Sistemi distribuiti

UniShare

Davide Cozzi
@dlcgold

Gabriele De Rosa
@derogab

Federica Di Lauro
@f_dila

Indice

Capitolo 1

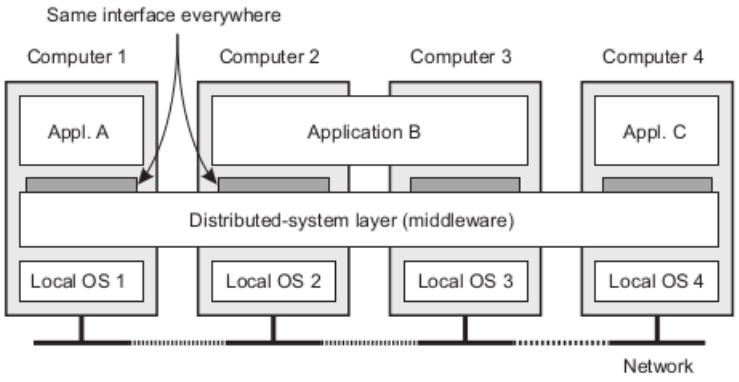
Introduzione

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/dlcgold/Appunti`.
Grazie mille e buono studio!

Capitolo 2

Introduzione ai Sistemi Distribuiti

Un sistema distribuito è un sistema nel quale componenti hardware e software, collocati in computer connessi alla rete, comunicano e coordinano le loro azione solitamente col passaggio di messaggi (a differenza delle chiamate di procedura che si hanno col passaggio di parametri su memoria condivisa). Ogni processo ha quindi una parte di logica applicativa e una parte di coordinamento. Altrimenti si ha questa definizione. *un sistema distribuito è un insieme di elementi autonomi di computazione che si interfacciano agli utenti come un singolo sistema "coerente".*



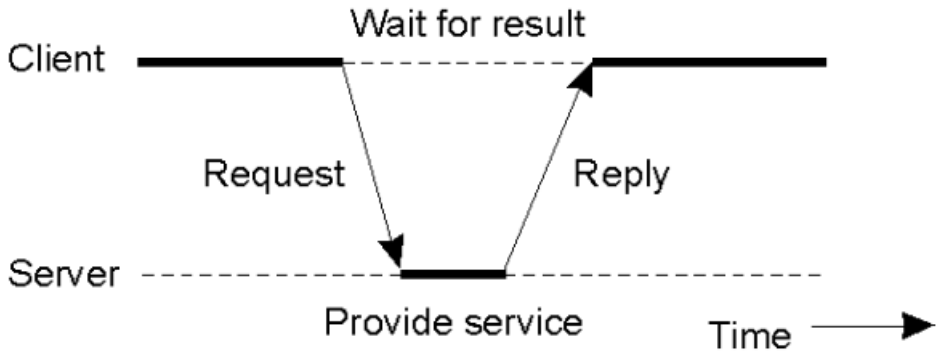
I sistemi distribuiti sono quindi sistemi complessi. Si hanno le seguenti caratteristiche:

- le unità autonome di computazione si chiamano **nodi** e possono essere device hardware o singoli processi software

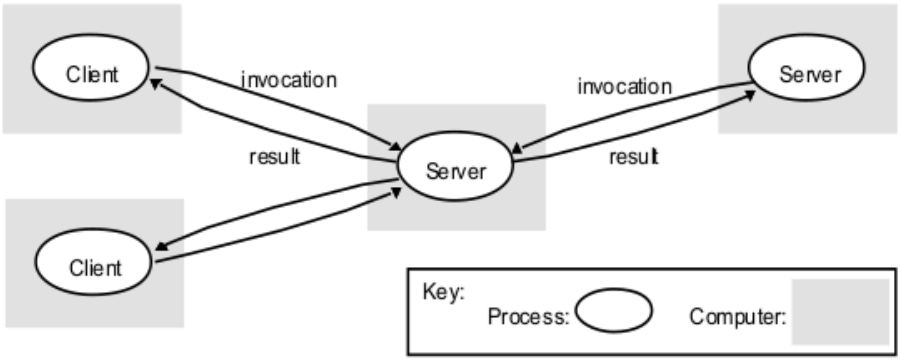
- ogni nodo "fa quello che vuole", ogni nodo è autonomo, e vanno tra loro sincronizzati e coordinati (programmazione concorrente). Ogni nodo ha la sua "nozione di tempo"
- utenti e applicazioni vedono un singolo sistema
- si possono aprire e chiudere gruppi di nodi

La parola chiave è **trasparenza di distribuzione (distribution trasparency)**. Trasparenza significa nascondere dettagli agli utenti che possono ignorare ciò che succede e che non possono modificare il servizio. Si ha che il sistema non va in errore se un solo nodo va in errore in quanto i nodi sono indipendenti ma è difficile occultare gli errori parziali dei singoli nodi ed è difficile sistemare gli eventuali errori del singolo nodo. Ovviamente non si ha memoria condivisa e non c'è uno stato globale. In un sistema distribuito non si ha un clock globale e non si può controllare globalmente o avere uno scheduling globale.

2.0.1 Il modello Client-Server



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 ha 4 problemi da fronteggiare:

1. **identificare la controparte**, che si risolve assegnando un nome, è la procedura di **naming**
2. **accedere alla controparte**, che si risolve con una reference, un **access point**
3. **comunicare (parte 1)**, che si risolve accettando e condividendo un formato, un **protocollo, "protocol"**
4. **comunicare (parte 2)**, 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:

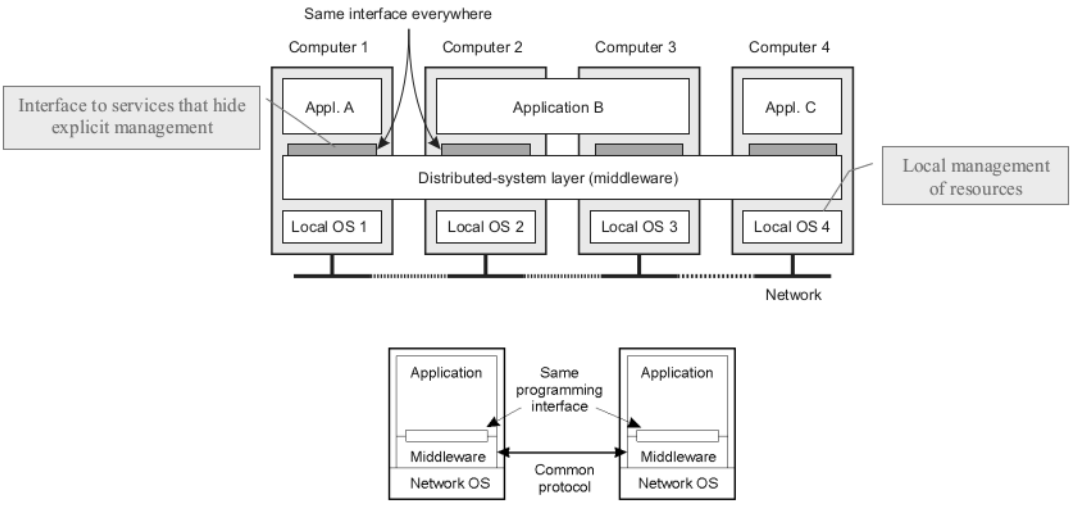
- **naming**, usare nomi simbolici per identificare le risorse che sono parte del sistema distribuito
- **access trasparency**, nascondere le differenze nella rappresentazione delle informazioni e nell'accedere ad un'informazione locale o remota
- **location trasparency**, nascondere dove è collocata una risorsa sulla rete
- **relocation or mobility transparency**, nascondere che una risorsa può essere stata trasferita ad un'altra locazione mentre è in uso

- **migration trasparency**, nascondere che una risorsa può essere trasferita
- **replication transparency**, nascondere che una risorsa può essere replicata
- **concurrency transparency**, nascondere che una risorsa può essere condivisa da molti utenti indipendenti
- **failure trasparency**, nascondere fallimenti e recovery di una risorsa
- **persistence trasparency**, nascondere se una risorsa è volatrile o memorizzata permanentemente

non si possono però nascondere:

- **ritardi e latenze di comunicazione**
- **nascondere completamente i failure della rete e dei nodi**, non puoi neanche distinguere bene rallentamenti e errori. Ovviamente non puoi sapere se sta per accadere un **crash**

Una trasparenza completa, oltre ad essere quasi impossibile a livello teorico, è anche estremamente "cara" a livello di performances e tempistiche (causa scrittura costante su dischi e mantenimento delle repliche).
Nascondere le informazioni è alla base dell'ingegneria del software. Bisogna separare il *cosa* si fa e il *come* lo si fa. Il *cosa* si fa mediante la definizione dell'interfaccia, *Interface Definition Languages (IDL)*, e il *come* mediante l'implementazione delle classi e dei metodi. Le interfacce sono definite mediante principi standard, sono complete e sono neutrali (indipendenti dall'implementazione).



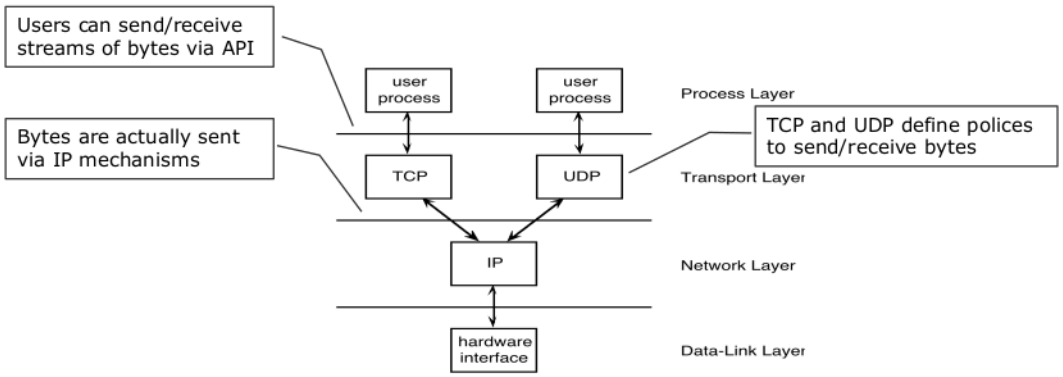
Tra i vari componenti si ha:

- **indipendenza logica**, con i vari componenti che lavorano autonomamente
- **composizione**, con la collaborazione dei vari processi

Si separano:

- **meccanismi**, ciò che è fatto dai componenti (esempio il context switch)
- **politiche**, come vengono applicati le varie funzionalità del sistema (esempio lo scheduling Round Robin RR)

Bisogna separare e bilanciare politiche e meccanismi



Ricapitolando il concetto di protocollo:

- per poter capire le richieste e formulare le risposte i due processi devono concordare un protocollo
- i protocolli (come *HTTP, FTP e SMTP*) 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

Vediamo un esempio di codice. Partiamo dall'*header.h*

```
// 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

// 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

// errori

#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

// definizione del messaggio

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
};
```

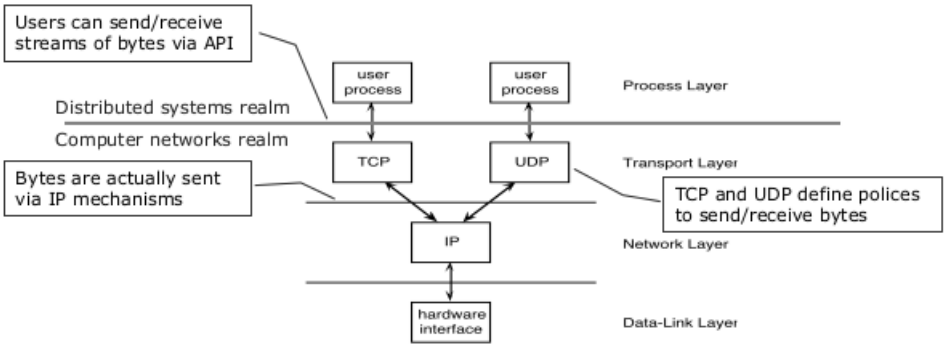
vediamo la struttura di un semplice server che realizza un semplice file server remoto:

```
#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){ // vari casi in base alla richiesta
      case CREATE:
        r = do_create(&m1, &m2);
        break;
      case CREATE:
        r = do_read(&m1, &m2);
        break;
      case CREATE:
        r = do_write(&m1, &m2);
        break;
      case CREATE:
        r = do_delete(&m1, &m2);
        break;
      default:
        r = E_BAD_OPCODE;
    }

    m2.result = r; // ritorna il risultato al client
    send(m1.source, &m2); // manda la risposta

  }
}
```

vediamo ora un client che usa il servizio per straferire un file:

```
#include <header.h>

int copy(char *src, char *dst){ // copia file usando il server
  strcut 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
}
```

2.0.2 Stream Communication

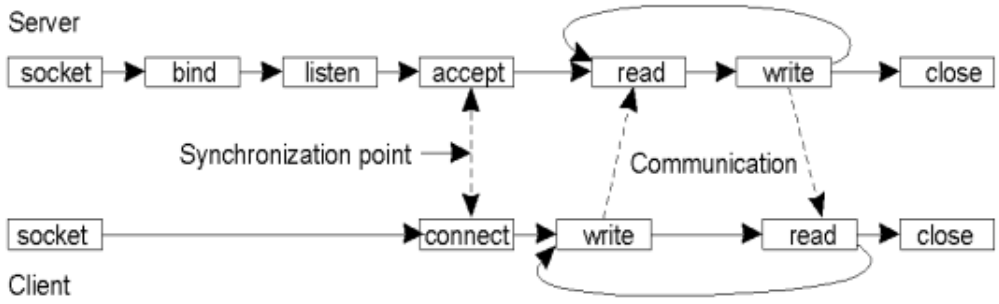


Si ha il modello ISO/OSI, che si basa sull'astrazione. Un informatico dovrebbe conoscere tutti i vari livelli. Gli utenti possono mandare e ricevere stream di byte via TCP mediante API. Poi al livello successivo si hanno datagrammi con i dati che vengono mandai via meccanismi IP. È lo sviluppatore a decidere le varie politiche. I sistemi distribuiti lavorano tra utenti e TCP/UDP. Ovviamente sono i processi che comunicano tra di loro. Ogni processo comunica attraverso canali. Un canale gestisce i flussi di dati in ingresso e uscita e dall'esterno ogni canale è identificato da un 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 (well-known port). TCP è orientato alla connessione (si ha un invio di richiesta di connessione), è affidabile, si ha un controllo di flusso, si ha un controllo della congestione ma non si hanno garanzie di banda e ritardo minimi. UDP non è affidabile e non offre nulla di quanto offerto da TCP ma è comodo, per esempio, nello streaming, che tollera perdite parziali. UDP scompone i messaggi in pacchetti che invia uno per volta ai servizi network. TCP scompone e invia come UDP ma ogni pacchetto viene numerato per garantire riordinamento, duplicazioni e perdite. Nei sistemi distribuiti non è necessario conoscere tutto questo funzionamento ma importa solo lo stream di byte. TCP utilizza variabili e buffer per realizzare il trasferimenti bidirezionale di flussi di bytes ("pipe") tra processi, prevede ruoli client/server durante la connessione ma non durante la comunicazione. TCP utilizza i servizi dello strato IP per l'invio dei flussi di bytes.
L'interfaccia tra applicazione e strato di trasporto è dato delle API, *Application Programming Interface* e i socker sono API per accedere a TCP o UDP,

due processi comunicano mediante socker nel modello client-server.
Si hanno delle criticità:

- 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

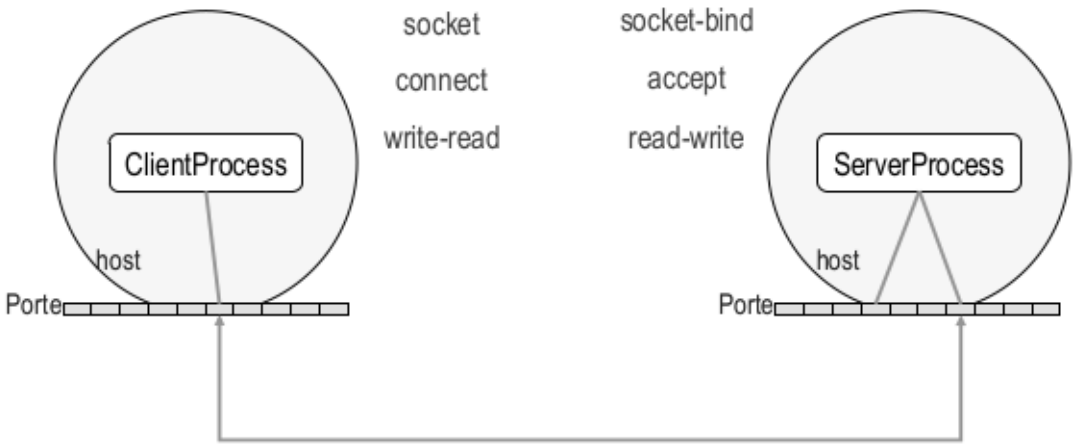
l'indirizzo del server può essere una costante nel codice, può essere chiesto all'utente, come nel browser, posso usare un nameserver o un repository (con DNS, *Domain Name Service*) o adottare altri protocolli, come DHCP. Il naming sarà il nome dell'host, l'indirizzo IP come access point mediante host e porta, il protocollo saranno stream di byte e la sintassi e la semantica saranno definiti dall'applicazione con http, smtp etc...
Tutto questo è a un basso livello di trasparenza in quanto sia utente che sviluppatore lo necessitano.
La comunicazione TCP/IP avviene attraverso flussi di byte (byte stream), dopo una connessione esplicita, tramite normali system call read/write. Queste due syscall sono sospensive (mettono il sistema in uno stato di *wait*) e usano un buffer per garantire flessibilità (per esempio la read definisce un buffer per leggere N caratteri, ma potrebbe ritornare avendone letti solo k<N)



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**

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:



Non uso la stessa porta per evitare perché comunicazione e handshake non sarebbero distinguibili
Quando si parla di socket non c'è il concetto di messaggio e read/write avvengono per un numero arbitrario di bytes. Quindi si devono prevedere cicli

di lettura che termineranno in base alla dimensione dei "messaggi" come stabilito dal formato del protocollo applicativo in uso.
Il prototipo della read in pseudocodice sarebbe:

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

con:

- byteLetti = byte effettivamente letti
- socket = canale da cui leggere
- buffer = sapzio 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
Java definisce alcune classi che costituiscono un'interfaccia ad oggetti alle system call:

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

Queste classi accorpano funzionalità e mascherano alcuni dettagli con il vantaggio di semplificarne 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. vediamo altro, iniziamo dai costruttori:

```
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 */
```

e passiamo ai metodi:

```
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.
```

abbiamo metodi per modificare i bytes:

```
public InputStream getInputStream()!
  throws IOException

/* 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 */
```

Vediamo ora i costruttori:

```
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.
```

I metodi per gestire le connessioni:

```
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 if 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 */
```

e infine vediamo qualche altro metodo:

```
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*/
```

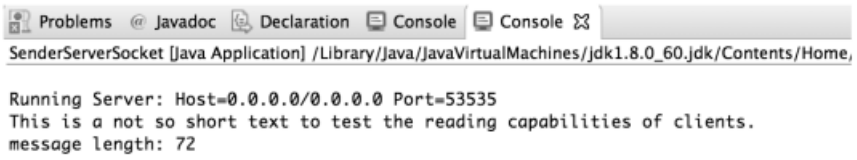
vediamo un esempio di un server che accetta una connessione da un client e manda uno stream di dati (una stringa) e di un client che legge lo stream di bytes. Partiamo col server

```
package serverWriter;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class SenderServerSocket {
  final static String message = "This is a not so
    short text to test the reading capabilities of clients."
   + " If they are not so smart, they will catch only part of it.";
  public static void main(String[] args) {
    try {
      Socket clientSocket;
      ServerSocket listenSocket;
      listenSocket = new ServerSocket(53535);
      System.out.println("Running Server: " + "Host="
        + listenSocket.getInetAddress() + " Port="
          + listenSocket.getLocalPort());

      while (true) {
        clientSocket = listenSocket.accept();
        System.out.println("Connected to client at port: "
           + clientSocket.getPort());

        PrintWriter out =
          new PrintWriter(clientSocket.getOutputStream(), true);
        System.out.println("Writing message: ");
        System.out.println(message);
        System.out.println("Message length: " + message.length());
        out.write(message);
        out.flush();
        clientSocket.close();
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

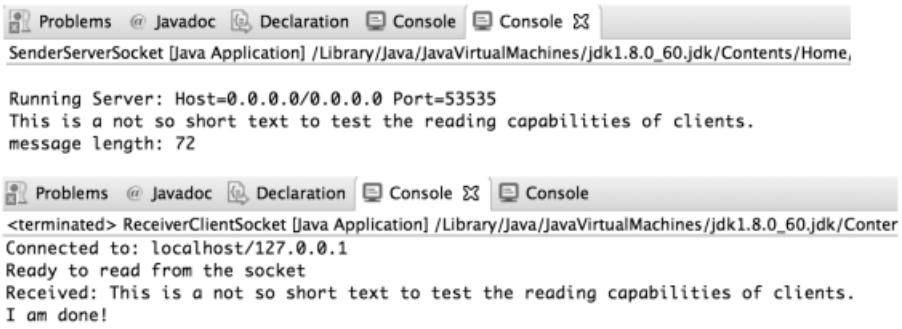
e il client:

```
package serverWriter;
import java.io.DataInputStream;
import java.io.IOException;
import java.net.InetAddress;
import java.net.Socket;

public class ReceiverClientSocket {

  public static void main(String[] args) {
    Socket socket; // my socket
    InetAddress serverAddress; // the server address
    int serverPort; // the server port
    DataInputStream in; // the source of stream of bytes
    byte[] byteReceived = new byte[1000]; // the temporary buffer
    String messageString = ""; // the text to be displayed

    try { // connect to the server
      serverAddress = InetAddress.getByName(args[0]);
      serverPort = Integer.parseInt(args[1]);
      socket = new Socket(serverAddress, serverPort);
      System.out.println("Connected to server: "
         + socket.getPort()); // the server port

      in = new DataInputStream(socket.getInputStream()); // the stream to
                                  // read from

      // The following code shows in detail how to read from a TCP socket
      System.out.println("Ready to read from the socket");
      int bytesRead = 0; // the number of bytes read while (true) {
      bytesRead = in.read(byteReceived);
      messageString += new String(byteReceived, 0, bytesRead);
      System.out.println("Received: " + messageString);
      System.out.println("I am done!");
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Quindi per il server si avrà:



e per il client:



Si nota però che ci sono dei problemi col client, funziona... ma non nella maniera corretta.

2.0.3 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.

2.0.4 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    "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    clues[currentJoke] +
^^I^^I^^I    " who?\"" +
^^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 unsano 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

Message oriented computing

Il metodo di comunicazione tra client e server è supportato dal web tramite HTTP (da client a server con delle *request* e all'opposto con delle *response*). Il client è realizzato da un *browser* mentre il server da un *HTTP server* o da un *Web Server*.
Un **web browser**, detto *user-agent*, è un programma che consente la navigazione sul web, interpretando il codice con cui sono e espresse le web pages rappresentandolo in forma di ipertesto. Una web page è costituita da diversi oggetti identificati da URL. Le page interagiscono mediante ipertesti e link. L'URL, che identifica un oggetto nella rete e specifica come interpretare i dati ricevuti attraverso il protocollo, è formato da:

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

- identificatore della risposta

ovvero:

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

la parte testuale dei documenti è espressa da:

- HTML per contenuti e impaginazione
- CSS per il render
- XML e JSON per i dati e la loro struttura

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..) . La dinamicità delle pagine web è data invece da linguaggi come Javascript, VBScript, Java/applet...
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. Si hanno diversi protocolli come HTTP (*HyperText Transfer Protoco*), FTP *File Transfer Protoco*) e SMTP *Simple Mail Transfer Protoco*).
HTTP usa il modello client/server, col server che visualizza su browser gli oggetti che richiede e server che invia gli oggetti. HTTP usa TCP, infatti inizia una connessione TCP creando una socket verso il server sulla porta 80. Il server accetta la connessione TCP dal client. Vengono quindi scambiati messaggi http (messaggi del protocollo di livello applicativo) tra il browser (client http) e il Web server (server http).
Per ora si hanno due versioni:

1. **http1.0: RFC 1945**
2. **http1.1: RFC 2068**

HTTP è **stateless** in quanto il server non mantiene informazioni sulle precedenti richieste che devono quindi contenere tutte le informazioni necessarie. Si ha che *requeste* e *response* hanno la stessa struttura, in ASCII con un formato testo leggibile, per esempio:

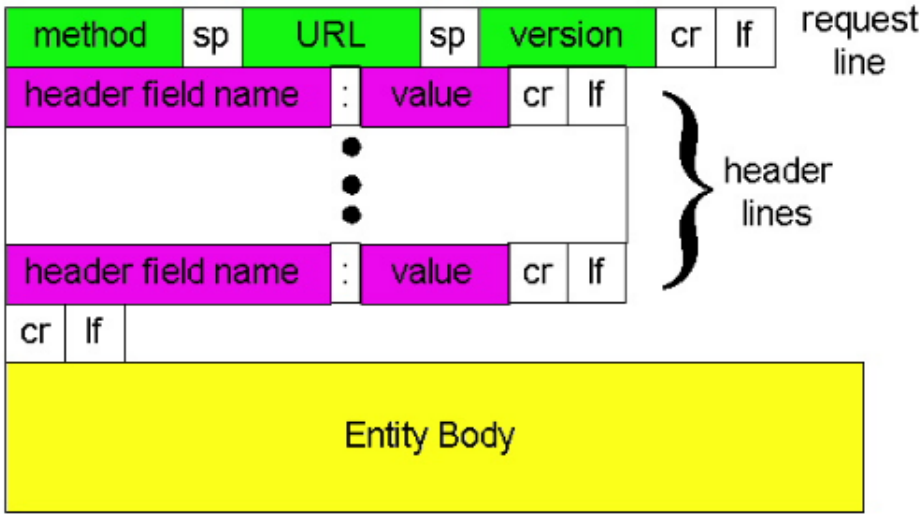
```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg
Accept-language: fr
```

Con la prima riga rappresentate la request line e le altre rappresentati l'header con le opzioni.
Si hanno quindi i metodi:

- **GET**:
  - restituisce una rappresentazione di una risorsa
  - include un eventuale input in coda alla URL della risorsa
  - è safe: l'esecuzione non ha effetti sul server => la risposta può essere gestita con una cache dal client
  - uso tipico: ottenere pagine html e immagini
- **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
- **HEAD**:
- simile al metodo get ma viene restituito solo l'Head della pagina Web
- spesso usato in fase di debugging

nel complesso:

		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			✓



vediamo un http response:

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

con la prima linea contenente protocollo, codice di stato e frase di stato.
si ha che:

- 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

ecco alcuni esempi di codici:

200 OK
- Successo, oggetto richiesto più avanti nel messaggio

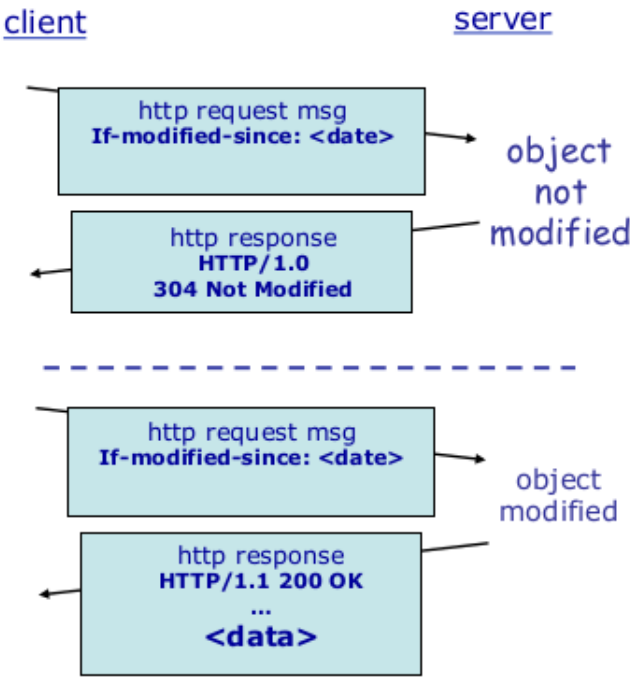
301 Moved Permanently
- L' oggetto richiesto è stato spostato. Il nuovo indirizzo è specificato più avanti (Location:)

400 Bad Request
- Richiesta incomprensibile al server

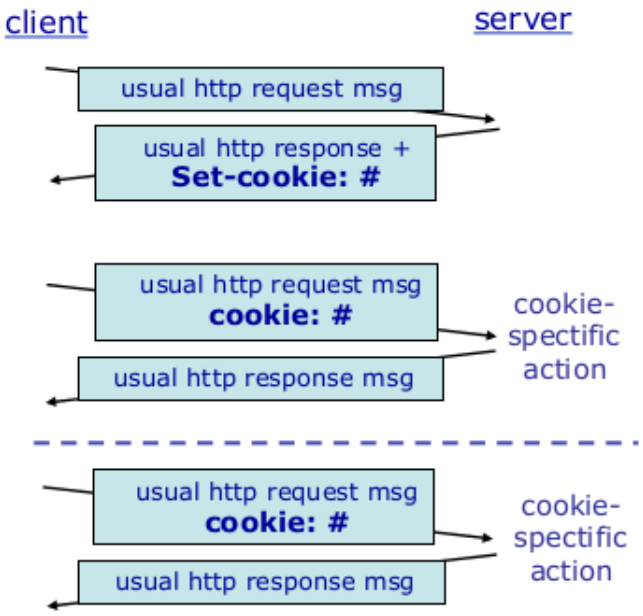
404 Not Found
- Il documento non è stato trovato sul server

505 HTTP Version Not Supported

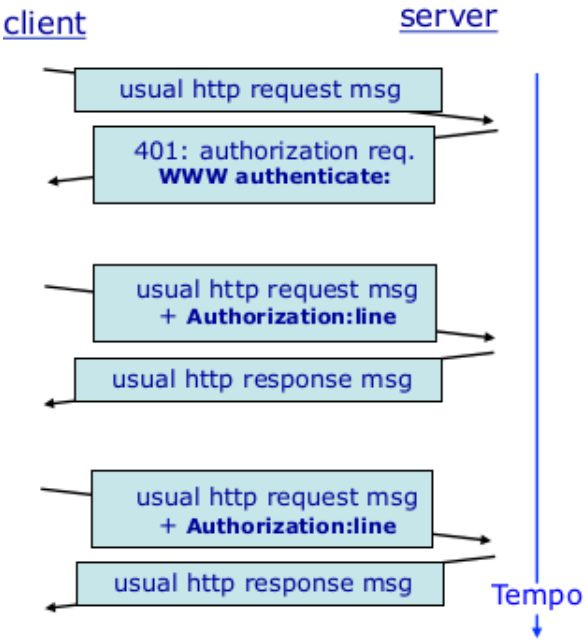
Si cerca di non inviare dati che il client ha già in cache e il server verifica la cosa:



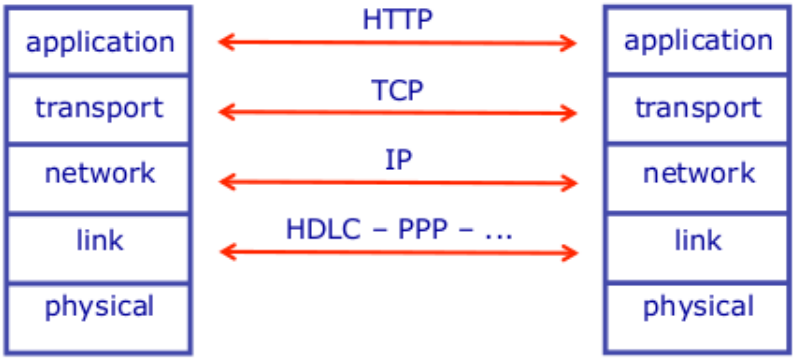
usando i cookie: **FOTO SBAGLIATA!!!**



Per gestire l'accesso ai documenti sul server, dato che http è stateless, si deve verificare ogni richiesta (ora i browser spesso forniscono il salvataggio di credenziali) e le informazioni necessarie all'autenticazione si trovano nell'header (*authorization: line*) senza le quali il server rifiuta la connessione (*www authenticate:*):



ovviamente i vari livelli hanno diversi protocolli:



Capitolo 3

HTML + CSS + JS

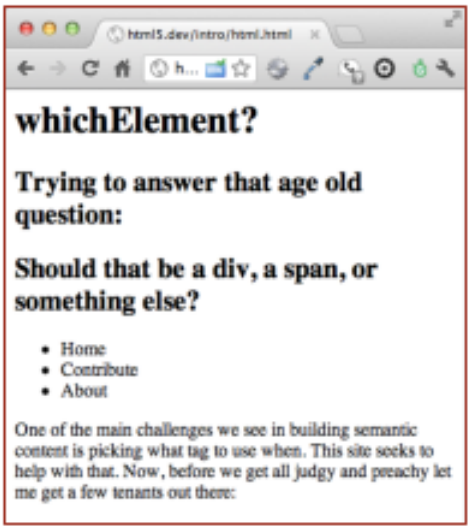
Innanzitutto una piccola distinzione:

- un **linguaggio di programmazione** è utilizzato per comunicare istruzioni a una macchina di calcolo, per definire programmi che controllino il comportamento di un calcolatore (lo sono *C, Java, C++...*)
- un **linguaggio di markup** è utilizzato per annotare un documento in modo tale che l'annotazione sia sintatticamente distinguibile dal testo (lo sono *LaTeX, HTML, XML...*). Qui le annotazioni possono avere diverse finalità:
  - di presentazione (definiscono come visualizzare il testo al quale sono associate)
  - procedurali (definiscono istruzioni per programmi che elaborino il testo al quale sono associate)
  - descrittive (etichettano semplicemente parti del testo, disaccoppiando la struttura dalla presentazione del testo stesso)

HTML (*Hyper Text Markup Language*) è un linguaggio di markup per dare struttura ai contenuti (web) e la sua specifica è definita dal **W3C** (*World Wide Web Consortium*).

Vediamo un piccolo esempio:

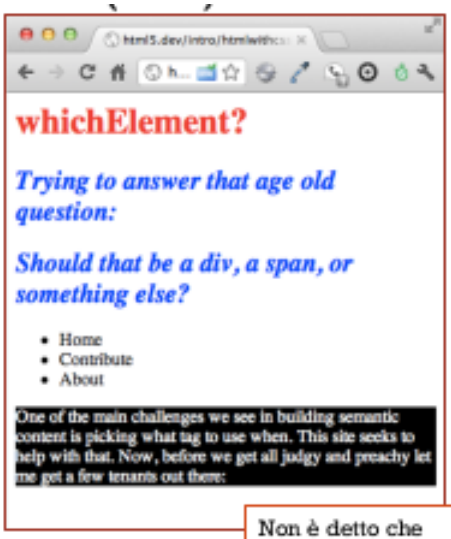
```
<h1>whichElement?</h1>

<h2>Trying to answer that age old
question:</h2>

<h2>Should that be a div, a span, or
something else?</h2>

<ul>

<li>Home</li>

<li>Contribute</li>

<li>About</li>

</ul>

<p>One of the main challenges we see in
building semantic content is picking what tag
to use when. This site seeks to help with that.
Now, before we get all judgy and preachy let
me get a few tenants out there:</p>
```



IL **CSS** (*Cascading Style Sheets*) è un linguaggio per dare uno stile (di presentazione/visuale) ai contenuti (web) e la sua specifica è definita dal **W3C** (*World Wide Web Consortium*).
Vediamo un esempio:

```
<style type="text/css">
    h1{
        color: red;
    }
    h2{
        color: blue;
        font-style: italic;
    }
    p{
        color: white;
        background-color: black;
    }
</style>
```

```
<head>
  <link rel="stylesheet" type="text/css" href="theme.css">
</head>
```

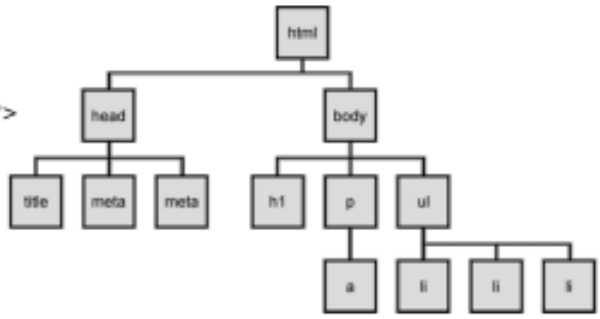


Non è detto che sia un bello stile!

```
<style type="text/css">

    h1{
        color: red;
        font-size: 16px;
    }

    h2{
        color: blue;
        font-size: 26px;
    }

    ul{
        list-style: none;
        clear: both;
        display: block;
        width: 100%;
        height: 30px;
        padding: 0;
    }

    li{
        float: left;
        padding-right: 20px;
    }

</style>
```

Il **DOM** (*Document Object Model*) è una interfaccia neutrale rispetto al linguaggio di programmazione e alla piattaforma utilizzata per consentire ai programmi l'accesso e la modifica dinamica di contenuto, struttura e stile di un documento (web). **DOM** è un API definita dal W3C (e implementata da ogni browser moderno) per l'accesso e la gestione dinamica di documenti XML e HTML.
Graficamente si ha:

```
<html>
   <head>
      <title>Example ...</title>
      <meta charset="UTF-8">
      <meta name="author" content="John Doe">
   </head>
   <body>
      <h1>Heading 1</h1>
      <p>This is a paragraph of text with a
      <a href="/path/page.html">link in it</a>.
      </p>
      <ul>
            <li>Coffee</li>
            <li>Tea</li>
            <li>Milk</li>
      </ul>
   </body>
</html>
```



Ogni nodo può essere caratterizzato da attributi che ne facilitano l'identificazione, la ricerca, la selezione:

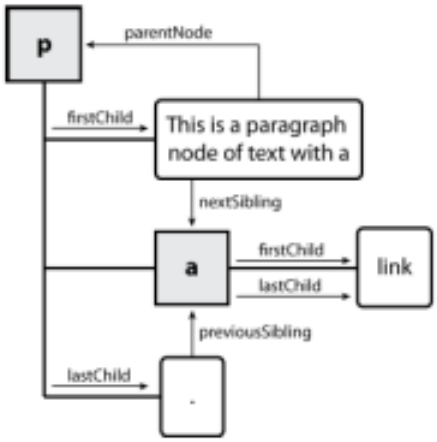
- un **identificatore univoco**, anche se il DOM non garantisce l'unicità
- una **classe** che indica l'appartenenza ad un insieme che ci è utile definire

Il browser stesso fornisce funzionalità di ricerca, per esempio:

- *getElementById(IdName)*
- *getElementsByClassName(ClassName)*

```
<h1>A One Page FAQ</h1>
<div id="introText">
  <p>bla bla bla</p>
</div>

<h2>I've heard that JavaScript is the
long-lost fountain of youth. Is this
true?</h2>
<div class="answer">
  <p>bla bla bla</p>
</div>

<h2>Can JavaScript really solve all of
my problems?</h2>
<div class="answer">
  <bla bla bla</p>
</div>
```

```
<p id="foo">
  This is a paragraph of text with a
  <a href="/path/to/page.html">link</a>
  .
</p>
```



Possiamo specificare un selettore più preciso rispetto al nome del tag, per esempio specificando il nome di una classe o un identificatore di elemento del DOM:

```
p.intro{
  color:red;
}
```

Questo stile sarà applicato solo ai tag di classe intro:

```
<p class = "intro">
```

Abbiamo quindi una lista di **selectors** principali:

- **tag name**, il semplice nome del tag:

  ```
  p{
     ...
  }
  ```

  così quanto scritto si applicherà a tutti i tags <p>

- il **dot (.)** applicabile a un tag, indica una classe:

  ```
  p.highlight{
     ...
  }
  ```

  così quanto scritto si applicherà a tutti i tags <p> con *class = "highlight"*

- lo **sharp character (#)**, applicabile a un tag, indica un identificativo:

  ```
  p#intro{
     ...
  }
  ```

  così quanto scritto si applicherà a tutti i tags <p> con *id = "intro"*

- i **two dots (:)** stati comportamentali (ad esempio evento mouseover):

  ```
  p:hover{
     ...
  }
  ```

  così quanto scritto si applicherà a tutti i tags <p> quando passa sopra il cursore

- le **brackets([attr = 'value'])** tag con un valore specifico per un attributo *'value'*:

  ```
  input[type="text"]{
     ...
  }
  ```

  così quanto scritto si applicherà ai tag di input di tipo *text*

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

HTML5

Introduce nuovi **elementi semantici** per la strutturazione delle pagine, per esempio:

- article
- section
- aside
- header
- footer

Introduce 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, ma anche audio, video e canvas.
Si hanno nuove **API javascript** per la manipolazione delle pagine: offline data, drag and drop, web storage.
quindi si usa:

- <p> *per definire un paragrafo*
- *per definire una lista di elementi il cui ordine non è importante*
- <address> *per indicare informazioni relative a un indirizzo*
- e *per contenere informazione che si vuole delimitare per qualche motivo (successive manipolazioni)*

Vediamo anche dei brutti usi degli stessi:

- <p> *per andare a capo*
- <blockquote> *per gestire l'indentazione*
- <h1> *per ingrandire del testo*

L'HTML dovrebbe non contenere informazione di presentazione, riservata ai CSS, quindi senza stili definiti "in linea" e senza l'uso di tag come <font>, <b>, <i>.
Vediamo come sono cambiate le cose con header e footer (quest'ultimo ovviamente con un tag dedicato diverso da quello dell'esempio).
Prima:

```
<div id="header">
  <h1>The Awesome Blog of Awesome</h1>
    <p class="tagline">Awesome is a State of Mind</p>
</div>
```

poi, con HTML5:

```
<header>
  <h1>The Awesome Blog of Awesome</h1>
  <h2>Awesome is a State of Mind</h2>
</header>
```

per i menù di navigazione:
Prima:

```
<div id="nav">
  <ul>
    <li><a href="">Home</a></li>
    <li><a href="">Blog</a></li>
    <li><a href="">About</a></li>
    <li><a href="">Contact</a></li>
  </ul>
</div>
```

poi, con HTML5:

```
<nav>
  <ul>
    <li><a href="">Home</a></li>
    <li><a href="">Blog</a></li>
```

```
    <li><a href="">About</a></li>
    <li><a href="">Contact</a></li>
  </ul>
</nav>
```

per articoli.
Prima:

```
<div class="post">
  <div class="postheader">
    <h3><a href="">I Scream My Thoughts</a></h3>
    <p class="date">August 10, 2011</p>
  </div>
  <p>You probably haven't heard of them duis</p>
</div>
```

poi, con HTML5:

```
<article>
  <header>
    <h3><a href="">I Scream My Thoughts</a></h3>
    <p class="date">August 10, 2011</p>
  </header>
  <p>You probably haven't heard of them duis</p>
<article>
```

o per i contenuti.
Prima:

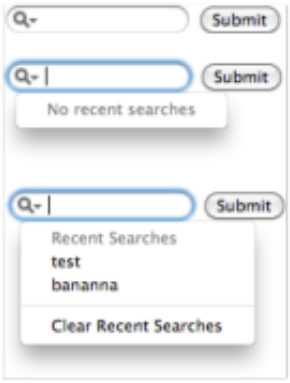
```
<div class="content">
  <div class="post">
    ...
  </div>
  <div class="post">
    ...
  </div>
  <div class="post">
  ...
  </div>
</div>
```

poi, con HTML5:

```
<section>
  <article>
    ...
  </article>
  <article>
    ...
  </article>
  <article>
    ...
  <article>
</section>
```

Si nota come 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.
Vediamo un widget per l'input, un widget per il search:

```
<input type="search" name="search" />
```



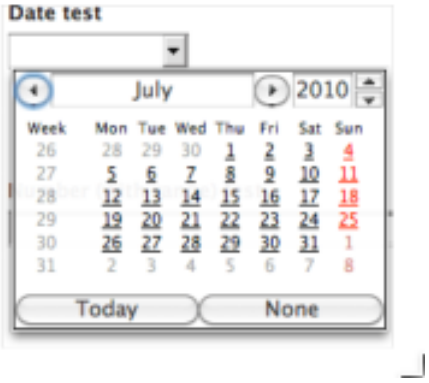
Questi widget non hanno supporto uniforme e vengono visualizzati diversamente a seconda dei browser ma è supportato da quasi tutti.
Vediamo un widget di input per una mail:

```
<input type="email" name="email" />
```

che fornisce una forma di validazione del dato inserito e forza la tastiera dedicata (con la chiocciola) su device mobili. È supportato da pochissimi browser.

Widget per il calendario:

```
<input type="date" name="dob" />
```



che fornisce una forma di validazione del dato inserito ma vengono visualizzati diversamente a seconda dei browser ma è supportato da quasi tutti.
Manca il modo di definire e governare la dinamicità della pagina, il modo della pagina di cambiare in modo reattivo al comportamento dell'utente o anche proattivo, senza necessariamente dover attendere uno stimolo dall'utente. Tutto ciò è svolto dal **JS**. Il **Javascript** è il linguaggio di scripting che svolge questa funzione, ne parleremo più avanti in esercitazioni dedicate.