |

<sup>1</sup>Expressed in hexabytes/month

## Context (2/3)

- Usage model of Internet has changed: more users, more data, more devices/user, connection everywhere and at every time
- Current Internet must cope with some limitations:
  - availability of unique IP addresses
  - decreasing performance of Internet
  - lots of security issues

# Context (3/3)

Substitute the current Internet

- Change the communication paradigm
- In-network caching
- Decoupling senders and receivers
- Better performance (i.e., fast, efficient, and secure data delivery, improved reliability)

# Outline

1 Context

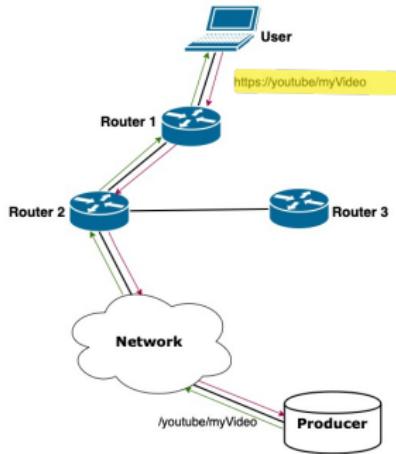
2 Information Centric Networking

3 Is ICN mature enough? IP-ICN coexistence

4 Secure Transition



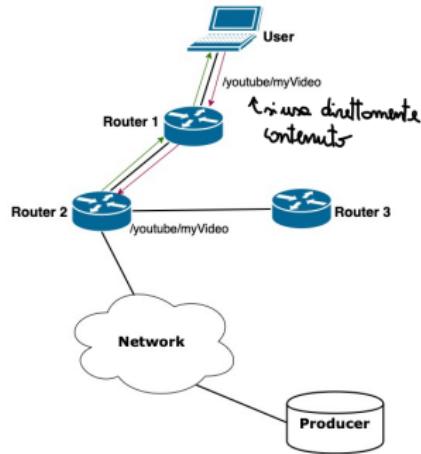
# Information Centric Networking (ICN)<sup>2</sup>



TCP/IP architecture

**Where**

Producer is addressed



ICN architecture

**What**

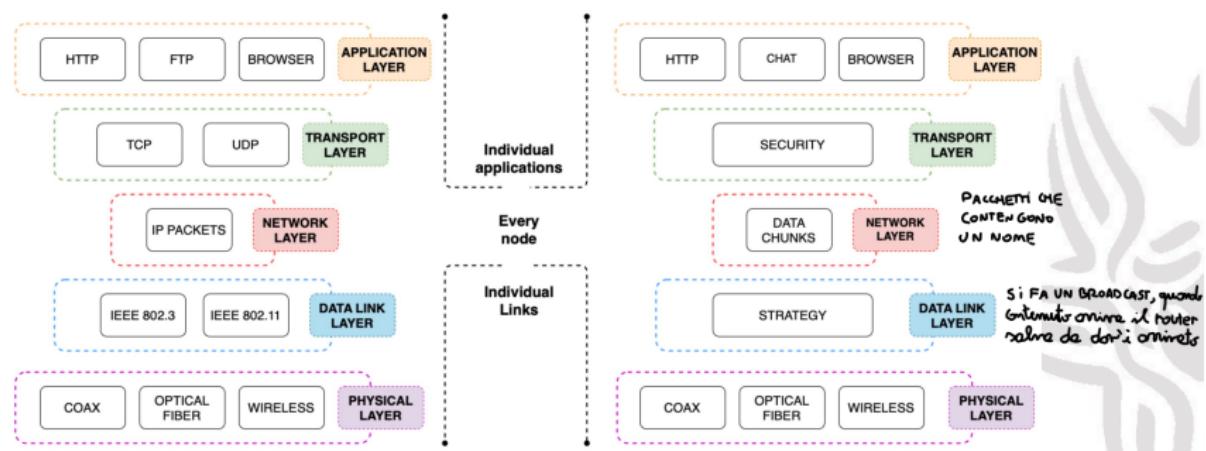
Content is addressed

<sup>2</sup>Bengt Ahlgren et al. "A survey of information-centric networking". In: *IEEE Communications Magazine* 50.7 (2012), pp. 26–36.

# ICN protocol stack

## Protocol stacks

- From IP to chunks of named content
- Universal agreement only in layer 3
- Security-in-mind (transport layer) in ICN



# ICN packet types

## ICN packets

- There are just two ICN packet types, **interest** (similar to HTTP GET) and **data** (similar to HTTP RESPONSE).
- Both are encoded in an efficient binary XML.

| <b>INTEREST PACKET</b>   |
|--|
| Name   |
| <b>Selectors</b><br>(order preference,<br>exclude filter etc.) |
| Nonce  |
| <b>Guiders</b><br>(scope, interest lifetime)                   |

| <b>DATA PACKET</b>  |
|---|
| Name  |
| <b>MetaInfo</b><br>(content type,<br>freshness period etc.) |
| Content   |
| <b>Signature</b><br>(signature type, key locator)           |

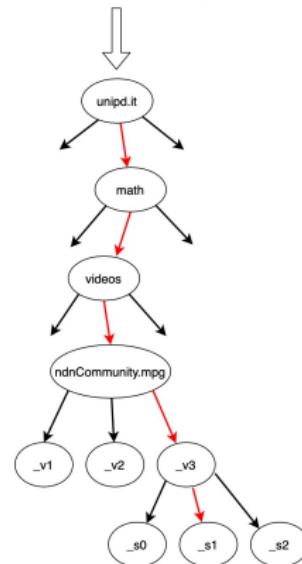
NON È ANONIMO c'è signature

# ICN naming schema

## Content naming schema

- Hierarchical naming schema (similar to URLs).
- A consumer broadcasts interest requests using content name over all available connectivity.
- Data is transmitted in response to interest request if name field of interest packet is a prefix of name field of data packet.

/unipd/math/videos/ndnCommunity.mpg/\_version3/\_s1

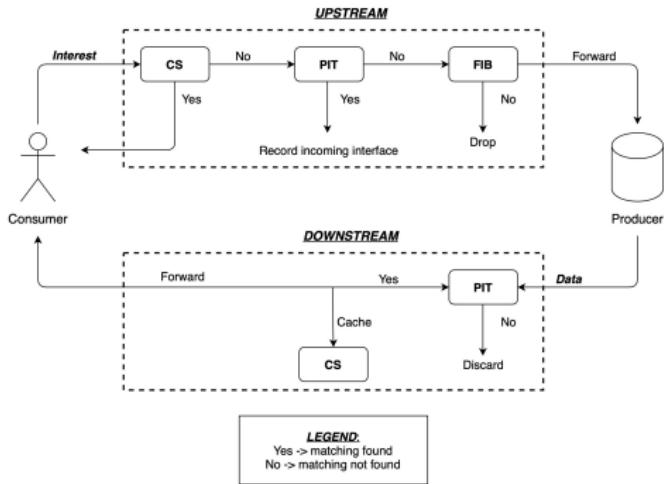


# ICN forwarding and routing

Three components on routers:

- **Content Store (CS)** SIMILAR ROUTING TABLE
- **Pending Interest Table (PIT)**
- **Forwarding Information Base (FIB)**

- FIB allows a list of outgoing interfaces to multiple sources
- CS allows routers to store most popular content
- PIT keeps track of interest forwarded up-stream, such that data can be sent downstream.



# Outline

1 Context

2 Information Centric Networking

3 Is ICN mature enough? IP-ICN coexistence

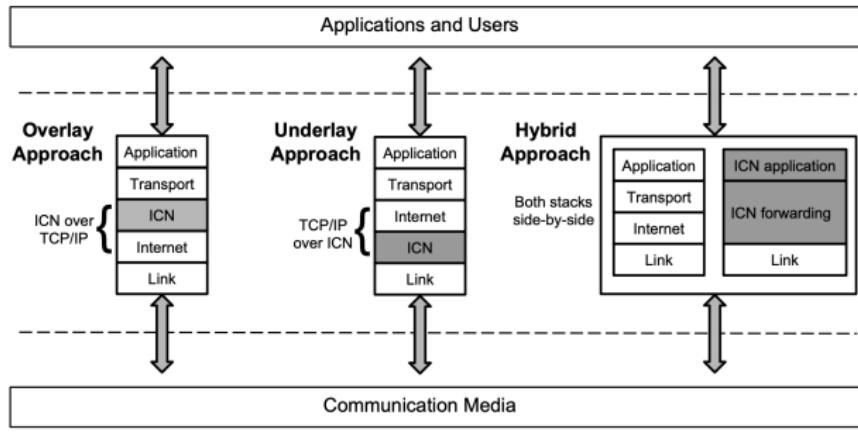
4 Secure Transition



# Coexistence IP-ICN in a nutshell (1/3)

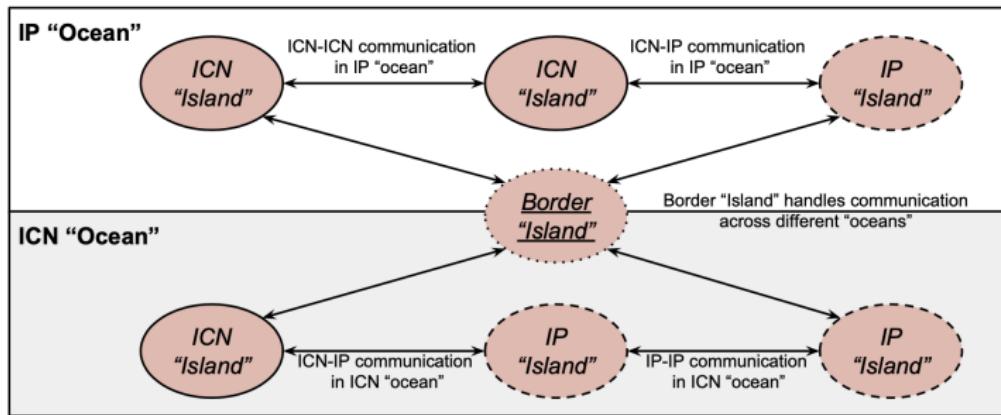
## The road ahead ICN

- Lessons learned from the past: IPv4/IPv6, 3G/4G, 4G/5G ⇒ expected IP/ICN coexistence
- Deployment approaches: overlay, underlay, hybrid



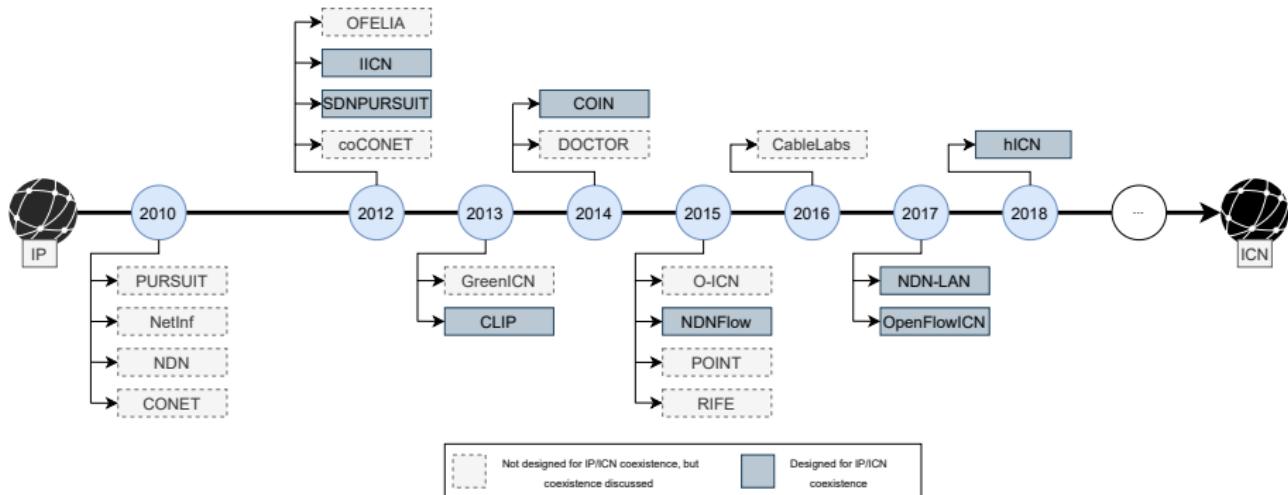
## Coexistence IP-ICN in a nutshell (2/3)

- Researchers foresee the presence of IP, ICN "islands" and IP, ICN "oceans"<sup>3</sup>
- Proposals for different combinations including also other technologies e.g., Software Defined Networking (SDN), Network Function Virtualization (NFV) etc.



<sup>3</sup>Mauro Conti et al. "The road ahead for networking: A survey on icn-ip coexistence solutions". In: *IEEE Communications Surveys & Tutorials* 22.3 (2020), pp. 2104–2129.

# Coexistence IP-ICN in a nutshell (3/3)



# Outline

1 Context

2 Information Centric Networking

3 Is ICN mature enough? IP-ICN coexistence

4 Secure Transition



# Security issues of ICN

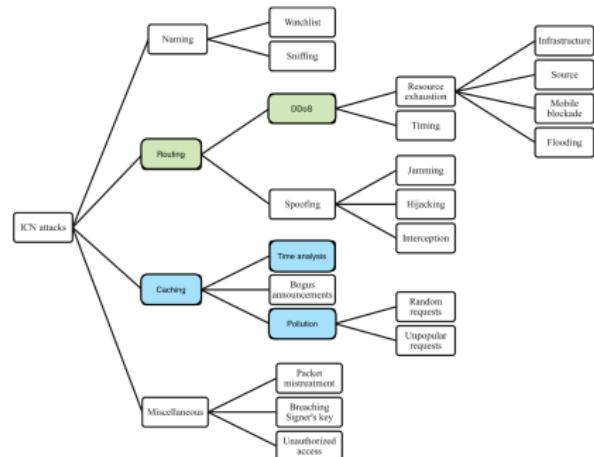
## Vulnerable points of ICN

- *Naming<sup>a</sup>*
- *Routing<sup>b</sup>*
- *Caching<sup>c</sup>*
- *Miscellaneous* - aims at degrading the ICN services

<sup>a</sup>Enkeleda Bardhi et al. "ICN PATTA: ICN Privacy Attacks". In: *Conference on Local Computer Networks (LCN)*. IEEE. 2020.

<sup>b</sup>Alberto Compagno et al. "NDN interest flooding attacks and countermeasures". In: *Annual Computer Security Applications Conference*. 2012.

<sup>c</sup>Gergely Acs et al. "Cache privacy in named-data networking". In: *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE. 2013, pp. 41–51.



## Open issues and challenges

- ① Study and mitigation of IP-ICN coexistence security and privacy issues
- ② Deal with different security models (rather content or host based)
- ③ Selection of the most efficient coexistence approach (overlay, underlay or hybrid)

# Conclusions

## Closing remarks

- Current Internet architecture is facing several limitations
- Information Centric Networking comes as a promising substitute
- A long coexistence between IP and ICN will lead Internet towards its future structure
- IP-ICN coexistence must be well studied and must ensure secure transition to the future Internet

# The Road Ahead Future Internet: From IP to ICN

**Enkeleda Bardhi**

Department of Computer, Control and Management Engineering  
Sapienza University of Rome

*bardhi@diag.uniroma1.it*

November 15, 2021



# References

-  Acs, Gergely et al. "Cache privacy in named-data networking". In: *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE. 2013, pp. 41–51.
-  Ahlgren, Bengt et al. "A survey of information-centric networking". In: *IEEE Communications Magazine* 50.7 (2012), pp. 26–36.
-  Bardhi, Enkeleda et al. "ICN PATTA: ICN Privacy Attack Through Traffic Analysis". In: *2021 IEEE 46th Conference on Local Computer Networks (LCN)*. IEEE. 2021, pp. 443–446.
-  Compagno, Alberto et al. "NDN interest flooding attacks and countermeasures". In: *Annual Computer Security Applications Conference*. 2012.
-  Conti, Mauro et al. "The road ahead for networking: A survey on icn-ip coexistence solutions". In: *IEEE Communications Surveys & Tutorials* 22.3 (2020), pp. 2104–2129.

# System and Network Security

Riccardo Lazzeretti – [lazzeretti@diag.uniroma1.it](mailto:lazzeretti@diag.uniroma1.it)

Chapters 14,15 of William Stallings: Operating Systems: Internals and Design Principles (7<sup>th</sup> ed.)

W. Stallings: "Cryptography and Network Security: Principles and Practice" (seventh edition), Pearson

## What “secure” means?



- . 1) Security is not just “a product” (e.g. a firewall); it is rather a “process”, which needs to be managed properly
- . 2) Nothing is 100% secure

*“The three golden rules for ensuring computer security: do not own a computer; do not power it on; and do not use it.”*  
- Robert (Bob) Morris (Former NSA Chief Scientist).

- 3) The security of a system is equivalent to the security of its less secure component (rule of the weakest link)

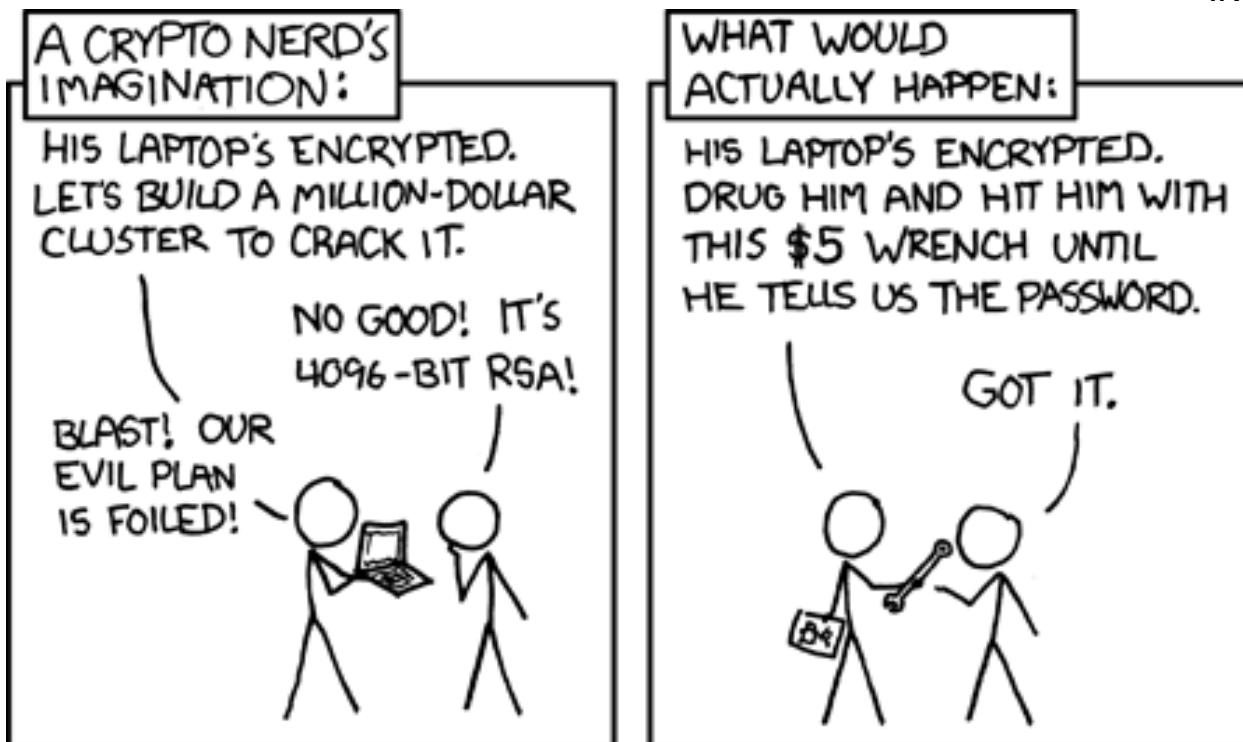


- 4) Security by obscurity never works
- 5) Cryptography is a powerful tool but...  
it is not enough!



*"The protection provided by encryption is based on the fact that most people would rather eat liver than do mathematics"*

Bill Neugent



- 4) Security by obscurity never works
- 5) Cryptography is a powerful tool but...  
it is not enough!



*"The protection provided by encryption is based on the fact that most people would rather eat liver than do mathematics"*

Bill Neugent



- 4) Security by obscurity never works
- 5) Cryptography is a powerful tool but...  
it is not enough!



*"The protection provided by encryption is based on the fact that most people would rather eat liver than do mathematics"*

Bill Neugent



## . 6) Do not rely on users!

*“Given a choice between dancing pigs and security, users will pick dancing pigs everytime.”*

*- Prof. Ed Felten (Princeton University)*



*“If the computer prompts him with a warning screen like: “The applet DANCING PIGS could contain malicious code that might do permanent damage to your computer, steal your life's savings, and impair your ability to have children,” he'll click OK without even reading it. Thirty seconds later he won't even remember that the warning screen even existed”*

*- Bruce Schneier*

So, what “secure” means?  
A network/system is secure when...



# Operating Systems: Internals and Design Principles

*The art of war teaches us to rely not on the likelihood of the enemy's not coming, but on our own readiness to receive him; not on the chance of his not attacking, but rather on the fact that we have made our position unassailable.*

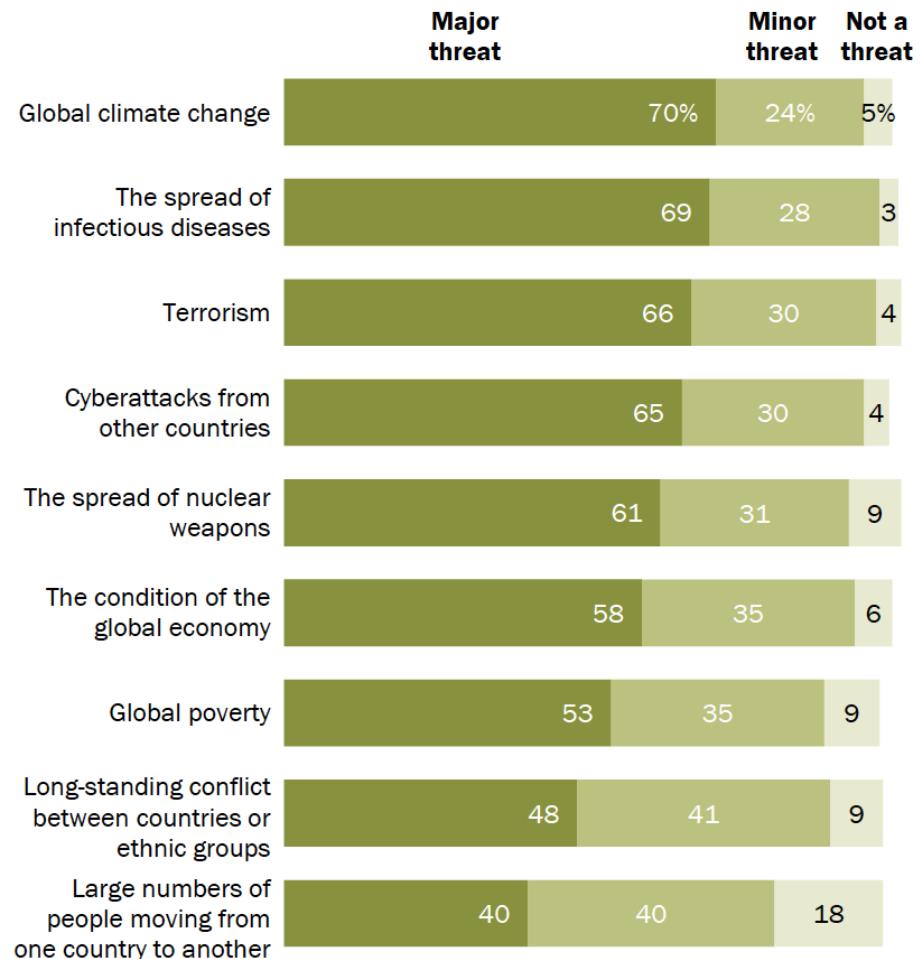


— *THE ART OF WAR,*  
*Sun Tzu*

# Is cybercrime really a problem?

## Across 14 countries polled, climate change and infectious diseases top list of global threats

Median % who say the following are a \_\_\_ to their country



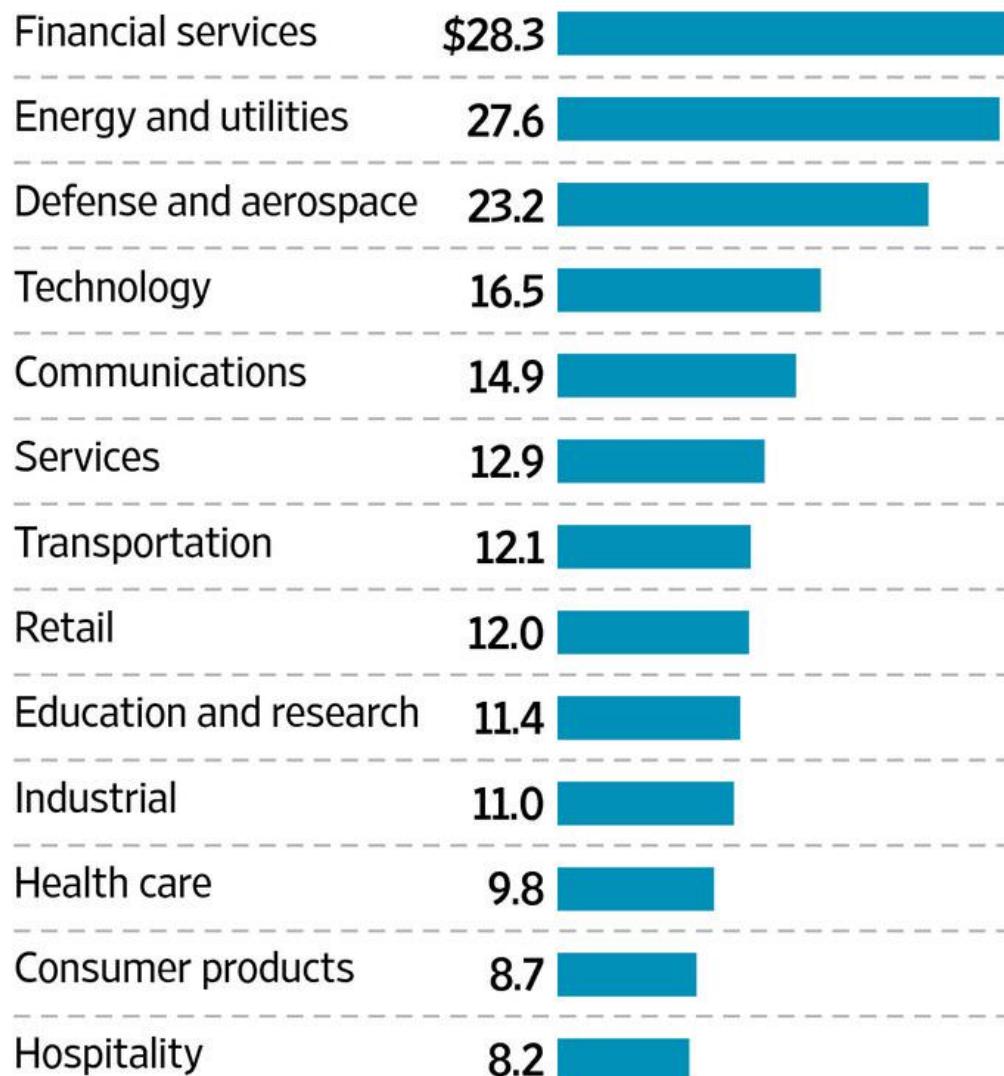
Note: Percentages are medians based on 14 countries surveyed: U.S., Canada, Belgium, Denmark, France, Germany, Italy, Netherlands, Spain, Sweden, UK, Australia, Japan and South Korea. Those who did not answer are not shown.

Source: Spring 2020 Global Attitudes Survey. Q13a-i.

“Despite Pandemic, Many Europeans Still See Climate Change as Greatest Threat to Their Countries”

# Cybercrime Costs

The financial toll of cyberattacks in selected U.S. industries in 2015, in millions



Source: Ponemon Institute/Hewlett Packard survey of  
58 U.S. organizations, August 2015     THE WALL STREET JOURNAL.

## Cyber Security Statistics in 2019

Almost half of all companies have over 1,000 sensitive pieces of information that are not protected



The biggest cost from a cyber attack is productivity



● Attack Cost 23%     ● Productivity Cost 77%

Attacks on healthcare are expected to increase by

400%

in 2020



The cost of cyber crime is expected to exceed

\$6 Trillion

Annually by 2021



# Computer Security

- The NIST Computer Security Handbook defines ***computer security*** as:

*The protection afforded to an automated information system in order to attain the applicable objectives of preserving the integrity, availability and confidentiality of information system resources (includes hardware, software, firmware, information/data, and telecommunications).*



# CIA Triad

- Security Objectives:

- Confidentiality

- a loss of confidentiality is the unauthorized disclosure of information

- Integrity

- a loss of integrity is the unauthorized modification or destruction of information

- Availability

- a loss of availability is the disruption of access to or use of information or an information system

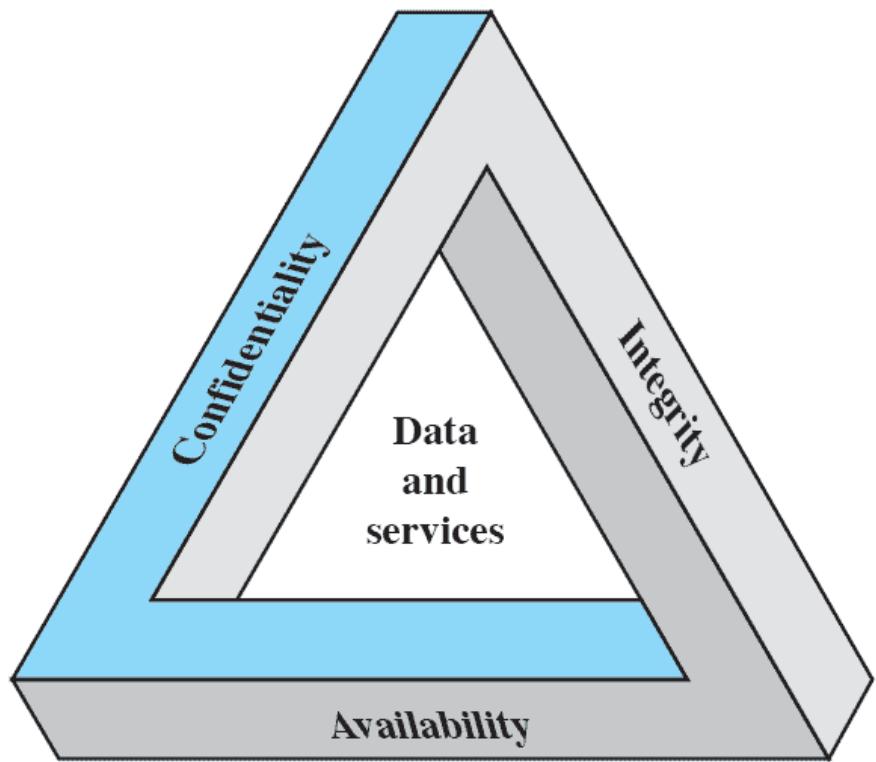


Figure 14.1 The Security Requirements Triad

# Key Objectives of Computer Security

- **Confidentiality**

- ***Data confidentiality*** assures that private or confidential information is not made available or disclosed to unauthorized individuals
- ***Privacy*** assures that individuals control or influence what information related to them may be collected and stored and by whom and to whom that information may be disclosed

- **Integrity**

- ***Data integrity*** assures that information and programs are changed only in a specified and authorized manner
- ***System integrity*** assures that a system performs its intended function in an unimpaired manner, free from deliberate or inadvertent unauthorized manipulation of the system

- **Availability**

- assures that systems work promptly and service is not denied to authorized users

# Additional Concepts

- Two further concepts are often added to the core of computer security:

## Authenticity

- The property of being genuine and being able to be verified and trusted; confidence in the validity of a transmission, a message, or message originator
- Verifying that users are who they say they are and that each input arriving at the system came from a trusted source

## Accountability

- The security goal that generates the requirement for actions of an entity to be traced uniquely to that entity
- We must be able to trace a security breach to a responsible party
- Systems must keep records of their activities to permit later forensic analysis to trace security breaches or to aid in transaction disputes

# Scope of System Security

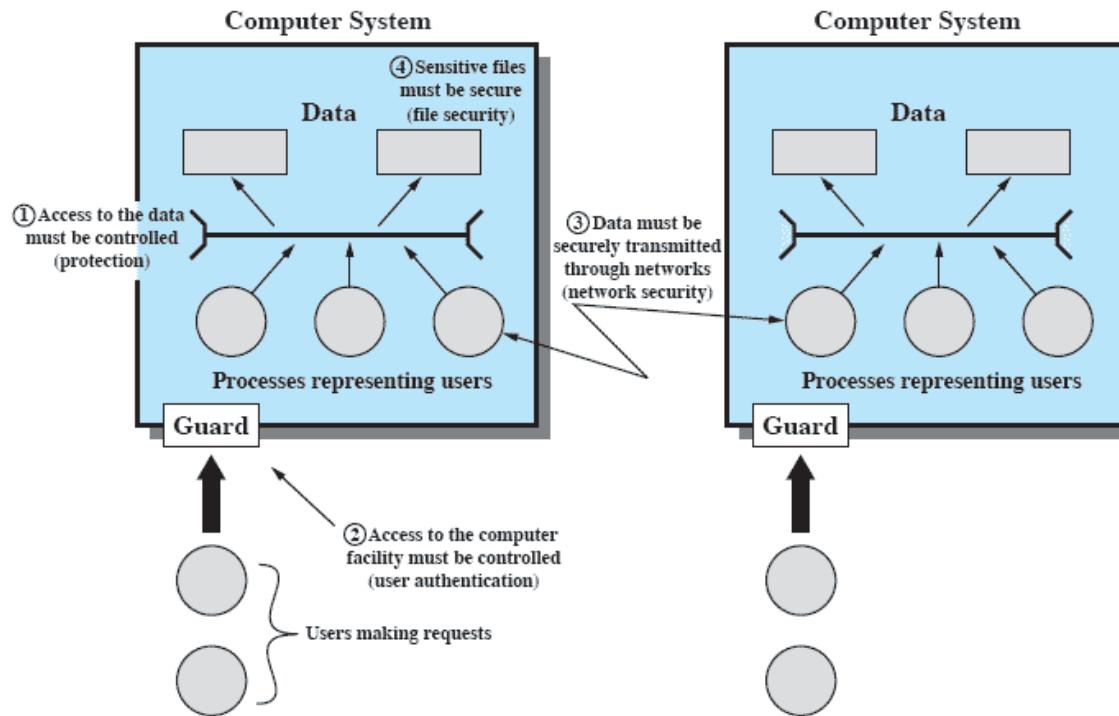


Figure 14.2 Scope of System Security

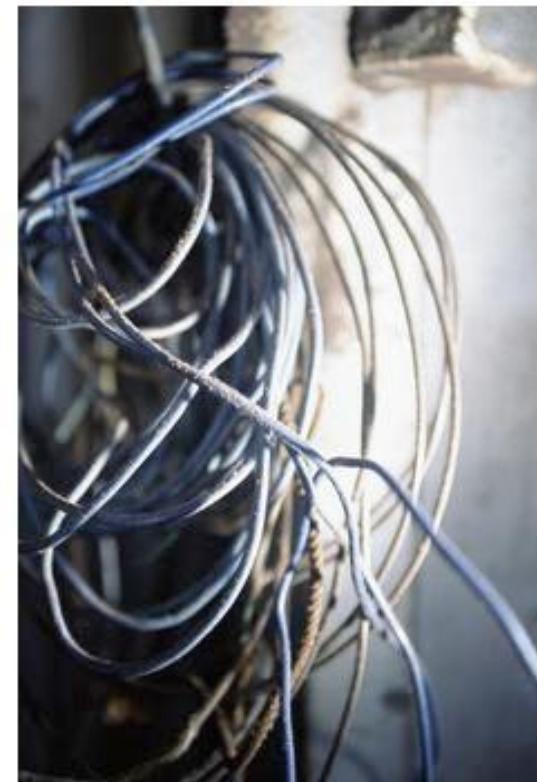
# Examples of Threats

|                            | <b>Availability</b>  | <b>Confidentiality</b>  | <b>Integrity</b>  |
|----------------------------|--|---|---|
| <b>Hardware</b>            | Equipment is stolen or disabled, thus denying service.                                       |   |   |
| <b>Software</b>            | Programs are deleted, denying access to users.   | An unauthorized copy of software is made.   | A working program is modified, either to cause it to fail during execution or to cause it to do some unintended task. |
| <b>Data</b>                | Files are deleted, denying access to users.  | An unauthorized read of data is performed. An analysis of statistical data reveals underlying data. | Existing files are modified or new files are fabricated.  |
| <b>Communication Lines</b> | Messages are destroyed or deleted. Communication lines or networks are rendered unavailable. | Messages are read. The traffic pattern of messages is observed.                                     | Messages are modified, delayed, reordered, or duplicated. False messages are fabricated.                              |

# Cyber Attacks

# Types of attack

- **Passive:** the attacker can only read any information
  - Tempest (signal intelligence)
  - Packet Sniffing
- **Active:** the attacker can read, modify, generate, destroy any information

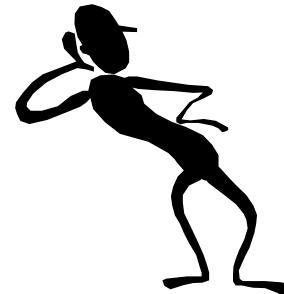


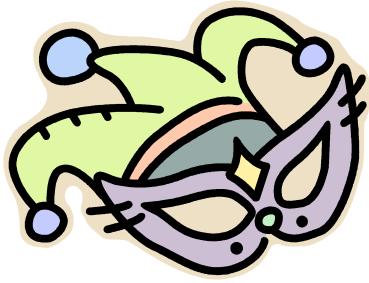
# Passive Attacks

- Attempts to learn or make use of information from the system but does not affect system resources
- Are in the nature of eavesdropping on, or monitoring of, transmissions
- Goal of the attacker is to obtain information that is being transmitted
- Difficult to detect because they do not involve any alteration of the data
  - is feasible to prevent the success of these attacks by means of encryption
- Emphasis in dealing with passive attacks is on prevention rather than detection

Types:

- release of message contents
- traffic analysis





# Active Attacks

- Involve some modification of the data stream or the creation of a false stream
- Four categories:
  1. **Replay**
    - involves the passive capture of a data unit and its subsequent retransmission to produce an unauthorized effect
  2. **Masquerade**
    - takes place when one entity pretends to be a different entity
  3. **Modification of messages**
    - some portion of a legitimate message is altered, or that messages are delayed or reordered, to produce an unauthorized effect
  4. **Denial of service**
    - prevents or inhibits the normal use or management of communications facilities
    - disruption of an entire network either by disabling the network or by overloading it with messages so as to degrade performance

# Intruder Patterns of Behavior: Hackers

- Traditionally those who hack do so for the thrill of it or for status
- Attackers often look for targets of opportunity and then share the information with others
- Benign intruders consume resources and may slow performance for legitimate users
- Intrusion detection systems (IDSs) and intrusion prevention systems (IPSs) are designed to counter this type of hacker threat
- Computer emergency response teams (CERTs) are cooperative ventures who collect information about system vulnerabilities and disseminate it to systems managers

1. Select the target using IP lookup tools such as NSLookup, Dig, and others.
2. Map network for accessible services using tools such as NMAP.
3. Identify potentially vulnerable services
4. Brute force (guess) password.
5. Install remote administration tool
6. Wait for administrator to log on and capture his password.
7. Use that password to access remainder of network.

(a) Hacker

# Intruder Patterns of Behavior: Insider Attacks

- Among the most difficult to detect and prevent
- Can be motivated by revenge or simply a feeling of entitlement
- Employees already have access to and knowledge of the structure and content of corporate databases

1. Create network accounts for themselves and their friends.
2. Access accounts and applications they wouldn't normally use for their daily jobs.
3. E-mail former and prospective employers.
4. Conduct furtive instant-messaging chats.
5. Visit Web sites that cater to disgruntled employees, such as fdcompany.com.
6. Perform large downloads and file copying.
7. Access the network during off hours.

**(c) Internal Threat**

# Intruder Patterns of Behavior: Criminal Enterprise

- Organized groups of hackers
- They meet in underground forums to trade tips and data and coordinate attacks
- Usually have specific targets, or at least classes of targets in mind
  - A common target is a credit card file at an e-commerce server
- Quick in and quick out attacks

1. Act quickly and precisely to make their activities harder to detect.
2. Exploit perimeter through vulnerable ports.
3. Use Trojan horses (hidden software) to leave backdoors for reentry.
4. Use sniffers to capture passwords.
5. Do not stick around until noticed.
6. Make few or no mistakes.

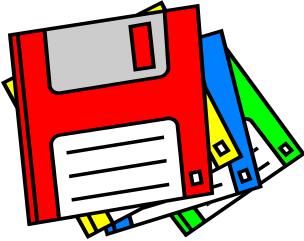
**(b) Criminal Enterprise**

# Intruder Patterns of Behavior: Advanced Persistant Threat

- Organized groups of hackers, state sponsored
- They are composed by skilled individuals (not only hackers)
- They target Critical Infrastructure in ordure to sabotage them or exfiltrate sensitive information
- They perform complex and continuous attacks until they reach their goal
- They stay hidden inside the target organization

- 1) Reconnaissance
- 2) Weaponization
- 3) Delivery
- 4) Exploitation
- 5) Installation
- 6) Contacting Command and Control (C&C) (C2) server
- 7) Action on Objectives
- 8) Conceal Activity

**(b) Advanced Persistent Threat**



# Malware



General term  
for any  
malicious  
software

Software  
designed to  
cause damage  
to or use up  
the resources  
of a target  
computer

Frequently  
concealed  
within or  
masquerades  
as legitimate  
software

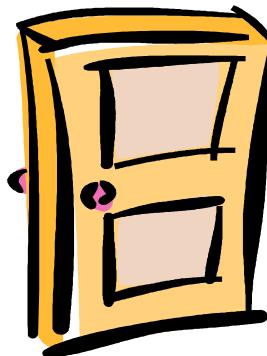
In some cases  
it spreads  
itself to other  
computers via  
e-mail or  
infected discs

# Terminology of Malicious Programs

| Name                      | Description  |
|---------------------------|--|
| Virus                     | Malware that, when executed, tries to replicate itself into other executable code; when it succeeds the code is said to be infected. When the infected code is executed, the virus also executes.  |
| Worm                      | A computer program that can run independently and can propagate a complete working version of itself onto other hosts on a network.  |
| Logic bomb                | A program inserted into software by an intruder. A logic bomb lies dormant until a predefined condition is met; the program then triggers an unauthorized act.   |
| Trojan horse              | A computer program that appears to have a useful function, but also has a hidden and potentially malicious function that evades security mechanisms, sometimes by exploiting legitimate authorizations of a system entity that invokes the Trojan horse program. |
| Backdoor (trapdoor)       | Any mechanisms that bypasses a normal security check; it may allow unauthorized access to functionality.   |
| Platform independent code | Software (e.g., script, macro, or other portable instruction) that can be shipped unchanged to a heterogeneous collection of platforms and execute with identical semantics.   |
| Exploits                  | Code specific to a single vulnerability or set of vulnerabilities.   |
| Downloaders               | Program that installs other items on a machine that is under attack. Usually, a downloader is sent in an e-mail.   |
| Auto-router               | Malicious hacker tools used to break into new machines remotely.   |
| Kit (virus generator)     | Set of tools for generating new viruses automatically.   |
| Spammer programs          | Used to send large volumes of unwanted e-mail.   |
| Flooders                  | Used to attack networked computer systems with a large volume of traffic to carry out a denial-of-service (DoS) attack.  |
| Keyloggers                | Captures keystrokes on a compromised system.   |
| Rootkit                   | Set of hacker tools used after attacker has broken into a computer system and gained root-level access.  |
| Zombie, bot               | Program activated on an infected machine that is activated to launch attacks on other machines.  |
| Spyware                   | Software that collects information from a computer and transmits it to another system.   |
| Adware                    | Advertising that is integrated into software. It can result in pop-up ads or redirection of a browser to a commercial site.  |

# Backdoor

- Also known as a trapdoor
- A secret entry point into a program that allows someone to gain access without going through the usual security access procedures
- A ***maintenance hook*** is a backdoor that programmers use to debug and test programs
- Become threats when unscrupulous programmers use them to gain unauthorized access
- It is difficult to implement operating system controls for backdoors



# Trojan Horse

- Useful, or apparently useful, program or command procedure that contains hidden code that, when invoked, performs some unwanted or harmful function
- Trojan horses fit into one of three models:
  - 1) continuing to perform the function of the original program and additionally performing a separate malicious activity
  - 2) continuing to perform the function of the original program but modifying the function to perform malicious activity or to disguise other malicious activity
  - 3) performing a malicious function that completely replaces the function of the original program

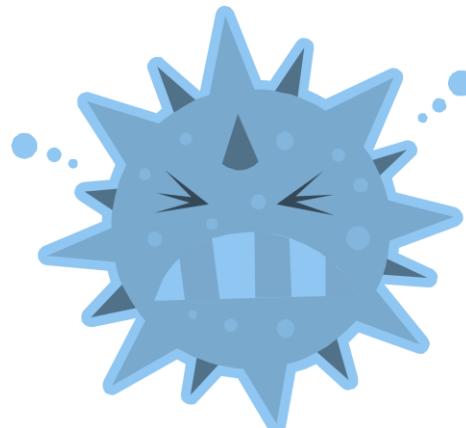


# Platform independent Code

- Programs that can be shipped unchanged to a heterogeneous collection of platforms and execute with identical semantics
- Transmitted from a remote system to a local system and then executed on the local system without the user's explicit instruction
- Often acts as a mechanism for a virus, worm, or Trojan horse to be transmitted to the user's workstation
- Takes advantages of vulnerabilities
- Popular vehicles for mobile code include Java applets, ActiveX, JavaScript, and VBScript

# Viruses

- Software that “infects” other programs by modifying them
  - carries instructional code to self duplicate
  - becomes embedded in a program on a computer
  - when the infected computer comes into contact with an uninfected piece of software, a fresh copy of the virus passes into the new program
  - infection can be spread by swapping disks from computer to computer or through a network
- A computer virus has three parts:
  - an infection mechanism
  - trigger
  - Payload
- Can infect:
  - Boot sector
  - Files
  - Macros

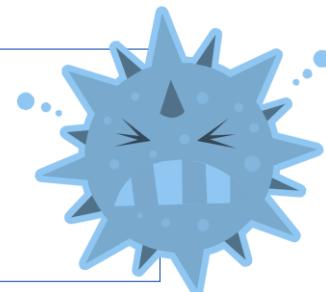


# Multiple-Threat Malware

- Infects in multiple ways
- Typically the multipartite virus is capable of infecting multiple types of files
- A blended attack uses multiple methods of infection or transmission to maximize the speed of contagion and the severity of the attack
- An example of a blended attack is the Stuxnet attack

Stuxnet uses four distribution methods:

- Usb infection
- Windows vulnerabilities
- Network analysis and Privilege escalation
- Specific applications for PLA programming



# Rootkit

- Set of programs installed on a system to maintain administrator (or root) access to that system
- Root access provides access to all the functions and services of the operating system
- The rootkit alters the host's standard functionality in a malicious and stealthy way
  - with root access an attacker has complete control of the system and can add or change programs and files, monitor processes, send and receive network traffic, and get backdoor access on demand
- A rootkit hides by subverting the mechanisms that monitor and report on the processes, files, and registries on a computer

# Rootkit Classification

- Rootkits can be classified based on whether they can survive a reboot and execution mode
- A rootkit may be:

## ***Persistent***

- activates each time the system boots

## ***Memory based***

- has no persistent code and therefore cannot survive a reboot

## ***User mode***

- intercepts calls to APIs and modifies returned results

## ***Kernel mode***

- can intercept calls to native APIs in kernel mode
- can hide the presence of a malware process by removing it from the kernel's list of active processes

# Buffer Overflow

- a very common attack mechanism
  - from 1988 Morris Worm to Code Red, Slammer, Sasser and many others
- prevention techniques known
- still of major concern due to
  - legacy of widely deployed buggy
  - continued careless programming techniques

# Buffer overflow

# Buffer Overflow Basics

- caused by programming error
- allows more data to be stored than capacity available in a fixed sized buffer
  - buffer can be on stack, heap, global data
- overwriting adjacent memory locations
  - corruption of program data
  - unexpected transfer of control
  - memory access violation
  - execution of code chosen by attacker

# Buffer Overflow Attacks

- to exploit a buffer overflow an attacker
  - must identify a buffer overflow vulnerability in some program
    - inspection, tracing execution, fuzzing tools
  - understand how buffer is stored in memory and determine potential for corruption

# How the stack works

The **program stack** (aka function stack, runtime stack) holds *stack frames* (aka *activation records*) for each function that is invoked.

- Very common mechanism for high-level language implementation
  - Exact mechanisms vary by CPU, OS, language, compiler, compiler flags.
- So has special CPU support
  - Stack pointer register: ESP
  - Frame pointer register: EBP
  - push and pop machine instructions

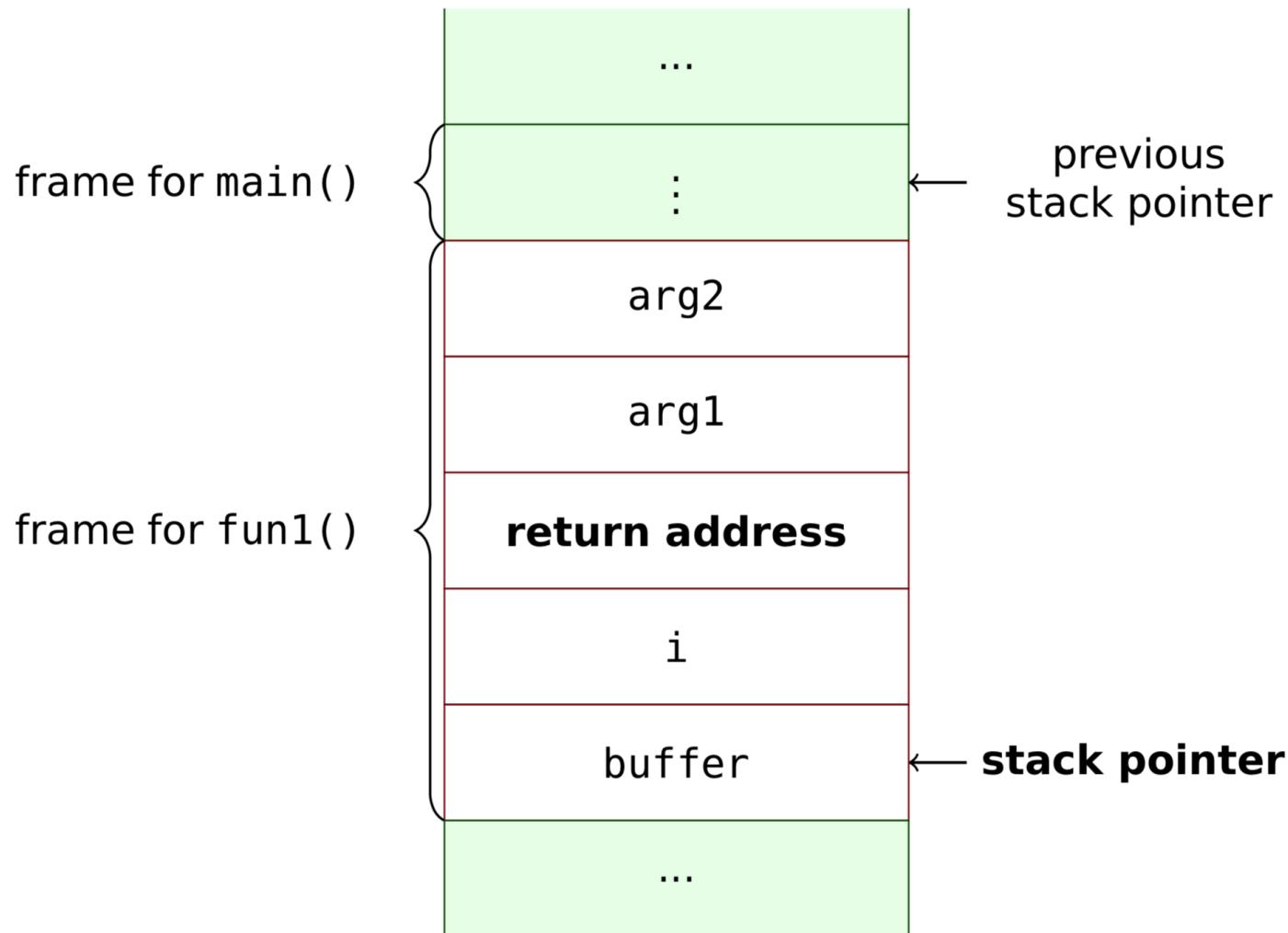
# Stack usage with function calls

- Parameters may be passed to the function body on the stack or in registers; the precise mechanism is called the **calling convention**.
- Local variables are allocated space on the stack.
- A **frame pointer** may be used to help locate arguments and local variables.

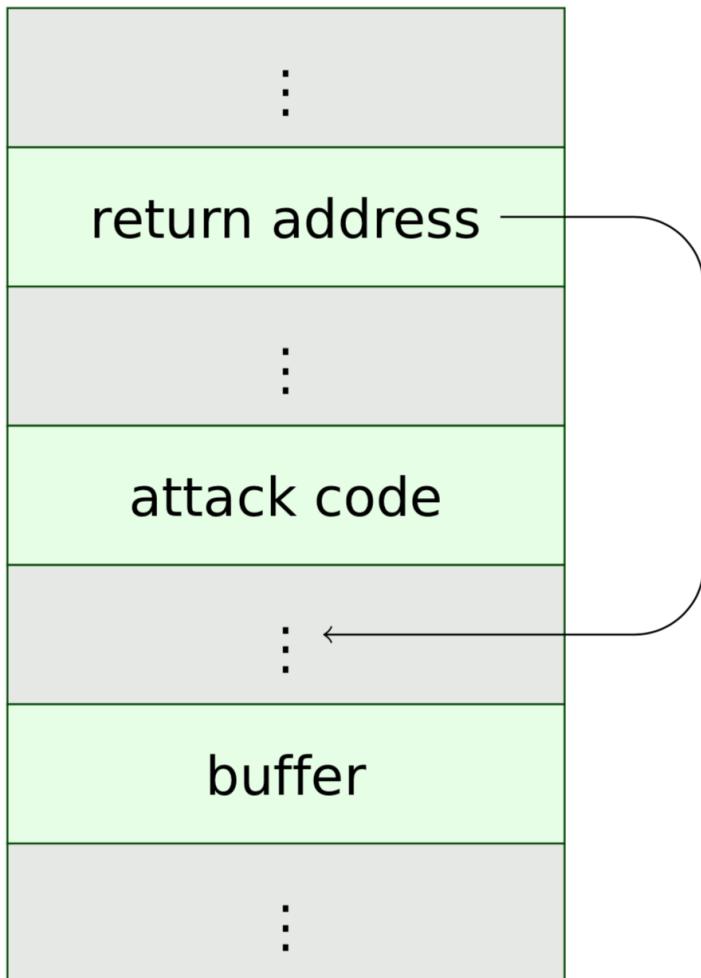
## An example

```
void fun1(char arg1, int arg2) {  
    int i;  
    char buffer[5];  
    buffer[0] = (char)i;  
}  
void main() {  
    fun1('a', 77);  
}
```

# Stack usage with function calls



# Stack overflow: high-level view



- The malicious payload overwrites all of the space allocated for the buffer, all the way to the return address location.
- The return address is altered to point back into the stack, somewhere before the attack code.
- Typically, the attack code executes a shell.

# So all the attacker needs to do...

- ... is stick a program in the buffer or environment!
  - Easy: attacker controls what goes in the buffer!
  - For instance, the payload starts a command shell from which the attacker can control the compromised machine
    - hence the name “shellcode”
  - What does such code look like?

# An example

```
void interact_with_user(char* pwd) {
    int authorized = check_auth(pwd); // 1 if right password provided
                                      // 0 otherwise
    char buf[256];
    printf("Enter your data: ");
    gets(buf);
    if (authorized) store_input_data(buf);
    else printf("Denied!");
}

int main(int argc, char** argv) {
    if (argc == 2) interact_with_user(argv[1]);
    return 0;
}
```

- 2 vulnerabilities:
  - I can set authorized without knowing the password
  - I can change the return pointer
    - Execute code in memory or in buf

# Role of programming languages

- Low-level languages manipulate memory directly
- Advantage: efficient, precise
- Disadvantage: easy to violate data abstractions
  - arbitrary access to memory
  - pointers and **pointer arithmetic**
  - mistakes violate *memory safety*

A programming language or analysis tool is said to enforce **memory safety** if it ensures that reads and writes stay within clearly defined memory areas, belonging to different parts of the program. Memory areas are often delineated with *types* and a *typing discipline*.

# How do we stop the attacks?

- Code more carefully!
- The best defense is proper **bounds checking**
  - sometimes it is not obvious how to do it
  - programmers are bound to forget it
- None of the existing countermeasures fully stop the problem of buffer overflows



➔ Are there any *system* defenses that can help?

# Buffer Overflow

Protection from stack buffer overflows can be broadly classified into two categories:



## ***Compile-time defenses***

- aims to harden programs to resist attacks in new programs



## ***Stack protection mechanisms***

- aims to detect and abort attacks in existing programs

# Compile Time Defenses

- Aim to prevent or detect buffer overflows by instrumenting programs when they are compiled
  - Choice of Programming Language
  - Safe Coding Techniques
  - Language Extensions and Use of Safe Libraries
  - Stack Protection Mechanisms



# Ok, what about avoiding some functions?

- Many C library functions do not check input size
  - gets is inherently unsafe and was recently removed
  - strcpy is safe as long as you're an educated C programmer 😊
    - If you pass a string of unknown length to it, or some data chunk that does not end with \0, you clearly made a mistake
    - Similar considerations apply for instance to strcat
- Variants of such functions explicitly check input size
  - strncpy is often suggested as replacement for strcpy...
    - However, did you know that no \0 character is appended at the end of destination if source is longer than the specified number of bytes?
    - Things like strlcpy and strlcat are better still!
  - snprintf is typically better than sprintf (at least in C99)
  - fgets is surely better than gets, but consider getline as well

# How do we stop the attacks?

A variety of tricks in combination:

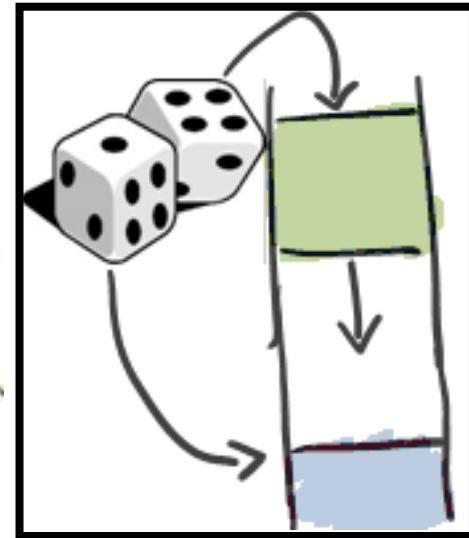
NX bit



Canaries



ASLR

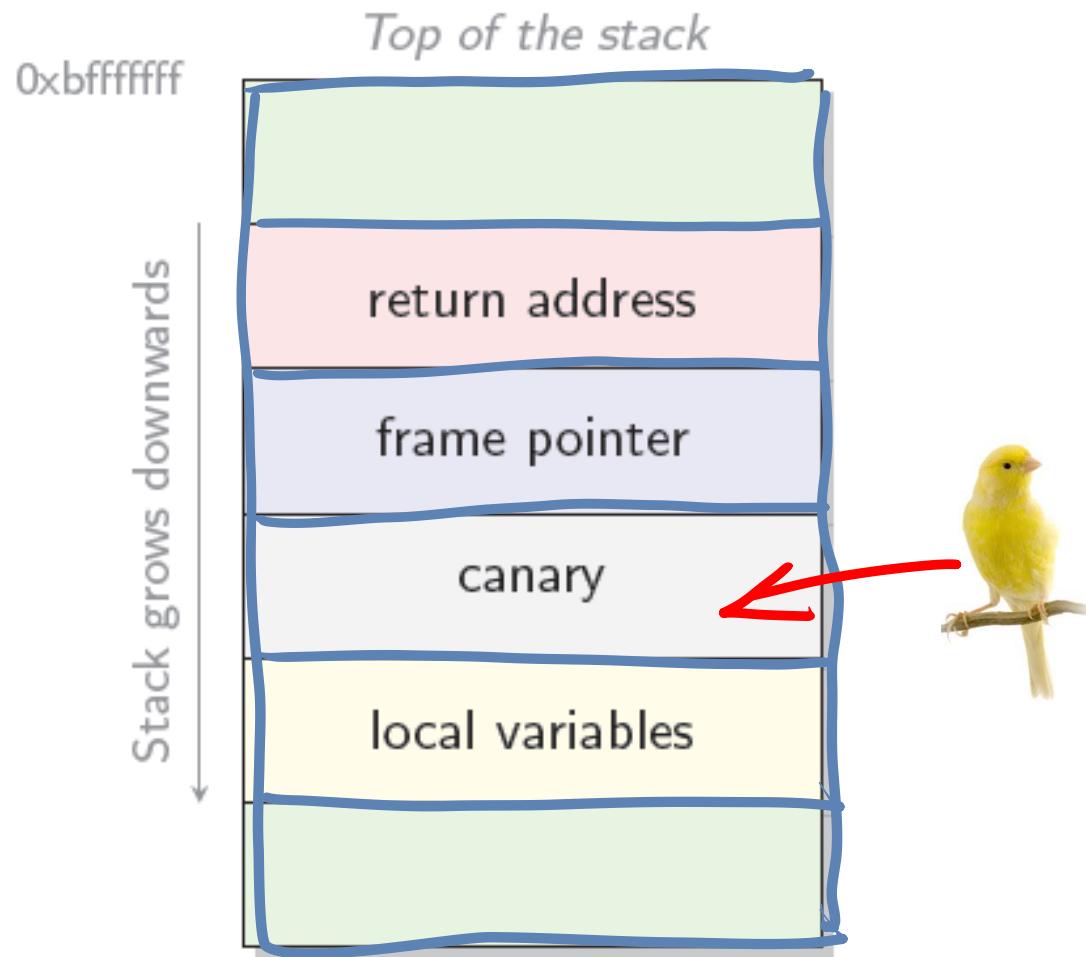


# Compiler-level techniques

## Canaries

- Goal: make sure we detect overflow of return address
  - The functions' prologues insert a *canary* on the stack
  - The canary is a 32-bit value inserted between the return address and local variables
- Types of canaries:
  1. Terminator
  2. Random
  3. Random XOR
- The epilogue checks if the canary has been altered
- Drawback: requires recompilation

# Canaries



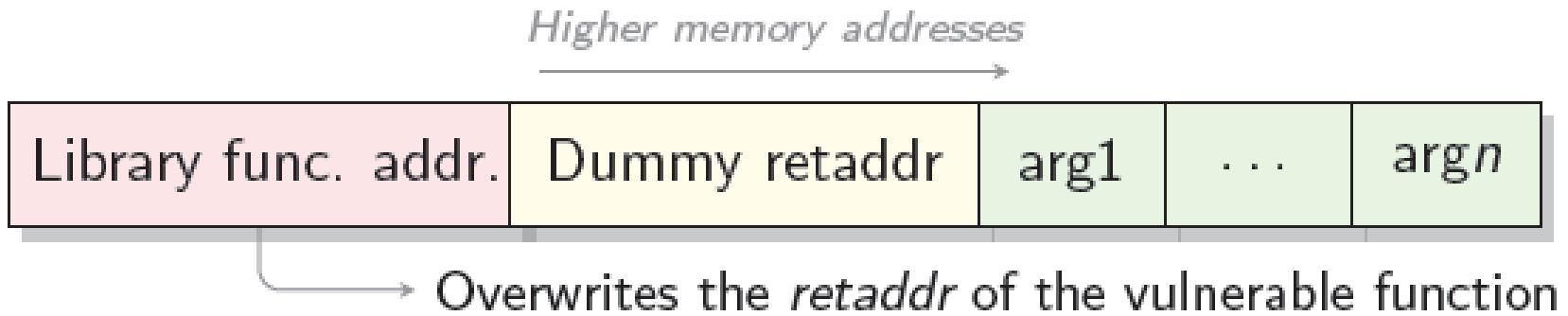
# System-level techniques

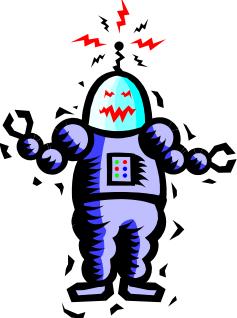
## DEP / NX bit / W⊕X

- Idea: separate executable memory locations from writable ones
  - A memory page cannot be both writable and executable at the same time
- “Data Execution Prevention (DEP)”

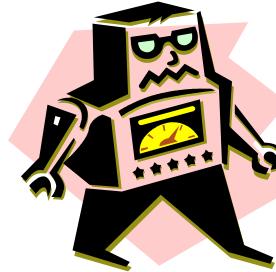
# Bypassing W⊕X

- Return into libc
- Three assumptions:
  - We can manipulate a code pointer (e.g., return address)
  - The stack is writable
  - We know the address of a “suitable” library function (e.g., `system()` can spawn a shell to run a command)





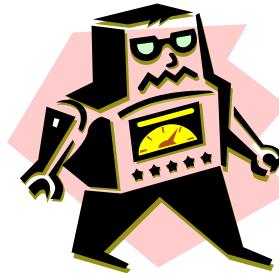
# Bots



- A program that secretly takes over another Internet-attached computer and then uses that computer to launch attacks that are difficult to trace to the bot's creator
  - also known as a Zombie or drone
- Typically planted on hundreds or thousands of computers belonging to unsuspecting third parties
- Collection of bots acting in a coordinated manner is a **botnet**
- A botnet exhibits three characteristics:
  - 1) the bot functionality
  - 2) a remote control facility
  - 3) a spreading mechanism to propagate the bots and construct the botnet

# Constructing the Attack Network

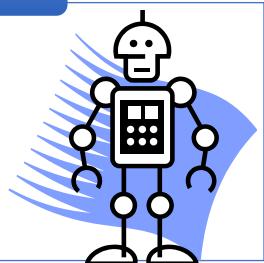
- The first step in a botnet attack is for the attacker to infect a number of machines with bot software that will ultimately be used to carry out the attack
- Essential ingredients:
  - 1) software that can carry out the attack
  - 2) a vulnerability in a large number of systems
  - 3) strategy for locating and identifying vulnerable machines (a process known as scanning or fingerprinting)



- In the scanning process the attacker first seeks out a number of vulnerable machines and infects them
  - the bot software in the infected machines repeats the same scanning process until a large distributed network of infected machines is created

## Scanning strategies:

- Random
- Hit list
- Topological
- Local subnet



# Uses of Bots

## Distributed denial-of-service (DDoS) attacks

- causes a loss of service to users

## Spamming

- sending massive amounts of bulk e-mail (spam)

## Sniffing traffic

- a packet sniffer is used to retrieve sensitive information like user names and passwords

## Keylogging

- captures keystrokes

## Spreading new malware

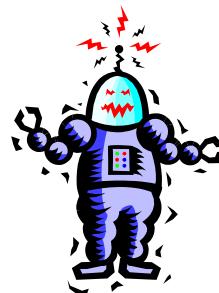
- botnets are used to spread new bots

## Installing advertisement add-ons and browser helper objects (BHOs)

- set up a fake Web site and negotiate a deal with hosting companies that pay for clicks on ads

## Attacking Internet Relay chat (IRC) chat networks

- victim is flooded with requests, bringing down the IRC network; similar to a DDoS attack



## Manipulating online polls/games

- every bot has a distinct IP address so it appears to be a real person

# Denial of Service

- **denial of service** (DoS): an action that prevents or impairs the authorized use of networks, systems, or applications by exhausting resources such as central processing units (CPU), memory, bandwidth, and disk space
- attacks
  - network bandwidth
  - system resources
  - application resources
- have been an issue for some time

# Classic Denial of Service Attacks

- can use simple flooding ping
- from higher capacity link to lower
- causing loss of traffic
- source of flood traffic easily identified

# Types of Flooding Attacks

classified based on network protocol used

## ➤ ICMP Flood

- uses ICMP packets, eg echo request
- typically allowed through, some required

## ➤ UDP Flood

- alternative uses UDP packets to some port

## ➤ TCP SYN Flood

- use TCP SYN (connection request) packets
- but for volume attack

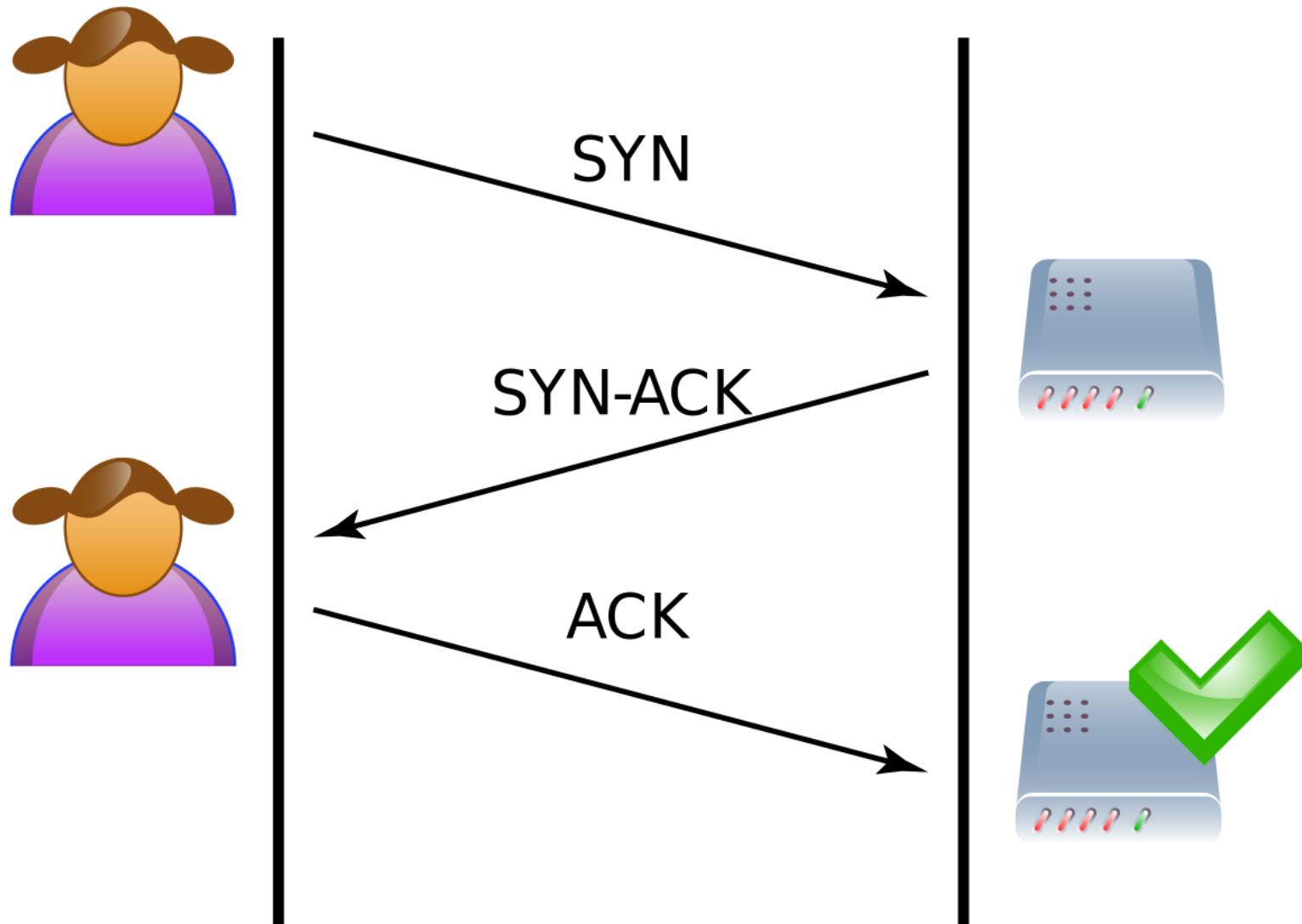
# SYN Spoofing Attack

- attacker often uses either
  - random source addresses
  - or that of an overloaded server
  - to block return of (most) reset packets
- has much lower traffic volume
  - attacker can be on a much lower capacity link

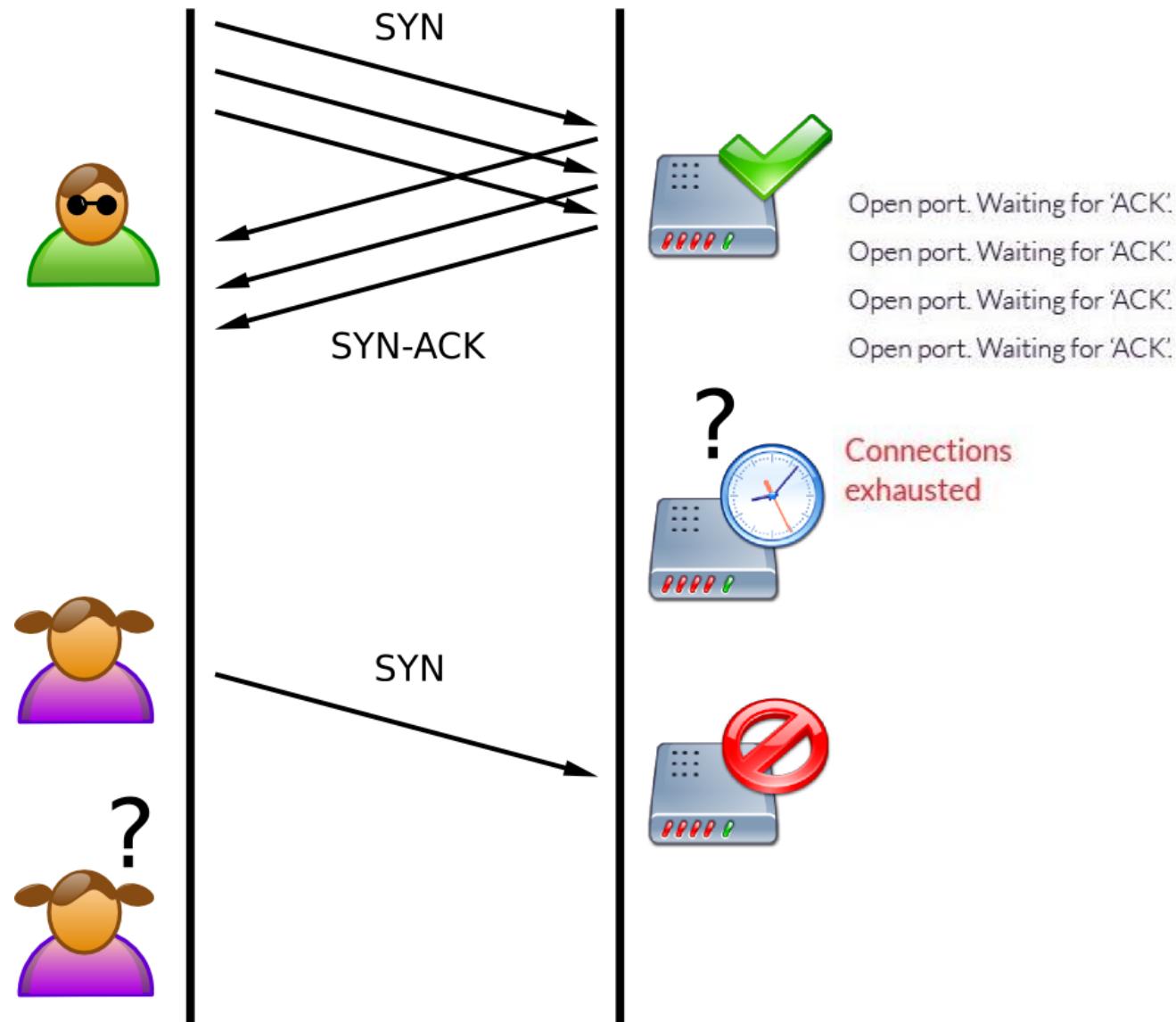
# Source Address Spoofing

- use forged source addresses
  - given sufficient privilege to “raw sockets”
  - easy to create
- generate large volumes of packets
- directed at target
- with different, random, source addresses
- cause same congestion
- responses are scattered across Internet
- real source is much harder to identify

# TCP Connection Handshake



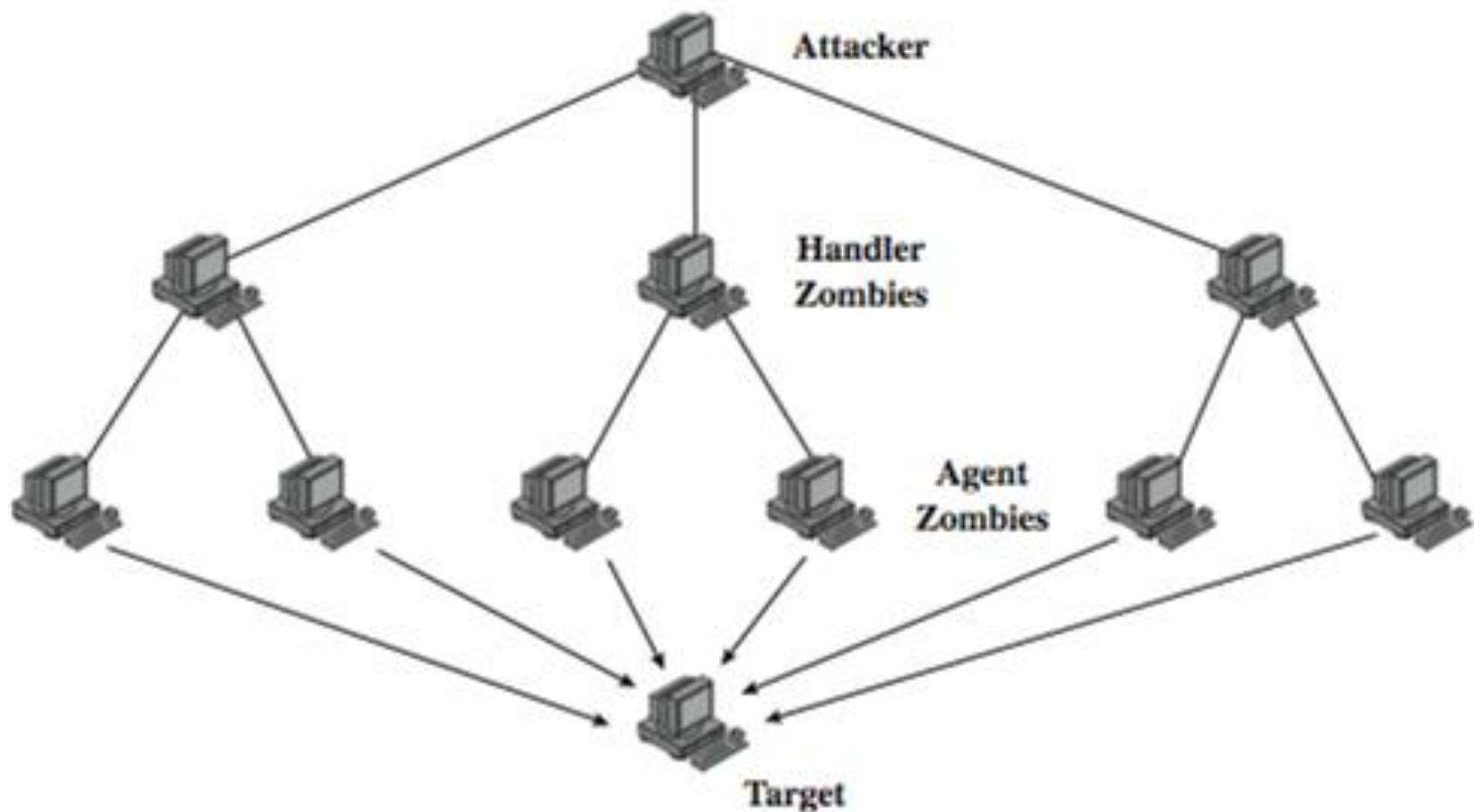
# SYN Spoofing Attack



# Distributed Denial of Service Attacks

- have limited volume if single source used
- multiple systems allow much higher traffic volumes to form a Distributed Denial of Service (DDoS) Attack
- often compromised PC's / workstations
  - zombies with backdoor programs installed
  - forming a botnet

# DDoS Control Hierarchy



# DoS Attack Defenses

- high traffic volumes may be legitimate
  - result of high publicity, e.g. “slash-dotted”
  - or to a very popular site, e.g. Olympics etc
- or legitimate traffic created by an attacker
- three lines of defense against (D)DoS:
  - attack prevention and preemption
  - attack detection and filtering
  - attack source traceback and identification

# Attack Prevention

- block spoofed source addresses
  - on routers as close to source as possible
  - still far too rarely implemented
- rate controls in upstream distribution nets
  - on specific packets types
  - e.g. some ICMP, some UDP, TCP/SYN
- use modified TCP connection handling
  - use SYN cookies when table full
  - or selective or random drop when table full

# Attack Prevention

- block IP directed broadcasts
- block suspicious services & combinations
- manage application attacks with “puzzles” to distinguish legitimate human requests
- good general system security practices
- use mirrored and replicated servers when high-performance and reliability required

# Responding to Attacks

- need good incident response plan
  - with contacts for ISP
  - needed to impose traffic filtering upstream
  - details of response process
- have standard filters
- ideally have network monitors and IDS
  - to detect and notify abnormal traffic patterns

# Responding to Attacks

- identify type of attack
  - capture and analyze packets
  - design filters to block attack traffic upstream
  - or identify and correct system/application bug
- have ISP trace packet flow back to source
  - may be difficult and time consuming
  - necessary if legal action desired
- implement contingency plan
- update incident response plan

# Protection techniques

# Cryptographic tools

# Cryptographic Tools

- cryptographic algorithms are important element in security services
- review various types of elements
  - symmetric encryption
  - public-key (asymmetric) encryption
  - digital signatures and key management
  - secure hash functions

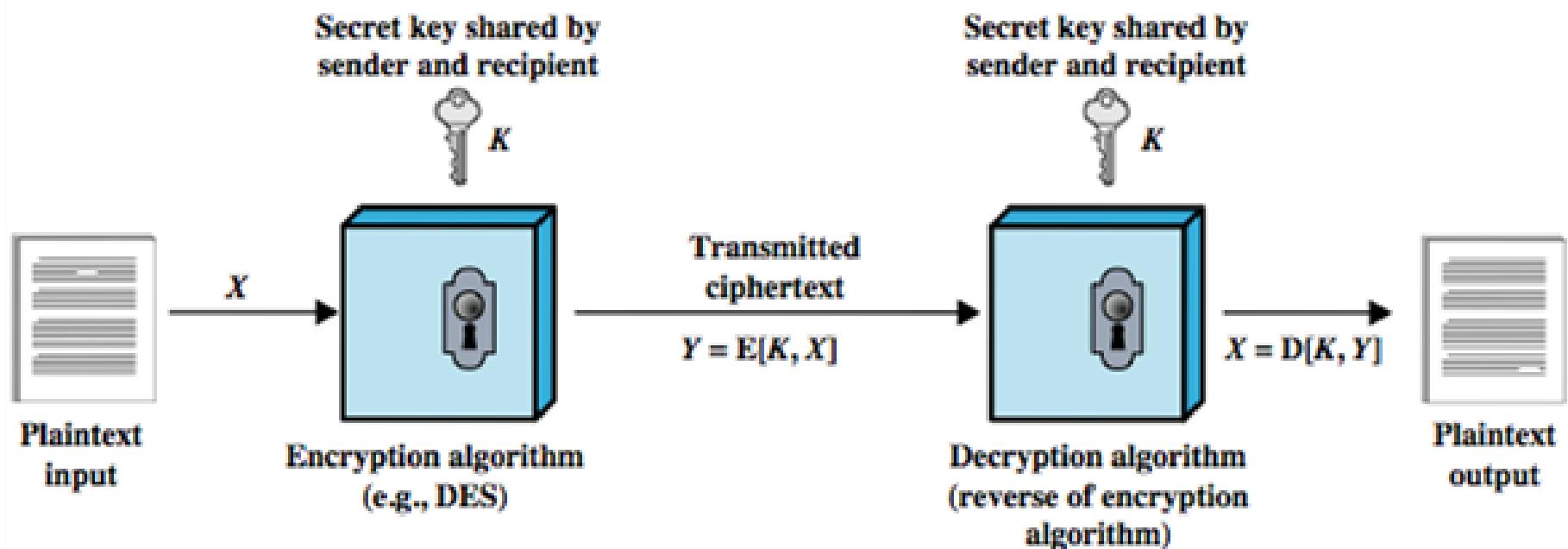
# Random Numbers

- random numbers have a range of uses
- requirements:
- randomness
  - based on statistical tests for uniform distribution and independence
- unpredictability
  - successive values not related to previous
  - clearly true for truly random numbers
  - but more commonly use generator

# Pseudorandom vs. Random Numbers

- often use algorithmic technique to create pseudorandom numbers
  - which satisfy statistical randomness tests
  - but likely to be predictable
- true random number generators use a nondeterministic source
  - e.g. radiation, gas discharge, leaky capacitors
  - increasingly provided on modern processors

# Symmetric Encryption



# Attacking Symmetric Encryption

➤ cryptanalysis

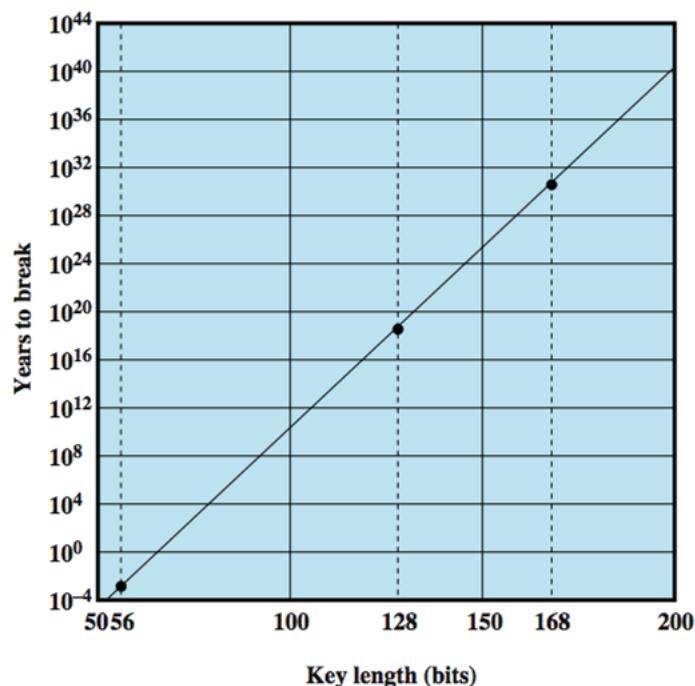
- rely on nature of the algorithm
- plus some knowledge of plaintext characteristics
- even some sample plaintext-ciphertext pairs
- exploits characteristics of algorithm to deduce specific plaintext or key

➤ brute-force attack

- try all possible keys on some ciphertext until get an intelligible translation into plaintext

# Exhaustive Key Search

| Key Size (bits)             | Number of Alternative Keys     | Time Required at 1 Decryption/ $\mu$ s                    | Time Required at $10^6$ Decryptions/ $\mu$ s |
|-----------------------------|--------------------------------|---|--|
| 32                          | $2^{32} = 4.3 \times 10^9$     | $2^{31} \mu\text{s} = 35.8$ minutes                       | 2.15 milliseconds                            |
| 56                          | $2^{56} = 7.2 \times 10^{16}$  | $2^{55} \mu\text{s} = 1142$ years                         | 10.01 hours                                  |
| 128                         | $2^{128} = 3.4 \times 10^{38}$ | $2^{127} \mu\text{s} = 5.4 \times 10^{24}$ years          | $5.4 \times 10^{18}$ years                   |
| 168                         | $2^{168} = 3.7 \times 10^{50}$ | $2^{167} \mu\text{s} = 5.9 \times 10^{36}$ years          | $5.9 \times 10^{30}$ years                   |
| 26 characters (permutation) | $26! = 4 \times 10^{26}$       | $2 \times 10^{26} \mu\text{s} = 6.4 \times 10^{12}$ years | $6.4 \times 10^6$ years                      |



# Symmetric Encryption Algorithms

|                              | DES | Triple DES | AES              |
|------------------------------|-----|------------|------------------|
| Plaintext block size (bits)  | 64  | 64         | 128              |
| Ciphertext block size (bits) | 64  | 64         | 128              |
| Key size (bits)              | 56  | 112 or 168 | 128, 192, or 256 |

DES = Data Encryption Standard

AES = Advanced Encryption Standard

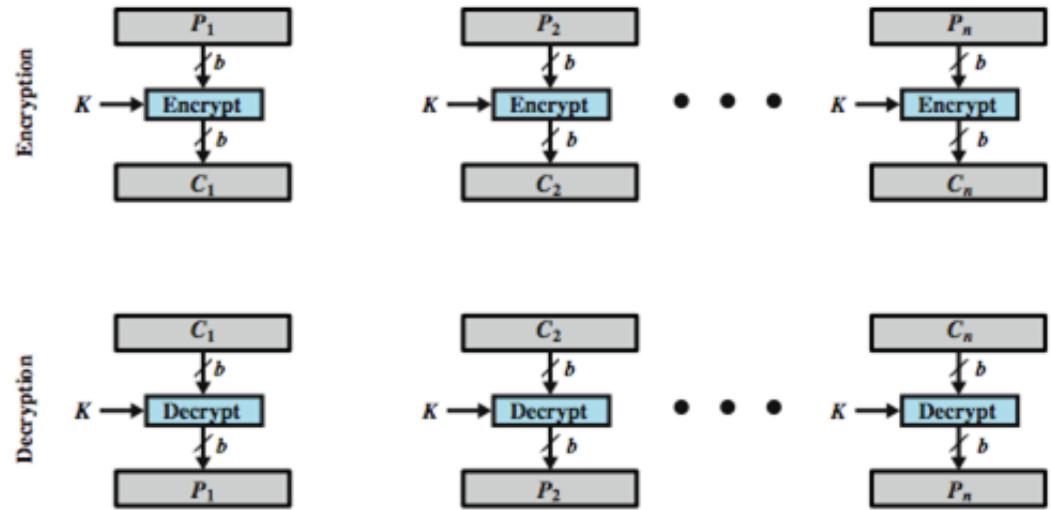
# **DES and Triple-DES**

- Data Encryption Standard (DES) is unfortunately one of the most widely used encryption schemes
  - uses 64 bit plaintext block and 56 bit key to produce a 64 bit ciphertext block
  - concerns about algorithm & use of 56-bit key
- Triple-DES
  - repeats basic DES algorithm three times
  - using either two or three unique keys
  - much more secure but also much slower

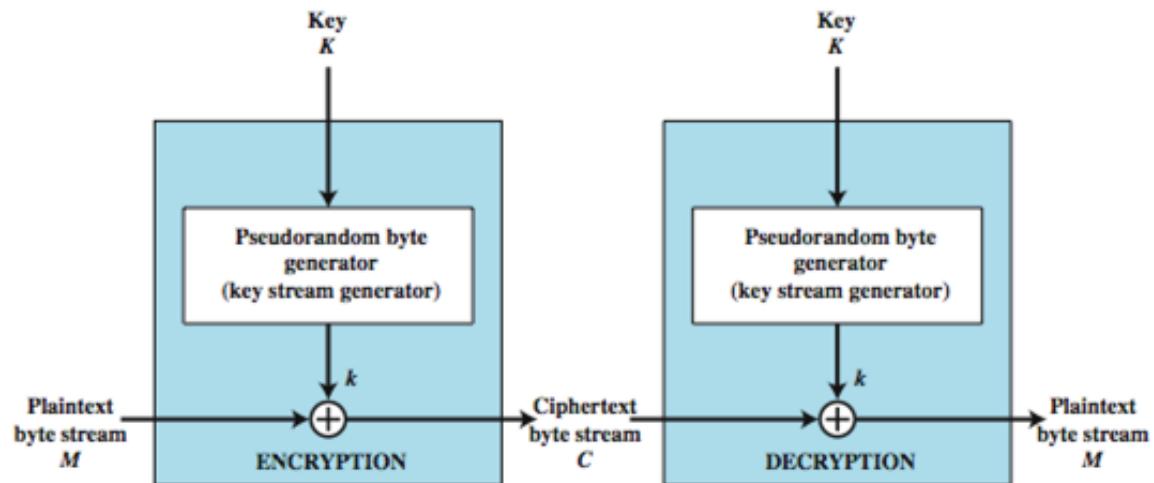
# **Advanced Encryption Standard (AES)**

- needed a better replacement for DES
- NIST called for proposals in 1997
- selected Rijndael in Nov 2001
- published as FIPS 197
- symmetric block cipher
- uses 128 bit data & 128/192/256 bit keys
- now widely available commercially

# Block vs. Stream Ciphers

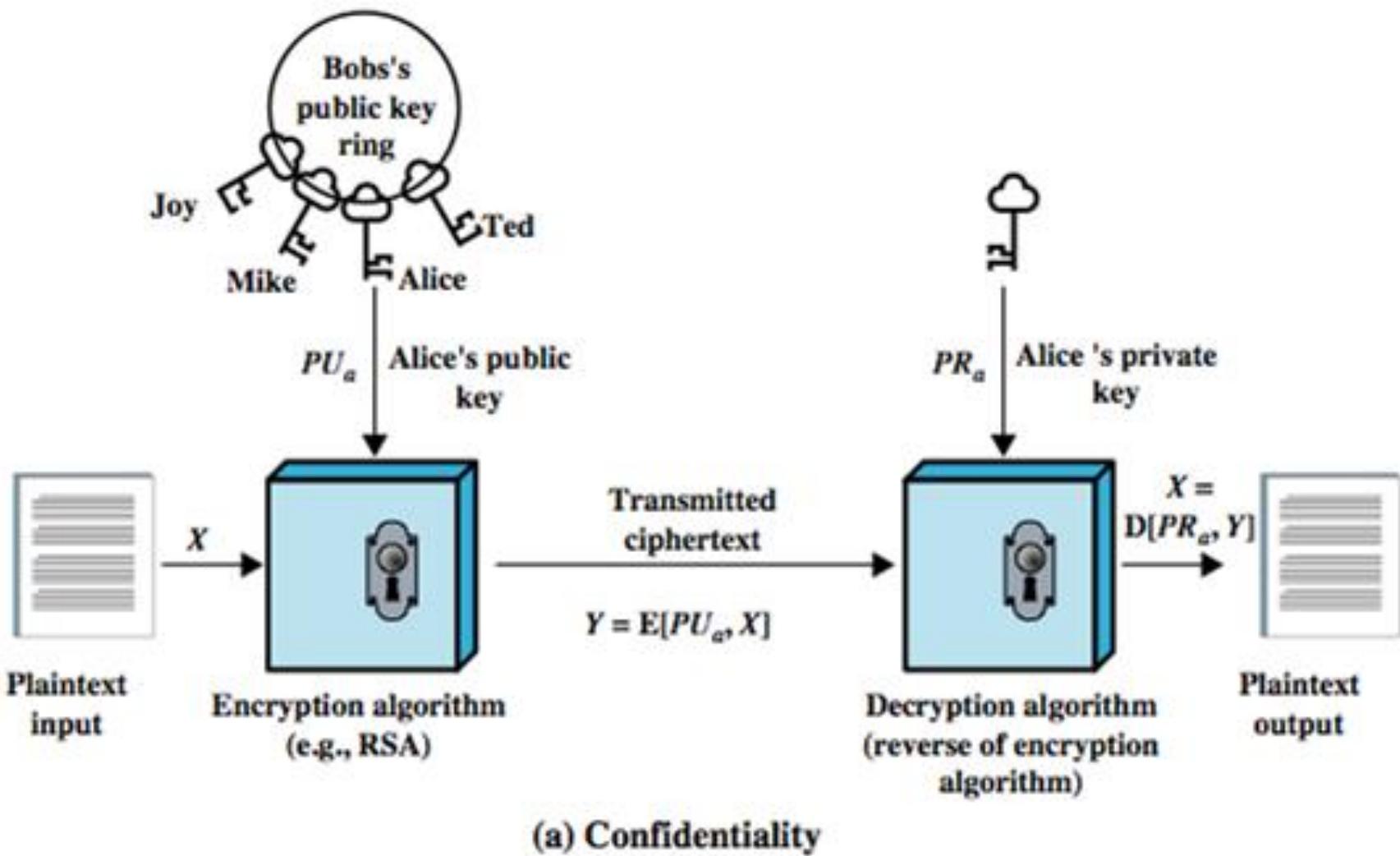


(a) Block cipher encryption (electronic codebook mode)



(b) Stream encryption

# Public Key Encryption



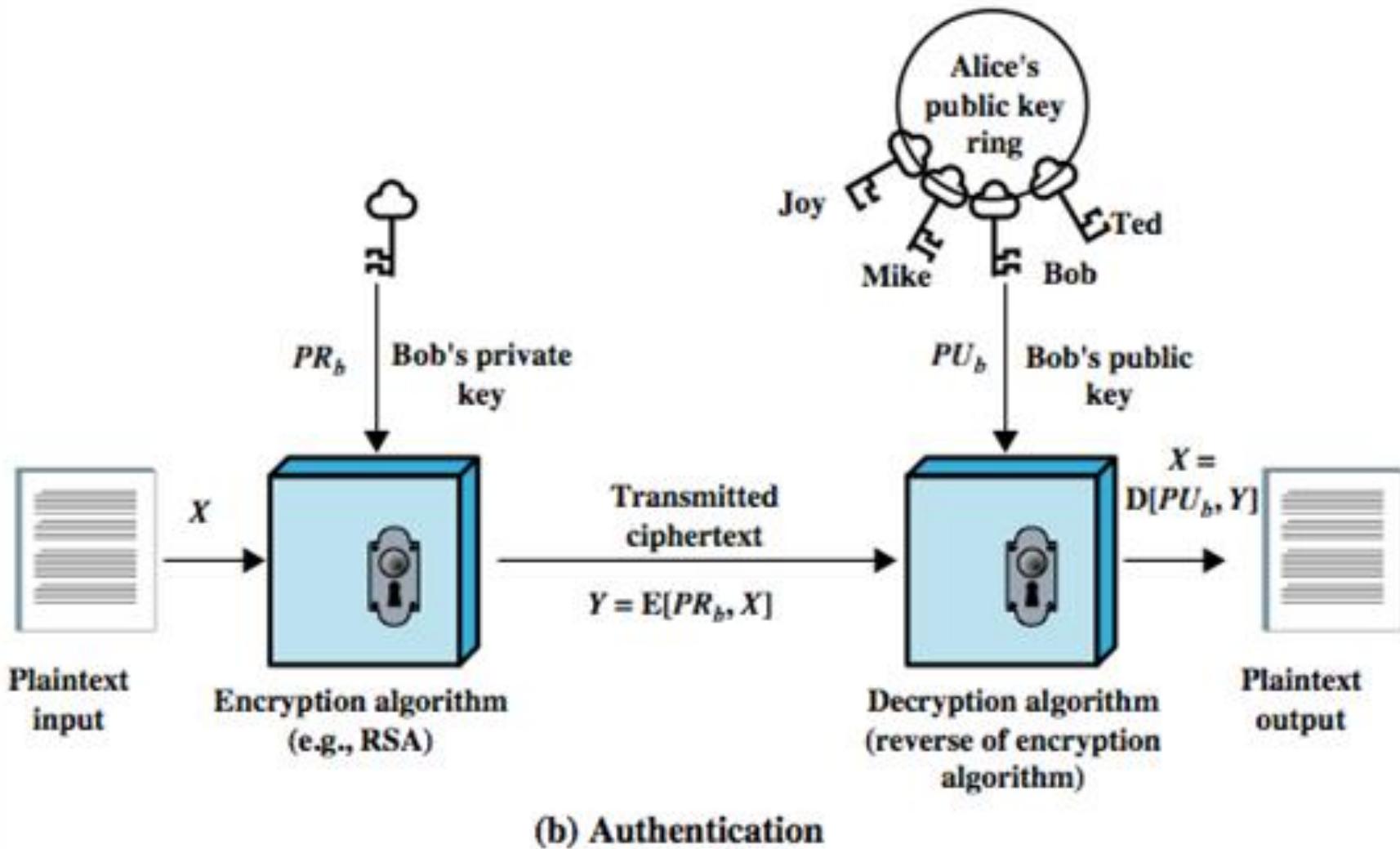
# Public Key Requirements

1. computationally easy to create key pairs
2. computationally easy for sender knowing public key to encrypt messages
3. computationally easy for receiver knowing private key to decrypt ciphertext
4. computationally infeasible for opponent to determine private key from public key
5. computationally infeasible for opponent to otherwise recover original message
6. useful if either key can be used for each role

# Public Key Algorithms

- RSA (Rivest, Shamir, Adleman)
  - developed in 1977
  - only widely accepted public-key encryption alg
  - given tech advances need 1024+ bit keys
- Diffie-Hellman key exchange algorithm
  - only allows exchange of a secret key
- Digital Signature Standard (DSS)
  - provides only a digital signature function with SHA-1
- Elliptic curve cryptography (ECC)
  - new, security like RSA, but with much smaller keys
- Homomorphic cryptography (HE)
  - Allows elaboration of encrypted data

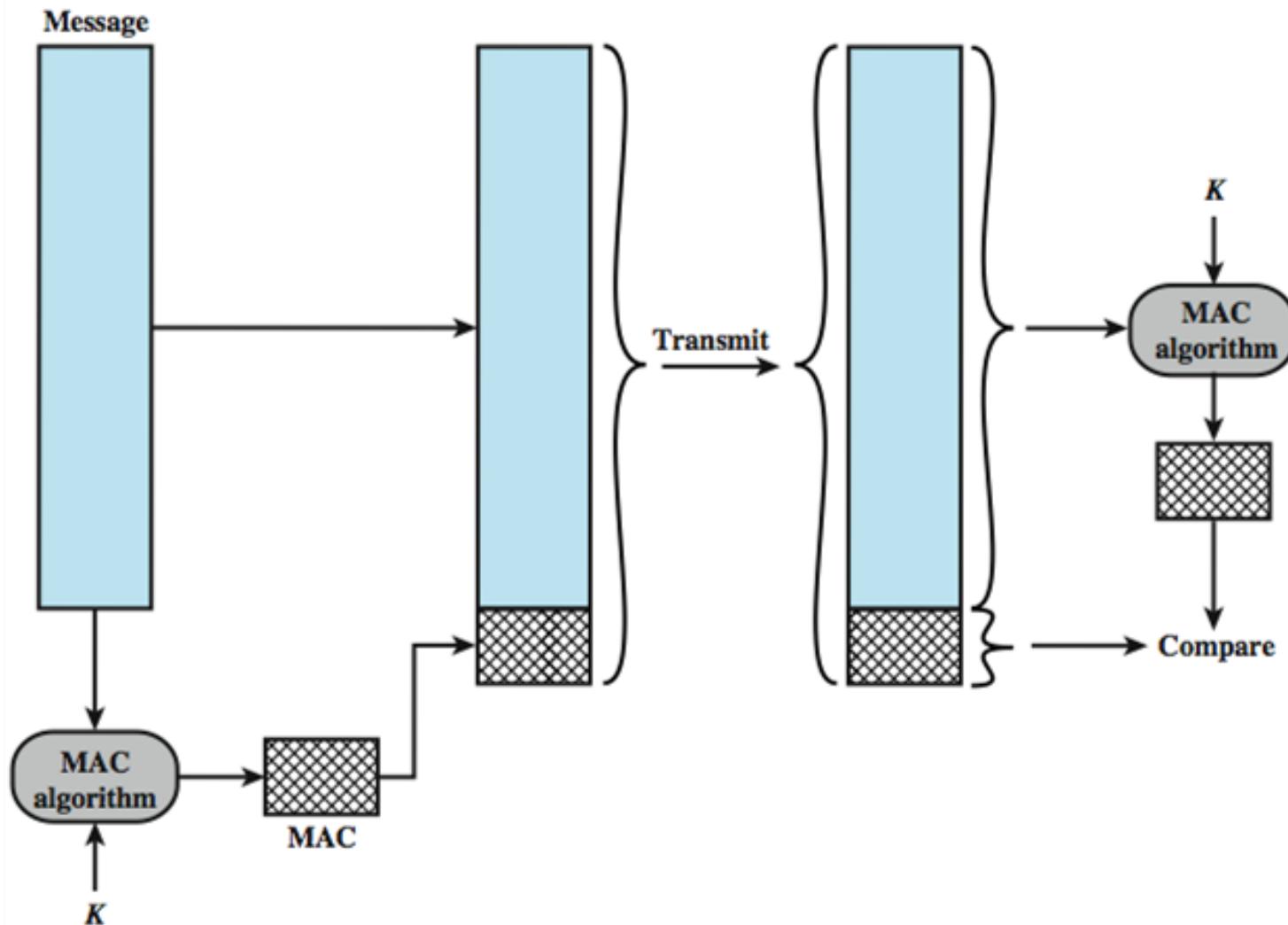
# Public Key Authentication



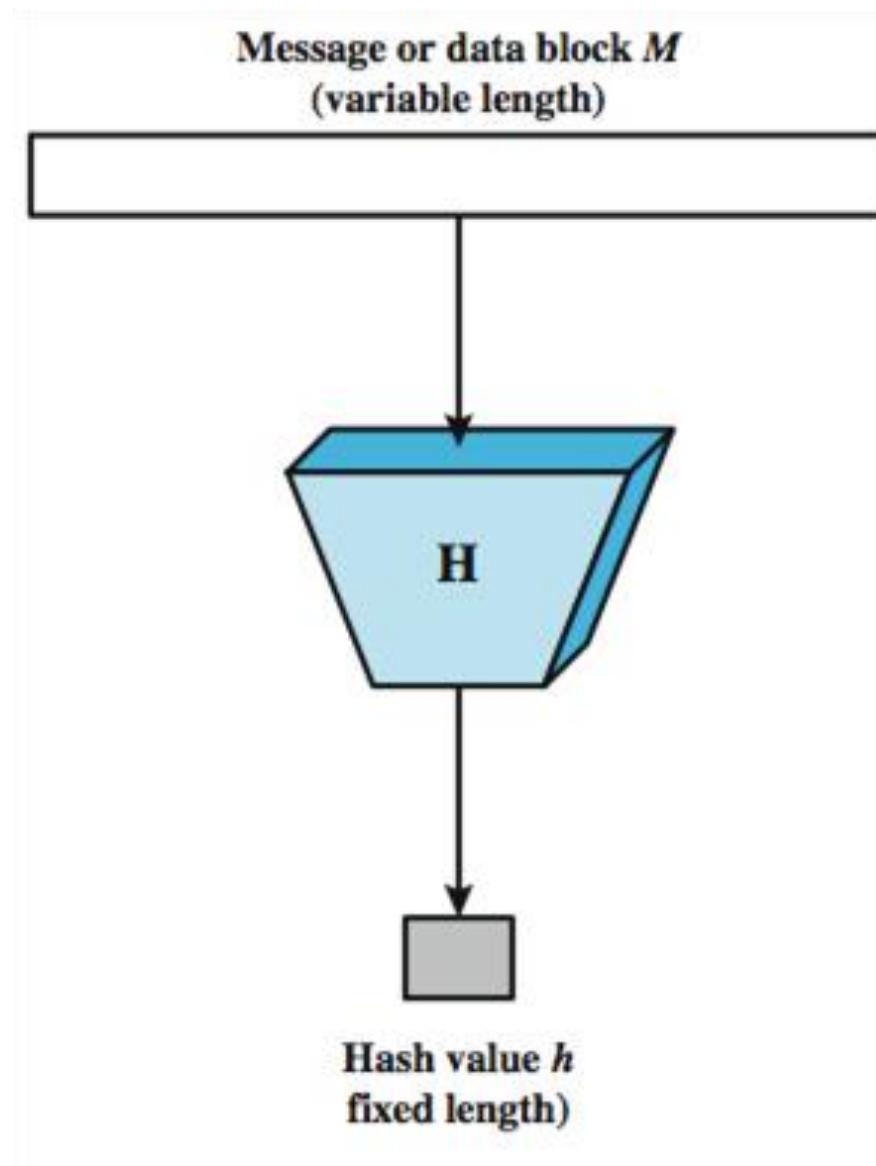
# Message Authentication

- protects against active attacks
- verifies received message is authentic
  - contents unaltered
  - from authentic source
  - timely and in correct sequence
- can use conventional encryption
  - only sender & receiver have key needed
- or separate authentication mechanisms
  - append authentication tag to cleartext message

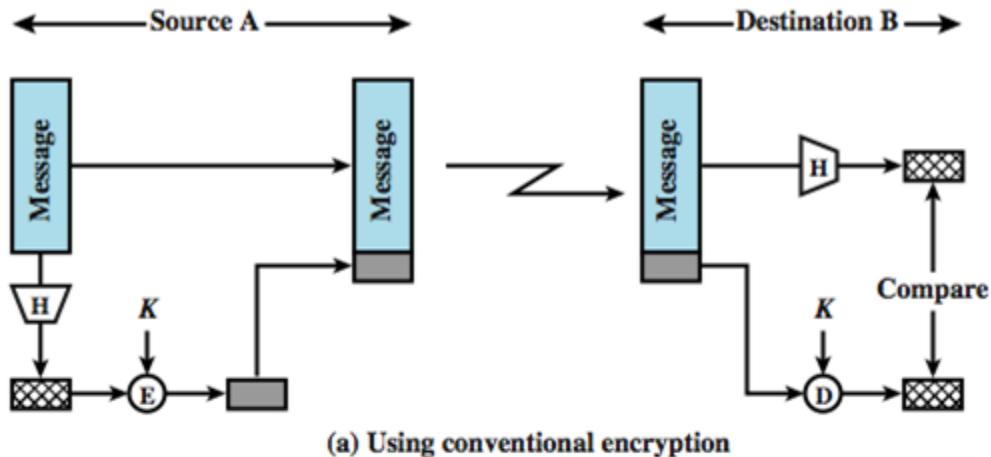
# Message Authentication Codes



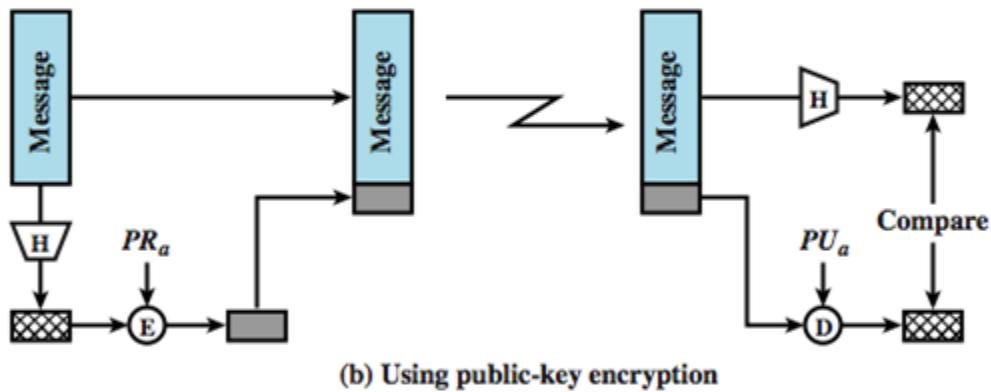
# Secure Hash Functions



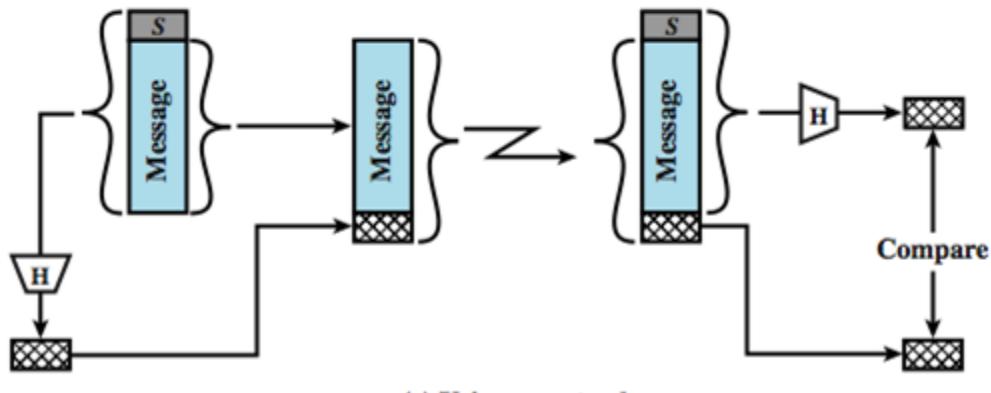
# Message Auth



(a) Using conventional encryption



(b) Using public-key encryption



(c) Using secret value

# Hash Function Requirements

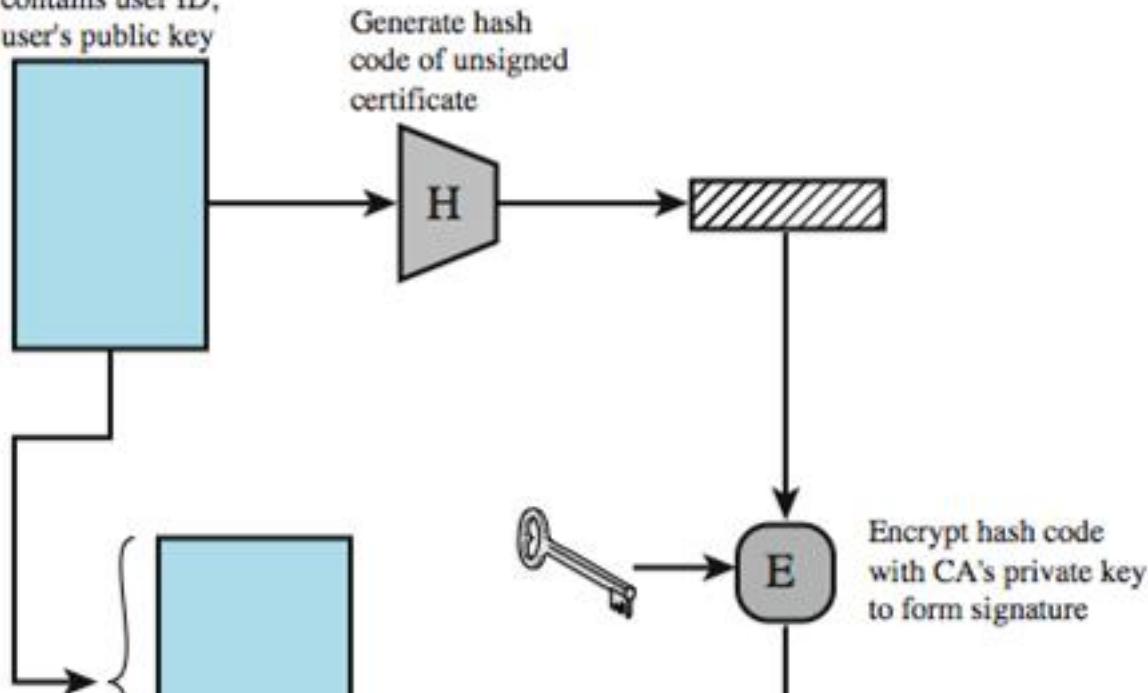
- applied to any size data
- $H$  produces a fixed-length output.
- $H(x)$  is relatively easy to compute for any given  $x$
- one-way property
  - computationally infeasible to find  $x$  such that  $H(x) = h$
- weak collision resistance
  - (given  $x$ ) computationally infeasible to find  $y \neq x$  such that  $H(y) = H(x)$
- strong collision resistance
  - computationally infeasible to find any pair  $(x, y)$  such that  $H(x) = H(y)$

# Hash Functions

- two attack approaches
  - cryptanalysis
    - exploit logical weakness in alg
  - brute-force attack
    - trial many inputs
    - strength proportional to size of hash code
- SHA most widely used hash algorithm
  - SHA-1 gives 160-bit hash
  - more recent SHA-256, SHA-384, SHA-512 provide improved size and security

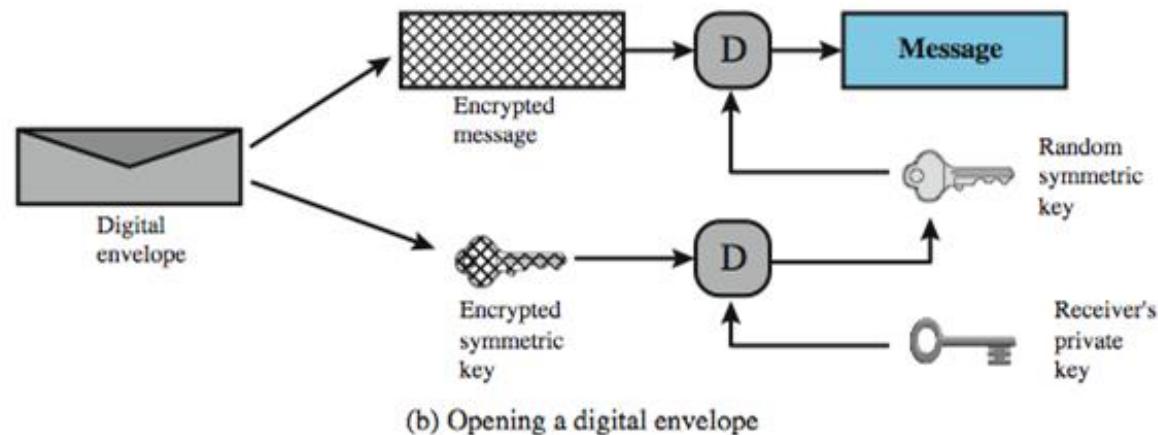
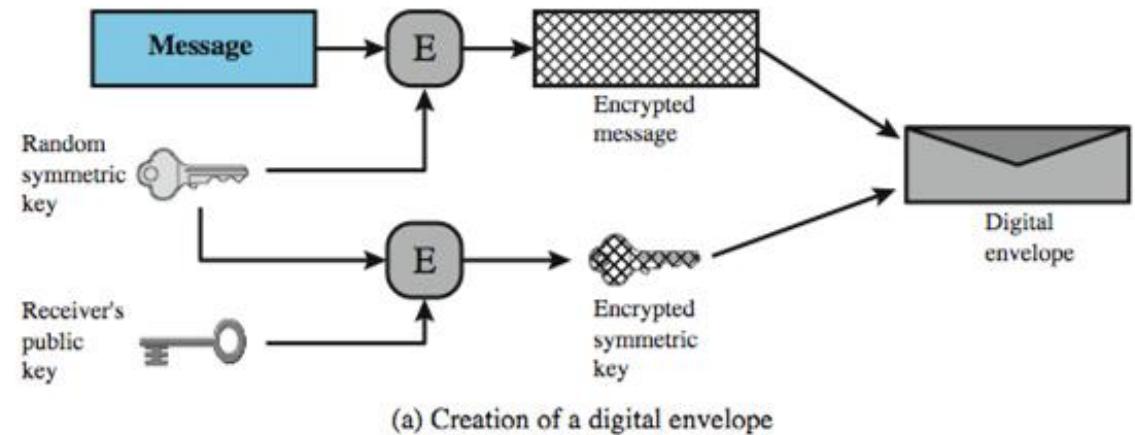
# Public Key Certificates

Unsigned certificate:  
contains user ID,  
user's public key



Signed certificate:  
Recipient can verify  
signature using CA's  
public key.

# Digital Envelopes



# Practical Application: Encryption of Stored Data

- common to encrypt transmitted data
- much less common for stored data
  - which can be copied, backed up, recovered
- approaches to encrypt stored data:
  - back-end appliance
  - library based tape encryption
  - background laptop/PC data encryption

# Authentication

# Password-Based Authentication

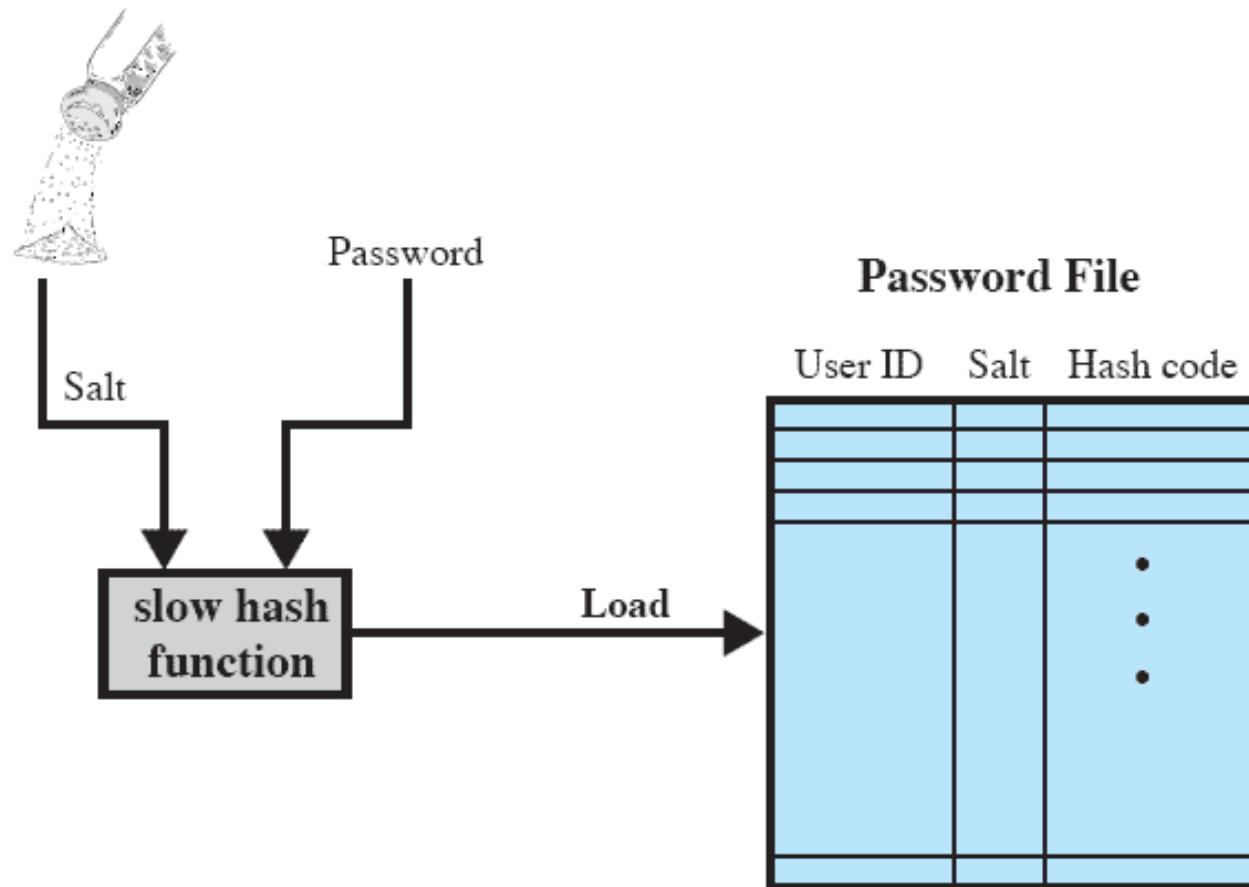
- A widely used line of defense against intruders is the password system
- The password serves to authenticate the ID of the individual logging on to the system
- The ID provides security by:

determining whether the user is authorized to gain access to a system

determining the privileges accorded to the user

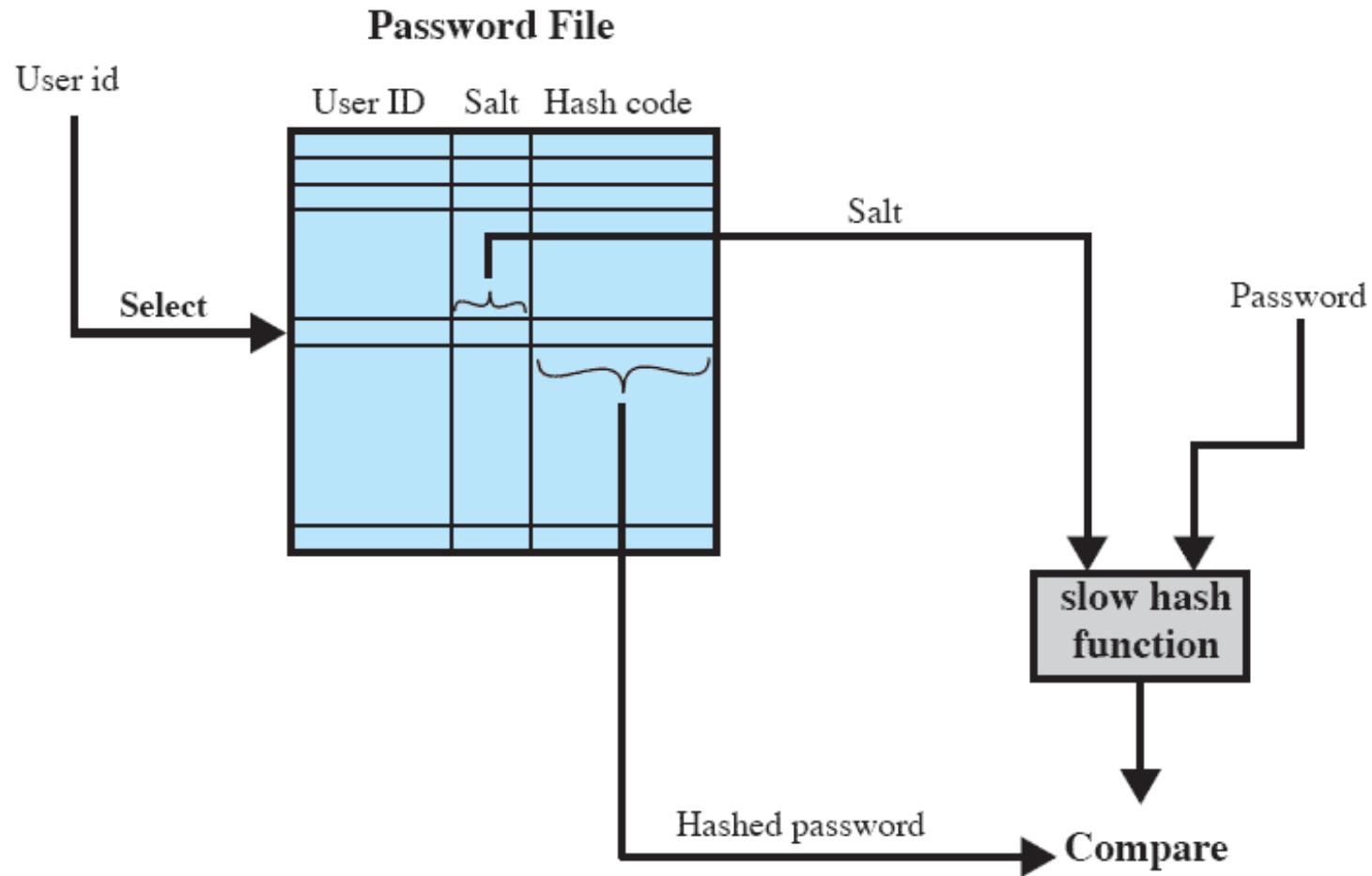
discretionary access control

# Hashed Passwords/Salt Value



**(a) Loading a new password**

# UNIX Password Scheme



**(b) Verifying a password**

# Salt



- Serves three purposes:
  1. prevents duplicate passwords from being visible in the password file
    - even if two users choose the same password, the passwords will be assigned different salt values
  2. greatly increases the difficulty of offline dictionary attacks
  3. it becomes nearly impossible to find out whether a person with passwords on two or more systems has used the same password on all of them

# UNIX Password Scheme

- There are two threats to the UNIX password scheme:
  - a user can gain access on a machine using a guest account
  - ***Password cracker*** – password guessing program
- If an opponent is able to obtain a copy of the password file, a cracker program can be run on another machine at leisure
  - this enables the opponent to run through millions of possible passwords in a reasonable period



# UNIX Implementations

- Most implementations have relied on a password scheme where each user selects a password of up to eight printable characters in length which is converted into a 56-bit value that serves as the key input to an encryption routine
- The hash routine, known as crypt(3) is based on DES
- The crypt(3) routine is designed to discourage guessing attacks
- Software implementations of DES are slow compared to hardware versions
- The recommended hash function for many UNIX systems, including Linux, Solaris, and FreeBSD is based on the MD5 secure hash algorithm
- The most secure version of the UNIX hash/salt scheme was developed for OpenBSD and uses a hash function (Bcrypt) based on the Blowfish symmetric block cipher

# Token-Based Authentication

- Objects that a user possesses for the purpose of user authentication are called tokens

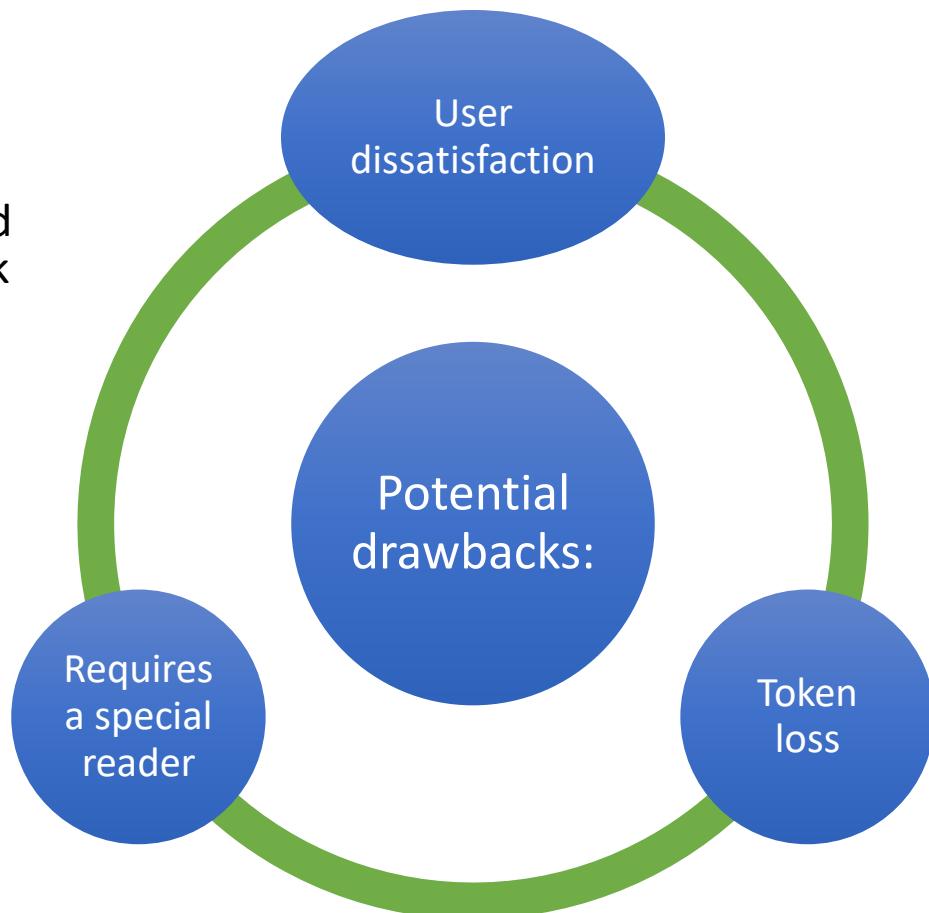
Two types of tokens are:

memory cards

smart cards

# Memory Cards

- Memory cards can store but not process data
  - the most common is the bank card with a magnetic stripe on the back
  - a magnetic stripe can store only a simple security code which can be read and reprogrammed by an inexpensive card reader
  - there are also memory cards that include an internal electronic memory
- Can be used alone for physical access or with some form of password or personal identification number (PIN)



# Smart Cards

## Physical characteristics:

- include an embedded processor
- a smart token that looks like a bank card is called a smart card
- smart tokens can look like calculators, keys, or other small portable objects

## Interface

- manual interfaces include a keypad and display for human/token interaction
- smart tokens with an electronic interface communicate with a compatible reader/writer

## Authentication protocol

- static
- dynamic password generator
- challenge-response

# Static Biometric Authentication

- Attempts to authenticate an individual based on his or her unique physical characteristics
- Includes static characteristics such as:
  - fingerprints
  - hand geometry
  - facial characteristics
  - retinal and iris patterns
- Dynamic characteristics such as:
  - voiceprint
  - signature



# Physical Characteristics

## Facial characteristics

- most common means of human-to-human identification
- examples: eyes, eyebrows, nose, lips, and chin shape
- alternative approach is to use an infrared camera to produce a face thermogram

## Fingerprints

- pattern of ridges and furrows on the surface of the fingertip
- unique across the entire human population

## Hand geometry

- identify features of the hand, including shape, and lengths and widths of fingers

## Retinal pattern

- the pattern formed by veins beneath the retinal surface is unique
- a retinal biometric system obtains a digital image of the retinal pattern by projecting a low-intensity beam of visual or infrared light into the eye

## Iris

- the detailed structure of the iris is unique

## Signature

- each individual has a unique style of handwriting

## Voice

- voice patterns are closely tied to the physical and anatomical characteristics of the speaker

# Cost versus Accuracy

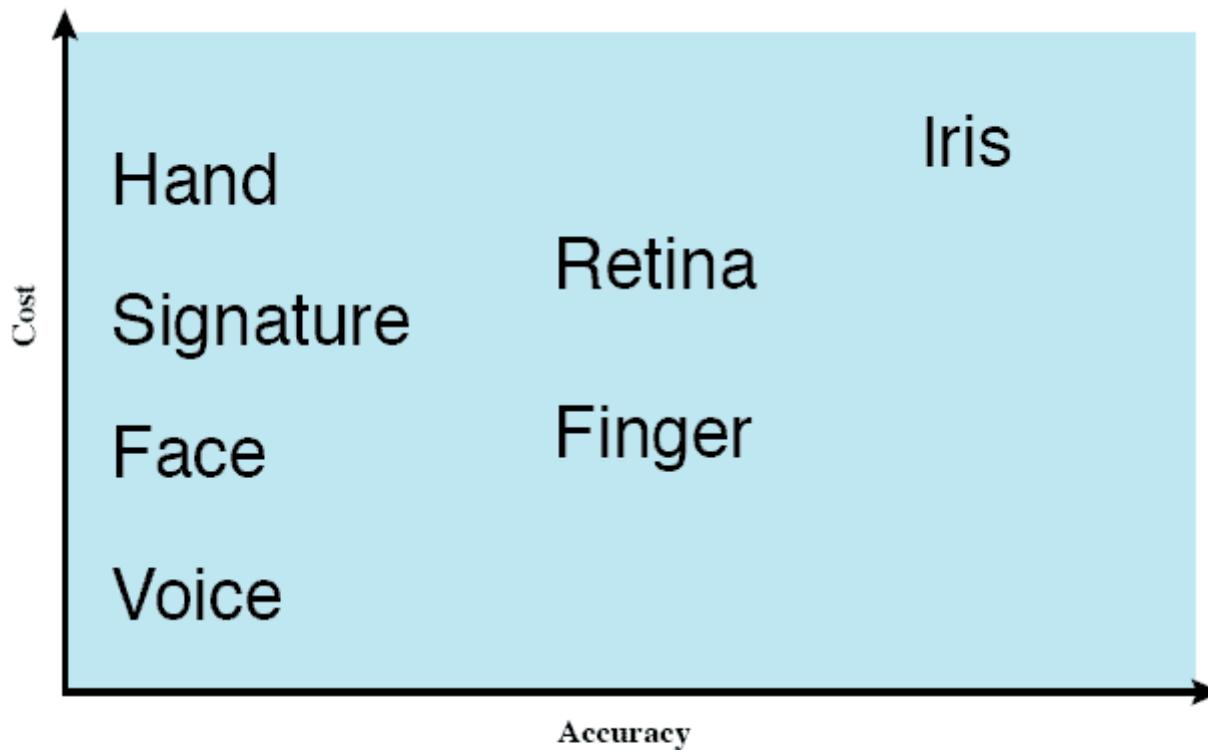


Figure 15.2 Cost Versus Accuracy of Various Biometric Characteristics in User Authentication Schemes.

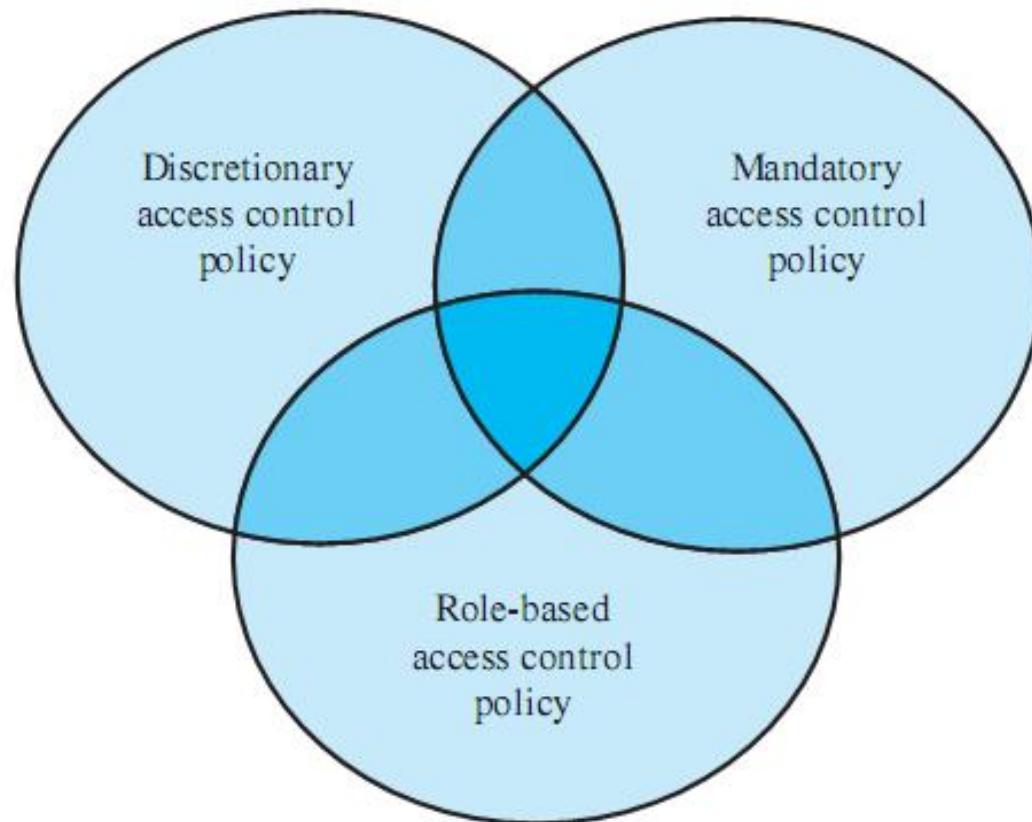
# Access control

# Access Control



- Dictates what types of access are permitted, under what circumstances, and by whom
- Access control policies are generally grouped into the following categories:
  - Discretionary access control (DAC)
    - controls access based on the identity of the requestor and on access rules stating what requestors are (or are not) allowed to do
  - Mandatory access control (MAC)
    - controls access based on comparing security labels with security clearances
  - Role-based access control (RBAC)
    - controls access based on the roles that users have within the system and on rules stating what accesses are allowed to users in given roles

# Access Control Policies



**Figure 15.3** Access Control Policies

# Extended Access Control Matrix

|          |                | OBJECTS        |                |                |                |                |                |                |                |                |
|----------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
|          |                | subjects       |                | files          |                | processes      |                | disk drives    |                |                |
|          |                | S <sub>1</sub> | S <sub>2</sub> | S <sub>3</sub> | F <sub>1</sub> | F <sub>2</sub> | P <sub>1</sub> | P <sub>2</sub> | D <sub>1</sub> | D <sub>2</sub> |
| SUBJECTS | S <sub>1</sub> | control        | owner          | owner control  | read *         | read owner     | wakeup         | wakeup         | seek           | owner          |
|          | S <sub>2</sub> |                | control        |                | write *        | execute        |                |                | owner          | seek *         |
|          | S <sub>3</sub> |                |                | control        |                | write          | stop           |                |                |                |

\* - copy flag set

Figure 15.4 Extended Access Control Matrix

# Organization of the Access Control Function

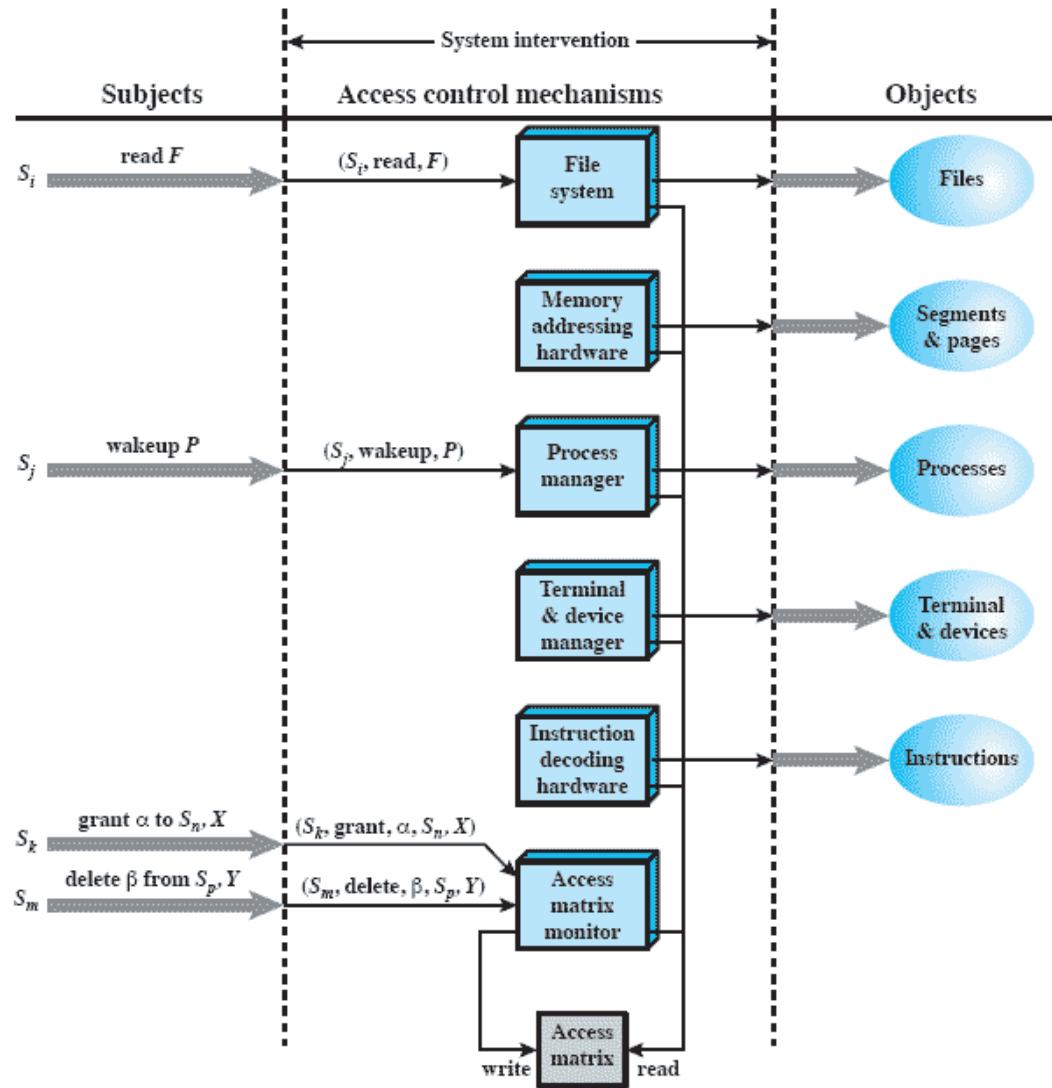


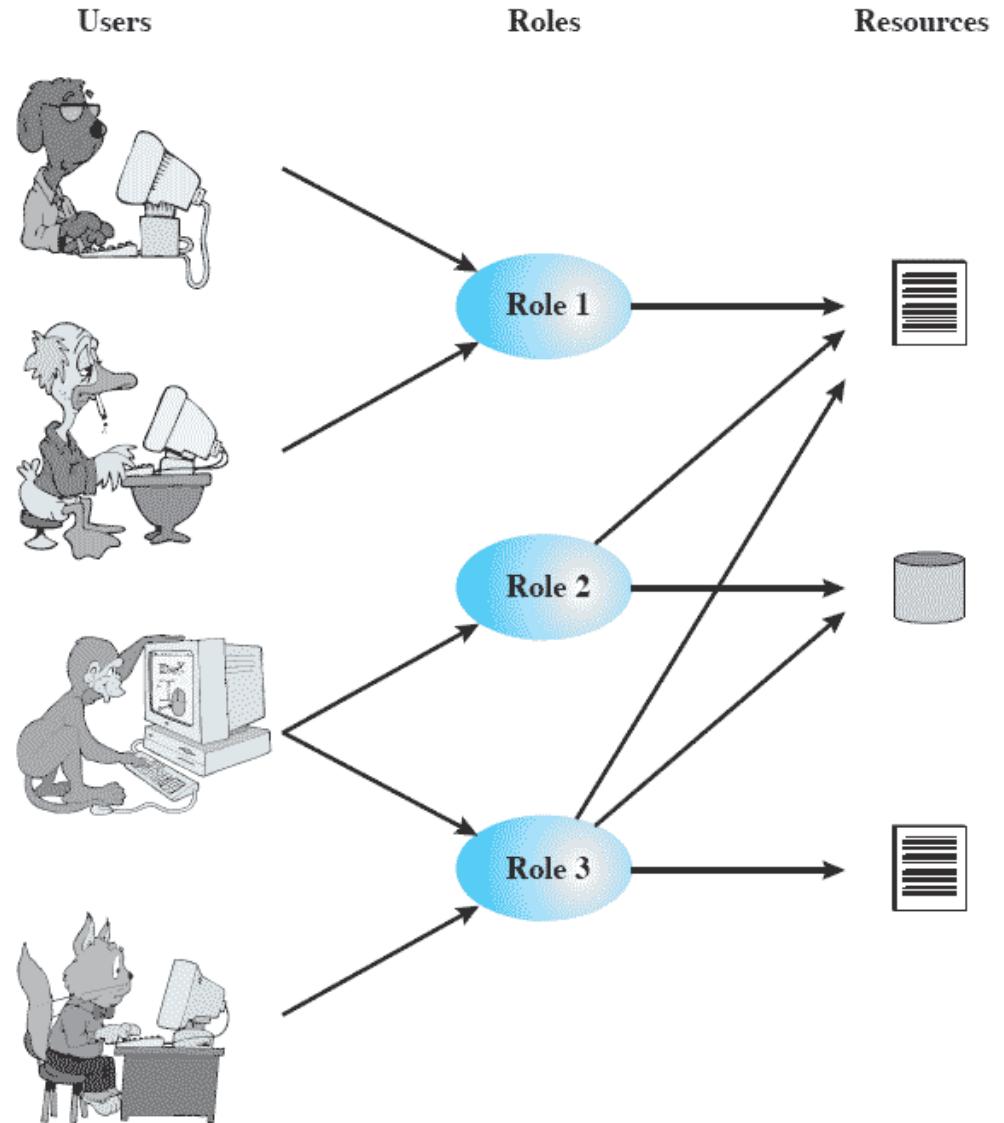
Figure 15.5 An Organization of the Access Control Function

# Role-Based Access Control

- Based on the roles that users assume in a system rather than the user's identity
- Models define a role as a job function within an organization
- Systems assign access rights to roles instead of individual users
  - in turn, users are assigned to different roles, either statically or dynamically, according to their responsibilities
- NIST has issued a standard that requires support for access control and administration through roles



Users  
Roles  
Resources



|                | R <sub>1</sub> | R <sub>2</sub> | • • • | R <sub>n</sub> |
|----------------|----------------|----------------|-------|----------------|
| U <sub>1</sub> | X              |                |       |                |
| U <sub>2</sub> | X              |                |       |                |
| U <sub>3</sub> |                | X              |       | X              |
| U <sub>4</sub> |                |                |       | X              |
| U <sub>5</sub> |                |                |       | X              |
| U <sub>6</sub> |                |                |       | X              |
| •              |                |                |       |                |
| U <sub>m</sub> | X              |                |       |                |

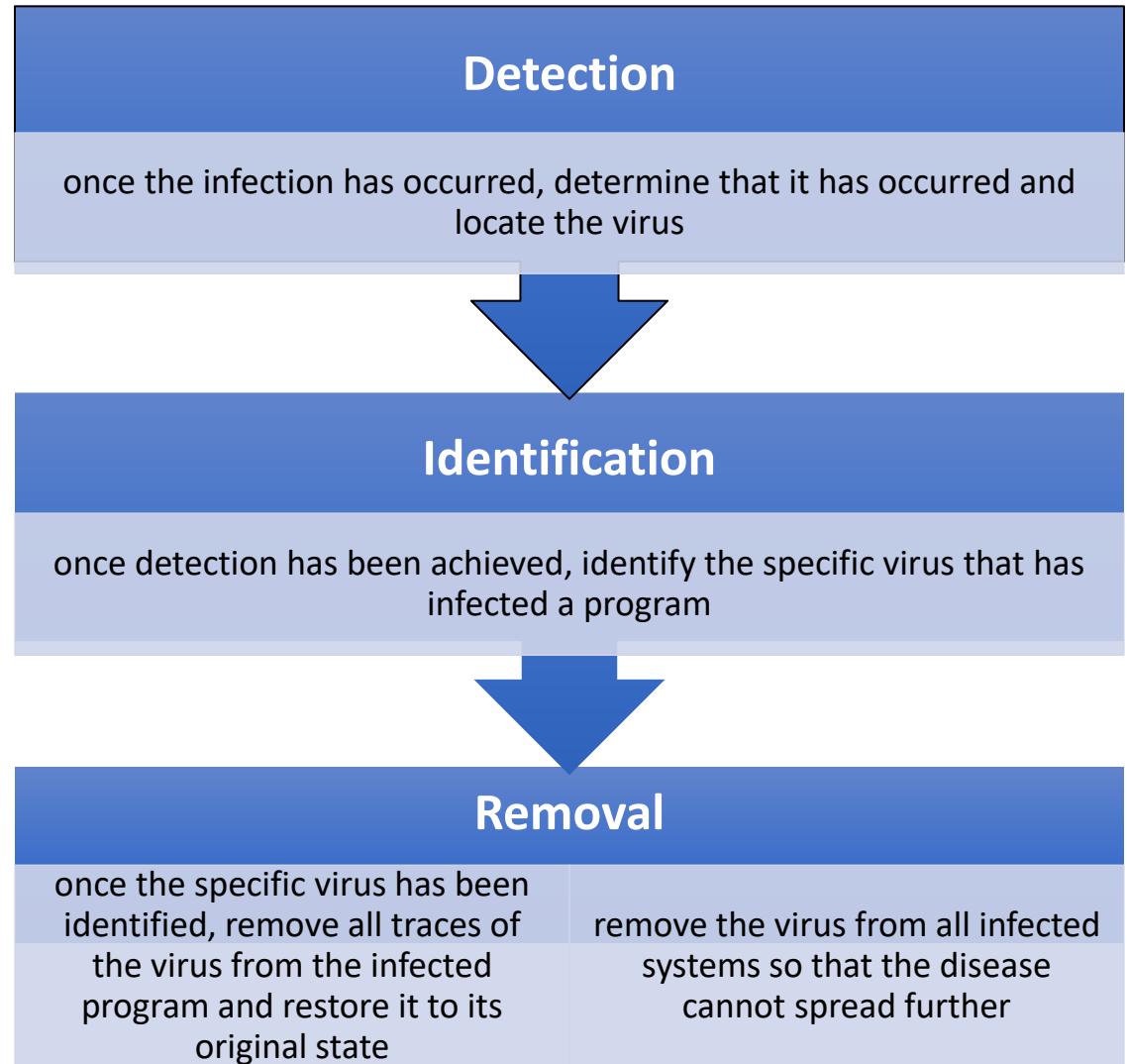
# Access Control Matrix Representation of RBAC

|       |                | OBJECTS        |                |                |                |                |                |                |                |                |
|-------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
|       |                | R <sub>1</sub> | R <sub>2</sub> | R <sub>n</sub> | F <sub>1</sub> | F <sub>1</sub> | P <sub>1</sub> | P <sub>2</sub> | D <sub>1</sub> | D <sub>2</sub> |
| ROLES | R <sub>1</sub> | control        | owner          | owner control  | read *         | read owner     | wakeup         | wakeup         | seek           | owner          |
|       | R <sub>2</sub> |                | control        |                | write *        | execute        |                |                | owner          | seek *         |
|       | •              |                |                |                |                |                |                |                |                |                |
|       | •              |                |                |                |                |                |                |                |                |                |
|       | R <sub>n</sub> |                |                | control        |                | write          | stop           |                |                |                |

# Antivirus

# Antivirus Approaches

- Ideal solution to the threat of viruses is prevention, don't allow a virus onto the system in the first place!
- That goal is, in general, impossible to achieve, although prevention can reduce the number of successful viral attacks
- If detection succeeds but either identification or removal is not possible, then the alternative is to discard the infected program and reload a clean backup version



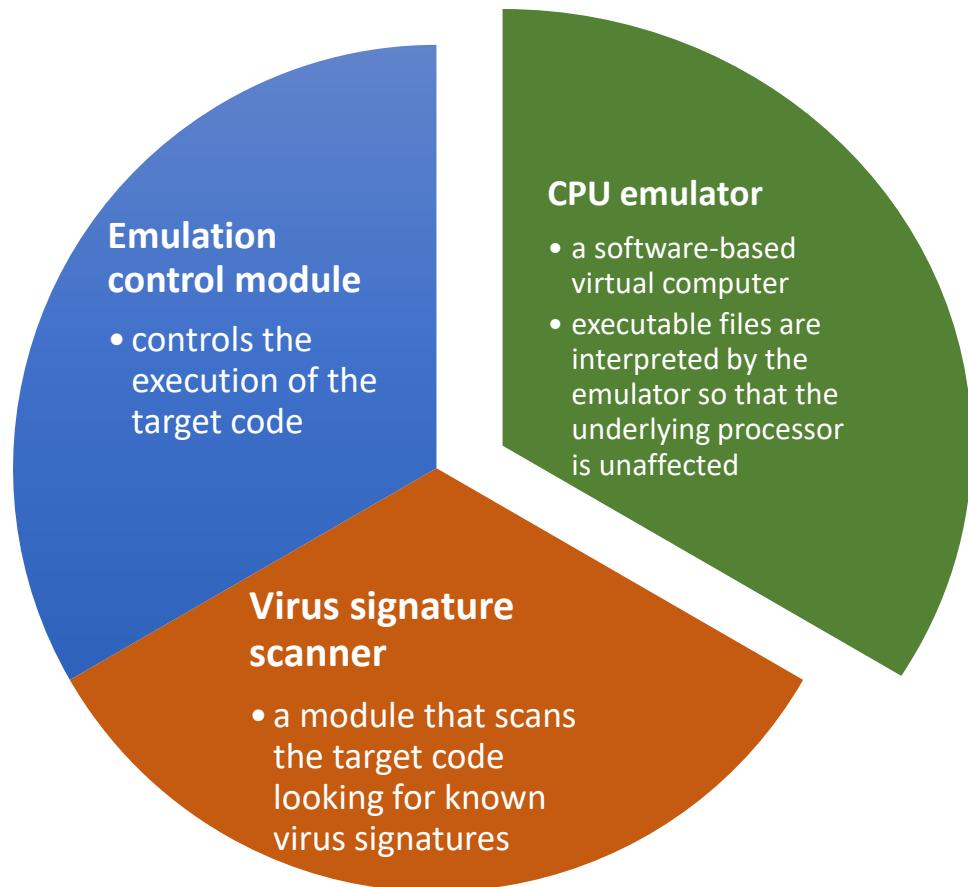
# Generic Decryption (GD)

- Enables the antivirus program to easily detect even the most complex polymorphic viruses while maintaining fast scanning speeds
- When a file containing a polymorphic virus is executed, the virus must decrypt itself to activate
- Executable files are run through a GD scanner



# GD Scanner

- GD scanner contains:
- Most difficult design issue with a GD scanner is to determine how long to run each interpretation



# Digital Immune System



- A comprehensive approach to virus protection developed by IBM and refined by Symantec
- Motivation for development has been the rising threat of Internet-based virus propagation
- Two major trends in Internet technology have had an increasing impact on the rate of virus propagation in recent years:
  - integrated mail systems
  - mobile-program systems
- Objective of the system is to provide rapid response time so that viruses can be stamped out almost as soon as they are introduced

# Digital Immune System

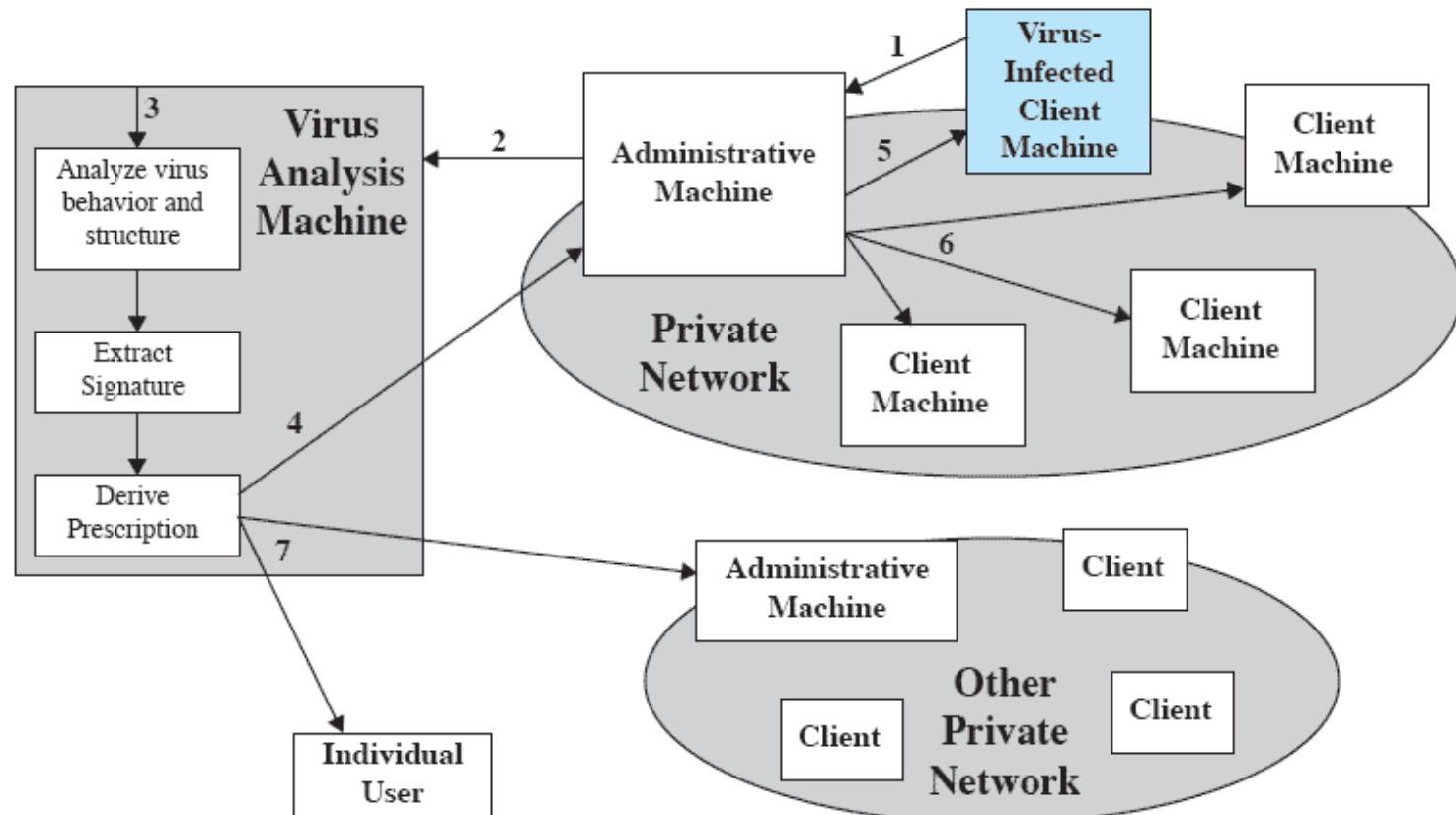


Figure 15.9 Digital Immune System

# Firewalls

# Firewalls and Intrusion Prevention Systems

- effective means of protecting LANs
- internet connectivity essential
  - for organization and individuals
  - but creates a threat
- could secure workstations and servers
- also use firewall as perimeter defence
  - single choke point to impose security

# Firewall Capabilities & Limits

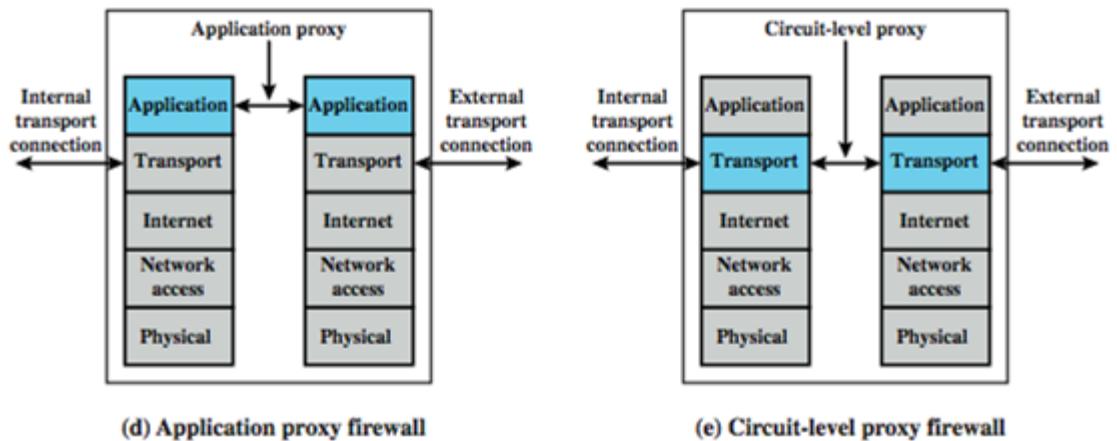
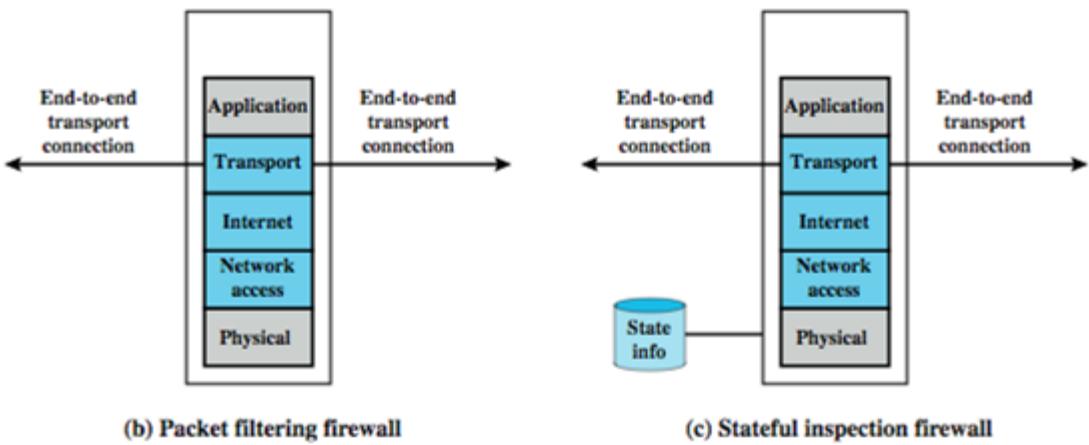
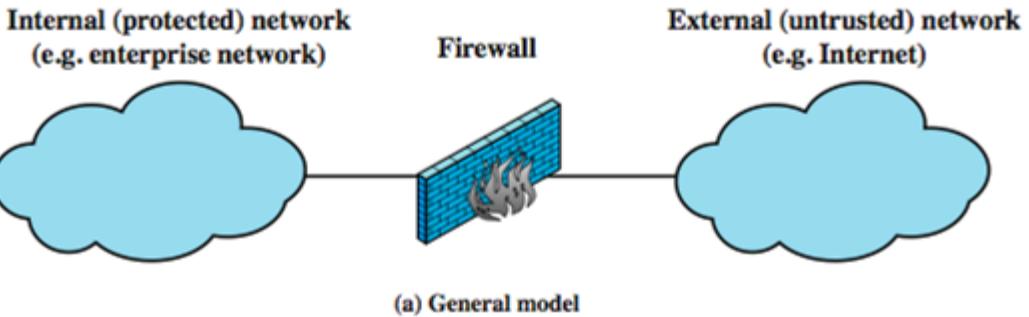
## ➤ capabilities:

- defines a single choke point
- provides a location for monitoring security events
- convenient platform for some Internet functions such as NAT, usage monitoring, IPSEC VPNs

## ➤ limitations:

- cannot protect against attacks bypassing firewall
- may not protect fully against internal threats
- improperly secure wireless LAN
- laptop, PDA, portable storage device infected outside then used inside

# Types of Firewalls



# Packet Filtering Firewall

- applies rules to packets in/out of firewall
- based on information in packet header
  - src/dest IP addr & port, IP protocol, interface
- typically a list of rules of matches on fields
  - if match rule says if forward or discard packet
- two default policies:
  - discard - prohibit unless expressly permitted
    - more conservative, controlled, visible to users
  - forward - permit unless expressly prohibited
    - easier to manage/use but less secure

# Packet Filter Rules

Rule Set A

| action | ourhost | port | theirhost | port | comment                     |
|--------|---------|------|-----------|------|-----------------------------|
| block  | *       | *    | SPIGOT    | *    | we don't trust these people |
| allow  | OUR-GW  | 25   | *         | *    | connection to our SMTP port |

Rule Set B

| action | ourhost | port | theirhost | port | comment |
|--------|---------|------|-----------|------|---------|
| block  | *       | *    | *         | *    | default |

Rule Set C

| action | ourhost | port | theirhost | port | comment                       |
|--------|---------|------|-----------|------|-------------------------------|
| allow  | *       | *    | *         | 25   | connection to their SMTP port |

Rule Set D

| action | src         | port | dest | port | flags | comment                        |
|--------|-------------|------|------|------|-------|--------------------------------|
| allow  | {our hosts} | *    | *    | 25   |       | our packets to their SMTP port |
| allow  | *           | 25   | *    | *    | ACK   | their replies                  |

Rule Set E

| action | src         | port | dest | port  | flags | comment               |
|--------|-------------|------|------|-------|-------|-----------------------|
| allow  | {our hosts} | *    | *    | *     |       | our outgoing calls    |
| allow  | *           | *    | *    | *     | ACK   | replies to our calls  |
| allow  | *           | *    | *    | >1024 |       | traffic to nonservers |

# Packet Filter Weaknesses

## ➤ weaknesses

- cannot prevent attack on application bugs
- limited logging functionality
- do not support advanced user authentication
- vulnerable to attacks on TCP/IP protocol bugs
- improper configuration can lead to breaches

## ➤ attacks

- IP address spoofing, source route attacks, tiny fragment attacks

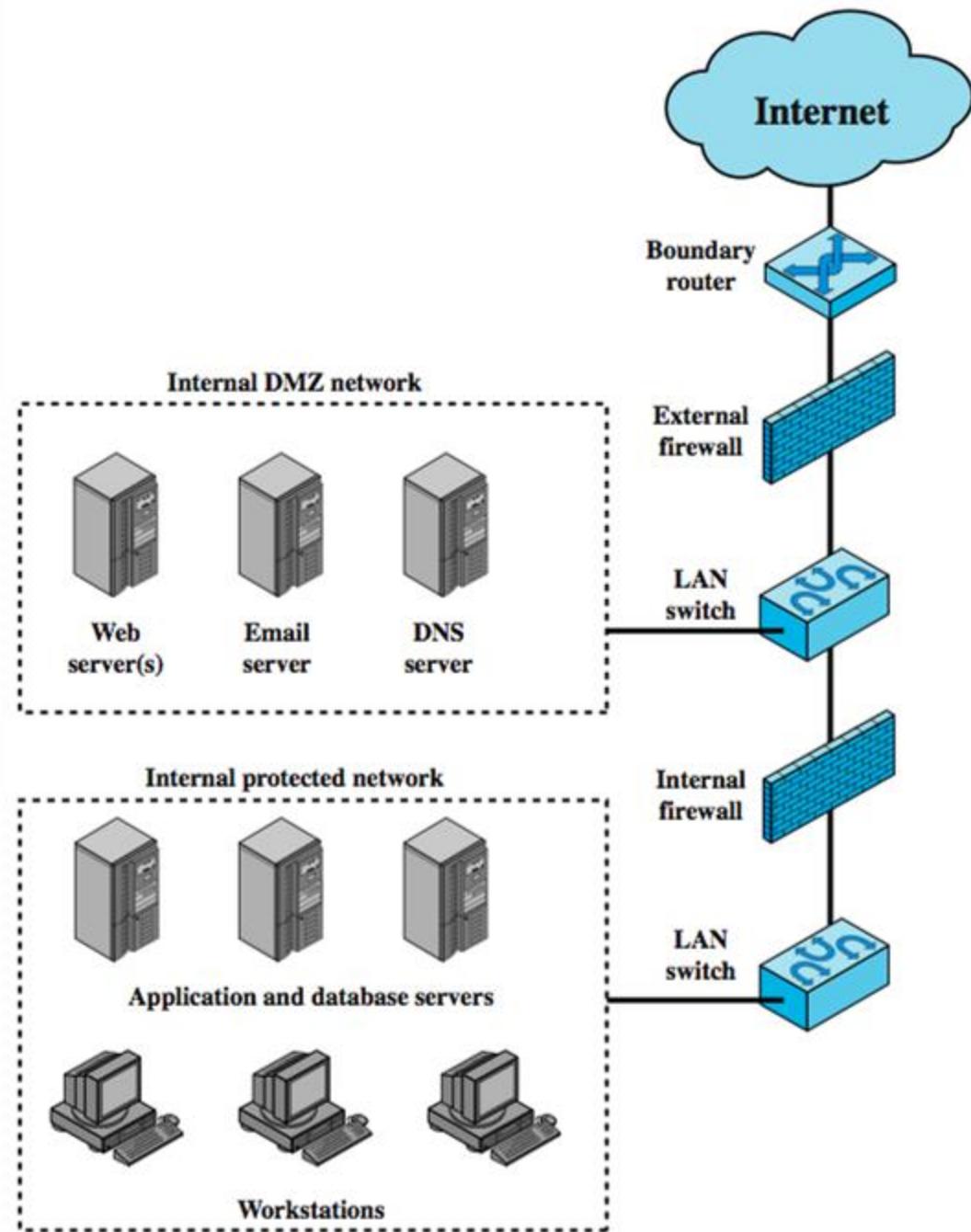
# Stateful Inspection Firewall

- reviews packet header information but also keeps info on TCP connections
  - typically have low, “known” port number for server
  - and high, dynamically assigned client port number
  - simple packet filter must allow all return high port numbered packets back in
  - stateful inspection packet firewall tightens rules for TCP traffic using a directory of TCP connections
  - only allow incoming traffic to high-numbered ports for packets matching an entry in this directory
  - may also track TCP seq numbers as well

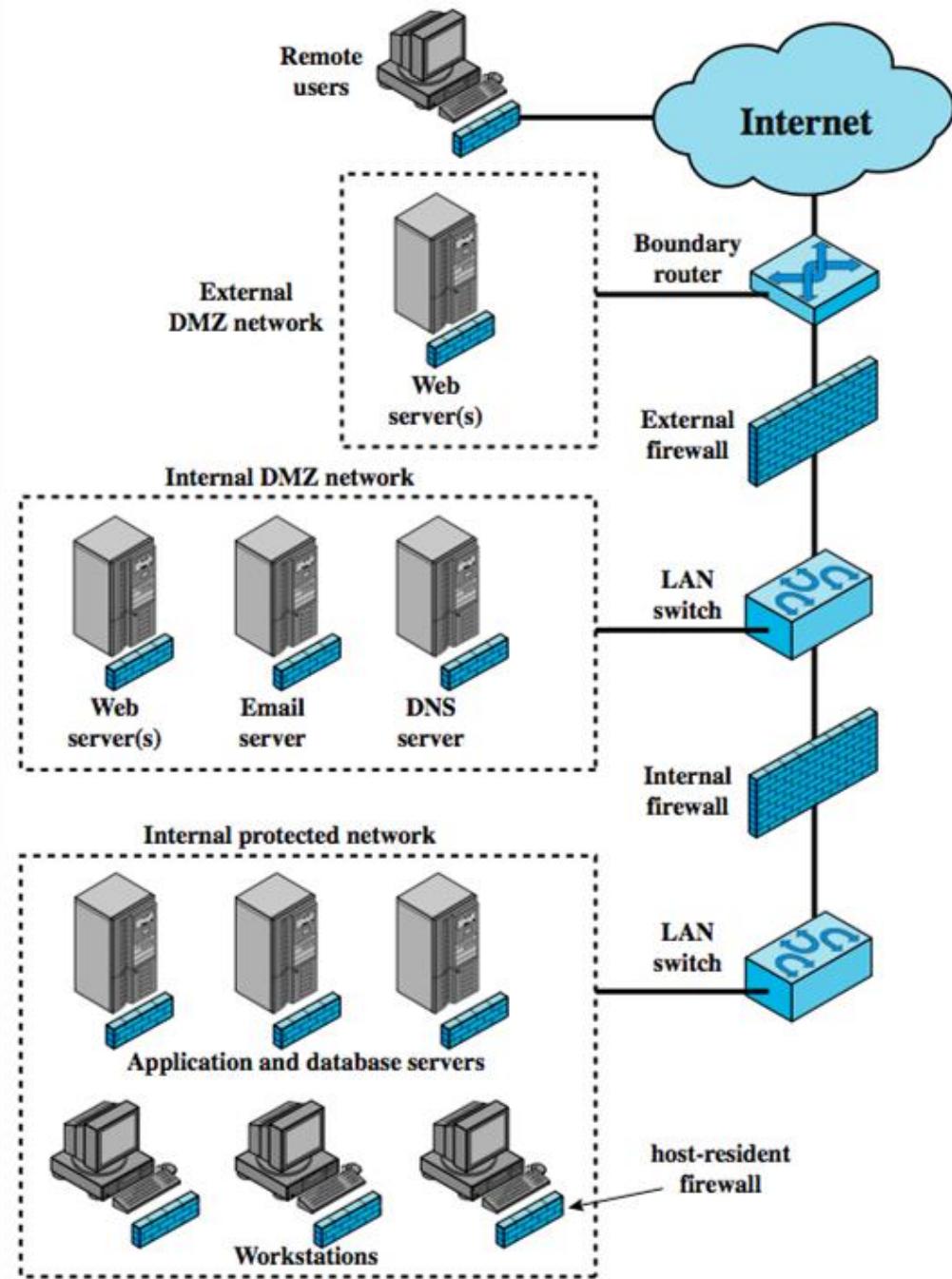
# Other solutions

- Application-level gateway (proxy)
- Circuit-level gateway
- Socket-level gateway

# Firewall Locations



# Distributed Firewalls



# Intrusion detection

# Intruders

- significant issue hostile/unwanted trespass
  - from benign to serious
- user trespass
  - unauthorized logon, privilege abuse
- software trespass
  - virus, worm, or trojan horse
- classes of intruders:
  - Hackers, masquerader, APTs

# Intrusion Detection

## - Basic Principles



- Intrusion detection is based on the assumption that the behavior of the intruder differs from that of a legitimate user in ways that can be quantified
- If an intrusion is detected quickly enough, the intruder can be identified and ejected from the system before any damage is done or any data are compromised
- An effective IDS can serve as a deterrent, thus acting to prevent intrusions
- Intrusion detection enables the collection of information about intrusion techniques that can be used to strengthen intrusion prevention measures

# Profiles of Behavior

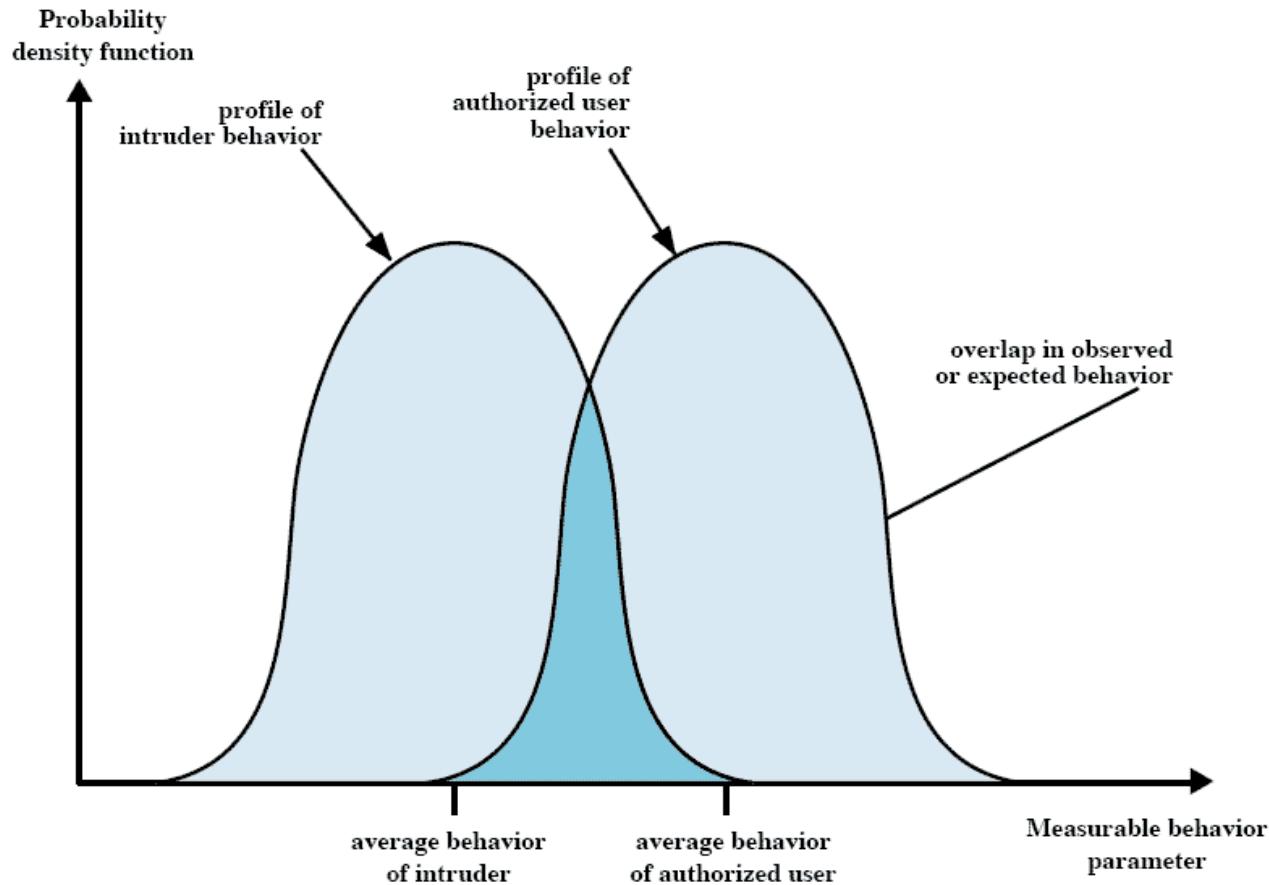


Figure 15.8 Profiles of Behavior of Intruders and Authorized Users

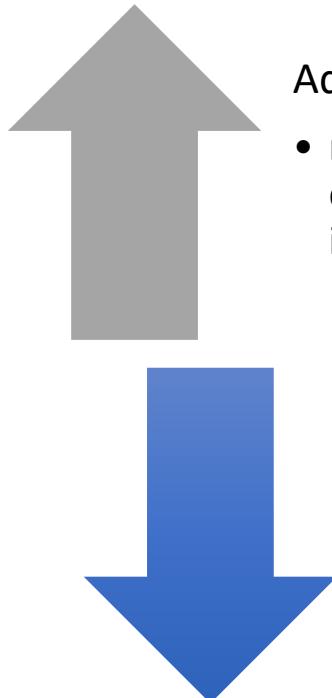
# Host-Based Intrusion Detection System (IDS)

- Monitors activity on the system in a variety of ways to detect suspicious behavior
- Primary purpose is to detect intrusions, log suspicious events, and send alerts
- Can detect both external and internal intrusions
  - Anomaly detection
    - collection of data relating to behavior of legitimate users over time
    - threshold detection
    - profile based detection
  - Signature detection
    - define a set of rules or attack patterns that can be used to decide that a given behavior is that of an intruder



# Audit Records

## Native



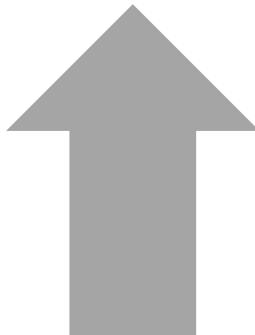
### Advantage

- no additional collection software is needed

### Disadvantage

- the native audit records may not contain the needed information or may not contain it in a convenient form

## Detection-specific



### Advantage

- it could be made vendor independent and ported to a variety of systems

### Disadvantage

- the extra overhead involved in having, in effect, two accounting packages running on a machine

# Anomaly Detection

## ➤ threshold detection

- checks excessive event occurrences over time
- alone a crude and ineffective intruder detector
- must determine both thresholds and time intervals

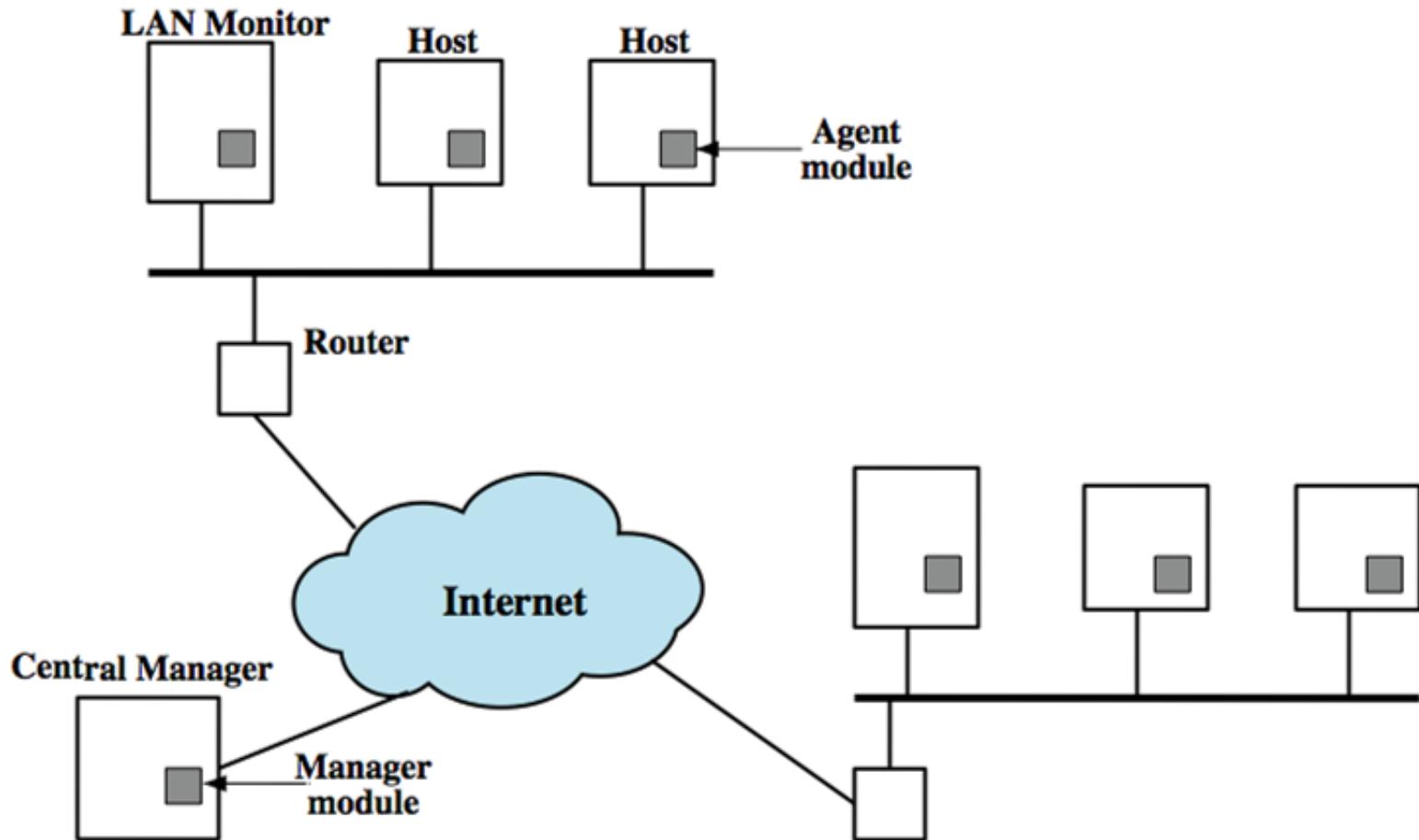
## ➤ profile based

- characterize past behavior of users / groups
- then detect significant deviations
- based on analysis of audit records
  - gather metrics: counter, gauge, interval timer, resource utilization
  - analyze: mean and standard deviation, multivariate, markov process, time series, operational model

# Signature Detection

- observe events on system and applying a set of rules to decide if intruder
- approaches:
  - rule-based anomaly detection
    - analyze historical audit records for expected behavior, then match with current behavior
  - rule-based penetration identification
    - rules identify known penetrations / weaknesses
    - often by analyzing attack scripts from Internet
    - supplemented with rules from security experts
  - Artificial Intelligence

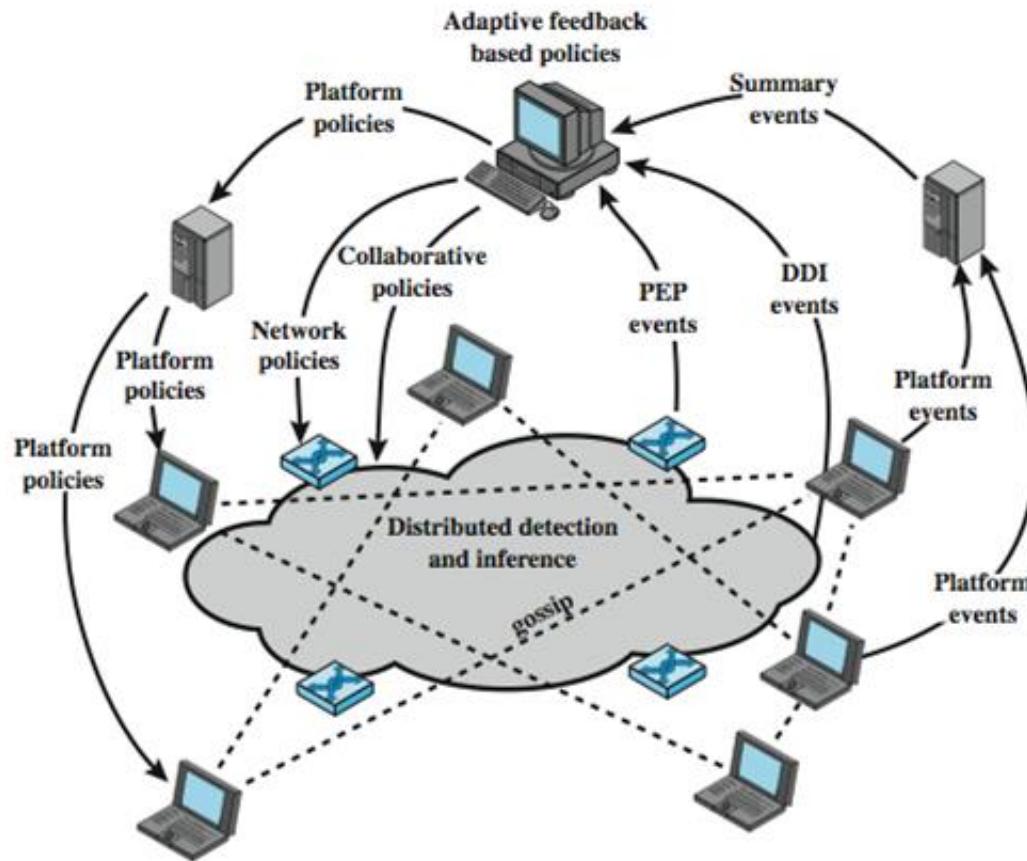
# Distributed Host-Based IDS



# Intrusion Detection Techniques

- signature detection
  - at application, transport, network layers; unexpected application services, policy violations
- anomaly detection
  - of denial of service attacks, scanning, worms
- when potential violation detected sensor sends an alert and logs information
  - used by analysis module to refine intrusion detection parameters and algorithms
  - by security admin to improve protection

# Distributed Adaptive Intrusion Detection



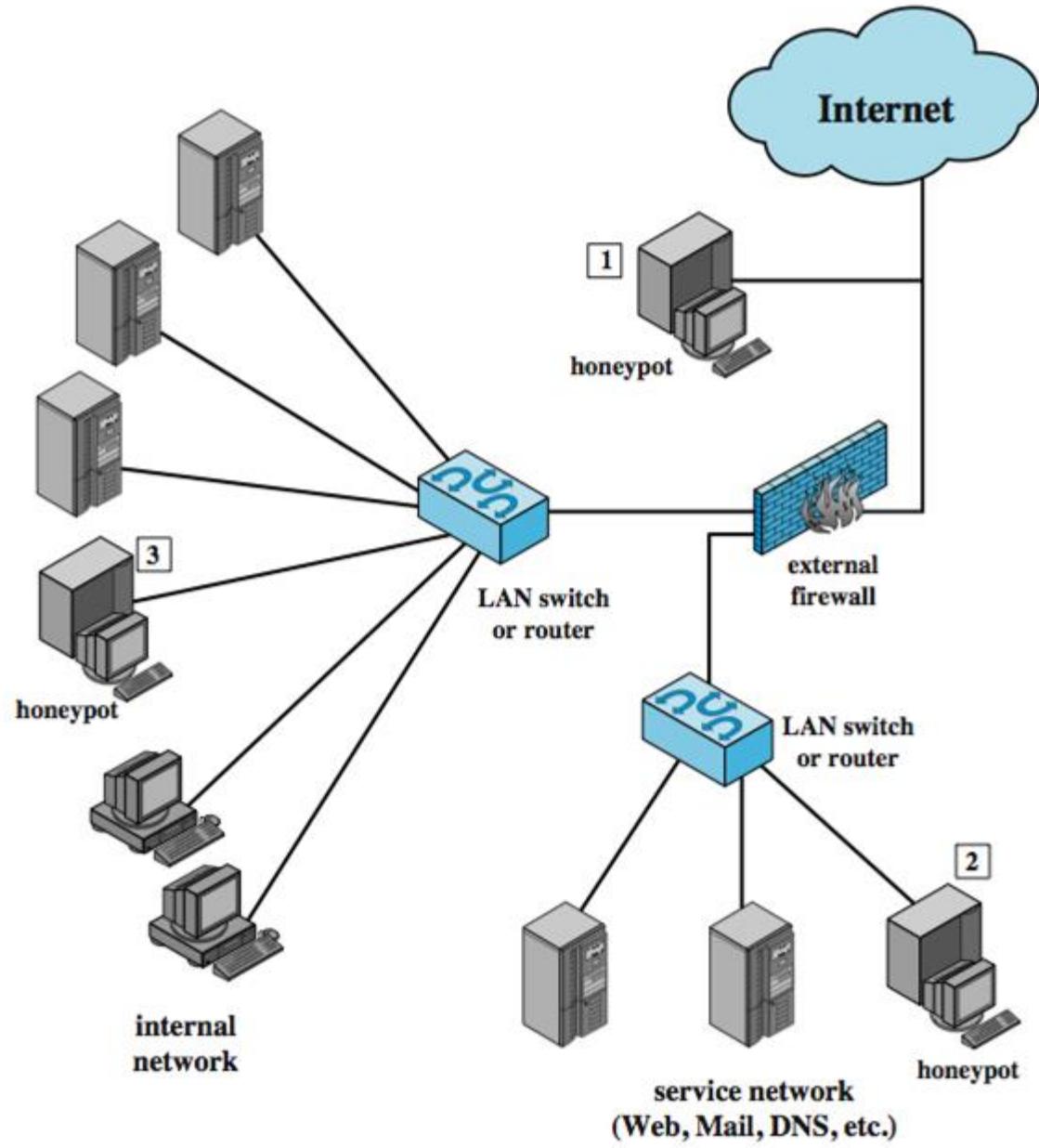
PEP = policy enforcement point

DDI = distributed detection and inference

# Honeypots

- are decoy systems
  - filled with fabricated info
  - instrumented with monitors / event loggers
  - divert and hold attacker to collect activity info
  - without exposing production systems
- initially were single systems
- more recently are/emulate entire networks

# Honeypot Deployment



# Software quality vs security

# Software Quality vs Security

- **software quality and reliability (in general)**
  - accidental failure of program
  - from theoretically random unanticipated input
  - improve using structured design and testing
  - not how many bugs, but how often triggered
- **software security is related**
  - but attacker chooses input distribution,  
specifically targeting buggy code to exploit
  - triggered by often very unlikely inputs
  - which common tests don't identify



# Defensive Programming

- a form of defensive design to ensure continued function of software despite unforeseen usage
- requires attention to all aspects of program execution, environment, data processed
- also called secure programming
- assume nothing, check all potential errors
- rather than just focusing on solving task
- must validate all assumptions

# Security by Design

- security and reliability common design goals in most engineering disciplines
  - society not tolerant of bridge/plane etc failures
- software development not as mature
  - much higher failure levels tolerated
- despite having a number of software development and quality standards
  - main focus is general development lifecycle
  - increasingly identify security as a key goal

# Handling Program Input

- incorrect handling a very common failing
- input is any source of data from outside
  - data read from keyboard, file, network
  - also execution environment, config data
- must identify all data sources

IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, VOL. 8, NO. 6, JUNE 2013

1027

## FM 99.9, Radio Virus: Exploiting FM Radio Broadcasts for Malware Deployment

Earlence Fernandes, Bruno Crispo, *Senior Member, IEEE*, and Mauro Conti, *Member, IEEE*

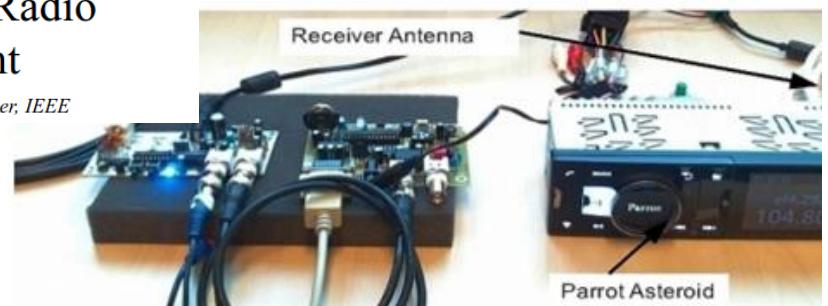


Fig. 4. System setup (devices placed adjacently only for illustrative purposes).

# Handling Program Input

- incorrect handling a very common failing
- input is any source of data from outside
  - data read from keyboard, file, network
  - also execution environment, config data
- must identify all data sources
- and explicitly validate assumptions on size and type of values before use

# Injection Attacks

- flaws relating to invalid input handling  
which then influences program execution
  - often when passed as a parameter to a helper program or other utility or subsystem
- most often occurs in scripting languages
  - encourage reuse of other programs / modules
  - often seen in web CGI scripts

# OS Security

- As a hardening process that includes:
    - planning installation, configuration, update, and maintenance.
    - for OS and key applications
  - For OS in general / Linux / Windows
- Small number of basic hardening measure
- Can prevent a large proportion of attacks seen in recent years
    - Australian report: implementing just the top four of the “top 35 Mitigation strategies” would have prevented over 70% of the targeted cyber intrusions investigated in 2009
      - 1) patch OS
      - 2) patch 3rd party apps
      - 3) restrict admin privileges to users who need them
      - 4) white-list approved applications