

Sistemi distribuiti

UniShare

Davide Cozzi
@dlcgold

Gabriele De Rosa
@derogab

Federica Di Lauro
@f_dila

Indice

1	Introduzione	2
2	Introduzione ai Sistemi Distribuiti	3
2.0.1	Stream Communication	7
2.0.2	Architettura dei server	27
2.0.3	Un esempio	28
2.1	L'architettura del web	36
3	HTML + CSS + JS	44

Capitolo 1

Introduzione

Questi appunti sono presi a le lezioni. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: <https://github.com/dlccgold/Appunti>.

Grazie mille e buono studio!

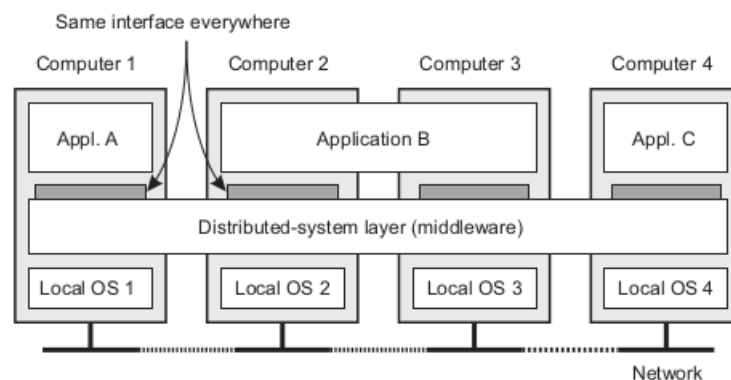
Capitolo 2

Introduzione ai Sistemi Distribuiti

In questo corso analizziamo i sistemi distribuiti, alla base di tutte le applicazioni software client/server, in cui è presente una comunicazione tra diversi host.

Definizione 1. *Un sistema distribuito è un sistema nel quale componenti hardware e software, collocati in computer connessi alla rete, in cui comunicano e coordinano le loro azioni col passaggio di messaggi. Ogni processo ha quindi una parte di logica applicativa e una parte di coordinamento.*

Definizione 2. *un sistema distribuito è un insieme di elementi autonomi di computazione che si interfacciano agli utenti come un singolo sistema "coerente".*



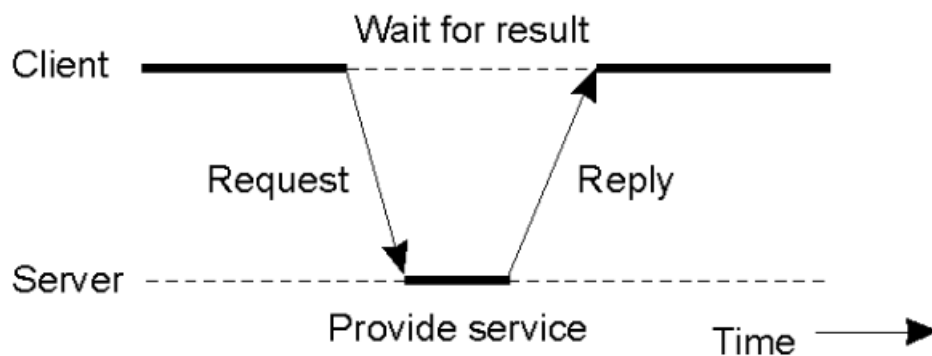
Le unità di computazioni sono dei nodi, i quali possono essere device hardware e/o singoli processi software, autonomi che devono essere sincronizzati

e coordinati (programmazione concorrente).

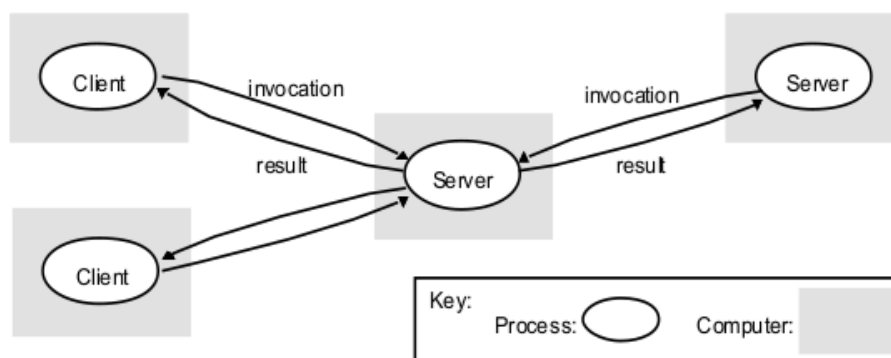
Gli utenti e le applicazioni vedono un singolo sistema, senza conoscere le varie segmentazioni e i nodi presenti, e questo permette di effettuare la **trasparenza di distribuzione**, ossia si nascondono i dettagli agli utenti che possono ignorare e non possono modificare il servizio.

Con la trasparenza in teoria si dovrebbero evitare la generazione degli errori, in quanto i nodi sono indipendenti, ma in pratica tutto ciò è difficile da fare, quindi i nodi sono completamente indipendenti.

In un sistema distribuito, in quanto la comunicazione avviene tramite messaggi, non è presente la memoria condivisa e non si ha un clock globale del sistema, ma ogni nodo si gestisce attraverso un clock interno.



Si ha che un client fa una richiesta e il server risponde con un certo risultato (con il conseguente ritardo, a differenza del modello a chiamata di procedura).



Si può accedere a server multipli (cluster con anche bilanciamento del carico) e si può accedere via proxy (dei server "finti" che fungono da concentratori). Un sistema distribuito per comunicare effettua le seguenti operazioni:

1. **identifica la controparte**, attraverso l'assegnazione di un nome(*naming*)
2. **si accede alla controparte**, attraverso un punto di accesso
3. **si definisce il protocollo**, al fine di stabilire le regole e le procedure per essere in grado di comunicare, senza alcun problema.
4. **si definisce**, che si risolve concordando *sintassi e semantica* per l'informazione da condividere (**quest'ultimo è però ancora un problema aperto**)

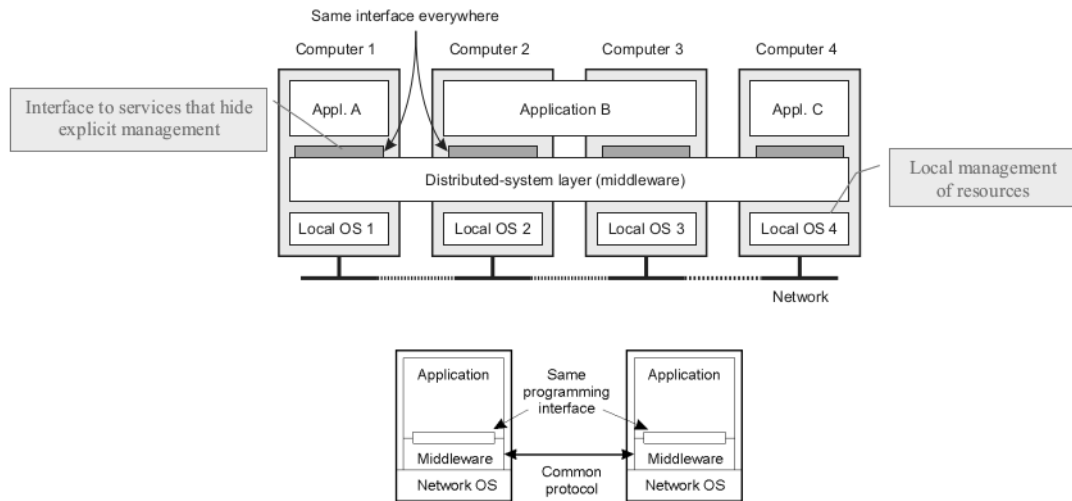
Si hanno le seguenti definizioni per quanto riguarda la trasparenza:

- **naming**, si usano nomi simbolici per identificare le risorse, facenti parte del sistema distribuito.
- **access transparency**, nascondere le differenze nella rappresentazione delle informazioni e nell'accesso ad un'informazione locale o remota
- **location transparency**, in cui si nasconde dove è collocata una risorsa sulla rete
- **relocation(mobility) transparency**, in cui si nasconde se la risorsa è stata trasferita ad un'altra locazione, mentre è in uso.
- **migration transparency**, in cui si nasconde che una risorsa può essere trasferita
- **replication transparency**, in cui si nasconde che una risorsa può essere replicata
- **concurrency transparency**, in cui si nasconde che una risorsa può essere condivisa da molti utenti indipendenti
- **failure transparency**, in cui si nascondono fallimenti e recovery di una risorsa
- **persistence transparency**, in cui si nasconde se una risorsa è volatile o memorizzata permanentemente

Da questa trasparenza non si è in grado di nascondere i ritardi e le latenze di comunicazione, ma soprattutto non si è in grado di effettuare una trasparenza completa, per motivi di performance e di latenza dell'operazioni su un sistema distribuito.

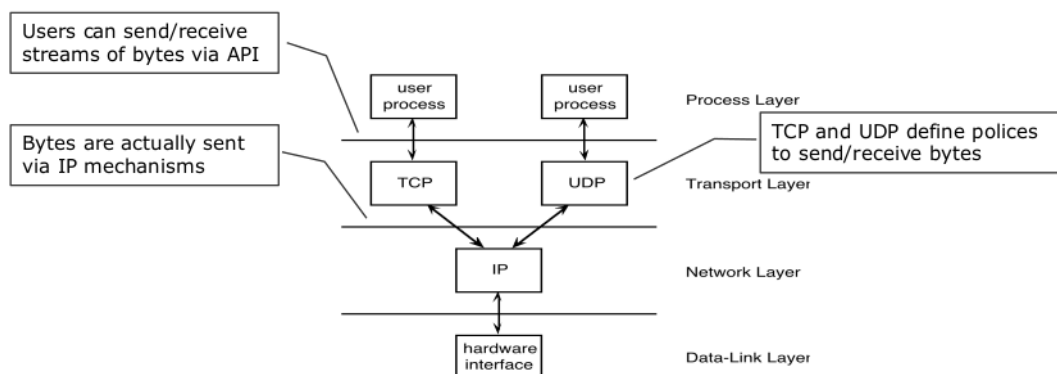
Questa volontà di astrarre le informazioni è alla base dell'ingegneria del software, in cui si separa il *cosa* dal *come*: il cosa si effettua tramite la

gura:interfaccia



definizione dell'interfaccia, complete ed indipendenti dalle diverse implementazioni, mentre il come avviene con l'effettiva implementazione delle classi e dei metodi.

Come si vede nella figura, l'interfaccia è unica mentre l'implementazione varia in ogni host locale del sistema distribuito e ciò può essere anche visto come la separazione tra un meccanismo e una politica, ossia come si implementa effettivamente una funzionalità del sistema.



Per poter capire le richieste ed eseguire i processi di comunicazione i due processi devono concordare un protocollo, in cui viene definito il formato, l'ordine di invio e di ricezione dei messaggi tra i diversi dispositivi, e per

vedere un esempio di una comunicazione tra i diversi processi si guardi il listato di codice , in cui si implementano sia l'header che l'implementazione del server mentre nel listato di codice si vede l'implementazione del client.

Non forniamo una spiegazione del codice dato che nel corso del corso impareremo come sviluppare ed implementare applicazioni client-server.

2.0.1 Stream Communication

Per la comunicazione tra due host si utilizza il modello ISO/OSI, basato sull'astrazione di cui tutti gli informatici ne dovrebbero conoscere in dettaglio tutti i vari livelli, in cui per mandare dei dati da un host ad un host si parte dal livello di applicazione, su cui si sviluppano ed operano i sistemi distribuiti, per poi andare ai livelli di trasporto, di rete e fisico, necessari per il trasferimento sulla rete delle informazioni, come si nota nella figura .

Ogni processo comunica attraverso canali, in cui vengono gestiti i flussi di dati in ingresso ed uscita, individuabili tramite un intero detto **porta** e noi studiamo le **socket**, particolare canale per la comunicazione in cui non vi è una condivisione della memoria e per potersi connettere da un processo A, il processo B deve conoscere l'host che esegue A e la porta in cui A è connesso.

Le socket possono essere principalmente di due tipi, come i principali protocolli di trasporti:

- **tcp socket**: utilizzano il protocollo TCP, orientato alla connessione, per la comunicazione tra i due processi, prevede un controllo di affidabilità dei messaggi, ossia viene assicurato che i messaggi arrivano nell'ordine previsto all'altro processo ed infine vi è un controllo di flusso e di congestione ma non si hanno garanzie di banda e dei ritardi.

Si utilizzano nelle applicazioni, in cui si deve avere la sicurezza dell'arrivo dei dati come ad esempio nel protocollo HTTP, per la comunicazione web, e nelle chat app come Telegram e Whatsapp.

- **udp socket**: utilizza il protocollo UDP, non affidabile e non orientato alla connessione, in cui viene solo garantito il trasferimento dei dati dalla rete all'applicazione e un controllo minimale degli errori, cosa che lo definisce come un sottoinsieme proprio del protocollo TCP. Viene utilizzato quando si deve avere un ritardo di trasmissione limitato ed una perdita minimale può non essere un problema, come ad esempio nei file multimediali e/o chiamate via voip.

Nei sistemi distribuiti non è necessario conoscere il funzionamento dei protocolli di trasporto ma basta considerare i servizi offerti e il fatto che si trasferiscono stream di byte, infatti le socket sono delle API(Application

stato:fileServer

```
// definizioni necessarie a client e server

#define TRUE 1
#define MAX_PATH 255 // lunghezza massima del nome di un file
#define BUF_SIZE 1024 // massima grandezza file trasferibili per volta
#define FILE_SERVER 243 // indirizzo di rete del file del server
167
// operazioni permesse

#define CREATE 1 // crea un nuovo file
#define READ 2 // legge il contenuto di un file e lo restituisce
#define WRITE 3 // scrive su un file
#define DELETE 4 // cancella un file
174
175 // errori
176
#define OK 0 // nessun errore
#define E_BAD_OPCODE -1 // operazione sconosciuta
#define E_BAD_PARAM -2 // errore in un parametro
#define E_IO -3 // errore del disco o errore di I/O
181
// definizione del messaggio
183
struct message{
    long source; // identità del mittente
    long dest; // identità del ricevente
    long opcode; // operazione richiesta
    long count; // numero di byte da trasferire
    long offset; // posizione sul file da cui far partire l'I/O
    long result; // risultato dell'operazione
    char name[MAX_PATH]; // nome del file
    char data[BUF_SIZE]; //informazione da leggere o scrivere
};

#include <header.h>

void main(void){
    struct message m1, m2; // messaggio in entrata e uscita
    int r; // risultato

    while(TRUE){ // il server è sempre in esecuzione
        receive(FILE_SERVER, &m1); // stato di wait in attesa di m1
        switch(m1.code){ // varî casi in base alla richiesta
            case CREATE:
                r = do_create(&m1, &m2);
                break;
            case CREATE:
                r = do_read(&m1, &m2);
                break;
            case CREATE:
```

stato:fileClient

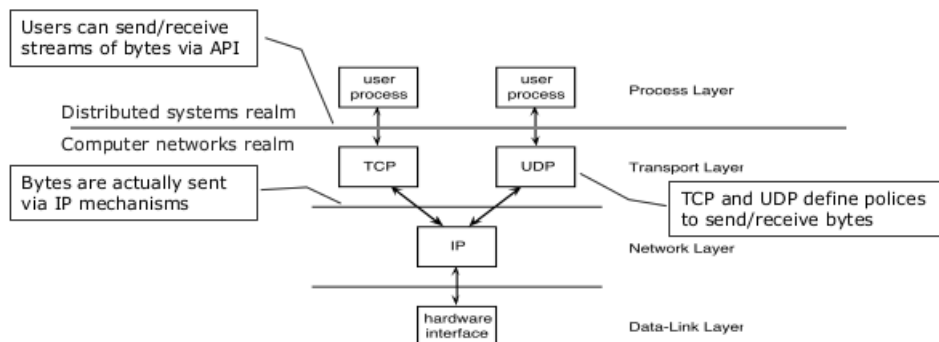
```
#include <header.h>

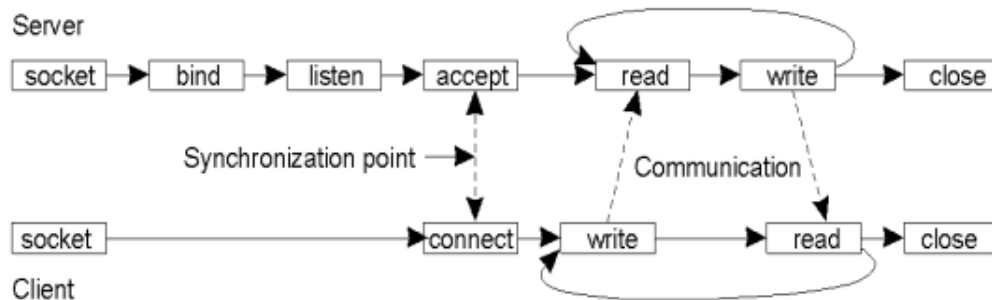
int copy(char *src, char *dst){ // copia file usando il server
    struct message m1; // buffer del messaggio
    long position; // attuale posizione del file
    long client = 110; // indirizzo del client

    initialize(); // prepara l'esecuzione
    position = 0;
    do{
        m1.opcode = READ; // operazione settata su READ
        m1.offset = position; // scelta la posizione nel file
        m1.count = BUF_SIZE; // byte da leggere
        strcpy(&m1.name, src); // nome file copiato in m1
        send(FILESERVER, &m1); // manda il messaggio al file server
        receive(client, &m1); // aspetta la risposta

        // scrive quanto ricevuto su un file di destinazione
        m1.opcode = WRITE; // operazione settata su WRITE
        m1.offset = position; // scelta la posizione nel file
        m1.count = BUF_SIZE; // byte da leggere
        strcpy(&m1.name, dst); // nome del file sul buffer
        send(FILESERVER, &m1); // manda il messaggio al file server
        receive(client, &m1); // aspetta la risposta
        position += m1.result // il risultato sono i byte scritti
    }while(m1.result > 0); // itera fino alla fine
    return(m1.result >= 0 ? OK : m1.result); // ritorna OK o l'errore
}
```

figure:livelliRete





Programming Interface) per accedere a TCP o UDP, in quanto due processi nel modello client-server comunicano mediante esso.

Si hanno delle criticità riguardanti alle socket e al modello client-server:

- gestione del ciclo di vita di cliente e server, attivazione/terminazione del cliente e del server
- identificazione e accesso al server
- comunicazione tra client e server
- ripartizione dei compiti tra client e server, che dipende dal tipo di applicazioni e la scelta influenza le prestazioni in relazione al carico

Quando viene definito l'indirizzo del server, esso può essere una costante, inserito dall'utente oppure inserendo un nameserver su cui si ricava l'indirizzo tramite il DNS(Domain Name Service) e questo comporta un basso livello di trasparenza dato che gli utenti devono conoscere l'indirizzo della rete e soprattutto parsare lo stream di byte nel message voluto.

La comunicazione TCP/IP avviene attraverso flussi di byte, dopo una connessione esplicita, attraverso normali system call read/write: queste due syscall sono sospensive, ossia mettono il sistema in attesa, e utilizzano un buffer per garantire la massima flessibilità, ad esempio la read definisce un buffer per leggere N caratteri ma potrebbe ritornare dopo aver letto solo $k < n$ caratteri.

Ci sono molte chiamate diverse per accedere i servizi TCP e UDP:

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

Per capire le operazioni e le procedure usate per permettere la comunicazione tra client e server, mostriamo l'implementazione client-server con udp, nel listato [2.1](#), e quella client-server tramite tcp, nel listato [2.2](#).

Nelle socket udp, protocollo in cui si implementa un sottoinsieme delle funzionalità delle tcp, si effettua la creazione della socket, sia nel client che nel server, e poi subito incomincia la comunicazione mentre nelle socket tcp si effettua prima un handshake a tre vie per settare la connessione tra client e server e poi incomincia la comunicazione.

Il server crea una nuova socket collegata (binded) a una nuova porta per comunicare con il client, in questo modo la well-known port resta dedicata a ricevere richieste di connessione:

Figura 2.1: Implementazione socket con protocollo udp

udpSocket

```
from socket import *

" Define address of the Server"
serverName = 'localhost'
serverPort = 12033

"Create a socket"
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = input("Insert a text: ")

"Send the message to the udp Server"
clientSocket.sendto(message.encode(), (serverName, serverPort))

"Receive the new message from the udpServer"
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)

print("We have receive " + modifiedMessage.decode() + " bytes")

"Close the udp Client socket"
clientSocket.close()

from socket import *

"Define the server Port"
serverPort = 12033
serverSocket = socket(AF_INET, SOCK_DGRAM) #Create a socket
serverSocket.bind(('', serverPort)) #Bind a socket to a port
print("Server is ready to receive")

#Loop where client can contact the server
while 1:
    " Receive a message from client Socket"
    message, clientAddress = serverSocket.recvfrom(2048)
    lengthMessage = str(len(message.decode()))
    "Send a message to a client socket"
    serverSocket.sendto(lengthMessage.encode(), clientAddress)
    "Close a socket "
    serverSocket.close()
serverSocket.close()
```

Figura 2.2: Implementazione socket con protocollo tcp

tcpSocket

```
import socket

"Define the server's address"
serverName = 'localhost'
serverPort = 12000

"Create a socket and try the connection to server address"
clientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientSocket.connect((serverName, serverPort))

message = input("Inserisci un testo: ")
"Send data to server"
clientSocket.sendto(message.encode(), (serverName, serverPort))

"Receive data from server socket"
modifiedMessage, address = clientSocket.recvfrom(2048)
print("I have receive from the server the text" + modifiedMessage.decode())

"Close the client socket"
clientSocket.close()

import socket

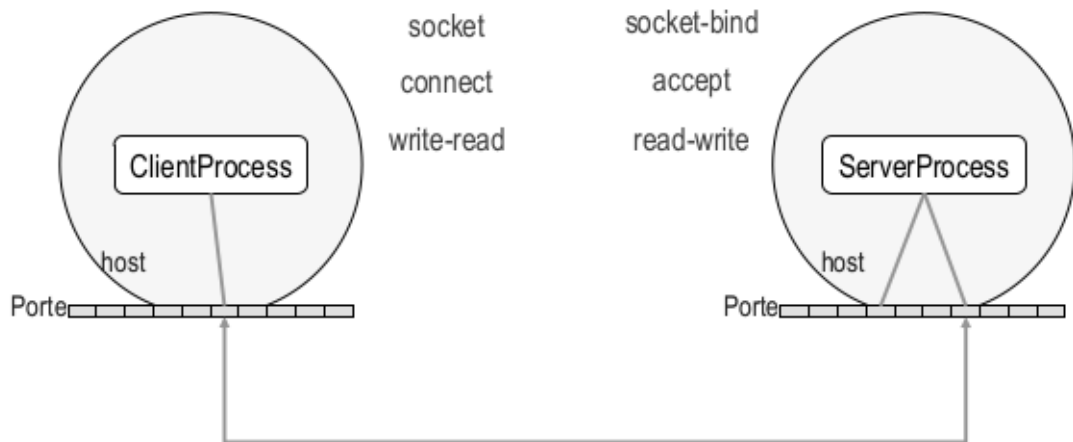
"Create a socket and bind to a port "
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serverSocket.bind(('localhost', 12000))

"Give the server able to accept at least 10 client connections"
serverSocket.listen(10)

while True:
    "Accept a socket and receive byte from clientSocket"
    (clientSocket, address) = serverSocket.accept()
    message, address = clientSocket.recvfrom(2048).decode()
    print address
    modifiedMessage = message.upper()

    "Send byte to a client socket"
    serverSocket.sendto(modifiedMessage.encode(), address)

    "Close the server Socket" 13
    serverSocket.close()
serverSocket.close()
```



```
byteLetti read(socket, buffer, dimBuffer);
```

con:

- byteLetti = byte effettivamente letti
- socket = canale da cui leggere
- buffer = spazio di memoria dove trasferire i byte letti
- dimBuffer = dimensione del buffer = numero max di caratteri che si possono leggere

Dopo aver mostrato come si implementano le socket in Python, per rendere semplice e facilmente comprensibile quali sono le funzionalità e come si definiscono le socket, analizziamo come vengono fatte con il linguaggio Java, usato nel corso, utilizzando alcune classi che costituiscono un'interfaccia ad oggetti delle system call, su cui sono definite tutte le operazioni delle socket:

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

Queste classi accorpano funzionalità e mascherano alcuni dettagli con il vantaggio di semplificarne l'uso e come ogni framework è necessario conoscerne il modello e il funzionamento per poterlo usare efficacemente.

I costrutti principali di queste due classi si possono trovare facilmente nella documentazione Java e negli esempi fatti a lezione, trovabili facilmente sul corso elearning.

Vediamo ora un esempio di un server ^{java:tcpServer} 2.3, scritto in Java che accetta una ^{java:tcpClient} connessione da un client e manda uno stream di dati, e di un client ^{2.4} che

Figura 2.3: Semplice implementazione TCP server in Java

java:tcpServer

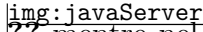
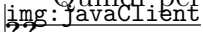
```
java.net.Socket  
java.net.ServerSocket
```

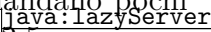
Figura 2.4: Semplice implementazione TCP client in Java

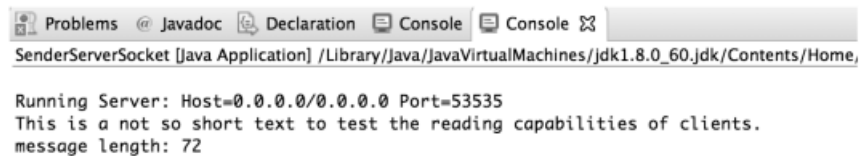
java:tcpClient

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

legge lo stream di bytes, con un esempio simile a quello fatto in python per spiegare le fasi del TCP socket.

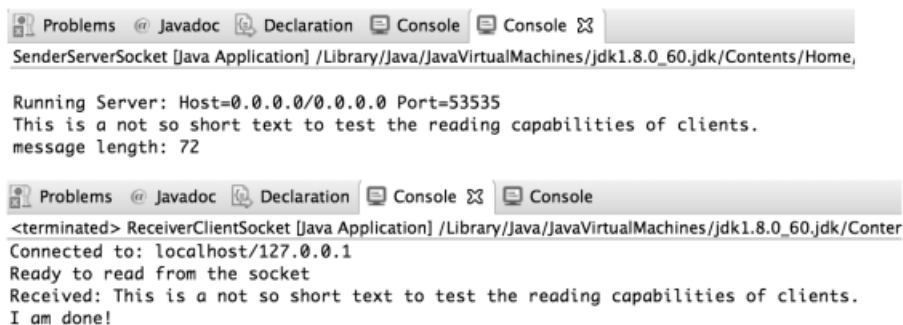
Quindi per il server si avrà la situazione  mentre nel client si verifica .

Si nota facilmente che il seguente modello di client non funziona nella maniera corretta, per cui si introducono i **lazy server** che mandano pochi byte per volta con un piccolo ritardo, come si nota nel listato  2.5



```
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
```

e per il client:



```
<terminated> ReceiverClientSocket [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Conter  
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!
```


Figura 2.5: Implementazione server Lazy

java:lazyServer

```
package serverWriter;

import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class LazySenderServerSocket {

    final static String message = "This is a not so short text to test the r
        + " If they are not so smart, they will catch only part
    final static int chunk = 9; // number of bytes sent every time

    public static void main(String[] args) {
        try {
            Socket clientSocket;
            ServerSocket listenSocket;

            listenSocket = new ServerSocket(53535);
            System.out.println("Running Server: " + "Host=" + listen
                + listenSocket.getLocalPort());

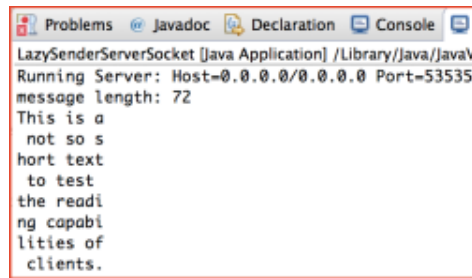
            while (true) {
                clientSocket = listenSocket.accept();

                PrintWriter out = new PrintWriter(clientSocket.g
                System.out.println("message length: " + message.

                int i;
                for (i = 0; i < message.length() - chunk; i +=
                    System.out.println(message.substring(i,
                    Thread.sleep(1000);
                    out.write(message.substring(i, i + chunk
                    out.flush();

                }
                System.out.println(message.substring(i, message.
                out.write(message.substring(i, message.length()
                out.flush();
                clientSocket.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```



```
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 readi
ng capabi
lities of
clients.
```

Per un'applicazione socket si ha le seguenti componenti:

1. **client** con l'architettura che è concettualmente più semplice di quella di un server, spesso è un'applicazione convenzionale in cui si hanno solo effetti sull'utente client e non comporta problemi di sicurezza.
2. **server**: crea una socket, gli assegna una porta nota ed entra in ciclo infinito in cui alternare: attesa di una richiesta, soddisfa la richiesta ed invio la risposta.
L'affidabilità di un server è strettamente dipendente dall'affidabilità della comunicazione tra lui e i suoi client, del resto però la modalità *connection-oriented* determina l'impossibilità di rilevare interruzioni sulle connessioni e la necessità di prevedere una connessione (una socket) per ogni comunicazione.

Ci sono diverse tipologie di server implementabili in un modello client-server:

- **iterativi**, in cui viene soddisfatta una richiesta alla volta
- **concorrenti processo singolo**, in cui viene simulata la presenza di un server dedicato
- **concorrenti multi-processo**, in cui vengono creati server dedicati ad ogni client.
- **concorrenti multi-thread**, in cui vengono creati dei thread specifici per ogni client.

Vediamo come 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. Si hanno degli svantaggi:

- viene servito un cliente alla volta, gli altri devono attendere

- un server impedisce l'evoluzione di molti client
- non scala

La soluzione sono i *server concorrenti*.

Vediamo un esempio di un server che semplicemente legge da una socket e scrive sulla console:

```
package SelectorExample;

import java.io.DataInputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class IterativeServer {

    public static void main(String[] args) {
        ServerSocket listenSocket;
        Socket clientSocket;
        byte[] byteReceived = new byte[1000];
        String messageString = "";

        try {
            listenSocket = new ServerSocket(53535);
            System.out.println("*** Running Server: " + "Host=" + listenSocket.getInetAddress()
                + listenSocket.getLocalPort());

            while (true) {
                clientSocket = listenSocket.accept();

                DataInputStream in = new DataInputStream(clientSocket.getInputStream());
                int bytesRead = 0;

                while (true) {
                    bytesRead = in.read(byteReceived);
                    if (bytesRead == -1)
                        break; // no more bytes
                    messageString += new String(byteReceived, 0, bytesRead);
                    System.out.println("Received: " + messageString);
                }

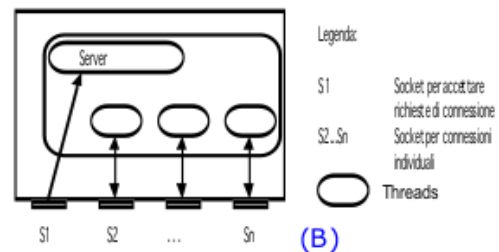
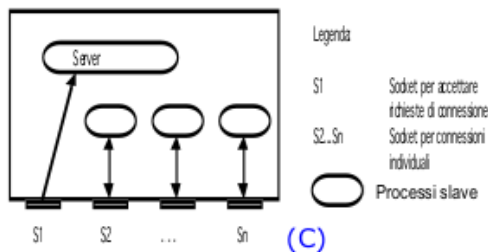
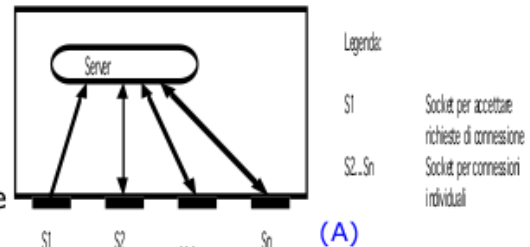
                clientSocket.close();
            }
        }
    }
}
```

```
        messageString = ""; // clear the string for the next client
        System.out.println("*** Available to the next client.");
    }
} catch (Exception e) {
    e.printStackTrace();
}
}
}
```

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 restituiscono i canali *ready*
(B) in Java: uso dei *Thread*
- reale creando nuovi processi slave
(C) in C: uso della funzione *fork*



In java si ha anche il **multiplexing** dove:

- i **channel** possono operare sia in modalità bloccante che non bloccante. In modalità non bloccante (dove solo canali stream-oriented, come socket e pipe, possono essere usati) il channel non mette mai il thread invocato in sleep. L'operazione richiesta o viene completata completamente o ritorna che nulla è stato fatto
- si hanno classi speciali per java, come *ServerSocketChannel*, *SocketChannel* e *DatagramChannel* e si usano i selector, permettendo un controllo più fine dei socket channels.

Un Selector è un multiplexor di oggetti *SelectableChannel* e viene creato con:

```
Selector selector = Selector.open();
```

I selector vengono poi registrati col metodo *register*:

```
channel.configureBlocking(false);  
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

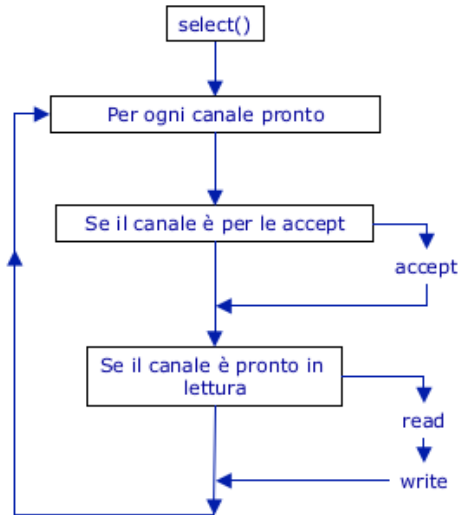
Con le seguenti *SelectionKey*:

SelectionKey.OP_CONNECT // quando un client tenta di connettersi al server
SelectionKey.OP_ACCEPT // quando il server accetta la connessione del client
SelectionKey.OP_READ // quando il server è pronto a leggere dal canale
SelectionKey.OP_WRITE // quando il server è pronto a scrivere sul canale

In generale ecco lo pseudo-codice per un server non bloccante:

```
create SocketChannel;  
create Selector;  
associate the SocketChannel with the Selector;  
while(true) {  
    waiting events from the Selector;  
    event arrived;  
    create keys;  
    for each key created by Selector {  
        check the type of request;  
        isAcceptable:  
            get the client SocketChannel;  
            associate that SocketChannel with the Selector;  
            record it for read/write operations  
            continue;  
        isReadable:  
            get the client SocketChannel;  
            read from the socket;  
            continue;  
        isWriteable:  
            get the client SocketChannel;  
            write on the socket;  
            continue;  
    }  
}
```

ovvero:



In java:

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.Socket;
import java.util.Scanner;

public class SenderClient {
    private Socket socket;
    private Scanner scanner;

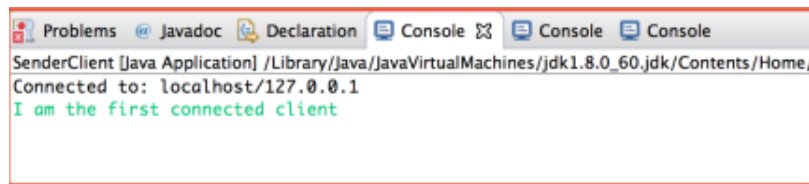
    private SenderClient(InetAddress serverAddress,
        int serverPort) throws Exception {
        this.socket = new Socket(serverAddress, serverPort);
        this.scanner = new Scanner(System.in);
    }

    private void start() throws IOException, InterruptedException {
        String input;
        PrintWriter out = new PrintWriter(this.socket.getOutputStream(), true);
        while (true) {
            input = scanner.nextLine();
            if (input.contentEquals("exit"))
                return;
            out.println(input);
        }
    }
}
```

```
        break;
    out.print(input);
    out.flush();
}
System.out.println("Client terminate.");
socket.close();
}

public static void main(String[] args) throws Exception {
    SenderClient client = new SenderClient(InetAddress.getByName(args[0]),
        Integer.parseInt(args[1]));

    System.out.println("Connected to: " + client.socket.getInetAddress());
    client.start();
}
}
```



Vediamo come farlo in modo concorrente:

```
package SelectorExample;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
import java.util.Iterator;
import java.util.Set;
```



```
public class ConcurrentServer {

    public static void main(String[] args) throws IOException {
        Selector selector = Selector.open();
        ServerSocketChannel server = ServerSocketChannel.open();
        server.bind(new InetSocketAddress("localhost", 53535));
        // set the channel in non blocking mode
        server.configureBlocking(false);
        // register the channel with the selector for the accept operation
        server.register(selector, SelectionKey.OP_ACCEPT);

        // Infinite server loop
        while (true) {
            // Waiting for events
            selector.select();
            // Get keys
            Set<SelectionKey> keys = selector.selectedKeys();
            Iterator<SelectionKey> i = keys.iterator();

            // For each keys...
            while (i.hasNext()) {
                SelectionKey key = (SelectionKey) i.next();

                // Remove the current key
                i.remove();

                if (key.isAcceptable()) // a client required a connection
                    acceptClientRequest(selector, server);

                if (key.isReadable()) // ready to read
                    readClientBytes(key);
            }
        }

        private static void acceptClientRequest(Selector selector,
            ServerSocketChannel server) throws IOException {
            // get client socket channel
            SocketChannel client = server.accept();
            // Non Blocking I/O
            client.configureBlocking(false);
        }
    }
}
```

```
// recording to the selector (reading)
client.register(selector, SelectionKey.OP_READ);
return;
}

private static void readClientBytes(SelectionKey key) throws IOException {
    SocketChannel client = (SocketChannel) key.channel();

    // Read byte coming from the client
    int BUFFER_SIZE = 256;
    ByteBuffer buffer = ByteBuffer.allocate(BUFFER_SIZE);
    try {
        if (client.read(buffer) == -1) {
            client.close();
            return;
        }
    } catch (Exception e) {
        // client is no longer active
        e.printStackTrace();
        client.close();
        return;
    }

    // Show bytes on the console
    buffer.flip(); // set the limit to the current position and then set
                  // the position to zero
    Charset charset = Charset.forName("UTF-8");
    CharsetDecoder decoder = charset.newDecoder();
    CharBuffer charBuffer = decoder.decode(buffer);
    int port = client.socket().getPort();
    System.out.println(port + ": " + charBuffer.toString());
    return;
}
}
```

```

Problems Javadoc Declaration Console Console Console
SenderClient [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/
Connected to: localhost/127.0.0.1
I am the first connected client

Problems Javadoc Declaration Console Console Console
SenderClient [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/
Connected to: localhost/127.0.0.1
I am the second connected client

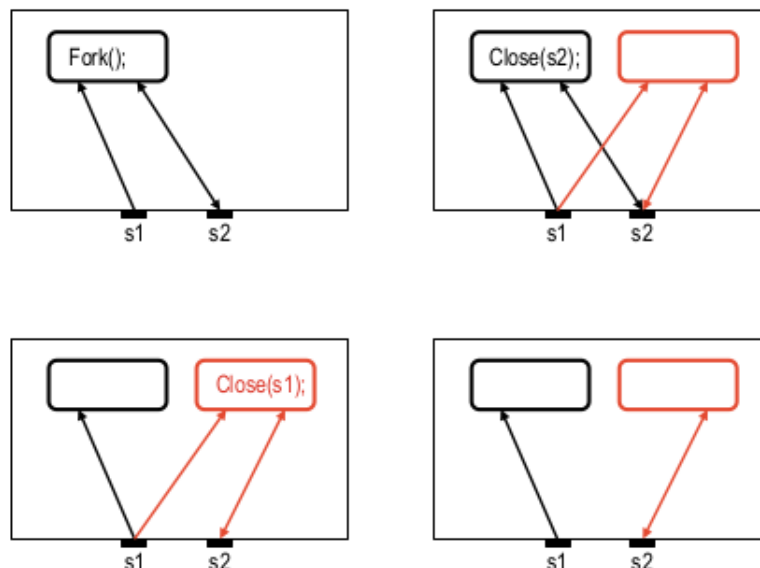
Problems Javadoc Declaration Console Console Console
ConcurrentServer [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/
63845: I am the first connected client
63846: I am the second connected client

Problems Javadoc Declaration Console Console Console
<terminated> SenderClient [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/
Connected to: localhost/127.0.0.1
I am the first connected client
exit
Client terminate.

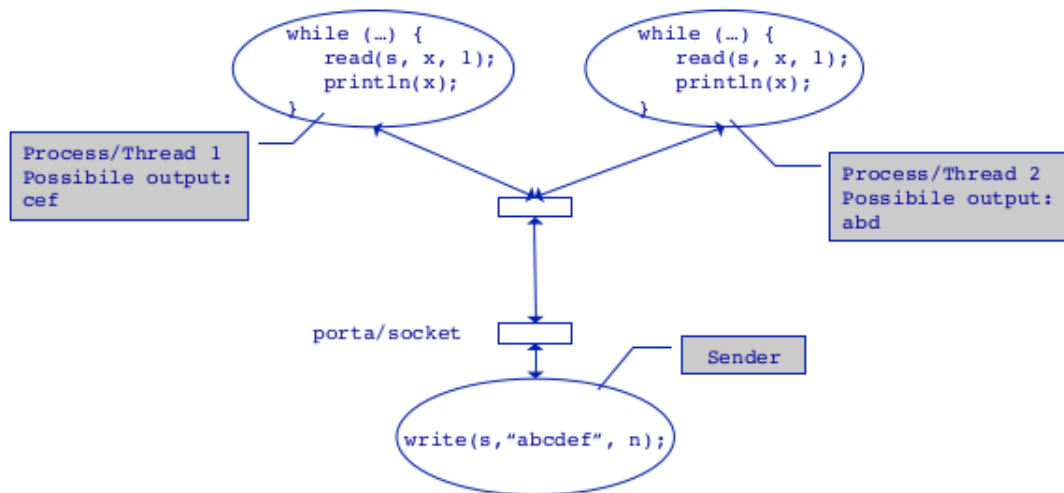
Problems Javadoc Declaration Console Console Console
SenderClient [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/
Connected to: localhost/127.0.0.1
I am the latest client

Problems Javadoc Declaration Console Console Console
ConcurrentServer [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/
63845: I am the first connected client
63846: I am the second connected client
63872: I am the latest client
    
```

Vediamo anche una rappresentazione della chiamata di sistema fork:



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



Si hanno quindi 2 modelli:

1. **mono processo (iterativo e concorrente)**, dove gli utenti condividono lo stesso spazio di lavoro. È adatto ad applicazioni cooperative che prevedono la modifica dello stato (lettura e scrittura)
2. **multi processo**, dove ogni utente ha uno spazio di lavoro autonomo. È adatto ad applicazioni autonome o che non modificano lo stato del server (sola lettura)

2.0.2 Architettura dei server

Per la gestione dei thread (che in java sono classi) esistono diversi **design pattern**:

- **un thread per pattern**, dove il *thread coordinatore* rileva la presenza di un nuovo client, lo connette ad un nuovo thread il quale:
 - decodifica la richiesta
 - chiama la funzione servente che la soddisfa
 - torna in ciclo per leggere una nuova richiesta

un thread per richiesta, dove un *thread coordinatore* riceve una richiesta e genera un thread per processarla. Questo nuovo thread:

- decodifica la richiesta
- chiama la funzione servente che la soddisfa
- termina

- **un thread per servente**, dove ogni servente ha un proprio thread e una coda. Il coordinatore riceve una richiesta e lo inserisce nella coda del servente giusto. Ogni thread servente legge ciclicamente una richiesta dalla propria coda e la esegue
- **una pool di thread**, dove, dato che la creazione di un thread è costosa, il costo viene ammortizzato facendo gestire ad ogni thread molte richieste. Questo *pool di thread* viene creato all'avvio del sistema e le richieste gli vengono assegnate man mano.

2.0.3 Un esempio

Vediamo un esempio client/server con socket TCP/IP in Java.

Vogliamo realizzare un semplice programma per giocare a “knock knock”, il popolare gioco di parole inglese che si basa su domande e risposte con parole ed espressioni omofone di significato differente:

Server: “Knock knock!”

Client: “Who’s there?”

Server: “Atch.”

Client: “Atch who?”

Server: “Bless you!”

Facciamo quindi tre classi (*qui il ruolo attivo viene assunto dal server che inizia la conversazione (in pratica un’inversione dei ruoli)*):

1. una **classe *KnockKnockProtocol*** che fornisce il protocollo, stabilendo domande e risposte e funzione la soluzione per formulare le risposte
2. una **classe *KnockKnockClient*** che stabilisce la connessione e invia i messaggi al server
3. una **classe *KnockKnockServer*** che accetta la connessione e interroga il client secondo il protocollo della prima classe

Vediamo la *KnockKnockServer*:

```
package KnockKnock;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class KnockKnockServer {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(4444);
        } catch (IOException e) {
            e.printStackTrace();
            System.err.println("Could not listen on port: 4444.");
            System.exit(1);
        }
        Socket clientSocket = null;
        try {
            clientSocket = serverSocket.accept();
        } catch (IOException e) {
            System.err.println("Accept failed.");
            System.exit(1);
        }
        PrintWriter out =
            new PrintWriter(clientSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
            new InputStreamReader(
                clientSocket.getInputStream()));
        String inputLine, outputLine;
        KnockKnockProtocol kkp = new KnockKnockProtocol();

        outputLine = kkp.processInput(null);
        out.println(outputLine);

        while ((inputLine = in.readLine()) != null) {
            outputLine = kkp.processInput(inputLine);
            out.println(outputLine);
        }
    }
}
```

```
        if (outputLine.equals("Bye."))
            break;
    }
    out.close();
    in.close();
    clientSocket.close();
    serverSocket.close();
}
}
```

passiamo al client:

```
package KnockKnock;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.UnknownHostException;

public class KnockKnockClient {

    public static void main(String[] args) throws IOException {
        Socket kkSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;

        try {package KnockKnock;

public class KnockKnockProtocol {
    private static final int WAITING = 0;
    private static final int SENTKNOCKKNOCK = 1;
    private static final int SENTCLUE = 2;
    private static final int ANOTHER = 3;

    private static final int NUMJOKES = 5;

    private int state = WAITING;
    private int currentJoke = 0;

    private String[] clues = { "Turnip", "Little Old Lady", "Atch", "Who", "Who"
```

```

private String[] answers = { "Turnip the heat, it's cold in here!",
                              "I didn't know you could yodel!",
                              "Bless you!",
                              "Is there an owl in here?",
                              "Is there an echo in here?" };

public String processInput(String theInput) {
    String theOutput = null;

    if (state == WAITING) {
        theOutput = "Knock! Knock!";
        state = SENTKNOCKKNOCK;
    } else if (state == SENTKNOCKKNOCK) {
        if (theInput.equalsIgnoreCase("Who's there?")) {
            theOutput = clues[currentJoke];
            state = SENTCLUE;
        } else {
            theOutput = "You're supposed to say \"Who's there?! \" +
~I~I~I~I    "Try again. Knock! Knock!";
        }
    } else if (state == SENTCLUE) {
        if (theInput.equalsIgnoreCase(clues[currentJoke] + " who?")) {
            theOutput = answers[currentJoke] + " Want another? (y/n)";
            state = ANOTHER;
        } else {
            theOutput = "You're supposed to say \"" +
~I~I~I~I    clues[currentJoke] +
~I~I~I~I    " who?\"" +
~I~I~I~I    "!! Try again. Knock! Knock!";
            state = SENTKNOCKKNOCK;
        }
    } else if (state == ANOTHER) {
        if (theInput.equalsIgnoreCase("y")) {
            theOutput = "Knock! Knock!";
            if (currentJoke == (NUMJOKES - 1))
                currentJoke = 0;
            else
                currentJoke++;
            state = SENTKNOCKKNOCK;
        } else {
            theOutput = "Bye.";
        }
    }
}

```



```
        state = WAITING;
    }
}
return theOutput;
}

}

    kkSocket = new Socket("localhost", 4444);
    out = new PrintWriter(kkSocket.getOutputStream(), true);
    in = new BufferedReader(new InputStreamReader(kkSocket.getInputStream()));
} catch (UnknownHostException e) {
    System.err.println("Don't know about host: taranis.");
    System.exit(1);
} catch (IOException e) {
    System.err.println("Couldn't get I/O for the connection to: taranis.");
    System.exit(1);
}

BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
String fromServer;
String fromUser;

while ((fromServer = in.readLine()) != null) {
    System.out.println("Server: " + fromServer);
    if (fromServer.equals("Bye."))
        break;

    fromUser = stdIn.readLine();
    if (fromUser != null) {
        System.out.println("Client: " + fromUser);
        out.println(fromUser);
    }
}

out.close();
in.close();
stdIn.close();
kkSocket.close();
}
```

}

e il server:

```
package KnockKnock;

public class KnockKnockProtocol {
    private static final int WAITING = 0;
    private static final int SENTKNOCKKNOCK = 1;
    private static final int SENTCLUE = 2;
    private static final int ANOTHER = 3;

    private static final int NUMJOKES = 5;

    private int state = WAITING;
    private int currentJoke = 0;

    private String[] clues = { "Turnip", "Little Old Lady",
        "Atch", "Who", "Who" };
    private String[] answers = { "Turnip the heat, it's cold in here!",
        "I didn't know you could yodel!",
        "Bless you!",
        "Is there an owl in here?",
        "Is there an echo in here?" };

    public String processInput(String theInput) {
        String theOutput = null;

        if (state == WAITING) {
            theOutput = "Knock! Knock!";
            state = SENTKNOCKKNOCK;
        } else if (state == SENTKNOCKKNOCK) {
            if (theInput.equalsIgnoreCase("Who's there?")) {
                theOutput = clues[currentJoke];
                state = SENTCLUE;
            } else {
                theOutput = "You're supposed to say \"Who's there?\"! " +
                    "Try again. Knock! Knock!";
            }
        } else if (state == SENTCLUE) {
            if (theInput.equalsIgnoreCase(clues[currentJoke] + " who?")) {
                theOutput = answers[currentJoke] + " Want another? (y/n)";
            }
        }
    }
}
```

```
        state = ANOTHER;
    } else {
        theOutput = "You're supposed to say \"" +
clues[currentJoke] +
" who?\"" +
"! Try again. Knock! Knock!";
        state = SENTKNOCKKNOCK;
    }
} else if (state == ANOTHER) {
    if (theInput.equalsIgnoreCase("y")) {
        theOutput = "Knock! Knock!";
        if (currentJoke == (NUMJOKES - 1))
            currentJoke = 0;
        else
            currentJoke++;
        state = SENTKNOCKKNOCK;
    } else {
        theOutput = "Bye.";
        state = WAITING;
    }
}
return theOutput;
}
}
```

I vari client sono quindi serviti in sequenza, con i conseguenti problemi di performance. Per servire più client in modo concorrente si usano i server multi-thread:

```
while (true) {
    accept a connection ;
    create a thread to deal with the client ;
end while
```

Si hanno due classi:

1. **KKMultiServer** per i server
2. **KKMultiServerThread** che realizza un server dedicato ad un client

```
package KnockKnock;
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class KKMultiServer {

    public static class KKMultiServerThread extends Thread {
        private Socket socket = null;

        public KKMultiServerThread(Socket socket) {
            super("KKMultiServerThread");
            this.socket = socket;
        }

        public void run() {
            try {
                PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
                BufferedReader in =
                    new BufferedReader(new InputStreamReader(socket.getInputStream()));
                String inputLine, outputLine;
                KnockKnockProtocol kkp = new KnockKnockProtocol();
                outputLine = kkp.processInput(null);
                out.println(outputLine);
                while ((inputLine = in.readLine()) != null) {
                    outputLine = kkp.processInput(inputLine);
                    out.println(outputLine);
                    if (outputLine.equals("Bye"))
                        break;
                }
                out.close();
                in.close();
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
public static void main(String[] args) {
    ServerSocket serverSocket = null;
    boolean listening = true;

    try {
        serverSocket = new ServerSocket(4444);

        while (listening)
            new KKMultiserverThread(serverSocket.accept()).start();
        serverSocket.close();
    } catch (IOException e) {
        System.err.println("Could not listen on port: 4444.");
        System.exit(-1);
    }
}
```

2.1 L'architettura del web

Per comunicare attraverso internet si utilizza un modello client-server usando principalmente il protocollo **HTTP**, attraverso l'esecuzione delle sue operazioni **request** e **response**: la prima indica la richiesta di un oggetto web, come ad esempio un'immagine e un file html, da parte del client verso il server mentre la seconda è la risposta da parte del server verso il client.

Nell'architettura di Internet il client viene realizzato mediante un browser, come ad esempio firefox, programma che fornisce la possibilità di navigare sul web, attraverso l'interpretazione del codice con cui sono espresse le pagine web, costituita da diversi oggetti identificati da un URL, mentre il server viene fornito da un Web Server, come ad esempio Apache.

L'URL identifica un oggetto nella rete e specifica come interpretare i dati ricevuti attraverso il protocollo, è formato dai seguenti elementi:

- nome del protocollo
- indirizzo IP dell'host
- porta del processo
- cammino/percorso dell'host

- identificatore della risposta

rappresentata nel seguente modo

`protocollo://indirizzo_IP[:porta]/cammino/risorsa`

La parte testuale dei documenti viene espressa da HTML, per contenuti ed impaginazione, da CSS per il rendering grafico mentre attraverso XML e JSON specifichiamo i dati e la loro struttura nel documento.

Si possono avere anche dati multimediali (foto, audio, etc...) con l'encoding MIME per definirne il formato (plain, html per i testi, jpeg, gif per le immagini, etc..) mentre la dinamicità delle pagine web viene data da linguaggi di programmazione come Javascript, VBScript, Java/applet

Vi sono diversi protocolli web, in cui si definiscono le regole di comunicazioni tra varie tipologie di applicazioni, come ad esempio HTTP, FTP(File Transfer Protocol) e SMTP(Simple Mail Transfer Protocol); questi esempi di protocolli utilizzano il protocollo di trasporto TCP ed utilizzano delle porte note: 80 per HTTP, 20 per FTP e 25 per SMTP.

Per ora si hanno due versioni del protocollo HTTP, definite in maniera standard:

1. **http1.0: RFC 1945**

2. **http1.1: RFC 2068**

HTTP è **stateless** in quanto il server non mantiene informazioni sulle precedenti richieste, per cui si devono sempre fornire le informazioni necessarie per cui i siti web per avere informazioni utilizzano i cookie, che analizzeremo alla fine di questo paragrafo.

Si ha che le *requeste response* hanno la stessa struttura, in ASCII con un formato testo leggibile, ad esempio:

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg
Accept-language: fr
```

Con la prima riga si rappresenta la request line mentre nelle altre sono rappresentati l'header con le opzioni.

Nel protocollo HTTP si hanno diverse tipologie di metodi e richieste, le principali sono le seguenti:

Figura 2.6: Tipologie di richieste HTTP

fig:httpMethod

		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			✓

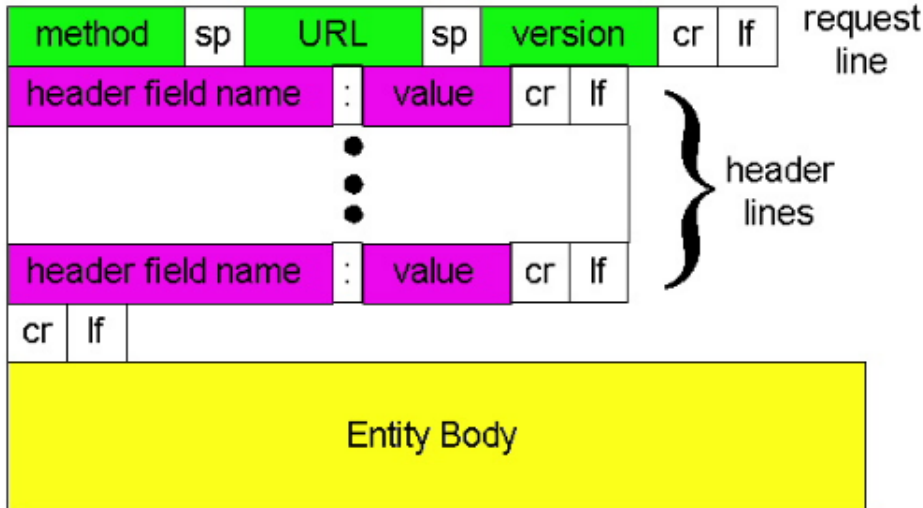
- **GET:** metodo HTTP, in cui viene restituita una rappresentazione di una risorsa web, senza effetti sul server, utile per ottenere pagine html ed immagini.
- **POST:** metodo HTTP, in cui vengono comunicati i dati da far elaborare al server oppure si crea una nuova risorsa, subordinata all'URL. Ogni richiesta POST causa degli effetti sul server, quindi non può venire gestita la richiesta da una cache e l'utilizzo tipico è quello per processare delle FORM html e di modificare dati presenti in un database.
- **HEAD:** viene utilizzato spesso in fase di debugging ed è simile al metodo get ma in questo metodo viene restituito soltanto l'head della pagina web.

Nel complesso tutte le tipologie di richieste HTTP si notano nella figura 2.6 mentre il formato di una richiesta http si vede nella figura 2.7. vediamo ora un esempio di risposta http:

```
HTTP/1.1 200 OK
Connection: close
Date: Thu, 06 Aug 1998 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998
...
Content-Length: 6821
```

Figura 2.7: Formato di una richiesta HTTP

fig:httpStructure



```
Content-Type: text/html
data data data data data ...
```

In molte applicazioni web il client e il server comunicano per un periodo esteso, in cui il client effettua una serie di richieste a cui il server risponde, in maniera intermittente oppure ogni tot periodo di tempo, per cui lo sviluppatore dell'applicazione deve decidere quale interazione deve avvenire tra ogni richiesta, ossia se usare la stessa connessione TCP oppure crearne sempre una nuova; la scelta comporta due diverse tipologie di utilizzo del protocollo HTTP, che si può estendere anche ai altri protocollo di livello applicativo:

- **connessione non persistente:** quando il server manda l'oggetto richiesto viene chiusa la connessione TCP, quindi successive interazioni tra lo stesso client e server richiedono la creazione e la chiusura di una nuova connessione TCP, con un aggravio di RTT, il tempo per mandare un pacchetto da client e server, per ogni richiesta solo per stabilire una nuova connessione TCP tra i due soggetti della comunicazione.
- **connessione persistente:** modalità usata di default dal protocollo HTTP, ma non per forza dagli altri protocolli, in cui al termine di una richiesta HTTP viene lasciata aperta la connessione TCP tra client e server, per cui ogni successiva richiesta tra essi non richiede la creazione di una nuova connessione, con conseguente maggiore efficienza e velocità del sistema.

Ovviamente la connessione TCP non viene lasciata aperta all'infinita,

Figura 2.8: HTTP Header codes

`http:headerCode`

200 OK

- Successo, oggetto richiesto più avanti nel messaggio

301 Moved Permanently

- L'oggetto richiesto è stato spostato. Il nuovo indirizzo è specificato più avanti (Location:)

400 Bad Request

- Richiesta incomprensibile al server

404 Not Found

- Il documento non è stato trovato sul server

505 HTTP Version Not Supported

in quanto tipicamente un server HTTP in caso di mancato utilizzo dopo un certo ammontare di tempo la chiude, per raggiungere una maggiore efficienza di memoria occupata.

Esistendo due diverse versioni del protocollo HTTP vi sono due diverse tipologie di client:

- Client HTTP 1.0: Server chiude connessione al termine della richiesta
- Client HTTP 1.1: mantiene aperta la connessione oppure chiude se la richiesta e quindi contiene `Connection: close`, al fine di poter fare una connessione non persistente.

nella figura [http:headerCode](#) 2.8 alcuni esempi di codici, usati per stabilire il tipo di risposta effettuata da HTTP ed è importante saperli, per capire cosa è avvenuto quando riceviamo, come client, la risposta HTTP.


Nonostante il protocollo HTTP è di tipo stateless, per i siti web è comodo poter identificare una persona, al fine di poter monitorare le abitudini, limitare l'accesso e personalizzare i contenuti, per cui sono stati introdotti i **cookies**, usati di default da tutti i siti.

I cookie prevedono 4 componenti: una header line nel messaggio di risposta HTTP, un header line nel messaggio di richiesta HTTP, un file di cookie tenuto nel sistema dell'utente e utilizzato dal browser, infine un database back-end nel sito web.

Per capire come funzionano i cookie guarda la figura [fig:cookie](#) 2.9, in cui si suppone che Susan accede al sito di Amazon per la prima volta, utilizzando nel header `Set - cookie : 1678`.

Nonostante i cookie semplificano le procedure di riconoscimento dell'utente,

come ad esempio si ha la possibilità di usare un'applicazione e-mail senza doversi ogni volta registrare, ci sono alcuni aspetti negativi riguardo alla privacy, dato che tramite i cookies ed informazioni ottenute sull'account di un utente, un sito web è in grado di sapere molto sull'utente e può vendere a terze parti, cosa che un utente vorrebbe sicuramente evitare.

La web cache, chiamata anche server proxy, è un rete in grado di soddisfare le richieste per conto di un web server e possiede un suo storage sul disco, per tenere le copie degli oggetti richiesti recentemente, per cui si può configurare, come si nota nella figura  fig:cache, il browser affinché ogni richiesta HTTP venga dirottata alla web cache, in cui in caso vi sia già una copia dell'oggetto viene subito mandata al browser, senza contattare il web server, altrimenti la web cache effettua una HTTP request al web server e dopo aver ottenuto l'oggetto lo salva internamente e lo manda, tramite una HTTP response, al browser che lo ha richiesto; come si può notare la web cache agisce sia da client che da server, infatti quando interagisce con il browser è un server mentre quando comunica con il web server agisce come un client e solitamente viene acquistata ed installata da un provider ISP.

La cache ha avuto un notevole utilizzo nel campo di internet per due ragioni:

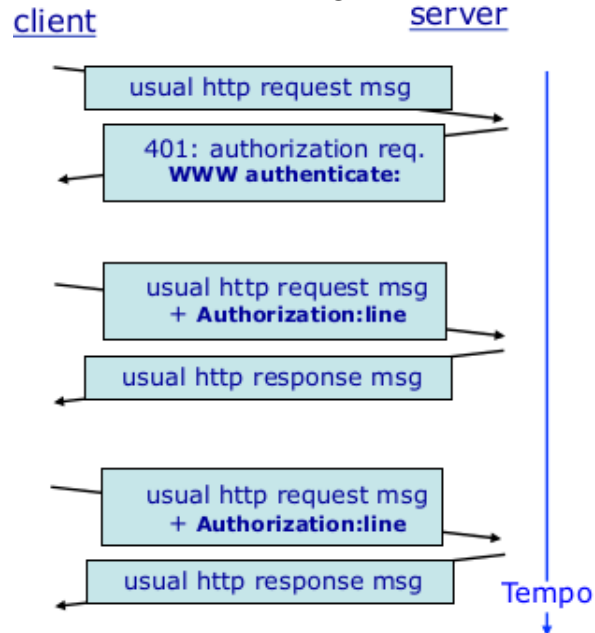
- può ridurre il tempo di risposta di una richiesta client, in particolare se il collegamento tra client e web cache possiede una banda più potente del collegamento tra client e server e solitamente ciò avviene, e se si ha un alto tasso di possesso dell'oggetto richiesto da parte della cache, al fine di evitare continue richieste al web server.
- riduce il traffico sul link di accesso ad internet di una compagnia e/o istituzione, con la possibilità di evitare un upgrade della banda, con notevoli risparmi di costi.

Questa riduzione del traffico fornisce un guadagno anche agli utilizzatori delle applicazioni web dato che vi è un miglioramento delle prestazioni.

Nonostante la cache riduca il tempo di risposta, introduce il problema sulla integrità della copia dell'oggetto rispetto a quella nel web server ma per risolvere il protocollo prevede un meccanismo, chiamato **conditional GET**, che permette alla cache di verificare se l'oggetto presente nel suo storage interno è aggiornato.

Un messaggio HTTP request prevede questo meccanismo in caso il messaggio usa il metodo GET ed include l'header **If-modified-since**, per cui la cache manda l'oggetto richiesto in caso in cui l'header if-modified-since coincide con il valore del header last-modified.

Figura 2.9: HTTP authentication



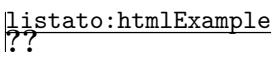
Per gestire l'accesso ai documenti sul server, dato che http è stateless, si deve verificare ogni richiesta e le informazioni necessarie all'autenticazione si trovano nell'header (*authorization: line*) senza le quali il server rifiuta la connessione (*www authenticate:*), come si nota nella figura ??.

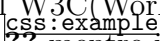
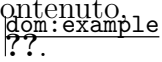
Capitolo 3

HTML + CSS + JS

Il linguaggio **HTML** è un linguaggio di markup, per dare struttura ai contenuti web, utilizzato per annotare un documento in maniera tale che l'annotazione sia sintatticamente distinguibile dal testo per diverse finalità:

- di presentazione, in cui si definisce come visualizzare il testo al quale sono associate.
- procedurali, in cui si definiscono istruzioni per programmi che elaborino il testo associato.
- descrittive, in cui si etichettano semplicemente parti del testo, al fine di disaccoppiare la struttura dalla presentazione del testo spesso.

Un esempio di una pagina html si nota nella figura 

Il **CSS**(Cascading Style Sheets) è un linguaggio per dare uno stile ai contenuti web e la sua specifica viene definita dal W3C(World Wide Web Consortium), un suo esempio si trova nella figura  mentre il DOM(Document Object Model) è un'interfaccia neutrale rispetto al linguaggio e alla piattaforma usata al fine di consentire l'accesso e la modifica dinamica di contenuto, struttura e lo stile di un documento web, come si nota nella figura .

Ogni nodo può essere caratterizzato da attributi per facilitarne l'identificazione, la ricerca, la selezione:

- un **identificatore univoco**, anche se il DOM non garantisce l'unicità
- una **classe** che indica l'appartenenza ad un insieme che ci è utile definire

Le **media query** possono essere viste come particolari selettori capaci di valutare le capacità del device di accesso alla pagina (schermi, stampanti, text-to-speech), controllando le dimensioni del device o della finestra, l'orientamento

Figura 3.1: Html example

listato:htmlExamp

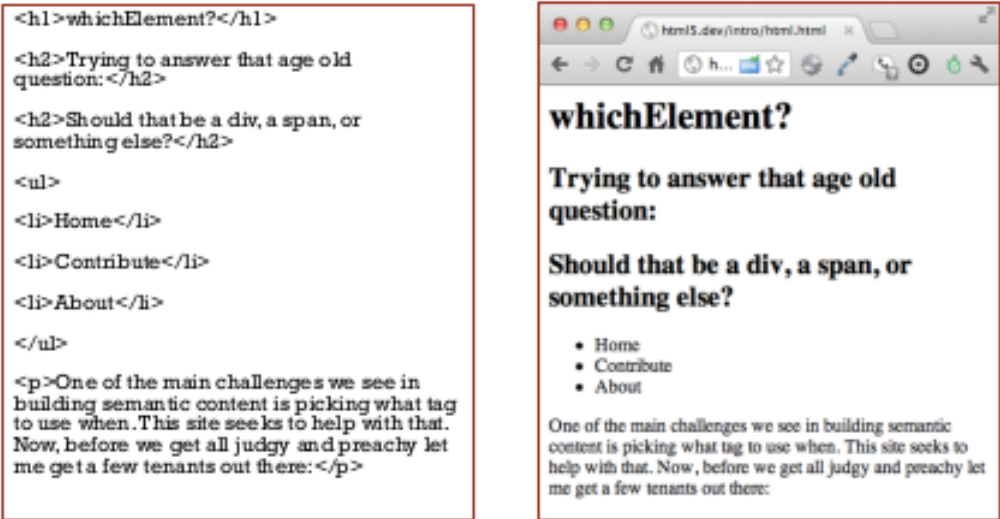


Figura 3.2: Css example

css:example

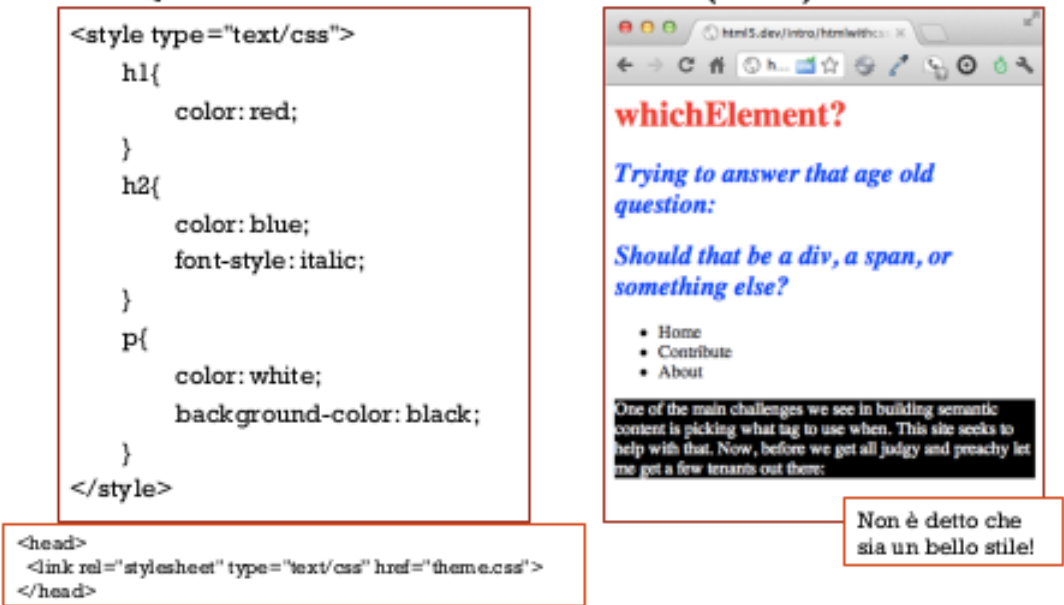
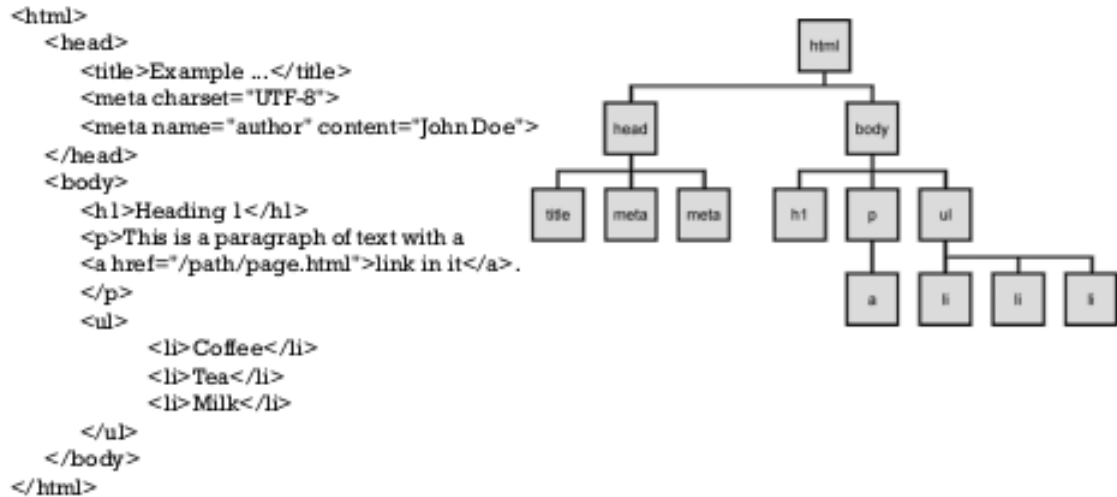


Figura 3.3: Dom example

dom:example



dello schermo e la risoluzione. Vediamo un esempio:

se la pagina è più larga di 480 pixel (e si sta visualizzando sullo schermo), applica determinati stili agli elementi con id "leftsidebar" (un menu) e "main" (la colonna centrale):

```

@media screen and (min-width: 480px){
  #leftsidebar {width: 200px; float: left;}
  #main {margin-left:216px;}
}

```

In CSS si ha il termine *cascading* perché esistono potenzialmente diversi stylesheet:

- l'autore della pagina in genere ne specifica uno (il modo più comunemente inteso) o più d'uno
- il browser ne ha uno, o un vero e proprio CSS o simulato nel loro codice
- il lettore, l'utente del browser, ne può definire uno proprio per customizzare la propria esperienza

Dei conflitti sono quindi inevitabili per cui è necessario definire un algoritmo per decidere quale stile vada applicato a un elemento.

Si ha il seguente ordine di importanza dei seguenti fattori associati alle regole:

1. **importanza** (flag specifico *!important* per un attributo)

2. **specificità** (per esempio, *id* > *class* > *tag*)
3. **ordine nel sorgente** (il più “recente” vince)

Il browser non è solamente un banale visualizzatore di pagine scritte in HTML, è un vero e proprio ambiente di sviluppo (in particolare contiene un interprete Javascript e vari strumenti di debug, ma ne parleremo più avanti) che fornisce numerose funzionalità abilitanti inoltre l'impostazione di HTML e CSS separa nettamente il contenuto dalla modalità di visualizzazione.

Esistono numerosi “front-end framework”, dai più sofisticati ai più semplici, naturalmente open source, ad esempio:

- **bootstrap**
- **foundation**
- **skeleton**

3.0.1 HTML5

HTML5 ha introdotto nuovi **elementi semantici** per la strutturazione delle pagine, per esempio:

- `article`
- `section`
- `aside`
- `header`
- `footer`

Introduce inoltre nuovi **elementi di input e multimediali**, widget per input search, email, url, number, tel, ma anche range, date ..., anche se il supporto a tutto ciò da parte dei browser non è uniforme.

L'HTML dovrebbe non contenere informazione di presentazione, riservata ai CSS, quindi senza stili definiti “in linea” e senza l'uso di tag come ``, ``, `<i>` ma dovrebbe solo occuparsi delle informazioni da volere rappresentare nel file web.

HTML5 sia più leggibile. In generale, a parte essere una notazione più concisa e che richiede meno definizioni di classi, le gerarchie di contenuti più leggibili e analizzabili in fase di progettazione, manutenzione e debug.

Per avere informazioni riguardo alla sintassi html e css si consiglia di guardare i tutorial presenti in internet su W3CSchool.