

Sistemi Distribuiti - Lezioni

Sara Angeretti

@Sara1798

2022/2023

Indice

1 Introduzione ai S. D.	4
1.1 Alcune definizioni	4
1.1.1 Definizione 1	4
1.1.2 Definizione 2	5
1.1.3 In sintesi	5
1.2 Sistemi come collezioni di nodi autonomi	6
1.2.1 Comportamenti	6
1.2.2 Collezioni di nodi	6
1.3 Sistemi coerenti	6
1.3.1 Essenzialmente	6
1.3.2 Trasparenza di come caratteristica fondamentale dei sistemi distribuiti	7
1.4 Sintesi delle caratteristiche di un sistema distribuito	7
2 Architetture Software	9
2.1 Definizione di architetture software	9
2.2 Modello base: Architetture a strati (layered)	9
2.2.1 Definizione	9
2.3 Architetture a livelli (tier)	10
2.4 Architetture basate sugli oggetti	10
2.5 Architetture centrate sui dati	10
2.6 Architetture basate su eventi	10
2.6.1 Sistemi Operativi Distribuiti	10
3 Il modello Client-Server	15
3.1 Visto ieri: Sintesi caratteristiche di un s.d.	15
3.2 Il modello Client-Server	16
3.3 Caratteristiche problematiche di ogni sistema distribuito	16
3.4 Trasparenza di distribuzione	17
3.5 Le basi dei sistemi distribuiti	17
3.5.1 Il concetto di protocollo	17

3.5.2	Elementi minimi per creare un'applicazione	18
4	Stream-oriented communication - Le socket	20
4.1	Contenuti sintetici	20
4.2	Modello ISO/OSI	20
4.3	Comunicazione fisica - layering	20
4.4	Network edge	20
4.5	Processi e programmi	20
4.6	I servizi di trasporto Internet	21
4.7	Politiche dei servizi TCP/UDP	21
4.8	Socket: funzionamento di base	22
4.9	Aspetti critici	22
4.10	Identificare il server	23
4.11	Problemi fondamentali e TCP/IP	23
4.12	Comunicazione via socket	24
4.13	API socket system calls (Berkeley)	24
4.14	Riassunto: Politiche dei servizi TCP/UDP	25
4.15	Riassunto: Socket: funzionamento di base	25
4.16	Riassunto: Aspetti critici	26
4.17	Riassunto: Identificare il server	26
4.18	Riassunto: Problemi e TCP/IP	27
4.19	Riassunto: Comunicazione via socket	27
4.20	Riassunto: API socket system calls (Berkeley)	28
4.21	Processi e socket	29
4.22	Read e write	30
5	Le socket in Java	32
5.1	java.net.Socket	32
5.2	java.net.ServerSocket	33
5.2.1	Un esempio	34
5.3	Progettare un'applicazione con le socket	38
6	Architetture dei server	39
6.1	Tipi di server	39
6.2	Progettare un server iterativo	39
6.3	Progettare un server concorrente	40
6.4	Riassunto: Progettare un server iterativo	40
6.5	Riassunto: Progettare un server concorrente	41
6.5.1	I/O bloccante	42
6.5.2	Select System Call	43
6.5.3	Concurrent server structure	44

6.5.4	Sender Client	45
6.5.5	Concurrent Server	45
6.5.6	Concurrent Execution	46
6.6	Progettare un server multiprocesso	46
6.6.1	In C	47
6.6.2	Condivisione del canale	47
6.6.3	In Java	47
6.7	Confronto fra modelli	48
6.8	Conclusioni socket	49
7	Laboratorio1	50
7.1	Ripasso teoria	50
7.2	Esercizio 1	50
8	Introduzione alla concorrenza	51
8.1	Outline	51
8.2	Concorrenza come contemporaneità	51
8.3	Concorrenza e parallelismo	52
8.3.1	Tipi di parallelismo	52
8.4	Programmazione concorrente	53
9	Processi	54
9.1	Concetto di processo	54
9.2	Programmi e processi	54
9.3	Multiprogrammazione e multitasking	54
9.3.1	Multiprogrammazione	55
9.3.2	Multiprogrammazione e memoria	55
9.3.3	Multitasking	55
9.4	Op sui processi	55
9.4.1	Creazione di processi	56
9.4.2	Terminazione di processi	56
9.4.3	Es.: API Posix	57
10	Implementazione dei processi	58
10.1	I processi	58
10.1.1	Struttura	58
10.1.2	Immagine	58
10.1.3	Stato	59
10.2	Process Control Block (PCB)	59
10.3	Commutazione di contesto	59

<i>INDICE</i>	5
11 Multithreading in Java	60
11.1 Java threads	60
11.1.1 Il thread main	60
11.1.2 La classe principale per i Thread in Java	61
11.1.3 Far partire i thread: start()	62
11.2 Runnable interface	62
11.3 Thread: Alive o Terminated?	63
11.4 Stati di un thread	63
11.5 Operazioni sui thread	64
11.5.1 Cancellazione dei thread	64
11.6 Fork-join	64
12 Sincronizzazione	65
12.1 Programmi concorrenti e sequenziali	65
12.2 Meccanismi di sincronizzazione	65
12.2.1 Sincronizzazione su eventi	66
12.3 Problemi	69
12.3.1 Race condition	69
12.3.2 2	69
12.4 Come implementare la sincronizzazione	69
12.5 Come implementare la sincronizzazione	69
12.6 Come implementare la sincronizzazione	69
12.7 Come implementare la sincronizzazione	69
13 Variabili atomiche in Java	70
13.1 Il problema della visibilità	70
13.1.1 java.util.concurrent	71
13.2 Meccanismo di Locking	72
13.2.1 Problemi associati	72
13.2.2 Alternative al Locking: optimistic retrying	72
13.2.3 Strumento: Compare-and-Swap (CAS)	73
14 Esercizi	74
14.1 Es1	74
14.1.1 Specifica	74
14.1.2 Analisi	75
14.1.3 Implementazione	76
15 Liveness	82

16 Message-oriented communication	83
16.1 L'architettura del Web	83
16.1.1 Il browser	84
16.1.2 Web Page	84
16.1.3 Gli ipertesti	85
16.1.4 URL(Uniform Resource Locator)	85
16.1.5 I linguaggi del Web	86
16.2 Protocollo HTTP	86
16.2.1 Il concetto di protocollo	86
16.2.2 Il Web: protocollo http	87
16.2.3 Formato dei messaggi http	88
16.2.4 Metodi e applicazioni web	88
16.2.5 Metodi HTTP	90
16.2.6 Formato dei messaggi http	90
17 Introduzione ai sistemi Web	91
17.1 Applicazioni Web	91
17.1.1 Caratteristiche principali di un'applicazione Web	92
17.1.2 Dal Web alle Applicazioni Web	93
17.2 Client Side - HTML	97
17.2.1 Richieste	97
17.3 Server Side - Java Servlet	97
17.3.1 Java Servlets	97
17.3.2 Interfaccia Servlet	99
17.3.3 Classi astratte	99
17.3.4 La classe HttpServlet	100
17.3.5 Richieste e risposte	100
17.3.6 Esempio con POST: animale preferito	101
17.3.7 Ciclo di vita di una servlet	104
17.3.8 Servlet e Thread	104
17.3.9 Terminazione	105
17.3.10 Realizzazione di applicazioni	105
17.4 SERVER SIDE - JSP	107
17.4.1 Java Server Pages	107
17.4.2 Ciclo di vita delle applicazioni JSP	108
17.4.3 JSP: esempio	108
17.4.4 Gli elementi di una JSP	109
17.4.5 Oggetti e loro "scope"	110
17.4.6 JavaBeans	111
17.4.7 Sessioni	113
17.4.8 Altro esempio: uso di dati dinamici	114

17.4.9 Es. JSP con Form HTML	115
17.5 Il pattern MVC - Model View Control	117
17.5.1 Il pattern MVC	117
17.5.2 Il design pattern MVC	117
17.5.3 Vantaggi e svantaggi	118
17.5.4 Il pattern Model 1	118
18 Services Computing	120
18.1 IoT	120
18.1.1 IoT e servizi	120
18.2 Services Computing: a change of perspective	121
18.3 Service Oriented Architecture - SOA	121
18.4 But... what is a “(Web) Service”?	124
18.5 Architectural building blocks	124
18.6 SOA core elements	125
18.7 Service Level Agreement (SLA)	125
18.8 Peculiarità dei Web Services	126
18.8.1 Challenges: how to do...	126
18.9 Composizione di servizi	126
18.10 Business Processes	127
18.10.1 Software side: Orchestration vs Choreography	127
18.11 Architetture orchestrate	128
18.12 Enterprise Service Bus	128
19 Soap Services	130
19.1 The Conceptual Web Service Stack	130
19.2 SOAP: Simple Object Access Protocol	130
19.2.1 Componenti di un messaggio SOAP	131
19.2.2 SOAP with HTTP	132
19.3 WSDL	132
19.3.1 What is WSDL?	133
19.3.2 WSDL 2.0: conceptual model	133
19.3.3 WSDL 2.0: definizione di binding	133
19.4 Perché non si usa più - Why SOAP dream faded away	134
20 Services Computing - REST	135
20.1 Web Services Evolution	135
20.2 REST vs SOAP	136
20.3 Composition = Establishing a Common Model	136
20.4 Principi REST	137
20.5 Representational State Transfer	137

20.6 Caching	138
20.6.1 Local caching	138
20.7 Alcune osservazioni	139
20.8 Using HTTP to build REST Applications	139
20.8.1 Defining the formats	140
20.8.2 Pick the operations	140
20.9 Operazioni HTTP	141
20.10 REST come implementazione CRUD	141
20.10.1 Response Status Code	141
20.11 PUT vs. POST	142

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

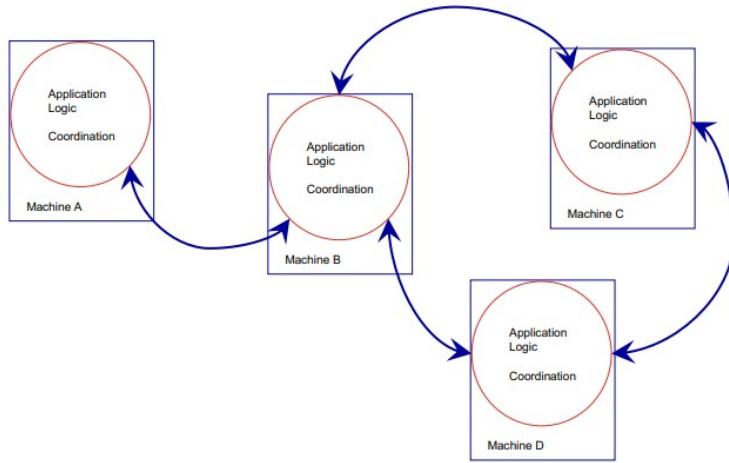
1.1 Alcune definizioni

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.

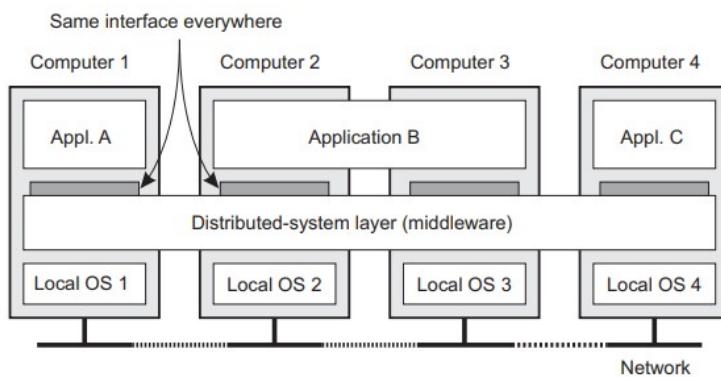


1.1.2 Definizione 2

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

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

1.4. SINTESI DELLE CARATTERISTICHE DI UN SISTEMA DISTRIBUITO

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

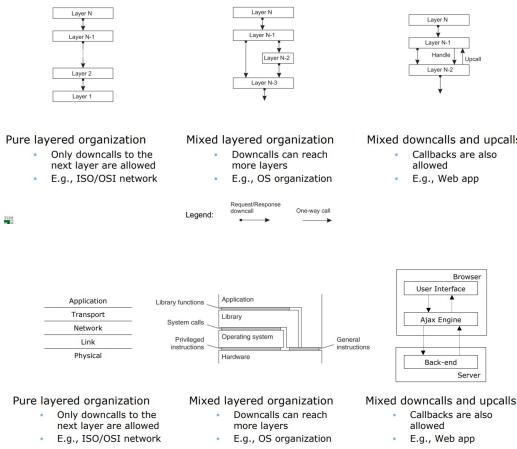
2.2.1 Definizione

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

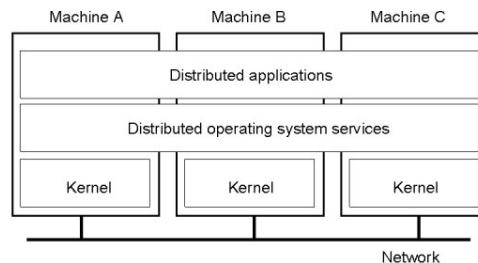
Diversi tipi:

- 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

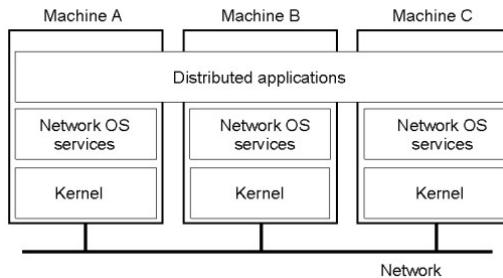
Computation Migration

- 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 form a distributed application
- Services, such as process migration, are handled by applications

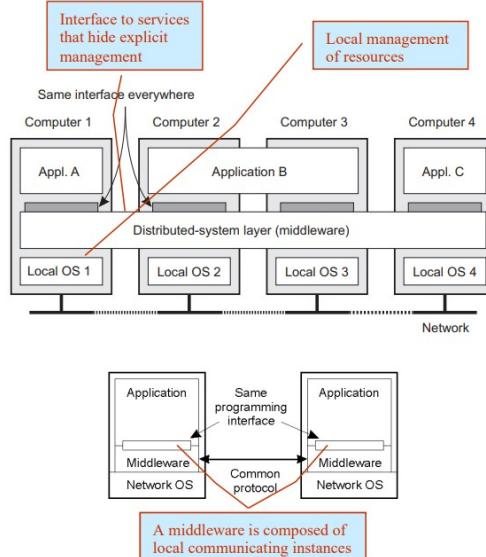
Access to resources of various machines is done explicitly by:

- 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., MacOSX, Windows10, 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)

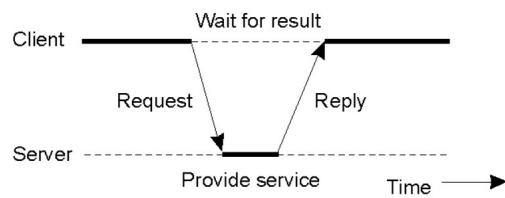
- ... defines and offers models to protect access to data and services (with different levels of permissions) and computation integrity

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 = $_i$ esecuzione concorrente
- Il coordinamento delle attività è importante per definire il comportamento di un sistema/applicazione costituita da più componenti

Gestione del tempo (temporizzazione)?

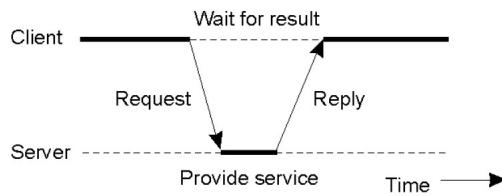
- Non c'è un clock globale
- Non c'è possibilità di controllo/scheduling globale
- Solo coordinamento via scambio messaggi

Tipi di fallimenti?

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

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

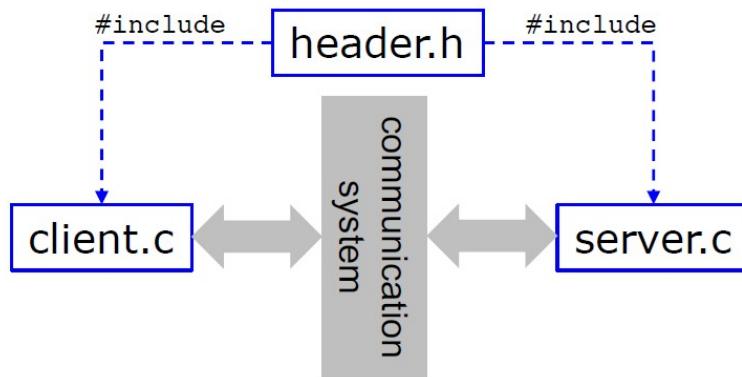
- HTTP - HyperText Transfer Protocol
- FTP - File Transfer Protocol
- SMTP - Simple Mail Transfer Protocol

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];
};

  
```

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.     while(TRUE) {
6.         receive(FILE_SERVER, &m1);    /* server runs forever */
7.         switch(m1.opcode) {          /* block waiting for a message */
8.             case CREATE: r = do_create(&m1, &m2); break;      /* dispatch on type of request */
9.             case READ: r = do_read(&m1, &m2); break;
10.            case WRITE: r = do_write(&m1, &m2); break;
11.            case DELETE: r = do_delete(&m1, &m2); break;
12.            default: r = E_BAD_OPCODE;
13.        }
14.        m2.result = r;                /* return result to client */
15.        send(m1.source, &m2);        /* send reply */
16.    }
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.     do {
9.         m1.opcode = READ;                   /* operation is a read */
10.        m1.offset = position;             /* current position in the file */
11.        m1.count = BUF_SIZE;              /* how many bytes to read */
12.        strcpy(m1.name, src);            /* copy name of file to be read to message */
13.        send(FILE_SERVER, &m1);          /* send the message to the file server */
14.        receive(client, &m1);            /* block waiting for the reply */

15.        /* Write the data just received to the destination file */
16.        m1.opcode = WRITE;               /* operation is a write */
17.        m1.offset = position;             /* current position in the file */
18.        m1.count = m1.result;             /* how many bytes to write */
19.        strcpy(m1.name, dst);            /* copy name of file to be written to buf */
20.        send(FILE_SERVER, &m1);          /* send the message to the file server */
21.        receive(client, &m1);            /* block waiting for the reply */
22.        position += m1.result;          /* m1.result is number of bytes written */
23.    } while(m1.result > 0);            /* iterate until done */

24.    return(m1.result >= 0 ? OK : m1.result); /* return OK or error code */
25. }
```

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)

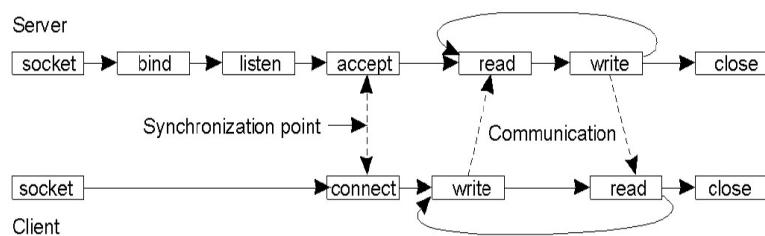
IMPORTANTE: TCP si può usare per qualsiasi tipo di comunicazione? Perché non c'è semantica. Quindi il punto di Comunicazione 2 non c'è. Questa potrebbe essere una domanda dell'esame.

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 $k < N$)



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

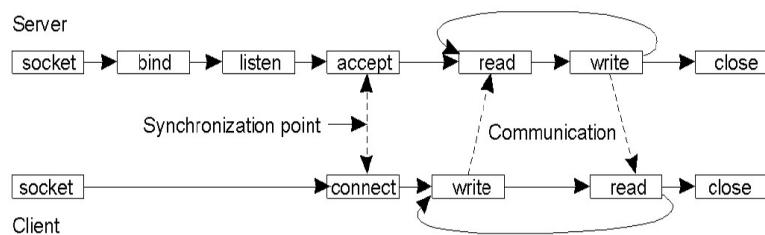
- conoscere l'indirizzo di rete
- fare il parsing di bytes per accedere al contenuto (messaggio)

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) –*i* 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.

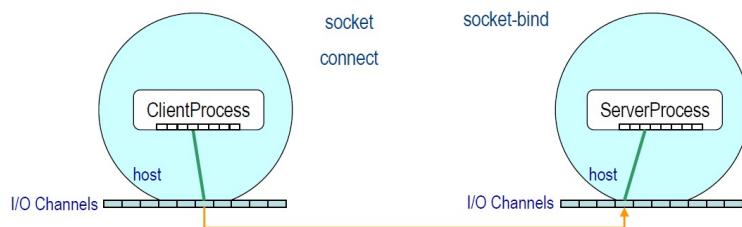
Le più importanti nella tabella seguente:

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.

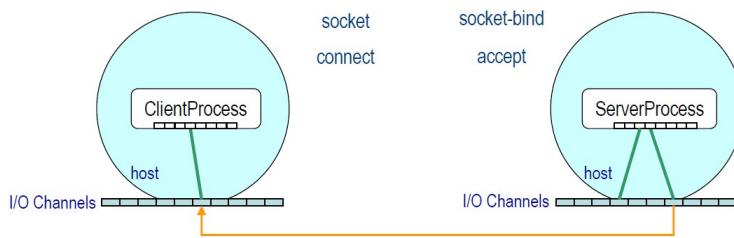
Con la accept(), il server crea una nuova socket, cioè un nuovo canale, dedicato alla comunicazione con il client.



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*.

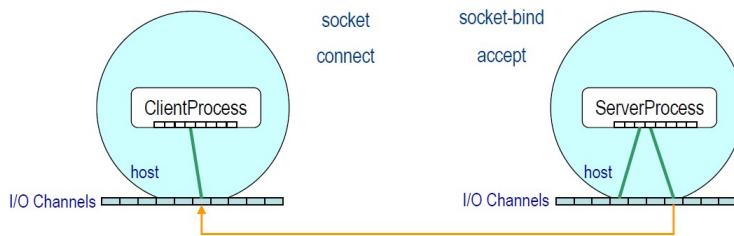
Si genera una seconda domanda: Perché non si usa la stessa socket?



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:

```

Problems @ Javadoc Declaration Console
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
<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.

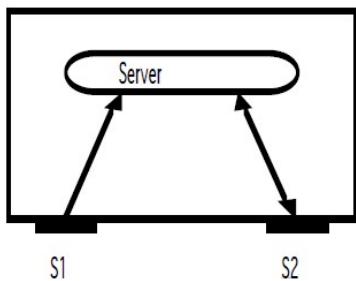
Vantaggi:

- 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

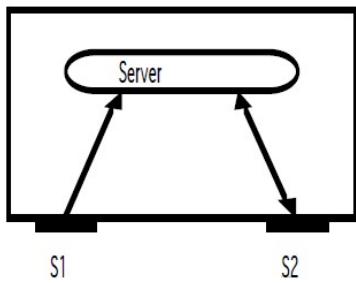
Manca una slide

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.

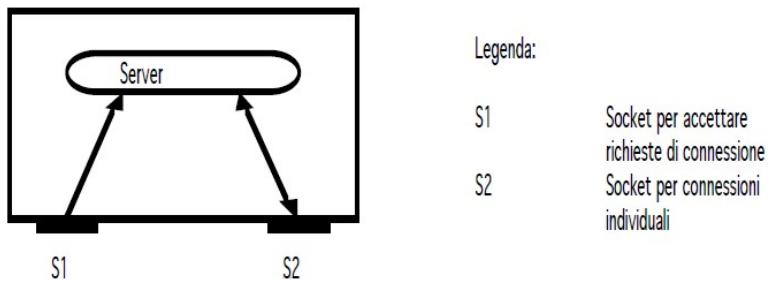
Vantaggi:

- 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.

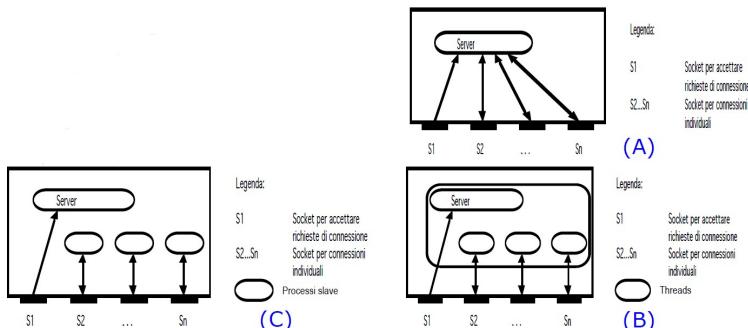


6.5 Riassunto: Progettare un server concorrente

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.

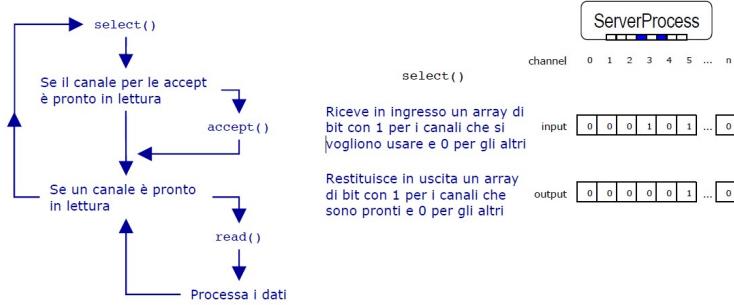
Il codice del server diventa:

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 descrittori 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 indefinitivamente 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/i-nio/>

Java Selector

6.5.3 Concurrent server structure

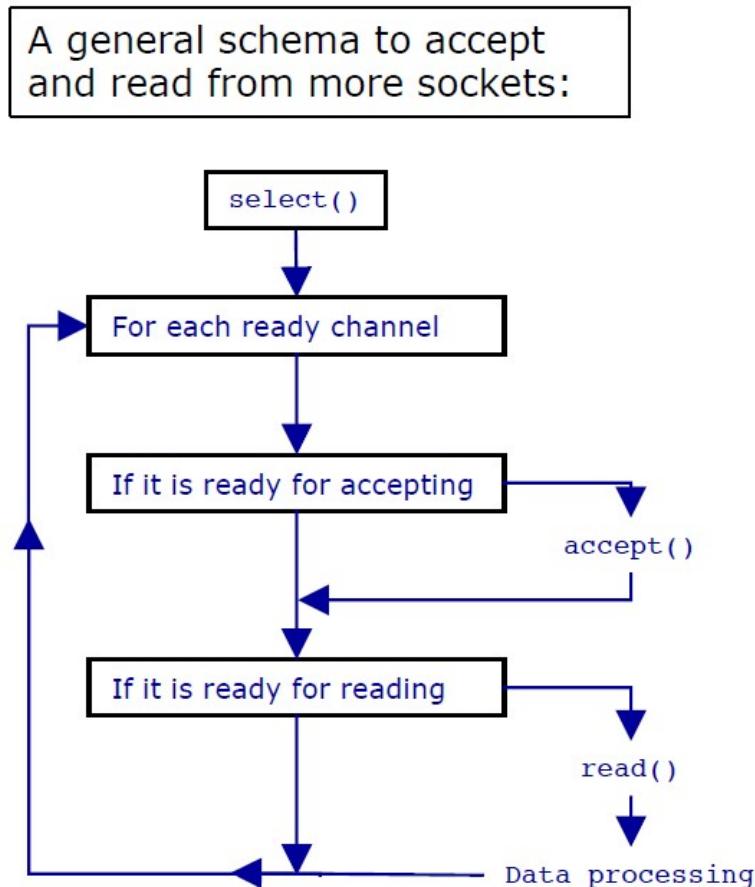
Codice + schema generale per accettare read da più sockets.

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()***.

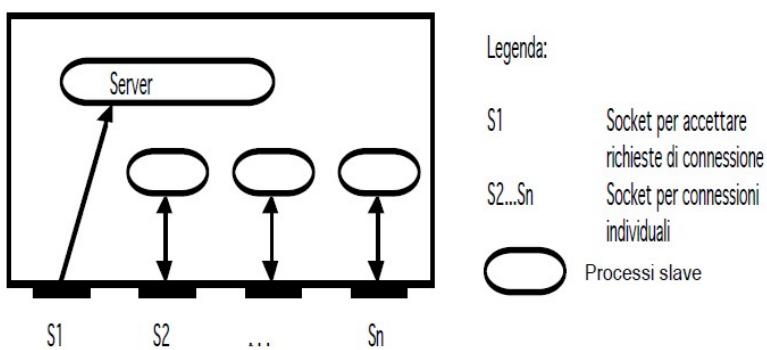
La ***fork()*** crea un processo clone del padre che

- 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.

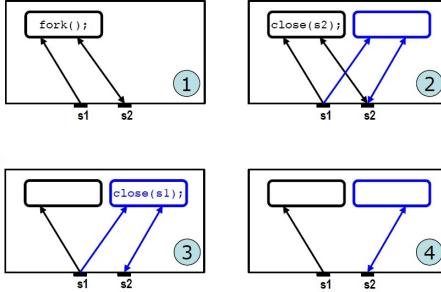


6.6.1 In C

```

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. }

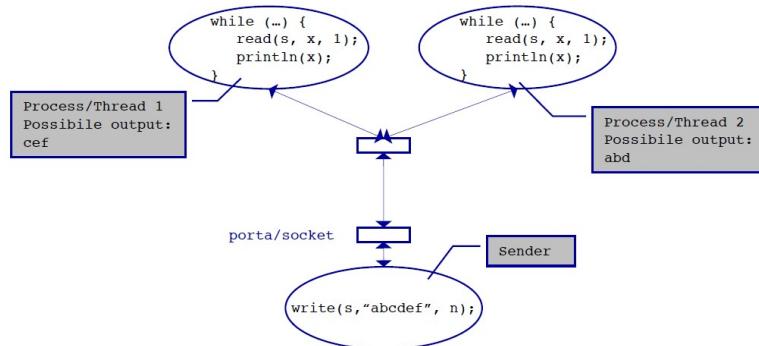
```



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

```

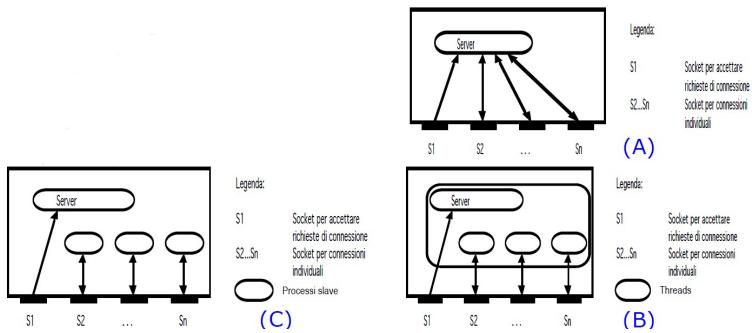
1. ProcessBuilder pb = new ProcessBuilder("C:\\Windows\\\\system32\\\\program.exe");
2. pb.inheritIO(); // <-- passes IO from forked process
3. try {
4.     Process p = pb.start(); // <-- forkAndExec on Unix
5.     p.waitFor(); // <-- waits for the forked process to complete
6. } catch (Exception e)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

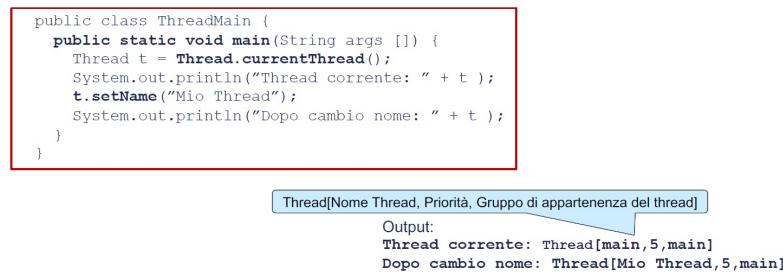
Modalità principali di creazione dei thread in Java:

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

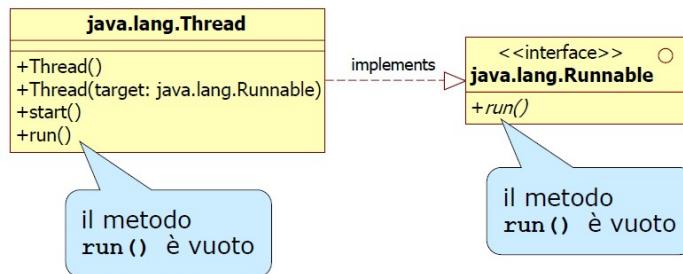
11.1.1 Il thread main

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()`



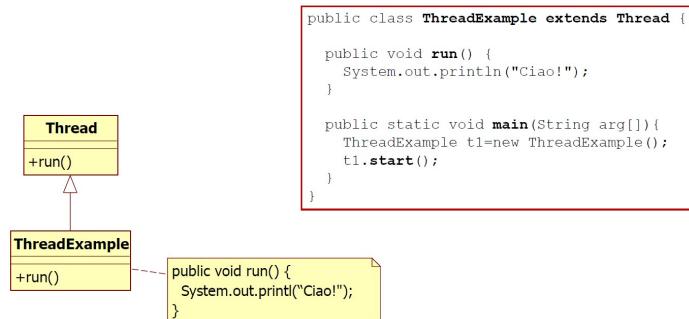
11.1.2 La classe principale per i Thread in Java



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. 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:

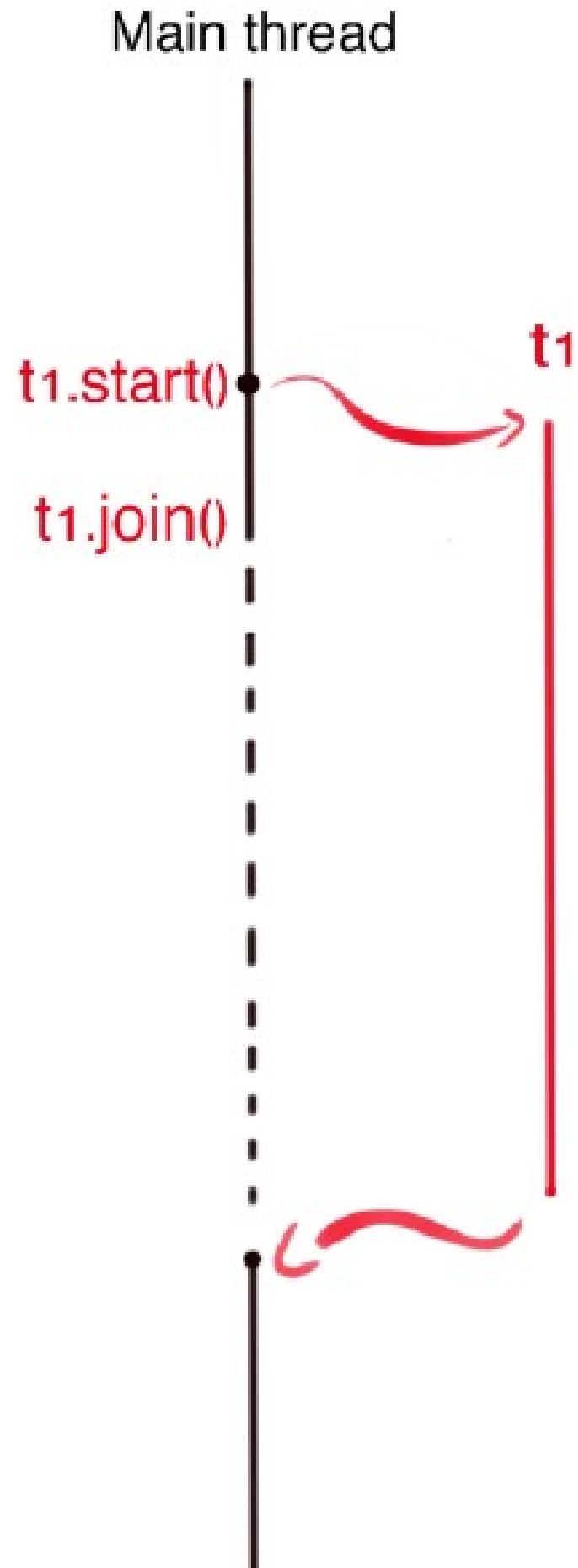
- di solito impliciti nelle primitive di *send* e *receive* == un thread può ricevere un messaggio solo dopo il suo invio
- **sincronizzazione su eventi:** si sospende l'esecuzione di un thread fino al verificarsi di un evento

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

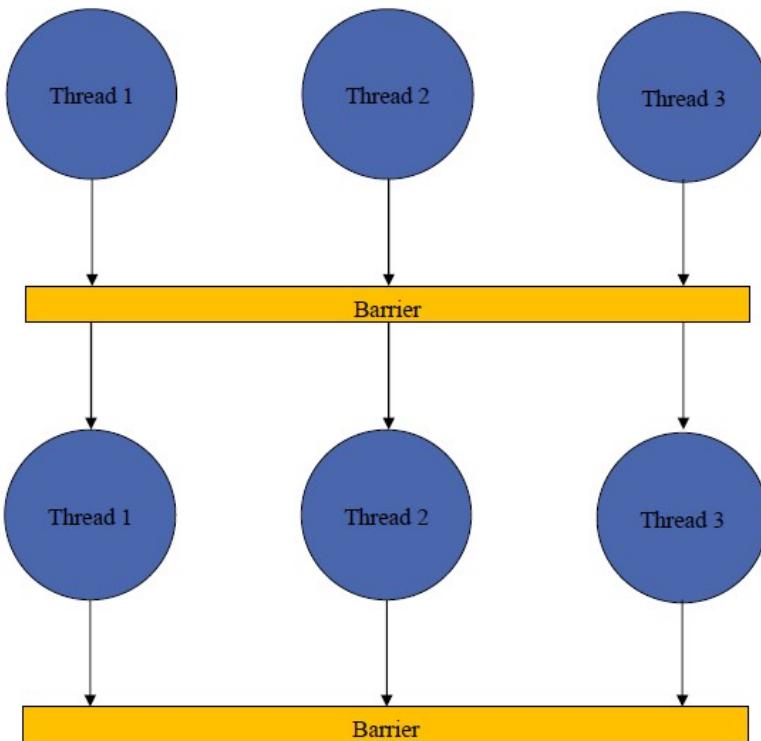
- I thread vengono messi progressivamente in attesa in attesa che una condizione globale non venga soddisfatta.

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:

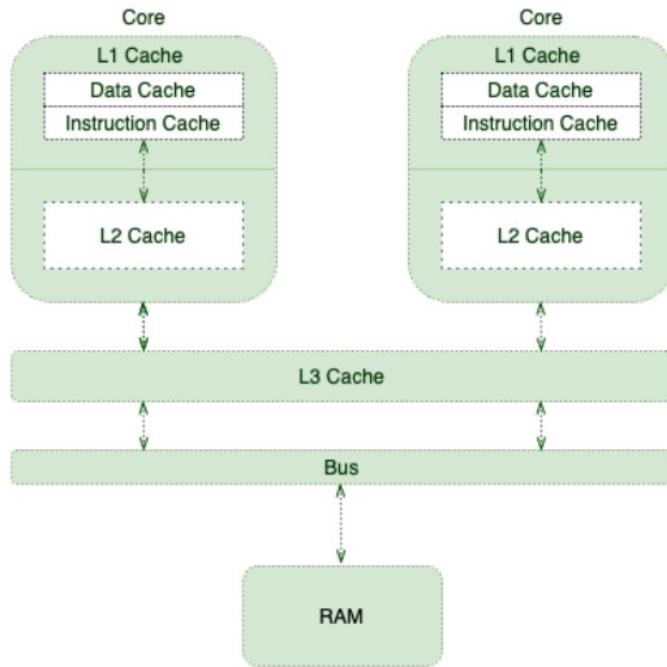
Mutua esclusione: solo un thread esegue una sezione critica alla volta.

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).

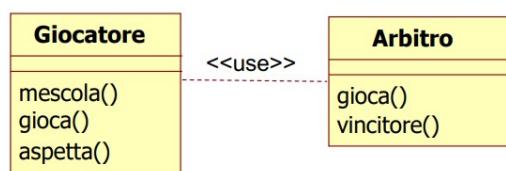
Il gioco è diviso in turni.

- 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 e
        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

Capitolo 16

Message-oriented communication

Cosa vedremo in questa sezione

Message-oriented communication:

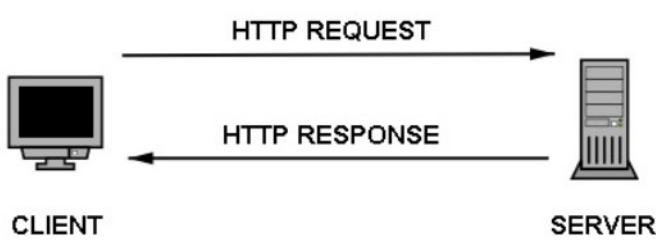
- The Web and HTTP messages
- Messages communication vs stream communication

Communication types

- Synchronous and asynchronous communication
- Persistent and volatile communication
- Queue-based communication

16.1 L'architettura del Web

Il Web supporta l'interazione tra client e server via HTTP.



Il client è realizzato da un “Browser” o “User-Agent”

Il server è realizzato da un “Web Server” o “HTTP Server”

16.1.1 Il browser

Il browser è l'applicazione per il Web sul lato del client.

Un web browser (detto User-Agent) è un programma che consente la navigazione nel Web da parte di un utente.

La funzione primaria di un browser è quella di interpretare il codice con cui sono espresse le informazioni (pagine web) e visualizzarlo (operazione di rendering) in forma di ipertesto.

- I browser moderni hanno anche funzioni più avanzate per
 - trattare altri tipi di dati: es. Multimedialità, RSS, XML, JSON ...
 - ospitare ed eseguire applicazioni: es. JavaScript (vedremo più avanti)
- Il rendering dipende dai dispositivi utilizzati, anche “non visuali”, ad esempio per supportare utenti non vedenti, es.:
 - sintesi vocale
 - alfabeto Braille
 - ...

16.1.2 Web Page

Una pagina web (web page, o anche documento) è costituita da diversi oggetti (risorse nella terminologia del web).

Una risorsa è un file, cioè una sequenza di dati (in formato digitale) residente in un computer, che è identificato da una URL (cioè un indirizzo univoco per la risorsa).

- Testo, immagini, musica, ...

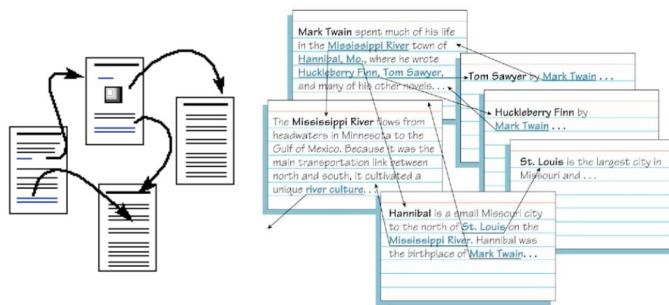
La maggior parte delle pagine web sono costituite da un file HTML che definisce la struttura e i contenuti della pagina, testuali più altri oggetti.

- HTML: HyperText Markup Language, è il linguaggio con cui si scrivono gli ipertesti (si definisce la struttura di una pagina, i collegamenti, si inseriscono immagini, ...)

Un Web Server è una applicazione che si occupa di gestire le risorse (file) su un computer e di renderle disponibili ai client.

16.1.3 Gli ipertesti

Un ipertesto (hypertext) è un insieme di testi o pagine leggibili con l'ausilio di un'interfaccia elettronica, in maniera non sequenziale, tramite hyperlink (o più semplicemente link, cioè collegamenti), che costituiscono un rete raggiata o variamente incrociata di informazioni organizzate secondo criteri paritetici o gerarchici (es. menu).



16.1.4 URL(Uniform Resource Locator)

Identifica un oggetto nella rete e specifica come interpretare i dati ricevuti attraverso il protocollo.

Ha cinque componenti principali:

1. nome del protocollo
2. indirizzo dell'host
3. porta del processo (la controparte)
4. percorso nell'host
5. Identificatore della risorsa

```
protocollo://indirizzo_IP[:porta]/cammino/risorsa
      1.           2.           3.           4.           5.
  ftp://www.adobe.com/download/acroread.exe
  http://www.biblio.unimib.it/go/Home/Home-English/Services
  http://www.biblio.unimib.it/link/page.jsp?id=47502837
  http://www.someSchool.edu/someDept/pic.gif
  http://www.someSchool.edu:80/someDept/pic.gif
```

16.1.5 I linguaggi del Web

I dati testuali sono espressi in linguaggi standard:

- HTML (per definire la struttura dei contenuti e la loro impaginazione)
- può contenere CSS (per gestire la presentazione, cioè il rendering), ...
- XML (focalizzato sui dati e la loro struttura)
- XSL, RDF, ...
- JSON (focalizzato sui dati e la loro struttura)

I dati possono essere non testuali (immagini, audio, video)

- Encoding MIME (definisce il formato dei contenuti)

La pagine Web possono contenere del codice espresso in linguaggi di scripting per arricchire l'interazione e rendere le pagine attive

- JavaScript, VBScript, Java/Applet, Adobe Flash ...

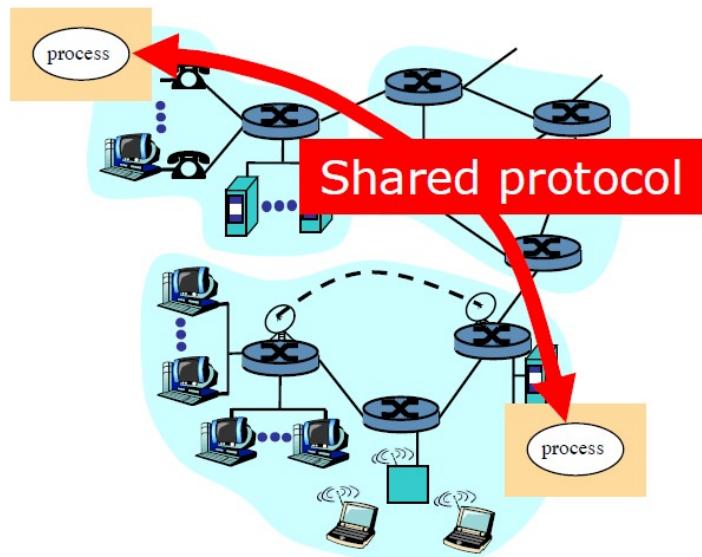
16.2 Protocollo HTTP

16.2.1 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. Esempi di protocollo

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



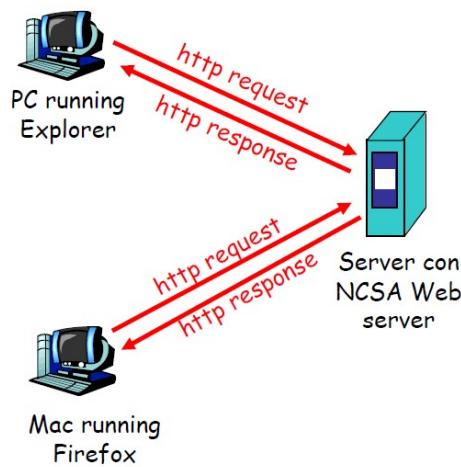
16.2.2 Il Web: protocollo http

http: hypertext transfer protocol

Protocollo di livello applicativo per il Web

Usa il modello client/server

- client: browser che richiede, riceve e “mostra” oggetti Web
- server: Web server che invia oggetti in risposta alle richieste



http1.0: RFC 1945

http1.1: RFC 2668

http usa TCP

Il client inizia una connessione TCP (crea una socket) verso il server sulla porta 80.

Il server accetta la connessione TCP dal client

Vengono scambiati messaggi http (messaggi del protocollo di livello applicativo) tra il browser (client http) e il Web server (server http).

http è "stateless"

Il server non mantiene informazione sulle richieste precedenti del client.

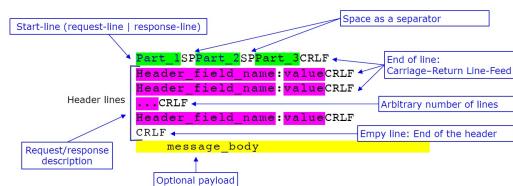
Quindi: ogni richiesta deve contenere tutte le informazioni necessarie per la sua esecuzione.

I protocolli che mantengono informazione di stato sono complessi (es. TCP)!

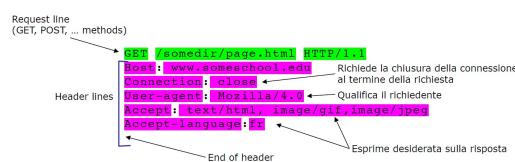
16.2.3 Formato dei messaggi http

Due tipi di messaggi http: ***request***, ***response***.

- ASCII (formato testo leggibile)
- Hanno la stessa struttura!



Messaggio http request



16.2.4 Metodi e applicazioni web

I principali metodi utilizzati sono:

Metodo GET

- Restituisce una rappresentazione di una risorsa
- Include eventuali parametri in coda alla URL della risorsa (vedi più avanti)
- È safe: l'esecuzione non ha effetti sul server =; la risposta può essere gestita con una cache dal client
- Uso tipico: ottenere dati in formato di pagine html e immagini, dati XML o JSON

La GET include eventuali parametri in coda alla URL della risorsa

```
GET resource[?key=value{&key=value}] HTTP1.1
{header lines}
```

Esempio: dammi tutti gli ordini di marzo per il prodotto con codice Q2345

```
/myCompany/orders?item="Q2345"&date="2022/03"
```

Metodo POST

- Comunica dei dati da elaborare lato server o crea una nuova risorsa subordinata all'URL indicata (vedi più avanti)
- L'input segue come documento autonomo (body)
- Non è idempotente: ogni esecuzione ha un diverso effetto =; La risposta NON può essere gestita con una cache dal client
- Uso tipico: processare FORM e modificare dati in un DB

La POST prevede l'input in coda come documento autonomo (body)

```
POST resource HTTP1.1
{header lines}
Body
```

Esempio: aggiorna i codici di un prodotto in tutti gli ordini di aprile

```
POST /myCompany/orders HTTP1.1
{header lines}
update=true&oldItem="Q2345"&newItem="Q68254"&date="2022/04"
```

Metodo HEAD

- Simile al metodo GET ma viene restituito solo l'Head della pagina Web
- Spesso usato in fase di debugging

16.2.5 Metodi HTTP

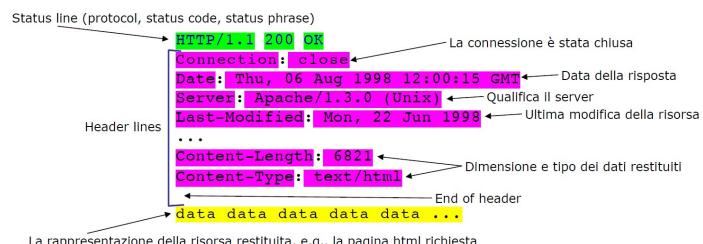
		cache	safe	idempotent
OPTIONS	represents a request for information about the communication options available on the request/response chain identified by the Request-URI			✓
GET	means retrieve whatever information (in the form of an entity) is identified by the Request-URI	✓	✓	
HEAD	identical to GET except that the server MUST NOT return a message-body in the response	✓	✓	
POST	is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line			
PUT	requests that the enclosed entity be stored under the supplied Request-URI			✓
DELETE	requests that the origin server delete the resource identified by the Request-URI			✓
TRACE	is used to invoke a remote, application-layer loop-back of the request message			✓

Safe = methods SHOULD NOT have the significance of taking an action other than retrieval.

Idempotent = the side-effects of $N \geq 0$ identical requests is the same as for a single request (aside from error or expiration issues).

16.2.6 Formato dei messaggi http

Messaggio http response



HTTP 1.0: Server chiude connessione al termine della richiesta

HTTP 1.1: mantiene aperta la connessione oppure chiude se la richiesta contiene Connection: close

Domanda:

Perché sono state inserite le informazioni sulla dimensione e il tipo dei dati restituiti?

```
Content-Length: 6812
Content-Type: text/html
```

Capitolo 17

Introduzione ai sistemi Web

Cosa vedremo nelle prossime lezioni:
Applicazioni Web (lezione di oggi e giovedì)

- Web come piattaforma
- Architettura di un'applicazione Web

Client side: HTML

Server side: Servlet

- Il modello
- HTTP Servlet

Server side: JSP

- Struttura di una JSP
- Gli elementi che compongono una JSP
- JSP e JavaBeans

Il pattern MVC

17.1 Applicazioni Web

Perché il Web?

Basato su Internet (non è la stessa cosa!)

- Ambiente standard (alla base c'è TCP/IP)

- Ampia diffusione

Semplicità e uniformità dell’interazione

- Interfaccia grafica (Browsers)
- Interazione attraverso un protocollo standard (HTTP)
- Definizione di una API - Application Programming Interface (applicazioni REST, vedremo meglio)

Infrastruttura completa

- Supporta sistemi aperti (connessione dinamica di nuovi componenti)
- Strumenti sempre più completi e potenti: evoluzioni di HTML, Ajax, Web App...

17.1.1 Caratteristiche principali di un’applicazione Web

Applicazione Web = adozione del protocollo HTTP

Oggiorno è essenziale avere un’infarinatura di come funzioni il protocollo HTTP.

Caratteristiche del protocollo HTTP

Formato a caratteri (lento): occorre tradurre e ritradurre i dati (es. da testo a numeri: da “256” a 256)

Utilizzo del linguaggio HTML per input e output:

- Uso di FORM per l’acquisizione dati (invio dati al server, lato front-end)
- Uso di pagine HTML in risposta (dal server verso il client, lato back-end)
- Possibilità di pagine dinamiche (JavaScript con dati scambiati in JSON)

(A proposito di invio di dati, ci sono due modi: appunto i FORM, ma anche le QUERIES.)

Utilizzo di payload di tipo MIME (Multimedia Internet Mail Extensions)

- Permettono di generalizzare oltre HTML (xml, json, zip, jpeg)
-

Conversazioni (interazioni client-server) prive di stato (memoria)

- Ogni richiesta è un messaggio autonomo, indipendente dagli altri.
- Per creare sessioni di lavoro (legare più richieste tra loro) servono informazioni esplicite (cookies, campi nascosti).
- Un cookie HTTP (web cookie, browser cookie) è un piccolo blocco di dati che un server invia al browser web di un utente. Il browser può memorizzare il cookie e rinviarlo allo stesso server con richieste successive. In genere, un cookie viene utilizzato per stabilire se due richieste provengono dallo stesso browser, ad esempio per mantenere un utente loggato.
- È impossibile pensare di creare un applicativo privo di stato, pensiamo solo alla sessione di attività dell'utente.

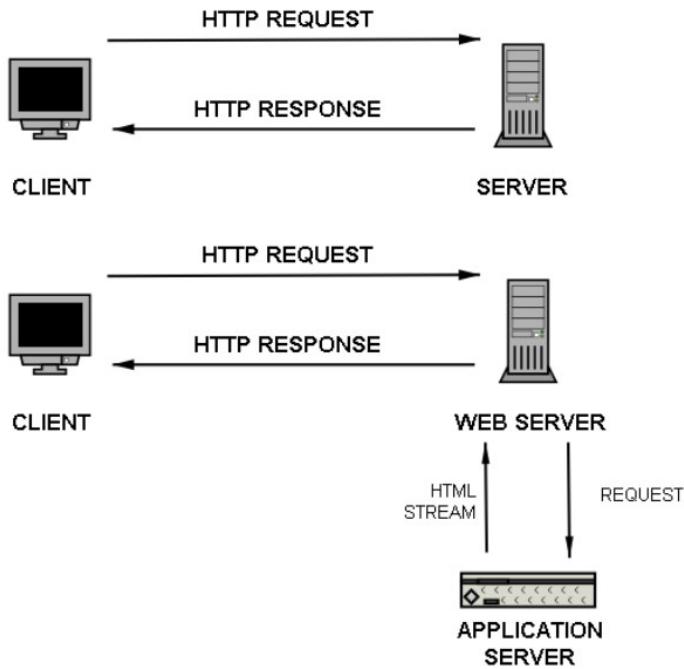
17.1.2 Dal Web alle Applicazioni Web

Il Web supporta l'interazione tra client e server via HTTP

Ben presto il web si è evoluto andando al di là del trasferimento di pagine web (file HTML). È diventato uno strumento per fornire accesso ad applicazioni remote

Per eseguire una applicazione, il Web Server utilizza un Application Server
Un Application server è caratterizzato dal protocollo di interazione con il Web Server

L'interazione con il client avviane sempre tramite HTTP



Tecnologia Server Side

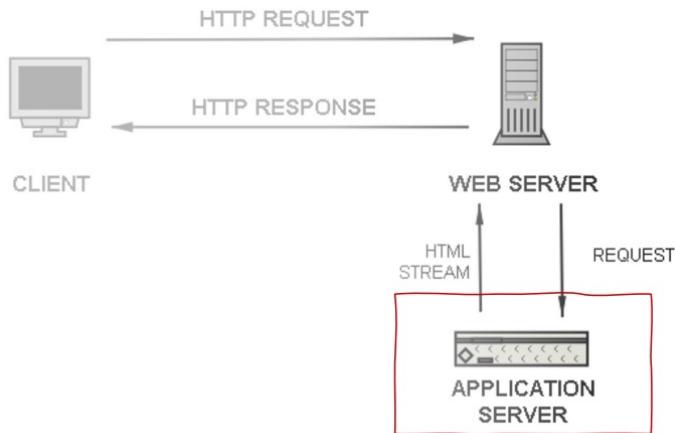
La computazione avviene lato server e può avvenire tramite *programmi compilati* o *script (interpretati)*.

Nel caso di programmi compilati il Web Server si limita ad invocare, su richiesta del client, un eseguibile.

- L'eseguibile può essere scritto in un qualsiasi linguaggio che supporti l'interazione con il Web Server
- Tra i linguaggi più diffusi C#, C++.

Nel caso di esecuzione di script, il Web Server ha al suo interno un motore (engine) in grado di interpretare il linguaggio di scripting usato.

- Si perde in velocità di esecuzione (parliamo di latenza, più che di velocità), ma si guadagna in facilità di scrittura dei programmi.
- Tra i linguaggi più diffusi Java, PHP, Python e Perl, più recentemente Nodejs.



Architettura di un'applicazione compilata

Uniform Resource Locator (URL)

- definisce un naming globale

HyperText Transfer Protocol (HTTP)

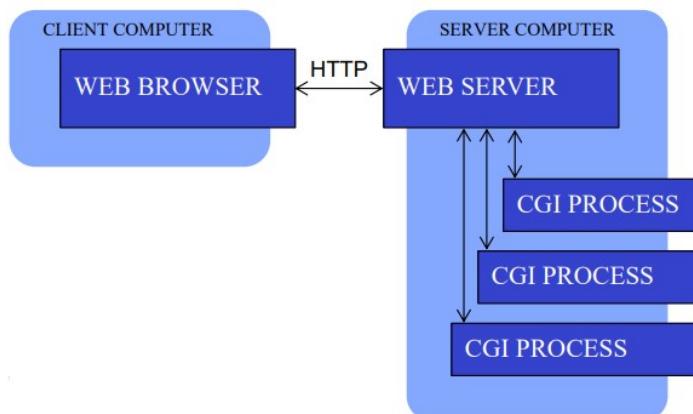
- permette di invocare i programmi sul server come fossero risorse

Questa parte è valida anche per applicazioni interpretate.

Common Gateway Interface (CGI). Protocollo che permette al server di:

- attivare un programma (crea un processo)
- passargli le richieste e i parametri provenienti dal client
- recuperare la risposta

Ogni applicazione CGI deve quindi implementare l'interprete del protocollo.



Quando un client richiede al server un URL corrispondente a un documento HTML, il server restituisce il documento stesso come un file di testo. Ciò significa che il documento viene generato una volta per tutte e poi viene inviato al client senza ulteriori elaborazioni. Quando l'URL richiesto corrisponde a un'applicazione CGI, il server esegue il programma in tempo reale, generando dinamicamente informazioni per l'utente.

Architettura di un'applicazione interpretata

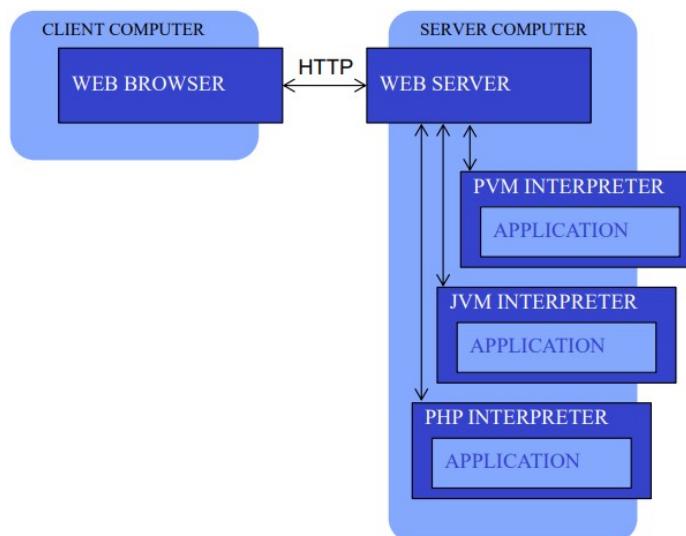
Il protocollo CGI viene gestito dall'interprete per il linguaggio usato (le applicazioni NON devono più gestire il protocollo CGI).

Vantaggi:

- Serve programmare solo le logiche delle applicazioni
- Modello delle applicazioni conformi al modello del linguaggio utilizzato
- Semplicità, portabilità, manutenibilità

Esempi

- Python
- Java/Servlet
- PHP
- Nodejs



17.2 Client Side - HTML

17.2.1 Richieste

Basate su Link

Un link in un documento HTML può essere usato per puntare ad una risorsa remota:

```
<p>Click the link to ask the servlet to send back an HTML document</p>
<a href="http://localhost:8080/SlideServlet/GetHTTPServlet">
    Get HTML Document
</a>
```

Il browser invia richieste del tipo:

```
GET /SlideServlet/GetHTTPServlet HTTP/1.1
```

Per mezzo di FORM

Anche il parametro action di un Form può essere usato per puntare ad una risorsa remota (applicazione)

```
<form action="http://localhost:8080/SlideServlet/GetHTTPServlet" method="get">
    <p>Click the button to have the servlet send an HTML document</p>
    <input type="submit" value="Get HTML Document">
</form>
```

Il browser invia richieste del tipo:

```
GET /SlideServlet/GetHTTPServlet HTTP/1.1
```

17.3 Server Side - Java Servlet

17.3.1 Java Servlets

Estensione della classe Servlet che risiede su un application server che la può gestire.

Ma cosa sono?

Sono piccole applicazioni Java residenti sul server (esempio: Apache Tomcat). Una servlet è un componente gestito in modo automatico da un container o engine.

Deve implementare un'interfaccia prestabilita che definisce il set di metodi:

- **Vantaggi:** semplicità e standardizzazione
- **Svantaggi:** rigidità del modello

Il container controlla le servlet (le attiva/disattiva) in base alle richieste dei client. Questo è possibile in maniera automatica perché le servlet implementano una interfaccia nota al server.

Sono oggetti Java residenti in memoria, quindi

- mantengono uno stato
- consentono l'interazione con altre servlet

Stateless vs. stateful

HTTP non prevede persistenza (*stateless*).

- Non si possono mantenere informazioni tra un messaggio e i successivi.
- Non si possono identificare i client.

Mantenere lo stato della conversazione è compito dell'applicazione (se vuole).

Gli strumenti:

Cookies: - Informazioni memorizzate a livello di client;
- Permettono di gestire sessioni di lavoro.

HttpSession: - Oggetto gestito automaticamente dal container/engine (con cookie o riscrittura delle URL);
- Le servlet possono accedervi per immagazzinare informazioni.
Da considerare come una mappa chiave-valore.

Ovviamente i dati possono essere memorizzati anche in un database. Quale sarebbe il vantaggio di un HttpSession invece di un Database?

È uno strumento già pronto, poi le informazioni lì immagazzinate sono **volutamente volatili** e in alcune situazioni torna utile.

Es.: prenotazione del biglietto di un treno. Se l'utente (per cui il biglietto in fase di acquisto è prenotato tra il momento della scelta del biglietto e quello dell'acquisto) non compra in diciamo 15 minuti, l'informazione dell'acquisto è persa e il biglietto torna disponibile per la vendita per altri clienti.

Es.: invece il carrello di Amazon non lo è, volatile. Amazon si ricorda della scelta dell'utente e gli ripropone l'articolo anche in tempi futuri.

17.3.2 Interfaccia Servlet

Ogni servlet implementa l'interfaccia jakarta.servlet.Servlet, con 5 metodi

```
void init(ServletConfig config)
```

Inizializza la servlet, viene invocato dopo la creazione della stessa

```
void destroy()
```

Chiamata quando la servlet termina (es: per chiudere un file o una connessione con un database)

```
void service(ServletRequest request, ServletResponse response)
```

Invocato per gestire le richieste dei client

```
ServletConfig getServletConfig()
```

Restituisce i parametri di inizializzazione e il ServletContext che da accesso all'ambiente

```
String getServletInfo()
```

Restituisce informazioni tipo autore e versione

17.3.3 Classi astratte

L'interfaccia è solo la dichiarazione dei metodi che, per essere utilizzabili, devono essere implementati in una classe.

Sono presenti due classi astratte, cioè che implementano i metodi dell'interfaccia in modo che non facciano nulla.

```
jakarta.servlet.GenericServlet
```

Definisce metodi indipendenti dal protocollo

```
jakarta.servlet.http.HttpServlet
```

Definisce metodi per l'uso in ambiente web.

Questo semplifica l'implementazione delle servlet vere e proprie in quanto basta implementare (ridefinendoli) solo i metodi che interessano.

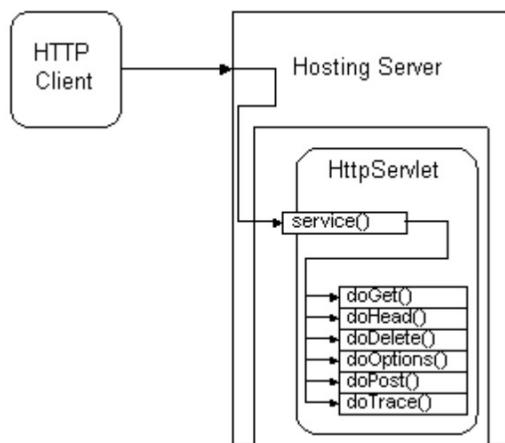
17.3.4 La classe HttpServlet

Implementa service() in modo da invocare i metodi per servire le richieste dal web.

Metodi doX X è un metodo HTTP (doGet, doPost, ...) doX è dedicato alle richieste di tipo X

Parametri: HttpServletRequest
HttpServletResponse

Eccezioni ServletException
IOException



17.3.5 Richieste e risposte

Anche i parametri sono stati adattati al protocollo HTTP, cioè consentono di ricevere (inviare) messaggi HTTP leggendo (scrivendo) i dati nell'header e nel body di un messaggio.

Interfaccia HttpServletRequest

- Viene passato un oggetto da service
- Contiene la richiesta del client
- Estende ServletRequest

Interfaccia HttpServletResponse

- Viene passato un oggetto da service

- Contiene la risposta destinata al client
- Estende ServletResponse

I metodi principali per le **richieste**:

String getParameter(String name) - Restituisce il valore del query parameter dato il nome (valore singolo)

Enumeration getParameterNames() - Restituisce l'elenco dei nomi degli argomenti

String[getParametersValues(String name)] - Restituisce i valori dell'argomento name (valore multiplo)

I metodi principali per le **risposte**:

void setContentType(String type) - Specifica il tipo MIME della risposta per dire al browser come visualizzare la risposta Es: “text/html” dice che è html

ServletOutputStream getOutputStream() - Restituisce lo stream di byte per scrivere la risposta

PrintWriter getWriter() - Restituisce lo stream di caratteri per scrivere la risposta

Altri metodi

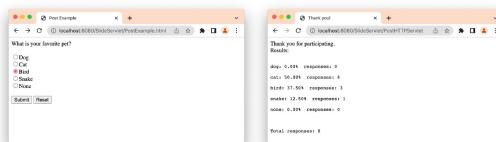
Cookie[getCookies()] - Restituisce i cookies del server sul client

void addCookie(Cookie cookie) - Aggiunge un cookie nell'intestazione (header) della risposta

HTTPSession getSession(boolean create) - Una HTTPSession identifica il client.

Viene creata se create=true

17.3.6 Esempio con POST: animale preferito



Lato client: la pagina HTML

```

1. <html>
2. <head>
3. <meta charset="UTF-8">
4. <title>Post Example</title>
5. </head>
6. <body>
7.   <form action="http://localhost:8080/SlideServlet/PostHTTPServlet" method="P
8.     What is your favorite pet?<br>
9.     <br> <input type="radio" name="animal" value="dog">Dog<br>
10.    <input type="radio" name="animal" value="cat">Cat<br> <input
11.      type="radio" name="animal" value="bird">Bird<br> <input
12.      type="radio" name="animal" value="snake">Snake<br> <input
13.      type="radio" name="animal" value="none" checked>None <br><br>
14.    <input type="submit" value="Submit"> <input type="reset">
15.  </form>
16.</body>
17.</html>

```

Lato server: il metodo doPost

```

1. public class PostHTTPServlet extends HttpServlet {
2.   // definisco l'elenco degli animali
3.   private String animalNames[] = { "dog", "cat", "bird", "snake", "none" };
4.   protected void doPost(HttpServletRequest request, HttpServletResponse res
5.     int animals[] = null; // contatori di preferenze
6.     int total = 0; // totale delle preferenze espresse
7.     // i dati sono memorizzati nel file "survey.dat"
8.     File f = new File("survey.dat"); // apro o creo il file
9.     if (f.exists()) {
10.       ObjectInputStream input = new ObjectInputStream(new FileInputStream(
11.         try {
12.           animals = (int[]) input.readObject(); // leggo il file e lo asse
13.         } catch (ClassNotFoundException e) {
14.           e.printStackTrace();
15.         }
16.         input.close(); // close stream
17.         // conto quante sono le risposte date in precedenza
18.         for (int i = 0; i < animals.length; ++i)
19.           total += animals[i];

```

```

20.     } else // creo un nuovo array di contatori
21.         animals = new int[5];
22.     // leggo il messaggio con la nuova preferenza
23.     String value = request.getParameter("animal");
24.     ++total; // aggiorno il totale delle risposte
25.     // determino quello votato e aggiorno il suo contatore
26.     for (int i = 0; i < animalNames.length; ++i)
27.         if (value.equals(animalNames[i]))
28.             ++animals[i];
29.     // scrivo i nuovi contatori sul file e lo chiudo
30.     ObjectOutputStream output = new ObjectOutputStream(new FileOutputStream("risposte.dat"));
31.     output.writeObject(animals);
32.     output.flush();
33.     output.close();
34.     // calcolo le percentuali
35.     double percentages[] = new double[animals.length];
36.     for (int i = 0; i < percentages.length; ++i)
37.         percentages[i] = 100.0 * animals[i] / total;
38.     // uso un Buffer di servizio per costruire la pagina html in
39.     StringBuffer buf = new StringBuffer();
40.     buf.append("<html>\n<title>Thank you!</title>\n");
41.     buf.append("Thank you for participating.\n");
42.     buf.append("<br>Results:\n<pre>");
43.     DecimalFormat twoDigits = new DecimalFormat("#0.00");
44.     for (int i = 0; i < percentages.length; ++i) {
45.         buf.append("<br>" + animalNames[i] + ": " + twoDigits.format(animals[i]));
46.         buf.append("% responses: " + animals[i] + "\n");
47.     }
48.     buf.append("\n<br><br>Total responses: " + total);
49.     buf.append("</pre>\n</html>");
50.     // costruisco l'head del messaggio di risposta
51.     response.setContentType("text/html"); // dichiaro il formato
52.     // predispongo il canale stream per la scrittura del body della risposta
53.     PrintWriter responseOutput = response.getWriter();
54.     // scrivo la pagina nella risposta
55.     responseOutput.println(buf.toString());
56. }
57. }
```

Perché a 4. è protected? Perché deve essere ereditabile.

17.3.7 Ciclo di vita di una servlet

Una servlet viene creata dal container/engine:

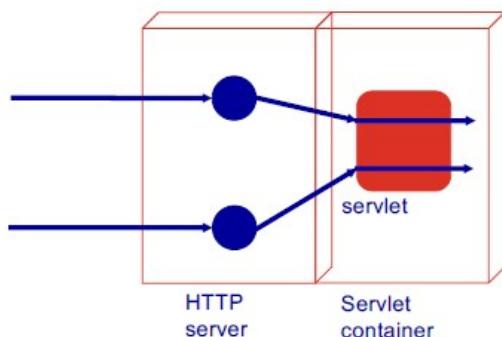
- Quando viene effettuata la prima chiamata
- La servlet viene condivisa da tutti client
- Ogni richiesta genera un **Thread** che esegue la **doXXX** appropriata

Il container/engine invoca il metodo **init()** per inizializzazioni specifiche.

Una servlet viene distrutta dall'engine all'occorrenza di uno dei due eventi:

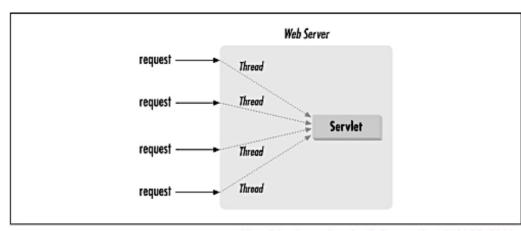
- Quando non ci sono thread in esecuzione su quella servlet
- Quando è scaduto un timeout predefinito

Viene invocato il metodo **destroy()** per terminare correttamente la servlet.



Una struttura del genere ha un nome, ma non ho capito quale. Penso "multitenant" perché ci sono più Client, ma ha senso come cosa?

17.3.8 Servlet e Thread



- Molti thread -*i* una singola istanza della servlet

- Più richieste vengono servite dalla stessa servlet, questo vuol dire che bisogna far attenzione a come la servlet implementa l'accesso alle sue risorse
- In questo modello spesso le risorse sono condivise a livello di database

Questa scelta ha diverse motivazioni:

- Utilizzare meno memoria (un solo oggetto)
- Ridurre il costo di gestione (per esempio tempi di inizializzazione inizializzazione) di molti oggetti che sarebbero spesso identici
- Abilita la persistenza (in memoria) di risorse condivisibili tra diverse richieste (e.g. una connessione a DB, oggetto "carrello")

Tornando all'esempio di prima, possiamo notare un problema abbastanza subdolo: non è gestito l'accesso in contemporanea al file.

17.3.9 Terminazione

Container e richieste dei client devono sincronizzarsi sulla terminazione

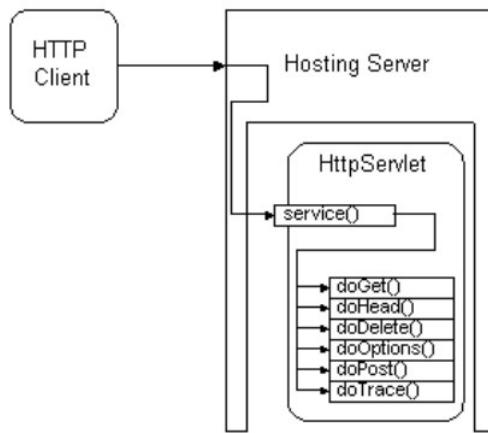
- Alla scadenza del *timeout* potrebbe essere ancora dei thread in esecuzione in **service()**

Bisogna:

- Tener traccia dei *thread* in esecuzione
- Progettare il metodo **destroy()** in modo da notificare lo shutdown e attendere il completamento del metodo **service()**
- Progettare i metodi lunghi in modo che verifichino periodicamente se è in corso uno shutdown e comportarsi di conseguenza

17.3.10 Realizzazione di applicazioni

Servlets "classiche", uso dell'ereditarietà. Supporta tutti i metodi.



Spring MVC:

- Spring@MVC, dalla versione 2.5 gestisce tutti i metodi HTTP
- Usa delle annotazioni per assegnare i metodi

```

@Controller
public class BookingsController {

    @RequestMapping(value="/bookings", method=RequestMethod.GET)
    public String getBookings() {
        return "someView";
    }

    @RequestMapping(value="/bookings", method=RequestMethod.POST)
    public void addBooking(HttpServletRequest request) {
        // read request
    }
}

```

”@Controller” è un’annotazione di Java (e Spring) che indica che la classe è un controller. Vengono usate per arricchire la classe descrivendo cosa fa. In questo caso, la classe è un controller che gestisce le richieste HTTP.

Annotation	Description
@PathVariable	Sets the path to base URL + “yourPath”. The base URL is based on your application name, the server and the URL pattern from the web.xml configuration file.
@POST	Indicates that the following method will answer to a HTTP POST request
@GET	Indicates that the following method will answer to a HTTP GET request
@PUT	Indicates that the following method will answer to a HTTP PUT request
@DELETE	Indicates that the following method will answer to a HTTP DELETE request
@Produces(MediaType.TEXT_PLAIN more-types)	Indicates which MIME type is delivered by a method annotated with @GET, in the example text (“plain”) is produced. Other examples would be “application/xml” or “application/json”
@Consumes(MediaType more-types)	Indicates which MIME type is consumed by this method
@PathParam	Used to inject values from the URL into a method parameter. This way you inject for example the ID of a resource into the method to get the correct object.

JAX-RS (e.g., jersey framework)

- JSR 311
- Framework REST basato su annotazioni

```

@Path("/hotels")
public class HotelsResource {

    @GET
    @ProduceMime( "application/xml")
    public HotelList getHotels(
        @QueryParam("searchString") String searchString) {
        // do something with query parameter
    }

    @Path("{hotelId}")
    public HotelResource getHotel() {
        return new HotelResource();
    }
}

https://jersey.java.net/documentation/latest/getting-started.html

```

17.4 SERVER SIDE - JSP

JSP sta per Java Server Pages.

17.4.1 Java Server Pages

È una tecnologia per la creazione di applicazioni web. Specifica l'interazione tra un contenitore/server ed un insieme di "pagine" che presentano informazioni all'utente.

Le pagine sono costituite da tag tradizionali (HTML, XML, WML, ...) e da tag applicativi che controllano la generazione del contenuto (generazione server-side).

JSP facilita la separazione tra logica applicativa e presentazione.

Analogo alla tecnologia Microsoft Active Server Page (ASP).

Differenze:

- una Java Server Page chiama un programma Java eseguito sul Web server
- una Active Server Page contiene uno script VBScript o JScript

JavaServer Pages (JSP) separano la parte dinamica delle pagine dal template HTML statico.

Il codice JSP va incluso in tag delimitati da

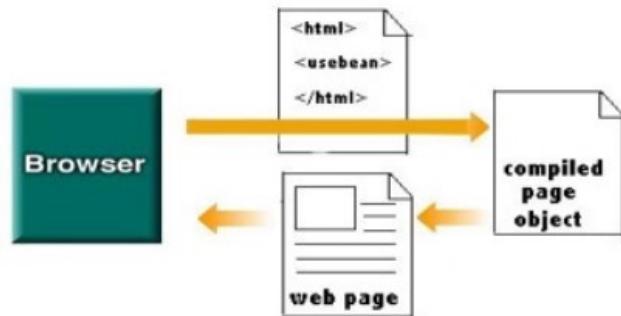
"<%" e "%>".

Esempio

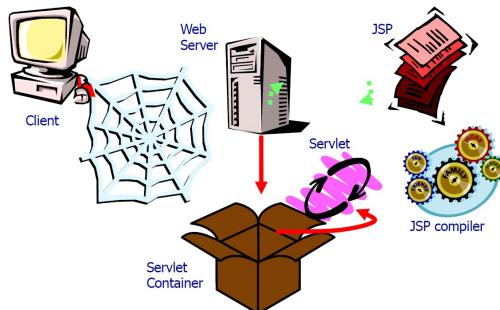
- una pagina che visualizza

Grazie per la scelta di Internet Guida Pratica

- quando l'utente si connette all'URL `http://host/OrderConfirmation.jsp?title=Internet+Guida+...`
 § La JSP contiene Grazie per la scelta di jijj § La pagina viene convertita automaticamente in una servlet java la prima volta che viene richiesta. Output:



17.4.2 Ciclo di vita delle applicazioni JSP



17.4.3 JSP: esempio

```

1. <%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
3. <html>
4.   <head>
5.     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6.   <title>Welcome</title>
7. </head>
8. <body>
9.   <h1> <%= "Welcome to JSP!" %> <h1>
10. </body>
11. </html>
  
```

17.4.4 Gli elementi di una JSP

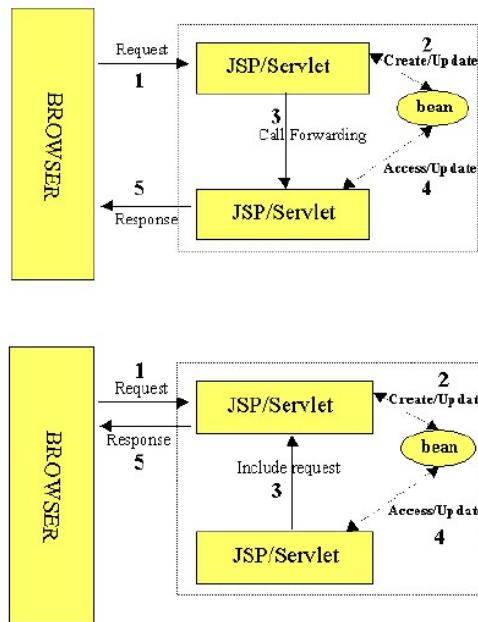
Template text § Le parti statiche della pagina HTML § Commenti j§ Direttive j§ Azioni jjsp:XXX attributes; body jjsp:XXX; § Elementi di scripting § Istruzioni nel linguaggio specificato nelle direttive § Sono di tre tipi: scriptlet, declaration, expression

Direttive

Le direttive non influenzano la gestione di una singola richiesta HTTP ma influenzano le proprietà generali della JSP e come questa deve essere tradotta in una servlet. page § Liste di attributi/valore § Vengono per la pagina in cui sono inseriti jjj§ include § Include in compilazione pagine HTML o JSP j§ taglib § Dichiara tag definiti dall'utente implementando opportune classi jjtable:loop; ... jjtable:loop;

Azioni

Le azioni permettono di supportare diversi comportamenti della pagina JSP. Vengono processati ad ogni invocazione della pagina JSP. Permettono di trasferire il controllo da una JSP all'altra, di interagire con i Java Data Beans, ecc. forward § determina l'invio della richiesta corrente, eventualmente aggiornata con ulteriori parametri, alla JSP indicata jjsp:forward page="login.jsp" ; jjsp:param name="username" value="user" /; jjsp:param name="password" value="pass" /; jjsp:forward; § include § invia dinamicamente la richiesta ad una data URL e ne include il risultato jjsp:include page="shoppingCart.jsp" /; § useBean § localizza ed istanzia (se necessario) un javaBean nel contesto specificato § Il contesto può essere § La pagina, la richiesta, la sessione, l'applicazione jjsp:useBean id="cart" scope="session" class="ShoppingCart" /;



Elementi di scripting

Declaration § Variabili o metodi statici usati nella pagina § Expression § Una espressione nel linguaggio di scripting (Java) che viene valutata e sostituita al tag § La radice di 2 vale § Scriptlet § Frammenti di codice che controllano la generazione del codice HTML, valutati alla richiesta § Tale codice diventerà parte dei metodi doGet (doPost) della servlet che viene associata la JSP. § table§ tr§ td§ /table§ In questo modo riusciamo a superare un ostacolo importante di HTML che è la staticità.

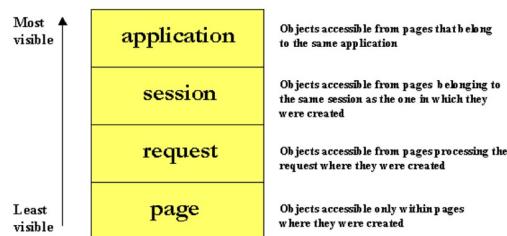
Un linguaggio di script ha lo scopo di: § interagire con oggetti java e altre servlet § gestire le eccezioni java § Può utilizzare anche oggetti impliciti (sono 9) § request § response § out § page § pageContext § session § application § config § exception Esempio: Grazie per la scelta di § i

17.4.5 Oggetti e loro "scope"

Lo scope è la durata del ciclo di vita di un oggetto.

Gli oggetti possono essere creati § implicitamente usando le direttive

JSP § esplicitamente con le azioni § direttamente usando uno script
 (raro) § Gli oggetti hanno un attributo che ne definisce lo “scope”



17.4.6 JavaBeans

Un javabean è una classe che segue regole precise (specifica) § Deve avere costruttori senza parametri § Dovrebbe avere campi (property) privati § I metodi di accesso ai campi (property) sono set/get setXxx getXxx/isXxx con xxx = property

```

1. class Book{
2.     private String title;
3.     private boolean available;
4.     void setTitle(String t) ...;
5.     String getTitle() ...;
6.     void setAvailable(boolean b) ...;
7.     boolean isAvailable () ...;
8. }
```

Azioni per utilizzare un bean

Accedere ad un bean (inizializzazione)

```

<jsp:useBean id="user" class="com.jguru.Person" scope="session" />
<jsp:useBean id="user" class="com.jguru.Person" scope="session" >
    <% user.setDate(DateTime.getDateInstance().format(new Date()) %>
</jsp:useBean>
```

Accedere alle proprietà

```

<jsp:getProperty name="user" property="name" />
<jsp:setProperty name="user" property="name" value="jGuru" />
<jsp:setProperty name="user" property="name" value="<% expression %>" />
```

Accesso ad un JavaBean

```
<jsp:useBean id="Attore" class="MyThread" scope="session" type="Thread"/>
```

Lo scope determina la vita e visibilità del bean:

page: è lo scope di default, viene messo in pageContext ed acceduto con getAttribute

request: viene messo in ServletRequest ed acceduto con getAttribute

session e application: se non esiste un bean con lo stesso id, ne viene creato uno nuovo

Il type permette di assegnargli una superclasse o un'interfaccia.

Al posto della classe si può usare il nome del bean

beanName=“nome” nome è la classe o un file serializzato

CounterBean.java

```
1. public class CounterBean {
2.     //declare a integer for the counter
3.     private int count;
4.
5.     1. public int getCount() {
6.         //return count
7.         3. return count;
8.     }
9.
10.    1. public void increaseCount() {
11.        //increment count
12.        3. count++;
13.    }
14. }
```

Counter.java

```
1. <%@ page import="it.is.mvc.CounterBean"%>
2. <jsp:useBean id="session_counter" class="it.is.mvc.CounterBean" scope="se
3. <jsp:useBean id="app_counter" class="it.is.mvc.CounterBean" scope="applic
4. <%
```

```

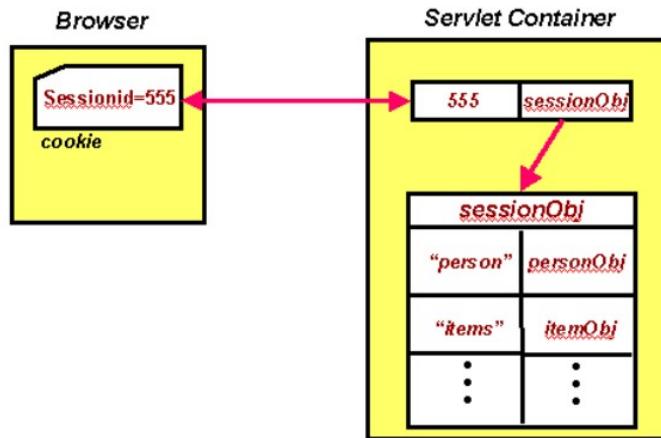
5.     session_counter.increaseCount();
6.     synchronized (page) {
7.       app_counter.increaseCount();
8.     }
9.   %>
10.  <h3>
11.    Number of accesses within this session:
12.    <jsp:getProperty name="session_counter" property="count" />
13.  </h3>
14.  <p></p>
15.  <h3>
16.    Total number of accesses:
17.    <% synchronized (page) { %>
18.      <jsp:getProperty name="app_counter" property="count" />
19.      <% } %>
20.  </h3>

```



17.4.7 Sessioni

Accede ad un oggetto HttpSession (session).
Mappa chiave-valore.



Esempi

- Memorizzazione

```
<% Foo foo = new Foo(); session.putValue("foo", foo); %>
```

- Recupero

```
<% Foo myFoo = (Foo) session.getValue("foo"); %>
```

- Esclusione di una pagina dalla sessione

```
<%@ page session="false" %>
```

17.4.8 Altro esempio: uso di dati dinamici

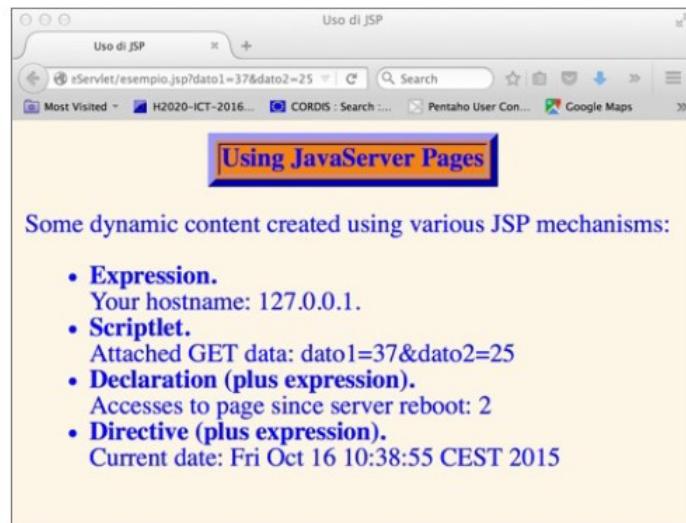
1. <%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
3. <html>
4. <head>
5. <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6. <title>Uso di JSP</title>
7. <link rel="stylesheet" href="My-Style-Sheet.css" type="text/css">
8. </head>
9. <body bgcolor="#FDF5E6" text="blue">
10. <center>

```

11.         <table border="5" bgcolor="#EF8429">
12.             <tr>
13.                 <th class="title"> Using JavaServer Pages </th>
14.             </tr>
15.             </table>
16.         </center>
17.         <br> Some dynamic content created using various JSP mechanisms
18.         <ul>
19.             <li><b>Expression.</b><br> Your hostname: <%=request.get
20.             <li><b>Scriptlet.</b><br> <% out.println("Attached GET d
21.             <li><b>Declaration (plus expression).</b><br> <%!private
22.                 Accesses to page since server reboot: <%=++accessCou
23.                 <li><b>Directive (plus expression).</b><br> <%@ page imp
24.             </ul>
25.         </body>
26.     </html>

```

Output:



17.4.9 Es. JSP con Form HTML

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <meta charset="UTF-8">

```

5. <title>JSP Example</title>
6. </head>
7. <body>

8. <form action="http://localhost:8080/SlideServlet/FC.jsp" method="GET">
9.     Converter <br><br>
10.    <input type="radio" name="unit" value="celsius">Celsius<br>
11.    <input type="radio" name="unit" value="fahrenheit">Fahrenheit<br>
12.    value <input type="text" name="value" ><br><br>
13.    <input type="submit" value="send">
14.    <input type="reset">
15.</form>

16.</body>
17.</html>
```

FC converter in versione JSP

```

1. <%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
2. <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
3. <html><head><meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
4. <title>F2C converter</title></head>
5. <body>
6. <%@ page import = "java.text.DecimalFormat" %>
7. <% Double result;
8. String unit = request.getParameter("unit");
9. Double value=Double.parseDouble(request.getParameter("value")); %>
10. <h1> <% if( unit.equals("fahrenheit")) { %>
11. Fahrenheit to Celsius
12. <% result=(value - 32) * 5 / 9;
13. } else { %>
14. Celsius to Fahrenheit
15. <% result=((value * 9) / 5) + 32; } %>
16. </h1>
17. <% DecimalFormat twoDigits = new DecimalFormat( "#0.00" ); %>
18. <h2> <%= unit %> : <%= twoDigits.format(value) %> </h2>
19. <h2> <% if( unit.equals("fahrenheit")) { %>
20. fahrenheit:
21. <%
22. } else { %>
```

```

23. celsius:
24. <% } %>
25. <%= twoDigits.format( result ) %>
26. </h2>
27. </body>
28. </html>

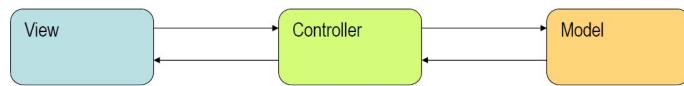
```

17.5 Il pattern MVC - Model View Control

17.5.1 Il pattern MVC

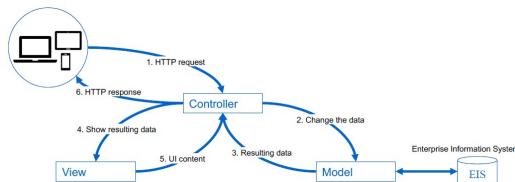
Il pattern Model View Controller (MVC) ha lo scopo di separare

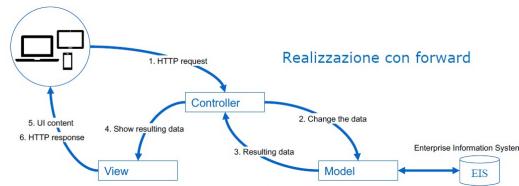
- I dati e i metodi principali per la loro manipolazione (Model)
- La presentazione, cioè l’interfaccia (View)
- Il coordinamento dell’interazione tra interfaccia (azioni degli utenti) e i dati (Controller)



17.5.2 Il design pattern MVC

Il Model-View-Controller (MVC) è un pattern architetturale che separa data model, user interface, e control logic in tre componenti distinte § Model § I dati (gli oggetti) trattati dall’applicazione, e le operazioni su di essi § View § La struttura dei dati restituiti al richiedente (e.g., la pagina HTML/CSS) § Control § Definisce le azioni da eseguire a fronte di una richiesta § Interagisce con il Model per modificare i dati e con la View per generare la risposta





17.5.3 Vantaggi e svantaggi

Vantaggi:

Chiara separazione tra logica di business e logica di presentazione § poter cambiare la view senza modificare il control e viceversa § poter arricchire la view senza appesantire il codice del controller § poter definire e scegliere a run-time la view da usare a seconda dell'interazione, dei device utilizzati, dello stato dei dati o delle preferenze dell'utente § Chiara separazione tra logica di business e modello dei dati § poter definire diversi mapping tra le azioni degli utenti sul controller e le funzioni sul model § Ogni componente ha una responsabilità ben definita § poter sviluppare in parallelo § poter manutenere e far evolvere ogni componente in modo indipendente, quindi semplificando la gestione § Ogni parte può essere affidata a esperti § poter assegnare lo sviluppo della view ad esperti di interfaccia e interazione (e.g., pagine jsp o asp) § uso tecnologie adatte allo sviluppo delle singole componenti

Svantaggi:

Aumento della complessità dovuta alla concorrenza (è un sistema distribuito).

Inefficienza nel passaggio dei dati alla view (un elemento in più tra cliente e controller).

17.5.4 Il pattern Model 1

Scopo: separazione tra dati, logica di business e visualizzazione

1. Il browser invia una richiesta per la pagina JSP
2. JSP accede a Java Bean e invoca la logica di business
3. Java Bean si connette al database e ottiene/salva i dati
4. La risposta, generata da JSP, viene inviata al browser

Architettura tipica:

Capitolo 18

Services Computing

18.1 IoT

The Internet of Things is a system of physical objects that can be discovered, monitored, controlled, or interacted with by electronic devices that communicate over various networking interfaces and eventually can be connected to the wider internet.

IoT deals with ubiquitous communication and connectivity

Older definitions focus mostly on connectivity and sensory requirements for entities involved in typical IoT environments.

New definitions focus give more value to the need for ubiquitous and autonomous networks of objects where identification and service integration have an important and inevitable role.

Parliamo di **sapere e agire**: fa l'esempio delle luci, ovvero c'è differenza fra sapere che le luci sono accese ed essere in grado di spegnerle.
Ma perché parliamo di tutto ciò?

18.1.1 IoT e servizi

Electronic devices are able to collect and exchange data to populate the platforms of the future =; servitization of the real world.

Things as a Service

- High number of components involved in single applications
- Dynamic (re)configuration of applications
- Things consume/expose services

Example: When a customer enter a shop holding a smartphone, he/she becomes part of an ecosystem that includes sensors installed in the shop, services accessing data from the retailer Information System, and external internet services.

Es. cellulare:

Piattaforma che genera contenuti usando servizi, che possono essere in parte in esecuzione sulla piattaforma, o il lato client-side delle nostre app (l'altro lato è la parte di servizi remota).

I servizi cloud permettono di montare diverse funzionalità all'interno di un unico back-end, che può essere utilizzato da più servizi.

18.2 Services Computing: a change of perspective

Finora abbiamo parlato di applicazioni web, client-server che comunicano fra loro tramite messaggi es http.

A change was felt needed in systems design, deployment and maintenance w.r.t. classical distributed architectures (web based applications) Traditional (distributed) systems are single systems developed top-down by decomposition (prendono anche il nome di repository? questo?)

- Generally, **client-server**
- **Tight coupled:** A single organization maintain/own the system

Di solito però si cerca di evitare l'accoppiamento stretto perché ovviamente quello che modifco ad un elemento va anche a tutto ciò a cui è accoppiato, penalizzando tutta l'interfaccia. § Contemporary distributed systems include third-party sub-systems § E.g., a banking system does not offer only banking products to customers. It offers insurance contracts, event tickets, ... § Loose coupling: each component is maintained independently from the others § Each software component should be able to serve different systems in different ways

18.3 Service Oriented Architecture - SOA

SOA is an architectural style that focuses on reusable discrete pieces (called services) instead of a monolithic (compatti) design to build applications. Sviluppo verticale.

A service provides functionalities to requestors (they can be other services)

Service access can be provided over the network (even the Internet)
Com'è possibile vedere nell'esempio prima delle SOA capitava anche di avere servizi scritti più volte, una per applicazione: pericoloso, perché nel caso di bug c'è il rischio di duplicare il suddetto bug. Con le SOA invece vado a suddividere in componenti che riutilizzerò nelle varie applicazioni, in modo da doverli scrivere solo una volta. Vado poi ad usare lo strumento della composizione di componenti per andare a creare le applicazioni. I servizi creati possono essere riutilizzati immediatamente nel mio software oppure vendere le singole funzionalità (servizi) a terzi.

Vantaggi e svantaggi Benefits:

- Reusable
- Agile and business-oriented development process
- Service economy
- Scalability (se ho problemi di scalabilità posso copiare il servizio, che avrà quindi diverse copie di se stesso e quindi è più riutilizzabile)
- Optimization and Cost reduction

Cons.

- Cumbersome life-cycle management (gestione del ciclo di vita dei servizi più complicata)
- Dependency hell (posso avere problemi se i servizi dipendono uno dall'altro in maniera così complicata da impedire la comprensione e l'utilizzo dei servizi, magari se poi non sono perfettamente sincronizzati si creano situazioni di blocco etc...)
- Integration with legacy solutions
- ...

Affinché un modello possa essere definito servizio, deve: (Each service should:)

1. Provide a description of its functionalities to be discovered and selected.
2. Provide access to its functionalities through well-known network protocols.

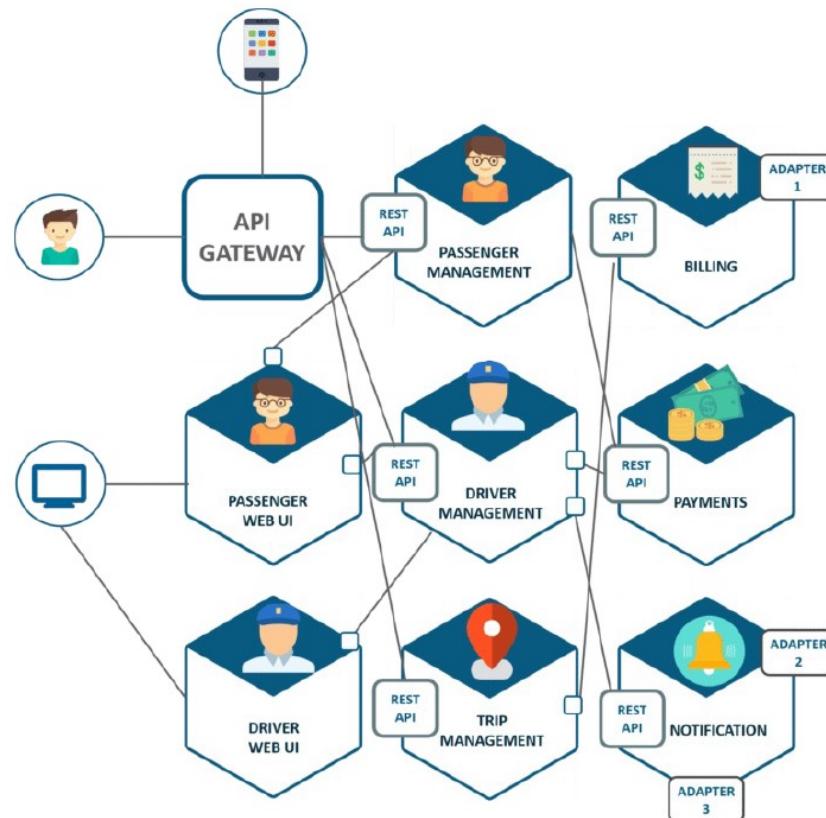
3. Support composition with other services to deliver complex solutions.
4. Address clients' business needs and domain requirements (functional and non-functional). [deve essere orientato ai bisogni dell'utente, dal pov funzionale e non.]
5. Guarantee a certain level of Quality of Service (QoS). [non riguarda il cosa venga fornito ma il come]

Esempio: SOA per Uber

Come anche per Netflix, abbiamo migliaia di servizi che cooperano.

SOA architecture at Uber (simplified)

At Uber services are mainly internal, but not exclusively
Services implements a known interface



18.4 But... what is a “(Web) Service”?

Informal definition: A service is an independent software entity that can be discovered and invoked by other software systems over a network. § Definition from standardization body W3C (organo che gestisce le specifiche web, così dice CoPilot): A Web service is a software application

1. identified by a URI (Uniform Resource Identifier),
2. whose interfaces [...] are capable of being defined, described and discovered [...] and
3. supports direct interactions with other software applications using [...] messages via Internet-based protocols (di solito documenti in formato json, html, xml, etc...).

A web service uses web technologies (especially HTTP) to provide functionalities. Non sono gli unici ma i più importanti in questo momento. A me interessa sapere come è fatto un servizio, come invocarlo ed utilizzarlo.

Altre definizioni: § Web services are § self-contained, § self-describing, § modular applications that can be published, located, and invoked across the Web. (Tidwell, IBM, 2001) § Web services are § encapsulated, § loosely coupled Web “components” that § can bind dynamically to each other. (Curbera, IBM, 2001)

18.5 Architectural building blocks

Services are independent “components” 1. Well-known interface 2. Unique access point 3. Document-based data exchange 1. Well-known interface § Standard description language (e.g., WSDL) linguaggio usato per la definizione logica dell’interfaccia di un particolare servizio. § Possible automatic management by middleware

The term middleware is most used for software that enables communication and management of data in distributed applications.

2. Unique **access point** § Use of URI (URL/URN) § Discoverable by means of name services (e.g., UDDI directory)

UDDI is an XML-based standard for describing, publishing, and finding web services. UDDI stands for Universal Description, Discovery, and Integration

3. **Document-based** data exchange § Use of standard representation format (e.g., XML, JSON)

18.6 SOA core elements

2 Componenti:

- Service
- Service Description

Ruoli:

- Service Providers: offer services/functionalities
- Service Brokers: menage service catalogs (i cataloghi dei servizi sono praticamente delle basi di dati in cui posso ricercare i servizi che meglio si adatta alle mie necessità fra tutti i servizi disponibili)
- Service Requestors: find a service and interacts with providers

Operazioni:

- Publish (a service)
- Find (service/endpoint)
- Interact (e.g., request-response)

18.7 Service Level Agreement (SLA)

SLA is a contract between the provider and the user of the services.

- Ensures that functionality is delivered correctly
- Non-functional properties

SLA (Service Level Agreement)

- Defines the non-functional characteristics guaranteed by the service
- An SLA includes several SLOs (Service Level Objectives) that define the quality of service to be guaranteed through specific metrics

Ovviamente dire che un programma (Java) è corretto non basta per dire che sia utilizzabile: potrebbe essere troppo complesso o elaborato per essere riutilizzato.

18.8 Peculiarità dei Web Services

Public components § “Discoverable” (naming, registries, accessing) § Public interfaces (standard protocols, machine handled) § Composability § Composite services: “orchestration” § Coordination: “choreography” § Semantic Descriptions § Discovery § Composition § Recommending Systems (in base alle parole che uso, mi consiglia servizi simili a quello che sto cercando) § ... § QoS § Basic: security, availability, performance, ... § Context awareness § ... § System organization § p2p (any organization, application managed) § ESB (any organization, middleware managed) § Grid (given model) [a griglia]

Queste modalità rappresentano anche delle sfide.

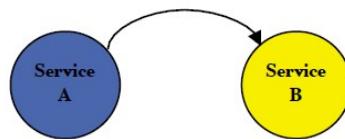
18.8.1 Challenges: how to do...

Description § functional and QoS § Composition § orchestration § choreography § Semantic ... § Infrastructure § Abstract/concrete architectural model § Services for services (servizi al servizio di altri servizi) § Service management § Global naming service § Software engineering § Development process § Deployment § Services for services § Global discovery service § Reputation § Transaction (workflow and exceptions) § Security § Monitoring (deve essere ben distribuito, visto che i servizi sono distinti) § Business § Business process support § Economic models § Organizational models

18.9 Composizione di servizi

What's a service composition? § A composition consists of a set of interconnected services, which may be used as a new service in other compositions § Two services are interconnected if at least one of the two requests the exposed functionality (a.k.a. endpoint a.k.a. API) of the other Compatibility is required between interacting services for service composition to be successful § They must talk the same language

(syntactically and semantically) § A service provider must present a published interface



18.10 Business Processes

Quando abbiamo tanti servizi, a livello aziendale possiamo pensare di metterli tutti assieme sullo stesso piano per riuscire a gestirli più facilmente. Parliamo di business process.

Business Process: a set of related activities (workflow) performed by people and applications to achieve a well-defined business outcome (service or product).

Software BPs are created from the composition (integration) of services. It may contain defined conditions triggering its initiation and defined outputs at its completion. § It may involve formal or relatively informal interactions between participants (humans and software). § It may contain a series of automated activities and/or manual activities. § It is distributed and customized across boundaries within and between organizations, often spanning multiple applications with different technology platforms. § It has a duration that may vary widely. § It is usually long running. A single instance may run for months or even years.

Dal punto di vista aziendale, per esempio questa parte del business process è affidata agli ingegneri del software.

18.10.1 Software side: Orchestration vs Choreography

Orchestration

describes how services interact with each other, including the business logic and execution order of the interactions from the perspective and under the control of a single actor (service). § For instance, there is

orchestration when a service uses other services' functionalities (in the right order) to implement its business logic § It requires active control. Cumbersome but easier to monitor.

Choreography

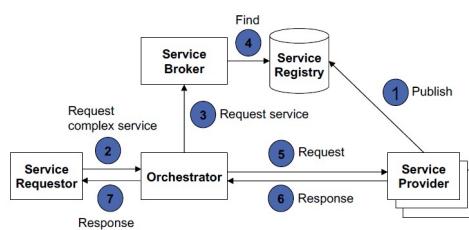
Interazione e cooperazione di tutti i servizi. Ogni servizio è consapevole di quale è la sequenza di invocazione degli altri servizi.

Describes the sequence of interactions among multiple parties involved in the process from the perspectives of all parties. It defines the shared state of the interactions between business entities.

Every service is responsible for completing its task also invoking other services. Usually implemented using events.

18.11 Architetture orchestrate

Molto semplici. It is therefore possible to create a dynamic network of services, where individual services can be offered by different organizations § One organization assumes the role of orchestrator and takes on the burden of implementing the controller service for other organizations § There are usually organizations called service brokers. § Find the best alignment between services requested by Service Requestors and services offered by Service Providers



18.12 Enterprise Service Bus

Pattern molto comune perché permette di facilitare l'integrazione fra applicazioni. The SOA architecture has allowed the simplification of integration between the different components of an information system § Exchangeable messages are defined a priori § It has led to the definition of the Enterprise Service Bus (ESB) § Communication system

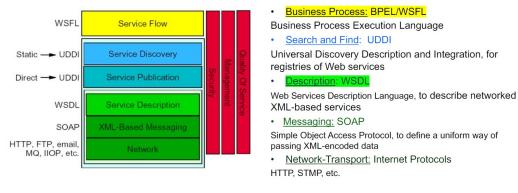
to support interaction and communication between components of an information system § Example of the choreographic approach § Usually implements the publish/subscribe pattern § Main Features § Message routing between applications and services § Message transformation § Secure communication § Extensible architecture

Capitolo 19

Soap Services

19.1 The Conceptual Web Service Stack

A livello di rete, permette la comunicazione fra servizi in modo da rendere possibile il funzionamento dei servizi web.

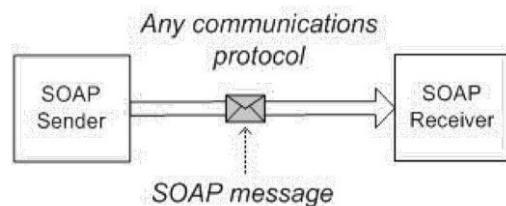


Alla base di tutto questo meccanismo c'è il protocollo SOAP.

19.2 SOAP: Simple Object Access Protocol

SOAP is an XML-based protocol to let software components and applications to communicate using XML messages § XML namespaces are used to provide semantics to data (not just data types) § It is agnostic of the transportation subsystem § Request and response messages § SOAP is a standard way to structure XML Messages (Data) § An application of the XML specification § Relies on XML Schema, XML Namespaces § It is not tied to any programming language or platform (not even

to web services) § Simple and extensible § Used in document-oriented services, ovvero c’è scambio di informazioni in formato testuale (se non fosse orientato a documenti, a cosa potrebbe essere orientato? Una possibilità sarebbero le chiamate a procedura remota, hanno la struttura di una funzione e l’unica cosa che c’è bisogno di fare è definire il nome e i parametri di I/O).



19.2.1 Componenti di un messaggio SOAP

SOAP Envelope

Wraps the contents of the message

SOAP Header (opzionale)

More flexibility, can be processed by nodes between the source and destination

Contains blocks of information regarding how to process the message:

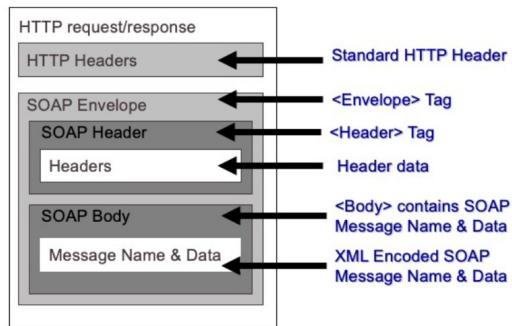
Routing and delivery settings

Authentication/authorization assertions

Transaction contexts

SOAP Body

Actual message to be delivered and processed
Both for request and response information



In questo esempio il messaggio SOAP è composto come un payload di un messaggio HTTP.

19.2.2 SOAP with HTTP

Although SOAP can use several transport protocols, HTTP is the one most commonly used. § HTTP requests consist of an HTTP, such as POST or GET, followed by the URL requested and the protocol version (e.g. HTTP 1.1). § Responses follow the semantics of HTTP to provide the status code of the response; § For example, a status code of the type 2xx indicates that the request has been received, understood, accepted, etc. There are two models for exchanging messages SOAP via HTTP (è necessario che il client sia quello che per primo crea la comunicazione): § the "SOAP request-response" model in which the POST method is used to bring SOAP messages into the body of the http requests/responses; § the "SOAP response" model in which in http requests is used the GET method to get the SOAP message and insert it into the Body of the response. § Using SOAP with HTTP one must remember to set the Content-Type header as application/SOAP+xml

examples:

19.3 WSDL

Linguaggio basato su XML pensato per descrivere l'interfaccia di un servizio web.

19.3.1 What is WSDL?

§ Web Service Description Language (It is pronounced [wizdel]) § Used to define “How and where to access the service” § XML-Based Language for describing web services, the messages and how to invoke them § A WSDL file contains a description of a service’s interface § WSDL allows to describe four main pieces of data for a service: § Interface information describing all publicly available operations of a service § Data type declarations for all messages. Complex types can be declared (using SOAP) and used § Binding information about the transport protocol (HTTP, SMTP, UDP) § Address information for locating the service (URI) § Abstract vs. concrete services § WSDL is a W3C Recommendation § WSDL 2.0 became a W3C Recommendation 26 June 2007

19.3.2 WSDL 2.0: conceptual model

In the abstract part § Description of a web service in terms of messages it sends and receives through a type system, typically W3C XML Schema § An operation associates message exchange patterns with one or more messages § Message exchange patterns define the sequence and cardinality of messages exchanged between nodes (services) § An interface groups these operations in a transport and wire independent manner

In the concrete part § Bindings specify the transport protocol for interfaces § An endpoint associates a URI with a binding § A service groups the endpoints that implement a common interface

19.3.3 WSDL 2.0: definizione di binding

Binding

The name attribute defines the name of the binding. With this name you can reference it when defining a service endpoint. Every binding name must be unique within the WSDL 2.0 target namespace. The interface attribute contains the name of one of the defined interfaces. The type attribute defines which message format is to be used. In this case is SOAP (other bindings are possible). wsoap:protocol defines the “transport” protocol. In the example, soap messages are going to be transported using HTTP

19.4. PERCHÉ NON SI USA PIÙ - WHY SOAP DREAM FADED AWAY 139

```
<binding name = "myServiceInterfaceSOAPBinding"
    interface = "tns:myServiceInterface"
    type = "http://www.w3.org/ns/wsdl/soap"
    wssoap:protocol = "http://www.w3.org/2003/05/soap/bindings/HTTP/"
    <operation ref = "tns:checkServiceStatusOp"
        wssoap:mep = "http://www.w3.org/2003/05/soap/mep/soap-response"/>
    <fault ref = "tns:dataFault" wssoap:code = "soap:Sender"/>
</binding>
```

Operation

The ref attribute of the operation element references a specific operation (already defined in the interface section) The wssoap:mep attribute defines the message exchange pattern for SOAP (GET request)

Fault

The ref attribute defines which fault (already defined in the interface section) will be referring The wssoap:code attribute of the fault element defines the fault code that will trigger sending this fault message

c'è una slide

19.4 Perché non si usa più - Why SOAP dream faded away

The stack of protocols collapsed under its own weight § WSDL and SOAP are too verbose, hard to understand, complex § A more simple and readable approach was felt necessary § Agreed upon semantics for method and JSON for the payload § DNS, human-readable names for endpoints § A new (fun) kid in town: REST

Troppo preciso, troppo pedante, troppi riferimenti a XML. Ci sono ancora molti servizi che usano questa tecnologia, ma quelli nuovi difficilmente lo usano.

Capitolo 20

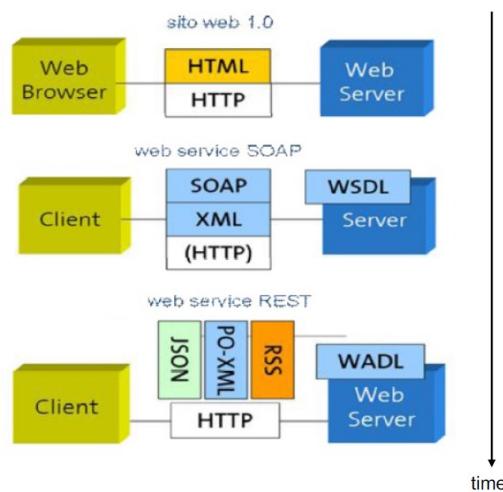
Services Computing - REST

20.1 Web Services Evolution

Initially only HTML and HTTP, so services were limited to providing web pages.

With SOAP the services are strongly characterized and precisely described. They could even be "compiled". However, services are complex and difficult to describe and understand.

In REST, we **go back to the principles of HTTP** by eliminating redundancies and assigning semantics to verbs (GET, POST, PUT...) and URIs.



Come abbiamo visto ieri, per la struttura al centro dell'immagine, SOAP è la struttura dei messaggi, XML la tecnologia abilitante (giu-

sto?) e HTTP il protocollo di scrittura. Questo ci porta ai servizi del server WSDL.

20.2 REST vs SOAP

	REST HTTP cell7	SOAP diversi protocolli (e.g., HTTP, TCP, SMTP) cell9
--	------------------------------	--

20.3 Composition = Establishing a Common Model

I sistemi distribuiti per funzionare devono basarsi su modelli comuni,

- WSDL/i sistemi SOAP SOA si devono accordare su API che comunicano tramite formati di messaggi comuni
- Very well defined but resulted overcomplex

REST is built around the idea of simplifying the agreement 1. **nouns** are required to name the resources that can be talked about (exchanged, deleted, created) 2. **verbs** are the operations that can be applied to named resources (get per ottenere una risorsa, post per non lo so, etc...) 3. **content types** (e.g., application/json) define which information representations are available, rappresentazione che vogliamo dare ad una risorsa

Es. piano registrazione esami studenti

Managing the university curriculum § Same data -; Different representations: administrative - didactics § Machine-oriented (XML/JSON) - human-oriented (HTML) representations § Links (URIs) define possible resource manipulation actions. Examples of endpoints: → GET unimib.it/matricola: returns information about the student and list of available actions: § unimib.it/matricola/cambiaPianoStudio § unimib.it/matricola/pianoStudio § unimib.it/matricola/administration → unimib.it/matricola/pianoStudio: § GET list of exams the student

is registered for, § POST register for an exam, § PUT modify exam registration, § DELETE exam registration → unimib.it/matricola/pianoStudio/code:
 § GET information about the exam the student is registered for

20.4 Principi REST

REST = REpresentational State Transfer REST is an **architectural style** for distributed systems

Resources are defined by URIs (che di solito hanno questa struttura: /baseURL/resourceType/id)

Resources are manipulated through their representations. There can be multiple representations for a resource (e.g., XML, JSON, HTML)
 La comunicazione avviene per messaggi, tutti semplici (stateless, privi di stato), autodescrittivi (devono contenere una descrizione dei dati); l'unico modo per interargirci è per richiesta-risposta. Una conseguenza è il poter usare solo get e post (ho capito bene?)

Application state is driven by resource manipulations.

Hypermedia is the way to control application behavior (altri link perché il Client possa scoprire da solo possibili manipolazioni effettuabili su tale risorsa).

N.B.: REST principles nicely fit the Web architectural components: URI and HTTP.

20.5 Representational State Transfer

Key information (data) abstraction: the resource
 § A resource is any information that can be named: documents, images, services, people, collections, etc.
 § Resources have/are state
 § Resources may change over time
 § Resources usually change with the interaction with the client
 § Resources have identifiers (constraint) che sono dinamici (unimib.it/matricola/pianoStudio)
 § matricola is called path parameter
 § A resource is anything important enough to be referenced
 § Resources expose a uniform interface (constraint)
 § GET, POST, PUT, DELETE + Identifiers in the URI
 § System architecture simplified; visibility improved.
 § Encourages independent evolvability of implementations. On request, a service may transfer a representation of the resource to a client
 § Requires a client-server architecture (constraint)
 § E.g., list of the exams registered for a certain student id

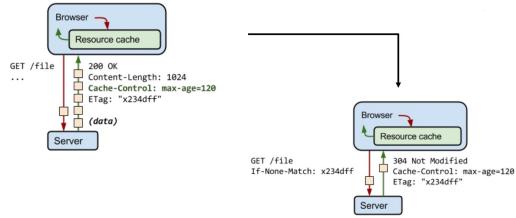
A client may transfer a modified representation of a resource
§ Manipulation of resources through representations (constraint) - no action invocation
§ E.g., a student can alter the list of exams by adding a new registration to the list
§ Representations returned from the server should link to additional representations/actions. Clients may follow a proposed link and assume a new state
§ Hypermedia as the engine of application state (constraint)
§ E.g., beside the list of registered exams, a link to add or delete a registration can be present
Stateless interactions (constraint)
§ Each request from client to server must contain all the information necessary to understand the request, and cannot take advantage of any stored context on the server
§ Statelessness necessitates **self-descriptive messages** (constraint)
§ Standard media types (example, application/json)
§ Meta-data and control-data
§ Uniform interface + Stateless + Self-descriptive = Cacheable (può essere mantenuta in una cache, che in un'interazione web è uno spazio di memoria dedicato alla comunicazione Client-Server, questo funziona solo se il messaggio è completo ovvero self-descriptive)
(constraint)
§ Cacheability necessitates a layered-system (constraint)
§ A cache layer is necessary
§ In the cache is stored the latest representation of a resource
§ A cache is invalidated upon resource state alteration (PUT, DELETE)

20.6 Caching

§ Caching reduces latency and network traffic
§ Only (non-SSL) GETs are cached – not POSTs etc.
§ Two kinds of caches
§ Client-side (for instance in the browser)
§ Proxy/server
§ Large SOAs use a caching proxy server (memcache, et similia)
§ You may be using it and not even know it
§ Remember, there can be caches all along the way

20.6.1 Local caching

Nell'immagine mostrata vediamo il server ricevere l'informazione: riceve come dato 200, e il cache-control mi dice "ok, nei prossimi 120 secondi (max-age = 120) l'informazione non dovrebbe cambiare, tieni l'informazione in cache".



20.7 Alcune osservazioni

REST is all about simplicity § Caching improves response time and reduces server load § Statelessness and less communication enables easier load balancing between servers § Less specialized software because the underlying technologies are well known and simple § Identification is done using standard mechanisms, no additional names necessary REST is (based on) standard(s) § REST principles emphasizes the correct and complete use of the HTTP protocol to publish services on the Web. § REST over HTTP provides a lightweight and layered mechanisms for data and service integration. § REST with HTTP protocol provides a distributed, hypermedia-driven application platform.

20.8 Using HTTP to build REST Applications

Important cache-control headers (in server response)	
Header	Property
Expires	HTTP date. Hold till expiry
Cache-Control: max-age	Seconds. Hold for this amount of time
Cache-Control: s-maxage	Seconds. As above, but proxies only
Cache-Control: public	Cacheable in any cache even if normally non cacheable
Cache-Control: no-cache	Cacheable, but cache must validate freshness
Cache-Control: no-store	Don't cache
Cache-Control: must-revalidate	Do not allow stale representations
Cache-Control: proxy-revalidate	As above, but proxies only

The REST Recipe: § Find all the nouns § Define the formats § Pick the operations § Highlight exceptional status codes Find all the nouns: saper scegliere i nomi a tutte le risorse è la cosa più difficile da fare in un REST. § Everything in a RESTful system is a resource – a noun § If you find yourself creating verbs, noun-ify them (ApproveExpense à ExpenseApprovals) § Every resource has a URI. Ideally just one (but don't sweat it) § URIs should be descriptive § <http://example.com/expenses/pending> § Spend some time here, but don't agonize over the perfect URI § URIs should be opaque (<https://www.w3.org/DesignIssues/Axioms.html#opaque>)

§ automated (non-human) clients should not infer meta-data from a URI
 § URIs should be cool - persistent (<http://www.w3.org/Provider/Style/URI>)
 § Cool URIs don't change (simplicity, stability, manageability)
 § Foundation requirement of the semantic web Avere una gerarchia d'accesso alle risorse obbliga il Cliente a fare che?

Capita che la versione dell'API di un servizio evolva: di solito si crea una versione nuova mantenendo attiva quella precedente.

Find all the nouns
 § Use path variables to encode hierarchy § /expenses/pending/123 (/expenses/pending/id)
 § Use other punctuation to avoid implying hierarchy /expenses/Q107;Q307 /expenses/lacey,peter
 § Use query variables to imply filtering conditions

```
/search?approved=false
```

§ You won't need query variables as much as you think

```
/expenses?start=20070101&end=20071231
```

should be /expenses/20070101-20071231
 § Caches tend to (wrongly) ignore URIs with query variables
 § URI space is infinite (but URI length is not 4K)
 § Don't leak platform information /expenses.php/123

20.8.1 Defining the formats

Neither HTTP nor REST mandate a single representation for data
 § A resource may have multiple representations XML, JSON, binary (e.g., jpeg), name/value pairs
 § Schema languages are not required (if even possible)
 § Avoid creating custom representations: use well-known media types (IANA registered MIME types)
 § Makes content more accessible
 § Part of the self-descriptive messaging constraint
 § Incoming representations should be the same as outgoing representations
 § A client should be able to GET, modify, and PUT a document
 § Server should discard any extraneous data

20.8.2 Pick the operations

§ Pick the operations which can be applied to resources
 § HTTP has a constrained user interface (set of verbs/operations/methods)
 § For most applications, HTTP's basic methods are sufficient
 § GET : Fetching a resource (there must be no side-effects)
 § POST : Adds to a new

resource on the server § PUT : Transfers a resource to a server (overwriting if there already is one) - Update § DELETE : Discards a resource § More methods are available § HEAD like GET but without body § OPTIONS (not widely supported) § Patch applies a partial update to the resource (you are only required to send the data that you want to update) § TRACE (not significant) § CONNECT (not significant) § All our resources will support GET

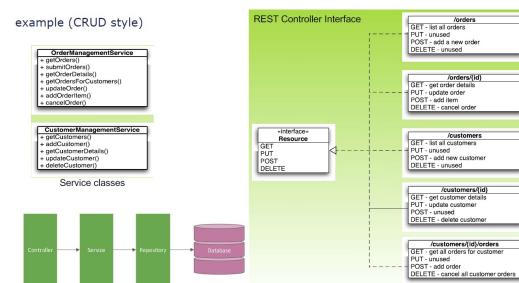
20.9 Operazioni HTTP

		cache	safe	Idempotent
OPTIONS	represents a request for information about the communication options available on the request/response chain identified by the Request-URI	✓	✓	✓
GET	means retrieve whatever information (in the form of an entity) is identified by the Request-URI	✓	✓	✓
HEAD	identical to GET except that the server MUST NOT return a message-body in the response	✓	✓	
POST	is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line			
PUT	requests that the enclosed entity be stored under the supplied Request-URI			✓
DELETE	requests that the origin server delete the resource identified by the Request-URI			✓
TRACE	is used to invoke a remote, application-layer loop-back of the request message		✓	✓
Patch	Is used to execute partial updates on a resource			Can be

Safe = An HTTP method is safe if it doesn't alter the state of the server. In other words, a method is safe if it leads to a read-only operation
Idempotent = the side-effects of N > 0 identical requests is the same as for a single request

Perché GET è idempotente? Me lo sono perso. Perché POST è idempotente? Perché ogni volta che lo eseguo mi si crea una nuova risorsa con il proprio identificativo.

20.10 REST come implementazione CRUD



20.10.1 Response Status Code

The response status code is generated by the server to indicate the outcome of a request.

The status code is a 3-digit number:

- 1xx (Informational): Request received; server is continuing the process.
- 2xx (Success): Request received, understood, accepted and serviced.
- 3xx (Redirection): Further action must be taken in order to complete the request.
- 4xx (Client Error): The request contains bad syntax or cannot be understood.
- 5xx (Server Error): The server failed to fulfill an apparently valid request.

Proper identification of status codes associated with a request is a crucial decision in REST development.

20.11 PUT vs. POST

When creating new resources § Use POST if the server chooses the URI (the id) § Use PUT if the client chooses the URI (the id) § POST can be used as "Process this" § POST does "something" and returns "something" § Usually masking a Remote Procedure Call (RPC) (overloaded POST) - function invocation § Don't use it casually § "Process this" POST sometimes valuable: § Resource factories § Sometimes you just want a verb § If you find yourself creating resources that can't be retrieved with GET, reexamine your design

`http://expense.example.com/pay_expense`

§ But don't sweat it, if it gets you out of a design jam

Ho saltato alcune slides

Hypermedia control

HATEOAS: Hypermedia As The Engine Of Application State § Application state manipulation = the current state manipulated by the client § Hypermedia = links and forms § Following a link can be seen as an application state transition § Server provides a new representation; client assumes that state § The server "guides" the client to new states by providing links inside resource representations § All representations should contain links § No links, no Web sono tre simplified