

Sistemi Distribuiti - Lezioni

Sara Angeretti

@Sara1798

2022/2023

Indice

1 Introduzione ai S. D.	6
1.1 Alcune definizioni	6
1.1.1 Definizione 1	6
1.1.2 Definizione 2	7
1.1.3 In sintesi	7
1.2 Sistemi come collezioni di nodi autonomi	8
1.2.1 Comportamenti	8
1.2.2 Collezioni di nodi	8
1.3 Sistemi coerenti	8
1.3.1 Essenzialmente	8
1.3.2 Trasparenza di come caratteristica fondamentale dei sistemi distribuiti	9
1.4 Sintesi delle caratteristiche di un sistema distribuito	9
2 Architetture Software	11
2.1 Definizione di architetture software	11
2.2 Modello base: Architetture a strati (layered)	11
2.2.1 Definizione	11
2.3 Architetture a livelli (tier)	12
2.4 Architetture basate sugli oggetti	12
2.5 Architetture centrate sui dati	12
2.6 Architetture basate su eventi	12
2.6.1 Sistemi Operativi Distribuiti	12
3 Il modello Client-Server	17
3.1 Visto ieri: Sintesi caratteristiche di un s.d.	17
3.2 Il modello Client-Server	18
3.3 Caratteristiche problematiche di ogni sistema distribuito	18
3.4 Trasparenza di distribuzione	19
3.5 Le basi dei sistemi distribuiti	19
3.5.1 Il concetto di protocollo	19

<i>INDICE</i>	3
---------------	---

3.5.2 Elementi minimi per creare un'applicazione	20
4 Stream-oriented communication - Le socket	22
4.1 Contenuti sintetici	22
4.2 Modello ISO/OSI	22
4.3 Comunicazione fisica - layering	22
4.4 Network edge	22
4.5 Processi e programmi	22
4.6 I servizi di trasporto Internet	23
4.7 Politiche dei servizi TCP/UDP	23
4.8 Socket: funzionamento di base	24
4.9 Aspetti critici	24
4.10 Identificare il server	25
4.11 Problemi fondamentali e TCP/IP	25
4.12 Comunicazione via socket	26
4.13 API socket system calls (Berkeley)	26
4.14 Riassunto: Politiche dei servizi TCP/UDP	27
4.15 Riassunto: Socket: funzionamento di base	27
4.16 Riassunto: Aspetti critici	28
4.17 Riassunto: Identificare il server	28
4.18 Riassunto: Problemi e TCP/IP	29
4.19 Riassunto: Comunicazione via socket	29
4.20 Riassunto: API socket system calls (Berkeley)	30
4.21 Processi e socket	31
4.22 Read e write	32
5 Le socket in Java	34
5.1 java.net.Socket	34
5.2 java.net.ServerSocket	35
5.2.1 Un esempio	36
5.3 Progettare un'applicazione con le socket	40
6 Architetture dei server	41
6.1 Tipi di server	41
6.2 Progettare un server iterativo	41
6.3 Progettare un server concorrente	42
6.4 Riassunto: Progettare un server iterativo	42
6.5 Riassunto: Progettare un server concorrente	43
6.5.1 I/O bloccante	44
6.5.2 Select System Call	45
6.5.3 Concurrent server structure	46

6.5.4	Sender Client	47
6.5.5	Concurrent Server	47
6.5.6	Concurrent Execution	48
6.6	Progettare un server multiprocesso	48
6.6.1	In C	49
6.6.2	Condivisione del canale	49
6.6.3	In Java	49
6.7	Confronto fra modelli	50
6.8	Conclusioni socket	51
7	Laboratorio1	52
7.1	Ripasso teoria	52
7.2	Esercizio 1	52
8	Introduzione alla concorrenza	53
8.1	Outline	53
8.2	Concorrenza come contemporaneità	53
8.3	Concorrenza e parallelismo	54
8.3.1	Tipi di parallelismo	54
8.4	Programmazione concorrente	55
9	Processi	56
9.1	Concetto di processo	56
9.2	Programmi e processi	56
9.3	Multiprogrammazione e multitasking	56
9.3.1	Multiprogrammazione	57
9.3.2	Multiprogrammazione e memoria	57
9.3.3	Multitasking	57
9.4	Op sui processi	57
9.4.1	Creazione di processi	58
9.4.2	Terminazione di processi	58
9.4.3	Es.: API Posix	59
10	Implementazione dei processi	60
10.1	I processi	60
10.1.1	Struttura	60
10.1.2	Immagine	60
10.1.3	Stato	61
10.2	Process Control Block (PCB)	61
10.3	Commutazione di contesto	61

<i>INDICE</i>	5
11 Multithreading in Java	62
11.1 Java threads	62
11.1.1 Il thread main	62
11.1.2 La classe principale per i Thread in Java	63
11.1.3 Far partire i thread: start()	64
11.2 Runnable interface	64
11.3 Thread: Alive o Terminated?	65
11.4 Stati di un thread	65
11.5 Operazioni sui thread	66
11.5.1 Cancellazione dei thread	66
11.6 Fork-join	66
12 Sincronizzazione	67
12.1 Programmi concorrenti e sequenziali	67
12.2 Meccanismi di sincronizzazione	67
12.2.1 Sincronizzazione su eventi	68
12.3 Problemi	71
12.3.1 Race condition	71
12.3.2 2	71
12.4 Come implementare la sincronizzazione	71
12.5 Come implementare la sincronizzazione	71
12.6 Come implementare la sincronizzazione	71
12.7 Come implementare la sincronizzazione	71
13 Variabili atomiche in Java	72
13.1 Il problema della visibilità	72
13.1.1 java.util.concurrent	73
13.2 Meccanismo di Locking	74
13.2.1 Problemi associati	74
13.2.2 Alternative al Locking: optimistic retrying	74
13.2.3 Strumento: Compare-and-Swap (CAS)	75
14 Esercizi	76
14.1 Es1	76
14.1.1 Specifica	76
14.1.2 Analisi	77
14.1.3 Implementazione	78
15 Liveness	84

Capitolo 1

Introduzione ai S. D.

Cominciamo ad introdurre i seguenti argomenti:

- Definizione di sistema distribuito
- Architetture software
- Il modello client-server
- Proprietà e caratteristiche fondamentali

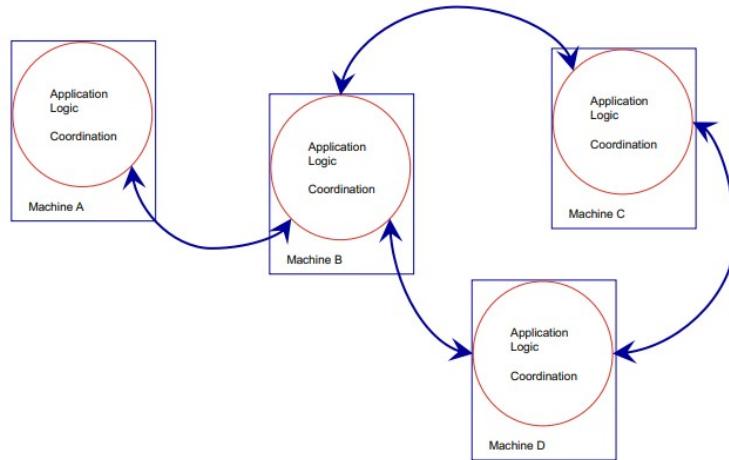
1.1 Alcune definizioni

1.1.1 Definizione 1

Ci sono diverse definizioni con alcuni aspetti in particolare in comune. Il libro di testo definire un *sistema distribuito* come un sistema in cui componenti hardware o software che sono localizzati in un sistema collegato alla rete, **comunicano** e **coordinano** le loro azioni solamente ("only by") passandosi messaggi.

In inglese: "We define a distributed system as one in which **hardware or software components** located at **networked computers** communicate and **coordinate** their actions **only by passing messages**."

Ricorda che stiamo parlando di **processi**, anche se ci sono processi che operano senza una rete ma sono eccezioni.

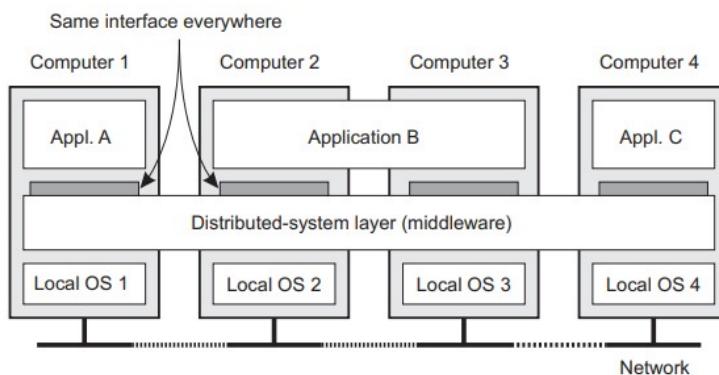


1.1.2 Definizione 2

Un'altra definizione è la seguente.

Un *sistema distribuito* è definito come un sistema di **elementi computativi autonomi** (che coesistono) che appaiono ad un utente come un'unica applicazione, un **unico sistema coerente**.

In inglese: "A distributed system is a collection of **autonomous computing elements** that appears to its users as a **single coherent system**."



1.1.3 In sintesi

Definizione:

- Un sistema distribuito è definito come un sistema di elementi computativi autonomi che coesistono e che appaiono ad un utente come un'unica applicazione, un unico sistema coerente.

Caratteristiche tipiche:

- Elementi computativi autonomi, anche noti come *nodi*; composti da device hardware o processi software
- Unico sistema coerente: gli utenti o le applicazioni percepiscono un singolo sistema

⇒ i nodi devono **collaborare**.

1.2 Sistemi come collezioni di nodi autonomi

1.2.1 Comportamenti

Sono **autonomi** e **indipendenti**, ovvero possono progredire come vogliono, ognuno ha la propria concezione del tempo ⇒ *non c'è* un clock globale, *non* è tutto sincronizzato.

Tutto ciò porta a *fondamentali problemi* di **sincronizzazione** e **coordinazione**.

1.2.2 Collezioni di nodi

Come gestire **appartenenze di gruppo** (o *group membership*)?

I **gruppi** possono essere **aperti/dinamici** (qualunque nodo può partecipare) o **chiusi/fissi** (solo nodi ben selezionati possono entrare nel sistema, questa nozione verrà commentata ulteriormente e comunque non troppo a fondo).

Ma quindi, come faccio a sapere se il nodo con cui sto comunicando è effettivamente **autorizzato**? Non ha proprio risposto.

1.3 Sistemi coerenti

1.3.1 Essenzialmente

La collezione di nodi opera nello stesso modo indipendentemente da dove, quando e come avvengono le interazioni fra l'utente e il sistema.

Esempi

1.4. SINTESI DELLE CARATTERISTICHE DI UN SISTEMA DISTRIBUITO

- Un utente finale (end user) non può dire dove stia avvenendo una computazione
- Dove i dati sono collezionati e stipati non dovrebbe essere rilevante per un'applicazione
- Se i dati siano stati replicati o meno è completamente nascosto

1.3.2 Trasparenza di come caratteristica fondamentale dei sistemi distribuiti

”Trasparenza di distribuzione (?)” come parola chiave. Da tradurre. Il problema principale: **fallimenti parziali**.

- È inevitabile che in qualsiasi momento solo una parte limitata del sistema distribuito fallisca.
- Nascondere fallimenti parziali e il loro ripristino è spesso molto complicato e generalmente impossibile da nascondere.

1.4 Sintesi delle caratteristiche di un sistema distribuito

Caratteristiche fondamentali per tutti i sistemi distribuiti:

Gestione della memoria?

- Non c'è memoria condivisa
- Comunicazione via scambio messaggi
- Non c'è stato globale: ogni componente (nodo, processo) conosce solo il proprio stato e può sondare lo stato degli altri.

Gestione dell'esecuzione?

- Ogni componente è autonomo =*i* esecuzione concorrente
- Il coordinamento delle attività è importante per definire il comportamento di un sistema/applicazione costituita da più componenti

Gestione del tempo (temporizzazione)?

- Non c'è un **clock globale**
- Non c'è possibilità di controllo/scheduling globale
- Solo coordinamento via scambio messaggi

Tipi di fallimenti?

- **Fallimenti indipendenti** dei singoli nodi (independent failures)
- Non c'è fallimento globale

Capitolo 2

Architetture Software

2.1 Definizione di architetture software

Un'architettura software definisce la struttura del sistema, le interfacce tra i componenti e i pattern di interazione (i protocolli).

I sistemi distribuiti possono essere organizzati secondo **diversi stili architettonici**.

2.2 Modello base: Architetture a strati (layered)

- Sistemi operativi
- Middleware

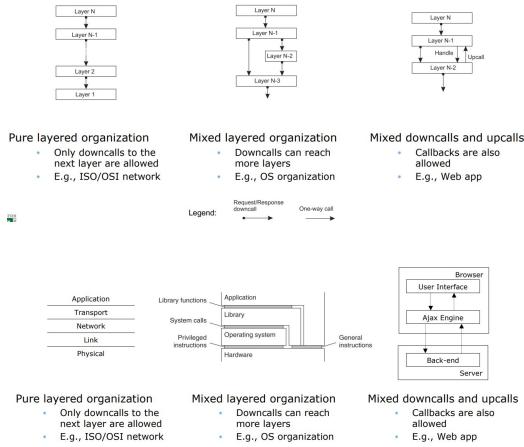
2.2.1 Definizione

Un'*architettura a strati* è un'architettura software che *organizza il software in strati*.

Ogni strato è *costruito sopra uno strato diverso più generico*.

Uno strato può essere definito liberamente insieme di (sotto)sistemi con lo stesso grado di generalità.

Gli strati più alti sono più specifici per applicazioni e i più bassi sono più generali/generici.



Quella al centro è da sapere benissimo in quanto oggetto di questo insegnamento.

2.3 Architetture a livelli (tier)

- Le applicazioni client server (2-tier, 3-tier)

2.4 Architetture basate sugli oggetti

- Java-Remote Method Invocation (RMI)

2.5 Architetture centrate sui dati

- Il Web come file system condiviso

2.6 Architetture basate su eventi

- Applicazioni Web dinamiche basate su callback (AJAX)

2.6.1 Sistemi Operativi Distribuiti

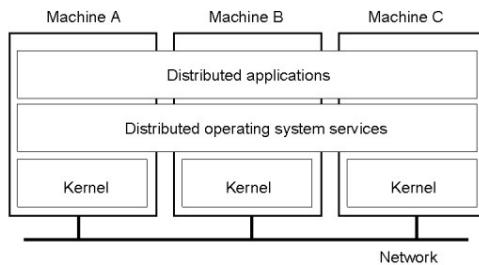
Diversi tipi:

- DOS (Distributed Operating Systems)
- NOS (Network Operating Systems)

- Middleware

System	Description	Main Goal
DOS	Tightly-coupled operating system for multi-processors and homogeneous multicompilers	Hide and manage hardware resources
NOS	Loosely-coupled operating system for heterogeneous multicompilers (LAN and WAN)	Offer local services to remote clients
Middleware	Additional layer atop of NOS implementing general-purpose services	Provide distribution transparency

DOS (Distributed Operating Systems)



Users not aware of multiplicity of machines

- Access to remote resources like access to local resources

Data Migration

- Transfer data by transferring entire file, or transferring only those portions of the file necessary for the immediate task

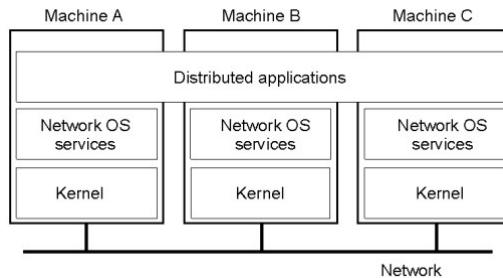
Computation Migration

- Transfer the computation, rather than the data, across the system

Process Migration - execute an entire process, or parts of it, at different sites

- Load balancing - distribute processes across network to even the workload
- Computation speedup - subprocesses can run concurrently on different sites
- Hardware preference - process execution may require specialized processor
- Software preference - required software may be available at only a particular site
- Data access - run process remotely, rather than transfer all data locally

NOS (Network Operating Systems)



Users are aware of multiplicity of machines
 NOS provides explicit communication features

- Direct communication between processes (socket)
- Concurrent (i.e., independent) execution of processes that from a distributed application
- Services, such as process migration, are handled by applications

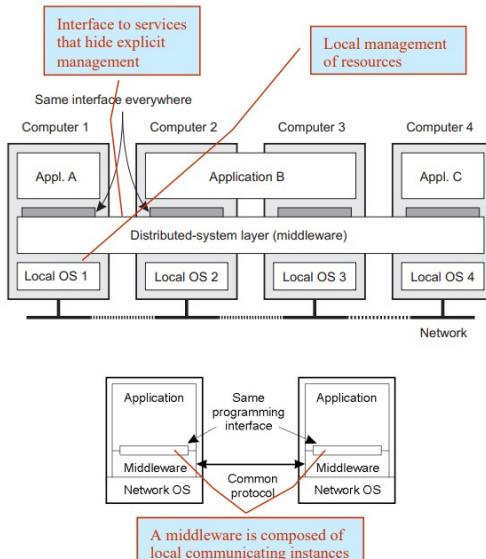
Access to resources of various machines is done explicitly by:

- Remote logging into the appropriate remote machine (telnet, ssh)
- Remote Desktop (Microsoft Windows)
- Transferring data from remote machines to local machines, via the File Transfer Protocol (FTP) mechanism

Middleware

Tieni a mente che è un concetto molto astratto. Qualsiasi cosa potrebbe essere considerata come un middleware (es. stupidi: un firewall, TCP/UDP, qualsiasi cosa funga da intermediario). È tutto ciò che sta "nel mezzo" fra l'inizio e la fine del tuo client/server (per ora lato server) serve a gestire servizi, e fare tante cose per aiutare la connessione. Sono funzionalità, sotto processi, cose.

Def. da Google: *Insieme di software che fungono da intermediari fra strutture e programmi informatici, permettendo loro di comunicare a dispetto della diversità dei protocolli o dei sistemi operativi.*



Distributed Operating Systems

- Make services (e.g., data storage and process execution) **transparent** to applications
- Rely on homogeneous machines (since they need to run the same software)

Network Operating Systems

- Services (e.g., data storage and process execution) **are explicitly managed** by applications
- Do not require homogeneous machines (since they may run different software)
- E.g., Mac OSX, Windows 10, Linux

Middleware

- Implements services (one or more) to **make them transparent** to applications
- E.g., Java/RMI

è importante capire che nel secondo schema dell'immagine, il middleware simula il comportamento dell'applicazione, ma i due middleware sono uguali ma possono comunicare tramite protocolli.

Servizi Middleware

Services can address several issues, from general to domain specific.
 Naming (il più importante) ovvero come faccio ad identificare un sistema operativo: astrazione

- Symbolic names are used to identify entities that are part of a DS
- They can be used by registries to provide the real addresses (e.g., DNS, RMI registries), or implicitly by the middleware

Access transparency (il più importante)

- ... defines and offers a communication model that hides details on message passing

Persistence

- ... defines and offers an automatic service for data storage (on file system or DB)

Distributed transactions (non vedremo tanto a fondo)

- ... defines and offers a persistence models to automatically ensure consistency on read/write operations (usually on DBs)

Security (non vedremo tanto a fondo)

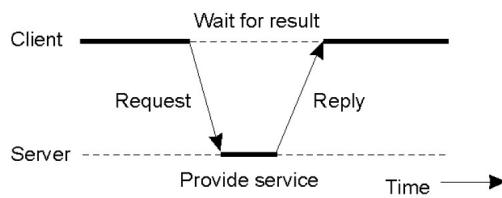
- ... defines and offers models to protect access to data and services (with different levels of permissions) and computation integrity

Item	Distributed OS		Network OS	Middleware-based OS
	Multiproc.	Multicomp.		
Degree of transparency	Very High	High	Low	High
Same OS on all nodes	Yes	Yes	No	No
Number of copies of OS	1	N	N	N
Basis for communication	Shared memory	Messages	Files	Model specific
Resource management	Global, central	Global, distributed	Per node	Per node
Scalability	No	Moderately	Yes	Varies
Openness	Closed	Closed	Open	Open

Capitolo 3

Il modello Client-Server

Il modello Client-Server è il modello di interazione tra un processo client e un processo server.



3.1 Visto ieri: Sintesi caratteristiche di un s.d.

Caratteristiche fondamentali per tutti i sistemi distribuiti:

Gestione della memoria?

- Non c’è memoria condivisa
- Comunicazione via scambio messaggi
- Non c’è stato globale: ogni componente (nodo, processo) conosce solo il proprio stato e può sondare lo stato degli altri.

Gestione dell’esecuzione?

- Ogni componente è autonomo = \downarrow esecuzione concorrente
- Il coordinamento delle attività è importante per definire il comportamento di un sistema/applicazione costituita da più componenti

Gestione del tempo (temporizzazione)?

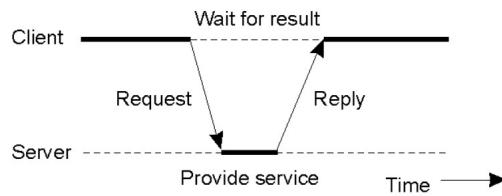
- Non c'è un clock globale
- Non c'è possibilità di controllo/scheduling globale
- Solo coordinamento via scambio messaggi

Tipi di fallimenti?

- Fallimenti indipendenti dei singoli nodi (independent failures)
- Non c'è fallimento globale

3.2 Il modello Client-Server

Il modello Client-Server è il modello di interazione tra un processo client e un processo server.



C'è uno strato verticale e uno orizzontale(?)

Configurazioni client/server

- Accesso a server multipli
- Accesso via proxy

3.3 Caratteristiche problematiche di ogni sistema distribuito

N.B.: (molto) probabile domanda d'esame.

Tutti i sistemi distribuiti vanno incontro a 4 problemi fondamentali che devono saper risolvere.

Vari step di risoluzione:

Identificare la controparte : fase di **naming**, dove assegnamo a un identificativo che deve necessariamente essere **univoco**;

Accedere alla controparte : fase di **access point**, una *reference* a cui possiamo fare riferimento;

Comunicazione 1 : fase di **protocol**, dove bisogna accordarsi su un formato condiviso di comunicazione (*ricevere l'informazione*);

Comunicazione 2 : questo è ancora un **open issue**, dove bisogna accordarsi su una convenzione di significato (*capire l'informazione*).

3.4 Trasparenza di distribuzione

Def.: consiste nel nascondere dettagli agli utenti, che ignorano cosa succede e (più importante) non possono influenzare il servizio fornito.

- Naming
 - Symbolic names are used to identify resources that are part of a distributed system
- Access transparency
 - Hide differences in data representation and how a local or remote resource is accessed
- Location transparency
 - Hide where a resource is located in the net
- Relocation or mobility transparency
 - Hide that a resource may be moved to another location while in use
- Migration transparency
 - Hide that a resource may move to another location
- Replication transparency
 - Hide that a resource is replicated
- Concurrency transparency
 - Hide that a resource may be shared by several independent users (ensuring state consistency)
- Failure transparency
 - Hide the failure and recovery of a resource.
- Persistence transparency
 - Hide that a resource is volatile or stored permanently

3.5 Le basi dei sistemi distribuiti

3.5.1 Il concetto di protocollo

Per poter capire le richieste e formulare le risposte i due processi devono concordare un **protocollo**.

I protocolli definiscono il **formato**, l'**ordine** di invio e di ricezione dei messaggi tra i dispositivi, il **tipo dei dati** e le **azioni** da eseguire quando si riceve un messaggio.

Le applicazioni su TCP/IP:

- si scambiano **stream di byte** di lunghezza infinita (il **meccanismo**)
- che possono essere segmentati in **messaggi** (la **politica**) definiti da un protocollo condiviso

Esempi di protocollo applicativi

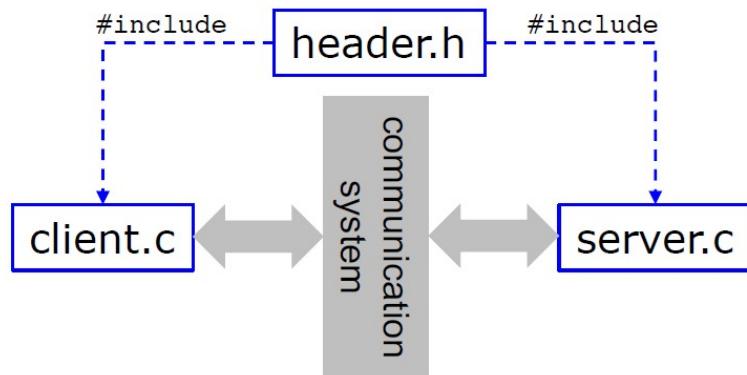
- HTTP - HyperText Transfer Protocol
- FTP - File Transfer Protocol
- SMTP - Simple Mail Transfer Protocol

3.5.2 Elementi minimi per creare un'applicazione

Definizione del protocollo di comunicazione.

Condivisione del protocollo tra gli attori dell'applicazione.

Un esempio in C:



Esempi: file server remoto

Il file header.h definisce il protocollo che usano sia il client sia il server.

```

/* Definitions needed by clients and servers. */
#define TRUE 1
#define MAX_PATH 255           /* maximum length of file name */
#define BUF_SIZE 1024          /* how much data to transfer at once */
#define FILE_SERVER 243         /* file server's network address */

/* Definitions of the allowed operations */
#define CREATE 1                /* create a new file */
#define READ 2                  /* read data from a file and return it */
#define WRITE 3                 /* write data to a file */
#define DELETE 4                /* delete an existing file */

/* Error codes. */
#define OK 0                    /* operation performed correctly */
#define E_BAD_OPCODE -1         /* unknown operation requested */
#define E_BAD_PARAM -2          /* error in a parameter */
#define E_IO -3                 /* disk error or other I/O error */

/* Definition of the message format. */
struct message {
    long source;              /* sender's identity */
    long dest;                /* receiver's identity */
    long opcode;              /* requested operation */
    long count;               /* number of bytes to transfer */
    long offset;              /* position in file to start I/O */
    long result;              /* result of the operation */
    char name[MAX_PATH];      /* name of file being operated on */
    char data[BUF_SIZE];      /* data to be read or written */
};
  
```

Struttura di un semplice server che realizza un rudimentale file server remoto.

```

1. #include <header.h>
2. void main(void) {
3.     struct message m1, m2;           /* incoming and outgoing messages */
4.     int r;                          /* result code */
5.
6.     while(TRUE) {
7.         receive(FILE_SERVER, &m1);   /* server runs forever */
8.         switch(m1.opcode) {        /* block waiting for a message */
9.             case CREATE: r = do_create(&m1, &m2); break;
10.            case READ: r = do_read(&m1, &m2); break;
11.            case WRITE: r = do_write(&m1, &m2); break;
12.            case DELETE: r = do_delete(&m1, &m2); break;
13.            default: r = E_BAD_OPCODE;
14.        }
15.        m2.result = r;              /* return result to client */
16.        send(m1.source, &m2);       /* send reply */
17.    }

```

Un client che usa il servizio per creare una copia di un file

```

1. #include <header.h>
2. int copy( char *src, char *dst){          /* procedure to copy file using the server */
3.     struct message m1;                   /* message buffer */
4.     long position;                      /* current file position */
5.     long client = 110;                  /* client's address */
6.     initialize();                       /* prepare for execution */
7.     position= 0;
8.
9.     do {
10.         m1.opcode = READ;                /* operation is a read */
11.         m1.offset = position;          /* current position in the file */
12.         m1.count = BUF_SIZE;           /* how many bytes to read */
13.         strcpy(m1.name, src);          /* copy name of file to be read to message */
14.         send(FILE_SERVER, &m1);        /* send the message to the file server */
15.         receive(client, &m1);          /* block waiting for the reply */
16.
17.         /* Write the data just received to the destination file */
18.         m1.opcode = WRITE;              /* operation is a write */
19.         m1.offset = position;          /* current position in the file */
20.         m1.count = m1.result;          /* how many bytes to write */
21.         strcpy(m1.name, dst);          /* copy name of file to be written to buf */
22.         send(FILE_SERVER, &m1);        /* send the message to the file server */
23.         receive(client, &m1);          /* block waiting for the reply */
24.         position += m1.result;         /* m1.result is number of bytes written */
25.     } while(m1.result > 0);
26.
27.     return(m1.result >= 0 ? OK : m1.result); /* return OK or error code */
28. }

```

Capitolo 4

Stream-oriented communication - Le socket

4.1 Contenuti sintetici

Breve ripasso del modello ISO/OSI per TCP/IP

Identificazione dei processi Indirizzi IP e Porte L'interfaccia API per le socket

Le socket in Java

I modelli architetturali

- Iterativo
- Concorrente mono processo
- Concorrente multi processo

4.2 Modello ISO/OSI

4.3 Comunicazione fisica - layering

4.4 Network edge

Chi è che si parla? I processi.

4.5 Processi e programmi

I programmi vengono eseguiti dai processi.

- Programma = sequenza di istruzioni eseguibili dalla “macchina”

I processi sono entità gestite dal Sistema Operativo

- Processo = area di memoria RAM per effettuare le operazioni e memorizzare i dati + registro che ricorda la prossima istruzione da eseguire + canali di comunicazione

Ogni processo comunica attraverso canali.

- Un canale gestisce flussi di dati in ingresso e in uscita (dati in formato binario o testuale)
- Per esempio lo schermo, la tastiera e la rete sono “canali”
- Dall'esterno ogni canale è identificato da un numero intero detto “porta”

Le socket sono particolari canali per la comunicazione tra processi che non condividono memoria (per esempio perché risiedono su macchine diverse).

Per potersi connettere o inviare dati ad un processo A, un processo B deve conoscere la macchina (host) che esegue A e la porta cui A è connesso (wellknown port).

4.6 I servizi di trasporto Internet

Servizio TCP

- *Orientato alla connessione*: il client invia al server una richiesta di connessione
- *Trasporto affidabile (reliable transfer)* tra processi mittente e ricevente
- *Controllo di flusso (flow control)*: il mittente rallenta per non sommergere il ricevente
- *Controllo della congestione (congestion control)*: il mittente rallenta quando la rete è sovraccarica
- *Non offre* garanzie di banda e ritardo minimi

Servizio UDP

- Trasporto non affidabile tra processi mittente e ricevente
- Non offre connessione, affidabilità, controllo di flusso, controllo di congestione, garanzie di ritardo e banda

D: perché esiste UDP?

Può essere conveniente per le applicazioni che tollerano perdite parziali (es. video e audio) a vantaggio delle prestazioni

4.7 Politiche dei servizi TCP/UDP

Servizio UDP:

- Scomponete il flusso di byte in segmenti
- Li inviate, uno per volta, ai servizi network

Servizio TCP:

- Scomponere e inviare come UDP
- Ogni segmento viene numerato per garantire:
 - Riordinamento dei segmenti arrivati
 - Controllo delle duplicazioni (scarto i segmenti con ugual numero d'ordine)
 - Controllo delle perdite (rinvio i segmenti mancanti)
- Per progettare e realizzare sistemi distribuiti
 - NON è necessario conoscere il funzionamento (information hiding) dei processi
 - Ciò che importa è lo scambio dati (stream di byte) tra i processi

4.8 Socket: funzionamento di base

TCP

- Utilizza variabili e buffer per realizzare il trasferimento bidirezionale di flussi di bytes (“pipe”) tra processi
- Prevede ruoli client/server durante la connessione
- NON prevede ruoli client/server per la comunicazione
- Utilizza i servizi dello strato IP per l’invio dei flussi di bytes

API: Application Programming Interface

- Definisce l’interfaccia tra applicazione e strato di trasporto

Socket: API per accedere a TCP e UDP

- Due processi (applicazione nel modello client server) comunicano inviando/leggendo dati in/da socket

4.9 Aspetti critici

Gestione del ciclo di vita di client e server

- Attivazione/terminazione del cliente e del server (es. Manuale o gestita da un middleware)
- “ Identificazione e accesso al server
- Informazioni che deve conoscere il cliente per accedere al server
- “ Comunicazione tra cliente e server
- Le primitive disponibili e

le modalità per la comunicazione (es. TCP/IP: Stream di dati inviati con send/receive) • Ripartizione dei compiti tra client e server • Dipende dal tipo di applicazione (es. controllo: una banca gestisce tutto lato server) • Influenza le prestazioni in relazione al carico (numero di clienti)

4.10 Identificare il server

Come fa il client a conoscere l'indirizzo del server? • Alternative: • inserire nel codice del client l'indirizzo del server espresso come costante (es. il client di un servizio bancario) • chiedere all'utente l'indirizzo (es. web browser) • utilizzare un name server o un repository da cui il client può acquisire le informazioni necessarie (es. Domain Name Service - DNS - per tradurre nomi simbolici) • adottare un protocollo diverso per l'individuazione del server (es. broadcast per DHCP)

4.11 Problemi fondamentali e TCP/IP

Come sono trattate le 4 problematiche fondamentali dei sistemi distribuiti con TCP e IP?

Identifico la controparte (naming):

Low level identification: the name of hosts and protocols

Accedo alla controparte (access point):

Use of the IP address (host:port) to access a process

Comunicazione 1 (protocollo):

Stream of bytes

Comunicazione 2 (sintassi e semantica):

Application protocols with predefined semantics (http, smtp)

What level of transparency? Very low: the programmer/user need to

- know network addresses
- parse bytes to get the content (message)

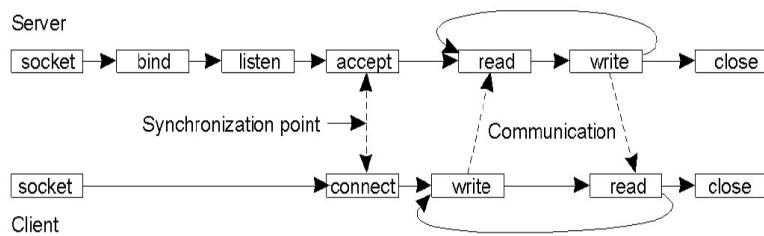
IMPORTANTE: TCP si può usare per qualsiasi tipo di comunicazione? Perché non c'è semantica. Quindi il punto di Comunicazione 2 non c'è. Questa potrebbe essere una domanda dell'esame.

4.12 Comunicazione via socket

La comunicazione TCP/IP avviene attraverso flussi di byte (byte stream), dopo una connessione esplicita, tramite normali system call read/write.

Read e write:

- Sono sospensive (bloccano il processo finché il sistema operativo non ha effettuato la lettura/scrittura)
 - Utilizzano un buffer per garantire flessibilità (es: la read definisce un buffer per leggere N caratteri, ma potrebbe ritornare avendone letti solo $k \leq N$)



4.13 API socket system calls (Berkeley)

Many calls are provided to access TCP and UDP services.

The most relevant ones are in the table below:

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket, set the queue length
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Write	Send some data over the connection
Read	Receive some data over the connection
Close	Release the connection

4.14 Riassunto: Politiche dei servizi TCP/UDP

Servizio UDP:

- Scomponete il flusso di byte in segmenti
- Li inviate, uno per volta, ai servizi network

Servizio TCP:

- Scomponete e inviate come UDP
- Ogni segmento viene numerato per garantire:
 - Riordinamento dei segmenti arrivati
 - Controllo delle duplicazioni (scarto i segmenti con ugual numero d'ordine)
 - Controllo delle perdite (rinvio i segmenti mancanti)
- Per progettare e realizzare sistemi distribuiti
 - NON è necessario conoscere il funzionamento (information hiding) dei processi
 - Ciò che importa è lo scambio dati (stream di byte) tra i processi

4.15 Riassunto: Socket: funzionamento di base

TCP

- Utilizza variabili e buffer per realizzare il trasferimento bidirezionale di flussi di bytes (“pipe”) tra processi
- Prevede ruoli client/server durante la connessione
- NON prevede ruoli client/server per la comunicazione
- Utilizza i servizi dello strato IP per l’invio dei flussi di bytes

API: Application Programming Interface

- Definisce l’interfaccia tra applicazione e strato di trasporto

Socket: API per accedere a TCP e UDP

Struttura usata per definire le comunicazioni

- Due processi (applicazione nel modello client server) comunicano inviando/leggendo dati in/da socket

4.16 Riassunto: Aspetti critici

Gestione del ciclo di vita di client e server

- Attivazione/terminazione del cliente e del server (es. Manuale o gestita da un middleware)

Identificazione e accesso al server

- Informazioni che deve conoscere il cliente per accedere al server

Comunicazione tra cliente e server

- Le primitive disponibili e le modalità per la comunicazione (es. TCP/IP: Stream di dati inviati con send/receive)

Ripartizione dei compiti tra i diversi componenti, in primo piano client e server

- Dipende dal tipo di applicazione (es. controllo: una banca gestisce tutto lato server)
- Influenza le prestazioni in relazione al carico (numero di clienti)

Qua naming non c'è.

4.17 Riassunto: Identificare il server

Come fa quindi il client a conoscere l'indirizzo del server? O anche il nome della macchina e della porta?

Alternative:

- inserire nel codice del client l'indirizzo del server espresso come costante (es. il client di un servizio bancario)
- chiedere all'utente l'indirizzo (es. web browser)

- utilizzare un name server o un repository da cui il client può acquisire le informazioni necessarie (es. Domain Name Service - DNS - per tradurre nomi simbolici)
- adottare un protocollo diverso per l'individuazione del server (es. broadcast per DHCP)

4.18 Riassunto: Problemi e TCP/IP

Come sono trattate le 4 problematiche fondamentali dei sistemi distribuiti con TCP e IP?

Sfruttiamo le 4 fasi tipiche che abbiamo già visto e previste per identificare ogni nuova tecnologia che ci troviamo davanti:

Identifico la controparte (naming): identificazione di basso livello, nome degli hosts e dei protocolli

Accedo alla controparte (access point): uso dell'indirizzo IP (host:port) per accedere ad un processo

Comunicazione 1 (protocollo): stream di bytes

Comunicazione 2 (sintassi e semantica): protocolli applicativi con semantiche predefinite (http, smtp)

What level of transparency? Molto basso: il programmatore/l'utente deve

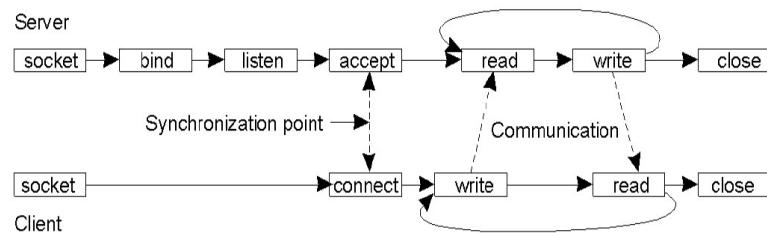
- conoscere l'indirizzo di rete
- fare il parsing di bytes per accedere al contenuto (messaggio)

IMPORTANTE: TCP si può usare per qualsiasi tipo di comunicazione? Perché non c'è semantica. Quindi il punto di Comunicazione 2 non c'è.
Questa potrebbe essere una domanda dell'esame.

4.19 Riassunto: Comunicazione via socket

La comunicazione TCP/IP avviene attraverso flussi di byte (byte stream), dopo una connessione esplicita, tramite normali **system call** **read/write**. Read e write:

- Sono sospensive (bloccano il processo finché il sistema operativo non ha effettuato la lettura/scrittura) → sistema-bloccanti
- Utilizzano un buffer per garantire flessibilità (es: la read definisce un buffer per leggere N caratteri, ma potrebbe ritornare avendone letti solo $k < N$)



N.B. importante: due cose e me ne sono persa una.

- le system call sono sistema-bloccanti
- ho read e write. Sono system calls ma pur sempre read e write, è il protocollo che uso che ne decide l'ordine. Ma è sempre "client-server"

Client e server prima di avviare la comunicazione devono **concordare** su un protocollo. Questo protocollo si occuperà di definire read e write (l'ultima è quella che effettivamente mi definisce la quantità di informazione con cui lavoro).

4.20 Riassunto: API socket system calls (Berkeley)

Questo è un po' un riassunto di quanto detto finora.

Molte chiamate sono previste per accedere ai servizi TCP e UDP.

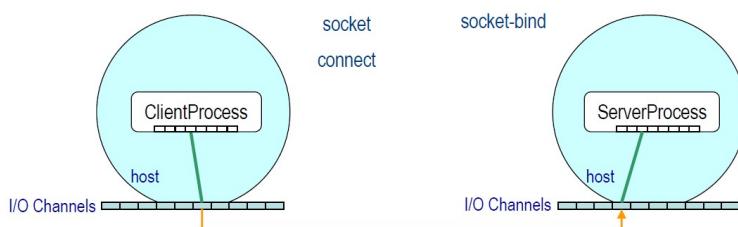
Le più importanti nella tabella seguente:

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket, set the queue length
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Write	Send some data over the connection
Read	Receive some data over the connection
Close	Release the connection

4.21 Processi e socket

Il server crea una socket collegata alla well-known port (che identifica il servizio fornito) dedicata a ricevere richieste di connessione.

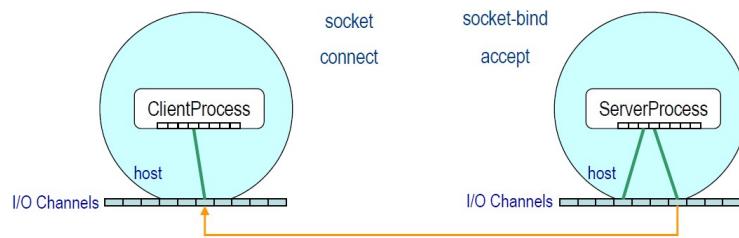
Con la accept(), il server crea una nuova socket, cioè un nuovo canale, dedicato alla comunicazione con il client.



Domanda: Chi stabilisce il formato della richiesta? L'applicazione o lo strato di trasporto (TCP)?

Ovviamente il protocollo. Ma dove avviene la connect? A livello di servizio, l'applicazione (qualunque essa sia) avviene da lì a destra (si occupa di read e write). Perciò la risposta è: *a livello di trasporto*.

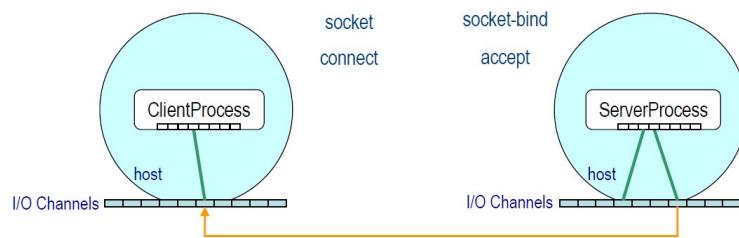
Si genera una seconda domanda: Perché non si usa la stessa socket?



Giriamo la domanda: **potrei** usare la stessa socket?

Sì, ma mi servirebbe un canale distinto per identificare i bit dell'indirizzo e quelli dell'informazione della comunicazione (parliamo di **programmazione strutturale**). Nel caso di uno stream, non sarei in grado di discriminare. Diventa complicato.

L'altro tipo di programmazione, p. dinamica, fa tutto lei invece di compartmentalizzare come fa la p. strutturale che abbiamo appena analizzato. La *programmazione dinamica* è più soggetta ad errori di quella *strutturale*.



4.22 Read e write

Le socket trasportano “stream” (= flussi) di bytes, quindi

- *non c'è il concetto di “messaggio”* (il flusso è continuo, senza fine),
- *la lettura/scrittura avviene per un numero **arbitrario** di byte*

Il prototipo (in pseudocodice) della read è quindi
`byteLetti read(socket, buffer, dimBuffer);`
 byteLetti = byte effettivamente letti
 socket = il canale da cui leggere
 buffer = lo spazio di memoria dove trasferire i byte letti
 dimBuffer = dimensione del buffer = numero max di caratteri che si possono leggere

Quindi si devono prevedere cicli di lettura che termineranno in base alla dimensione dei “messaggi” come stabilito dal formato del protocollo applicativo in uso.

Ovviamente quella read lì definita va in un while per passare tutta l'informazione.

Capitolo 5

Le socket in Java

Java definisce alcune classi che costituiscono un'interfaccia ad oggetti alle system call illustrate in precedenza. Ricorda che Java nasconde la gestione della memoria (e garbage collector) ma all'interno avviene come illustrato nella sezione precedente.

Le principali:

- `java.net.Socket`
- `java.net.ServerSocket`

Queste **classi** accorpano funzionalità e mascherano alcuni dettagli con il vantaggio di semplificare l'uso.

Come per ogni framework è necessario conoscerne il modello e il funzionamento per poterlo utilizzare in modo efficace.

Le prossime slide discutono i principali metodi delle due classi. SONO DA SISTEMARE

5.1 `java.net.Socket`

Constructors

`public Socket()`

Creates an unconnected socket, with the system-default type of `SocketImpl`.
`public Socket(String host, int port)` throws `UnknownHostException`, `IOException`
Creates a stream socket and connects it to the specified port number on the named host. If the specified host is null, the loopback address is assumed. The `UnknownHostException` is thrown if the IP address of the host could not be determined.
`public Socket(InetAddress address, int port)`

throws IOException Creates a stream socket and connects it to the specified port number at the specified IP address.

Methods to manage connections

public void bind(SocketAddress bindpoint) throws IOException Binds the socket to a local address. If the address is null, then the system will pick up an ephemeral port and a valid local address to bind the socket. public void connect(SocketAddress endpoint) throws IOException Connects this socket to the server. public void connect(SocketAddress endpoint, int timeout) throws IOException Connects this socket to the server with a specified timeout value (in milliseconds). public void close() Closes this socket.

Sono tutte bloccanti queste chiamate. Methods to establish I/O channels to exchange bytes public InputStream getInputStream() throws IOException Returns an input stream for this socket. If this socket has an associated channel then the resulting input stream delegates all of its operations to the channel. If the channel is in non-blocking mode then the input stream's read operations will throw an IllegalBlockingModeException. When a broken connection is detected by the network software the following applies to the returned input stream:

- The network software may discard bytes that are buffered by the socket. Bytes that aren't discarded by the network software can be read using read.
- If there are no bytes buffered on the socket, or all buffered bytes have been consumed by read, then all subsequent calls to read will throw an IOException.
- If there are no bytes buffered on the socket, and the socket has not been closed using close, then available will return 0.

public OutputStream getOutputStream() throws IOException Returns an output stream for writing bytes to this socket. If this socket has an associated channel, then the resulting output stream delegates all of its operations to the channel. If the channel is in non-blocking mode, then the output stream's write operations will throw an IllegalBlockingModeException.

5.2 java.net.ServerSocket

Constructors

public ServerSocket() throws IOException Creates an unbound server socket.
public ServerSocket(int port) throws IOException Creates a server socket, bound to the specified port. A port of 0 creates a socket on any free port. The maximum queue length for incoming connection indications (a request to connect) is set to 50. If a connection indication arrives when the queue is full, the connection is refused. public ServerSocket(int port, int backlog)

throws IOException Creates a server socket and binds it to the specified local port number, with the specified backlog. A port of 0 creates a socket on any free port. The maximum queue length for incoming connection indications (a request to connect) is set to the backlog parameter. If a connection indication arrives when the queue is full, the connection is refused.

Methods to manage connections

public void bind(SocketAddress endpoint) throws IOException Binds the ServerSocket to a specific address (IP address and port number). If the address is null, then the system will pick up an ephemeral port and a valid local address to bind the socket.
public void bind(SocketAddress endpoint, int backlog) throws IOException Binds the ServerSocket to a specific address (IP address and port number). If the address is null, then the system will pick up an ephemeral port and a valid local address to bind the socket. The backlog argument must be a positive value greater than 0. If the value passed is equal or less than 0, then the default value will be assumed.
public Socket accept() throws IOException Listens for a connection to be made to this socket and accepts it. Returns the new Socket. The method blocks until a connection is made.

Utility methods

public InetAddress getInetAddress() Returns the local address of this server socket or null if the socket is unbound.
public int getLocalPort() Returns the port on which this socket is listening or -1 if the socket is not bound yet.
public SocketAddress getLocalSocketAddress() Returns the address of the endpoint this socket is bound to, or null if it is not bound yet.

5.2.1 Un esempio

Let's study an example in which:

- A server accept a connection with a client, and then sends a stream of bytes (e.g., a string of characters)
- A client request a connection, and then reads the stream of bytes sent by the server

The goal is to give evidence that:

- The behavior of the involved processes is independent from each other

- They exchange streams of byte
- The notion of “message” or “application protocol” is not part of the socket definition

Sender Server Socket

```

1. import java.io.PrintWriter;
2. import java.net.ServerSocket;
3. import java.net.Socket;

4. public class SenderServerSocket {

5.     final static String message =
6.         "This is a not so short text to test the reading capabilities of clients.';

7.     public static void main(String[] args) {

8.         try {
9.             Socket clientSocket;
10.            ServerSocket listenSocket;

11.            listenSocket = new ServerSocket(53535);
12.            System.out.println("Running Server:" +
13.                " Host= " + listenSocket.getInetAddress() +
14.                " Port= " + listenSocket.getLocalPort());

15.            // loop to open a connection, send the message, close the connection
16.            while (true) {
17.                clientSocket = listenSocket.accept();
18.                System.out.println("Connected to client at port: " + clientSocket.getPort());
19.                PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);

20.                System.out.println("I am sending the message: \n" + message);
21.                System.out.println("message length: " + message.length());

22.                out.write(message);
23.                out.flush();

24.                clientSocket.close();
25.            }
26.        } catch (Exception e) {
27.            e.printStackTrace();
28.        }
29.    }
30.}

```

Receiver Client Socket

Nella slide seguente è possibile vedere un parser: serve perché i caratteri immessi sono char e stringhe.

```

1. import java.io.DataInputStream;
2. import java.io.IOException;
3. import java.net.InetAddress;
4. import java.net.Socket;

5. public class ReceiverClientSocket {

6.     public static void main(String[] args) {
7.         Socket socket; // my socket
8.         InetAddress serverAddress; // the server address
9.         int serverPort; // the server port

10.        try { // connect to the server
11.            serverAddress = InetAddress.getByName(args[0]);
12.            serverPort = Integer.parseInt(args[1]);
13.            socket = new Socket(serverAddress, serverPort);

14.            System.out.println("Connected to: " + socket.getInetAddress());

15.            DataInputStream in; // the source of stream of bytes
16.            byte[] byteReceived = new byte[1000]; // the temporary buffer
17.            String messageString = ""; // the text to be displayed

18.            // the stream to read from
19.            in = new DataInputStream(socket.getInputStream());
20.            System.out.println("Ready to read from the socket");

21.            // The following code shows in detail how to read from a TCP socket
22.            int bytesRead = 0; // the number of bytes read
23.            bytesRead = in.read(byteReceived);
24.            messageString += new String(byteReceived, 0, bytesRead);

25.            System.out.println("Received: " + messageString);
26.            System.out.println("I am done!");

27.            socket.close();
28.        } catch (IOException e) {
29.            e.printStackTrace();
30.        }
31.    }
32.}

```

Esecuzione

```

Problems @ Javadoc Declaration Console Console
SenderServerSocket [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home,
Running Server: Host=0.0.0.0/0.0.0.0 Port=53535
This is a not so short text to test the reading capabilities of clients.
message length: 72

Problems @ Javadoc Declaration Console Console
<terminated> ReceiverClientSocket [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Content
Connected to: localhost/127.0.0.1
Ready to read from the socket
Received: This is a not so short text to test the reading capabilities of clients.
I am done!

```

It works!

But the client coding is not correct! Why?

To discover the answer, let's introduce the Lazy Server: it's a server that sends a few bytes at a time (9 a 9) with a delay between sendings.

Lazy Sender Server Socket

```

15. // loop to open a connection, send the message, close the connection
16. while (true) {
17.     clientSocket = listenSocket.accept();
18.     PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
19.     System.out.println("message length: " + message.length());
20.     final int chunck = 9; // number of bytes sent each time
21.     boolean end = false;
22.     int i;
23.     for (i = 0; i < message.length() - chunck; i += chunck) {
24.         System.out.println(message.substring(i, i + chunck));
25.         Thread.sleep(1000); // lazy sender: wait 1" before sending
26.         out.write(message.substring(i, i + chunck));
27.         out.flush();
28.     }
29.     System.out.println(message.substring(i, message.length()));
30.     out.write(message.substring(i, message.length()));
31.     out.flush();
32.     clientSocket.close();
33. }
34. ...

```

The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The output window displays the following text:

```

LazySenderServerSocket [Java Application] /Library/Java/JavaV
Running Server: Host=0.0.0.0/0.0.0.0 Port=53535
message length: 72
This is a
not so s
hort text
to test
the ready
ng capabi
lities of
clients.

```

Receiver Client Socket execution

```

15. DataInputStream in; // the source of stream of bytes
16. byte[] byteReceived = new byte[1000]; // the temporary buffer
17. String messageString = ""; // the text to be displayed
18. // the stream to read from
19. in = new DataInputStream(socket.getInputStream());
20. System.out.println("Ready to read from the socket");
21. // The following code shows in detail how to read from a TCP socket
22. int bytesRead = 0; // the number of bytes read
23. bytesRead = in.read(byteReceived);
24. messageString += new String(byteReceived, 0, bytesRead);
25. System.out.println("Received: " + messageString);
26. System.out.println("I am done!");
27. socket.close();
28. } catch (IOException e) {
29.     e.printStackTrace();
30. }
31. }
32. }

```

Q: What's wrong with the code?

The screenshot shows the Eclipse IDE interface with the 'Terminal' tab selected. The output window displays the following text:

```

Problems @ Javadoc Declaration Console
<terminated> ReceiverClientSocket [Java Application]
Connected to: localhost/127.0.0.1
Ready to read from the socket
Received: This is a
I am done!

```

5.3 Progettare un'applicazione con le socket

Client: L'architettura è concettualmente più semplice di quella di un server

- È spesso un'applicazione convenzionale che usa una socket anziché da un altro canale I/O
- Ha effetti solo sull'utente client: non ci sono particolari problemi di sicurezza

Server: L'architettura generale prevede che

- venga creata una socket con una porta nota per accettare le richieste di connessione
- entri in un ciclo infinito in cui alternare:
 1. attesa/accettazione di una richiesta di connessione da un client
 2. ciclo lettura-esecuzione, invio risposta fino al termine della conversazione (stabilito spesso dal client)
 3. chiusura connessione

Problematiche connesse:

- L'affidabilità del server è strettamente dipendente dall'affidabilità della comunicazione tra lui e i suoi client
- La modalità connection-oriented determina:
 - l'impossibilità di rilevare interruzioni sulle connessioni (il client controlla il server) che potrebbero essere intromissioni esterne o problemi sulla rete;
 - la necessità di una connessione (una socket) per ogni conversazione;
 - problemi di sicurezza per la condivisione dei dati e il controllo affidato al client.

Capitolo 6

Architetture dei server

6.1 Tipi di server

I server possono essere:

iterativi: soddisfano una richiesta alla volta

concorrenti processo singolo: simulano la presenza di un server dedicato

concorrenti multi-processo: creano server dedicati

concorrenti multi-thread: creano thread dedicati

6.2 Progettare un server iterativo

Al momento di una richiesta di connessione il server crea una socket temporanea per stabilire una connessione diretta con il client.

Le eventuali ulteriori richieste per il server verranno accodate alla porta nota per essere successivamente soddisfatte.

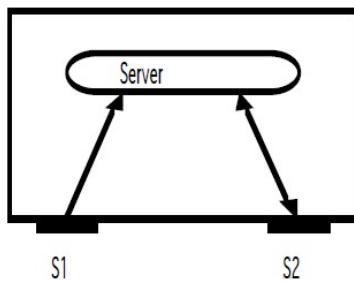
Vantaggi:

- Semplice da progettare

Svantaggi:

- Viene servito un cliente alla volta, gli altri devono attendere
- Un client può impedire l'evoluzione di altri client
- Non scala

Soluzione: server concorrenti



Legenda:

- S1 Socket per accettare richieste di connessione
- S2 Socket per connessioni individuali

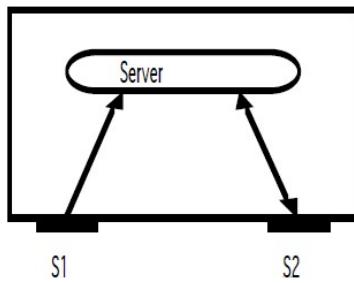
Manca una slide

6.3 Progettare un server concorrente

Un server concorrente può gestire più connessioni client.

La sua realizzazione può essere

- simulata con un solo processo (A) in C: funzione select, in Java: uso Selector che conoscere i canali ready to use (B) in Java: uso dei Thread
- reale creando nuovi processi slave (C) in C: uso della funzione fork



Legenda:

- S1 Socket per accettare richieste di connessione
- S2 Socket per connessioni individuali

(A) viene discusso nel seguito
 (B) verrà discusso nella parte di concorrenza
 (C) dovrebbe essere noto da sistemi operativi (faremo un breve ripasso)

Manca una slide

6.4 Riassunto: Progettare un server iterativo

Al momento di una richiesta di connessione il server crea una socket temporanea per stabilire una connessione diretta con il client. Di volta in volta, ad ogni chiamata.

Le eventuali ulteriori richieste per il server verranno accodate alla porta nota per essere successivamente soddisfatte.

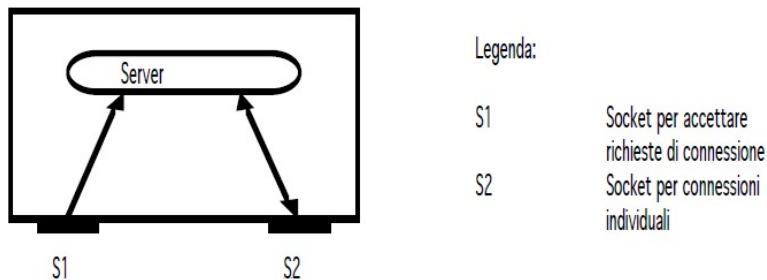
Vantaggi:

- Semplice da progettare

Svantaggi:

- Viene servito **un cliente alla volta**, gli altri devono attendere
- Un client può impedire l'evoluzione di altri client
- Non scala

Soluzione: server concorrenti, per poter servire più clients contemporaneamente.



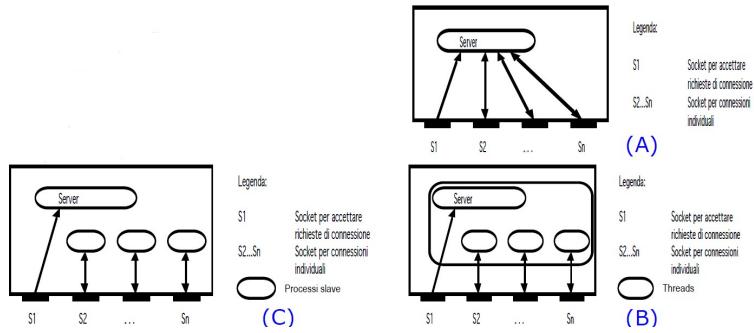
6.5 Riassunto: Progettare un server concorrente

Un server concorrente può gestire più connessioni client.

La sua realizzazione può essere

- simulata con un solo processo (A)
 - in C: funzione select
 - in Java: uso la classe Selector che riconosce i canali *ready to use*: questi canali semplificano la comunicazione perché
- (B) in Java: uso dei Thread • reale creando nuovi processi slave (C) in C: uso della funzione fork

B non è un sistema distribuito, perché gli elementi condividono memoria. In C invece non c'è condivisione di memoria.



Il problema qua è la gestione.

- (A) viene discusso nel seguito
 (B) verrà discusso nella parte di concorrenza
 (C) dovrebbe essere noto da sistemi operativi (faremo un breve ripasso)

```

1. ... // declarations
2. Socket[] clientSocket = new Socket[2];
3. DataInputStream[] in = new DataInputStream[2];
4. try {
5.     listenSocket = new ServerSocket(53535);
6.     // accept two connections
7.     clientSocket[0] = listenSocket.accept();
8.     clientSocket[1] = listenSocket.accept();
9.     // create two reading channels
10.    in[0] = new DataInputStream(clientSocket[0].getInputStream());
11.    in[1] = new DataInputStream(clientSocket[1].getInputStream());
12.    // what happens when start reading from one of the two channels?
13.    int bytesRead = 0;
14.    while (true) {
15.        bytesRead = in[0].read(byteReceived);
16.        if (bytesRead == -1)
17.            break; // no more bytes
18.        messageString += new String(byteReceived, 0, bytesRead);
19.        System.out.println("Received: " + messageString);
20.    }
21.    ... // the rest of the code (e.g., a similar loop to read from in[1])

```

Let's consider a server that accepts two clients to read from sockets and write on the console

This server is not concurrent.
 Why?
 How can we change the code?

6.5.1 I/O bloccante

Le operazioni di lettura e scrittura comportano l'uso di system call bloccanti.
 Ma cosa vuol dire?

Bloccante = si attende la conclusione dell'operazione richiesta prima di restituire il controllo al chiamante.

Per leggere in modo non bloccante serve sapere prima di fare una operazione di lettura o scrittura se il canale è pronto (cioè se faccio una operazione di lettura/scrittura il controllo mi viene restituito immediatamente).

La system call select() ha questo compito.

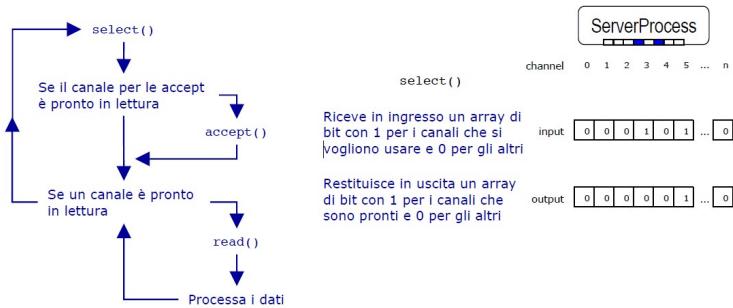
Il codice del server diventa:

1. Dico al sistema quali canali voglio usare in modalità non bloccante
2. Chiamo la select() che controlla quali canali sono «pronti»
3. Sui canali pronti effettuo l'operazione read() o write() desiderata
4. Ciclo tornando al punto 1

6.5.2 Select System Call

La select() permette gestire in modo non bloccante i diversi canali di I/O: sospende il processo finché non è possibile fare una operazione di I/O.

Un server concorrente realizza un ciclo:



La slide successiva non è importante, basta aver capito la logica (la fai facile) che ci sta dietro.

In C

```

#include <sys/types.h>
#include <sys/time.h>
int select(int maxfdp, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);



- Usa una maschera di bit fd_set di lunghezza maxfdp (= max_canali + 1)
  - max_canali è definito in <stdio.h>
    - In System V è la costante FOPEN_MAX
    - In 4.3BSD è dato da getdtablesize()
- Una invocazione di select segnala che
  - uno dei file descriptor di readfd è pronto per la lettura, o che
  - uno dei file descriptor di writefd è pronto per la scrittura, o che
  - uno dei file descriptor di exceptfd è in una eccezione pendente.
- Usa la maschera di bit di descriptors di file fd_set definita in <sys/types.h>
- Le macro per manipolare la maschera:
            

```

FD_ZERO(fd_set *fdset); /* clear all bits in fdset */
FD_SET(int fd, fd_set *fdset); /* turn the bit for fd on in fdset */
FD_CLR(int fd, fd_set *fdset); /* turn the bit for fd off in fdset */
FD_ISSET(int fd, fd_set *fdset); /* test the bit for fd in fdset */

```
- È possibile specificare un timeout
  - Se ha valore 0, allora ritorna subito dopo aver controllato i descrittori;
  - Se ha valore > 0, allora attende che uno dei descrittori sia pronto per l'I/O, ma non oltre il tempo fissato da timeout;
  - Se timeout ha valore NULL, allora attende indefinitamente che uno dei descrittori sia pronto per l'I/O.

```

NB: non fa parte del programma di esame. È uno spunto per chi volesse approfondire.

In Java

- Dalla versione 1.4 è stato introdotto i `channel` che possono operare in modo bloccante o non bloccante
 - Un canale non-bloccante non mette il chiamante in sleep
 - L'operazione richiesta o viene completata immediatamente o restituisce un risultato che nulla è stato fatto
- Solo canali di tipo socket possono essere usati nei due modi
- I canali socket permettono di interagire con i canali di rete
 - Sono implementati dalle classi `ServerSocketChannel`, `SocketChannel`, e `DatagramChannel`
- I canali socket in modo non bloccante possono essere usati con i `selector`
 - Possono essere gestite in modo più efficiente delle socket definite in `java.net`
 - Permettono di gestire più canali in multiplex

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/nio/channels/package-summary.html>
<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/nio/channels/Selector.html>
<https://developer.ibm.com/tutorials/j-nio/>

Java Selector

6.5.3 Concurrent server structure

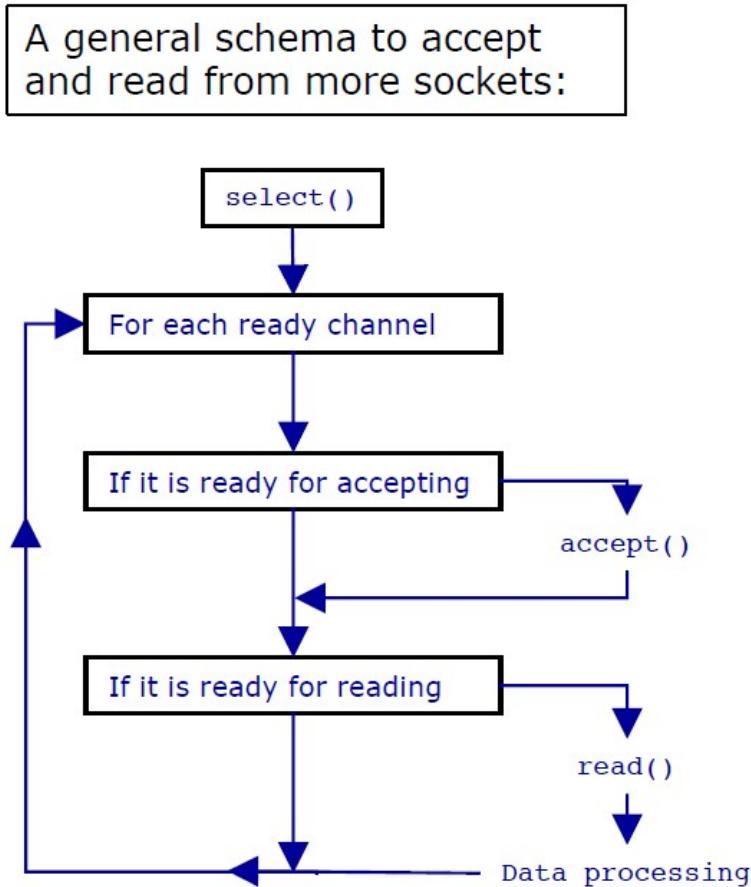
Codice + schema generale per accettare read da più sockets.

NB: di solito le system call (ma è la select di Java, non è la select in C che è sistema bloccante) sono bloccanti, qua c'è un middleware che agisce modificando la read e rendendola non bloccante.

```

1.  create ServerSocketChannel;
2.  create Selector;
3.  set the ServerSocketChannel in non-blocking mode
4.  associate the ServerSocketChannel with the Selector;
5.  while(true) {
6.      waiting events from the Selector;
7.      event arrived;
8.      create keys;
9.      for each key created by Selector {
10.          check the type of request;
11.          isAcceptable:
12.              get the client SocketChannel;
13.              associate that SocketChannel with the Selector;
14.              record it for read/write operations
15.              continue;
16.          isReadable:
17.              get the client SocketChannel;
18.              read from the socket;
19.              process the read data
20.              continue;
21.          isWriteable:
22.              get the client SocketChannel;
23.              write on the socket;
24.              continue;
25.      }
26.  }

```



6.5.4 Sender Client

Due slides da screeshottare.

6.5.5 Concurrent Server

Due slides da screeshottare.

Il secondo presenta la `accept()`. Mi restituisce una `SocketChannel`, dicendogli di volerci leggere sopra (posso anche scrivere ma l'esempio per problemi di spazio non presenta la `write`).

N.B.: due socket, due chiavi.

Glielo devo dire io che è una `SocketChannel` (riga 33).

6.5.6 Concurrent Execution

Una slide da screeshottare.

Si vede bene che il primo client chiude la socket (quarto riquadro dall'alto).
N.B.: è il client che scrive al server e il server legge.

6.6 Progettare un server multiprocesso

Un server concorrente che crea nuovi processi slave in C: uso della funzione ***fork()***.

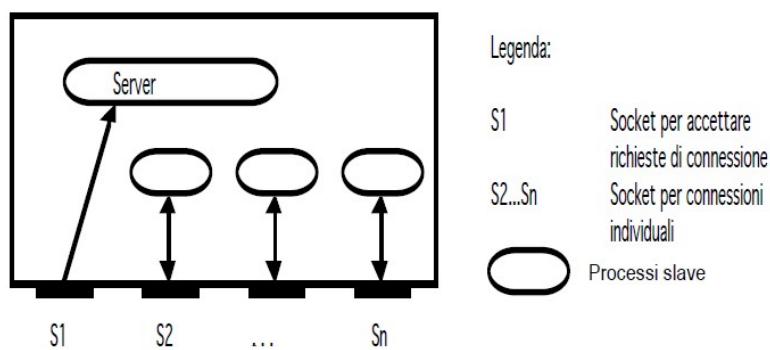
La ***fork()*** crea un processo clone del padre che

- eredita i canali di comunicazione
- esegue lo stesso codice

Il codice deve prevedere quindi che:

- Il padre chiuda la socket per la conversazione con il client
- Il figlio chiuda la socket per l'accettazione di nuove connessioni

La struttura del server è la stessa della versione iterativa in quanto ogni server gestisce un solo client.

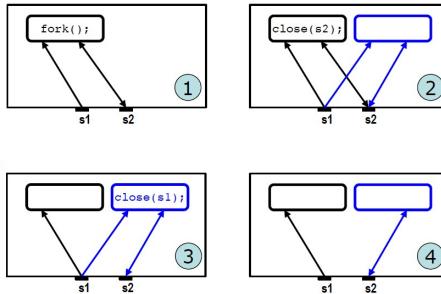


6.6.1 In C

```

1. int s1, s2;
2. ...
3. pid_t pid;
4. pid = fork();
5. if (pid == -1) {
6.     /* fork error - cannot create child */
7.     perror("Cannot create child");
8.     exit(1);
9. } else if (pid == 0) {
10.    /* code for child */
11.    close(s1);
12.    /* probably a few statements then an exec() */
13. } else {
14.    /* code for parent */
15.    printf("Pid of latest child is %d\n", pid);
16.    close(s2);
17.    /* more code */
18.    exit(0);
19. }

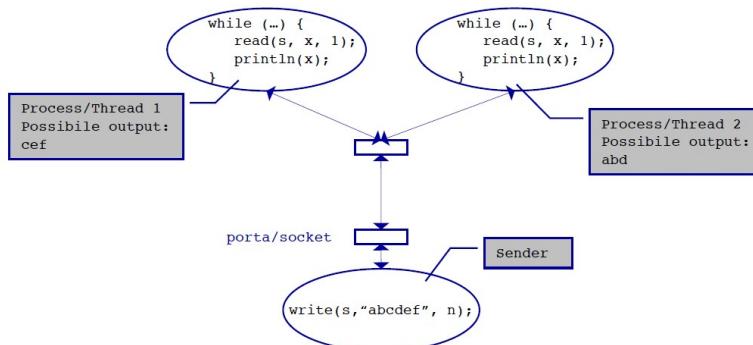
```



NB: non fa parte del programma di esame. È uno spunto per chi volesse approfondire.

6.6.2 Condivisione del canale

La lettura/scrittura su una socket da parte di più processi determina un problema di concorrenza: accesso ad una risorsa condivisa (mutua esclusione).



6.6.3 In Java

Niente fork(), ma i processi possono essere clonati:

```
public final class ProcessBuilder extends Object
```

This class is used to create operating system processes

- Each `ProcessBuilder` instance manages a collection of process attributes
- The `start()` method creates a new `Process` instance with those attributes
- The `start()` method can be invoked repeatedly from the same instance to create new subprocesses with identical or related attributes

An example

```

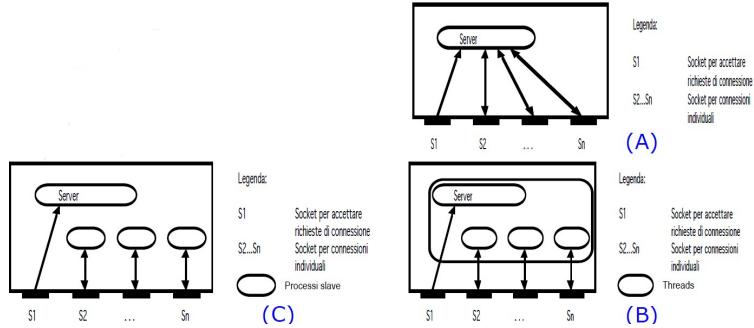
1. ProcessBuilder pb = new ProcessBuilder("C:\\Windows\\\\system32\\\\program.exe");
2. pb.inheritIO(); // <-- passes IO from forked process
3. try {
4.     Process p = pb.start(); // <-- forkAndExec on Unix
5.     p.waitFor(); // <-- waits for the forked process to complete
6. } catch (Exception e) {
7.     e.printStackTrace();
8. }

```

Reference: <https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/ProcessBuilder.html>
Example: <https://stackoverflow.com/questions/30355085/how-do-i-fork-an-external-process-in-java>

6.7 Confronto fra modelli

Tornando allo schema



Quindi quando conviene (B) e quando (C)?

Abbiamo detto che (B) prevede memoria condivisa. Quindi possiamo dire che (B) sia utile per casi in cui ho dati condivisi.

Es.: iscrizione ad un esame. Dato che è un'operazione piccola, mi basterebbe un server iterativo. Ma mi piace farlo concorrente. (B) o (C)? Decisamente (B), dove tutti vedono lo stesso registro (quindi elemento di memoria condivisa), ma devo **stare attento a dove scrivo, problema di concorrenza** che impareremo a gestire (così come anche per esempio la prenotazione di biglietti al teatro).

Ma posso usare (C)? Sì, ma dovrei creare un file esterno a cui accedere ogni volta e la lettura e scrittura del file è fuori dal mio controllo perché se ne occupa il sistema operativo. Mi complico la vita per niente. Basta (B) mettendo il controllo del mutuo accesso.

Un esempio irl per (C): così funziona UNIX con i suoi servizi. Ho un super server, arriva una richiesta a port 80: ho un processo che ascolta tutte le porte e quando arriva una richiesta su una certa porta per un servizio, fa una fork e una exec e fa partire il servizio. Basta che ci sia qualcuno che ascolta.

Quando conviene progettare un server mono processo (iterativo o concorrente) oppure multi processo? In altri termini: quali caratteristiche hanno? Mono processo (iterativo e concorrente)

- gli utenti condividono lo stesso spazio di lavoro
- adatto ad applicazioni cooperative che prevedono la modifica dello stato (lettura e scrittura)

Multi processo

- ogni utente ha uno spazio di lavoro autonomo
- adatto ad applicazioni cooperative che non modificano lo stato del server (sola lettura)
- adatto ad applicazioni autonome che modificano uno spazio di lavoro proprio (lettura e scrittura)

6.8 Conclusioni socket

Il protocollo

- di basso livello (flusso di byte/caratteri)
- l'applicazione si deve far carico della codifica/decodifica dei dati
- NB: non ci sono “messaggi” predefiniti: sono definiti a livello applicazione dal progettista

I servizi

- elementari: bassa trasparenza (solo meccanismi base)

Connessione

- non c'è un servizio di naming
- indirizzo fisico (host:port) per accedere

Non c'è supporto alla gestione del ciclo di vita

- creazione e attivazione esplicita dei componenti (client e server) (es. superserver in Unix)

Capitolo 7

Laboratorio 1

7.1 Ripasso teoria

Definizioni sistemi distribuiti, sockets, tipi di server per le connessioni client/server.

Client — Server

Il socket si occupa di aprire il canale di comunicazione fra i due lati. Abbiamo socket cliente e socket server.

Ci servono canali I/O per lo scambio di dati.

7.2 Esercizio 1

Eclipse workspace dedicato a SD a.a. 22/23, in particolare oggi abbiamo affrontato l'es.1.

Capitolo 8

Introduzione alla concorrenza

Prof. Ciavotta.

8.1 Outline

Quando ad un programma viene dato il *diritto* di operare su dei dati, questo diventa un processo.

Non ho capito perché, ma se un programma è statico il processo è dinamico.
Altra domanda: processi senza sistema operativo, possono esistere? Risposta: microcontrollori, sono processi senza s.o. monoprogrammati in cui è in esecuzione un unico programma che costituisce l'unico processo.

Vantaggi/svantaggi di un sistema monoprogrammato?

Non potrei eseguire nient'altro oltre quel singolo programma, e addio produttività. Per dire, già solo se stesse effettuando un'operazione di I/O o un calcolo molto potente: non risponderà più.

8.2 Concorrenza come contemporaneità

Def. concorrenza: identifica la contemporaneità di esecuzione di parti diverse di un programma. Questo programma può essere costituito da moduli chiamati *processi* o *thread* o ancora *agenti*.

due agenti possono esser in esecuzione "contemporanea" anche condividendo la CPU.

Due scenari:

Programmazione contemporanea: contemporaneità di esecuzione su **una stessa macchina** (con 1+ CPU/Core)

Programmazione distribuita: il programma è in esecuzione su **macchine distinte** collegate da una rete di comunicazione

Questa situazione presenta delle difficoltà: nel primo caso non avremo mai un'effettiva contemporaneità, ma li avrei sullo stesso sistema operativo quindi non riscontro particolari problemi di comunicazione. Nel secondo caso non ho capito benissimo, ma la rete di comunicazione è esterna quindi più soggetta a problemi.

Quindi avremo meccanismi che permettono l'esecuzione e la comunicazione tra gli agenti, soprattutto forniti dal SO, e costrutti di linguaggio che espandono la programmazione dal paradigma puramente sequenziale.

Quando parliamo di s.d. parliamo di un approccio che riguarda tanti settori del mondo dell'informatica.

8.3 Concorrenza e parallelismo

Concorrenza: capacità di far progredire più di un'attività nel tempo

Parallelismo: capacità di eseguire più di un'attività simultaneamente (da esecutori diversi)

Single core + multiprogrammazione = concorrenza **senza** parallelismo

Multicore = concorrenza **attraverso** il parallelismo.

8.3.1 Tipi di parallelismo

p. di dati: un certo dataset viene partizionato e le sue partizioni vengono assegnate ad attività diverse (roba dalla slide che era chiara)

p. di attività: attività diverse che lavorano sullo stesso gruppo di dati
sistemi ibridi

Perché sfruttare il parallelismo?

Risparmio di tempo, principalmente, ma anche efficienza di interazione con l'utente...

Se ho un programma con attività che possono essere eseguite in parallelo, boh con le chiacchiere non sento.

Viene formulata una legge, la **legge di Amdahl**, che mi fornisce il guadagno in termini di performance derivante dall'aggiunta di core ad un'applicazione che ha componenti sia sequenziali che parallele. In particolare, nella formula la parte parallelizzabile è quella del denominatore $"(1 - S)/N"$. Nota che all'aumentare del numero dei core, aumenta anche il valore dell'incremento di velocità.

8.4 Programmazione concorrente

La p. concorrente è la pratica di implementare dei programmi che contengano più flussi di esecuzione (processi o thread).

Il vantaggio di sfruttare processi multi-core (p. concorrente)? Posso strutturare in modo più adeguato un programma, creando per es. una sezione che effettui i calcoli, una che si occupi dell'interfaccia utente, ...

Principalmente, diverse sezioni per gestire diversi eventi.

Ma cos'è un evento? Chi lo gestisce?

Posso definirlo con parole mie come la situazione che viene creata dall'operato concorrente di utente e s.o.; gli eventi sono anche noti come segnali del s.o., messaggi che possono triggerare processi (di esecuzione, di termine, etc...).

Rispondendo

Capitolo 9

Processi

9.1 Concetto di processo

Un processo per un s.o. è un'**astrazione** di una parte attiva di un processo.
Cosa vuol dire attivare un processo?

Una fase è: prendo la memoria presente sul disco e la porto in memoria principale.

Un'altra è la creazione di azioni associate ad un processo, una serie di segnali che avviseranno altri processi che questo processo è stato creato, e vengono create le risposte a questi segnali.

9.2 Programmi e processi

Copia le slide.

Ma pensandoci, cos'è lo stack? è una struttura dati che aiuta il programma, con l'aiuto del processore, a ricordare quale fosse la funzione chiamante in un sistema multi-metodo o multi-funzione. Mi devo ricordare chi l'ha chiamata, quali sono i valori delle sue variabili, etc.

Ma è parte del programma (costituito essenzialmente da istruzioni) o del processo? Del processo ovviamente.

9.3 Multiprogrammazione e multitasking

Tra gli obiettivi del sistema operativo:

- Massimizzare l'utilizzo della CPU = Mantenere impegnata la (o le) CPU il maggior tempo possibile nell'esecuzione dei programmi

- Dare l'illusione che ogni processo abbia una CPU dedicata. Astrazione utile a chi sviluppa il programma

Due tecniche adottate nei sistemi operativi sono la multiprogrammazione e il multitasking (o timesharing)

- *Obiettivo della multiprogrammazione:* impedire che un programma che non è in condizione di proseguire l'esecuzione mantenga la CPU
- *Obiettivo del multitasking:* far sì che un programma interattivo reagisca agli input utente in un tempo accettabile (notare che è una tecnica non rilevante per i sistemi puramente batch)

9.3.1 Multiprogrammazione

Il sistema operativo mantiene in memoria i processi da eseguire. Li carica e gli assegna la memoria e una serie di altre informazioni.

Quando una CPU non è impegnata ad eseguire un processo, il sistema operativo seleziona un processo non in esecuzione e gli assegna la CPU.

Quando un processo non può proseguire l'esecuzione (ad es. perché deve attendere il termine dell'input di dati che gli servono per proseguire), la sua CPU viene assegnata ad un altro processo non in esecuzione.

Come risultato, se i programmi da eseguire sono più delle CPU, queste saranno impegnate nell'esecuzione di qualche processo per la maggior parte del tempo.

9.3.2 Multiprogrammazione e memoria

Heap: immagazzina strutture dati e dati creati in esecuzione per affiancare appunto l'esecuzione del programma.

Al momento di creazione di un oggetto, esso viene allocato nello heap.

9.3.3 Multitasking

Heap: immagazzina strutture dati e dati creati in esecuzione per affiancare appunto l'esecuzione del programma.

Al momento di creazione di un oggetto, esso viene allocato nello heap.

9.4 Op sui processi

I sistemi operativi di solito forniscono delle chiamate di sistema con le quali un processo può creare/terminare/manipolare altri processi.

Dal momento che solo un processo può creare un altro processo, all'avvio il sistema operativo crea dei processi «primordiali» dai quali tutti i processi utente e di sistema vengono progressivamente creati.

Principali operazioni: un processo può essere creato e terminato.

Si possono rappresentare con un diagramma ad albero, in quanto c'è un qualcosa(l'utente?) che crea un processo, il quale crea altri processi, e finché non li termino continuo a diramare.

9.4.1 Creazione di processi

Di solito nei sistemi operativi i processi sono organizzati in maniera gerarchica.

Un processo (padre) può creare altri processi (figli).

Questi a loro volta possono essere padri di altri processi figli, creando un albero di processi.

Uno di questi processi è lo shell, che serve ad interagire direttamente col s.o. La relazione padre/figlio è di norma importante per le politiche di condivisione risorse e di coordinazione tra processi.

Possibili politiche di condivisione di risorse: § Padre e figlio condividono tutte le risorse... § ... o un opportuno sottoinsieme... § ... o nessuna

Possibili politiche di creazione spazio di indirizzi: § Il figlio è un duplicato del padre (stessa memoria e programma)... § ... oppure no, e bisogna specificare quale programma deve eseguire il figlio

Possibili politiche di coordinazione padre/figli: § Il padre è sospeso finché i figli non terminano... § ... oppure eseguono in maniera concorrente

9.4.2 Terminazione di processi

I processi di regola richiedono esplicitamente la propria terminazione al sistema operativo.

Un processo padre può attendere o meno la terminazione di un figlio

Un processo padre può forzare la terminazione di un figlio. Possibili ragioni:

- Il figlio sta usando risorse in eccesso (tempo, memoria...)
- Le funzionalità del figlio non sono più richieste (ma è meglio terminarlo in maniera «ordinata» tramite IPC)
- Il padre termina prima che il figlio termini (in alcuni sistemi operativi)

Riguardo all'ultimo punto, alcuni sistemi operativi non permettono ai processi figli di esistere dopo la terminazione del padre

- Terminazione in cascata: anche i nipoti, pronipoti... devono essere terminati
- La terminazione viene iniziata dal sistema operativo

9.4.3 Es.: API Posix

fork() crea un nuovo processo figlio; il figlio è un duplicato del padre ed esegue concorrentemente ad esso; ritorna al padre un numero identificatore (PID) del processo figlio, e al figlio il PID 0 § exec() sostituisce il programma in esecuzione da un processo con un altro programma, che viene eseguito dall'inizio; viene tipicamente usata dopo una fork() dal figlio per iniziare ad eseguire un programma diverso da quello del padre § wait() viene chiamata dal padre per attendere la fine dell'esecuzione di un figlio; ritorna: § Il PID del figlio che è terminato § Il codice di ritorno del figlio (passato come parametro alla exit()) § exit() fa terminare il processo che la invoca: § Accetta come parametro un codice di ritorno numerico (tipicamente 0 o 1) § Il sistema operativo elimina il processo e recupera le sue risorse § Quindi restituisce al processo padre il codice di ritorno (se ha invocato wait(), altrimenti lo memorizza per quando l'invocherà) § Viene implicitamente invocata se il processo esce dalla funzione main (in C, e Java per esempio) § abort() fa terminare forzatamente un processo figlio

Capitolo 10

Implementazione dei processi

10.1 I processi

10.1.1 Struttura

Un processo è composto da diverse parti:

- Lo stato dei registri del processore che esegue il programma, incluso il program counter
- Lo stato della regione di memoria centrale usata dal programma, o immagine del processo
- Lo stato del processo stesso
- Le risorse del sistema operativo in uso al programma (files, semafori, regioni di memoria condivisa...)

Le risorse del sistema operativo possono essere condivise tra processi (a seconda del tipo di risorsa).

Processi distinti invece hanno *immagini* distinte.

10.1.2 Immagine

L'immagine di un processo è formata da:
§ Il codice del programma (text section)
§ La data section, contenente le variabili globali
§ Lo stack delle chiamate, contenente parametri, variabili locali e indirizzo di ritorno
§ Lo heap, contenente la memoria allocata dinamicamente durante l'esecuzione
§ Text e data section hanno dimensioni costanti per tutta la vita del processo
§ Stack e heap invece crescono / decrescono durante la vita del processo

10.1.3 Stato

Durante l'esecuzione, un processo cambia più volte stato. Gli stati possibili di un processo sono:

- § Nuovo (new): il processo è creato, ma non ancora ammesso all'esecuzione
- § Pronto (ready): il processo può essere eseguito (è in attesa che gli sia assegnata una CPU)
- § In esecuzione (running): le sue istruzioni vengono eseguite da qualche CPU
- § In attesa (waiting): il processo non può essere eseguito perché è in attesa che si verifichi qualche evento (ad es. il completamento di un'operazione di I/O)
- § Terminato (terminated): il processo ha terminato l'esecuzione

10.2 Process Control Block (PCB)

Detto anche **task control block**.

È la struttura dati del kernel che contiene tutte le informazioni relative ad un processo:

- Process state: ready, running...
 - Process number (o PID): identifica il processo
 - Program counter: contenuto del registro «istruzione successiva»
 - Registers: contenuto dei registri del processore
 - Informazioni relative alla gestione della memoria: memoria allocata al processo
 - Informazioni sull'I/O: dispositivi assegnati al processo, elenco file aperti...
 - Informazioni di scheduling: priorità, puntatori a code di scheduling...
 - Informazioni di accounting: CPU utilizzata, tempo trascorso...
- ...

10.3 Commutazione di contesto

Alla slide aggiungo che

Capitolo 11

Multithreading in Java

I thread sono **astrazioni del linguaggio**.

Diversi modelli in base all'astrazione che vogliamo.

11.1 Java threads

Sono astrazioni offerte della JVM e gestiti dalla stessa.

Tipicamente implementati sfruttando il modello di threading offerto dal sistema operativo (ma lo standard JVM non specifica come).

La JVM offre quindi una libreria per la definizione dei thread e per l'interazione con il Sistema Operativo per la loro esecuzione.

Modalità principali di creazione dei thread in Java:

- Sottoclasse della classe standard `java.lang.Thread`
- (Meglio) Implementazione metodo `run()` interfaccia `java.lang.Runnable`

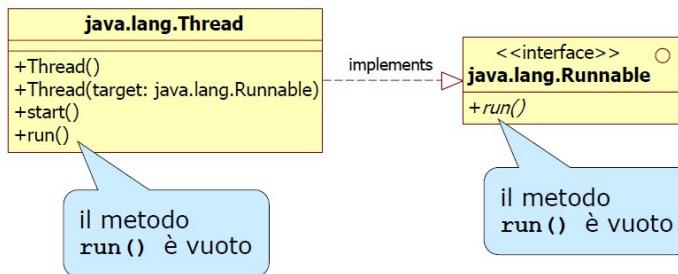
11.1.1 Il thread main

In Java ogni programma in esecuzione è un thread,

- Il metodo `main()` è associato al thread «main»
- Per poter accedere alle proprietà del thread in esecuzione è necessario ottenerne un riferimento tramite il metodo `currentThread()`



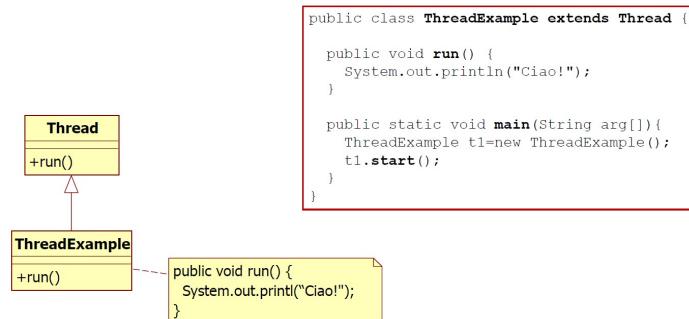
11.1.2 La classe principale per i Thread in Java



Il modo più semplice per creare ed eseguire un Thread è:

1. Estendere la classe `java.lang.Thread` (che contiene un metodo `run()` vuoto)
2. Riscrivere (ridisegnare, `override`) il metodo `run()` nella sottoclasse.
 - Il codice eseguito dal thread è incluso nel metodo `run()` e nei metodi invocati direttamente o indirettamente da `run()`
 - Questo è il codice che verrà eseguito in parallelo a quello degli altri thread
3. Creare un'istanza della sottoclasse
4. Richiamare il metodo `start()` su questa istanza
 NB: spesso si estende il costruttore in modo che questo invochi `start()`: così creare l'istanza della sottoclasse fa anche partire il thread.
 Questo viene chiamato **Pattern** del Thread.

Estensione della classe Thread



Domanda: quale thread termina per primo (sapendo che il main chiama un altro thread)?

Risposta: non possiamo saperlo, perché i thread non si attendono fra loro.

11.1.3 Far partire i thread: start()

- Una chiamata `t.start()` rende il thread `t` pronto (ready) all'esecuzione.
Chi chiama il metodo start? Il programmatore.
 - Prima o poi (quando lo scheduler lo riterrà opportuno) il thread verrà mandato in esecuzione e verrà invocato il metodo `run()` del thread `t`
È importante notare che non dobbiamo chiamare esplicitamente il metodo `run`, la sua invocazione avviene in maniera automatica (è implementata internamente alla classe `Thread`)
 - I due thread (creante e creato) saranno eseguiti in modo concorrente ed indipendente
Una applicazione Java ha almeno un thread
 - Importante:
L'ordine con cui ogni thread eseguirà le proprie istruzioni è noto, ma l'ordine globale in cui le istruzioni dei vari thread saranno eseguite effettivamente è indeterminato (nondeterminismo)
- Una delle conseguenze me la sono persa

11.2 Runnable interface

Siccome Java non consente l'ereditarietà multipla, un modo alternativo di realizzare un thread è implementare solamente il metodo `run` (ovvero implementare l'interfaccia `Runnable`), avendo la possibilità di estendere un'altra

classe base (maggior flessibilità).

La classe base Thread (il cui metodo run non fa nulla) può essere inizializzata con un oggetto Runnable, del quale utilizzerà il metodo run una volta che viene fatta partire.

11.3 Thread: Alive o Terminated?

Così abbiamo terminato la **creazione** di un thread. Ora lo dobbiamo usare. L'invocazione del metodo start() porta il Thread ad eseguire il metodo **run()** (l'entry point del thread):

- Un thread è considerato alive finché il metodo **run()** non termina
- Quando **run()** ritorna, il thread è considerato terminated

Una volta che un thread è terminato non può essere rieseguito (pena un'eccezione IllegalThreadStateException) -*i* se deve creare una nuova istanza.

Non si può far partire lo stesso thread (la stessa istanza) più volte!

Esiste il predicato (metodo che ritorna un booleano) **isAlive()** che può essere usato per valutare se il thread sia stato fatto partire e al contempo se non sia stato terminato (ovvero, se il metodo **run()** sia già terminato).

NB:

Solo la chiamata di **start()** crea un nuovo thread. Si può invocare **run()** direttamente, ma in questo modo il metodo verrà eseguito normalmente, sullo stack del thread corrente, senza che un nuovo thread venga creato: non lo fate!

IllegalThreadStateException

11.4 Stati di un thread

Ma chi riceve le richieste del thread? Il sistema operativo, che sa anche quando un thread è terminato e sa dove va accodato.

Es

In questo esempio il main non fa altro che inizializzare il programma e far partire un thread.

Appena eseguito il metodo **start()**, per un breve periodo, sono presenti due thread vivi allo stesso tempo.

Il primo thread a terminare è (probabilmente) quello del main.

Un programma termina quando tutti i suoi thread terminano.

11.5 Operazioni sui thread

3:

1. creazione
2. messa in pausa
3. terminazione

Per quanto riguarda la fase di pausa, due tipi:

- attiva
- passiva

Alcuni sistemi operativi non permettono ai processi di andare in wait, ovvero in un tipo di pausa passiva. Introduciamo allora un busy loop, che serve letteralmente a perdere un po' di tempo, in modo da avere il nostro tipo di wait.

Busy Loop

11.5.1 Cancellazione dei thread

`interrupt()` manda un flag, ovvero un booleano, al thread per farlo terminare. Però non si può assumere che sia effettivamente terminato.

Es.:

Problemi di `interrupt()`

11.6 Fork-join

Problema: se semplice, risolvi direttamente; se complesso, scomponi in task e risolvi singolarmente.

Capitolo 12

Sincronizzazione

12.1 Programmi concorrenti e sequenziali

I programmi concorrenti hanno delle proprietà molto diverse rispetto ai più comuni programmi sequenziali con i quali i programmatori hanno maggiore familiarità.

Interazioni tra agenti concorrenti

Cooperazione: interazioni "prevedibili e desiderate". La loro presenza è necessaria per la logica del programma. Avviene tramite scambio di informazioni (anche semplici come segnali) - Sincronizzazione diretta o esplicita

Competizione: gli agenti competono per accedere ad una risorsa condivisa. Politiche di accesso alla risorsa sono necessarie - Sincronizzazione può essere indiretta o implicita

Interferenze: interazioni "non prevedibili e non desiderate" - errori di programmazione, spesso dipendenti dalle tempistiche (time dependent) e non facilmente riproducibili

12.2 Meccanismi di sincronizzazione

Sono i meccanismi che permettono di controllare l'ordine relativo delle varie attività dei processi/thread
Se modello di memoria condivisa:

1. **mutua esclusione:** dati, regioni critiche del codice, non sono accessibili contemporaneamente a più thread

2. **sincronizzazione su condizione:** si sospende l'esecuzione di un thread fino al verificarsi di una opportuna condizione sulle risorse condivise

Se modello a scambio di messaggi:

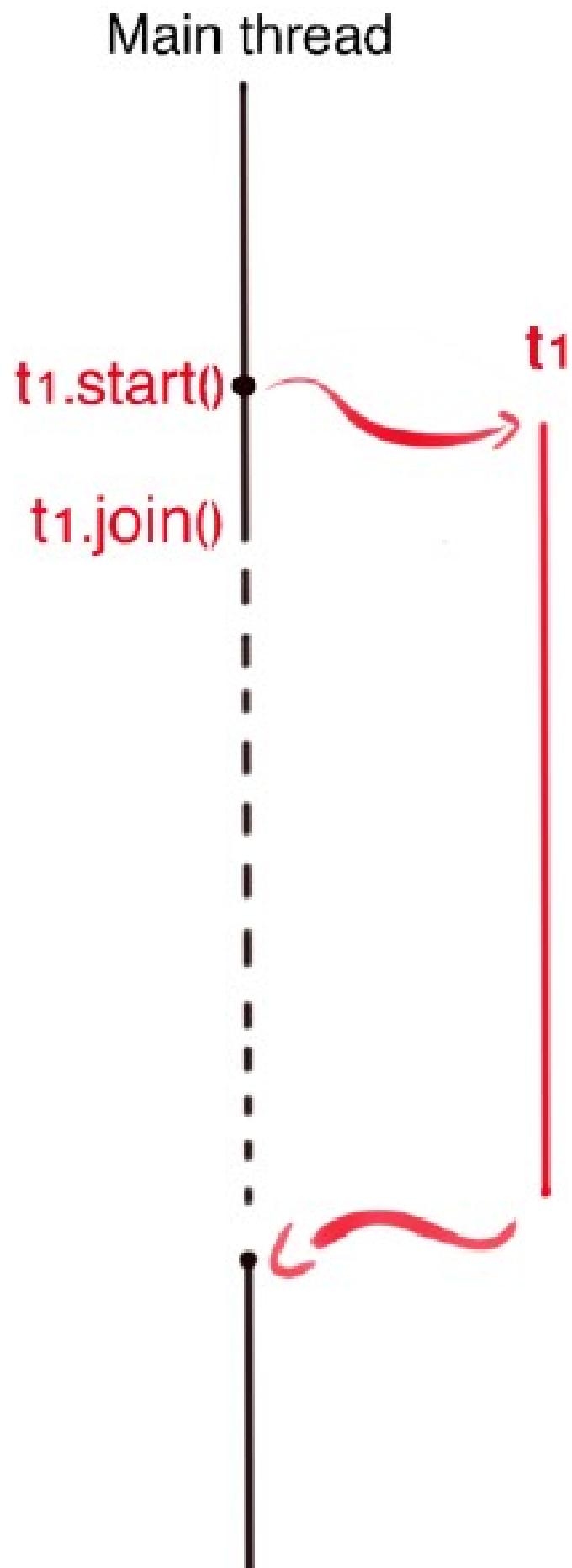
- di solito impliciti nelle primitive di *send* e *receive* == un thread può ricevere un messaggio solo dopo il suo invio
- **sincronizzazione su eventi:** si sospende l'esecuzione di un thread fino al verificarsi di un evento

Questi meccanismi vengono messi a disposizione dal sistema operativo nel caso di processi. In Java ritroviamo le stesse soluzioni implementate a livello di JVM per i thread.

12.2.1 Sincronizzazione su eventi

Join

- Esempio di metodo per la sincronizzazione temporale di 2 thread
- Il metodo `join` è definito nella classe Thread
- Quando si invoca il metodo `join()` su un thread, il thread chiamante entra in uno stato di attesa (wait). Rimane in tale stato finché il thread chiamato (`t1`) non termina
- Il metodo `join()` può anche ritornare se il thread chiamato viene interrotto. In questo caso, il metodo lancia una `InterruptedException`
- Infine, se il thread chiamato è già terminato o non è stato avviato, la chiamata al metodo `join()` ritorna immediatamente



Barriere

Meccanismo di sincronizzazione

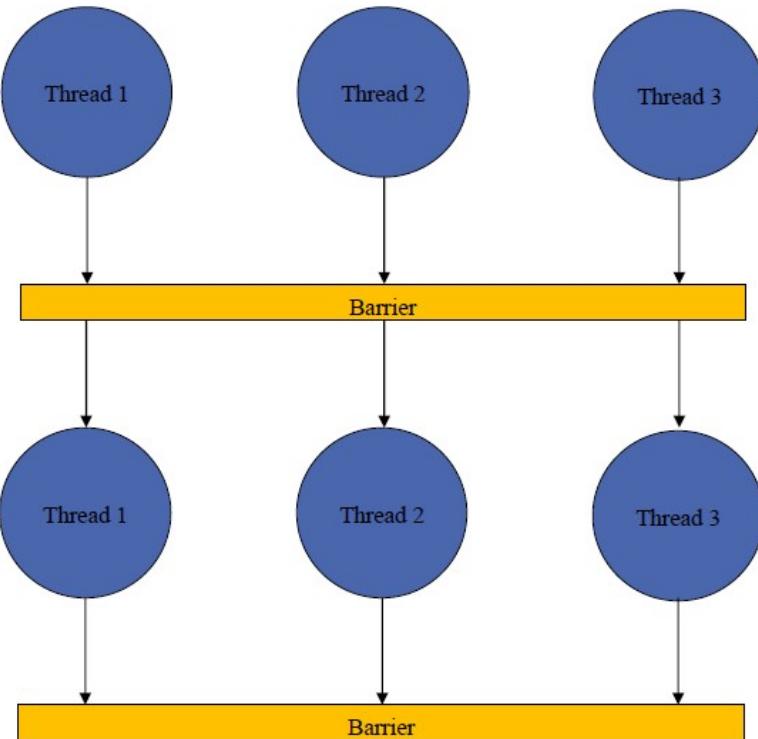
- I thread vengono messi progressivamente in attesa in attesa che una condizione globale non venga soddisfatta.

Esempio:

- un compito comune suddiviso in più fasi e ogni fase è assegnata a più thread
- ogni thread esegue il suo task ed attende che tutti gli altri abbiano terminato prima di proseguire alla fase successiva

Possibile implementazione:

- contatore condiviso conta il numero di processi che devono terminare
- il contatore decrementa ogni volta che un thread termina il suo task
- aggiornamento del contatore ”atomico”



12.3 Problemi

12.3.1 Race condition

12.3.2 2

12.4 Come implementare la sincronizzazione

12.5 Come implementare la sincronizzazione

12.6 Come implementare la sincronizzazione

12.7 Come implementare la sincronizzazione

Capitolo 13

Variabili atomiche in Java

Nel package `java.util.concurrent.atomic` sono definite una serie di classi che supportano le operazioni atomiche su singole variabili.

In ambienti multi-threaded, dove diversi thread operano su una singola variabile, un thread per volta può accedere e leggere o modificare la variabile. Se così non fosse si creerebbero stati di inconsistenza dei dati (race condition).

Tipi di atomicità:

Reale: una sola istruzione di CPU viene impiegata per eseguire l'operazione

Virtuale: il thread “crede” di avere accesso atomico alla variabile, concettualmente simile a monitor (oggetti con metodi `synchronized`)

13.1 Il problema della visibilità

Per le applicazioni multi-thread, è necessario garantire due condizioni per un comportamento coerente:

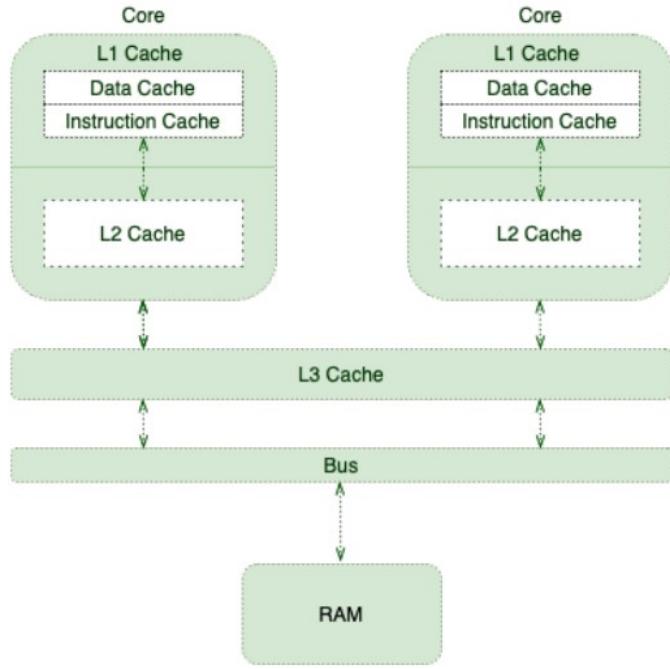
Mutua esclusione: solo un thread esegue una sezione critica alla volta.

Visibilità: le modifiche apportate da un thread ai dati condivisi sono visibili agli altri thread per mantenere la coerenza dei dati.

Problemi di visibilità si creano quando due thread sono in esecuzione su core diversi, perché le variabili condivise vengono tenute in cache per ragioni di performance.

I metodi e i blocchi sincronizzati forniscono entrambe le proprietà, a scapito delle performance.

Si può ovviare al problema della visibilità utilizzando variabili `volatile`.



Esempio: Contatore Non Sincronizzato

4 slides

13.1.1 java.util.concurrent

Alcune classi della libreria `java.util.concurrent` hanno delle performance superiori alle loro alternative bloccanti (es. `BlockingQueue` vs `ConcurrentLinkedQueue`), perché?

- Variabili atomiche (`AtomicInteger`, `AtomicLong`, ecc)
- Algoritmi non bloccanti

Variabili atomiche: `java.util.concurrent.atomic`

- ”Variabili volatile migliorate”
- Operazioni di aggiornamento che non richiedono lock, sono basate su operazioni atomiche

Algoritmi non bloccanti

- Sono thread-safe senza ricorrere a lock

- Usati per process scheduling, garbage collection, implementazione dei lock
- Più complessi degli equivalenti basati su lock bloccanti

13.2 Meccanismo di Locking

Lock in breve

1. **Acquisizione** - accesso ai diritti di sincronizzazione
2. **Algoritmo di attesa** - spin o sleep in attesa che la sincronizzazione sia disponibile
3. **Rilascio** - i diritti di accesso vengono rilasciati ed altri thread possono richiederli

13.2.1 Problemi associati

Se un thread non riesce ad acquisire un lock viene sospeso.

Context switch, risvegliare un thread presenta un costo.

Un thread in attesa non può eseguire nessuna operazione.

Un thread a bassa priorità può bloccare thread che ne hanno una più alta (priority inversion), infatti quando un thread acquisisce un lock nessun altro thread che ha bisogno di quel lock può proseguire.

Il processo di locking viene detto pessimistico. Infatti se la contesa è non è frequente, nella maggior parte dei casi la richiesta e l'esecuzione di un lock non è necessaria ed aggiunge overhead.

13.2.2 Alternative al Locking: optimistic retrying

Optimistic retrying

- Nessuna sincronizzazione in lettura
- Per eseguire una scrittura
 1. Lettura della variabile (creazione di una copia locale)
 2. Aggiornamento della copia
 3. Scrittura della variabile se non c'è collisione, altrimenti riprovare

Quest'approccio è particolarmente adatto all'aggiornamento delle strutture accessibili per mezzo di un unico indirizzo; es: aggiornamento di un Integer. I processori moderni offrono istruzioni (per la collision detection) di supporto

alla tecnica di optimistic retrying. 1: tmp = readMem(pos); 2: tmp = update(tmp) 3: if Collision(pos) goto 1 3: else writeMem(pos,tmp)

13.2.3 Strumento: Compare-and-Swap (CAS)

È una operazione atomica messa a disposizione da alcuni processori che prende in ingresso 3 argomenti:

- Una posizione di memoria V
- Un valore atteso E
- Un nuovo valore N

L'aggiornamento ha avuto successo

- Se il valore restituito è uguale al valore atteso E
- Altrimenti non c'è stato aggiornamento

compare-and-set Come CAS ma restituisce un true se l'operazione si è conclusa con successo, false altrimenti

Capitolo 14

Esercizi

14.1 Es1

14.1.1 Specifica

Un gruppo di amici ha inventato un semplice gioco di carte che ha le seguenti regole:

1. ogni giocatore riceve N carte di valore crescente da 1 a N;
2. ogni giocatore gioca una carta consegnandola ad un Arbitro senza vedere quella dell'avversario;
3. il giocatore che ha giocato la carta più alta ha realizzato una “presa”; se le carte hanno pari valore nessuno prende;
4. l’Arbitro comunica l’esito della giocata ai giocatori;
5. si ripete dal punto 2 fino ad esaurimento delle carte;
6. vince chi ha realizzato più “prese”; se le “prese” sono pari non vince nessuno;
7. il gioco termina con la dichiarazione “hoVinto” o “nonHoVinto” da parte dei giocatori.

Si chiede di implementare questa specifica in Java utilizzando Thread e monitor.

NOTA: per semplicità considerare N=10 e 3 giocatori. Si trascurino i dettagli non precisati nel testo (per esempio non conta l’ordine con cui vengono giocate le carte (punto 2) ...).

14.1.2 Analisi

Ogni giocatore può essere modellato usando un thread.

- Un giocatore gioca una carta, aspetta un certo tempo e prova a giocare ancora.
- Per poter giocare ancora è necessario che la mano sia finita.
- Il thread finisce quando tutte le carte del giocatore sono state giocate e il giocatore ha saputo che se ha vinto o ha perso.

L'arbitro è la risorsa condivisa.

- Deve implementare un metodo per giocare una carta.
- Metodo per annunciare il vincitore.
- Deve conoscere il turno attuale.
- Deve ricordare lo stato di ogni turno precedente (quali carte sono state giocate e chi ha vinto ogni turno).

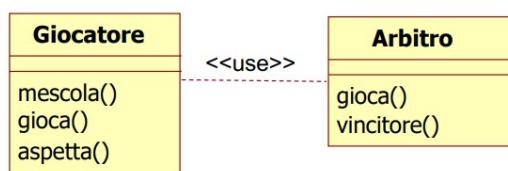
Il gioco è diviso in turni.

- Un giocatore deve aspettare che tutti gli altri abbiano finito per giocare ancora (condizione di accesso alla risorsa).
- Il turno è un punto di sincronizzazione. Quando scatta un nuovo turno eventuali giocatori in attesa possono giocare la loro carta.
- Un giocatore chiede se ha vinto quando ha giocato tutte le carte, deve essere finito l'ultimo turno.

Class Diagram

Diagramma delle classi di alto livello con l'obiettivo di identificare chiaramente gli attori coinvolti e le azioni principali. Modellato in UML.

Le azioni identificate possono anche non essere implementate sotto forma di metodi di classe.



Giocatore

Ogni giocatore ha un nome (id numerico) ed N carte (da il cui valore va da 1 a N, semplifichiamo).

Il giocatore mescola le carte (*-i shuffle* di numeri interi).

Ci sono tanti turni quante carte, quindi ciclicamente succede che:

- Il giocatore gioca una carta.
- Il giocatore aspetta un certo tempo random e prova a giocare di nuovo.
- Se il turno non è finito, il giocatore aspetta (va messo in wait).

Quando il giocatore ha finito le carte, questo chiede all'arbitro il nome del vincitore.

- Se l'arbitro restituisce il suo id scrive «Sono io il vincitore».
- Se l'arbitro restituisce l'id di un altro giocatore scrive «Non ho vinto».

Dopo aver stampato a schermo il risultato della partita il thread del giocatore termina.

Arbitro

Metodo `gioca(id_giocatore, carta)`

Deve essere sincronizzato, solo un giocatore per volta può dare la sua carta

Se un giocatore ha già giocato e prova a giocare di nuovo deve essere messo in attesa che il turno finisca

Quando un turno finisce (tutti i giocatori hanno giocato) i giocatori in attesa potranno essere risvegliati

Metodo `vincitore()`

Restituisce l'id del giocatore vincitore

Se è sincronizzato possiamo usare `wait()` per evitare che un giocatore chieda l'esito della partita prima che questa termini.

14.1.3 Implementazione

Classe Giocatore:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
```

```
public class Giocatore implements Runnable {  
  
    private int idGiocatore;  
  
    private Arbitro arbitro;  
  
    private int numeroCarte;  
  
    private List<Integer> listaCarte = null;  
  
    public Giocatore(int idGiocatore, Arbitro arbitro, int numeroCarte) {  
        this.idGiocatore = idGiocatore;  
        this.arbitro = arbitro;  
        this.numeroCarte = numeroCarte;  
        listaCarte = new ArrayList<>(numeroCarte);  
        for (int i = 0; i < numeroCarte; i++) {  
            listaCarte.add(i + 1);  
        }  
        Collections.shuffle(listaCarte);  
    }  
  
    public void run(){  
  
        for(int i = 0; i < numeroCarte; i++) {  
            arbitro.gioca(idGiocatore, listaCarte.get(i));  
            try {  
                Thread.sleep(100);  
            } catch (InterruptedException e) {  
                break;  
            }  
        }  
  
        int vincitore = arbitro.vincitore();  
        if(vincitore == idGiocatore){  
            System.out.println("Sono io il vincitore! - Thread " + idGiocatore);  
        }  
        else {  
            // System.out.println("Sono io il vincitore! - Thread " + idGiocatore);  
            System.out.println("Sono il thread " + idGiocatore + " e ho perso!");  
        }  
    }  
}
```

```
    }
}
```

Classe Arbitro:

```
public class Arbitro{

    private int numeroGiocatori;

    private int numeroCarte;

    private final int maxTurni;

    private int[] vincite;

    private int[][] giocate;

    private int turno = 1;

    private int numeroGiocate = 0;

    private boolean gameOver = false;

    public Arbitro(int numeroGiocatori, int numeroCarte) {
        this.numeroGiocatori = numeroGiocatori;
        this.numeroCarte = numeroCarte;
        this.maxTurni = numeroCarte;
        vincite = new int[numeroGiocatori];
        //la riga sopra assume che i giocatori siano ordinati in modo sequenziale
        for(int i = 0; i < numeroGiocatori; i++) {
            vincite[i] = 0;
        }
        giocate = new int[numeroGiocatori][maxTurni];
        for(int i = 0; i < numeroGiocatori; i++) {
            for(int j = 0; j < maxTurni; j++) {
                giocate[i][j] = 0;
            }
        }
    }

    public synchronized void gioca(int idGiocatore, int carta) {
```

```
while(giocate[idGiocatore] [turno-1] != 0){
    try {
        System.out.println("Il giocatore " + idGiocatore + " è in pausa.");
        wait();
    } catch (InterruptedException e) {
    }
}

giocate[idGiocatore] [turno-1] = carta;
System.out.println("Il giocatore");
numeroGiocate++;

if(numeroGiocate == numeroGiocatori) {
    System.out.println("___Turno terminato___");
    aggiornaVincitore();
    numeroGiocate = 0;
    turno++;
    if(turno > maxTurni) {
        gameOver = true;
        System.out.println("___Partita terminata___");
    }
    notifyAll();
}
}

private void aggiornaVincitore() {
    int max = 0;
    int idMax = 0;
    for (int i = 0; i < numeroGiocatori; i++) {
        if(giocate[i] [turno-1] > max) {
            max = giocate[i] [turno-1];
            idMax = i;
        }
    }

    for (int i = 0; i < numeroGiocatori; i++) {
        if(giocate[i] [turno-1] == max && i != idMax) {
            return;
        }
    }
}
```

```

        vincite[idMax]++;
    }

public synchronized int vincitore() {
    while(!gameOver) {
        try {
            wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    int max = 0;
    int idMax = 0;

    for (int i = 0; i < numeroGiocatori; i++) {
        if(vincite[i] > max) {
            max = vincite[i];
            idMax = i;
        }
    }

    for (int i = 0; i < numeroGiocatori; i++) {
        if(vincite[i] == max && idMax != i) {
            return -1;
        }
    }
    return idMax;
}
}

```

Classe Main:

```

public class Main{

    public static int NUMERO_GIOCATORI =3;

    public static int NUMERO_CARTE =10;

    public static void main(String[] args) {
        System.out.println("Inizio programma");
    }
}

```

```
Arbitro a = new Arbitro(NUMERO_GIOCATORI, NUMERO_CARTE);
Thread[] threads = new Thread[NUMERO_GIOCATORI];

for(int i = 0; i < NUMERO_GIOCATORI; i++) {
    threads[i] = new Thread(new Giocatore(i, a, NUMERO_CARTE));
    threads[i].start();
}

try {
    for(int i = 0; i < NUMERO_GIOCATORI; i++) {
        threads[i].join();
    }
} catch (InterruptedException e) {
}

System.out.println("Gioco terminato");
}
```

Capitolo 15

Liveness