

Programmazione e sicurezza delle reti

VR443470

aprile 2023

Indice

1 Scrittura di applicazioni di rete mediante interfaccia socket	4
1.1 Host, processo e applicazione	4
1.2 Modalità di trasmissione in Internet	5
1.2.1 Applicazioni orientate al datagramma (UDP)	5
1.2.2 Applicazioni orientate alla connessione (TCP)	5
1.3 Schemi di applicazioni che utilizzano la rete	6
1.3.1 Modello client/server	6
1.4 Creazione dell'interfaccia Socket	6
1.5 Esempi di codice	7
1.5.1 Esecuzione degli esempi	7
1.5.2 Client UDP	7
1.5.3 Server UDP	8
1.5.4 Client_inc UDP	9
1.5.5 Server_inc UDP	10
1.5.6 Client TCP	11
1.5.7 Server TCP	12
1.5.8 Codice per la copia di un file	13
1.6 Esercizi	14
1.6.1 Esercizio 1 - UDP	14
1.6.2 Esercizio 2 - UDP	14
1.6.3 Esercizio 3 - UDP	15
1.6.4 Esercizio 4 - UDP	16
1.6.5 Esercizio 5 - UDP	16
1.6.6 Esercizio 6 - UDP	17
1.6.7 Esercizio 7 - UDP	19
1.6.8 Esercizio 8 - Sommatrice UDP	19
1.6.9 Esercizio 9 - Sommatrice UDP e perdita di pacchetti	22
1.6.10 Esercizio 10 - Sommatrice UDP e influenze reciproche	23
1.6.11 Esercizio 11 - Sommatrice TCP	26
1.6.12 Esercizio 12 - Sommatrice TCP e influenze reciproche	28
1.6.13 Esercizio 13 - Sommatrice TCP e perdita di pacchetti	29
2 Dal Web ai Webservices	30
2.1 Protocollo HTTP/HTTPS	30
2.2 Hyper Text Markup Language (HTML) e Cascading Style Sheets (CSS)	32
2.2.1 HTML: tag per richiamare immagini	32
2.2.2 HTML: tag per il collegamento ipertestuale	33
2.2.3 Document Object Model (DOM)	33
2.3 Javascript	33
2.3.1 Javascript e Document Object Model (DOM)	34
2.4 Uniform Resource Locator (URL)	36
2.5 Passare dei dati al server web col metodo GET	37
2.6 Passare dei dati al server web col metodo POST	39
2.7 Common Gateway Interface (CGI)	41
2.8 Web socket	43
2.8.1 Approfondimento WebSocket	45
2.8.2 Limitazioni	45

2.9	WebSocket-Chat	46
2.9.1	Node.js e l'approccio asincrono	46
2.9.2	Descrizione dell'applicazione	46
2.9.3	Codice Back-end server	47
2.9.4	Codice Front-end HTML	48
2.9.5	Codice Front-end JavaScript	49
2.9.6	Esecuzione del Front-end e del Back-end	50
2.9.7	Esercizi	51
2.10	Architetture orientate ai servizi (Service-Oriented Architecture, SOA)	57
2.10.1	Funzioni remote e webservice	57
2.10.2	Webservice basati su REST	58
2.10.3	Breve introduzione ai file JSON	59

1 Scrittura di applicazioni di rete mediante interfaccia socket

1.1 Host, processo e applicazione

L'**host** (colui che ospita) è una **macchina** sempre identificata da un indirizzo IP a cui, opzionalmente, può essere associato un nome Internet.

Il **processo** è un **programma in esecuzione** sull'host, il quale trasmette/-riceve pacchetti verso/da altri processi su altri host attraverso la rete. Viene identificato tramite un numero di porta nell'intervallo 0 - 65535.

Un **applicazione** è una collaborazione tra un **insieme di processi** sparsi sulla rete per fare qualcosa di utile per l'utente, per esempio chat, e-mail, ecc.

Alcuni **esempi**:

- Eseguendo una ricerca su internet:
 - Il *web* è l'applicazione;
 - Mentre i browser (Chrome, Firefox, Edge, Safari) sono il processo di esecuzione;
 - L'host è il PC, tablet o smartphone su cui viene aperto il browser;
 - Apache o NGINX è il processo di esecuzione sulla macchina remota, anch'essa identificata come host.
- Aprendo un'applicazione come Telegram:
 - Telegram è l'applicazione;
 - Il processo di esecuzione è sempre l'app Telegram che è in esecuzione sul dispositivo attualmente in uso (PC, tablet, ecc.) che funge da host;
 - Il server di Telegram è il processo di esecuzione sulla macchina remota, anch'essa identificata come host.

1.2 Modalità di trasmissione in Internet

Su Internet la modalità di trasmissione è una sequenza di byte chiamata: pacchetto, Protocol Data Unit (PDU), Datagram. A seconda del livello del protocollo, ci sono nomi diversi.

1.2.1 Applicazioni orientate al datagramma (UDP)

Alcune **applicazioni sono orientate al datagramma**, quindi **ogni pacchetto scambiato tra gli host è indipendente dai precedenti e successivi**. Le **perdite** di pacchetti non vengono tenute in considerazione ed un **esempio** può essere la **trasmissione di temperature**: il ricevitore può non tener conto di alcune perdite poiché le informazioni ricevute non sono necessarie per il futuro.

1.2.2 Applicazioni orientate alla connessione (TCP)

Invece, alcune **applicazioni sono orientate alla connessione**. A differenza delle applicazioni orientate al datagramma, quelle orientate alla connessione devono tener conto delle perdite poiché solitamente le informazioni scambiate sono di dimensioni rilevanti (e.g. un'immagine). Di conseguenza, una perdita provocherebbe una lettura parziale o impossibile da parte del ricevitore.

Il **socket** si preoccupa di **numerare i pacchetti appartenenti alla stessa connessione** per rilevare eventuali pacchetti persi e poterli ritrasmettere.

Il sistema operativo introduce all'interno dei pacchetti un numero di sequenza così che possa rilevare eventuali pacchetti persi e ritrasmetterli.

- **Vantaggi:**

- L'utente scrive/legge su un archivio remoto con la stessa naturalezza di quando scrive/legge su un archivio locale come se la rete in mezzo non ci fosse.

- **Svantaggi:**

- Gli host mittente e destinatario eseguono un lavoro più complesso con il sistema operativo;
 - Ritardo di ritrasmissione nel caso in cui i pacchetti vengano persi.

1.3 Schemi di applicazioni che utilizzano la rete

Le applicazioni di rete sono insiemi di processi su host diversi che si scambiano messaggi attraverso la rete. **Esistono degli schemi base che regolano lo scambio di messaggi:**

- Modello client/server;
 - Modello Publisher/Subscriber (Pub/Sub)
-

1.3.1 Modello client/server

Il **modello client/server** è quello più utilizzato e funziona nel seguente modo (attenzione all'ordine!):

1. Il *client* esegue una **richiesta** inviando dei dati al *server*;
2. Il *server* riceve i dati del *client*, processa i dati e invia la **risposta** al *client*. Infine, si mette in attesa di altre richieste.

I dati inviati dal *client* possono essere delle trasmissioni di dati o delle richieste di dati. In ogni caso, **il ruolo è determinato dall'ordine dei messaggi e non dal contenuto**. Si noti che il *client* e *server* sono processi e non host. Infatti, l'insieme di un client e un server costituisce l'applicazione di rete.

Un **esempio** di applicazione client/server è il **sensore di temperatura corporea** che funge da client e invia al server una temperatura. Le risposte del server sono “OK” per confermare l'avvenuta ricezione.

Un altro **esempio** di applicazione client/server è il display che funge da cliente e chiede al server una temperatura. Le risposte del server in questo caso saranno i dati richiesti.

1.4 Creazione dell'interfaccia Socket

Il programma, prima di utilizzare la rete, deve essere in grado di creare un oggetto di tipo **socket**. Esso è identificato principalmente da tre parametri:

- Indirizzo IP locale;
- Porta locale, la quale è un intero senza segno di 16 bit (quindi da 0 a 65'535). Nel modello client/server:
 - Il **server** deve decidere esplicitamente il numero di porta affinché i client possano saperlo (da 0 a 1023 le porte sono chiamate “porte note” perché sono utilizzate per protocoli famosi come HTTP);
 - Il **client** può decidere se scegliere il numero di porta esplicitamente oppure delegare la scelta al sistema operativo.
- Modalità di trasmissione: UDP o TCP.

1.5 Esempi di codice

1.5.1 Esecuzione degli esempi

1. Aprire due terminali
2. Compilare il server e successivamente eseguirlo con il comando:

```
1 -gcc network.c serverUDP.c -o serverUDP -lpthread
```

3. Compilare il client e successivamente eseguirlo con il comando:

```
1 -gcc network.c clientUDP.c -o clientUDP -lpthread
```

1.5.2 Client UDP

Il client UDP ha il seguente codice:

```
1 #include "network.h"
2
3 int main(void) {
4     socketif_t socket;
5     char request[]="Ciao sono il client!\n";
6     char response[MTU];
7     char hostAddress[MAXADDRESSLEN];
8     int port;
9
10    socket = createUDPIface(20000);
11    printf("[CLIENT] Spedisco messaggio al server\n");
12    printf("[CLIENT] Contenuto: %s\n", request);
13    UDPSend(socket, request, strlen(request), "127.0.0.1", 35000);
14    UDPReceive(socket, response, MTU, hostAddress, &port);
15    printf("[CLIENT] Ho ricevuto un messaggio da host/porta %s/%d\n",
16          hostAddress, port);
17    printf("[CLIENT] Contenuto: %s\n", response);
```

- (4-8) dichiarazione delle variabili tra cui la variabile socket;
- (10) inizializzazione dell’interfaccia socket sulla porta 20’000;
- (13) invio, tramite UDP, il messaggio “Ciao sono il client!” al destinatario avente indirizzo IP “127.0.0.1” (*localhost*) e porta 35’000;
- (14) attesta dell’arrivo della risposta del server;
- (15-16) scrittura sul terminale della porta, dell’indirizzo del mittente e del messaggio.

1.5.3 Server UDP

Il server UDP ha il seguente codice:

```
1 #include "network.h"
2
3 int main(void) {
4     socketif_t socket;
5     char response[]="Sono il server: ho ricevuto correttamente il
6     tuo messaggio!\n";
7     char request[MTU];
8     char hostAddress[MAXADDRESSLEN];
9     int port;
10
11     socket = createUDPIface(35000);
12     printf("[SERVER] Sono in attesa di richieste da qualche client\
13     n");
14     UDPReceive(socket, request, MTU, hostAddress, &port);
15     printf("[SERVER] Ho ricevuto un messaggio da host/porta %s/%d\n",
16     hostAddress, port);
17     printf("[SERVER] Contenuto: %s\n", request);
18     UDPSend(socket, response, strlen(response), hostAddress, port);
19 }
```

- (4-8) dichiarazione delle variabili tra cui la variabile socket;
- (10) inizializzazione dell'interfaccia socket sulla porta 35'000;
- (12) attesta dell'arrivo di qualche messaggio da parte di qualche client;
- (13-14) ricezione di un messaggio da parte di un client e stampa sul terminale dell'indirizzo, della porta e del messaggio del mittente;
- (15) invio della risposta del server al client.

1.5.4 Client_inc UDP

Il client UDP (paragrafo 1.5.2) può essere riscritto nel seguente modo:

```
1 #include "network.h"
2
3 int main(void) {
4     int request;
5     int response;
6     socketif_t socket;
7     char hostAddress[MAXADDRESSLEN];
8     int port;
9
10    socket = createUDPInterface(20000);
11    printf("Inserisci un numero intero:\n");
12    scanf("%d", &request);
13    UDPSend(socket, &request, sizeof(request), "127.0.0.1", 35000);
14    UDPReceive(socket, &response, sizeof(response), hostAddress, &
15    port);
16    printf("[CLIENT] Ho ricevuto un messaggio da host/porta %s/%d\n",
17    hostAddress, port);
18    printf("[CLIENT] Contenuto: %d\n", response);
19 }
```

- (4-8) dichiarazione delle variabili tra cui la variabile socket;
- (10) inizializzazione dell’interfaccia socket sulla porta 20’000;
- (11-12) inserimento di un numero intero da parte dell’utente;
- (13) invio, tramite UDP, del numero inserito dall’utente al destinatario aventure indirizzo IP “127.0.0.1” (*localhost*) e porta 35’000;
- (14) attesta dell’arrivo della risposta del server;
- (15-16) scrittura sul terminale della porta, dell’indirizzo del mittente e del messaggio.

1.5.5 Server_inc UDP

Il server UDP (paragrafo 1.5.3) può essere riscritto nel seguente modo:

```
1 #include "network.h"
2
3 int main(void) {
4     int request;
5     int response;
6     socketif_t socket;
7     char hostAddress[MAXADDRESSLEN];
8     int port;
9
10    socket = createUDPInterface(35000);
11    printf("[SERVER] Sono in attesa di richieste da qualche client\
n");
12    UDPReceive(socket, &request, sizeof(request), hostAddress, &
13    port);
14    printf("[SERVER] Ho ricevuto un messaggio da host/porta %s/%d\n",
15    hostAddress, port);
16    printf("[SERVER] Contenuto: %d\n", request);
17    response = request + 1;
18    UDPSend(socket, &response, sizeof(response), hostAddress, port)
19    ;
20 }
```

- (4-8) dichiarazione delle variabili tra cui la variabile socket;
- (10) inizializzazione dell'interfaccia socket sulla porta 35'000;
- (12) attesta dell'arrivo di qualche messaggio da parte di qualche client;
- (13-14) ricezione di un messaggio da parte di un client e stampa sul terminale dell'indirizzo, della porta e del messaggio del mittente;
- (15) incremento di uno del valore intero ottenuto;
- (15) invio della risposta del server al client con il valore incrementato.

1.5.6 Client TCP

Il client TCP ha il seguente codice:

```
1 #include "network.h"
2
3 int main(void) {
4     connection_t connection;
5     int request, response;
6
7     printf("[CLIENT] Creo una connessione logica col server\n");
8     connection = createTCPConnection("localhost", 35000);
9     if (connection < 0) {
10         printf("[CLIENT] Errore nella connessione al server: %i\n",
11               connection);
12     }
13     else
14     {
15         do
16         {
17             printf("[CLIENT] Inserisci un numero intero:\n");
18             scanf("%d", &request);
19             printf("[CLIENT] Invio richiesta con numero al server\n");
20             TCPSSend(connection, &request, sizeof(request));
21         } while (request != 0);
22         TCPReceive(connection, &response, sizeof(response));
23         printf("[CLIENT] Ho ricevuto il seguente risultato dal
24         server: %d\n", response);
25         closeConnection(connection);
26     }
27 }
```

- (4-5) dichiarazione delle variabili tra cui la variabile `connection` per gestire la connessione;
- (7-8) creazione di una connessione TCP con il server utilizzando `localhost` e la porta 35'000.
- (9-10) controllo del valore della connessione per verificare se c'è stato un errore. In tal caso, la connessione termina con la stampa dell'errore su terminale;
- (12-15) in caso di connessione riuscita, il client richiede l'inserimento di un valore intero all'utente;
- (16-17) invio del valore intero al server;
- (18) attesa di una risposta da parte del server;
- (19-20) al momento della ricezione della risposta, il client stampa la risposta del server e chiude la connessione.

1.5.7 Server TCP

Il server TCP ha il seguente codice:

```
1 #include "network.h"
2
3 int main(void) {
4     int request, response;
5     socketif_t socket;
6     connection_t connection;
7
8     socket = createTCPServer(35000);
9     if (socket < 0){
10         printf("[SERVER] Errore di creazione del socket: %i\n",
11               socket);
12     }
13     else
14     {
15         printf("[SERVER] Sono in attesa di richieste di connessione
16               da qualche client\n");
17         connection = acceptConnection(socket);
18         printf("[SERVER] Connessione instaurata\n");
19
20         int somma = 0;
21         do
22         {
23             TCPReceive(connection, &request, sizeof(request));
24             printf("[SERVER] Ho ricevuto la seguente richiesta dal
25                   client: %d\n", request);
26             somma += request;
27             } while (request != 0);
28             printf("[SERVER] Invio il risultato al client\n");
29             TCPSend(connection, &somma, sizeof(somma));
30             closeConnection(connection);
31     }
32 }
```

- (4-6) dichiarazione delle variabili tra cui la variabile `connection` per gestire la connessione e `socket` per gestire i dati;
- (8) inizializzazione di un socket TCP utilizzando la porta 35'000.
- (9-10) controllo del valore del socket per verificare se c'è stato un errore. In tal caso, la creazione termina con la stampa dell'errore su terminale;
- (12-14) in caso di creazione del socket riuscita, il server attende la connessione da parte di qualche client;
- (15-17) attesa di una richiesta da parte di qualche client. Nel momento in cui viene ricevuta una richiesta, il server la accetta e instaura la connessione e attende la ricezione dei dati;
- (18-19) all'arrivo dei dati da parte del client, il server esegue un incremento di uno del valore ricevuto dal client;
- (20-22) il server invia il valore al client e infine chiude la connessione.

1.5.8 Codice per la copia di un file

Il codice per la copia di un file è strutturato nel seguente modo:

```
1 #include <stdio.h>
2 #include <stdlib.h> // For exit()
3
4 int main()
5 {
6     FILE *fptr1, *fptr2;
7     char filename[100], c;
8
9     printf("Enter the filename to open for reading \n");
10    scanf("%s", filename);
11
12    // Open one file for reading
13    fptr1 = fopen(filename, "r");
14    if (fptr1 == NULL)
15    {
16        printf("Cannot open file %s \n", filename);
17        exit(0);
18    }
19
20    printf("Enter the filename to open for writing \n");
21    scanf("%s", filename);
22
23    // Open another file for writing
24    fptr2 = fopen(filename, "w");
25    if (fptr2 == NULL)
26    {
27        printf("Cannot open file %s \n", filename);
28        exit(0);
29    }
30
31    // Read contents from file
32    c = fgetc(fptr1);
33    while (c != EOF)
34    {
35        fputc(c, fptr2);
36        c = fgetc(fptr1);
37    }
38
39    printf("\nContents copied to %s", filename);
40
41    fclose(fptr1);
42    fclose(fptr2);
43    return 0;
44 }
```

- (6-29) dichiarazione dei puntatori ai file e tentativi di apertura dei due file richiesti dall'utente;
- (32-37) viene eseguita la lettura dal primo file e salvata nella variabile `c`. A questo punto, finché viene letto un carattere valido, ovvero che non sia la fine del file (*End Of File*, EOF), il contenuto della variabile `c` viene inserito nel secondo file;
- (39-43) al termine del processo di copia, viene stampato il file nel quale sono stati copiati i valori e chiusi i rispettivi file descriptors.

1.6 Esercizi

1.6.1 Esercizio 1 - UDP

Lanciare prima il server e poi il client. Cosa si osserva? Invertire la sequenza di lancio. Cosa si osserva?

Soluzione

Lanciando il server e successivamente il client, il primo attende la connessione da parte di qualcuno. Quindi, una volta avviato il client, le due parti inizieranno a comunicare.

Invece, avviando prima il client e successivamente il server, le due parti non riescono a comunicare. Questo perché il client tenta di raggiungere un host non esistente.

1.6.2 Esercizio 2 - UDP

Modificare i sorgenti per consentire al server di ricevere sulla porta 10000 e il client di trasmettere sulla propria porta 30000 (ogni modifica dei sorgenti richiede una loro ricompilazione).

Soluzione

Le modifiche da effettuare sono banali, ecco qua di seguito i codici del client (spiegazione al paragrafo 1.5.2):

```
1 #include "network.h"
2
3 int main(void) {
4     socketif_t socket;
5     char request[]="Ciao sono il client!\n";
6     char response[MTU];
7     char hostAddress[MAXADDRESSLEN];
8     int port;
9
10    socket = createUDPInterface(30000);
11
12    printf("[CLIENT] Spedisco messaggio al server\n");
13    printf("[CLIENT] Contenuto: %s\n", request);
14    UDPSend(socket, request, strlen(request), "localhost", 10000);
15
16    UDPReceive(socket, response, MTU, hostAddress, &port);
17    printf("[CLIENT] Ho ricevuto un messaggio da host/porta %s/%d\n
18    ", hostAddress, port);
19    printf("[CLIENT] Contenuto: %s\n", response);
```

E del server (spiegazione al paragrafo 1.5.3):

```
1 #include "network.h"
2
3 int main(void) {
4     socketif_t socket;
5     char response []="Sono il server: ho ricevuto correttamente il
6     tuo messaggio!\n";
7     char request[MTU];
8     char hostAddress[MAXADDRESSLEN];
9     int port;
10
11     socket = createUDPIInterface(10000);
12
13     printf("[SERVER] Sono in attesa di richieste da qualche client\
14         ");
14     UDPReceive(socket, request, MTU, hostAddress, &port);
15     printf("[SERVER] Ho ricevuto un messaggio da host/porta %s/%d\n",
16             hostAddress, port);
15     printf("[SERVER] Contenuto: %s\n", request);
16     UDPSend(socket, response, strlen(response), hostAddress, port);
17 }
```

1.6.3 Esercizio 3 - UDP

Mettere il server in ascolto sulla porta 100 e osservare cosa succede:

- Bisogna modificare anche il client? Se sì, dove?
- Per chi usa il proprio PC con Linux o una virtual machine Linux, lanciare il server con il comando “`sudo ./serverUDP`” e osservare cosa cambia.

Soluzione

Modificando il codice e inserendo il numero di porta 100, il server non riesce ad essere eseguito per un problema della porta. Infatti, la porta 100 fa parte delle *well-known port* e non può essere utilizzata per altri scopi.

Modificando anche il client e inviando il messaggio al localhost con porta 100, il codice viene compilato ed eseguito correttamente. Ovviamente il client rimane in attesa del server.

Compilando il server con la modalità `sudo` è possibile forzare la modifica della porta 100 e mettersi in ascolto su tale porta. Di conseguenza, il server si metterà in ascolto sulla porta 100 e il client che eseguirà l’invio di un messaggio in tale porta, riuscirà a trasmettere il messaggio.

1.6.4 Esercizio 4 - UDP

Sostituire “127.0.0.1” (o la stringa “localhost”) con localhost (o al contrario con 127.0.0.1) e poi con “pippo” e osservare cosa succede.

Soluzione

Modificando il parametro della chiamata a funzione `UDPSend` nel client viene modificato l’indirizzo del destinatario:

- Inserendo “127.0.0.1” si sta inserendo l’indirizzo privato creato per identificare l’host stesso;
 - Inserendo `localhost` si sta utilizzando un *alias*, dunque è come scrivere l’indirizzo “127.0.0.1”;
 - Inserendo “Pippo” si sta provando a identificare l’alias Pippo a qualche indirizzo. Purtroppo non esiste nessun alias all’interno del sistema operativo con tale valore, tuttavia su Linux (forse anche su Windows) è possibile creare alias di rete con il relativo indirizzo IP.
-

1.6.5 Esercizio 5 - UDP

(Da fare solo se in Lab Delta) Accordarsi per lavorare su coppie di macchine in modo che server e client siano su macchine diverse. Come bisogna modificare i sorgenti?

Soluzione

La modifica è alquanto semplice. Supponendo che i due host siano connessi sulla stessa rete, quindi non per forza la rete universitaria ma va bene anche un hotspot tramite telefono, è necessario modificare nel seguente modo i codici:

- Server: non apportare nessuna modifica in quanto il server (destinatario) deve solo aprire una porta, nel caso d’esempio la 35000;
- Client: indipendentemente dalla porta aperta nel client, di default nell’esempio la 20000, il client necessita di una modifica nei parametri della chiamata a funzione `UDPSend`. In particolare, al posto di `localhost` o dell’indirizzo “127.0.0.1”, basterà inserire l’indirizzo IP del server che ha all’interno della rete. È doveroso cambiare anche il numero di porta nel caso in cui sia stata cambiata nel server.

Attenzione, nelle virtual machine non è così semplice la faccenda. Infatti esse utilizzano un bridge per collegarsi alla scheda di rete e di conseguenza l’IP che viene visualizzato non è “reale”. Si consiglia dunque un sistema operativo linux (o windows) non virtualizzato.

1.6.6 Esercizio 6 - UDP

Modificare il server in maniera che soddisfi 5 richieste prima di terminare.

- E se volessi che non terminasse mai?

Soluzione

Per soddisfare almeno 5 richieste, è necessario modificare il codice del server di modo che esegua la parte di codice della ricezione almeno 5 volte. Quindi il relativo codice del server sarà:

```
1 #include "network.h"
2
3 int main(void) {
4     socketif_t socket;
5     char response[]="Sono il server: ho ricevuto correttamente il
6     tuo messaggio!\n";
7     char request[MTU];
8     char hostAddress[MAXADDRESSLEN];
9     int port;
10
11    socket = createUDPInterface(10000);
12
13    for (int i = 1; i <= 5; i++)
14    {
15        printf("[SERVER] Sono in attesa di richieste da qualche
16        client\n");
17        UDPReceive(socket, request, MTU, hostAddress, &port);
18        printf("[SERVER] Ho ricevuto un messaggio da host/porta %s
19       /%d\n", hostAddress, port);
20        printf("[SERVER] Contenuto: %s\n", request);
21        UDPSend(socket, response, strlen(response), hostAddress,
22        port);
23        printf("[SERVER] Ho soddisfatto la richiesta numero: %d!\n"
24        , i);
25    }
26 }
```

Mentre quello del client non viene modificato.

La simulazione dell'esecuzione sarà la seguente (prossima pagina):

```

andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ bash ./exe
c_client.sh
[CLIENT] Spedisco messaggio al server
[CLIENT] Contenuto: ciao sono il client!

[CLIENT] Ho ricevuto un messaggio da host/porta 127.0.0.1/35000
[CLIENT] Contenuto: Sono il server: ho ricevuto correttamente il tuo messaggio!

andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ bash ./exe
c_client.sh
[CLIENT] Spedisco messaggio al server
[CLIENT] Contenuto: ciao sono il client!

[CLIENT] Ho ricevuto un messaggio da host/porta 127.0.0.1/35000
[CLIENT] Contenuto: Sono il server: ho ricevuto correttamente il tuo messaggio!

andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ bash ./exe
c_client.sh
[CLIENT] Spedisco messaggio al server
[CLIENT] Contenuto: ciao sono il client!

[CLIENT] Ho ricevuto un messaggio da host/porta 127.0.0.1/35000
[CLIENT] Contenuto: Sono il server: ho ricevuto correttamente il tuo messaggio!

andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ bash ./exe
c_client.sh
[CLIENT] Spedisco messaggio al server
[CLIENT] Contenuto: ciao sono il client!

[CLIENT] Ho ricevuto un messaggio da host/porta 127.0.0.1/35000
[CLIENT] Contenuto: Sono il server: ho ricevuto correttamente il tuo messaggio!
/////
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ █

```

Figura 1: Terminale client.

```

andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ bash ./exe
c_server.sh
[SERVER] Sono in attesa di richieste da qualche client
[SERVER] Ho ricevuto un messaggio da host/porta 127.0.0.1/20000
[SERVER] Contenuto: ciao sono il client!

[SERVER] Ho soddisfatto la richiesta numero: 1!
[SERVER] Sono in attesa di richieste da qualche client
[SERVER] Ho ricevuto un messaggio da host/porta 127.0.0.1/20000
[SERVER] Contenuto: ciao sono il client!

[SERVER] Ho soddisfatto la richiesta numero: 2!
[SERVER] Sono in attesa di richieste da qualche client
[SERVER] Ho ricevuto un messaggio da host/porta 127.0.0.1/20000
[SERVER] Contenuto: ciao sono il client!

[SERVER] Ho soddisfatto la richiesta numero: 3!
[SERVER] Sono in attesa di richieste da qualche client
[SERVER] Ho ricevuto un messaggio da host/porta 127.0.0.1/20000
[SERVER] Contenuto: ciao sono il client!

[SERVER] Ho soddisfatto la richiesta numero: 4!
[SERVER] Sono in attesa di richieste da qualche client
[SERVER] Ho ricevuto un messaggio da host/porta 127.0.0.1/20000
[SERVER] Contenuto: ciao sono il client!

[SERVER] Ho soddisfatto la richiesta numero: 5!
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ █

```

Figura 2: Terminale server.

1.6.7 Esercizio 7 - UDP

Compilare ed eseguire il secondo esempio.

Soluzione

Si esegue il secondo esempio che riguarda il clientUDP e serverUDP in versione _inc, rispettivamente paragrafo 1.5.4 e 1.5.5.

1.6.8 Esercizio 8 - Sommatrice UDP

Modificare il codice in modo tale da costruire una semplice sommatrice:

- Il client acquisisce ripetutamente da tastiera un numero intero e lo invia al server finché l'utente digita zero;
- Il server accumula in una variabile “somma” i valori mandati dal client finché il client manda zero;
- Quando il client manda zero il server risponde al client con la somma ottenuta.

Soluzione

Il codice del client:

```
1 #include "network.h"
2
3 int main(void) {
4     int request;
5     int response;
6     socketif_t socket;
7     char hostAddress[MAXADDRESSLEN];
8     int port;
9
10    socket = createUDPInterface(20000);
11    do
12    {
13        printf("Inserisci un numero intero:\n");
14        scanf("%d", &request);
15        UDPSend(socket, &request, sizeof(request), "127.0.0.1",
16                35000);
17        } while (request != 0);
18        UDPReceive(socket, &response, sizeof(response), hostAddress, &
19        port);
20        printf("[CLIENT] Ho ricevuto un messaggio da host/porta %s/%d\n",
21        hostAddress, port);
22        printf("[CLIENT] Il risultato: %d\n", response);
23    }
```

- (11-16) Viene richiesto l'inserimento di un numero intero all'utente finché tale numero non è diverso da zero. Ogni numero viene inviato al server (15).
- (17-19) Nel momento in cui il numero inserito è zero, il programma termina aspettando il risultato che verrà inviato dal server. Alla ricezione, verrà stampato.

```

andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ bash ./exe
c_client.sh
Inserisci un numero intero:
5
Inserisci un numero intero:
4
Inserisci un numero intero:
-2
Inserisci un numero intero:
0
[CLIENT] Ho ricevuto un messaggio da host/porta 127.0.0.1/35000
[CLIENT] Il risultato: 7
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ █

```

Figura 3: Esempio di esecuzione del client.

Il codice del server:

```

1 #include "network.h"
2
3 int main(void) {
4     int request;
5     int response;
6     socketif_t socket;
7     char hostAddress[MAXADDRESSLEN];
8     int port;
9
10    socket = createUDPIInterface(35000);
11
12    int somma = 0;
13    do
14    {
15        printf("[SERVER] Sono in attesa di valori da qualche client
16 \n");
17        UDPReceive(socket, &request, sizeof(request), hostAddress,
18 &port);
19        printf("[SERVER] Sommo il numero %d...\n", request);
20        somma += request;
21    } while (request != 0);
22    UDPSend(socket, &somma, sizeof(somma), hostAddress, port);
23    printf("[SERVER] Ho ricevuto il valore di terminazione da host/
24 porta %s/%d\n", hostAddress, port);
25    printf("[SERVER] Contenuto: %d\n", request);
26 }

```

- (12-19) Viene dichiarata la variabile somma, come richiesto dall'esercizio. Successivamente, il server attende che qualche client gli invii un valore intero. Una volta arrivato tale informazione, il server somma il valore ad una sua variabile locale. Il ciclo continua finché non riceve un valore pari a zero.
- (20-22) Quando viene ricevuto un valore pari a zero, il server invia la somma effettuata al client e stampa chi è il client che ha richiesto la terminazione.

```
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ bash ./exe
c_server.sh
[SERVER] Sono in attesa di valori da qualche client
[SERVER] Sommo il numero 5...
[SERVER] Sono in attesa di valori da qualche client
[SERVER] Sommo il numero 4...
[SERVER] Sono in attesa di valori da qualche client
[SERVER] Sommo il numero -2...
[SERVER] Sono in attesa di valori da qualche client
[SERVER] Sommo il numero 0...
[SERVER] Ho ricevuto il valore di terminazione da host/porta 127.0.0.1/20000
[SERVER] Contenuto: 0
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$
```

Figura 4: Esempio di esecuzione del server.

1.6.9 Esercizio 9 - Sommatrice UDP e perdita di pacchetti

Usare la sommatrice su due macchine distinte provando, sulla macchina del client, a staccare il cavo di rete prima di un invio di un dato, ad esempio:

- Digitare “2345” + INVIO
- Digitare “5187” + INVIO
- **Staccare il cavo**
- Digitare “2” + INVIO
- **Riattaccare il cavo e aspettare 30 sec che il sistema operativo si riassesti**
- Digitare “1” + INVIO
- “0”

Che somma leggo? È corretta?

Soluzione

Si parte con il rispondere prima alla domanda e poi a fornire la motivazione. La somma letta non è corretta. La motivazione è la seguente:

- Il client invia il valore 2345 al server. Quest’ultimo riceve correttamente il valore. Somma progressiva: 2345;
- Il client invia il valore 5187 al server. Quest’ultimo riceve correttamente il valore. Somma progressiva: $2345 + 5187 = 7532$;
- Viene staccato il cavo;
- Inserimento del valore 2 all’interno del client. Quest’ultimo invia il pacchetto al localhost, il quale non è collegato alla rete, di conseguenza il pacchetto viene perso. Dato che il protocollo è UDP, il client non sa se il pacchetto è stato ricevuto dal destinatario oppure no, di conseguenza ricomincia il ciclo e richiede un numero intero. La somma nel server rimane 7532, mentre la somma corretta dovrebbe essere $7532 + 2 = 7534$;
- Viene riattaccato il cavo;
- Il client invia il valore 1 al server. Quest’ultimo riceve correttamente il valore. Somma progressiva: $7532 + 1 = 7533$;
- Valore zero, il client e il server si fermano.

1.6.10 Esercizio 10 - Sommatrice UDP e influenze reciproche

Invocare il server della sommatrice con due client diversi (tutti e tre possono anche essere sulla stessa macchina ovviamente su finestre terminali diverse), ad esempio:

Client A	Client B
Digitare "2345" + INVIO	Digitare "2" + INVIO
Digitare "5187" + INVIO	Digitare "8" + INVIO
Digitare "2" + INVIO	"0"
Digitare "1" + INVIO	
"0"	

Che somma leggo da ciascun client? È la somma che ciascun client si aspetterebbe?

Soluzione

Mantenendo lo schema dell'esercizio 8 (paragrafo 1.6.8), l'esecuzione dei due client (A e B) come nello schema dell'esercizio:

```
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ bash ./exe
c_client.sh
Inserisci un numero intero:
2345
Inserisci un numero intero:
5187
Inserisci un numero intero:
2
Inserisci un numero intero:
1
Inserisci un numero intero:
0

andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ bash ./exe
c_client.sh
Inserisci un numero intero:
2
Inserisci un numero intero:
8
Inserisci un numero intero:
0
[CLIENT] Ho ricevuto un messaggio da host/porta 127.0.0.1/35000
[CLIENT] Il risultato: 7544
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ 
```

Ricordando di eseguire prima un'operazione del client A, poi un'operazione del client B, poi A, poi B, e così via. Il risultato ottenuto sul server è il seguente:

```

andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/client-server-socket/Modifica$ bash ./exe
c_server.sh
[SERVER] Sono in attesa di valori da qualche client
[SERVER] Sommo il numero 2345...
[SERVER] Sono in attesa di valori da qualche client
[SERVER] Sommo il numero 2...
[SERVER] Sono in attesa di valori da qualche client
[SERVER] Sommo il numero 5187...
[SERVER] Sono in attesa di valori da qualche client
[SERVER] Sommo il numero 8...
[SERVER] Sono in attesa di valori da qualche client
[SERVER] Sommo il numero 2...
[SERVER] Sono in attesa di valori da qualche client
[SERVER] Sommo il numero 0...
[SERVER] Ho ricevuto il valore di terminazione da host/porta 127.0.0.1/20000
[SERVER] Contenuto: 0

```

Quindi, nel client B viene letta la somma corretta, ovvero quella eseguita prima che il client B inviasse il valore zero e richiedesse la terminazione del server. Al contrario, nel client A il valore inserito 1 non viene ricevuto dal server, ma dato che è un protocollo UDP, il client continua ad eseguire il codice. Il problema nel client A sorge nel momento in cui esce dal ciclo do...while, poiché deve attendere una risposta (il risultato) dal server. Tuttavia, dato che il client B ha inviato il valore di terminazione “0” e di conseguenza ha cessato la sua esecuzione, il client A rimarrà in un stato di attesa infinita.

Quale potrebbe essere una **possibile modifica per evitare questa attesa infinita?** Ci sono molteplici soluzioni, una tra queste è quella di inserire un ciclo do...while al termine del codice del server e inserendo al suo interno un'attesa di ricezione con il conseguente invio del valore corretto. Il codice muterebbe in questo modo:

```

1 #include "network.h"
2
3 int main(void) {
4     int request, response;
5     socketif_t socket;
6     char hostAddress[MAXADDRESSLEN];
7     int port;
8
9     socket = createUDPInterface(35000);
10
11    int somma = 0;
12    do
13    {
14        printf("[SERVER] Sono in attesa di valori da qualche client
15 \n");
16        UDPReceive(socket, &request, sizeof(request), hostAddress,
17 &port);
18        printf("[SERVER] Sommo il numero %d...\n", request);
19        somma += request;
20    } while (request != 0);
21    UDPSend(socket, &somma, sizeof(somma), hostAddress, port);
22    printf("[SERVER] Ho ricevuto il valore di terminazione da host/
23 porta %s/%d\n", hostAddress, port);
24    printf("[SERVER] Contenuto: %d\n", request);
25    // Aggiorna i client che si collegano
26    do
27    {
28        UDPReceive(socket, &request, sizeof(request), hostAddress,
29 &port);
30        UDPSend(socket, &somma, sizeof(somma), hostAddress, port);
31    } while (true);
32 }

```

E l'esecuzione del client diventerebbe:

```
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ bash ./exe
c_client.sh
Inserisci un numero intero:
2345
Inserisci un numero intero:
5187
Inserisci un numero intero:
2
Inserisci un numero intero:
1
Inserisci un numero intero:
0
[CLIENT] Ho ricevuto un messaggio da host/porta 127.0.0.1/35000
[CLIENT] Il risultato= 7544
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ □

andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ bash ./exe
c_client.sh
Inserisci un numero intero:
2
Inserisci un numero intero:
8
Inserisci un numero intero:
0
[CLIENT] Ho ricevuto un messaggio da host/porta 127.0.0.1/35000
[CLIENT] Il risultato= 7544
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$
```

Con il server che rimarrebbe in attesa di essere terminato dal programmatore:

```
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ bash ./exe
c_server.sh
[SERVER] Sono in attesa di valori da qualche client
[SERVER] Sommo il numero 2345...
[SERVER] Sono in attesa di valori da qualche client
[SERVER] Sommo il numero 2...
[SERVER] Sono in attesa di valori da qualche client
[SERVER] Sommo il numero 5187...
[SERVER] Sono in attesa di valori da qualche client
[SERVER] Sommo il numero 8...
[SERVER] Sono in attesa di valori da qualche client
[SERVER] Sommo il numero 2...
[SERVER] Sono in attesa di valori da qualche client
[SERVER] Sommo il numero 0...
[SERVER] Ho ricevuto il valore di terminazione da host/porta 127.0.0.1/20000
[SERVER] Contenuto: 0
^C
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ □
```

1.6.11 Esercizio 11 - Sommatrice TCP

Scrivere la sommatrice (quella dell'esercizio 8 al paragrafo 1.6.8) usando TCP, compilare ed eseguire.

Soluzione

Il codice del client:

```
1 #include "network.h"
2
3 int main(void) {
4     connection_t connection;
5     int request, response;
6
7     printf("[CLIENT] Creo una connessione logica col server\n");
8     connection = createTCPConnection("localhost", 35000);
9     if (connection < 0) {
10         printf("[CLIENT] Errore nella connessione al server: %i\n",
11               connection);
12     }
13     else
14     {
15         do
16         {
17             printf("[CLIENT] Inserisci un numero intero:\n");
18             scanf("%d", &request);
19             printf("[CLIENT] Invio richiesta con numero al server\n");
20             TCPSend(connection, &request, sizeof(request));
21             } while (request != 0);
22             TCPReceive(connection, &response, sizeof(response));
23             printf("[CLIENT] Ho ricevuto il seguente risultato dal
24                   server: %d\n", response);
25             closeConnection(connection);
26     }
27 }
```

Un esempio di esecuzione del client:

```
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifico$ bash ./exe
c_client.sh
[CLIENT] Creo una connessione logica col server
[CLIENT] Inserisci un numero intero:
7
[CLIENT] Invio richiesta con numero al server
[CLIENT] Inserisci un numero intero:
6
[CLIENT] Invio richiesta con numero al server
[CLIENT] Inserisci un numero intero:
1
[CLIENT] Invio richiesta con numero al server
[CLIENT] Inserisci un numero intero:
0
[CLIENT] Invio richiesta con numero al server
[CLIENT] Ho ricevuto la seguente risposta dal server: 14
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifico$
```

Il codice del server:

```
1 #include "network.h"
2
3 int main(void) {
4     int request, response;
5     socketif_t socket;
6     connection_t connection;
7
8     socket = createTCPServer(35000);
9     if (socket < 0){
10         printf("[SERVER] Errore di creazione del socket: %i\n",
11               socket);
12     }
13     else
14     {
15         printf("[SERVER] Sono in attesa di richieste di connessione
16               da qualche client\n");
17         connection = acceptConnection(socket);
18         printf("[SERVER] Connessione instaurata\n");
19
20         int somma = 0;
21         do
22         {
23             TCPReceive(connection, &request, sizeof(request));
24             printf("[SERVER] Ho ricevuto la seguente richiesta dal
25                   client: %d\n", request);
26             somma += request;
27             } while (request != 0);
28             printf("[SERVER] Invio il risultato al client\n");
29             TCPSend(connection, &somma, sizeof(somma));
30             closeConnection(connection);
31         }
32     }
```

Un esempio di esecuzione del server:

```
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ bash ./exe
c_server.sh
[SERVER] Sono in attesa di richieste di connessione da qualche client
[SERVER] Connessione instaurata
[SERVER] Ho ricevuto la seguente richiesta dal client: 7
[SERVER] Ho ricevuto la seguente richiesta dal client: 6
[SERVER] Ho ricevuto la seguente richiesta dal client: 1
[SERVER] Ho ricevuto la seguente richiesta dal client: 0
[SERVER] Invio la risposta al client
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$
```

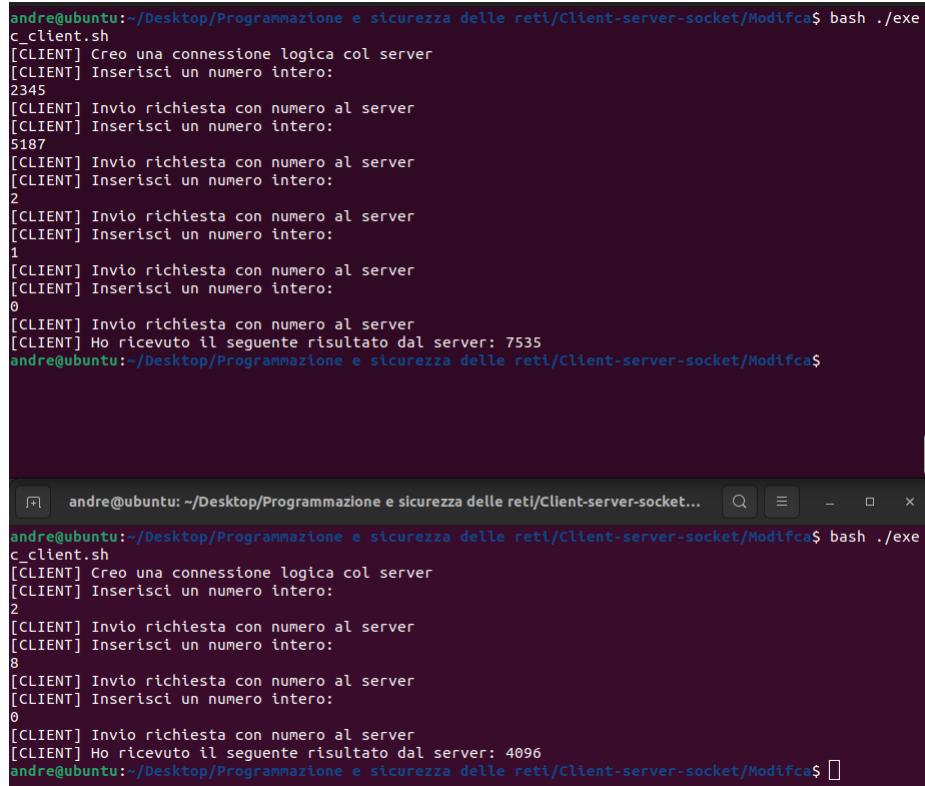
1.6.12 Esercizio 12 - Sommatrice TCP e influenze reciproche

Provare a rifare l'esercizio 10 (paragrafo 1.6.10) ma con questa nuova versione della sommatrice.

Cosa si può osservare? Che soluzioni si può trovare? C'è influenza reciproca tra i due client?

Soluzione

In questo caso, non vi è influenza reciproca poiché il protocollo TCP è più restrittivo:

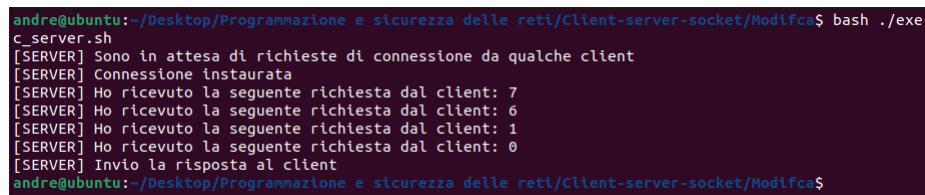


```
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ bash ./exe
c_client.sh
[CLIENT] Creo una connessione logica col server
[CLIENT] Inserisci un numero intero:
2345
[CLIENT] Invio richiesta con numero al server
[CLIENT] Inserisci un numero intero:
5187
[CLIENT] Invio richiesta con numero al server
[CLIENT] Inserisci un numero intero:
2
[CLIENT] Invio richiesta con numero al server
[CLIENT] Inserisci un numero intero:
1
[CLIENT] Invio richiesta con numero al server
[CLIENT] Inserisci un numero intero:
0
[CLIENT] Invio richiesta con numero al server
[CLIENT] Ho ricevuto il seguente risultato dal server: 7535
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$
```



```
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket... ④
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ bash ./exe
c_client.sh
[CLIENT] Creo una connessione logica col server
[CLIENT] Inserisci un numero intero:
2
[CLIENT] Invio richiesta con numero al server
[CLIENT] Inserisci un numero intero:
8
[CLIENT] Invio richiesta con numero al server
[CLIENT] Inserisci un numero intero:
0
[CLIENT] Invio richiesta con numero al server
[CLIENT] Ho ricevuto il seguente risultato dal server: 4096
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$
```

Il server ha la seguente esecuzione:



```
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$ bash ./exe
c_server.sh
[SERVER] Sono in attesa di richieste di connessione da qualche client
[SERVER] Connessione instaurata
[SERVER] Ho ricevuto la seguente richiesta dal client: 7
[SERVER] Ho ricevuto la seguente richiesta dal client: 6
[SERVER] Ho ricevuto la seguente richiesta dal client: 1
[SERVER] Ho ricevuto la seguente richiesta dal client: 0
[SERVER] Invio la risposta al client
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket/Modifica$
```

1.6.13 Esercizio 13 - Sommatrice TCP e perdita di pacchetti

Riprovare l'esercizio 9 (paragrafo 1.6.9) utilizzando questa volta la sommatrice TCP. Si analizzi il risultato.

Soluzione

Nonostante non sia possibile provare questo esercizio in Delta, è possibile formulare la risposta grazie alle conoscenze teoriche sul protocollo TCP.

Nel momento in cui vengono inviati i primi due valori (2345 e 5187), il server riceve correttamente il valore. Successivamente, avviene un down di rete, ovvero viene staccato il cavo. Il client tenta di inviare un pacchetto contenente il valore 2 ma fallisce. Per definizione del protocollo, entra in gioco l'RTO (*Retransmission TimeOut*) che inizia ad aumentare e a riprovare l'invio finché non riesce. Alla fine del down di rete, il server riceve il pacchetto con il valore 2, successivamente riceve il pacchetto inviato dal client con il valore 1 e termina con zero.

2 Dal Web ai Webservices

2.1 Protocollo HTTP/HTTPS

Il **protocollo HTTP** venne inventato per fruire dei contenuti in rete (*World Wide Web*). Tuttavia, al giorno d'oggi viene usato per l'invocazione di funzionalità remote, tecnica chiamata **Webservice**.

Il protocollo si divide in più **fasi**:

1. Apertura di una connessione TCP;
2. Nel caso del protocollo HTTPS, avviene l'autenticazione del server e negoziazione di una chiave di cifratura;
3. Invio di messaggi di *request* e *response*;
4. Chiusura della connessione TCP.

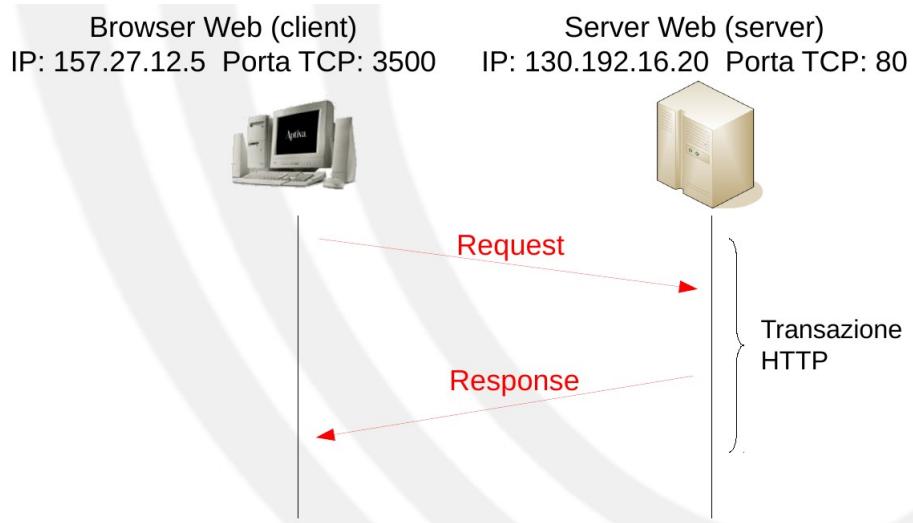


Figura 5: Esempio di scambio di messaggi nel protocollo HTTP.

Nel **protocollo HTTPS** i messaggi che passano nella connessione TCP, sono gli stessi del protocollo HTTP con l'aggiunta di una **cifratura dei dati in transito** e di una **autenticazione del server mediante certificato digitale**. Inoltre, il server lavora sulla porta 443 e non sulla porta classica 80.

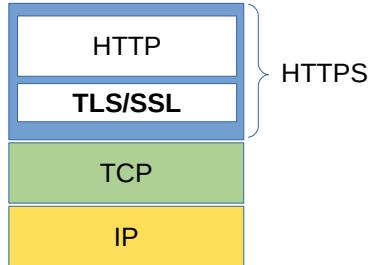


Figura 6: Protocollo HTTPS.

Un **esempio** di richiesta:

```

    GET /it/i-nostri-servizi/servizi-per-studenti HTTP/1.1
    User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.1.8)
    Gecko/20100214 Ubuntu/9.10 (karmic) Firefox/3.5.8
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
    Accept-Language: en-us,en;q=0.5
    Accept-Encoding: gzip,deflate
    Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
    Keep-Alive: 300
    Connection: keep-alive
    !!!Riga vuota!!!
  
```

Figura 7: Messaggio di richiesta.

Un **esempio** di risposta:

```

    HTTP/1.1 200 OK
    Date: Mon, 17 May 2022 16:10:48 GMT
    Server: Apache
    Last-Modified: Mon, 29 Mar 2022 13:57:17 GMT
    Keep-Alive: timeout=15, max=100
    Connection: Keep-Alive
    Transfer-Encoding: chunked
    Content-Type: text/html
    !!!Riga vuota!!!
    <html>
    ...
    </html>
  
```

Le parentesi graffe a destra riuniscono le righe da "HTTP/1.1 200 OK" a "Content-Type: text/html" e sono etichettate come "Intestazione della risposta". Le parentesi graffe a destra riuniscono le righe da "!!!Riga vuota!!!" a "</html>" e sono etichettate come "Corpo della risposta".

Figura 8: Messaggio di risposta.

2.2 Hyper Text Markup Language (HTML) e Cascading Style Sheets (CSS)

HTML è un linguaggio testuale di descrizione di una pagina, in particolare è la specializzazione del generico XML (*eXtensible Markup Language*). HTML si basa sui “tag” annidati, i quali eventualmente contengono attributi.

Questo linguaggio, spesso viene utilizzato con un altro linguaggio chiamato **CSS**.

Esempi di codice HTML:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <link rel="stylesheet" href="styles.css">
5 </head>
6 <body>
7
8 <h1>This is a heading</h1>
9 <p>This is a paragraph.</p>
10
11 </body>
12 </html>
```

E CSS:

```
1 body {
2   background-color: powderblue;
3 }
4 h1 {
5   color: blue;
6 }
7 p {
8   color: red;
9 }
```

2.2.1 HTML: tag per richiamare immagini

Viene utilizzato “img src” per richiamare le immagini:

```
1 <!DOCTYPE html>
2 <html>
3 <body>
4
5 <h2>I trulli di Alberobello</h2>
6 
7
8 </body>
9 </html>
```

2.2.2 HTML: tag per il collegamento ipertestuale

Viene utilizzato “`href`” per il collegamento ipertestuale:

```
1 <!DOCTYPE html>
2 <html>
3 <body>
4
5 <h1>HTML Links</h1>
6
7 <p><a href="https://www.w3schools.com/">Visit W3Schools.com!</a></p>
8
9 </body>
10 </html>
```

2.2.3 Document Object Model (DOM)

Il **Document Object Model (DOM)** è una forma di rappresentazione dei documenti (pagina) strutturati come modello orientato agli oggetti.

2.3 Javascript

Javascript è un linguaggio di programmazione multi paradigma orientato agli eventi, utilizzato sia nella programmazione lato client web che lato server. È facile trovarlo all'interno di codice HTML anche grazie al suo tag riconoscibile: `<script>`.

Tuttavia, è difficile trovare del codice Javascript pure scritto nelle pagine HTML. Solitamente vengono create vere e proprie librerie così da rendere il codice più leggibile e mantenibile.

Un **esempio** di codice Javascript:

```
1 <!DOCTYPE html>
2 <html>
3 <body>
4
5 <h2>Use JavaScript to Change Text</h2>
6 <p>This example writes "Hello JavaScript!" into an HTML element
   with id="demo":</p>
7
8 <p id="demo"></p>
9
10 <script>
11   document.getElementById("demo").innerHTML = "Hello
12   JavaScript!";
13 </script>
14 </body>
15 </html>
```

Javascript trova il suo grande utilizzo con gli **eventi causati dall'utente**, per esempio con la pressione di un bottone all'interno di una pagina web:

```
1 <!DOCTYPE html>
2 <html>
3 <body>
4
5 <button onclick="document.getElementById('demo').innerHTML=Date()">
6 Che ora e'?
7 </button>
8
9 <p id="demo"></p>
10
11 </body>
12 </html>
```

2.3.1 Javascript e Document Object Model (DOM)

Il *Document Object Model* consente di trasformare una pagina web da documento statico a *Graphical User Interface* (GUI), cioè interattivo.

Infatti, grazie al codice Javascript contenuto nella pagina HTML ed eseguito dal browser, l'utente può modificare lo stato della pagina web a seconda di determinate azioni. Quindi, la pagina web si automodifica e assume le sembianze di una applicazione web, chiamata in gergo *web application*.

Alcuni **esempi** di codice interattivo:

- Per avere un riquadro con la pagina web ANSA:

```
1 <!DOCTYPE html>
2 <html>
3 <body>
4
5 <iframe id="area" height="2000" width="1000"></iframe>
6
7
8 <script>
9   document.getElementById("area").src = "https://www.ansa.it/
  sito/notizie/topnews/index.shtml";
10 </script>
11
12 </body>
13 </html>
```

- Per avere un timer che alla fine del tempo stampa “Hello”:

```

1 <!DOCTYPE html>
2 <html>
3 <body>
4
5 <h1>The Window Object</h1>
6 <h2>The setInterval() Method</h2>
7
8 <p id="demo"></p>
9
10 <script>
11 setInterval(displayHello, 1000);
12
13 function displayHello() {
14   document.getElementById("demo").innerHTML += "Hello";
15 }
16 </script>
17
18 </body>
19 </html>
```

- Per avere lo stesso effetto del punto precedente ma utilizzando una **funzione**:

```

1 <!DOCTYPE html>
2 <html>
3 <body>
4
5 <h1>The Window Object</h1>
6 <h2>The setInterval() Method</h2>
7
8 <p id="demo"></p>
9
10 <script>
11 setInterval(function() {document.getElementById("demo").
12   innerHTML += "Hello"}, 1000);
13 </script>
14
15 </body>
16 </html>
```

2.4 Uniform Resource Locator (URL)

Chiamato anche Universal Resource Locator, l'**URL** consente di identificare in maniera univoca una risorsa HTTP in qualsiasi parte della rete mondiale.

È **strutturato** in tre parti:

- Il protocollo utilizzato a livello di applicazione, di trasporto e la porta utilizzata, per esempio HTTP con protocollo TCP e porta 80;
- Nome/IP dell'host che eroga tale risorsa;
- Nome della risorsa con il percorso logico completo.

2.5 Passare dei dati al server web col metodo GET

Il codice HTML per utilizzare il metodo GET è il seguente:

```
1 <!DOCTYPE html>
2 <html>
3 <body>
4
5 <h2>The method Attribute</h2>
6
7 <p>This form will be submitted using the GET method:</p>
8
9 <form action="http://127.0.0.1/action" target="_blank" method="get">
10   <label for="fname">First name:</label><br>
11   <input type="text" id="fname" name="fname" value="John"><br>
12   <label for="lname">Last name:</label><br>
13   <input type="text" id="lname" name="lname" value="Doe"><br><br>
14   <input type="submit" value="Invia">
15 </form>
16
17 <p>After you submit, notice that the form values is visible in the
18   address bar of the new browser tab.</p>
19 </body>
20 </html>
```

La pagina web visualizzata è la seguente:

The method Attribute

This form will be submitted using the GET method:

First name:

Last name:

After you submit, notice that the form values is visible in the address bar of the new browser tab.

Una volta inserito nome e cognome, alla pressione del tasto, i dati saranno inviati al *localhost* aggiungendo come parametri nell'URL `fname=name` e `lname=name`, dove al posto di `name` vengono inseriti nome e cognome. Il link risulta: <http://127.0.0.1/action?fname=John&lname=Doe>.

→ Parametri (max 2048 caratteri)

```
GET /action_page.php?fname=John&lname=Doe HTTP/1.1
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.1.8)
Gecko/20100214 Ubuntu/9.10 (karmic) Firefox/3.5.8
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
!!!Riga vuota!!!
```

Figura 9: La richiesta GET HTTP.

2.6 Passare dei dati al server web col metodo POST

Il codice HTML per utilizzare il metodo POST è il seguente:

```
1 <!DOCTYPE html>
2 <html>
3 <body>
4
5 <h2>The method Attribute</h2>
6
7 <p>This form will be submitted using the POST method:</p>
8
9 <form action="http://127.0.0.1/action" target="_blank" method="post">
10   <label for="fname">First name:</label><br>
11   <input type="text" id="fname" name="fname" value="John"><br>
12   <label for="lname">Last name:</label><br>
13   <input type="text" id="lname" name="lname" value="Doe"><br><br>
14   <input type="submit" value="Submit">
15 </form>
16
17 <p>Notice that the form values is NOT visible in the address bar of
18   the new browser tab.</p>
19 </body>
20 </html>
```

La pagina web visualizzata è la seguente:

The method Attribute

This form will be submitted using the POST method:

First name:

Last name:

Notice that the form values is NOT visible in the address bar of the new browser tab.

Una volta inserito nome e cognome, alla pressione del tasto, i dati saranno inviati al *localhost*. A differenza del metodo GET, i parametri non vengono specificati nell'URL, di conseguenza la sicurezza aumenta. Il link dunque risulta: <http://127.0.0.1/action>.

I valori vengono inseriti all'interno della richiesta HTTP.

```
POST /action_page.php HTTP/1.1
User-Agent: Mozilla/5.0
Accept: text/html
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
!!!Riga vuota!!!
fname=John&lname=Doe
!!!Riga vuota!!!
```

The diagram illustrates an HTTP POST request. It consists of two main parts: the header (Intestazione della richiesta) and the body (Corpo della richiesta). The header includes standard HTTP headers like User-Agent, Accept, and Content-Type, along with specific parameters for the POST method such as fname and lname. The body contains two empty lines, indicated by the text '!!!Riga vuota!!!'.

Figura 10: La richiesta POST HTTP.

2.7 Common Gateway Interface (CGI)

Il **Common Gateway Interface (CGI)** è una tecnologia utilizzata dai *web server* per interfacciarsi con applicazioni esterne generando contenuti web dinamici.

Ogni qualvolta che un *client* richiede al web server un URL corrispondente a un documento HTML, gli viene restituito un documento statico. Al contrario, se l'URL corrisponde a un programma CGI, il server lo esegue in tempo reale, generando dinamicamente informazioni per l'utente. Sostanzialmente è l'**esecuzione di un determinato programma sul server**.

Di conseguenza, il browser diventa il *client* di molte applicazioni di rete, per esempio la posta elettronica.

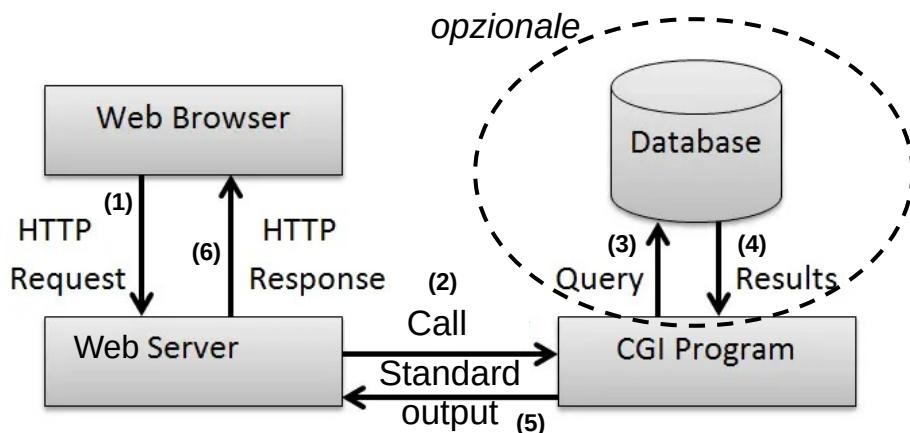


Figura 11: Fasi del CGI.

Degli **esempi** di esecuzione lato server di un programma sono un eseguibile come il client della posta elettronica, oppure un codice PHP, Java, NodeJS, ecc.

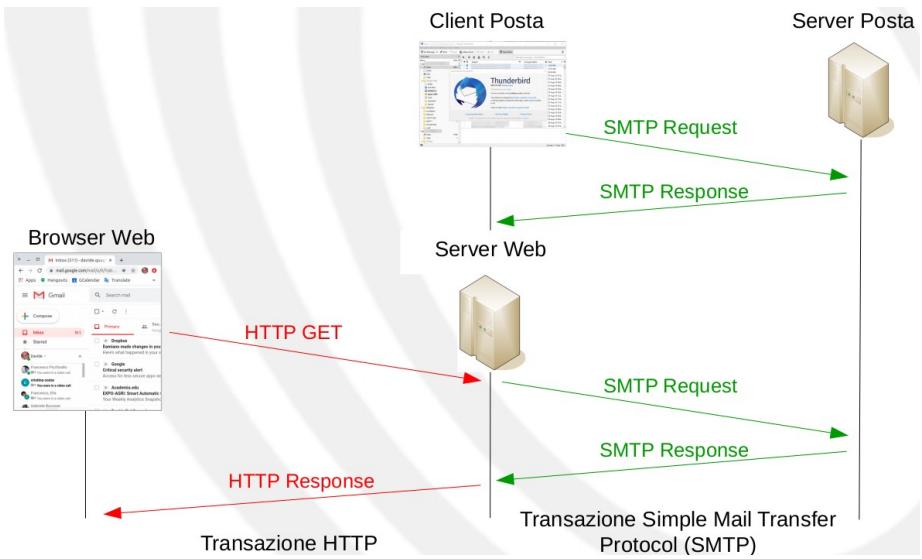


Figura 12: Esempio di CGI, la classica posta elettronica.

2.8 Web socket

La **web socket** è una tecnologia web che fornisce canali di comunicazione chiamati *full-duplex*, cioè bidirezionali, attraverso una singola connessione TCP. Viene utilizzato principalmente per realizzare applicazioni che forniscono contenuti e giochi in tempo reale. Questo perché **il protocollo consente maggiore interazione tra browser e server** grazie al alcune caratteristiche.

Innanzitutto è un **protocollo a livello di applicazione**, per cui è un **metodo alternativo a HTTP e HTTPS**. Ha la caratteristica fondamentale di **comunicazione simmetrica** tra *browser* e *web server*, ovverosia che **i processi possono “prendere l'iniziativa” e inviare dei dati alla controparte**. Inoltre, nasce da una sessione HTTP/HTTPS attraverso un'operazione chiamata **Protocol Upgrade**.

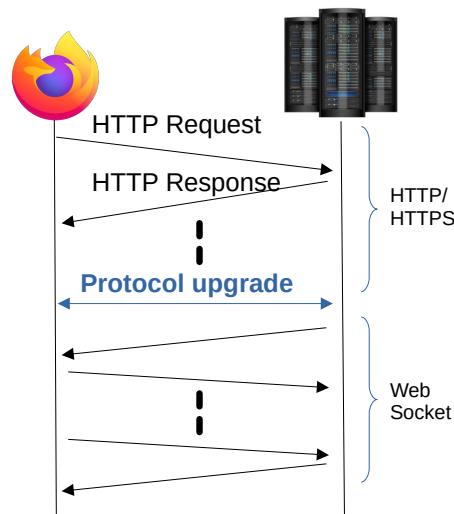


Figura 13: Esempio di web socket e Protocol Upgrade.

Nel messaggio HTTP, ci sono due campi che vengono modificati per indicare il Protocol Upgrade: `Upgrade` e `Connection`. Entrambi i campi vengono modificati con i rispettivi valori `websocket` e `Upgrade`.



Figura 14: Il Protocol Upgrade nel dettaglio.

2.8.1 Approfondimento WebSocket

Il **WebSocket** è un protocollo di comunicazione web che fornisce un canale di **comunicazione bidirezionale** attraverso una singola connessione TCP inizialmente utilizzata per il protocollo HTTP.

Il protocollo consente **maggior interazione tra browser e server**, facilitando inoltre la realizzazione di applicazioni web che devono fornire contenuti in tempo reale. Tutto questo è possibile poiché i **WebSocket concedono al server di “prendere l’iniziativa” ed effettuare dei push autonomi di dati verso il browser per aggiornarlo**. Ovviamente questo non è possibile con il classico protocollo HTTP. Per **comunicazione bidirezionale** si intende una comunicazione in entrambe le direzioni simultaneamente.

I WebSocket sono basati sul protocollo TCP e nascono da una connessione HTTP grazie ad un **Upgrade Request** richiesto dal client al server. Il browser comunica questa richiesta speciale al server tramite alcune voci nell'intestazione del messaggio. Inoltre, il WebSocket consente connessioni per un lungo periodo.

In sintesi, le **caratteristiche** di questo protocollo:

- Si **basa sul protocollo TCP**, la quale inizialmente utilizza il protocollo HTTP, ma grazie alla richiesta speciale Upgrade Request, essa muta nel protocollo WebSocket;
 - **Connessione bidirezionale** (*full-duplex*), quindi aumento della facilità nella realizzazione di applicazioni web che forniscono contenuti in tempo reale;
 - **Utilizzo di porte note** (*Well-known ports*), in particolare quelle dedicate al protocollo HTTP, quindi la 80 e la 443;
 - **Connessioni per un lungo periodo**.
-

2.8.2 Limitazioni

Nonostante la grande utilità che apporta il protocollo WebSocket, esso non è la soluzione a tutto. Infatti, **HTTP ha ancora un ruolo chiave nella comunicazione client-server** per vari motivi:

- **Invio e chiusura delle connessioni per trasferimenti di dati di tipo one-time**, come i caricamenti iniziali. Il protocollo HTTP è più efficiente del WebSocket;
- **Utilizzo più intelligente** da parte di HTTP delle **risorse** grazie alla chiusura delle connessioni una volta terminate le operazioni. Al contrario, il WebSocket mantiene una connessione attiva più a lungo rischiando di sprecare risorse inutilmente;
- **WebSocket riservato solo agli utenti con JavaScript abilitato** e quindi a coloro che posseggono browser moderni a discapito, per esempio, dei sistemi *embedded*.

2.9 WebSocket-Chat

WebSocket-Chat è un progetto “giocattolo” **realizzato per comprendere al meglio la tecnologia offerta dal protocollo WebSocket**. Esso utilizza vari linguaggi di programmazione: JavaScript, Node.js ([framework per realizzare applicazioni Web in JavaScript](#)), HTML, CSS e una parte aggiuntiva chiamata Console con Ispezione Network (monitoraggio da parte del browser con scambio tra i pacchetti).

2.9.1 Node.js e l'approccio asincrono

Il framework **Node.js** ([Wikipedia](#), [sito ufficiale Node.js](#)) è nato per realizzare applicazioni Web in JavaScript. Solitamente viene utilizzato lato client (*client-side*) per realizzare applicazioni tipicamente lato server (*server-side*).

La **caratteristica principale** di Node.js è la possibilità di accedere alle risorse del sistema operativo in modalità *event-driven* ([programmazione orientata agli eventi](#)) e non sfruttando il classico modello basato su processi/thread concorrenti, utilizzato dai classici web server.

Il modello *event-driven*, tradotto in programmazione orientata agli eventi, si basa sulla **mutazione di stato nel momento in cui si manifesta un evento**.

A differenza della programmazione procedurale (*C-style*) in cui ogni azione viene eseguita una dopo l'altra con un determinato ordine, nella programmazione ad eventi le azioni sono asincrone e seguono un ordine dettato dalla manifestazione degli eventi.

L'**approccio asincrono** comporta una **grande efficienza** soprattutto in ambito di *networking* poiché capita spesso di effettuare richieste e di rimanere in attesa di un'eventuale risposta. Grazie all'approccio asincrono, **durante l'attesa possibile effettuare altre operazioni che non dipendono dalla richiesta effettuata**.

2.9.2 Descrizione dell'applicazione

Il progetto mira ad avere una chat multiutente a cui collegarsi tramite browser. Le *features* implementate sono le seguenti:

- Registrazione del nome di contatto che si vuole avere quando si accede alla chat.
- Invio dei messaggi in broadcast a tutti gli utenti attualmente collegati alla chat.
- Visualizzazione dei messaggi inviati col nome della persona che lo ha inviato.

2.9.3 Codice Back-end server

```
1 //Server
2
3 var express = require('express');
4 var socket = require('socket.io');
5
6 //Chat setup
7 var app = express();
8 /*in questo momento il server e' in attesa delle
9 connessioni HTTP sulla porta 4000
10 */
11 var server = app.listen(4000, function(){
12     console.log('waiting for HTTP requests on port 4000,');
13 });
14
15 // Static files
16 /*con questa funzione viene specificato a Node.js che
17 una volta ricevuta una connessione deve andare a
18 cercare nella cartella public il file html da fornire
19 al client
20 */
21 app.use(express.static('public'));
22
23 // Socket setup & pass server
24 /*una volta che la connessione e' stata ricevuta
25 qui viene effettuato l'upgrade ad una connessione
26 websocket e il server si mette in attesa degli
27 eventi ai quali rispondere
28 */
29 var io = socket(server);
30 io.on('connection', function(webSocket){
31
32     console.log('made webSocket connection', webSocket.id);
33
34     // Ricezione di un messaggio da inoltrare ai client
35     webSocket.on('message', function(data){
36         io.sockets.emit('UploadChat', data);
37     });
38 });


```

- (3) **Express.js** è una libreria di Node.js che consente di costruire applicazioni web molto facilmente ([Wikipedia, sito ufficiale](#)). L'unica cosa importante da sapere è che la prima linea di codice crea una variabile **express**, che necessita della relativa libreria Express.js, e viene **utilizzata per creare il server web in ascolto sulla porta 4000**.
- (4) **Socket.IO** è una libreria JavaScript **utilizzata per implementare il protocollo WebSocket** e racchiude molte **funzioni** tra le quali:
 - **Broadcasting** a tutti i socket collegati;
 - **Salvataggio** dei dati riguardanti ciascun utente;
 - **Approccio asincrono di I/O.**

- (6-13) Alla riga 7 viene effettuato il vero e proprio *import* della libreria e viene creato il socket;

Alla riga 11 il server si mette in ascolto, sulla porta 4000, e (riga 12) stampa sul terminale la stringa “waiting for HTTP requests on port 4000.”.

- (21) Una volta ricevuta una connessione da parte di un client, il server ricerca all'interno della cartella chiamata **public**, il relativo file “.html” da inviare al mittente.
 - (29) Alla conferma di connessione instaurata e di file ricevuto, il server prende l'iniziativa ed esegue un **Upgrade Request** trasformando la connessione in una WebSocket.
 - (30-38) Il server rimane in attesa, in particolare questo accade alla linea 35. Nel momento in cui un client scatena un evento, ovvero invia un messaggio con il tag **message**, il server lo inoltrerà a tutti i client connessi mediante l'oggetto **sockets**.
-

2.9.4 Codice Front-end HTML

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <title>WebSockets Chat</title>
6     <script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/2.1.0/socket.io.dev.js"></script>
7     <link href="/styles.css" rel="stylesheet" />
8   </head>
9   <body>
10    <div id="mario-chat">
11      <h2>Chat</h2>
12      <div id="chat-window">
13        <div id="output"></div>
14        <div id="feedback"></div>
15      </div>
16      <h4 id="sender" style="padding-left: 20px">Handle</h4>
17      <input id="message" type="text" placeholder="Message"/>
18      <button id="send">Send</button>
19    </div>
20  </body>
21  <script src="/chat.js"></script>
22 </html>
```

Il codice HTML consente di far visualizzare l'interfaccia utente creata da JavaScript. In particolare, nella pagina è possibile leggere e inviare messaggi.

L'unica osservazione da fare è il tag **<script>**, il quale contiene il link del file **.js** da eseguire all'apertura del file HTML (paragrafo 2.9.5). Ovviamente si intende il codice a riga 21, ovvero quello necessario per la chat.

2.9.5 Codice Front-end JavaScript

```

1 //Client
2
3 var name= prompt("What's your name?");
4 while(name==""){
5     name=prompt("You have to choose a name. \n What's your name?")
6 }
7
8 // Query DOM
9 var message = document.getElementById('message'),
10    sender = document.getElementById('sender'),
11    btn = document.getElementById('send'),
12    output = document.getElementById('output'),
13    feedback = document.getElementById('feedback');
14
15 sender.innerHTML=name;
16 sender.value=name;
17
18 // Invio richiesta di connessione al server
19 var webSocket = io.connect();
20
21 // Listen for events
22 btn.addEventListener('click', function(){
23     if (message.value!=""){
24         webSocket.emit('message', {
25             message: message.value,
26             sender: sender.value,
27         });
28         message.value = "";
29     }
30 });
31
32 webSocket.on('UploadChat', function(data){
33     feedback.innerHTML = '';
34     output.innerHTML += '<p><strong>' + data.sender + ': </strong>' +
35     + data.message + '</p>';
36 });

```

- (3-6) Finché non viene inserito un nome, il codice continua a chiederlo.
- (9-13) Vengono inizializzate le variabili che acquisiscono i tag presenti nella pagina HTML.
- (15-16) Viene scritto il nome dell'utente nella pagina web e impostato il valore.
- (19) Viene creato il socket che deve connettersi al server.
- (22-30) Sul bottone di invio del messaggio, viene aggiunto un nuovo evento JavaScript. Quest'ultimo si attiverà nel momento in cui l'utente cliccherà sul bottone. Una volta premuto, se il messaggio non sarà vuoto, verrà inviato al server (con tag message!) il messaggio (riga 25) e il nome dell'utente (riga 26). Una volta inviato, viene svuotato il valore del messaggio.
- (32-35) Sono la stampa del messaggio ricevuto. Si noti il segno “+=” che indica che i vari messaggi ricevuti vengono concatenati.

2.9.6 Esecuzione del Front-end e del Back-end

Il server web utilizzato è Node.js che consente di eseguire codice JavaScript *server-side* per creare il Back-end. Invece, il Front-end è realizzato mediante codice JavaScript eseguito dentro il browser.

Passaggi da eseguire su Windows:

1. Download del file di installazione dal sito ufficiale: <https://nodejs.org/it/download>
2. Eseguire il Back-end:
 - (a) Scaricare lo zip fornito su Moodle con nome “WebSocket_Chat”;
 - (b) Aprire un terminale e posizionarsi nella cartella contenente il file `server.js`;
 - (c) Eseguire il comando: `node.exe server.js`.
3. Eseguire il Front-end:
 - (a) Aprire il browser alla pagina: <http://localhost:4000>;
 - (b) (Opzionale) Aprire più finestre del browser così da simulare l’accesso da parte di più utenti.

2.9.7 Esercizi

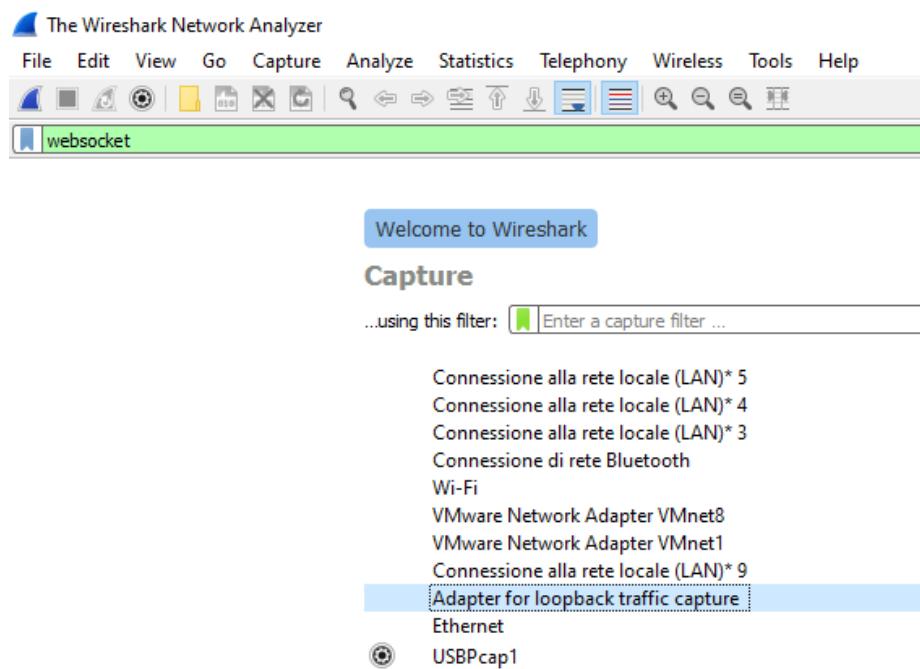
Esercizio 1

Lanciare l'applicazione dopo aver fatto partire l'ispezione del Network tramite la console di sviluppo del browser. Ogni quanto tempo il client fa sapere al server che è ancora connesso? È un'azione dovuta all'implementazione della chat o insita nel WebSocket? A cosa serve tale procedura?

Lanciare Wireshark e vedere cosa passa in rete sulla connessione TCP interessata.

Soluzione esercizio 1

Per analizzare la rete si utilizza il software Wireshark che consente di analizzare il flusso di pacchetti in entrata e in uscita. All'apertura del software, andando nella sezione “*Adapter for loopback traffic capture*” sarà possibile seguire tutti i pacchetti che riguardano il localhost. Per filtrare il risultato dei pacchetti, si inserisce la stringa “websocket” nella barra in alto, così da mostrare solamente quei pacchetti con protocollo WebSocket:



A questo punto, si aprono tre, quattro client. Quindi, si scrive l'URL localhost:4000 nel browser. Dato che il server non è in esecuzione, il browser non riesce a collegarsi al localhost:4000 poiché vede tale porta inutilizzata. Di conseguenza, il traffico catturato da Wireshark è inesistente.

Avviando il server, in automatico vedrà il collegamento dei 3/4 client avviati precedentemente. Di conseguenza, su Wireshark appariranno dei pacchetti corrispondenti al collegamento dei client al server:

No.	Time	Source	Destination	Protocol	Length	Info
3777	234.656388	::1	::1	WebSocket	76	WebSocket Text [FIN] [MASKED]
3779	234.657653	::1	::1	WebSocket	72	WebSocket Text [FIN]
3786	234.763868	::1	::1	WebSocket	71	WebSocket Text [FIN] [MASKED]
3807	235.266748	::1	::1	WebSocket	76	WebSocket Text [FIN] [MASKED]
3809	235.267019	::1	::1	WebSocket	72	WebSocket Text [FIN]
3818	235.279132	::1	::1	WebSocket	76	WebSocket Text [FIN] [MASKED]
3820	235.279339	::1	::1	WebSocket	72	WebSocket Text [FIN]
3824	235.370623	::1	::1	WebSocket	73	WebSocket Text [FIN] [MASKED]
3828	235.390373	::1	::1	WebSocket	71	WebSocket Text [FIN] [MASKED]
3843	236.269003	::1	::1	WebSocket	76	WebSocket Text [FIN] [MASKED]
3845	236.269236	::1	::1	WebSocket	72	WebSocket Text [FIN]
3849	236.376232	::1	::1	WebSocket	71	WebSocket Text [FIN] [MASKED]

Cliccando su uno dei pacchetti con flag [MASKED] è possibile notare una cosa interessante riguardo il protocollo TCP. Ovvero, il numero di porta d'origine e destinazione. Per esempio, nell'immagine è possibile vedere come un client con porta 51021 (*Source Port*) stia comunicando con il server sulla sua porta 4000 (*Destination Port*):

```
▼ Transmission Control Protocol, Src Port: 51021, Dst Port: 4000, Seq: 633, Ack: 130, Len: 12
  Source Port: 51021
  Destination Port: 4000
  [Stream index: 327]
  [Conversation completeness: Incomplete, DATA (15)]
  [TCP Segment Len: 12]
  Sequence Number: 633    (relative sequence number)
  Sequence Number (raw): 286870038
  [Next Sequence Number: 645    (relative sequence number)]
  Acknowledgment Number: 130    (relative ack number)
  Acknowledgment number (raw): 1135635018
  0101 .... = Header Length: 20 bytes (5)
  > Flags: 0x018 (PSH, ACK)
  Window: 10229
  [Calculated window size: 2618624]
  [Window size scaling factor: 256]
  Checksum: 0xfcdb [unverified]
  [Checksum Status: Unverified]
  Urgent Pointer: 0
  > [Timestamps]
  > [SEQ/ACK analysis]
  TCP payload (12 bytes)
  [PDU Size: 12]
```

Adesso che è chiaro quale siano i client (quelli “marchiati” con MASKED e il motivo per cui i messaggi sono mascherati è dovuto ad una questione di sicurezza) e quale il server, è possibile vedere sulla colonna (la terza) di sinistra qual’è il tempo in cui ogni client comunica al server che è ancora vivo:

3915 259.663380	::1	::1	WebSocket	71 WebSocket Text [FIN] [MASKED]
3917 259.663975	::1	::1	WebSocket	67 WebSocket Text [FIN]
3919 261.263680	::1	::1	WebSocket	71 WebSocket Text [FIN] [MASKED]
3921 261.263753	::1	::1	WebSocket	71 WebSocket Text [FIN] [MASKED]
3923 261.264048	::1	::1	WebSocket	67 WebSocket Text [FIN]
3925 261.264493	::1	::1	WebSocket	67 WebSocket Text [FIN]
3927 262.260089	::1	::1	WebSocket	71 WebSocket Text [FIN] [MASKED]
3929 262.260377	::1	::1	WebSocket	67 WebSocket Text [FIN]
3975 284.676735	::1	::1	WebSocket	71 WebSocket Text [FIN] [MASKED]
3977 284.677049	::1	::1	WebSocket	67 WebSocket Text [FIN]
3983 287.263603	::1	::1	WebSocket	71 WebSocket Text [FIN] [MASKED]
3985 287.263652	::1	::1	WebSocket	71 WebSocket Text [FIN] [MASKED]
3987 287.263850	::1	::1	WebSocket	67 WebSocket Text [FIN]
3989 287.264050	::1	::1	WebSocket	67 WebSocket Text [FIN]
3991 288.260063	::1	::1	WebSocket	71 WebSocket Text [FIN] [MASKED]
3993 288.260350	::1	::1	WebSocket	67 WebSocket Text [FIN]

In questo caso, al tempo 259 il client con porta 51021 ha comunicato al server che è ancora vivo. Ovviamente il server ha risposto con un ACK e successivamente gli altri 3 client hanno comunicato al server la loro presenza. Al tempo 284, nuovamente il client con porta 51021 ricomunica al server che è ancora vivo (idem per gli altri). Si deduce che il client fa sapere al server che è ancora connesso ogni 25 secondi circa ($284 - 259 = 25$).

Documentazione ufficiale riguardo al Keep-Alive nel protocollo WebSocket: <https://websockets.readthedocs.io/en/stable/topics/timeouts.html>

RFC documentation: <https://www.rfc-editor.org/rfc/rfc6455#page-36>

Esercizio 2

Modificare il sorgente del codice per fare in modo che ad ogni utente connesso alla chat arrivi nella console il messaggio “l’utente sta scrivendo...”.

NOTA: Lato client, bisogna spedire al server un evento apposito (ad es. “typing”) quando l’utente scrive sulla tastiera (catturando l’evento di sistema “keypress”). Lato server, la chiamata `webSocket.broadcast.emit('typing', data)` rilancia l’evento “typing” a tutti i client connessi tranne che a quello dalla quale si è ricevuto il messaggio. Lato client infine gestire la ricezione del messaggio “typing” che arriva dal server (si veda la gestione del messaggio “UploadChat”).

Soluzione esercizio 2

```
1 //Server
2
3 var express = require('express');
4 var socket = require('socket.io');
5
6 //Chat setup
7 var app = express();
8 /* in questo momento il server e' in attesa delle connessioni
9 HTTP sulla porta 4000
10 */
11 var server = app.listen(4000, function(){
12   console.log('waiting for HTTP requests on port 4000,');
13 });
14
15 // Static files
16 /*con questa funzione viene specificato a Nodejs che
17 una volta ricevuta una connessione deve andare a
18 cercare nella cartella public il file html da fornire
19 al client
20 */
21 app.use(express.static('public'));
22
23 // Socket setup & pass server
24 /*una volta che la connessione e' stata ricevuta qui
25 qui viene effettuato l'upgrade ad una connessione
26 websocket e il server si mette in attesa degli
27 eventi ai quali rispondere
28 */
29 var io = socket(server);
30 io.on('connection', function(webSocket){
31   console.log('made webSocket connection', webSocket.id);
32
33   // Ricezione di un messaggio da inoltrare ai client
34   webSocket.on('message', function(data){
35     io.sockets.emit('UploadChat', data);
36   });
37   webSocket.on('typing', function(data){
38     io.sockets.emit('typing', data);
39   });
40 });

Il codice è rimasto lo stesso (paragrafo 2.9.3), l'unica modifica effettuata è stata dalla riga 38 alla riga 40 in cui si impone al server di inoltrare il messaggio ricevuto, con tag typing, a tutti gli altri client.
```

```

1 //Client
2
3 var name= prompt("What's your name?");
4 while(name==""){
5     name=prompt("You have to choose a name. \n What's your name?")
6 }
7
8 // Query DOM
9 var message = document.getElementById('message'),
10    sender = document.getElementById('sender'),
11    btn = document.getElementById('send'),
12    output = document.getElementById('output'),
13    feedback = document.getElementById('feedback');
14 sender.innerHTML=name;
15 sender.value=name;
16
17 // Invio richiesta di connessione al server
18 var webSocket = io.connect();
19
20 // Trigger with event key down
21 message.onkeydown = function(e) {
22     e = e || window.event;
23     webSocket.emit('typing', {
24         sender: sender.value
25     })
26 }
27
28 // Listen for events
29 btn.addEventListener('click', function(){
30     if (message.value!=""){
31         webSocket.emit('message', {
32             message: message.value,
33             sender: sender.value,
34         });
35         message.value = "";
36     }
37 });
38
39 // UploadChat event
40 webSocket.on('UploadChat', function(data){
41     feedback.innerHTML = '';
42     var current_date = new Date();
43     output.innerHTML += '<p>' + 'Time: ' + current_date.getHours() +
44     ':'
45     + current_date.getMinutes() + ':'
46     + current_date.getSeconds()
47     + ' - ' + '<strong>' + data.sender
48     + ': </strong>' + data.message + '</p>';
49 });
50
51 // Typing event
52 webSocket.on('typing', function(data){
53     if (data.sender != sender.value) {
54         var current_date = new Date();
55         feedback.innerHTML = '<p>' + 'Time: ' + current_date.getHours()
56         () + ':'
57         + current_date.getMinutes() + ':'
58         + current_date.getSeconds()
59         + ' - ' + '<strong>' + data.sender
60         + ': </strong>' + 'typing...' + '</p>';
61     }
62 });

```

Il codice del front-end è rimasto pressoché identico (paragrafo 2.9.5) tranne a due modifiche importanti:

- (20-26) Sull'elemento message della pagina HTML, si crea un evento. Nel momento in cui una lettera viene premuta, il client invierà il messaggio con tag typing al server, inserendo nel payload il nome del mittente (`sender.value`).
 - (50-60) Alla ricezione dell'evento typing da parte del server, il client stamperà la scritta typing... se e solo se supera il controllo alla riga 52, ovvero se non è lui il mittente (come da specifica dell'esercizio).
-

Esercizio 3

Modificare a piacimento il contenuto del file `public/index.html` e valutare l'impatto grafico.

Soluzione esercizio 3

Una modifica che è possibile fare è l'aggiunta di un altro titolo con tag h2. Ovviamente i colori saranno in tema con quelli specificati dal file CSS (`styles.css`).

Esercizio 4

Provare a collegarsi allo stesso server da browser presenti su diversi PC collegati in rete.

Soluzione esercizio 4

In teoria dovrebbe funzionare anche con localhost, ma non ci sono riuscito per cui non posso fornire più di tante informazioni a riguardo.

2.10 Architetture orientate ai servizi (Service-Oriented Architecture, SOA)

Solitamente le applicazioni sono monolitiche, ovvero hanno un’interfaccia utente, la quale richiama delle funzionalità fornite da una serie di librerie linkate in un unico programma che è eseguito sulla macchina dell’utente.

Al giorno d’oggi esiste un altro approccio di sviluppo delle applicazioni che riguarda SOA. Le **architetture orientate ai servizi** (*Service-Oriented Architecture*) riguarda lo sviluppo di applicazioni complesse attraverso la **combinazione di diversi programmi attraverso la rete**:

- L’interfaccia utente e qualche funzionalità di base sono eseguiti sull’host dell’utente.
- Le funzionalità principali dell’applicazione sono fornite da programmi che sono eseguiti su uno o più server.

I **vantaggi** sono molteplici:

- **Potenza di calcolo e memoria** sono delegate al server;
- **Protezione** della proprietà intellettuale su **algoritmi** strategici;
- **Annullamento** della necessità di distribuire **aggiornamenti** software quando le modifiche riguardano solo il codice dei server;
- Nuovo modello economico: **pay per use**;
- **Eliminazione della pirateria**.

Nonostante i grandi vantaggi che propone, c’è un **requisito fondamentale** che deve essere rispettato: **la presenza e l’affidabilità della rete**.

2.10.1 Funzioni remote e webservice

Queste architetture **si basano** completamente sui servizi offerti, ovvero sulle **funzioni remote**.

Le **funzioni remote** hanno il seguente funzionamento. Il **server espone una API¹**, la quale **descrive una serie di funzioni che il client** (non il web browser) **può invocare**. Quindi, l’implementazione effettiva del **codice si trova server-side**, mentre il codice che richiama una funzione specifica dell’API si trova *client-side*.

In passato le tecnologie utilizzate per eseguire una chiamata a funzione remota erano scritte completamente in C e venivano chiamate *Remote Procedure Call* (RPC). Dopodiché, vennero semplificate e programmate in Java tramite il *Java Remote Method Invocation* (JAVA RMI). Successivamente, ci fu l’invenzione del *Common Object Request Broker Architecture* (CORBA) che permise di scrivere e di far comunicare più client/server con linguaggi differenti.

¹ Application program interface (API): insieme delle funzioni/metodi esposte da una certa libreria.

Al giorno d'oggi, grazie all'approdo delle **webservice**, al protocollo HTTP/-HTTPS e alla metodologia REST, le cose sono molto più semplici.

2.10.2 Webservice basati su REST

I webservice basati su REST hanno molteplici vantaggi. Questa metodologia consente di poter utilizzare **due linguaggi di programmazione differenti** *client-side* e *server-side*. Anche l'architettura può essere differente. Questo grazie a due componenti fondamentali:

- **Funzione STUB**, *client-side*: codifica dei parametri trasmessi e la decodifica dei valori di ritorno;
- **Componente SKELETON**, *server-side*: decodificare l'input, eseguire il codice della libreria, codificare l'eventuale risultato e inviarlo al client.

La metodologia REST utilizza il protocollo HTTP/HTTPS. Il suo **servizio principale sono le chiamate remote e per farlo utilizza l'URL**, il quale è stato mappato (*mapping*).

I **parametri possono essere passati tramite URL**, metodo GET, oppure dopo l'header nel **payload**, con il metodo POST o PUT.

Alcune dei metodi HTTP più famosi in base ai quali viene eseguita una funzione specifica:

- **POST**: usato da funzioni che **creano un nuovo oggetto** sul server;
- **PUT**: usato da funzioni che **aggiornano un oggetto esistente** sul server;
- **GET**: usato da funzioni che **recuperano informazioni di un oggetto** sul server;
- **DELETE**: usato da funzioni che **distruggono un oggetto esistente** sul server;

Soltanamente i **valori di ritorno** sono sotto forma di file JSON.

I **vantaggi** sono:

- A differenza di CORBA, il **webservice utilizza il protocollo HTTP/HTTPS**. Di conseguenza, elementi come **firewall o NAT sono già predisposti all'utilizzo** senza troppe complicazioni;
- Il **debugging è facilitato** (e quindi anche lo sviluppo di applicazioni) dall'utilizzo di contenuti testuali nelle transazioni, per esempio i file JSON.

2.10.3 Breve introduzione ai file JSON

I file **JSON** sono dei file testuali nati con JavaScript, ma ad oggi supportati da quasi tutti i linguaggi di programmazione.

La sua è una **struttura gerarchica** facilmente leggibile da un umano e parse-rizzabile da un programma. La sua sintassi è semplice ed è formato da coppie “attributo:valore”.

Possono essere presenti anche array, rappresentati con le parentesi quadre, e strutture dati, rappresentate con le parentesi graffe. Un **esempio** di formattazione JSON:

```
1 {"impiegati": [
2   {
3     "nome": "Giovanni",
4     "cognome": "Rossi"
5   },
6   {
7     "nome": "Anna",
8     "cognome": "Bianchi"
9   },
10  {
11    "nome": "Pietro",
12    "cognome": "Verdi"
13  }
14 ]}
```