

Programmazione e sicurezza delle reti

VR443470

aprile 2023

Indice

1 Scrittura di applicazioni di rete mediante interfaccia socket	5
1.1 Host, processo e applicazione	5
1.2 Modalità di trasmissione in Internet	6
1.2.1 Applicazioni orientate al datagramma (UDP)	6
1.2.2 Applicazioni orientate alla connessione (TCP)	6
1.3 Schemi di applicazioni che utilizzano la rete	7
1.3.1 Modello client/server	7
1.4 Creazione dell'interfaccia Socket	7
1.5 Esempi di codice	8
1.5.1 Esecuzione degli esempi	8
1.5.2 Client UDP	8
1.5.3 Server UDP	9
1.5.4 Client_inc UDP	10
1.5.5 Server_inc UDP	11
1.5.6 Client TCP	12
1.5.7 Server TCP	13
1.5.8 Codice per la copia di un file	14
1.6 Esercizi	15
1.6.1 Esercizio 1 - UDP	15
1.6.2 Esercizio 2 - UDP	15
1.6.3 Esercizio 3 - UDP	16
1.6.4 Esercizio 4 - UDP	17
1.6.5 Esercizio 5 - UDP	17
1.6.6 Esercizio 6 - UDP	18
1.6.7 Esercizio 7 - UDP	20
1.6.8 Esercizio 8 - Sommatrice UDP	20
1.6.9 Esercizio 9 - Sommatrice UDP e perdita di pacchetti	23
1.6.10 Esercizio 10 - Sommatrice UDP e influenze reciproche	24
1.6.11 Esercizio 11 - Sommatrice TCP	27
1.6.12 Esercizio 12 - Sommatrice TCP e influenze reciproche	29
1.6.13 Esercizio 13 - Sommatrice TCP e perdita di pacchetti	30
1.6.14 Esercizio 14 - Trasferimento di un file	31
2 Dal Web ai Webservices	33
2.1 Protocollo HTTP/HTTPS	33
2.2 Hyper Text Markup Language (HTML) e Cascading Style Sheets (CSS)	35
2.2.1 HTML: tag per richiamare immagini	35
2.2.2 HTML: tag per il collegamento ipertestuale	36
2.2.3 Document Object Model (DOM)	36
2.3 Javascript	36
2.3.1 Javascript e Document Object Model (DOM)	37
2.4 Esercizio di HTML e Javascript	39
2.5 Uniform Resource Locator (URL)	39
2.6 Esercizio su un semplice web server	40
2.7 Passare dei dati al server web col metodo GET	43
2.7.1 Esercizio	44
2.8 Passare dei dati al server web col metodo POST	45

2.8.1	Esercizio	46
2.9	Esercizi di modifica del web server	47
2.9.1	Esercizio 1 - Ricerca di una pagina in locale	47
2.9.2	Esercizio 2 - Upload di un file sul server	51
2.10	Common Gateway Interface (CGI)	57
2.10.1	Esercizio web server esteso con gestione CGI	58
2.11	Web socket	61
2.11.1	Approfondimento WebSocket	63
2.11.2	Limitazioni	63
2.12	WebSocket-Chat	64
2.12.1	Node.js e l'approccio asincrono	64
2.12.2	Descrizione dell'applicazione	64
2.12.3	Codice Back-end server	65
2.12.4	Codice Front-end HTML	66
2.12.5	Codice Front-end JavaScript	67
2.12.6	Esecuzione del Front-end e del Back-end	68
2.12.7	Esercizi	69
2.13	Architetture orientate ai servizi (Service-Oriented Architecture, SOA)	75
2.13.1	Funzioni remote e webservice	75
2.13.2	Webservice basati su REST	76
2.13.3	Breve introduzione ai file JSON	77
2.14	Esercizi su webservice basati su REST	78
2.14.1	Esercizio 1 - Analizzare server e clientREST-GET	78
2.14.2	Esercizio 2 - ClientREST in Java	81
2.14.3	Esercizio 3 - Modifica server per calcolare anche i numeri primi	83
2.14.4	Esercizio 4 - ClientThreadREST con thread concorrenti	88
3	Introduzione alla sicurezza	90
3.1	Quali risorse si vogliono proteggere	90
3.1.1	Confidenzialità	90
3.1.2	Integrità	91
3.1.3	Disponibilità	91
3.1.4	Autenticità	91
3.1.5	Tracciabilità	91
3.2	Come vengono minacciate le risorse	92
3.3	Come contrastare le minacce	94
3.3.1	Principi fondamentali di progettazione della sicurezza	95
3.3.2	Politiche di sicurezza e meccanismi	96
3.3.3	Ottenere un sistema sicuro	97
4	Analisi di rete con Wireshark e da linea di comando	98
4.1	Introduzione agli analizzatori di rete	98
4.1.1	Sniffing e la motivazione del sudo	98
4.2	Interfaccia grafica di Wireshark	99
4.2.1	Sniffing della rete	100
4.2.2	Applicazione dei filtri	100
4.2.3	Seguire il flusso di una conversazione	102
4.3	Comando ping	103

4.4	Comando traceroute (tracert Windows)	104
4.5	Comando nslookup	105
4.6	Comando ifconfig (ipconfig Windows)	106
4.7	Comando route (route PRINT Windows)	107
4.8	Comando whois	107
4.9	Esercizi	109
4.9.1	Esercizio 1 - File capture.cap	109
4.9.2	Esercizio 2 - File simpleHTTP.cap	116
4.9.3	Esercizio 3 - File busyNetwork.cap	122
4.9.4	Esercizio 4 - File pingCapture.cap	124
4.9.5	Esercizio 5 - Comando traceroute	125
4.9.6	Esercizio 6 - Interfacce di rete	126
5	Docker	127
5.1	Da quali bisogni è nato Docker?	127
5.2	Virtualizzazione e Containerizzazione	127
5.3	Struttura tecnica di Docker	127
5.3.1	Dockerfile	127
5.3.2	Docker image	127
5.3.3	Docker container	127
5.4	Comandi utili	127
5.5	Esercizi ed esempio di utilizzo	127
5.5.1	Esempio	127
5.5.2	Esercizi	127
5.6	Breve introduzione a Docker Compose	127

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 protocolli 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$
```

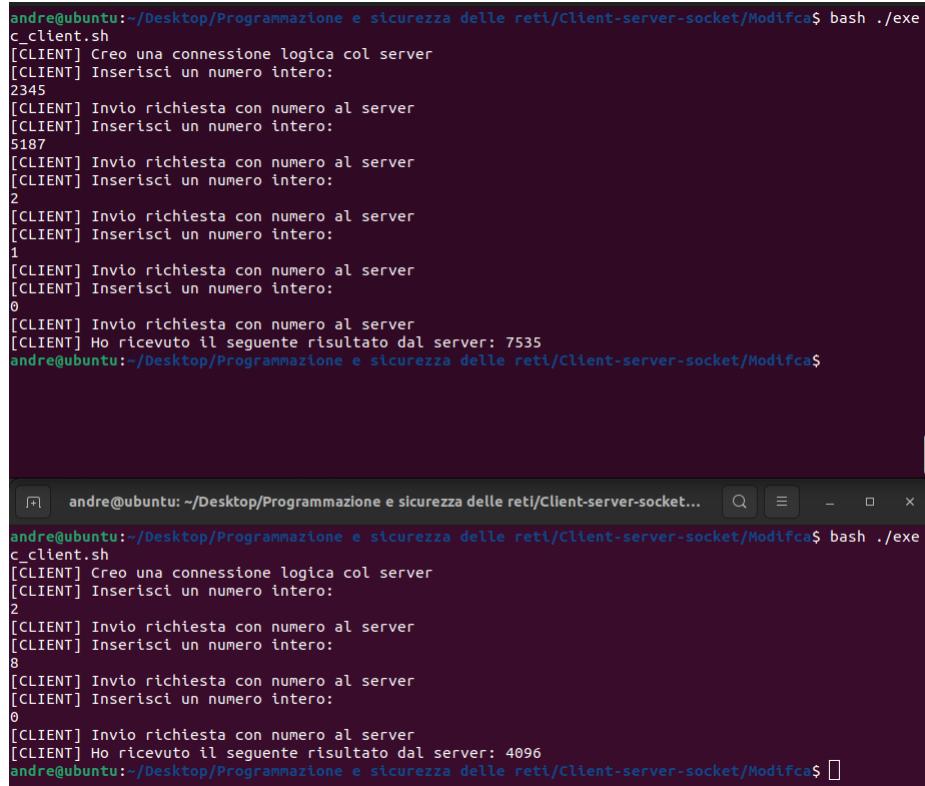
1.6.12 Esercizio 12 - Sommatrice TCP e influenze reciproche

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:

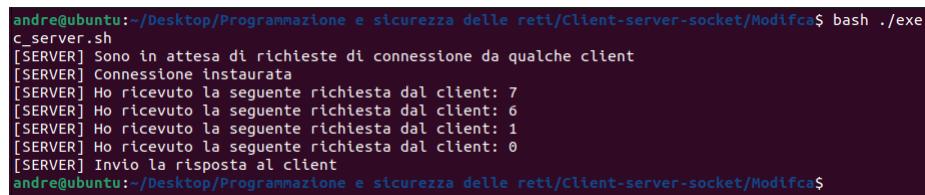


```
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket$ bash ./exe_c_client.sh
[CLIENT] Creo una connessione logica col server
[CLIENT] Inserisci un numero intero:
2345
[CLIENT] Invio richiesta con numero al server
[CLIENT] Inserisci un numero intero:
5187
[CLIENT] Invio 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$
```



```
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket... ④
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket$ 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$
```

Il server ha la seguente esecuzione:



```
andre@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Client-server-socket$ 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$
```

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.

1.6.14 Esercizio 14 - Trasferimento di un file

- Il client chiede al server un file specificandone il nome
- Il server lo trasmette un byte alla volta
- Il client salva in locale con lo stesso nome

Quale protocollo si utilizza? Lanciare client e server su due macchine diverse, trasferire un file di grosse dimensioni in modo da avere il tempo di staccare e riattaccare il cavo di rete. Cosa succede al file trasferito?

Soluzione

Il protocollo utilizzato nella soluzione è il TCP. In questo modo, nel caso di eventuali perdite (ultima domanda), i due host saranno in grado di riprendere la comunicazione. Il codice del client è:

```
1 #include "network.h"
2
3 int main()
4 {
5     FILE *fptr1, *fptr2;
6     char filename[100], c;
7
8     printf("[CLIENT] Inserire il nome del file:\n");
9     scanf("%s", filename);
10
11    fptr1 = createTCPConnectionFD("localhost", 35000);
12
13    for(int i = 0; filename[i] != '\0'; i++)
14        fputc(filename[i], fptr1);
15    fputc('\0', fptr1);
16
17    // apertura del file destinazione in scrittura
18    fptr2 = fopen(filename, "w");
19    if (fptr2 == NULL)
20    {
21        printf("[CLIENT] Errore apertura file %s\n", filename);
22        return 1;
23    }
24
25    // lettura primo byte dalla sorgente
26    c = fgetc(fptr1);
27    while (c != EOF)
28    {
29        // scrittura del byte nella destinazione
30        fputc(c, fptr2);
31        c = fgetc(fptr1);
32    }
33
34    printf("[CLIENT] Contenuto trasferito su %s\n", filename);
35
36    fclose(fptr2);
37    fclose(fptr1);
38    return 0;
39 }
```

Il codice del server è:

```
1 #include "network.h"
2
3 int main()
4 {
5     char filename[MTU], c;
6     socketif_t server;
7     FILE *fptr1, *fptr2;
8     int i;
9
10    server = createTCPServer(35000);
11    if (server < 0){
12        printf("[SERVER] Error: %i\n", server);
13        return -1;
14    }
15
16    while(1){
17        fptr1 = acceptConnectionFD(server);
18
19        i = 0;
20        filename[i] = fgetc(fptr1);
21        while(filename[i]!='\0') {
22            i++;
23            filename[i] = fgetc(fptr1);
24        }
25
26        printf("[SERVER] Nome del file richiesto: %s\n", filename);
27
28        // apertura file fptr2 in lettura
29        fptr2 = fopen(filename, "r");
30        if (fptr2 == NULL)
31        {
32            printf("[SERVER] Errore apertura file %s \n", filename)
33        ;
34            fclose(fptr1);
35            continue;
36        }
37
38        // lettura primo byte dalla fptr2
39        c = fgetc(fptr2);
40        while (c != EOF)
41        {
42            // scrittura del byte nella fptr1
43            fputc(c, fptr1);
44            fflush(fptr1);
45            c = fgetc(fptr2);
46        }
47
48        printf("[SERVER] File inviato\n");
49
50        fclose(fptr2);
51        fclose(fptr1);
52    }
53    return 0;
54 }
```

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.

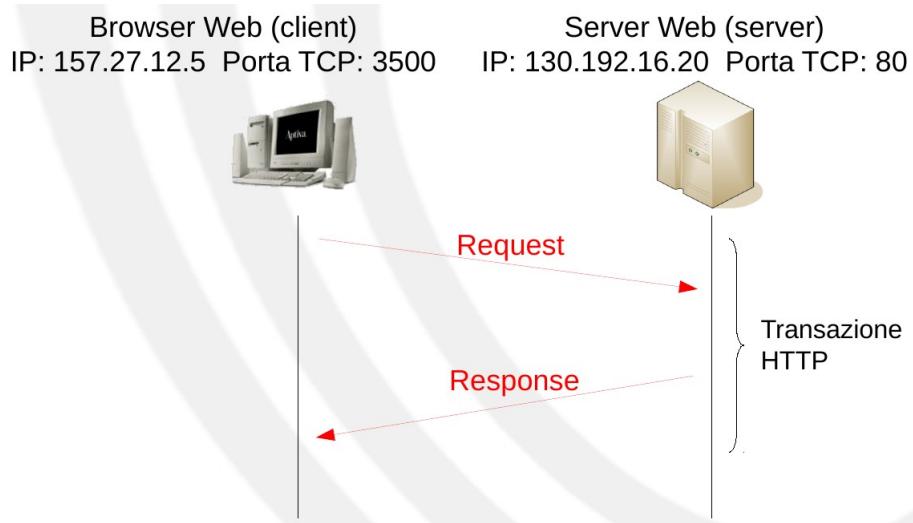


Figura 5: Esempio di scambio di messaggi nel protocollo HTTP.

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.

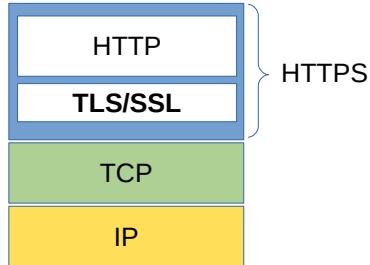


Figura 6: Protocollo HTTPS.

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 Esercizio di HTML e Javascript

Esercizio

Scrivere e provare una pagina HTML che ricarica periodicamente il sito dell'ANSA.

Soluzione

Il seguente codice esegue ogni due secondi il *refresh* del sito:

```
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/
10    notizie/topnews/index.shtml";
11   function autoRefresh() {
12     window.location = window.location.href;
13   }
14   setInterval('autoRefresh()', 2000);
15 </script>
16 </body>
17 </html>
```

2.5 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.

2.6 Esercizio su un semplice web server

Esercizio

- Aprire il file `serverHTTP.c` in `Esempi-web/` e analizzarne il contenuto;
- Compilarlo come nell'esercitazione sull'interfaccia socket ed eseguirlo;
- Provare ad aprire un secondo terminale nella stessa cartella e a rilanciare lo stesso server. Funziona? Perché?
- Aprire il browser preferito e impostare la URL “<http://127.0.0.1:8000/>”. Cosa si vede sul browser e sul terminale?
- Aprire il browser preferito e impostare la URL “<http://localhost:8000/>”. Cosa cambia?

Soluzione

Aprendo il file `serverHTTP.c`, il codice è il seguente:

```
1 #include "network.h"
2
3 int main(){
4     char *HTMLResponse = "HTTP/1.1 200 OK\r\n\r\n<html><head><title>
5 >An Example Page</title></head><body>Hello World, this is a
6 very simple HTML document.</body></html>\r\n";
7     socketif_t sockfd;
8     FILE* connfd;
9     int res, i;
10    long length=0;
11    char request[MTU], method[10], c;
12
13    sockfd = createTCPServer(8000);
14    if (sockfd < 0){
15        printf("[SERVER] Errore: %i\n", sockfd);
16        return -1;
17    }
18
19    while(true) {
20        connfd = acceptConnectionFD(sockfd);
21
22        fgets(request, sizeof(request), connfd);
23        printf("%s", request);
24        strcpy(method,strtok(request, " "));
25        while(request[0]!='\r') {
26            fgets(request, sizeof(request), connfd);
27            printf("%s", request);
28            if(strstr(request, "Content-Length:")!=NULL) {
29                length = atol(request+15);
30                //printf("length %ld\n", length);
31            }
32        }
33        if(strcmp(method, "POST")==0) {
34            for(i=0; i<length; i++) {
35                c = fgetc(connfd);
36                printf("%c", c);
37            }
38        }
39    }
40}
```

```

38         fputs(HTMLResponse , connfd);
39         fclose(connfd);
40
41         printf("\n\n[SERVER] sessione HTTP completata\n\n");
42     }
43
44     closeConnection(sockfd);
45     return 0;
46 }

```

- (4-9) Vengono dichiarate le variabili necessarie per il funzionamento del server:
 - `HTMLResponse` contiene la pagina web da inviare come risposta ai richiedenti;
 - `sockfd` è la variabile utilizzata per creare il socket e il server;
 - `connfd` è il file ricevuto tramite la connessione;
 - `i` è utilizzata all'interno dei cicli;
 - `length` è la lunghezza della richiesta;
 - `request` è la richiesta del client, `method` è il metodo HTTP (POST, GET, ...) e `c` è una variabile temporanea utilizzata per stampare il contenuto del file.
- (11-15) Viene creata la socket sulla porta 8000 e in caso di errore il programma termina.
- (18) Si attende una richiesta di connessione per l'invio di un file descriptor (FD).
- (20-30) Al collegamento di un client, il server stampa l'intero header della richiesta sul terminale.
- (32-37) Se viene eseguita una richiesta POST, viene stampato il contenuto del file passato.
- (39-46) Viene inviata come risposta la pagina web, chiusa la connessione/-socket e fermato il programma.

Una volta compilato ed eseguito sul terminale, il server rimane in attesa di una connessione da parte di un client. Il tentativo di eseguire lo stesso programma fallisce poiché non è possibile creare un'interfaccia socket sulla porta 8000 (già occupata). Collegandosi ad una delle due pagine web (l'URL è lo stesso ma cambia solo l'alias), il risultato nel terminale è il seguente:

```
andrea@ubuntu:~/Desktop/Programmazione e sicurezza delle reti/Esempi-web$ ./serverHTTP
GET / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/112.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: it-IT,it;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1

[SERVER] sessione HTTP completata




Firefox browser window showing the title "An Example Page" and the URL "localhost:8000". The page content is "Hello World, this is a very simple HTML document."


```

2.7 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>.

```

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.

2.7.1 Esercizio

Eseguire il server web `serverHTTP.c` e con il browser preferito aprire il file `form-get.html`. Cosa si vede? Provare ad analizzare il contenuto della connessione TCP con Wireshark. Cosa si vede?

Soluzione

Avviano il server web e apprendo la pagina web, è possibile visualizzare lo stesso form visualizzabile nella pagina precedente.

Eseguendo il metodo GET, quindi cliccando su “Invia”, ed analizzando la comunicazione con Wireshark, è possibile osservare lo scambio di messaggi tra client e server. In particolare, tralasciando i pacchetti del protocollo TCP, è possibile vedere la richiesta GET del client diretta verso il server e la risposta di quest’ultimo con il metodo OK:

No.	Time	Source	Destination	Protocol	Length	Info
4	0.007167095	127.0.0.1	127.0.0.1	HTTP	564	GET /action?fname=John&lname=Doe HTTP/1.1
8	0.007447671	127.0.0.1	127.0.0.1	HTTP	66	HTTP/1.1 200 OK
14	0.078455606	127.0.0.1	127.0.0.1	HTTP	495	GET /favicon.ico HTTP/1.1
18	0.078703514	127.0.0.1	127.0.0.1	HTTP	66	HTTP/1.1 200 OK

È interessante osservare anche come ogni richiesta GET venga stampata sul terminale dal server con una identazione leggibile.

Infine, è possibile notare che nell’URL della pagina web di risposta sono presenti i valori inseriti nel form. Questa è una debolezza del metodo GET che potrebbe essere sfruttata in modo malevolo.

2.8 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.

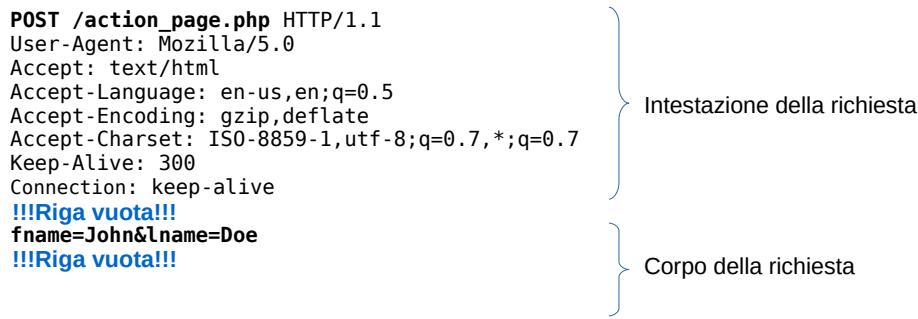


Figura 10: La richiesta POST HTTP.

2.8.1 Esercizio

Eseguire il server web `serverHTTP.c` e con il browser preferito aprire il file `form-post.html`. Cosa si vede? Provare ad analizzare il contenuto della connessione TCP con Wireshark. Cosa si vede?

Soluzione

Avviano il server web e aproendo la pagina web, è possibile visualizzare lo stesso form visualizzabile nella pagina precedente.

Eseguendo il metodo POST, quindi cliccando su “Invia”, ed analizzando la comunicazione con Wireshark, è possibile osservare lo scambio di messaggi tra client e server. In particolare, tralasciando i pacchetti del protocollo TCP, è possibile vedere la richiesta POST del client diretta verso il server e la risposta di quest’ultimo con il metodo OK:

È interessante osservare anche come ogni richiesta POST venga stampata sul terminale dal server con una identazione leggibile.

A differenza del metodo GET, il metodo POST non consente di visualizzare i valori direttamente nell'URL, ma solo all'interno della richiesta. Quindi, è necessario uno *sniffer* per catturare i dati.

2.9 Esercizi di modifica del web server

2.9.1 Esercizio 1 - Ricerca di una pagina in locale

Modificare il file `serverHTTP.c` in modo che, invece di restituire sempre la solita pagina web di prova, restituisca una delle pagine html usate nel capitolo attraverso l'uso del browser. Suggerimenti:

- Provare a fare la nuova richiesta col browser usando il `serverHTTP.c` in modo da vedere la richiesta HTTP per capire dove si trova la stringa con il nome del file nella richiesta che fa il browser (aiutarsi anche con Wireshark)
 - Cosa devo scrivere nella barra del browser?
- Capire come recuperare la stringa con il nome del file della richiesta HTTP (si veda `esempio-parser.c`);
- Per costruire la risposta riciclare parte del codice usato nell'esercizio del trasferimento di file.

Prova finale: cosa succede se chiedo al server di restituire i file `form-get.html` e `form-post.html`?

Soluzione

Il codice necessita di alcune modifiche. L'esercizio richiede l'accesso completo alla cartella in cui è presente il server. Infatti, il client deve essere in grado di richiamare la pagina all'interno della cartella, scrivendo nell'URL la pagina interessata. Per esempio, scrivendo `http://localhost:8000/css.html`, il server dovrebbe inviare il file `css.html` al client e quest'ultimo visualizzare la pagina web. Il codice:

```
1 #include "network.h"
2
3 int main(){
4     socketif_t sockfd;
5     FILE* connfd;
6     int i;
7     long length=0;
8     char request[MTU], method[10], c;
9     char tmp[MTU];
10    char *f_split, *page_req;
11    char *buffer = 0;
12    FILE *fptr1;
13
14    sockfd = createTCPServer(8000);
15    if (sockfd < 0){
```

```

16     printf("[SERVER] Errore: %i\n", sockfd);
17     return -1;
18 }
19
20 while(true) {
21     connfd = acceptConnectionFD(sockfd);
22
23     fgets(request, sizeof(request), connfd);
24     printf("%s", request);
25     // save temp the request
26     strcpy(tmp, request);
27     strcpy(method,strtok(request, " "));
28     while(request[0]!='\r') {
29         fgets(request, sizeof(request), connfd);
30         printf("%s", request);
31         if(strstr(request, "Content-Length:")!=NULL) {
32             length = atol(request+15);
33             //printf("length %ld\n", length);
34         }
35     }
36
37     if(strcmp(method, "POST")==0) {
38         for(i=0; i<length; i++) {
39             c = fgetc(connfd);
40             printf("%c", c);
41         }
42     }
43
44     // take the page requested
45     f_split = strtok(tmp, "/ ");
46     page_req = strtok(NULL, "/ ");
47     // debug: printf("The page requested: %s\n", page_req);
48
49     // open page web requested to send the content
50     fptr1 = fopen(page_req, "r");
51     if (fptr1 == NULL)
52     {
53         // if the page doesn't exist, error 404
54         printf("Cannot open file %s \n", page_req);
55         fputs("HTTP/1.1 404 NOT FOUND\r\n\r\n<html><head><title>404 Not Found</title></head><body>404 Page Not Found.</body></html>\r\n", connfd);
56     }
57     else
58     {
59         // if the page exists, read contents from file.
60         // so, go to the EOF
61         fseek(fptr1, 0, SEEK_END);
62         // take the length
63         length = ftell(fptr1);
64         // come back to the start of file
65         fseek(fptr1, 0, SEEK_SET);
66         // read and save contents of file
67         buffer = malloc(length);
68         if (buffer)
69             fread(buffer, 1, length, fptr1);
70         else
71         {
72             printf("Error malloc\n");
73             exit(-1);
74         }
75         fclose(fptr1);

```

```

76     // check errors and send to client
77     if (buffer)
78     {
79         // send OK code
80         fputs("HTTP/1.1 200 OK\r\n\r\n", connfd);
81         // send web page
82         fputs(buffer, connfd);
83         fputs("\r\n", connfd);
84     }
85     else
86     {
87         printf("Error fread\n");
88         exit(-1);
89     }
90     free(buffer);
91 }
92
93     // close File Descriptor
94     fclose(connfd);
95     printf("\n\n[SERVER] sessione HTTP completata\n\n");
96 }
97
98     closeConnection(sockfd);
99     return 0;
100 }
```

- (4-12) Vengono dichiarate nuove variabili ed eliminate le vecchie. Vengono dichiarate variabili per eseguire lo *split*, necessario per capire quale file ha richiesto il client, per eseguire la lettura del file in locale (*buffer*) e un puntatore al file.
- (23-27) Oltre a prendere la richiesta (GET, POST, ecc), viene copiato il contenuto di *request*, il quale verrà modificato alla riga 27 a causa della funzione *strtok*¹.
- (44-47) Come detto in precedenza, vengono eseguite due *strtok* per acquisire il nome del file inserito dall'utente. Dato che la richiesta sarà nella forma del tipo:

GET /form-get.html HTTP/1.1

Viene utilizzato un primo delimitatore *slash* e *spazio* (“/ ”) per dividere la stringa in più parti e infine viene riutilizzata la funzione *strtok* per prendere il nome del file.

- (49-56) Il server tenta di aprire il file specificato dal client. Nel caso in cui il puntatore *fptr1* rimanga nullo, il file o non esiste o non può essere aperto. Di conseguenza, il server risponde al server con un 404 Not Found e stampa l'errore.

¹La funzione *strtok* esegue una divisione di stringhe a seconda del delimitatore impostato ([link approfondimento](#))

- (57-85) In questa parte di codice l'obiettivo è leggere il contenuto del file richiesto dall'utente e inviarglielo come pagina web.
 - (61) La funzione `fseek` cambia la posizione del puntatore all'interno di un file. In questo caso, è utile posizionarsi alla fine del file (`SEEK_END`) per capire la grandezza del file;
 - (63) Come accennato precedentemente, grazie alla funzione `ftell` è possibile ottenere la grandezza del file. O meglio, quanti caratteri ci sono all'interno così di allocare uno spazio in memoria della grandezza esatta;
 - (65) Dopo aver calcolato il numero di caratteri dentro il file, il puntatore del file stream viene riposizionato all'inizio (`SEEK_SET`);
 - (67-69) Allocazione del buffer della lunghezza equivalente al numero di caratteri dentro il file e controllo di eventuali errori di allocazione. Nel caso in cui l'allocazione sia stata eseguita correttamente, viene letto l'intero file e salvato ogni carattere all'interno della stringa `buffer`;
 - (77-84) Se durante la lettura da file non ci sono stati errori, viene inviata una risposta affermativa al client, quindi un codice 200 del protocollo HTTP; viene inviato l'intero contenuto del buffer, quindi del file; infine, vengono inviati dei caratteri di identazione (non necessari).

2.9.2 Esercizio 2 - Upload di un file sul server

Modificare il server web `serverHTTP.c` in modo che accetti con il metodo POST un intero file da salvare nella cartella del server (il cosiddetto “upload sul server”). Suggerimenti:

- Utilizzare il file `form-file.html` con `serverHTTP.c` non modificato ed analizzare il contenuto della connessione TCP con Wireshark in modo da capire nella richiesta HTTP:
 - Dove si trova il nome del file
 - Dove si trova il contenuto del file
- Nella lettura della richiesta HTTP sul server aggiungere il codice che salva il file prendendo spunto dal codice usato nell'esercizio del trasferimento di file;
- Invece la risposta HTTP può essere molto statica come nella versione originale di `serverHTTP.c`.

Soluzione

Prima di parlare del codice, è necessario fare una premessa. Quando viene inviato un file (immagine, testuale, audio, video, ...) tramite il protocollo HTTP, esso viene “immagazzinato” all'interno di un MIME (Multipurpose Internet Mail Extensions). Tale protocollo racchiude il file all'interno di due limitatori, chiamati *boundary*. All'interno di questi limitatori, è possibile trovare alcuni dati come il Content-Type o il filename (parametro da tenere sotto osservazione per questo esercizio). Una buona introduzione al protocollo è possibile trovarla qui: [Wikipedia](#).

Il codice si presenta un po' lungo e apparentemente complesso, ma viene fornita una spiegazione dettagliata a fine paragrafo:

```
1 #include "network.h"
2
3 int main(){
4     char *HTMLResponse = "HTTP/1.1 200 OK\r\n\r\n<html><head><title>
5 >An Example Page</title></head><body>Hello World, this is a
6 very simple HTML document.</body></html>\r\n";
7     socketif_t sockfd;
8     FILE *connfd, *fptr1;
9     long length=0;
10    char request[MTU], method[10], c;
11
12    sockfd = createTCPServer(8000);
13    if (sockfd < 0){
14        printf("[SERVER] Errore: %i\n", sockfd);
15        return -1;
16    }
17
18    while(true) {
19        connfd = acceptConnectionFD(sockfd);
20
21        fgets(request, sizeof(request), connfd);
22        printf("%s", request);
23        strcpy(method,strtok(request, " "));
```

```

22     while(request[0]!='\r') {
23         fgets(request, sizeof(request), connfd);
24         printf("%s", request);
25         if(strstr(request, "Content-Length:")!=NULL) {
26             length = atol(request+15);
27             //printf("length %ld\n", length);
28         }
29     }
30
31     if(strcmp(method, "POST")==0)
32     {
33         // filename to rcv
34         char filename[100] = "";
35         // flag used to obtain the filename string
36         bool flag = 1;
37
38         // read MIME part
39         while(true)
40         {
41             // take a letter
42             c = fgetc(connfd);
43             printf("%c", c);
44
45             /* verify if it's filename */
46             if (flag == 1 && c == ';')
47             {
48                 /* read (space) */
49                 c = fgetc(connfd);
50                 printf("%c", c);
51
52                 /* read f */
53                 c = fgetc(connfd);
54                 printf("%c", c);
55                 if (c == 'f')
56                     // jump to the end of the word...
57                     for (int j = 0; j < 7; j++)
58                     {
59                         c = fgetc(connfd);
60                         printf("%c", c);
61                     }
62                 // ...and verify if it's an 'e'
63                 if (c == 'e')
64                 {
65                     /* read = */
66                     c = fgetc(connfd);
67                     printf("%c", c);
68
69                     /* read " */
70                     c = fgetc(connfd);
71                     printf("%c", c);
72
73                     /* read name of file */
74                     while (true)
75                     {
76                         c = fgetc(connfd);
77                         printf("%c", c);
78                         // read until there is a letter or
79                         other
80                         if (c == ',')
81                         {
82                             flag = 0;
83                             break;

```

```

83         }
84     else
85         // build filename
86         strncat(filename, &c, 1);
87     }
88 }
89
90 /* verify if it's the start of file contents */
91 if (c == '\r')
92 {
93     /* read \n */
94     c = fgetc(connfd);
95     printf("%c", c);
96     if (c == '\n')
97     {
98         /* read another \r */
99         c = fgetc(connfd);
100        printf("%c", c);
101        if (c == '\r')
102        {
103            /* read another \n */
104            c = fgetc(connfd);
105            printf("%c", c);
106            if (c == '\n')
107            {
108                /* INSIDE THE FILE */
109                // Create local file
110                fptr1 = fopen(filename, "w");
111                if (fptr1 == NULL)
112                {
113                    printf("Cannot open file %s \n"
114 , filename);
115                    exit(-1);
116                }
117
118                /* read every letter inside the
file and
119                 write it in the new file */
120                 c = fgetc(connfd);
121                 // until last boundary
122                 while (c != '\r')
123                 {
124                     printf("%c", c);
125                     fputc(c, fptr1);
126                     fflush(fptr1);
127                     c = fgetc(connfd);
128                 }
129                 fclose(fptr1);
130
131             /* read \r */
132             c = fgetc(connfd);
133
134             /* read \n */
135             c = fgetc(connfd);
136
137             /* clean the pipe of connection */
138             while (c != '\n')
139                 c = fgetc(connfd);
140                 // exit
141                 break;
142 }

```

```

143
144
145
146
147
148
149     fputs(HTMLResponse, connfd);
150     fclose(connfd);
151
152     printf("\n\n[SERVER] sessione HTTP completata\n\n");
153 }
154
155 closeConnection(sockfd);
156 return 0;
157 }
```

- (4-8) Alcune variabili sono le solite del web server iniziale, mentre altre, come `fptr1`, sono necessarie alla copia dei caratteri all'interno del nuovo file in locale sul server.
- (10-29) Il codice è lo stesso e la spiegazione è possibile trovarla nel codice sorgente (paragrafo 2.6).
- (31) Se il metodo è di tipo POST, il codice accoglie il file, altrimenti invia la pagina di default classica (`HTMLResponse`) e chiude la connessione di file descriptor.
- (33-36) La variabile `filename` è necessaria per salvare il nome del file. Invece, la variabile `flag` viene utilizzata per verificare che prima del nome del file all'interno del pacchetto HTTP, ci sia la stringa `filename`. Quindi, se sia effettivamente il nome del file e non ci si trovi in un altro punto.
- (39-148) Il vero *core* del codice. Il ciclo è apparentemente infinito e continua a leggere dalla pipe socket, tutti i dati inviati dal client. L'iterazione si conclude nel momento in cui è stata svuotata l'intera pipe:
 - Prima di iniziare a descrivere il codice, è importante capire la struttura di un MIME. Nella seguente immagine è possibile osservare la struttura del pacchetto dopo l'upload di un file chiamato “`upload.sh`”:

```

> Frame 235: 1089 bytes on wire (8712 bits), 1089 bytes captured (8712 bits) on interface lo, id 0
> Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 40992, Dst Port: 8000, Seq: 1, Ack: 1, Len: 1023
> Hypertext Transfer Protocol
> MIME Multipart Media Encapsulation, Type: multipart/form-data, Boundary: "-----3063446421998555818734781761-----"
  [Type: multipart/form-data]
  First boundary: -----3063446421998555818734781761-----\r\n
  > Encapsulated multipart part: (application/x-shellsCript)
    Content-Disposition: form-data; name="myfile"; filename="update.sh"\r\n
    Content-Type: application/x-shellsCript\r\n\r\n
  > Media Type
    Media type: application/x-shellsCript (175 bytes)
Last boundary: \r\n-----3063446421998555818734781761--\r\n

```

Figura 11: Esempio di struttura del protocollo MIME.

Nell'immagine è possibile vedere i due *boundary* che delimitano il contenuto. All'interno è possibile trovare il nome del file (`update.sh`),

il tipo di contenuto, quindi uno script della shell (*shellscript*) e ovviamente il contenuto del file in bytes.

N.B.: a fine di ogni linea ci sono alcuni *escape characters*. Essi sono molto importanti.

- (41-89) Per leggere il nome del file, si esegue questo pezzo di codice:

- * (41-46) Viene effettuata la lettura del contenuto. Nel caso in cui la **flag** sia a 1, quindi ancora non è stato trovato il **filename**, e il carattere letto corrisponde al punto e virgola ;, si inizia la possibile lettura del **filename**.

Perché il carattere letto deve essere ;? Analizzando il pacchetto MIME, è facilmente visibile che uno dei pochi parametri che non può cambiare è il ; o il form-data, ecc. In questo caso, viene preso il punto e virgola come riferimento.

- * (48-61) Supponendo quindi che la lettura di ogni singolo carattere sia vicino a **filename**, eseguendo due letture (49 e 53), si dovrebbe leggere uno spazio e poi la **f** (iniziale del parametro **filename**). Per verificare che la supposizione sia vera, viene controllata tale condizione (55). Se la parola inizia con la **f**, per avere una certezza in più, e anche per avanzare con la lettura del file, vengono eseguite 7 letture, quindi partendo da **f**:

1. lettura, lettera **i**
2. lettura, lettera **l**
3. lettura, lettera **e**
4. lettura, lettera **n**
5. lettura, lettera **a**
6. lettura, lettera **m**
7. lettura, lettera **e**

- * (62-71) Se la sequenza è corretta, l'ultima parola letta dovrebbe essere una **e** e quindi viene verificata tale condizione (63). Il parametro **filename** all'interno del MIME è seguito poi da un uguale e un doppio apice (di solito **filename** viene scritto così dentro il MIME: **filename="nome_file"**). Vengono effettuate due letture per scartare, e stampare, tali valori.

- * (73-89) Viene creato un piccolo ciclo while che ha l'obiettivo di leggere qualsiasi carattere che si trova all'interno del **filename**. La lettura (76) e la concatenazione nella stringa risultato (86) continuano finché non viene trovato il doppio apice (79) che chiude il valore del **filename**. Viene anche impostata la **flag** a zero così da evitare eventuali controlli in futuro del **filename** poiché esso è già stato acquisito.

- (91-148) Per leggere il contenuto del file che si trova all'interno dei *boundary*, si esegue questo pezzo di codice:
 - * (91-92) Osservando la struttura del MIME a pagina 54, è possibile notare che ogni riga del protocollo termina con un \e e \n. Quindi, è possibile sfruttare questa caratteristica per capire quando inizia il contenuto del file.
 - * (94-108) In questa parte di codice vi è una serie di letture di caratteri. Viene controllata questa alternanza poiché il contenuto del file inizia dopo la sequenza: \r\n\r\n. Quindi, nel caso in cui vi è questa alternanza, il puntatore è all'interno del file.
 - * (109-116) Creazione banale in locale di un file che ha nome ed estensione quella che è stata ricevuta. L'apertura avviene in scrittura e viene controllato l'errore.
 - * (118-129) Inizia la lettura/scrittura vera e propria del file. Quindi, fintantoché non vi è il carattere che si trova alla fine del file, ovvero \r, vengono copiati tutti i caratteri all'interno del nuovo file. Una volta copiati nel nuovo file, quest'ultimo viene chiuso.
 - * (131-148) Avvengono una serie di letture della pipe per consentire di svuotarla.
- (149-157) Il server invia una risposta di positiva di avvenuta ricezione, inviando la pagina di default. Infine, la connessione viene chiusa.

2.10 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.

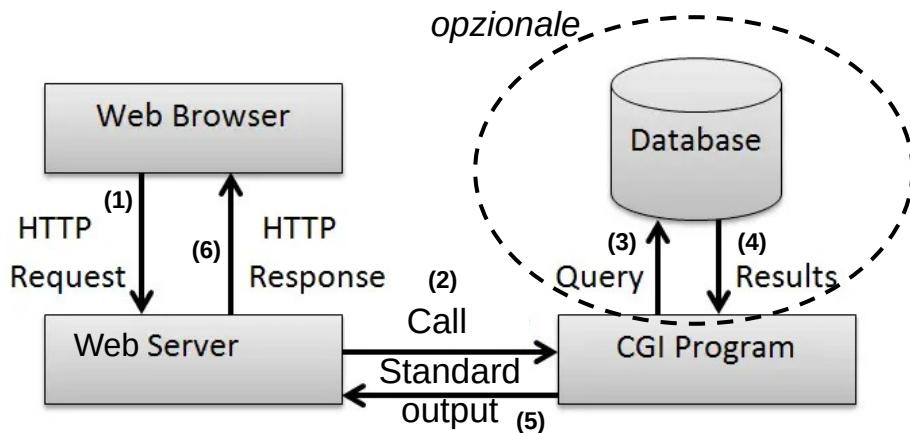


Figura 12: Fasi del CGI.

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.

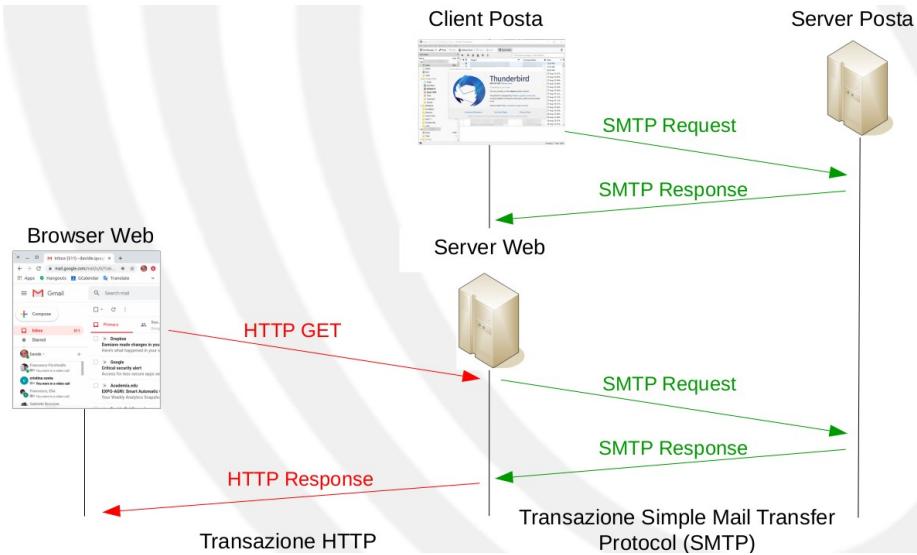


Figura 13: Esempio di CGI, la classica posta elettronica.

2.10.1 Esercizio web server esteso con gestione CGI

Aprire il file `serverHTTP-CGI.c` in `Esempi-web/` e analizzarne il contenuto. Compilarlo come al solito ed eseguirlo. Aprire il browser preferito e il file `sommatrice-web.html`:

- Cosa si vede sul browser?
- NOTA: Provare con numeri positivi, negativi, con parte decimale...

A cosa corrisponde il secondo parametro della funzione `sommatrice()`?

Soluzione

Il codice del `serverHTTP-CGI.c` è il seguente:

```

1 #include "network.h"
2
3 void sommatrice(char *url, FILE *out) {
4     char *function, *op1, *op2;
5     float somma, val1, val2;
6
7     function = strtok(url, "?&");
8     op1 = strtok(NULL, "?&");
9     op2 = strtok(NULL, "?&");
10    strtok(op1, "=");
11    val1 = atof(strtok(NULL, "="));
12    strtok(op2, "=");
13    val2 = atof(strtok(NULL, "="));
14
15    somma = val1 + val2;
16

```

```

17     fprintf(out,"HTTP/1.1 200 OK\r\n\r\n<html><head><title>Risultato
18     </title></head><body>Risultato=%f</body></html>\r\n\r\n", somma
19 );
20 }
21
22 int main(){
23     socketif_t sockfd;
24     FILE* connfd;
25     int res, i;
26     long length=0;
27     char request[MTU], url[MTU], method[10], c;
28     char *html="HTTP/1.1 200 OK\r\n\r\n<html><head><title>An
29     Example Page</title></head><body>Hello World, this is a very
30     simple HTML document.</body></html>\r\n\r\n";
31
32     sockfd = createTCPServer(8000);
33     if (sockfd < 0){
34         printf("[SERVER] Errore: %i\n", sockfd);
35         return -1;
36     }
37
38     while(true) {
39         connfd = acceptConnectionFD(sockfd);
40
41         fgets(request, sizeof(request), connfd);
42         strcpy(method,strtok(request, " "));
43         strcpy(url,strtok(NULL, " "));
44         while(request[0]!='\r') {
45             fgets(request, sizeof(request), connfd);
46             if(strstr(request, "Content-Length:")!=NULL) {
47                 length = atol(request+15);
48             }
49         }
50
51         if(strcmp(method, "POST")==0) {
52             for(i=0; i<length; i++) {
53                 c = fgetc(connfd);
54             }
55         }
56
57         if(strstr(url, "sommatrice")==NULL) {
58             printf("Pagina statica\n");
59             fputs(html, connfd);
60         }
61         else {
62             printf("Pagina dinamica\n");
63             // passo al programma CGI sia url sia stream su cui
64             scrivere
65             sommatrice(url, connfd);
66         }
67
68         fclose(connfd);
69
70         printf("\n\n[SERVER] sessione HTTP completata\n\n");
71     }
72
73     closeConnection(sockfd);
74     return 0;
75 }
```

Le uniche differenze degne di nota sono:

- (53-61) Viene controllato l'URL richiesto dall'utente. Se si tratta della pagina sommatrice, viene invocata la funzione (60) , altrimenti viene restituita la pagina statica. La funzione `strstr` consente di verificare se è presente quella sequenza di caratteri all'interno della stringa. Nel caso in cui sia presente, ritorna il puntatore al primo carattere della sequenza, altrimenti `NULL`.
- (3-18) La funzione sommatrice che prende come argomenti l'URL e la pipe di output. Le operazioni che esegue sono quelli di prendere i due operandi ed eseguire una semplice operazione di somma. La funzione si conclude con l'invio di una pagina statica contenente il risultato.

2.11 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**.

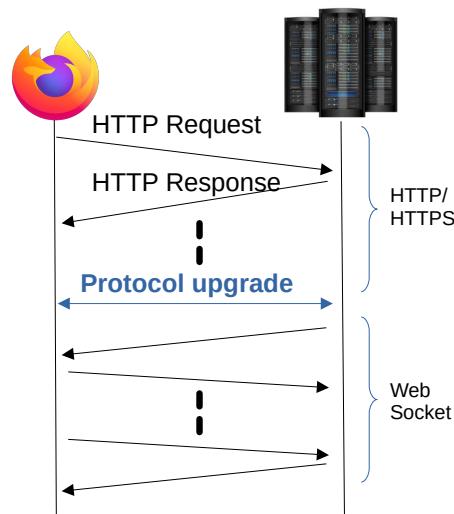


Figura 14: Esempio di web socket e Protocol Upgrade.

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 15: Il Protocol Upgrade nel dettaglio.

2.11.1 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 **maggiori interazioni 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**.
-

2.11.2 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*.

2.12 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).

2.12.1 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**.

2.12.2 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.

2.12.3 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`.
-

2.12.4 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 2.12.5). Ovviamamente si intende il codice a riga 21, ovvero quello necessario per la chat.

2.12.5 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.

2.12.6 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.

2.12.7 Esercizi

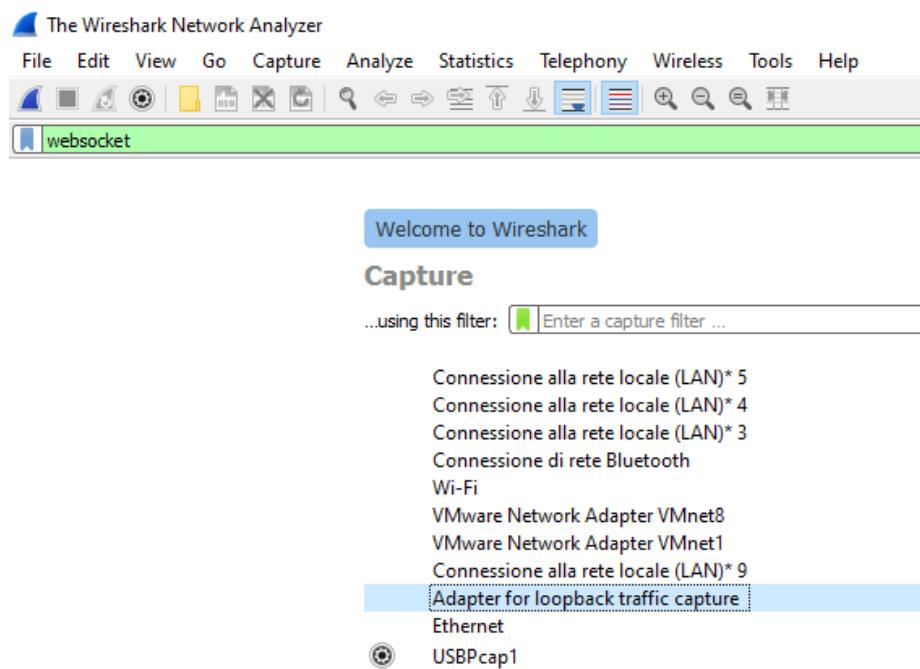
Esercizio 1

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     webSocket.broadcast.emit('typing', data);
39   });
40 });

Il codice è rimasto lo stesso (paragrafo 2.12.3), l'unica modifica effettuata è stata dalla riga 37 alla riga 39 in cui si impone al server di inoltrare il messaggio ricevuto, con tag typing, a tutti gli altri client eccetto il client mittente.
```

```

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     var current_date = new Date();
54     feedback.innerHTML = '<p>' + 'Time: ' + current_date.getHours() +
55     ':'
56     + current_date.getMinutes() + ':'
57     + current_date.getSeconds()
58     + ' - ' + '<strong>' + data.sender
      + ': </strong>' + 'typing...' + '</p>';
59 });

```

Il codice del front-end è rimasto pressoché identico (paragrafo 2.12.5) 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`).
 - (50-58) Alla ricezione dell'evento typing da parte del server, il client stamperà la scritta typing... .
-

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. Ovviamamente 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.

2.13 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**.

2.13.1 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 API²**, 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.

²Application 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.

2.13.2 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.

2.13.3 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 ]}
```

2.14 Esercizi su webservice basati su REST

2.14.1 Esercizio 1 - Analizzare server e clientREST-GET

Considerare la cartella WebService/. Aprire il file serverHTTP-REST.c e analizzarne il contenuto. Compilarlo come al solito ed eseguirlo. Aprire il file clientREST-GET.c e analizzarne il contenuto. Compilarlo come al solito ed eseguirlo:

- Che parametri devo passare in linea di comando?
- Cosa si può vedere analizzando lo scambio di dati tramite Wireshark?
- Quale è la *signature* della funzione calcolaSomma() sul server e sul client?
Perché ha senso che siano uguali?

Soluzione

Il codice del server non viene commentato poiché quasi identico all'esercizio:

```
1 #include "network.h"
2
3 float calcolaSomma(float val1, float val2) {
4     return (val1 + val2);
5 }
6
7 int main(){
8     socketif_t sockfd;
9     FILE* connfd;
10    int res, i;
11    long length=0;
12    char request[MTU], url[MTU], method[10], c;
13
14    sockfd = createTCPServer(8000);
15    if (sockfd < 0){
16        printf("[SERVER] Errore: %i\n", sockfd);
17        return -1;
18    }
19
20    while(true) {
21        connfd = acceptConnectionFD(sockfd);
22
23        fgets(request, sizeof(request), connfd);
24        strcpy(method,strtok(request, " "));
25        strcpy(url,strtok(NULL, " "));
26        while(request[0]!='\r') {
27            fgets(request, sizeof(request), connfd);
28            if(strstr(request, "Content-Length:")!=NULL) {
29                length = atol(request+15);
30            }
31        }
32
33        if(strcmp(method, "POST")==0) {
34            for(i=0; i<length; i++) {
35                c = fgetc(connfd);
36            }
37        }
38
39        if(strstr(url, "calcola-somma")==NULL) {
40            fprintf(connfd,"HTTP/1.1 200 OK\r\n\r\n");
41            fprintf(connfd,"Funzione non riconosciuta!\r\n");
42        }
43    }
44}
```

```

41
42     }
43     else {
44         printf("Chiamata a funzione sommatrice\n");
45
46         char *function, *op1, *op2;
47         float somma, val1, val2;
48
49         // skeleton: decodifica (de-serializzazione) dei
50         // parametri
51         function = strtok(url, "?&");
52         op1 = strtok(NULL, "?&");
53         op2 = strtok(NULL, "?&");
54         strtok(op1, "=");
55         val1 = atof(strtok(NULL, "="));
56         strtok(op2, "=");
57         val2 = atof(strtok(NULL, "="));
58
59         // chiamata alla business logic
60         somma = calcolaSomma(val1, val2);
61
62         // skeleton: codifica (serializzazione) del risultato
63         fprintf(connfd, "HTTP/1.1 200 OK\r\n\r\n{\r\n    \"somma"
64         "\":%f\r\n}\r\n", somma);
65     }
66
67     fclose(connfd);
68
69     closeConnection(sockfd);
70     return 0;
71 }
```

Il codice del client è il seguente:

```
1 #include "network.h"
2
3 float calcolaSomma(float val1, float val2) {
4     char request[MTU], response[MTU];
5
6     // stub: codifica (serializzazione) dei parametri
7     sprintf(request, "http://localhost:8000/calcola-somma?param1=%f
8 &param2=%f", val1, val2);
9
10    // chiamata del webservice
11    int res = doGET(request, response, MTU);
12    if (res < 0) {
13        printf("Errore: %i\n", res);
14        return -1;
15    }
16
17    printf("Risposta dal server:\n%s\n", response);
18
19    // stub: de-codifica (de-serializzazione) del risultato
20    return atof(strstr(response, ":")+1);
21
22 int main(int argc, char **argv){
23
24     if(argc < 4) {
25         printf("USAGE: %s tipofunzione op1 op2\n", argv[0]);
26         return -1;
27     }
28     else if(strcmp(argv[1], "calcola-somma")==0) {
29         printf("Risultato: %f\n", calcolaSomma(atof(argv[2]), atof(
30             argv[3])));
31     }
32
33     return 0;
34 }
```

Come si può vedere dal codice (24), il programma accetta solo un tipo di operazione, ovvero la somma, e due operandi numerici.

Analizzando i pacchetti con Wireshark, è possibile vedere una richiesta GET e i parametri inseriti tramite linea di comando.

Con *signature* si intende la dichiarazione delle funzioni. In questo caso, il client esegue una richiesta inserendo i parametri ottenuti dalla linea di comando, all'interno di un URL. Successivamente, esegue una REST di tipo GET indirizzata al server, ottenuto grazie all'URL. Il server acquisisce i parametri dall'URL creato dal client e li utilizza per fare la somma nella sua funzione `calcolaSomma`.

È ovvio che la *signature* della funzione `calcoloSomma()` debba essere uguale. In caso contrario, potrebbe manifestarsi un risultato inesatto o un comportamento inaspettato.

2.14.2 Esercizio 2 - ClientREST in Java

Prendere in considerazione il file ClientREST.java. Dopo aver installato l'ambiente base di Java, si può compilare con javac ClientREST.java ed eseguire con java ClientREST. Il fatto che il server sia fatto in C e il client in Java è un problema? Perché?

Soluzione

Il client scritto in java è il seguente:

```
1 import java.io.*;
2 import java.net.*;
3
4 class ClientREST
5 {
6     public static void main(String args[])
7     {
8         RESTAPI service1=new RESTAPI("127.0.0.1");
9
10        if(args.length < 3)    {
11            System.out.println("USAGE: java ClientREST tipofunzione
12                op1 op2");
13        }
14        else if(args[0].equals("calcola-somma")) {
15            System.out.println("Risultato: " + service1.
16            calcolaSomma(Float.parseFloat(args[1]), Float.parseFloat(args
17                [2])));
18        }
19    }
20
21    class RESTAPI
22    {
23        String server;
24
25        RESTAPI(String remoteServer) {
26            server = new String(remoteServer);
27        }
28
29        float calcolaSomma(float val1, float val2) {
30
31            URL u = null;
32            float risultato=0;
33            int i;
34
35            try
36            {
37                u = new URL("http://" +server+ ":8000/calcola-somma?
38                param1="+val1+"&param2="+val2);
39                System.out.println("URL aperto: " + u);
40            }
41            catch (MalformedURLException e)
42            {
43                System.out.println("URL errato: " + u);
44            }
45
46            try
47            {
48                URLConnection c = u.openConnection();
49                c.connect();
```

```

47     BufferedReader b = new BufferedReader(new
48     InputStreamReader(c.getInputStream()));
49     System.out.println("Lettura dei dati...");
50     String s;
51     while( (s = b.readLine()) != null ) {
52         System.out.println(s);
53         if((i = s.indexOf("somma"))!=-1)
54             risultato = Float.parseFloat(s.substring(i+7));
55     }
56     catch (IOException e)
57     {
58         System.out.println(e.getMessage());
59     }
60
61     return (float)risultato;
62 }
63
64 }
```

Grazie alle caratteristiche del protocollo REST (paragrafo 2.13.2), comunicare con il server non è un problema. Infatti, questa tecnologia si basa sul protocollo HTTP e non su quali linguaggi di programmazione sono scritti il client e il server. Di conseguenza, anche se il server fosse scritto in Python, non ci sarebbero problemi. L'unica cosa da tenere in considerazione è la *signature* della funzione di calcolo della somma che deve rispettare i parametri.

2.14.3 Esercizio 3 - Modifica server per calcolare anche i numeri primi

Estendere il webservice `serverHTTP-REST.c` in modo che esponga un secondo servizio relativo al calcolo dei numeri primi compresi nell'intervallo $[min, max]$:

- Si traggia spunto dal programma `prime-number-interval.c`

Successivamente:

- Estendere il file `ClientREST.java` in modo da poter chiamare, a scelta, entrambe le funzionalità della nuova API.
- Provare il client con il calcolo dei numero primi compresi nell'intervallo $[1, 1000000]$: quanto tempi ci mette? Ho dovuto tradurre l'algoritmo dei numeri primi in Java? Perché?

Soluzione

Il codice del server viene modificato introducendo un nuovo metodo: `calcoloPrimi`. Tale funzione consente di calcolare i numeri primi e salvarli all'interno di una stringa già formattati pronti per essere inviati al client:

```
1 #include "network.h"
2
3 float calcolaSomma(float val1, float val2) {
4     return (val1 + val2);
5 }
6
7 char * calcolaPrimi(int val1, int val2) {
8     bool flag;
9
10    // lower bound
11    int a = val1;
12
13    // upper bound
14    int b = val2;
15
16    char *result = malloc(val2);
17
18    // Traverse each number in the interval
19    // with the help of for loop
20    for (int i = a; i <= b; i++) {
21        // Skip 0 and 1 as they are
22        // neither prime nor composite
23        if (i == 1 || i == 0)
24            continue;
25
26        // flag variable to tell
27        // if i is prime or not
28        flag = 1;
29
30        for (int j = 2; j <= i / 2; ++j) {
31            if (i % j == 0) {
32                flag = 0;
33                break;
34            }
35        }
36
37        // flag = 1 means i is prime
```

```

38         // and flag = 0 means i is not prime
39     if (flag == 1)
40     {
41         char *str = malloc(1000);
42         sprintf(str, "%d", i);
43         strcat(result, str);
44         strcat(result, "\n");
45         free(str);
46     }
47 }
48 return result;
49 }

50 int main(){
51     socketif_t sockfd;
52     FILE* connfd;
53     int res, i;
54     long length=0;
55     char request[MTU], url[MTU], method[10], c;

56     sockfd = createTCPServer(8000);
57     if (sockfd < 0){
58         printf("[SERVER] Errore: %i\n", sockfd);
59         return -1;
60     }

61     while(true) {
62         connfd = acceptConnectionFD(sockfd);

63         fgets(request, sizeof(request), connfd);
64         strcpy(method,strtok(request, " "));
65         strcpy(url,strtok(NULL, " "));
66         while(request[0]!='\r') {
67             fgets(request, sizeof(request), connfd);
68             if strstr(request, "Content-Length:")!=NULL) {
69                 length = atol(request+15);
70             }
71         }

72         if(strcmp(method, "POST")==0) {
73             for(i=0; i<length; i++) {
74                 c = fgetc(connfd);
75             }
76         }

77         if(strstr(url, "calcola-somma")==NULL)
78             if(strstr(url, "calcola-num-primi")==NULL)
79                 fprintf(connfd,"HTTP/1.1 200 OK\r\n\r\n");
80             else
81             {
82                 printf("Chiamata a funzione numero primo\n");
83             }
84         }

85         if(strstr(url, "calcola-somma")!=NULL)
86             if(strstr(url, "calcola-num-primi")!=NULL)
87                 fprintf(connfd,"HTTP/1.1 200 OK\r\n\r\n");
88             else
89             {
90                 printf("Chiamata a funzione numero primo\n");
91             }
92         }

93         // skeleton: decodifica (de-serializzazione) dei
94         parametri
95         function = strtok(url, "?&");
96         op1 = strtok(NULL, "?&");
97         op2 = strtok(NULL, "?&");


```

```

98         strtok(op1, "=");
99         val1 = atof(strtok(NULL, "="));
100        strtok(op2, "=");
101        val2 = atof(strtok(NULL, "="));
102
103        char *primi;
104
105        // chiamata alla business logic
106        primi = calcolaPrimi(val1, val2);
107
108
109        // skeleton: codifica (serializzazione) del
110        risultato
111        fprintf(connfd, "HTTP/1.1 200 OK\r\n\r\n\f\r\n      \
112        numeri primi\"%s\r\n}\r\n", primi);
113
114        free(primi);
115    }
116    else
117    {
118        printf("Chiamata a funzione sommatrice\n");
119
120        char *function, *op1, *op2;
121        float somma, val1, val2;
122
123        // skeleton: decodifica (de-serializzazione) dei
124        parametri
125        function = strtok(url, "?&");
126        op1 = strtok(NULL, "?&");
127        op2 = strtok(NULL, "?&");
128        strtok(op1, "=");
129        val1 = atof(strtok(NULL, "="));
130        strtok(op2, "=");
131        val2 = atof(strtok(NULL, "="));
132
133        // chiamata alla business logic
134        somma = calcolaSomma(val1, val2);
135
136        // skeleton: codifica (serializzazione) del risultato
137        fprintf(connfd, "HTTP/1.1 200 OK\r\n\r\n\f\r\n      \
138        \"%f\r\n}\r\n", somma);
139
140        fclose(connfd);
141
142        printf("\n\n[SERVER] sessione HTTP completata\n\n");
143    }
144
145    closeConnection(sockfd);
146    return 0;
147 }
```

Il codice del client viene modificato introducendo sempre un nuovo metodo, ma l'unica differenza è il salvataggio dei numeri nell'ArrayList locale (riga 95):

```
1 import java.io.*;
2 import java.net.*;
3 import java.util.ArrayList;
4 import java.util.List;
5
6 class ClientREST
7 {
8     public static void main(String args[])
9     {
10         RESTAPI service1=new RESTAPI("127.0.0.1");
11
12         if(args.length < 3)      {
13             System.out.println("USAGE: java ClientREST tipofunzione
14 op1 op2");
15         }
16         else
17             if(args[0].equals("calcola-somma"))
18                 System.out.println("Risultato: " + service1.
19 calcolaSomma(Float.parseFloat(args[1]), Float.parseFloat(args
20 [2])));
21             else
22                 if(args[0].equals("calcola-num-primi"))
23                     System.out.println("Risultato: " + service1.
24 calcolaPrimi(Integer.parseInt(args[1]), Integer.parseInt(args
25 [2])));
26     }
27
28     class RESTAPI
29     {
30         String server;
31
32         RESTAPI(String remoteServer) {
33             server = new String(remoteServer);
34         }
35
36         float calcolaSomma(float val1, float val2) {
37             URL u = null;
38             float risultato=0;
39             int i;
40
41             try
42             {
43                 u = new URL("http://"+server+":8000/calcola-somma?
44 param1="+val1+"&param2="+val2);
45                 System.out.println("URL aperto: " + u);
46             }
47             catch (MalformedURLException e)
48             {
49                 System.out.println("URL errato: " + u);
50             }
51
52             try
53             {
54                 URLConnection c = u.openConnection();
55                 c.connect();
56                 BufferedReader b = new BufferedReader(new
57 InputStreamReader(c.getInputStream()));
58                 System.out.println("Lettura dei dati... ");
59                 String s;
```

```

54     while( (s = b.readLine()) != null ) {
55         System.out.println(s);
56         if((i = s.indexOf("somma"))!=-1)
57             risultato = Float.parseFloat(s.substring(i+7));
58     }
59 }
60 catch (IOException e)
61 {
62     System.out.println(e.getMessage());
63 }
64
65     return (float)risultato;
66 }
67
68 List<Integer> calcolaPrimi(int val1, int val2){
69     URL u = null;
70     List<Integer> risultato = new ArrayList<>();
71     int i;
72
73     try
74     {
75         u = new URL("http://"+server+":8000/calcola-num-primi?
param1="+val1+"&param2="+val2);
76         System.out.println("URL aperto: " + u);
77     }
78     catch (MalformedURLException e)
79     {
80         System.out.println("URL errato: " + u);
81     }
82
83     try
84     {
85         URLConnection c = u.openConnection();
86         c.connect();
87         BufferedReader b = new BufferedReader(new
InputStreamReader(c.getInputStream()));
88         System.out.println("Lettura dei dati...");
89         String s;
90         while( (s = b.readLine()) != null ) {
91             System.out.println(s);
92             if((i = s.indexOf("primi"))!=-1)
93                 risultato.add(Integer.parseInt(s.substring(i+7))
));
94             try {
95                 risultato.add(Integer.parseInt(s));
96             }
97             catch (Exception e){}
98         }
99     }
100    catch (IOException e)
101    {
102        System.out.println(e.getMessage());
103    }
104
105    return risultato;
106 }
107 }
```

2.14.4 Esercizio 4 - ClientThreadREST con thread concorrenti

Prendere in considerazione il file clientThreadREST.java:

- Esso invoca calcolaSomma() in 3 thread concorrenti
- Però la macchina su cui gira il server chiamato è la stessa e quindi non ci guadagno in prestazioni
- Come si dovrebbe modificare il codice in modo che le 3 invocazioni finiscano su 3 macchine diverse?
- Bisogna modificare anche il codice del server?

Si consideri il servizio che calcola i numeri primi nell'intervallo $[min, max]$ costruito nell'esercizio precedente:

- Si trovi un modo efficiente, sfruttando diversi server in rete, per calcolare i numeri primi tra 1 e 1000000
- Di quanto migliorano le prestazioni?
- Devo modificare anche il codice del server?

Soluzione

Il codice clientThreadREST.java è il seguente:

```
1 import java.io.*;
2 import java.net.*;
3
4 class ClientThreadREST
5 {
6     public static void main(String args[])
7     {
8         if(args.length < 3)    {
9             System.out.println("USAGE: java ClientREST tipofunzione
10                op1 op2");
11         }
12         else    {
13             RESTAPI service1=new RESTAPI("127.0.0.1", args[0], args
14 [1], args[2]);
15             RESTAPI service2=new RESTAPI("127.0.0.1", args[0], args
16 [1], args[2]);
17             RESTAPI service3=new RESTAPI("127.0.0.1", args[0], args
18 [1], args[2]);
19             service1.start();
20             service2.start();
21             service3.start();
22         }
23     }
24     class RESTAPI extends Thread
25     {
26         String server, service, param1, param2;
27
28         public void run()    {
29             if(service.equals("calcola-somma"))    {
30                 System.out.println("Risultato: " + calcolaSomma(Float.
31 parseFloat(param1), Float.parseFloat(param2)));
32             }
33         }
34     }
35 }
```

```

29     }
30     else    {
31         System.out.println("Servizio non disponibile!");
32     }
33 }
34 }
35
36 RESTAPI(String remoteServer, String srvc, String p1, String p2)
37 {
38     server = new String(remoteServer);
39     service = new String(srvc);
40     param1 = new String(p1);
41     param2 = new String(p2);
42 }
43
44 synchronized float calcolaSomma(float val1, float val2) {
45
46     URL u = null;
47     float risultato=0;
48     int i;
49
50     try
51     {
52         u = new URL("http://"+server+":8000/calcola-somma?
53 param1="+val1+"&param2="+val2);
54         System.out.println("URL aperto: " + u);
55     }
56     catch (MalformedURLException e)
57     {
58         System.out.println("URL errato: " + u);
59     }
60
61     try
62     {
63         URLConnection c = u.openConnection();
64         c.connect();
65         BufferedReader b = new BufferedReader(new
66 InputStreamReader(c.getInputStream()));
67         System.out.println("Lettura dei dati...");
68         String s;
69         while( (s = b.readLine()) != null ) {
70             System.out.println(s);
71             if((i = s.indexOf("somma"))!=-1)
72                 risultato = Float.parseFloat(s.substring(i+7));
73         }
74     }
75     catch (IOException e)
76     {
77         System.out.println(e.getMessage());
78     }
79 }
80
81     return (float)risultato;
82 }
83 }
```

Per eseguire il codice su tre macchine differenti, sarebbe necessario modificare l'URL di destinazione alle righe 12, 13 e 14. Stando attenti e cambiando anche a riga 51 l'URL così da collegarsi all'URL esatto. Si potrebbe provare a cambiare solamente alle righe 12, 13 e 14 gli URL, ed eseguire il codice del serverHTTP-REST su altre tre macchine differenti. In questo modo si sfrutterebbero la piena potenza delle tre macchine.

3 Introduzione alla sicurezza

Per comprendere al meglio la sicurezza informatica, è necessario procedere a piccoli passi. Prima di tutto, ci sono delle domande da porsi:

- Quali risorse si vogliono proteggere? Risposta al paragrafo 3.1
 - Come vengono minacciate tali risorse? Risposta al paragrafo 3.2
 - Cosa è necessario fare per contrastare tali minacce? Risposta al paragrafo 3.3
-

3.1 Quali risorse si vogliono proteggere

Le risorse che solitamente si vogliono proteggere sono:

- Risorse **hardware**, come i sistemi, i componenti, dischi. In questo caso si parla di **sicurezza “fisica”**;
- Risorse **software**, come i sistemi operativi e gli applicativi;
- Risorse di **dati**, come file e database;
- Risorse di **rete**, come i collegamenti e gli apparati.

Per **proteggere** queste risorse è necessario **garantire le proprietà di**:

- Confidenzialità
 - Integrità
 - Disponibilità
 - Autenticità
 - Tracciabilità
-

3.1.1 Confidenzialità

Per **confidenzialità** si intende che **nessun utente deve poter ottenere o dedurre dal sistema informazioni che non è autorizzato a conoscere**. In questo caso ci sono due caratteristiche fondamentali:

- La **riservatezza dei dati**, ovvero le informazioni confidenziali non devono essere rilevate o rilevabili da utenti non autorizzati;
- La **privacy**, ovvero dare l'opportunità all'utente di controllare i dati che il sistema con cui sta interagendo può collezionare e memorizzare.

3.1.2 Integrità

Per **integrità** si intende di impedire l'alterazione diretta o indiretta delle informazioni, sia da parte di utente e processi non autorizzati, che a seguito di eventi accidentali. Nel caso in cui i dati venissero alterati, sarebbe necessario fornire strumenti per poter verificare facilmente tale alterazione.

Anche qui ci sono due caratteristiche fondamentali:

- **Integrità dei dati**, ovvero le informazioni e i programmi possono essere modificati solo se autorizzati;
 - **Integrità del sistema**, ovvero il sistema funziona e non è compromesso.
-

3.1.3 Disponibilità

Per **disponibilità** si intende la possibilità di rendere disponibili a ciascun utente abilitato, le informazioni alle quali ha diritto di accedere, nei tempi e modi previsti. Quindi in determinate condizioni o in un preciso istante o in un intervallo di tempo. Nei sistemi informatici, i requisiti di disponibilità includono caratteristiche di prestazioni e robustezza.

3.1.4 Autenticità

Per **autenticità** si intende il dovere da parte di ciascun utente di verificare l'autenticità delle informazioni. Per esempio messaggi, mittenti e destinatari. Per garantire l'autenticità, si richiede di poter verificare se un'informazione è stata manipolata.

3.1.5 Tracciabilità

Per **tracciabilità** si intende che le azioni di un'entità devono essere tracciate in modo univoco così da supportare la non-ripudiabilità e l'isolamento della responsabilità. Ad esempio, nessun utente deve poter ripudiare o negare in tempi successivi messaggi da lui spediti o firmati.

3.2 Come vengono minacciate le risorse

Le minacce compromettono le proprietà di confidenzialità, integrità e disponibilità. Per **esempio**:

	Confidenzialità	Integrità	Disponibilità
HW			Calcolatore rubato
SW	Copia non autorizzata	Eseguibile modificato	Eseguibili cancellati
Dati	Lettura non autorizzata	File modificati	File cancellati
Rete	Lettura messaggi inviati	Messaggi modificati/ritardati/duplicati	Messaggi distrutti, rete fuori uso

Una **minaccia** è una **possibile violazione della sicurezza**. Mentre un **attacco** è una **violazione effettiva della sicurezza**. Gli **attacchi** possono essere:

- **Attivi**, tentativi di **alterare risorse o modificare il funzionamento** dei sistemi;
- **Passivi**, tentativi di **ottenere informazioni e utilizzarle** senza intaccare le risorse;
- **Interni**, iniziati da un'**entità interna** al sistema;
- **Esterne**, iniziati da un'**entità esterna**, tipicamente attraverso la rete.

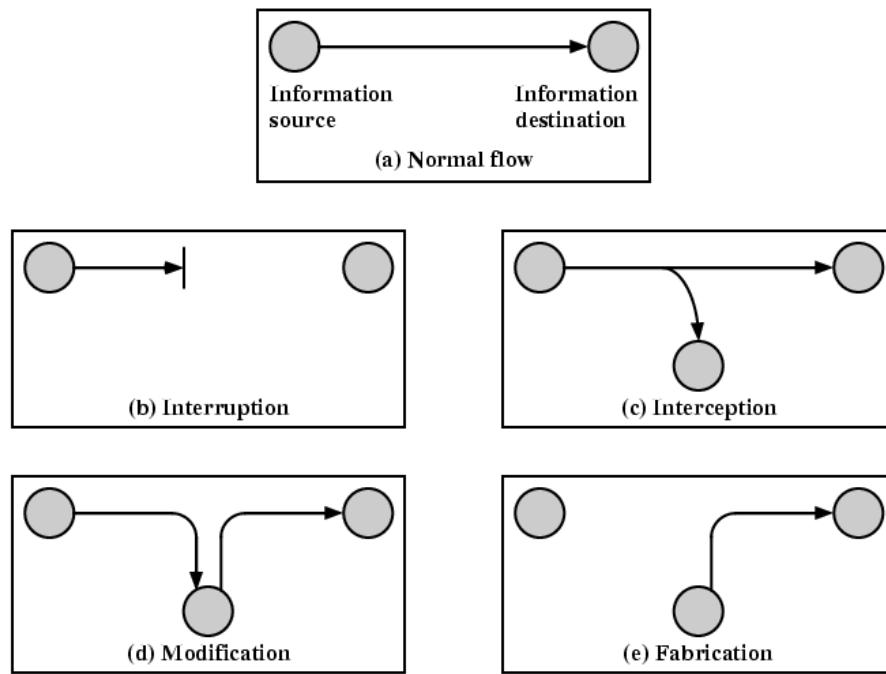


Figura 16: Esempi di attacchi.

Infine, esistono alcune classi di attacchi o minacce:

- ***Disclosure***, accesso non autorizzato alle informazioni
- ***Deception***, accettazione di dati falsi
- ***Disruption***, interruzione o prevenzione di operazioni corrette
- ***Usurpation*** controllo non autorizzato di alcune parti del sistema

3.3 Come contrastare le minacce

Il contrasto delle minacce è la parte più difficile della sicurezza. Innanzitutto non esiste una risposta unica poiché dipende da una serie di fattori. Inoltre, le risorse cambiano con il tempo e quello che oggi è considerato come “sistema sicuro”, un domani non potrebbe più esserlo.

Durante la progettazione di sistemi è necessario considerare la possibilità di **attacchi potenziali**. Quindi, l'obiettivo è di considerare quali possibili attacchi un attaccante potrebbe mettere in atto. Nello sviluppo di meccanismi di difesa, è intelligente utilizzare delle **soluzioni contro-intuitive** così da complicare la vita ad eventuali malintenzionati.

I **meccanismi di sicurezza** devono essere inseriti sia a livello fisico che a livello logico, quindi protocollore. Tuttavia, nonostante il grande impegno che può esserci per sviluppare sistemi sicuri, la **sicurezza dipende anche dagli utenti**. Per esempio le informazioni possedute, come password, e la creazione/distribuzione/protezione di tali informazioni dovrebbero avere un occhio di riguardo.

Purtroppo sempre più spesso la **sicurezza viene vista come un impedimento e/o rallentamento** del normale funzionamento dei sistemi per una serie di motivi:

- Gli amministratori conducono una battaglia no-stop contro gli **attaccanti**, ai quali, a differenza dei difensori che devono eliminare qualsiasi minaccia possibile, è sufficiente sfruttare una singola vulnerabilità;
- Dispensio di energie e forze senza un reale risultato tangibile. Questo si traduce in uno svantaggio e non in un beneficio. Ovviamente finché non accade un incidente di sicurezza...
- Sempre più elementi aggiuntivi al sistema.

3.3.1 Principi fondamentali di progettazione della sicurezza

Nonostante anni di ricerca, è impossibile progettare sistema senza falle di sicurezza. Tuttavia, grazie ad un insieme di pratiche e regole è possibile creare un sistema molto difficile da attaccare con successo. Queste **regole** vengono chiamate **principi fondamentali di progettazione della sicurezza**:

- **Aspetti economici dei meccanismi**, la progettazione delle misure di sicurezza deve essere il più semplice possibile, sia da implementare che da verificare;
- ***Fail-safe default***, i comportamenti non specificati devono prevedere un default sicuro, ad esempio i permessi di accesso;
- **Progettazione aperta** è preferibile rispetto al codice segreto;
- **Tracciabilità delle operazioni**, così che qualsiasi operazione può essere ricostruita e il sistema ripristinato;
- **Separazione dei privilegi** dalle risorse create da ciascun utente e da quelle critiche;
- **Separazione delle funzionalità**, ovvero la distinzione dei ruoli nei diversi punti del sistema fisico e logico;
- **Isolamento dei sottosistemi**, ovvero un sistema compromesso non dovrebbe compromettere gli altri;
- **Modularità**, quindi meccanismi di sicurezza indipendenti, sostituibili e riusabili.

3.3.2 Politiche di sicurezza e meccanismi

Una **politica di sicurezza** è un'indicazione di cosa è permesso e di cosa non è concesso. Le regole riguardano i dati, le operazioni possibili, gli utenti singoli e i profili.

Un **meccanismo di sicurezza** è un metodo (strumento/procedura) per garantire una politica di sicurezza.

Quindi, data l'unione di queste due definizioni, si dice: data una politica che distingue le azioni sicure da quelle non sicure, i meccanismi di sicurezza **prevengono, scoprono o recuperano** da un attacco.

- **Prevenzione** di un attacco, ovvero il meccanismo di sicurezza deve rendere impossibile l'attacco. Spesso l'adozione di questa politica interferisce con il sistema al punto da renderlo scomodo da usare. Per **esempio**, la richiesta di password come modo di autenticazione.
- **Scoperta** di un attacco, quindi il meccanismo di sicurezza è in grado di scoprire che un attacco è in corso. È **utile** quando:
 - Non è possibile prevenire l'attacco;
 - È necessario valutare le misure preventive.

Solitamente vengono monitorate le risorse del sistema, cercando eventuali tracce di attacchi.

- **Recupero** da un attacco. Questa politica è possibile applicarla in due modi diversi:
 1. **Fermare l'attacco** e recuperare o ricostruire la situazione prima dell'attacco, per esempio con un backup;
 2. **Continuare a far funzionare il sistema correttamente durante l'attacco**, in gergo viene chiamata *fault-tolerant*.

Degli **esempi** di meccanismi (specifici) di sicurezza sono: la **crittografia**, cioè la trasformazione dei dati in un formato non facilmente comprensibile da un essere umano, la **firma digitale**, usata per provare la sorgente, l'**autenticazione e controllo degli accessi**, come la gestione dei diritti degli utenti rispetto le risorse. I meccanismi generali possono essere il rilevamento degli eventi, la gestione degli Audit ([Wikipedia](#)) o le Recovery ([CyberSecurity360](#)).

3.3.3 Ottenere un sistema sicuro

Per ottenere un **sistema sicuro**, i primi passi sono uno sviluppo consapevole. È dunque necessario seguire le seguenti **fasi**:

1. **Specifico**, descrive il funzionamento del sistema desiderato e avere una visione d'insieme;
2. **Progetto**, tradurre le specifiche in componenti che le implementano ed elencare le loro caratteristiche;
3. **Implementazione**, creare il sistema vero e proprio che soddisfi le specifiche.

Durante queste fasi è fondamentale continuare a verificare la correttezza dell'implementazione.

Inoltre, è importante **considerare alcune scelte implementative** inevitabili durante la progettazione di un sistema:

- **Analizzare i costi-benefici** dell'implementazione della **sicurezza** e dei suoi meccanismi;
- **Analizzare i rischi** in caso di attacchi e gli eventuali danni/perdite che possono causare;
- **Aspetti legali** riguardanti la sicurezza e la privacy ed eventuali **aspetti morali**;
- **Analizzare i problemi organizzativi**, infatti l'implementazione di meccanismi di sicurezza, come detto nei paragrafi precedenti, potrebbe creare situazioni di difficoltà per gli utenti che utilizzano il sistema. Questo si potrebbe tradurre nell'aumento di perdite invece di utili;
- **Aspetti comportamentali** delle persone coinvolte.

4 Analisi di rete con Wireshark e da linea di comando

4.1 Introduzione agli analizzatori di rete

Esistono diversi **strumenti software che consentono di analizzare i pacchetti** che arrivano alla propria interfaccia di rete:

- **TCDUMP** tool da linea di comando per sistemi operativi Linux;
- **WinDump** tool da linea di comando per sistemi operativi Windows;
- **Wireshark** tool sia da linea di comando che da GUI per sistemi operativi Linux, Windows e MacOS.

Tutti gli **strumenti di rete si basano sulla libreria** del linguaggio programmazione C, chiamata **libpcap**. Le sue funzionalità sono tre:

1. Cercare e trovare interfacce di rete;
 2. Gestione avanzata di filtri di cattura;
 3. Gestione degli errori e statistiche di cattura.
-

4.1.1 Sniffing e la motivazione del sudo

Con il termine **sniffing** viene intesa l'abilità del software, in questo caso Wireshark, di **catturare i pacchetti in arrivo e in partenza** dalla propria interfaccia di rete. Esistono due tipi di *sniffing*:

- *Sniffing* all'interno di **reti non-switched**. Come accade, per esempio, nelle **reti WiFi**, tutte le schede di rete dei PC collegati al router ricevono tutti i pacchetti, sia i propri sia quelli destinati ad altri.

In questo caso, lo *sniffing* consente di **catturare sia i pacchetti destinati al dispositivo che sta eseguendo questa operazione, sia i pacchetti di cui non è il destinatario**. Per eseguire questa operazione è necessario avviare Wireshark in modalità amministratore (comando *sudo* su Linux). Questo perché a priori l'**interfaccia di rete scarta i pacchetti a cui non è interessato**, mentre con Wireshark viene attivata la **modalità promiscua** che **consente di bypassare il sistema operativo e interrogare direttamente la CPU per salvare tutti i pacchetti in entrata e in uscita**. La modalità promiscua viene disattivata dal sistema operativo per motivi di *performance*. Infatti, se fosse sempre attiva, il sistema operativo avrebbe un carico di lavoro talmente alto sulla CPU che calerebbero le prestazioni generali del sistema.

- *Sniffing* all'interno di **reti Ethernet switched**. In questo caso, l'apparato centrale della rete (*switch*), si preoccupa di **inoltrare su ciascuna porta solo il traffico destinato ai dispositivi collegati a quella porta**. Di conseguenza, ciascuna interfaccia di rete riceve solo i pacchetti destinati a lei (o multicast/broadcast).

In altre parole, la **modalità promiscua non consente la cattura di pacchetti di altre schede di rete**.

4.2 Interfaccia grafica di Wireshark

Come specificato nel paragrafo 4.1.1, Wireshark necessita di essere avviato con la modalità amministratore (o `sudo` su Linux).

Una volta avviato, la schermata iniziale sarà la seguente. Al centro sono presenti una serie di voci in cui vengono indicati i vari dispositivi di rete su cui è possibile ascoltare il traffico. Oltre all'interfaccia Wi-Fi (se presente sulla macchina), è possibile trovare interfacce del tipo bluetooth, ethernet (LAN), USB e interfacce di rete delle macchine virtuali.

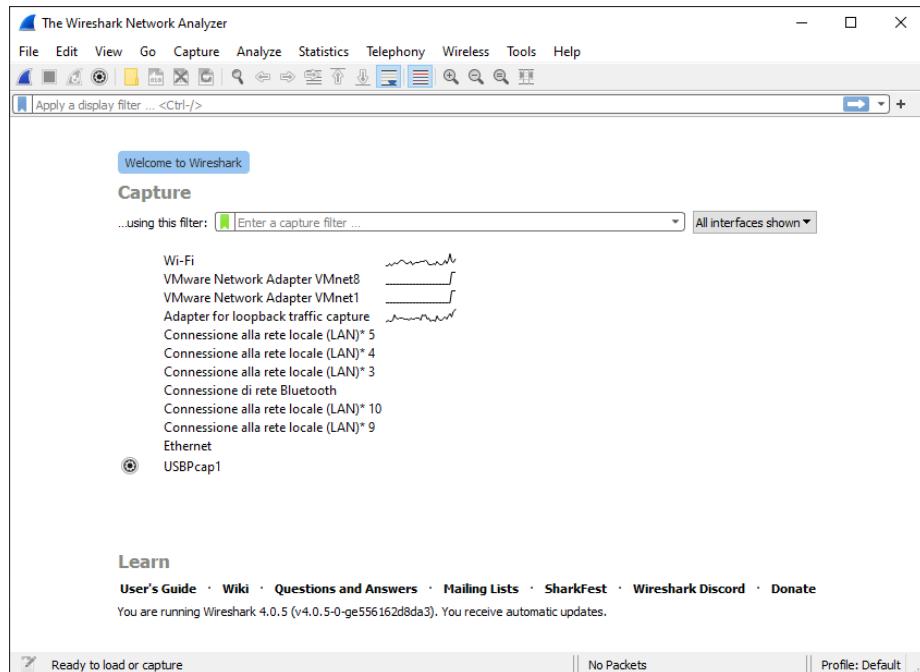


Figura 17: Schermata iniziale di Wireshark.

4.2.1 Sniffing della rete

Cliccando su una delle interfacce, il software inizierà ad ascoltare il traffico di pacchetti nella rete. La grafica si presenta nel seguente modo e con i seguenti colori:

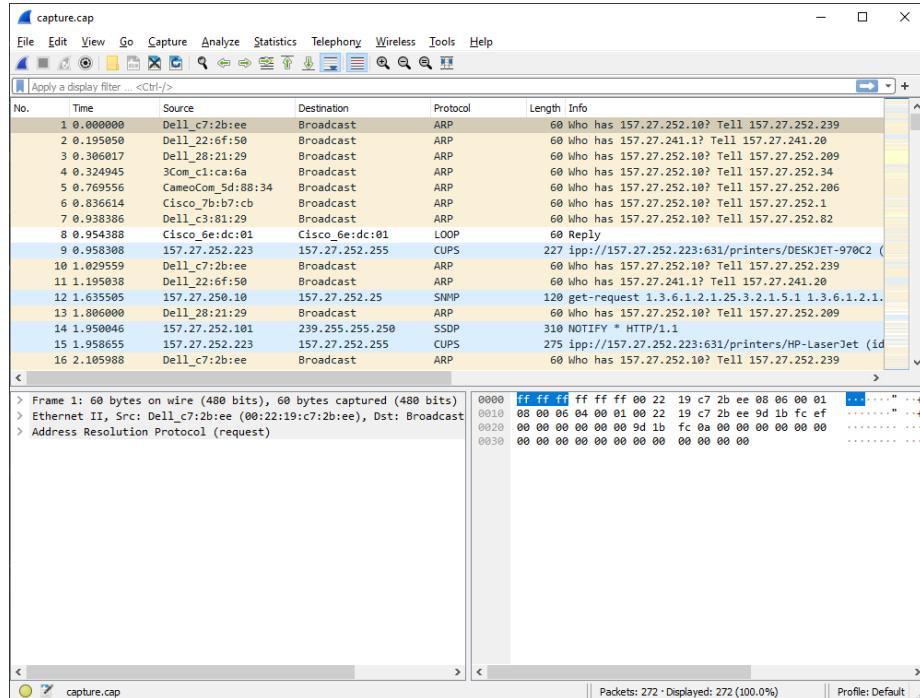


Figura 18: Schermata di *sniffing* nella rete.

4.2.2 Applicazione dei filtri

Data la grande mole di pacchetti che possono presentarsi durante lo *sniffing*, Wireshark dà la possibilità di applicare una serie di filtri (la serie di comandi che è possibile utilizzare: [documentazione ufficiale](#)). Essi possono essere applicati scrivendo il relativo comando nella barra in alto in cui è scritto “Apply a display filter ...”. Per **esempio** è possibile ricercare un protocollo scrivendolo nella barra (se quest’ultima viene evidenziata di verde, il comando è corretto, altrimenti viene evidenziata di rosso per segnalare un’inesattezza nel comando).

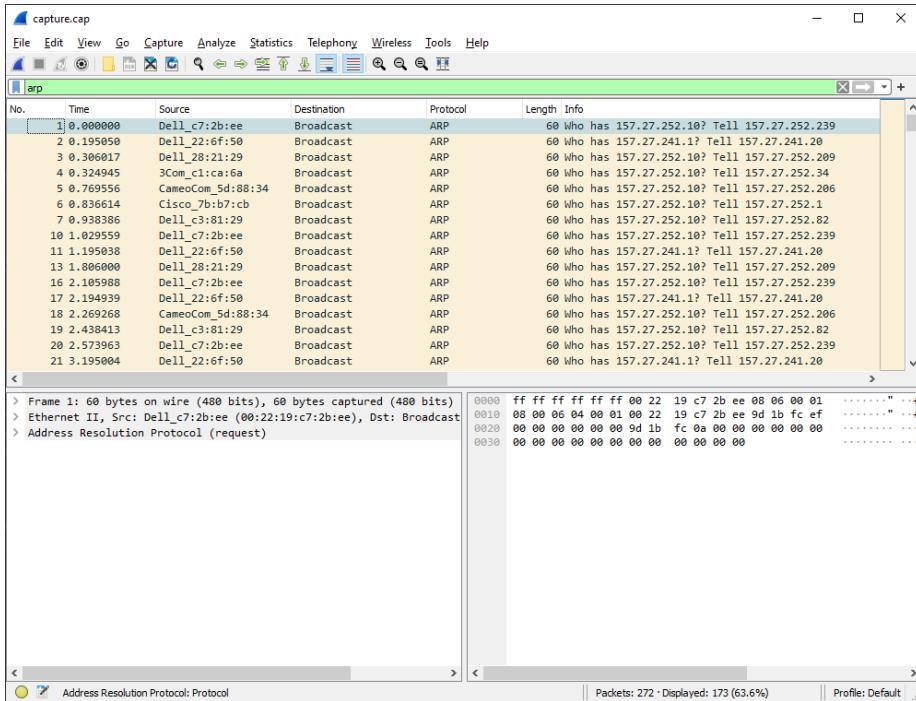


Figura 19: Applicazione corretta di un filtro.

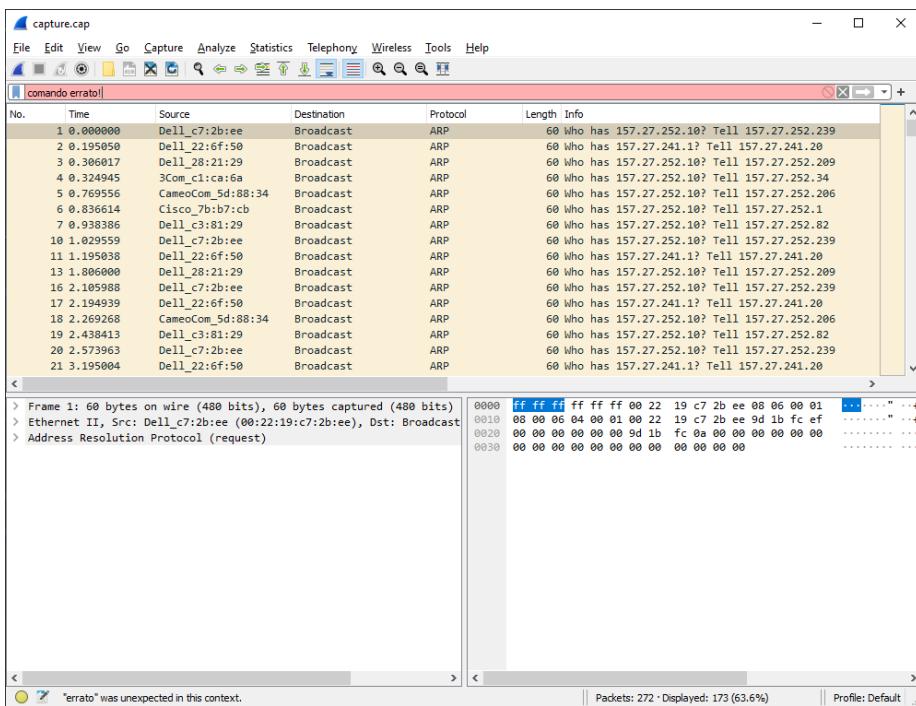


Figura 20: Applicazione errata di un filtro.

4.2.3 Seguire il flusso di una conversazione

È possibile seguire il flusso dati della “conversazione” di un pacchetto. Per farlo è necessario cliccare sul pacchetto interessato, andare nella barra degli strumenti e cliccare *Analyze* → *Follow* e selezionare il protocollo interessato. Per **esempio**, nella seguente conversazione viene seguito il protocollo UDP.

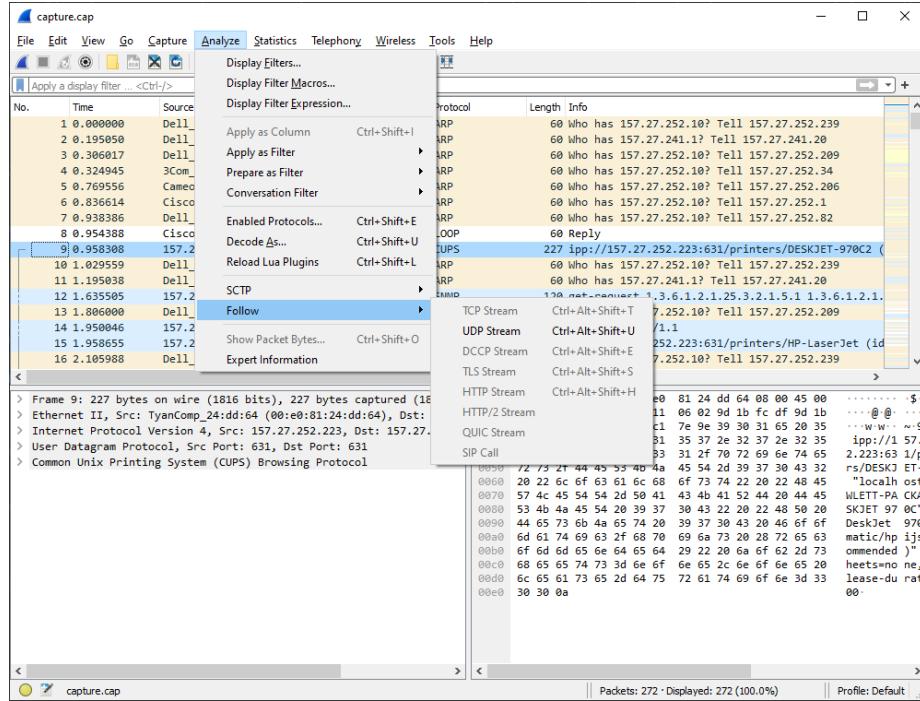


Figura 21: Analisi del flusso di un pacchetto 1 di 2.

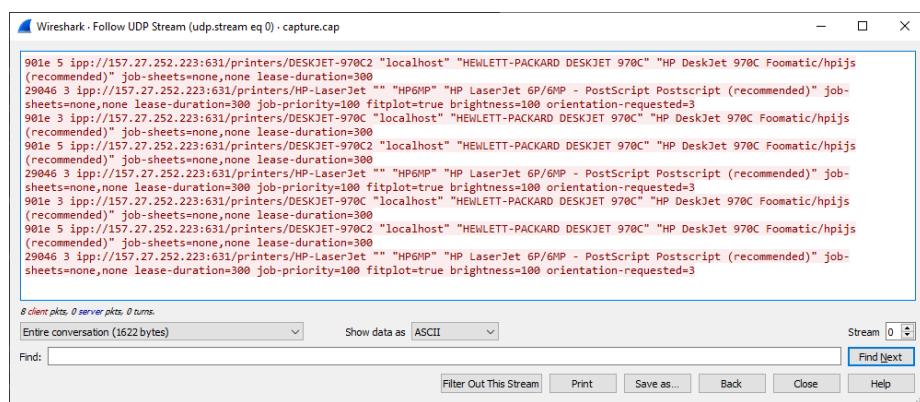
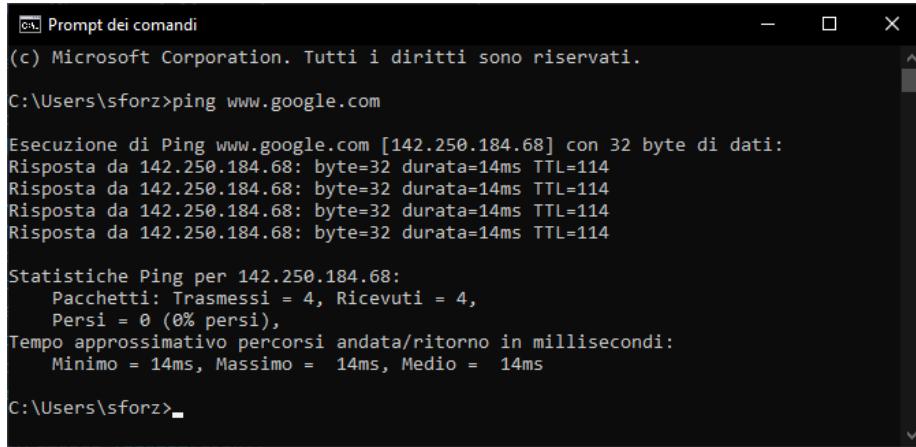


Figura 22: Analisi del flusso di un pacchetto 2 di 2.

4.3 Comando ping

Il comando **ping** consente di verificare la raggiungibilità di un computer connesso alla rete e il relativo Round Trip Time (RTT)³. Per questa operazione viene utilizzato il protocollo ICMP (Internet Control Message Protocol) che è un servizio per trasmettere informazioni riguardanti malfunzionamenti, informazioni di controllo o messaggi tra vari componenti di una rete di calcolatori.



```
C:\ Prompt dei comandi
(c) Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\sforz>ping www.google.com

Esecuzione di Ping www.google.com [142.250.184.68] con 32 byte di dati:
Risposta da 142.250.184.68: byte=32 durata=14ms TTL=114

Statistiche Ping per 142.250.184.68:
  Pacchetti: Trasmessi = 4, Ricevuti = 4,
  Persi = 0 (0% persi),
  Tempo approssimativo percorsi andata/ritorno in millisecondi:
    Minimo = 14ms, Massimo = 14ms, Medio = 14ms

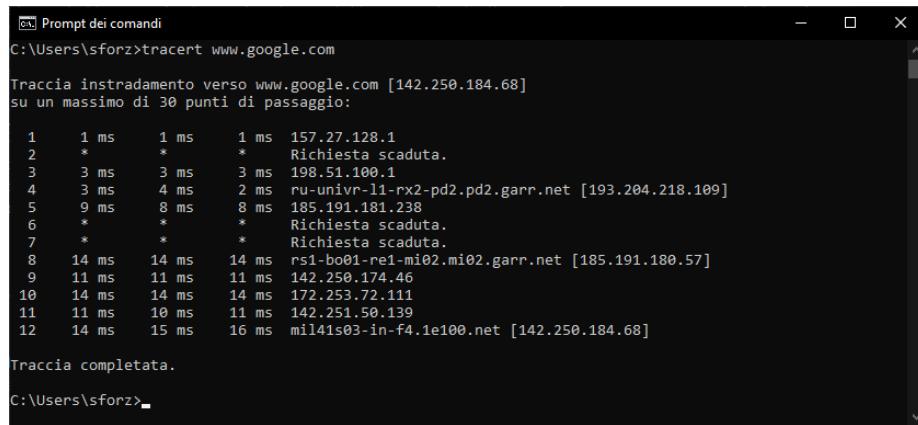
C:\Users\sforz>
```

Figura 23: Esempio di esecuzione del comando ping.

³Tempo che intercorre dalla partenza del pacchetto inviato fino al ritorno della risposta.

4.4 Comando traceroute (tracert Windows)

Il comando **traceroute** (tracert in Windows) è un semplice strumento per **tracciare il percorso che un pacchetto segue dalla sorgente alla destinazione**. Il comando mostra un elenco di tutte le interfacce dei router che il pacchetto attraversa finché non raggiunge la destinazione. Per questioni di sicurezza, alcuni nodi possono non essere visibili tramite il comando traceroute, questo per evitare che venga resa nota la struttura della rete.



```
C:\ Prompt dei comandi
C:\Users\sforz>tracert www.google.com

Traccia instradamento verso www.google.com [142.250.184.68]
su un massimo di 30 punti di passaggio:

 1  1 ms      * ms      1 ms  157.27.128.1
 2  *          * ms      *      Richiesta scaduta.
 3  3 ms      3 ms      3 ms  198.51.100.1
 4  3 ms      4 ms      2 ms  ru-univr-11-rx2-pd2.pd2.garr.net [193.204.218.109]
 5  9 ms      8 ms      8 ms  185.191.181.238
 6  *          * ms      *      Richiesta scaduta.
 7  *          * ms      *      Richiesta scaduta.
 8  14 ms     14 ms     14 ms  rs1-bo01-re1-mi02.mi02.garr.net [185.191.180.57]
 9  11 ms     11 ms     11 ms  142.250.174.46
10  14 ms     14 ms     14 ms  172.253.72.111
11  11 ms     10 ms     11 ms  142.251.50.139
12  14 ms     15 ms     16 ms  mil41s03-in-f4.1e100.net [142.250.184.68]

Traccia completata.

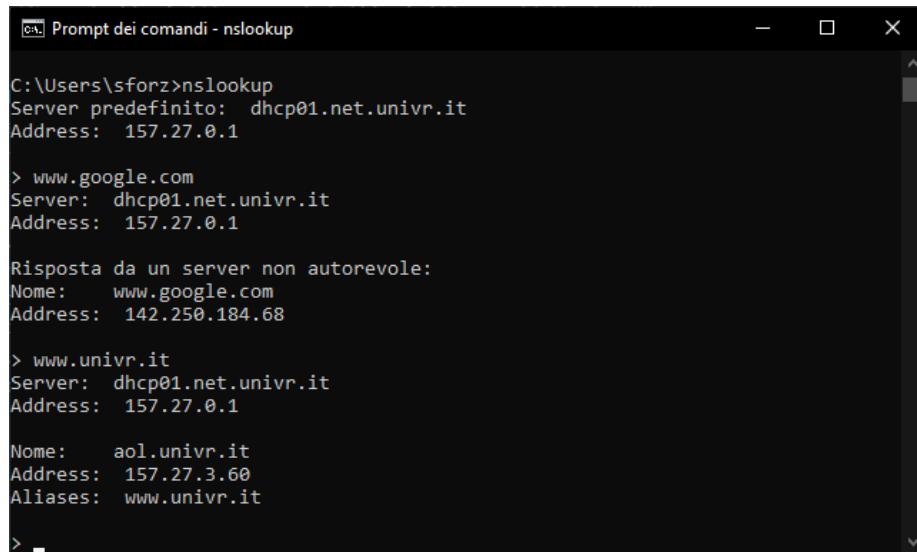
C:\Users\sforz>
```

Figura 24: Esempio di esecuzione del comando tracert su Windows (traceroute su Linux).

4.5 Comando nslookup

Il comando **nslookup** consente di effettuare un'interrogazione ai server DNS (Domain Name System)⁴ per poter ottenere da un hostname il relativo indirizzo IP, o viceversa. Esso può essere utilizzato in modalità: interattiva o non interattiva.

La modalità **interattiva** consente di effettuare più interrogazioni e visualizza i singoli risultati. Viene attivata eseguendo il comando senza parametri.



```
C:\Users\sforz>nslookup
Server predefinito: dhcp01.net.univr.it
Address: 157.27.0.1

> www.google.com
Server: dhcp01.net.univr.it
Address: 157.27.0.1

Risposta da un server non autorevole:
Nome: www.google.com
Address: 142.250.184.68

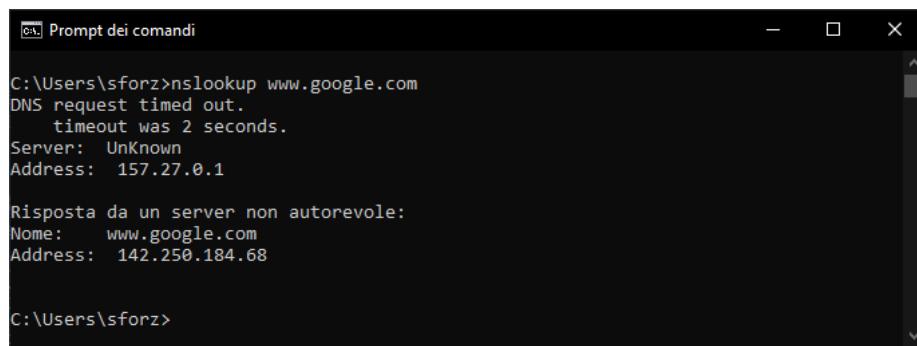
> www.univr.it
Server: dhcp01.net.univr.it
Address: 157.27.0.1

Nome: aol.univr.it
Address: 157.27.3.60
Aliases: www.univr.it

>
```

Figura 25: Esempio di esecuzione del comando nslookup in modalità interattiva.

La modalità **non interattiva** consente di effettuare una sola interrogazione visualizzandone il risultato. Viene attivata se viene inserito un parametro che corrisponde ad un *host-to-find*.



```
C:\Users\sforz>nslookup www.google.com
DNS request timed out.
    timeout was 2 seconds.
Server: UnKnown
Address: 157.27.0.1

Risposta da un server non autorevole:
Nome: www.google.com
Address: 142.250.184.68

C:\Users\sforz>
```

Figura 26: Esempio di esecuzione del comando nslookup in modalità non interattiva.

⁴Sistema di server organizzato gerarchicamente per la gestione del namespace.

4.6 Comando ifconfig (ipconfig Windows)

Il comando **ifconfig** (ipconfig in Windows) è utilizzato per **configurare e controllare un'interfaccia di rete TCP/IP** da riga di comando. La sua esecuzione con il parametro “-a” stampa su terminale le informazioni di tutte le interfacce di rete.

Vengono stampate molteplici interfacce di rete, ma tra queste le più famose sono:

- eth0 è la prima interfaccia Ethernet;
- lo è l'interfaccia *loopback*, sempre presente. È “speciale” poiché il sistema la utilizza per comunicare con sé stesso;
- wlan0 è il nome della prima interfaccia di rete Wireless del sistema.

```
tecmint@tecmint ~ $ ifconfig
eth0      Link encap:Ethernet HWaddr 28:d2:44:eb:bd:98
          inet addr:192.168.0.104 Bcast:192.168.0.255 Mask:255.255.255.0
              inet6 addr: fe80::2ad2:44ff:feeb:bd98/64 Scope:Link
                  UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
                  RX packets:342087 errors:0 dropped:0 overruns:0 frame:0
                  TX packets:233764 errors:0 dropped:0 overruns:0 carrier:0
                  collisions:0 txqueuelen:1000
                  RX bytes:406375041 (406.3 MB) TX bytes:25096967 (25.0 MB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
              inet6 addr: ::1/128 Scope:Host
                  UP LOOPBACK RUNNING MTU:65536 Metric:1
                  RX packets:5146 errors:0 dropped:0 overruns:0 frame:0
                  TX packets:5146 errors:0 dropped:0 overruns:0 carrier:0
                  collisions:0 txqueuelen:0
                  RX bytes:469809 (469.8 KB) TX bytes:469809 (469.8 KB)

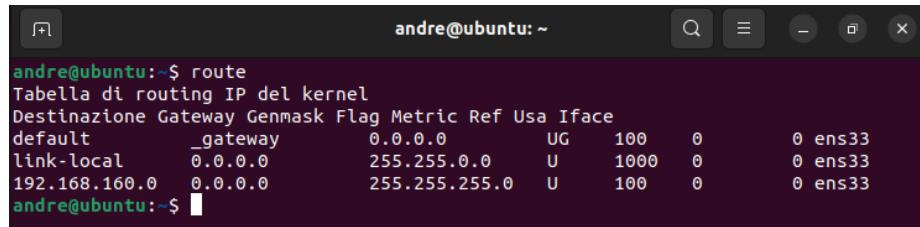
wlan0     Link encap:Ethernet HWaddr 38:b1:db:7c:78:c7
          UP BROADCAST MULTICAST MTU:1500 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

tecmint@tecmint ~ $
```

Figura 27: Esempio di esecuzione del comando ifconfig su Linux.

4.7 Comando route (route PRINT Windows)

Il comando **route** (route PRINT su Windows) è utilizzato per **visualizzare e modificare le tabelle di routing**. L'esecuzione consente di visualizzare la tabella di routing dell'host.



```
andre@ubuntu:~$ route
Tabella di routing IP del kernel
Destinazione Gateway Genmask Flag Metric Ref Usa Iface
default _gateway 0.0.0.0 UG 100 0 0 ens33
link-local 0.0.0.0 255.255.0.0 U 1000 0 0 ens33
192.168.160.0 0.0.0.0 255.255.255.0 U 100 0 0 ens33
andre@ubuntu:~$
```

Figura 28: Esempio di esecuzione del comando ifconfig su Linux.

4.8 Comando whois

Il comando **whois** consente, mediante l'interrogazione di database server da parte di un client, di **stabilire il nome privato, o dell'azienda, o dell'ente, al quale è intestato un determinato indirizzo IP o uno specifico DNS**. Solitamente vengono mostrate anche informazioni riguardanti l'intestatario, data di registrazione e data di scadenza.

```

andre@ubuntu:~$ whois univr.it

*****
* Please note that the following result could be a subgroup of      *
* the data contained in the database.                            *
*                                                               *
* Additional information can be visualized at:                  *
* http://web-whois.nic.it                                         *
*****


Domain:          univr.it
Status:          ok
Signed:          no
Created:         1996-01-29 00:00:00
Last Update:    2023-02-14 00:56:50
Expire Date:   2024-01-29

Registrant
Organization: Universita' di Verona
Address:        Via S.Francesco, 22
                Verona
                37129
                VR
                IT
Created:        2007-03-01 10:49:11
Last Update:   2011-03-24 11:01:07

Admin Contact
Name:            Giovanni Bianco
Address:         SIA - Servizi Informatici di Ateneo
                Via S.Francesco, 22
                Verona
                37129
                VR
                IT
Created:        2006-02-14 00:00:00
Last Update:   2011-03-24 11:01:08

Technical Contacts
Name:            Alberto Manzoni
Address:         SIA - Servizi Informatici di Ateneo
                Via S.Francesco, 22
                Verona
                37129
                VR
                IT
Created:        2004-03-18 00:00:00
Last Update:   2011-03-24 11:01:09

Name:            Andrea Sartori
Address:         Via dell'Artigliere, 19
                Verona
                37129
                VR
                IT
Created:        2004-03-18 00:00:00
Last Update:   2019-03-06 10:55:13

Registrar
Organization: Consortium GARR
Name:           GARR-REG
Web:            http://www.garr.it
DNSSEC:         no

Nameservers
dns01.univr.it
dns02.univr.it
ns1.garr.net

andre@ubuntu:~$
```

Figura 29: Esempio di esecuzione del comando whois su Linux.

4.9 Esercizi

4.9.1 Esercizio 1 - File **capture.cap**

Avviare Wireshark, aprire il menù **File/Open** e selezionare il file **capture.cap**. Si prenda in considerazione il pacchetto numero 9 e si risponda alle seguenti domande/esercizi:

1. Che tipo di protocollo di livello Data-link è utilizzato? Come fa Wireshark a capirlo?
2. Disegnare la PDU di livello Data-link indicando il valore dei vari campi.
3. Qual è il MAC sorgente? Di che tipo è: unicast o broadcast?
4. Qual è il MAC destinatario? Di che tipo è: unicast o broadcast?
5. Che tipo di protocollo di livello Network è utilizzato? Come fa Wireshark a capirlo?
6. Qual è la lunghezza dell'header IP?
7. Quali sono gli indirizzi IP sorgente e destinazione?
8. Che tipo di protocollo di livello trasporto è contenuto in IP? Come fa Wireshark a capirlo?
(suggerimento: basta scrivere **arp** nella barra **Filter** sotto la toolbar; si ricordi di premere su **Apply** dopo aver scritto **arp**).
9. Quali sono le porte sorgente e destinazione a livello trasporto?
10. Creare un filtro per visualizzare solo i pacchetti che hanno ARP come protocollo.
(suggerimento: vedere entrambi i valori nella barra di stato in basso).
11. Dopo aver applicato il filtro precedente qual è la percentuale di pacchetti che rimangono visualizzati rispetto al totale?
(suggerimento: vedere entrambi i valori nella barra di stato in basso).
12. Creare un filtro per visualizzare solo i pacchetti che hanno destinazione MAC 00:01:e6:57:4b:e0.
(suggerimento: usare l'editore di espressioni; la categoria da selezionare è **Ethernet**; per l'indirizzo MAC usare la notazione esadecimale con i due punti come separatori; si ricordi di premere su **Apply** dopo aver creato l'espressione).
13. Dopo aver applicato il filtro precedente qual è la percentuale di pacchetti che rimangono visualizzati rispetto al totale?
(suggerimento: vedere entrambi i valori nella barra di stato in basso).
14. Creare un filtro per visualizzare solo i pacchetti che hanno destinazione MAC broadcast.
(suggerimento: nell'editor di espressioni la categoria da usare è **Ethernet**; per l'indirizzo MAC usare la notazione esadecimale con i due punti come separatori; si ricordi di premere su **Apply** dopo aver creato l'espressione).
15. Dopo aver applicato il filtro precedente qual è la percentuale di pacchetti che rimangono visualizzati rispetto al totale? Sono molti? Perché?

1. Domanda: Che tipo di protocollo di livello Data-link è utilizzato? Come fa Wireshark a capirlo?

1. Risposta: Con il livello Data-link si intende il livello 2, ovvero il livello di collegamento. Tra i vari protocolli disponibili, nel pacchetto numero 9 viene utilizzato il protocollo Ethernet. È individuabile guardando l'header del pacchetto grazie a Wireshark. Esso è possibile vederlo in basso, come in figura.

```
> Frame 9: 227 bytes on wire (1816 bits), 227 bytes captured (1816 bits)
> Ethernet II, Src: TyanComp_24:dd:64 (00:e0:81:24:dd:64), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
  > Destination: Broadcast (ff:ff:ff:ff:ff:ff)
    Address: Broadcast (ff:ff:ff:ff:ff:ff)
    .... ..1. .... .... .... = LG bit: Locally administered address (this is NOT the factory default)
    .... ..1. .... .... .... = IG bit: Group address (multicast/broadcast)
  > Source: TyanComp_24:dd:64 (00:e0:81:24:dd:64)
    Address: TyanComp_24:dd:64 (00:e0:81:24:dd:64)
    .... ..0. .... .... .... = LG bit: Globally unique address (factory default)
    .... ..0. .... .... .... = IG bit: Individual address (unicast)
  Type: IPv4 (0x0800)
> Internet Protocol Version 4, Src: 157.27.252.223, Dst: 157.27.252.255
> User Datagram Protocol, Src Port: 631, Dst Port: 631
> Common Unix Printing System (CUPS) Browsing Protocol
```

Figura 30: Header livello Data-link del pacchetto numero 9.

Grazie alla libreria `libpcap`, Wireshark riesce ad ottenere i pacchetti dall'interfaccia di rete interessata dall'utente. Per analizzare il pacchetto, ha bisogno di una serie di strumenti che vengono concessi dalla libreria `libpcap` e non solo. La struttura di Wireshark è visibile nella seguente figura:

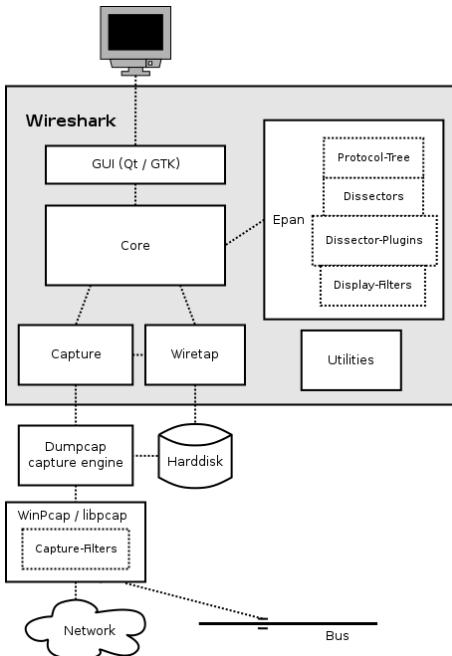


Figura 31: Struttura interna di Wireshark ([fonte ufficiale](#)).

- La parte di *GUI* riguarda l’interfaccia mostrata all’utente e dunque gestisce tutti gli input e output del software;
- La parte di *core* è il cuore pulsante e controlla gli altri strumenti collegati (Capture, Wiretap, Epan). In gergo viene chiamato *glue code*;
- *Wiretap* è una libreria utilizzata per leggere e scrivere i file catturati in formato libpcap, pcapng e altri;
- *Capture* è l’interfaccia del motore di cattura;
- **Epan** (*Enhanced Packet Analyzer*) è il motore di analisi per i pacchetti. Al suo interno è possibile trovare quattro strumenti fondamentali:
 - *Protocol Tree* che consente di dissezionare le informazioni da un pacchetto;
 - *Dissectors* che contiene i vari protocolli dissezionati;
 - *Dissector Plugins* che implementa strumenti di supporto per dissezionare;
 - *Display Filters* che implementa un motore per effettuare i filtri.

Quindi, Wireshark è in grado di capire il protocollo grazie alla sua struttura e in particolare alla parte *Epan*.

2. Domanda: *Disegnare la PDU di livello Data-link indicando il valore dei vari campi.*

2. Risposta: Nella risposta alla precedente domanda, è possibile vedere sia il protocollo che il suo *header*.

3. Domanda: *Qual è il MAC sorgente? Di che tipo è: unicast o broadcast?*

3. Risposta: Guardando la figura della prima risposta a pagina 110 è possibile visualizzare il MAC sorgente: 00:e0:81:24:dd:64. Si deduce che è di tipo *unicast*.

4. Domanda: *Qual è il MAC destinatario? Di che tipo è: unicast o broadcast?*

4. Risposta: Guardando la figura della prima risposta a pagina 110 è possibile visualizzare il MAC destinatario: ff:ff:ff:ff:ff:ff. Si deduce che è di tipo *broadcast*.

5. Domanda: Che tipo di protocollo di livello Network è utilizzato? Come fa Wireshark a capirlo?

5. Risposta: Il protocollo a livello Network è IPv4 (Internet Protocol Version 4):

```
> Frame 9: 227 bytes on wire (1816 bits), 227 bytes captured (1816 bits)
> Ethernet II, Src: TyanComp_24:dd:64 (00:e0:81:24:dd:64), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
< Internet Protocol Version 4, Src: 157.27.252.223, Dst: 157.27.252.255
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
    < Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
        0000 00.. = Differentiated Services Codepoint: Default (0)
        .... ..00 = Explicit Congestion Notification: Not ECN-Capable Transport (0)
    Total Length: 213
    Identification: 0x0000 (0)
    < 010. .... = Flags: 0x2, Don't fragment
        0... .... = Reserved bit: Not set
        .1. .... = Don't fragment: Set
        ..0. .... = More fragments: Not set
        ...0 0000 0000 0000 = Fragment Offset: 0
    Time to Live: 64
    Protocol: UDP (17)
    Header Checksum: 0x0602 [validation disabled]
        [Header checksum status: Unverified]
    Source Address: 157.27.252.223
    Destination Address: 157.27.252.255
    < User Datagram Protocol, Src Port: 631, Dst Port: 631
    > Common Unix Printing System (CUPS) Browsing Protocol
```

Figura 32: Protocollo IPv4 a livello Network nel pacchetto numero 9.

Wireshark per capirlo utilizza lo stesso meccanismo utilizzato anche per il livello Data-link.

6. Domanda: Qual è la lunghezza dell'header IP?

6. Risposta: Come si vede dalla foto sopra, il campo Header Length comunica che la lunghezza dell'header è di 20 bytes.

7. Domanda: Quali sono gli indirizzi IP sorgente e destinazione?

7. Risposta: Come si vede dalla foto sopra, l'IP sorgente è 157.27.252.223 e l'IP destinazione è 157.27.252.255.

8. Domanda: Che tipo di protocollo di livello trasporto è contenuto in IP? Come fa Wireshark a capirlo?

8. Risposta: Come si vede dalla foto sopra nel campo Protocol, il protocollo contenuto è UDP (User Datagram Protocol). Wireshark riesce a capirlo per gli stessi motivi spiegati per il protocollo Ethernet.

9. Domanda: Quali sono le porte sorgente e destinazione a livello trasporto?

9. Risposta: A livello di trasporto, il pacchetto contiene i seguenti dati:

```
> Frame 9: 227 bytes on wire (1816 bits), 227 bytes captured (1816 bits)
> Ethernet II, Src: TyanComp_24:dd:64 (00:e0:81:24:dd:64), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
> Internet Protocol Version 4, Src: 157.27.252.223, Dst: 157.27.252.255
< User Datagram Protocol, Src Port: 631, Dst Port: 631
    Source Port: 631
    Destination Port: 631
    Length: 193
    Checksum: 0x7e9e [unverified]
        [Checksum Status: Unverified]
        [Stream index: 0]
    < [Timestamps]
        [Time since first frame: 0.000000000 seconds]
        [Time since previous frame: 0.000000000 seconds]
    UDP payload (185 bytes)
> Common Unix Printing System (CUPS) Browsing Protocol
```

Figura 33: Protocollo UDP a livello di trasporto nel pacchetto numero 9.

Come si vede dalla figura, la porta sorgente è la 631 e quella di destinazione è identica.

10. Domanda: Creare un filtro per visualizzare solo i pacchetti che hanno ARP come protocollo.

(suggerimento: basta scrivere arp nella barra Filter sotto la toolbar; si ricordi di premere su **Apply** dopo aver scritto arp).

10. Risposta: Come veniva anticipato tramite un esempio al capitolo 4.2.2, la creazione di un filtro è possibile eseguirla scrivendo nella barra in alto. Il risultato dell'esercizio:

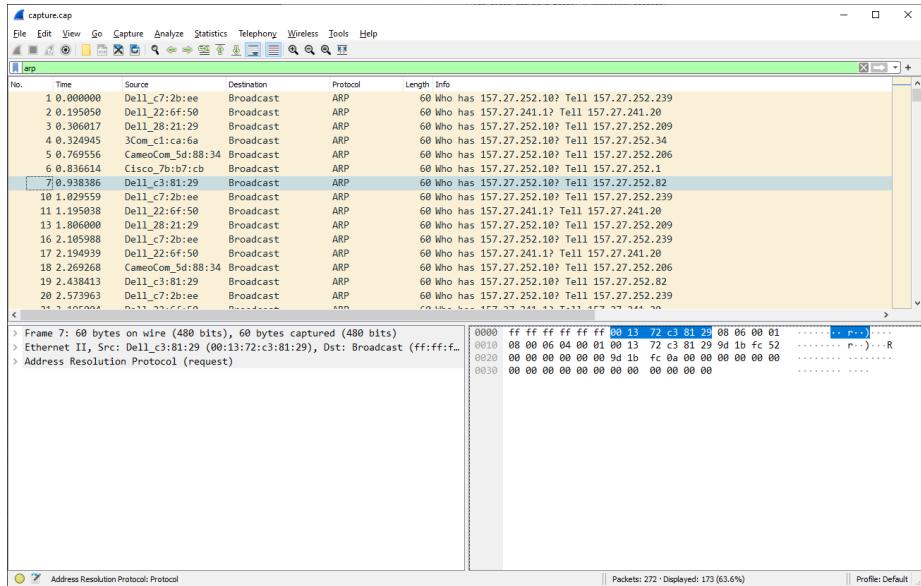


Figura 34: Applicazione del filtro arp all'interno del pacchetto capture.cap.

11. Domanda: Dopo aver applicato il filtro precedente qual è la percentuale di pacchetti che rimangono visualizzati rispetto al totale? (suggerimento: vedere entrambi i valori nella barra di stato in basso).

11. Risposta: La percentuale di pacchetti che rimangono è pari al 63.6%, cioè 173 pacchetti su un totale di 272.

12. Domanda: Creare un filtro per visualizzare solo i pacchetti che hanno destinazione MAC 00:01:e6:57:4b:e0.

(suggerimento: usare l'editore di espressioni; la categoria da selezionare è **Ethernet**; per l'indirizzo MAC usare la notazione esadecimale con i due punti come separatori; si ricordi di premere su **Apply** dopo aver creato l'espressione).

12. Risposta: L'indirizzo MAC è nel protocollo Ethernet. Dunque, è necessario selezionare la voce MAC di questo protocollo e verificare che sia uguale (condizione logica) all'indirizzo MAC fornito dall'esercizio. Nel campo di filtro quindi si scriverà:

```
eth.dst == 00:01:e6:57:4b:e0
```

In questo modo viene indicata come destinazione l'indirizzo MAC. Wireshark capisce che si sta parlando dell'indirizzo MAC perché è stato specificato il protocollo Ethernet (eth). Il **risultato** è il pacchetto numero 12.

13. Domanda: Dopo aver applicato il filtro precedente qual è la percentuale di pacchetti che rimangono visualizzati rispetto al totale? (suggerimento: vedere entrambi i valori nella barra di stato in basso).

13. Risposta: La percentuale è del 0.4% e cioè 1 pacchetto.

14. Domanda: Creare un filtro per visualizzare solo i pacchetti che hanno destinazione MAC broadcast.

(suggerimento: nell'editor di espressioni la categoria da usare è **Ethernet**; per l'indirizzo MAC usare la notazione esadecimale con i due punti come separatori; si ricordi di premere su **Apply** dopo aver creato l'espressione).

14. Risposta: Per visualizzare i pacchetti con destinazione broadcast si applica lo stesso filtro della domanda 12 ma con indirizzo di destinazione broadcast, ovvero:

```
eth.dst == ff:ff:ff:ff:ff:ff
```

15. Domanda: Dopo aver applicato il filtro precedente qual è la percentuale di pacchetti che rimangono visualizzati rispetto al totale? Sono molti? Perché?

15. Risposta: Il protocollo ARP viene utilizzato per conoscere gli indirizzi MAC degli altri host e per conoscere i MAC in una rete privata. Quindi, questo protocollo lavora principalmente sul canale *broadcast* dato che deve aggiornare spesso le varie tabelle di ARP. Il **risultato** post filtro è di 228 pacchetti trovati con una percentuale di 83.8% su 272.

4.9.2 Esercizio 2 - File simpleHTTP.cap

Occorre aprire il menu **File/Open** e selezionare il file **simpleHTTP.cap**.

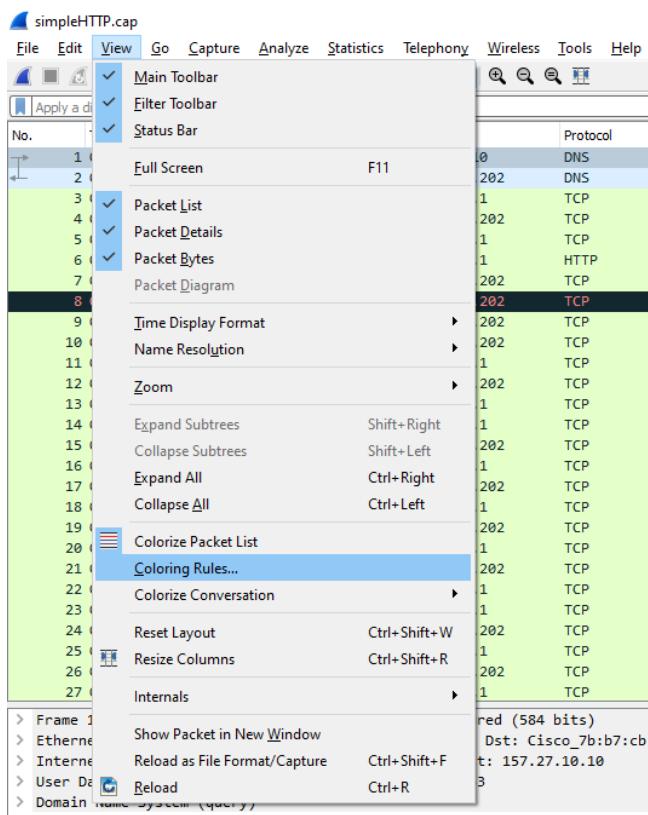
1. Colorare di rosso tutti i pacchetti che contengono UDP e di verde tutti i pacchetti che contengono TCP.
(suggerimento: nell'editore delle regole di colorazione è sufficiente portare in alto due regole già esistenti e modificarle per cambiarne i colori di sfondo).
2. Cosa contengono i primi due pacchetti della sessione di cattura?
 - IP sorgente, IP destinazione.
 - Tipo di protocollo di trasporto.
 - Tipo di protocollo di livello Applicazione. Come fa Wireshark a capirlo?
 - Messaggio contenuto nel Payload di livello Applicazione.
3. Prendere in considerazione il pacchetto n. 3.
 - IP sorgente, IP destinazione.
 - Tipo di protocollo di trasporto.
 - IP sorgente e destinazione sono in qualche modo collegati con i messaggi scambiati a livello applicazione nei primi due pacchetti? È possibile fare delle ipotesi su cosa serve il protocollo di livello applicazione dei primi due pacchetti?
4. Prendere in considerazione il pacchetto n. 6.
 - IP sorgente, IP destinazione.
 - Tipo di protocollo di trasporto.
 - Tipo di protocollo di livello Applicazione.
 - Perché prima della trasmissione del primo messaggio HTTP c'è lo scambio di tre pacchetti puramente TCP? Quali sono i flag settati nell'header TCP di questi tre pacchetti?
5. Creare un filtro per visualizzare solo i pacchetti TCP (compresi i pacchetti HTTP) e determinarne il numero.
6. Creare un filtro per visualizzare solo i pacchetti TCP (esclusi i pacchetti HTTP) e determinarne il numero.
 - Qual è la percentuale sul totale dei pacchetti TCP trovata al punto 5?
 - A cosa servono tali pacchetti?
 - Se il protocollo DNS dei pacchetti 1 e 2 avesse usato il protocollo TCP, quanti pacchetti IP sarebbero stati generati? Sarebbe stato utile?
7. Selezionare il pacchetto 3 e seguire lo stream TCP col comando da menu **Analyze/Follow TCP Stream**.

- Cosa si può leggere?
 - Qual è il messaggio contenuto nel payload della PDU di livello Applicazione?
-

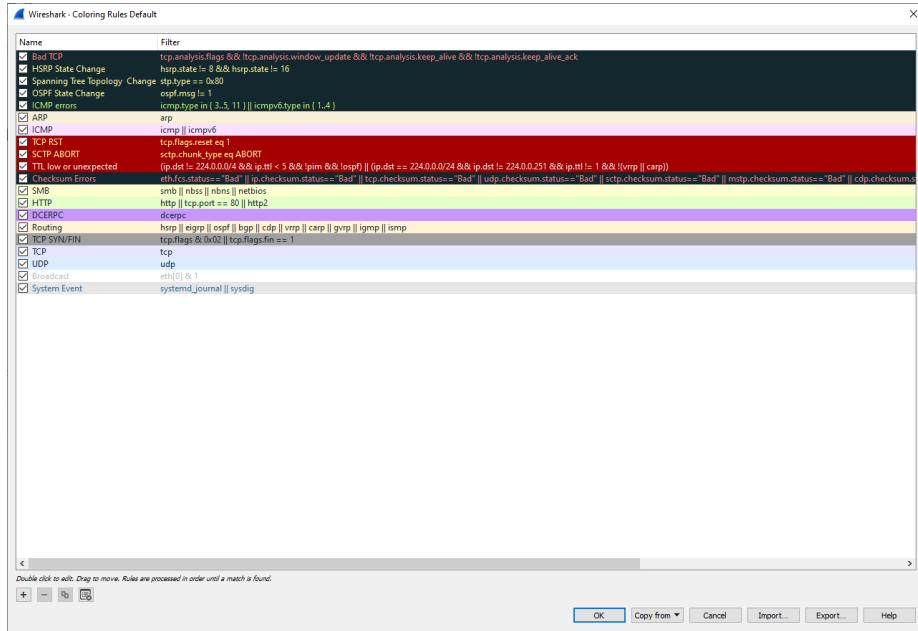
1. Domanda: Colorare di rosso tutti i pacchetti che contengono UDP e di verde tutti i pacchetti che contengono TCP.

(suggerimento: nell'editore delle regole di colorazione è sufficiente portare in alto due regole già esistenti e modificarle per cambiarne i colori di sfondo).

1. Risposta: Il colore dei pacchetti è possibile modificarlo cambiando le regole di colorazione dal menù. Quindi, cliccando su View → Coloring Rules...



All'apertura della schermata:



Basterà cliccare su ogni voce interessata, andare in basso a destra e modificare i colori:



2. Domanda: Cosa contengono i primi due pacchetti della sessione di cattura?

- **IP sorgente, IP destinazione.**
- **Tipo di protocollo di trasporto.**
- **Tipo di protocollo di livello Applicazione. Come fa Wireshark a capirlo?**
- **Messaggio contenuto nel Payload di livello Applicazione.**

2. Risposta: Il primo pacchetto contiene l'IP sorgente 157.27.252.202 e l'IP destinazione 157.27.10.10. Il secondo pacchetto, essendo una risposta al primo, contiene l'IP sorgente 157.27.10.10 e l'IP destinazione 157.27.252.202.

Il protocollo di trasporto utilizzato è UDP. Invece, il protocollo utilizzato a livello di applicazione è DNS. Wireshark riesce a capirlo grazie alla sua grande librerie e alla possibilità di analizzare il pacchetto (spiegazione ampia a pagina 110).

Il primo pacchetto ha all'interno del payload una *query* al DNS per capire l'indirizzo IP del sito web www.polito.it.

Invece, il secondo pacchetto ha all'interno del payload la risposta alla *query* (si veda l'immagine sotto). Quindi, contiene:

- *Queries*: la *query* che le è stata fatta;
- *Answer*: il percorso che deve fare il pacchetto per arrivare al server di destinazione (e viceversa);
- *Authoritative nameservers*: sono i server che si occupano di rispondere alle richieste ricorsive dei DNS. Per capire meglio: [link sito Cisco](#);
- *Additional records*: informazioni in più utili al mittente per trovare la destinazione.

```
    < Queries
      > www.polito.it: type A, class IN
    < Answers
      < www.polito.it: type CNAME, class IN, cname web01.polito.it
        Name: www.polito.it
        Type: CNAME (Canonical NAME for an alias) (5)
        Class: IN (0x0001)
        Time to live: 63277 (17 hours, 34 minutes, 37 seconds)
        Data length: 8
        CNAME: web01.polito.it
      < web01.polito.it: type A, class IN, addr 130.192.73.1
        Name: web01.polito.it
        Type: A (Host Address) (1)
        Class: IN (0x0001)
        Time to live: 63277 (17 hours, 34 minutes, 37 seconds)
        Data length: 4
        Address: 130.192.73.1
      < Authoritative nameservers
        < polito.it: type NS, class IN, ns leonardo.polito.it
          Name: polito.it
          Type: NS (authoritative Name Server) (2)
          Class: IN (0x0001)
          Time to live: 55296 (15 hours, 21 minutes, 36 seconds)
          Data length: 11
          Name Server: leonardo.polito.it
        > polito.it: type NS, class IN, ns ns1.garr.net
        > polito.it: type NS, class IN, ns giove.polito.it
      < Additional records
        < ns1.garr.net: type A, class IN, addr 193.206.141.38
          Name: ns1.garr.net
          Type: A (Host Address) (1)
          Class: IN (0x0001)
          Time to live: 61528 (17 hours, 5 minutes, 28 seconds)
          Data length: 4
          Address: 193.206.141.38
        > ns1.garr.net: type AAAA, class IN, addr 2001:760:ffff:ffff::aa
        > giove.polito.it: type A, class IN, addr 130.192.3.24
        > leonardo.polito.it: type A, class IN, addr 130.192.3.21
      [Request In: 1]
      [Time: 0.004021000 seconds]
```

3. Domanda: Prendere in considerazione il pacchetto n. 3.

- IP sorgente, IP destinazione.
- Tipo di protocollo di trasporto.
- IP sorgente e destinazione sono in qualche modo collegati con i messaggi scambiati a livello applicazione nei primi due pacchetti? È possibile fare delle ipotesi su cosa serve il protocollo di livello applicazione dei primi due pacchetti?

3. Risposta: L'indirizzo IP della sorgente 157.27.252.202 e l'IP della destinazione è 130.192.73.1.

Il protocollo di trasporto è TCP.

L'IP sorgente corrisponde all'IP che ha eseguito una richiesta al server DNS per trovare l'indirizzo IP del sito www.polito.it. Nel terzo pacchetto, l'indirizzo di destinazione è 130.192.73.1 ed è lo stesso comunicato dal server DNS e visibile a livello di applicazione nel pacchetto numero 2. Osservando anche la figura alla pagina precedente, è possibile vedere l'indirizzo IP 130.192.73.1 sotto la voce *Answer*.

4. Domanda: Prendere in considerazione il pacchetto n. 6.

- IP sorgente, IP destinazione.
- Tipo di protocollo di trasporto.
- Tipo di protocollo di livello Applicazione.
- Perché prima della trasmissione del primo messaggio HTTP c'è lo scambio di tre pacchetti puramente TCP? Quali sono i flag settati nell'header TCP di questi tre pacchetti?

4. Risposta: L'indirizzo IP della sorgente 157.27.252.202 e l'IP della destinazione è 130.192.73.1.

Il protocollo di trasporto è TCP e il protocollo a livello di applicazione HTTP.

I tre pacchetti scambiati prima del primo messaggio HTTP riguardano il *three-way handshake*. Questa tecnica viene utilizzata dal protocollo TCP per assicurarsi che il mittente e la destinazione siano pronti per ricevere e inviare i messaggi. Il primo dei tre pacchetti TCP ha la flag SYN ad 1, il secondo ha le due flag SYN e ACK ad 1 e infine, il terzo pacchetto ha la flag ACK ad 1.

5. Domanda: Creare un filtro per visualizzare solo i pacchetti TCP (compresi i pacchetti HTTP) e determinarne il numero.

5. Risposta: Il filtro da applicare è semplicemente `tcp && http`. In questo modo si ottengono tutti quei pacchetti che a livello di trasporto hanno il seguente protocollo. Il numero di pacchetti corrisponde a 134 su 823 totali.

6. Domanda: Creare un filtro per visualizzare solo i pacchetti TCP (esclusi i pacchetti HTTP) e determinarne il numero.

- Qual è la percentuale sul totale dei pacchetti TCP trovata al punto 5?
- A cosa servono tali pacchetti?
- Se il protocollo DNS dei pacchetti 1 e 2 avesse usato il protocollo TCP, quanti pacchetti IP sarebbero stati generati? Sarebbe stato utile?

6. Risposta: Il filtro da applicare è `tcp && !http`. Il numero di pacchetti trovati al punto precedente è 134 su 823, che corrisponde al 16.3%.

Tali pacchetti vengono utilizzati per effettuare le richieste REST, quindi sono pacchetti HTTP di tipo GET, OK.

Se il protocollo DNS dei primi due pacchetti avesse utilizzato il protocollo TCP a livello di trasporto, avrebbe utilizzato risorse in più e non sarebbe stato utile poiché avrebbe rallentato la risposta, a causa del three-way handshake iniziale. Il numero di pacchetti scambiati sarebbero stati minimo 5, di cui tre per instaurare la connessione e due per scambiarsi le *query*. Se in aggiunta una delle due parti avesse voluto concludere la comunicazione, sarebbero serviti altri tre pacchetti, per un totale di 8 pacchetti.

7. Domanda: Selezionare il pacchetto 3 e seguire lo stream TCP col comando da menu **Analyze/Follow TCP Stream**.

- Cosa si può leggere?
- Qual è il messaggio contenuto nel payload della PDU di livello Applicazione?

7. Risposta: Quello che è possibile leggere sono le varie pagine HTTP inviate dal sito www.polito.it al mittente. Inoltre, è possibile osservare anche l'invio di altri dati come file (.gif, .jpeg).

Il messaggio contenuto nel payload è il codice della pagina web del politecnico di Torino.

4.9.3 Esercizio 3 - File busyNetwork.cap

Occorre aprire il menu File/Open e selezionare il file busyNetwork.cap.

1. Elencare i protocolli di livello Applicazione che entrano in azione in questa cattura classificandoli in base al livello di trasporto utilizzato.
 2. Provare ad analizzare diversi stream TCP con sopra diversi protocolli di livello applicazione.
 3. Che differenza c'è tra il contenuto trasmesso in una connessione TCP per il protocollo FTP e quello trasmesso per il protocollo SSH?
-

1. Domanda: *Elencare i protocolli di livello Applicazione che entrano in azione in questa cattura classificandoli in base al livello di trasporto utilizzato.*

1. Risposta: Analizzando il file è possibile trovare 1392 pacchetti al suo interno. A livello di trasporto vengono usati due protocolli: TCP e UDP.

- Con il protocollo UDP a livello di trasporto, a livello di applicazione viene utilizzato solo il protocollo DNS;
- Con il protocollo TCP a livello di trasporto, a livello di applicazione vengono utilizzati:
 - SSH
 - Short Frame
 - HTTP
 - FTP
 - Data

Le precedenti statistiche sono state ottenute andando sul menù delle applicazioni → Statistics → Protocol Hierarchy e il risultato che si ottiene è il seguente:

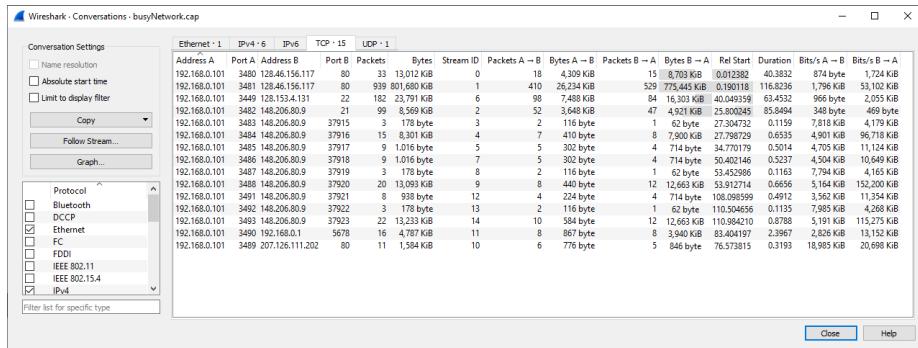
Protocol	Percent Packets	Packets	Percent Bytes	Bytes	Bits/s	End Packets	End Bytes	End Bits/s	PDUs
Frame	100.0	1392	9.8	89337	6107	0	0	0	1392
Ethernet	100.0	1392	2.1	19634	1342	0	0	0	1392
Internet Protocol Version 4	100.0	1392	3.0	27840	1903	0	0	0	1392
User Datagram Protocol	0.3	4	0.0	32	2	0	0	0	4
Domain Name System	0.3	4	0.0	104	7	0	0	0	4
Short Frame	0.3	4	0.0	0	0	4	0	0	4
Transmission Control Protocol	99.7	1388	4.6	41727	2852	650	16754	1145	1388
SSH Protocol	9.6	133	0.2	1862	127	0	0	0	133
Short Frame	10.7	149	0.0	0	0	149	0	0	149
Hypertext Transfer Protocol	34.1	475	0.7	6650	454	213	2982	203	475
Short Frame	18.8	262	0.0	0	0	262	0	0	262
File Transfer Protocol (FTP)	6.0	83	0.1	1043	71	83	1043	71	83
Data	2.2	31	0.0	434	29	31	434	29	31

2. Domanda: Provare ad analizzare diversi stream TCP con sopra diversi protocolli di livello applicazione.

2. Risposta: Provando a filtrare per ogni protocollo a livello di applicazione, si ottiene il seguente risultato:

- Protocollo SSH: vi è uno scambio di dati tra client e server. Purtroppo, provando ad analizzare lo stream, la conversazione è illeggibile poiché criptata;
- Protocollo HTTP: vi è una serie di richieste GET da parte del client e una serie di risposte OK da parte del server. Il contenuto non sembra comprensibile;
- Protocollo FTP: vi è una serie di messaggi che sembra un tentativo da parte del client di login.

Per vedere tutte le possibili conversazioni e tutti i possibili stream che è possibile seguire, è necessario andare su **Statistics → Conversations** e apparirà la seguente schermata:



3. Domanda: Che differenza c'è tra il contenuto trasmesso in una connessione TCP per il protocollo FTP e quello trasmesso per il protocollo SSH?

3. Risposta: Analizzando i pacchetti con il protocollo FTP e SSH, l'intestazione a livello di TCP non varia di molto. La grande differenza è che per il protocollo FTP i dati scambiati sono visibili in chiaro, mentre i dati scambiati con il protocollo SSH sono criptati dal client e dal server, quindi non è possibile leggerne il contenuto.

4.9.4 Esercizio 4 - File pingCapture.cap

Occorre aprire il menu File/Open e selezionare il file pingCapture.cap.

1. Individuare le richieste ping inviate e le relative risposte. Quante sono?
 2. Quali sono IP sorgente e destinazione della richiesta ICMP? A quale ente o azienda sono intestati?
 3. Provare a invocare il comando ping dal proprio PC verso www.google.com e verso il proprio Default Gateway (come faccio a sapere il suo IP?) e osservare il RTT medio e la sua variazione. Chi mostra la media più grande? Perché? Chi mostra la variazione più grande? Perché?
-

1. Domanda: *Individuare le richieste ping inviate e le relative risposte. Quante sono?*

1. Risposta: Per individuare le richieste ping inviate e ricevute è necessario applicare il filtro icmp. Questo perché le richieste/risposte ping utilizzano il protocollo ICMP (Internet Control Message Protocol), il quale è sfruttato principalmente per trasmettere informazioni riguardanti malfunzionamenti, informazioni di controllo o messaggi. I pacchetti sono in totale 22.

2. Domanda: *Quali sono IP sorgente e destinazione della richiesta ICMP? A quale ente o azienda sono intestati?*

2. Risposta: L'IP sorgente, quindi l'host che esegue le richieste, è 157.27.143.46. Mentre l'IP destinazione, quindi l'host che risponde alle richieste, è 216.58.211.196. L'azienda a cui è intestato l'indirizzo IP è PaloAlto. Questo dato è possibile ottenerlo guardando a livello logico (*link layer*) il protocollo Ethernet e osservando il mittente e la destinazione.

3. Domanda: *Provare a invocare il comando ping dal proprio PC verso www.google.com e verso il proprio Default Gateway (come faccio a sapere il suo IP?) e osservare il RTT medio e la sua variazione. Chi mostra la media più grande? Perché? Chi mostra la variazione più grande? Perché?*

3. Risposta: Eseguendo il comando ping www.google.com, si ottiene un RTT medio di 10ms. Invece, eseguendo il comando ping e inserendo l'ip del proprio default gateway (ottenibile eseguendo il comando ipconfig su Windows, paragrafo 4.6), il tempo medio RTT è di 2ms. Questa netta differenza è dovuta al fatto che il server di google è sicuramente più distante del default gateway che è il router all'interno della rete che consente di collegarsi ad Internet.

4.9.5 Esercizio 5 - Comando traceroute

Entrare nel sistema Linux e digitare il comando traceroute www.google.com

1. Individuare le interfacce dei router attraversati;
 2. Individuare i nomi delle organizzazioni a cui sono intestati gli IP delle interfacce dei router attraversati.
-

1. Domanda: *Individuare le interfacce dei router attraversati.*

1. Risposta: Dopo aver eseguito il comando traceroute è possibile vedere un risultato del tipo:

```
C:\Users\sforz>tracert www.google.com

Traccia instradamento verso www.google.com [142.250.180.132]
su un massimo di 30 punti di passaggio:
      1    2 ms    2 ms    1 ms  xxx.xxx.xxx.xxx < Default Gateway
      2    *        *        *      Richiesta scaduta.
      3    2 ms    3 ms    2 ms  198.51.100.1
      4    2 ms    2 ms    4 ms  ru-univr-l1-rx2-pd2.pd2.garr.net [193.204.218.109]
      5   12 ms   11 ms   11 ms  185.191.181.238
      6    *        *        *      Richiesta scaduta.
      7    *        *        *      Richiesta scaduta.
      8   61 ms   12 ms   10 ms  rs1-bo01-re1-mi02.mi02.garr.net [185.191.180.57]
      9   13 ms   14 ms   14 ms  142.250.164.230
     10   14 ms   14 ms   15 ms  108.170.245.65
     11   14 ms   14 ms   13 ms  142.250.211.29
     12   11 ms   11 ms   11 ms  mil04s43-in-f4.1e100.net [142.250.180.132]

Traccia completata.
```

In cui i router sono identificabili sulla colonna di destra.

2. Domanda: *Individuare i nomi delle organizzazioni a cui sono intestati gli IP delle interfacce dei router attraversati.*

2. Risposta: Aprendo il terminale di Linux e usando il comando whois seguito dall'indirizzo IP interessato, è possibile notare che:

- L'indirizzo IP 198.51.100.1, si riferisce all'organizzazione IANA (Internet Assigned Numbers Authority).
- L'indirizzo ru-univr-l1-rx2-pd2.pd2.garr.net [193.204.218.109], è di Consortium GARR (Italian academic and research network).
- L'indirizzo IP 185.191.181.238, si riferisce al ORG-GIRa1-RIPE, ovvero DIR-GARR - Roma.
- L'indirizzo rs1-bo01-re1-mi02.mi02.garr.net [185.191.180.57], è di Consortium GARR (Italian academic and research network).
- L'indirizzo IP 142.250.164.230, si riferisce all'organizzazione Google LLC.

- L'indirizzo IP 108.170.245.65, si riferisce all'organizzazione Google LLC.
 - L'indirizzo IP 142.250.211.29, si riferisce all'organizzazione Google LLC.
 - L'indirizzo mil04s43-in-f4.1e100.net [142.250.180.132], è di Google LLC.
-

4.9.6 Esercizio 6 - Interfacce di rete

1. Cercare quali interfacce sono attualmente attive sul proprio PC. Qual è l'indirizzo IP dell'interfaccia che state utilizzando sul vostro host? E la netmask corrispondente?
 2. Qual è l'indirizzo IP di www.univr.it?
-

1. Domanda: *Cercare quali interfacce sono attualmente attive sul proprio PC. Qual è l'indirizzo IP dell'interfaccia che state utilizzando sul vostro host? E la netmask corrispondente?*

1. Risposta: Per conoscere le interfacce di rete attive sul proprio PC è necessario utilizzare il comando ipconfig su Windows e ifconfig su Linux. L'indirizzo IP è possibile trovarlo accanto alla voce IPv4 (e IPv6) per ogni interfaccia utilizzata. La netmask corrispondente è possibile vederla accanto alla voce *mask*.

2. Domanda: *Qual è l'indirizzo IP di www.univr.it?*

2. Risposta: L'indirizzo IP di www.univr.it è possibile trovarlo scrivendo un semplice comando di ping. Si scopre che facendo ping www.univr.it l'indirizzo IP è 157.27.3.60.

5 Docker

- 5.1 Da quali bisogni è nato Docker?**
- 5.2 Virtualizzazione e Containerizzazione**
- 5.3 Struttura tecnica di Docker**
 - 5.3.1 Dockerfile**
 - 5.3.2 Docker image**
 - 5.3.3 Docker container**
- 5.4 Comandi utili**
- 5.5 Esercizi ed esempio di utilizzo**
 - 5.5.1 Esempio**
 - 5.5.2 Esercizi**
- 5.6 Breve introduzione a Docker Compose**