

Programmazione e sicurezza delle reti

UNIVR - Dipartimento di Informatica

Amos Lo Verde

10 settembre 2023

Indice

Prefazione	1
1 Programmazione di rete con interfaccia socket	3
1.1 Host, processo e applicazione	3
1.1.1 Scoprire l'indirizzo IP degli host	4
1.1.2 Applicazioni orientate al datagram e alla connessione	4
1.2 Modello Client e Server	5
1.2.1 Creazione dell'interfaccia Socket (fase 1)	5
1.2.2 Trasmissione e ricezione (fase 2)	6
2 Approccio ai WebSocket	9
2.1 Introduzione ai WebSocket	9
2.1.1 Limitazioni dei WebSocket	10
2.1.2 Introduzione Node.js	10
2.1.3 Approccio asincrono	10
2.2 Descrizione esempio della chat	10
2.2.1 Introduzione a Express	11
2.2.2 Creazione WebSocket	11
2.2.3 Funzionamento del back-end	12
2.2.4 Funzionamento del front-end	13
2.2.5 Implementazione del front-end della chat	13
2.2.6 Esecuzione della parte front-end e back-end	14
3 Analisi di rete con Wireshark e da linea di comando	15
3.1 Introduzione agli analizzatori di rete	15
3.1.1 Concetti chiave dello sniffing	15
3.2 Utilizzo di Wireshark	16
3.2.1 Cattura dei pacchetti	16
3.2.2 Applicazione dei filtri nella cattura	18
3.2.3 Finestra principale	18
3.2.4 Regole di colorazione	19
3.2.5 Applicazione dei filtri di visualizzazione nella finestra principale	20
3.2.6 Analisi del flusso TCP	20

3.2.7	Visualizzazione del livello Data-link delle PDU non Ethernet	20
3.2.8	Cattura di traffico di rete all'interno di una macchina virtuale	21
3.3	Comandi di rete	22
3.3.1	Comando ping	22
3.3.2	Comando traceroute	22
3.3.3	Comando nslookup	23
3.3.4	Comando ifconfig	24
3.3.5	Comando route	25
3.3.6	Comando whois	25
4	Dal Web ai Webservices	27
4.1	Protocolli HTTP/HTTPS	27
4.1.1	Richiesta HTTP	28
4.1.2	Risposta HTTP	29
4.2	Linguaggi per le pagine web	30
4.2.1	Linguaggio HTML	30
4.2.2	Linguaggio CSS	33
4.2.3	Linguaggio JavaScript	33
4.3	Web server	35
4.3.1	Common Gateway Interface (CGI)	36
4.3.2	WebSocket	37
4.4	Scrittura di applicazioni che usano la rete	38
4.4.1	Architetture orientate ai servizi SOA	38
4.4.2	Webservice basati su REST	41
4.4.3	Differenze tra SOA e REST	43
4.5	Cloud computing	43
5	Utilizzo di Docker	47
5.1	Virtualizzazione e containerizzazione	48
5.2	Struttura tecnica di Docker	49
5.2.1	Struttura di un Dockerfile	49
5.2.2	Struttura di un Docker Image	50
5.2.3	Struttura di un Docker container	50
5.2.4	Comandi utili	50
5.3	Introduzione a Docker Compose	51
6	Paradigma Publish/Subscribe	53
6.1	Modello Pub/Sub	53
6.2	Confronto tra Client/Server e Pub/Sub	54
6.2.1	Aspetti positivi dell'architettura Pub/Sub	56
6.3	Protocollo MQTT	56
6.3.1	QoS livello 0	56
6.3.2	QoS livello 1	57
6.3.3	QoS livello 2	58

<i>INDICE</i>	iii
6.4 Vari approfondimenti	58
6.4.1 Struttura del topic	58
6.4.2 Approfondimento Wildcards	59
6.4.3 Approfondimento Broker	59
7 Sicurezza delle reti	61
7.1 Principi fondamentali della sicurezza	61
7.1.1 Quali risorse si vogliono proteggere?	61
7.1.2 In che modo le risorse sono minacciate?	63
7.1.3 Cosa bisogna fare per contrastare le minacce?	64
7.2 La crittografia	66
7.2.1 Elementi del processo crittografico	67
7.2.2 Tipologie di crittografia	68
7.3 Meccanismi di autenticazione e autorizzazione	73
7.3.1 Firma digitale	73
7.3.2 Analisi dell'autenticazione	76
7.3.3 Compiti della CA	79
7.3.4 Autorizzazione (controllo degli accessi)	80
7.4 Firewall e IDS	83
7.4.1 I firewall	83
7.4.2 Sistema di rilevamento delle intrusioni (IDS)	85
7.4.3 Architettura di un IDS	87
8 Introduzione alla programmazione web	89
8.1 Modello client/server	89
8.1.1 AJAX	89
8.1.2 JSON	90
8.2 Strumenti del browser	90
9 WePlant	91
9.1 Configurazione su Linux	91
9.2 Struttura dell'applicazione	93
9.2.1 Lato server	94
9.2.2 Lato client	94
9.3 Sistemi di sicurezza	95
9.3.1 Autenticazione password	95
9.3.2 Autenticazione e autorizzazione API REST: JWT	95
9.4 Archiviazione	97
9.5 Documentazione API	97
9.6 Design Responsive	97

10 Esercizi del corso	99
10.1 Esercizi sulle applicazioni di rete con interfaccia socket	99
10.1.1 Esercizio 1 - UDP	99
10.1.2 Esercizio 2 - UDP	99
10.1.3 Esercizio 3 - UDP	100
10.1.4 Esercizio 4 - UDP	101
10.1.5 Esercizio 5 - UDP [In laboratorio Delta]	101
10.1.6 Esercizio 6 - UDP	102
10.1.7 Esercizio 7 - UDP	102
10.1.8 Esercizio 8 - Sommatrice UDP	104
10.1.9 Esercizio 9 - Sommatrice UDP e perdita di pacchetti	105
10.1.10 Esercizio 10 - Sommatrice UDP e influenze reciproche	106
10.1.11 Esercizio 11 - Sommatrice TCP	108
10.1.12 Esercizio 12 - Sommatrice TCP e influenze reciproche	109
10.1.13 Esercizio 13 - Sommatrice TCP e perdita dei pacchetti	109
10.1.14 Esercizio 14 - Trasferimento di un file	110
10.2 Esercizi dal Web ai Webservices	113
10.2.1 Esercizio di HTML e JavaScript	113
10.2.2 Esercizio su un semplice web server	113
10.2.3 Passare dei dati al server web col metodo GET	115
10.2.4 Passare dei dati al server web col metodo POST	116
10.3 Esercizi di modifica del web server	117
10.3.1 Esercizio 1 - Ricerca di una pagina in locale	117
10.3.2 Esercizio 2 - Upload di un file sul server	120
10.4 Esercizio web server esteso con gestione CGI	126
10.5 Esercizi sulla WebSocket Chat	129
10.5.1 Esercizio 1	129
10.5.2 Esercizio 2	131
10.5.3 Esercizio 3	133
10.6 Esercizi su Webservice basati su REST	134
10.7 Esercizi su Wireshark	147
10.7.1 Esercizio 1 - File capture.cap	147
10.7.2 Esercizio 2 - File simpleHTTP.cap	151
10.7.3 Esercizio 3 - File busyNetwork.cap	156
10.7.4 Esercizio 4 - File pingCapture.cap	157
10.7.5 Esercizio 5 - Comando traceroute	158
10.7.6 Esercizio 6 - Interfacce di rete	159
10.8 Esercizi Docker	160
10.9 Esercizi MQTT	162
10.9.1 Domande sull'esempio Pub/Sub	162

Prefazione

Questa dispensa è stata scritta mettendo assieme le nozioni prese dalle: lezioni seguite in presenza (A.A. 2022/2023), videolezioni caricate sulla piattaforma Moodle e dal testo di riferimento consigliato per il corso.

L'utilizzo di questo materiale non deve sostituire la frequentazione del corso, ma solo supportarlo a fine didattico.

È possibile condividere gratuitamente questa dispensa. Inoltre si ricorda che tutti i materiali sono sempre disponibili al canale Telegram <https://t.me/univrinfo>.

Crediti

Professore del corso (A.A. 2022/2023): Davide Quaglia.

Testi di riferimento consigliati:

Reti di calcolatori ~ bottom-up (Andrew S. Tanenbaum, Nick Feamster, David J. Wetherall)
Reti di calcolatori e internet ~ top-down (James F. Kurose, Keith W. Ross).

Programmazione di rete con interfaccia socket

1.1 Host, processo e applicazione

Definizione 1.1.1: Host

È un calcolatore identificato da un indirizzo IP a cui può essere opzionalmente associato un nome *Internet*.

Definizione 1.1.2: Processo

È un programma in esecuzione sull'*host* che trasmette/riceve pacchetti verso/da altri processi su altri *host* attraverso la rete.

Il processo si identifica con un numero di porta appartenente all'intervallo da 0 fino 65535.

Definizione 1.1.3: Applicazione

È la collaborazione tra un insieme di processi sparsi sulla rete per fare qualcosa di utile per l'utente, esempi: *web*, *chat*, *email*, ecc.

Esempio

- Telegram è l'applicazione;
- APP Telegram è il processo che gira sullo *smartphone* che funge da *host*;
- Telegram *server* è il processo in esecuzione sulla macchina remota (anch'essa è *host*).

1.1.1 Scoprire l'indirizzo IP degli host

L'indirizzo IP all'interno di un calcolatore è ottenibile via terminale tramite il comando `ifconfig -a` su sistemi operativi Unix e Mac-OS, mentre su Windows il comando diventa `ipconfig /all`.

Di base entrambi i comandi forniscono informazioni sulla configurazione dell'interfaccia di rete:

- **Indirizzo IP** (IPv4 e IPv6): identificatore numerico assegnato a ciascun dispositivo collegato a una rete.
- **Indirizzo MAC**: è un identificatore univoco assegnato a ciascuna scheda di rete o dispositivo di rete.
- **Subnet Mask**: è un valore numerico che definisce la proporzione di un indirizzo IP che rappresenta la rete e la porzione che rappresenta il dispositivo all'interno di quella rete.
- **Default Gateway**: è l'indirizzo IP del *router* che viene utilizzato da un dispositivo per instradare il traffico verso destinazioni esterne alla sua rete locale.

1.1.2 Applicazioni orientate al datagram e alla connessione

La trasmissione dei dati in *Internet* non avviene spedendo un *bit* o un *byte* per volta, poiché non è efficiente. La migliore modalità di trasmissione è quella di inviare sequenze di *byte* aventi il nome di: pacchetto, *Protocol Data Unit* (PDU) oppure *datagram*.

Applicazioni orientate al datagram

Ogni pacchetto scambiato tra gli *host* è indipendente dai precedenti e dai successivi. Tant'è che le perdite non vengono tenute in considerazione.

Esempio

Si considera la trasmissione di temperature in una stanza. Il trasmettitore invia periodicamente al *display* del ricevitore le temperature ricavate mediante un sensore. Questi valori sono tra loro tutti indipendenti e la perdita di uno di questi non comporta alcun interesse da parte del ricevitore: semplicemente il pacchetto viene perso e si attende il prossimo.

Applicazioni orientate alla connessione

Tra i pacchetti trasmessi esiste una relazione, ovvero tutti assieme formano un'informazione più grande, come può essere per esempio un'immagine. Il messaggio più "grande" deve essere incluso in un'entità logica chiamata **connessione**.

È compito del sistema operativo enumerare in sequenza i pacchetti e rilevare eventuali perdite per richiedere in seguito la ritrasmissione.

- Vantaggi: l'utente può scrivere e/o leggere su un qualunque archivio remoto con la stessa naturalezza di quando scrive e/o legge su un archivio locale.
- Svantaggi: gli *host* trasmettitori e ricevitori devono “lavorare” maggiormente dentro il sistema operativo. Inoltre può causarsi un maggiore ritardo di trasmissione nel caso di ritrasmissione di pacchetti persi.

1.2 Modello Client e Server

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:

- *client/server*;
- *publisher/subscribers* (pub/sub).

Nel **modello client/server**, il *client* effettua sempre il primo passo con una richiesta verso il *server*. Quest'ultimo esegue il secondo passo rispondendo e successivamente rimane in attesa di altre richieste.

Ciò che determina il ruolo del *client* e del *server* è l'ordine dei messaggi inviati e non il loro contenuto.

Esempio

Il sensore di temperatura corporea funge da *client* e manda al *server* una temperatura. Il *server* risponde al *client* con il messaggio “OK”.

È importante fare attenzione su diversi punti di questo modello:

- *client* e *server* sono **processi** e non *host*;
- l'insieme di almeno un *client* e un *server* costituisce l'applicazione di rete.
 - All'interno di un'applicazione un certo processo gioca il ruolo di *client* o *server*.
- In un'applicazione *client/server*, il *client* è sempre quello che compie il primo passo indipendentemente che chieda o trasmetta un dato.

1.2.1 Creazione dell'interfaccia Socket (fase 1)

Il programma prima di usare la rete crea l'oggetto di tipo **socket**, il quale viene identificato da 3 parametri:

- indirizzo IP locale;

6 CAPITOLO 1. PROGRAMMAZIONE DI RETE CON INTERFACCIA SOCKET

- porta locale;
- modalità di trasmissione: UDP oppure TCP.

La **porta** è un meccanismo che consente a un singolo *host* di gestire più flussi di dati in ingresso o in uscita contemporaneamente. Nella pratica le porte vanno da 0 fino a 65535 (16 bit senza segno). Il programma *server* deve decidere esplicitamente il numero di porta locale affinché i *client* possano conoscerlo.

- Le porte da 0 a 1023 sono riservate a protocolli applicativi noti: 80 HTTP, 443 HTTPS, ecc. (per usarle occorre possedere i privilegi di *root*).
- Le porte dal 1024 fino a 65535 sono usufruibili senza richiedere i privilegi di *root*.

1.2.2 Trasmissione e ricezione (fase 2)

In questa fase serve conoscere 2 ulteriori parametri:

- l'indirizzo IP della destinazione;
- la porta della destinazione.

Al *client* basta conoscere il nome *Internet* del *server*, perché il sistema operativo ricava in automatico il suo indirizzo IP (risoluzione del nome) mediante il DNS.

Diversamente nel lato *server* l'indirizzo IP è contenuto nelle informazioni di mittente della richiesta, di conseguenza non ha bisogno di richiederlo.

- Nel caso dell'UDP:
 - Chiamata a funzione per trasmettere.
 - Chiamata a funzione per ricevere.
- Nel caso del TCP:
 - Chiamata a funzione per aprire la connessione.
 - Chiamata a funzione per trasmettere.
 - Chiamata a funzione per ricevere.
 - Chiamata a funzione per chiudere la connessione.

Nome localhost oppure 127.0.0.1

Questo indirizzo speciale è utilizzato per scopi particolari in cui il *client* e *server* potrebbero girare sulla stessa macchina. Non serve una vera rete fisica, bensì si utilizza una rete fittizia detta **loopback**. In tal caso l'interfaccia *socket* sia del *client* e sia del *server* hanno indirizzo IP 127.0.0.1 e nome *localhost*.

Si può utilizzare questo indirizzo per testare il *client* e *server* sul PC prima di distribuire il prodotto sulla rete.

Scoprire la porta dell'interlocutore

Al lato *client* la porta del *server* deve essere nota:

- chi programma il *server* rende nota la porta da utilizzare;
- si può ricavare dal tipo di protocollo impiegato.

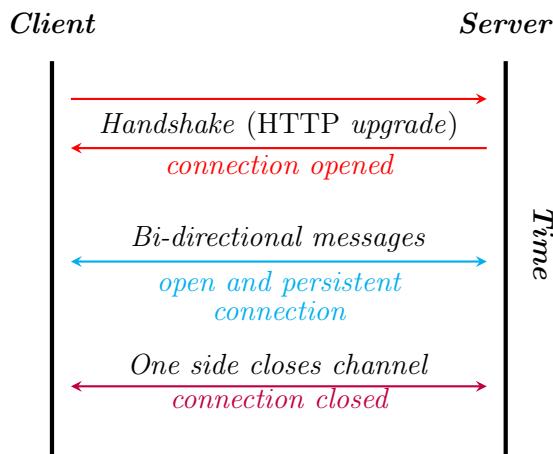
Nel lato *server* la porta è già nota, poiché contenuta nelle informazioni del mittente della richiesta.

Approccio ai WebSocket

2.1 Introduzione ai 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.

Permette una maggiore interazione tra un *browser* e un *server*, facilitando la realizzazione di applicazioni *web* che devono fornire contenuti in **tempo reale**. Questo è possibile grazie al permesso rilasciato al *server* di effettuare *push* autonomi di dati verso il *browser* per aggiornarlo. Ciò non può avvenire con il tradizionale HTTP.



I **sistemi bidirezionali** (*full-duplex*) permettono la comunicazione in entrambe le direzioni simultaneamente. Nel protocollo HTTP non è presente questa possibilità: è il lato *client* a fare il primo passo.

Con i *WebSocket* torna la connessione TCP di base, ma con la bidirezionalità e con HTTP che la instaura all'inizio (perciò è compatibile con *firewall* presenti su *Internet*). Quindi i *WebSocket* sono basati su TCP e nascono da una connessione HTTP attraverso una **UPGRADE REQUEST** verso il *server*.

- HTTP usa **UPGRADE HEADER** in cui viene chiesto di passare dal protocollo HTTP a quello *WebSocket*.

- HTTP fornisce un meccanismo speciale che permette di stabilire un *upgrade* di protocollo da usare durante la connessione. Questo meccanismo può essere inizializzato solo dal *client*. Il *server* decide se accettare o meno il servizio stabilito dal *client*.

I *WebSocket* usano le porte 80 oppure 443 e sono compatibili con le impostazioni di sicurezza dei *firewall*, *proxy web* e meccanismo NAT. A differenza del traffico HTTP, che usa un protocollo di richiesta/risposta, le connessioni *WebSocket* possono rimanere aperte per un lungo periodo e permettono lo scambio paritetico di dati tra *browser* e *server*. Infatti i *firewall*, NAT e *proxy* interposti devono consentire questo tipo di comunicazioni.

2.1.1 Limitazioni dei WebSocket

HTTP è utile per inviare e chiudere connessioni per trasferimenti di dati di tipo *one-time*, come i caricamenti iniziali. Inoltre è più efficiente rispetto il *WebSocket* per questo tipo di richieste, cioè chiudendo le connessioni una volta usate. I *WebSocket* devono avere JavaScript abilitato e ciò non è agevole per i sistemi *embedded* e inoltre potrebbero richiedere più risorse nell'architettura di rete, poiché è una connessione persistente.

2.1.2 Introduzione Node.js

Node.js è un *framework event-driven* per realizzare *app web* in JavaScript. Tipicamente è utilizzato nel *client-side*, anche per la scrittura di *server-side*. La piattaforma è basata sul JavaScript Engine V8, che è il *runtime* di Google utilizzato anche da Chrome e disponibile sulle principali piattaforme.

2.1.3 Approccio asincrono

La principale caratteristica di Node.js è quella di accedere alle risorse del sistema operativo in modalità *event-driven* e non sfruttando il classico modello basato su processi e *thread* concorrenti.

Il **modello *event-driven*** (oppure **programmazione a eventi**) si basa sul concetto d'esecuzione di un'azione nel momento in cui accade qualcosa. Dunque ogni azione risulta essere asincrona e garantisce una certa efficienza delle *app* grazie a un sistema di *callback* gestito a basso livello al “*runtime engine*”. Questo durante le attese può occuparsi di altro, per esempio aver a che fare con la logica applicativa.

2.2 Descrizione esempio della chat

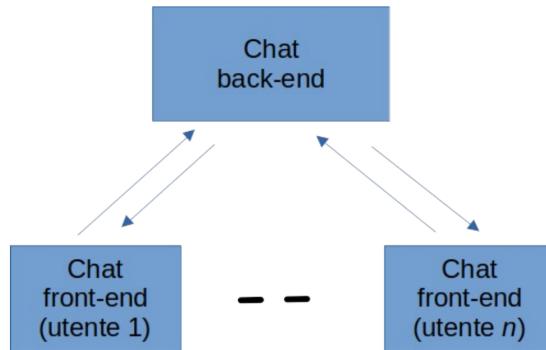
- **JavaScript**: linguaggi di *scripting* orientato agli oggetti ed eventi, generalmente utilizzato nella programmazione *web* lato *client* per la creazione di siti e applicazioni *web*.
- **HTML**: linguaggio di Markup, nato per la formattazione e impaginazione dei documenti ipertestuali.

- **CSS:** linguaggio che definisce la formattazione di documenti HTML, XHTML e XML.
- **Console con ispezione Network:** monitoraggio da parte del *browser* dello scambio di pacchetti.



L'esempio di *chat* multiutente è sviluppata nella seguente maniera:

- 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 con il nome della persona che lo ha inviato.



2.2.1 Introduzione a Express

Express è una libreria di Node.js che permette di costruire molto facilmente applicazioni *web*.

- La variabile `var express = require('express');` crea una variabile che necessita di Express. Successivamente questa variabile viene utilizzata per creare il *server web* in ascolto sulla porta 4000.

2.2.2 Creazione WebSocket

Socket.IO è una libreria in JavaScript utilizzata per implementare il protocollo *WebSocket* e racchiude numerose funzioni, incluse: *broadcasting* a tutti i *socket* collegati, salvataggio dei dati riguardanti ciascun utente e l'approccio asincrono I/O.

Listing 2.1: Back-end server

```
1 var express = require('express');
2 var socket = require('socket.io');
```

```

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

```

2.2.3 Funzionamento del back-end

Alla riga 10 viene creato il *socket* HTTP. Dopo la creazione del *socket*, il *server* si mette in attesa delle connessioni HTTP da parte dei *client*. Quando viene aperta la connessione dal *browser* viene effettuato l'*upgrade* a *WebSocket*. Successivamente il *server* esegue la funzione “*on connection*” e si mette in ascolto del successivo evento che richiederà il suo intervento, ovvero quello di ricevere un messaggio con il *tag* “*message*”, che farà sì che il *server* inoltri il messaggio a tutti i *client* connessi mediante l’oggetto “d’insieme” chiamato “*sockets*”.

2.2.4 Funzionamento del front-end

Il codice HTML, che crea l'interfaccia utente nella quale si potrà visualizzare la *chat*, viene utilizzato da JavaScript per estrarre il contenuto di alcuni *tag*, in cui vengono inserite le informazioni che devono essere inviate al server e in cui vengono visualizzati i messaggi provenienti dagli altri utenti. Il *tag* `<script>` contiene invece il *link* del *file .js* da eseguire all'apertura del *file HTML*; esso rappresenta il codice di implementazione della *chat* lato *client* (vedere sezione successiva). Oltre alla struttura generale di HTML5, si veda in particolare l'uso che JavaScript fa degli identificatori dei *tag*.

Listing 2.2: Codice 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
7              /2.1.0/socket.io.dev.js"></script>
8          <link href="/styles.css" rel="stylesheet" />
9      </head>
10     <body>
11         <div id="mario-chat">
12             <h2>Chat</h2>
13             <div id="chat-window">
14                 <div id="output"></div>
15                 <div id="feedback"></div>
16             </div>
17             <h4 id="sender" style="padding-left: 20px">Handle</h4>
18             <input id="message" type="text" placeholder="Message" />
19             <button id="send">Send</button>
20         </div>
21     </body>
22     <script src="/chat.js"></script>
23 </html>

```

2.2.5 Implementazione del front-end della chat

Una volta che la *chat* è stata aperta sul *browser* e si ha effettuato la registrazione col nome utente, alla riga 15 viene usata la funzione `innerHTML`. HTML per modificare il contenuto della pagina con il nome con la quale si è effettuata la registrazione. A questo punto alla riga 19 viene creato il *socket* che deve connettersi con il *server*. Alla riga 22 viene aggiunto un *listener* all'evento “*click*” del bottone che, quando verrà premuto, invierà al *server* il proprio nome utente e il contenuto del testo del messaggio, così da poterlo inoltrare agli altri utenti. Importante è notare che viene passata la stringa “`message`” in modo che il *server* sappia come gestire questo evento.

Listing 2.3: Front-end

```

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

```

2.2.6 Esecuzione della parte front-end e back-end

Il *server web* utilizzato è Node.js che permette di eseguire codice JavaScript a lato *server* per creare il *back-end*. Il *front-end* è realizzato mediante codice JavaScript eseguito dentro il *browser*.

Analisi di rete con Wireshark e da linea di comando

3.1 Introduzione agli analizzatori di rete

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

- **TCPDUMP**: *tool* da linea di comando (per OS Linux);
- **WinDump**: *tool* da linea di comando (per OS Windows);
- **Wireshark**: *tool* con GUI disponibile per Linux, Windows e Mac.

Tutti questi *software* si basano sulla libreria C **libpcap**. Le principali funzionalità di questa libreria sono la possibilità di cercare e trovare interfacce di rete, gestire in modo avanzato i filtri di cattura e gestire gli errori e statistiche di cattura.

3.1.1 Concetti chiave dello sniffing

- *Sniffing all'interno di reti non-switched*: in questa tipologia di reti il mezzo trasmissivo è condiviso, di conseguenza tutte le schede di rete dei PC ricevono tutti i pacchetti, anche quelli destinati ad altri. I propri sono invece selezionati a seconda dell'indirizzo MAC (indirizzo *hardware* specifico della scheda di rete).

In tal caso lo *sniffing* consiste nell'impostare sull'interfaccia di rete la **modalità promiscua** che disattiva il “filtro *hardware*” basato sul MAC (uno dei principali motivi per cui occorrono i privilegi `sudo`). Così facendo si permette al sistema l'ascolto di tutto il traffico passante sul cavo. Un esempio di rete *non-switched* è la rete WiFi.

- *Sniffing all'interno di reti Ethernet switched*: in questo caso l'apparato centrale di rete (detto *switch*) si preoccupa di inoltrare su ciascuna porta solo il traffico destinato ai dispositivi collegati a quella porta. Quindi ciascuna interfaccia di rete riceve solo i pacchetti destinati al proprio indirizzo, i pacchetti *multicast* e quelli *broadcast*.

L'impostazione della modalità promiscua è insufficiente per poter intercettare il traffico in una rete gestita da *switch*.

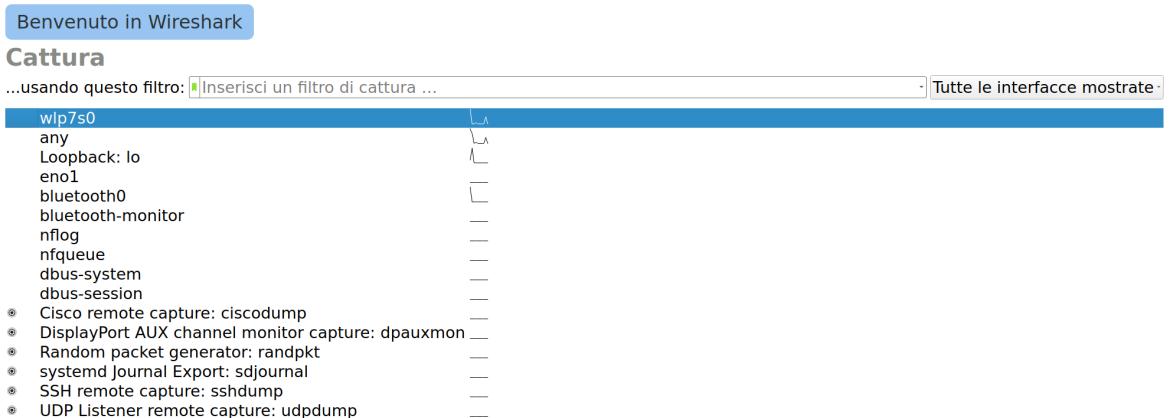
3.2 Utilizzo di Wireshark

- I **dati** possono essere acquisiti direttamente dall'interfaccia di rete (reti Ethernet, WiFi, ADSL, ecc.), oppure possono essere letti su un *file* di cattura precedente.
- I **dati di rete** possono essere esplorati nelle loro parti tramite un'interfaccia grafica.
- I **filtri di visualizzazione** possono essere usati per colorare o visualizzare le informazioni sommarie sui pacchetti.
- I **protocolli di comunicazione** possono essere scomposti dato che Wireshark riesce a comprenderne la struttura. È dunque possibile visualizzare encapsulamenti, campi singoli e interpretare il loro significato.
- È possibile studiare le statistiche di una connessione TCP e di estrarne il contenuto.

Per poter utilizzare le funzionalità di cattura diretta in ambiente Linux bisogna essere autenticati come utente *root*, oppure, aver installato il *tool* con **setuid** a *root*.

3.2.1 Cattura dei pacchetti

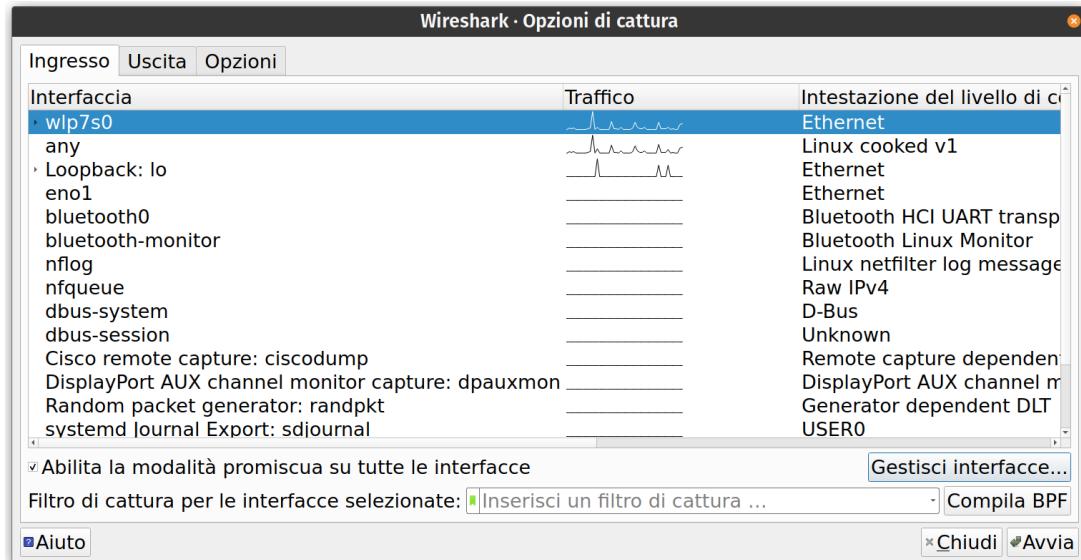
Per avviare la **cattura dei pacchetti** è necessario specificare da quale interfaccia si vuole effettuare la cattura. Per fare ciò basta avviare una delle varie interfacce presentate nella schermata di apertura:



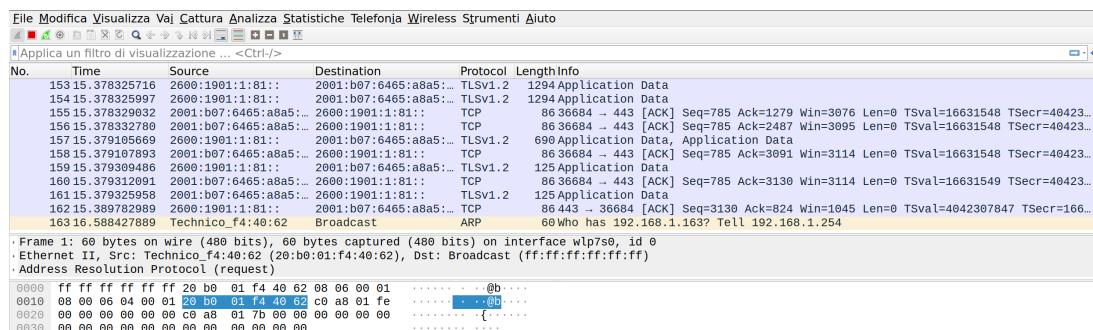
3.2. UTILIZZO DI WIRESHARK

17

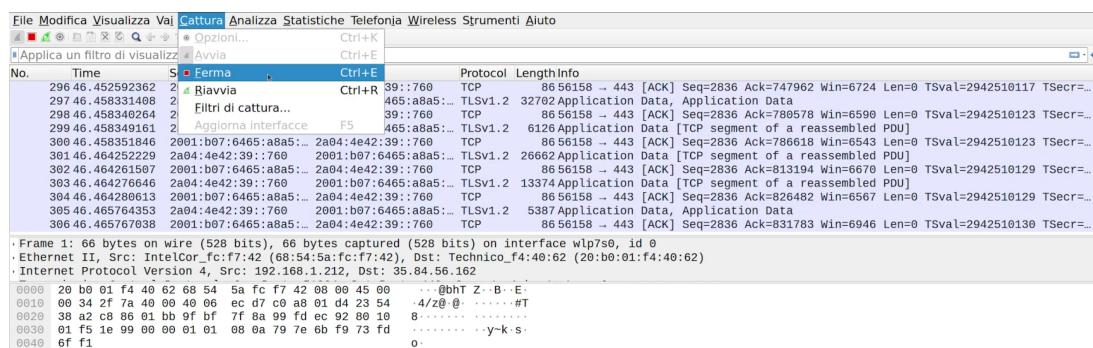
Se invece si vogliono applicare varie opzioni alle interfacce, allora passa attraverso le voci **Cattura/Opzioni** presenti nel menù:



Da quest'ultima schermata mostrata, premendo sul pulsante "Avvia" si inizia la cattura:



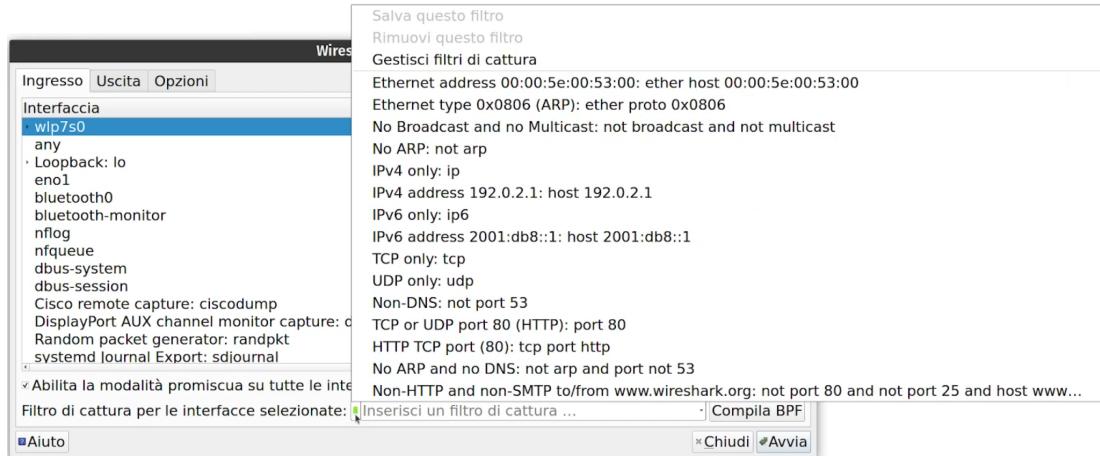
Invece per terminare la cattura (se non si è impostato un limite in pacchetti, secondi o byte) si clicca sul pulsante “Ferma” dal menù Cattura (oppure quello già presente in alto a sinistra):



Sui sistemi Linux i nomi delle interfacce di rete potrebbero essere nel nuovo formato dove, `eth` è sostituito da `enp` e `wlan` è sostituito da `wlp`, ecc.

3.2.2 Applicazione dei filtri nella cattura

È possibile limitare la cattura ai soli pacchetti che rispettano specifici requisiti imponendo un filtro di cattura nella finestra **Opzioni di cattura** e premendo sul tasto che raffigura un segnalibro verde oppure scrivendolo nella *textbox* a fianco:



I **filtri di cattura** programmano la scheda di rete con l'intento di catturare solo determinati pacchetti. Questi sono solitamente utilizzati quando la quantità di pacchetti che passano sul tratto di rete osservato è tale per cui se tutti i pacchetti venissero passati alla CPU, allora le sue prestazioni sarebbero compromesse.

Quindi è possibile scrivere l'espressione del filtro manualmente, oppure selezionarne uno preesistente cliccando sull'apposito bottone e al limite modificarlo.

Al termine della cattura la finestra principale di Wireshark mostra i dati catturati.

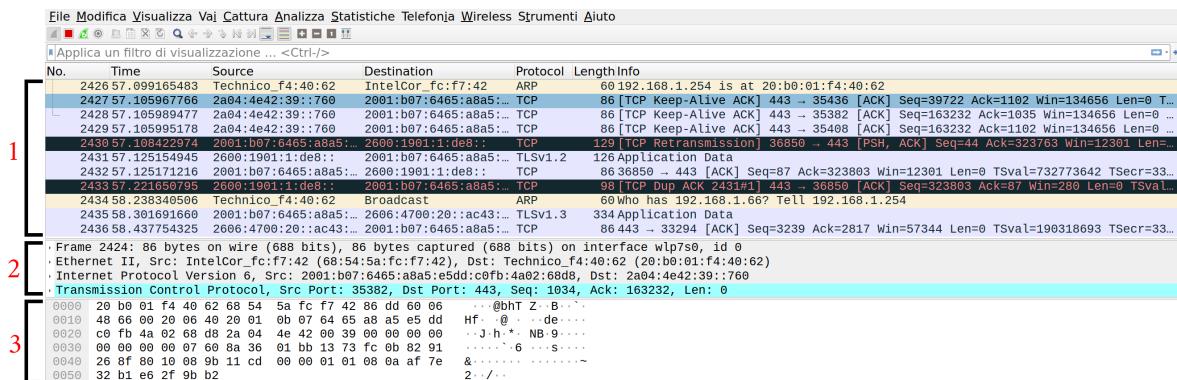
3.2.3 Finestra principale

La finestra principale è suddivisa in tre parti, nel seguente ordine:

1. Tabella dei pacchetti catturati.
2. Vista sulla encapsulazione dei protocolli del pacchetto selezionato nella tabella.
3. Vista in versione binaria dei dati del pacchetto selezionato nella tabella.

3.2. UTILIZZO DI WIRESHARK

19



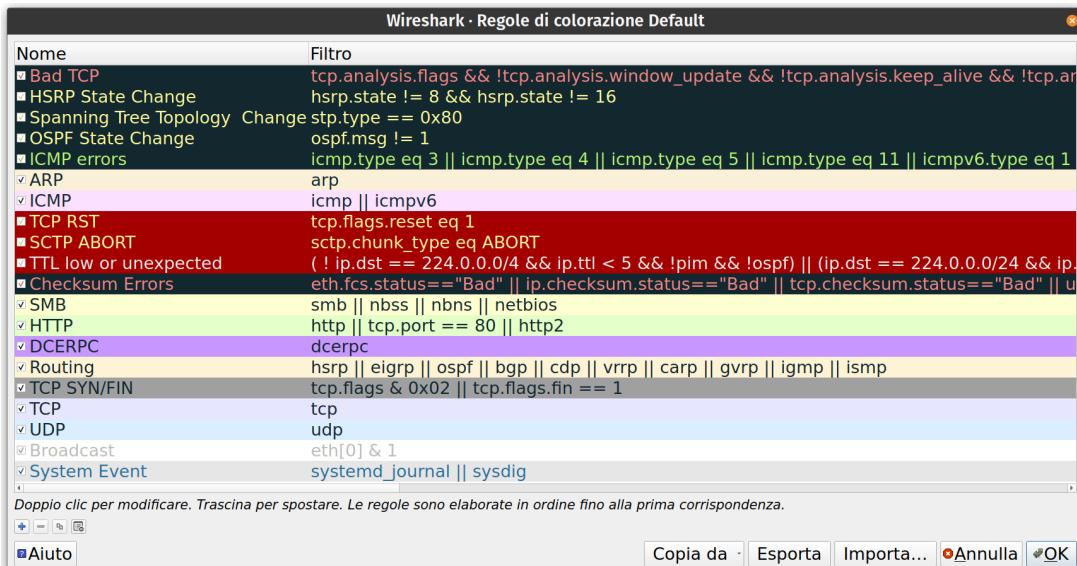
Inoltre è possibile visualizzare un sommario scegliendo il menuù **Statistiche/Proprietà file di cattura**.

3.2.4 Regole di colorazione

È possibile migliorare la visualizzazione dei vari pacchetti nella tabella principale colorando le righe in base al tipo di protocollo, oppure di indirizzi coinvolti. I filtri di coloramento possono essere impostati nel menuù **Visualizza/Regole di colorazione**.

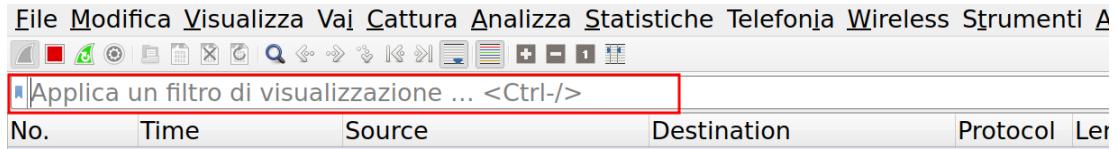
È possibile creare nuovi filtri di colori attraverso il pulsante “+” presente in basso a sinistra della finestra **Regole di colorazione** precedentemente aperta. Le regole sono espressioni booleane sui campi del pacchetto; si impostano con un linguaggio diverso da quello dei filtri di cattura.

Per ciascun pacchetto da visualizzare, le regole di colorazione sono considerate dal programma “dall’alto verso il basso”. Quando una regola è soddisfatta, il pacchetto viene visualizzato con i colori corrispondenti. Se tutte le regole vengono passate in rassegna e nessuna è vera, il pacchetto viene visualizzato in nero su bianco.



3.2.5 Applicazione dei filtri di visualizzazione nella finestra principale

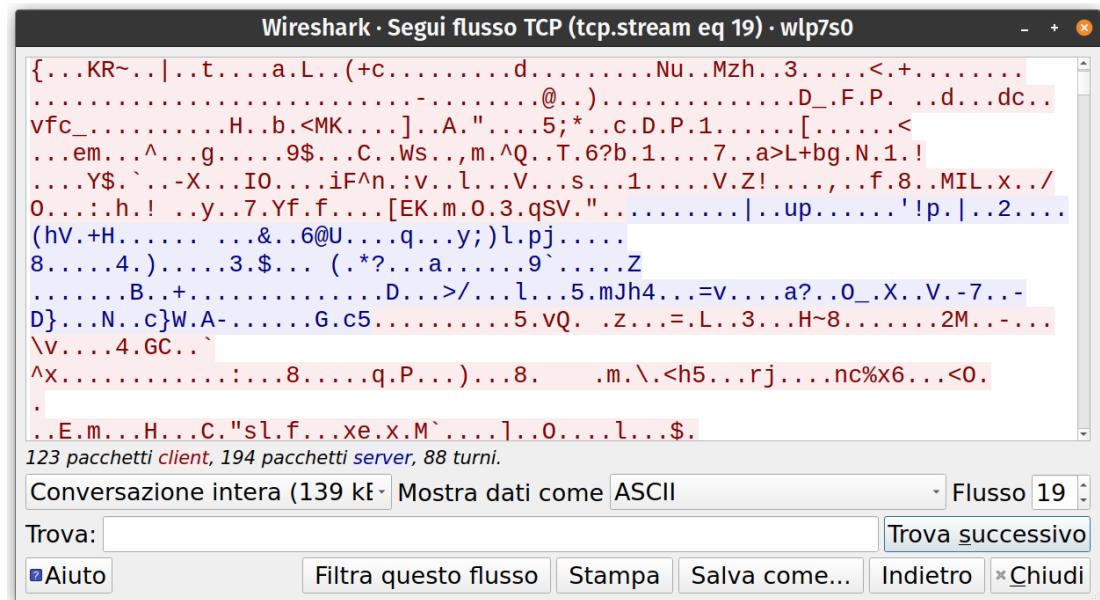
È possibile creare un filtro per limitare il numero di pacchetti visualizzati in una cattura già avvenuta. Per fare questo si utilizza la barra dei filtri presente nella schermata principale:



Attraverso il tasto raffigurato dal segnalibro azzurro è possibile specificare un filtro esistente, oppure crearne di nuovi.

3.2.6 Analisi del flusso TCP

Per quanto riguarda TCP, selezionando un pacchetto TPC nella finestra principale, è possibile seguire l'intero flusso dati di quella “conversazione” mediante la voce **Analizza/Segui/Flusso TCP**:

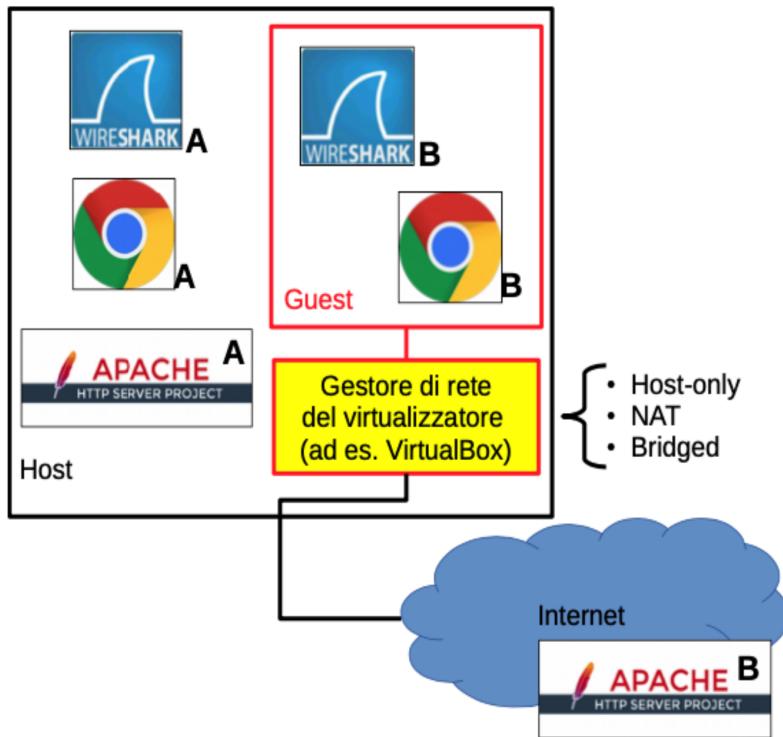


3.2.7 Visualizzazione del livello Data-link delle PDU non Ethernet

Mentre il *driver* della scheda Ethernet fornisce a Wireshark l'esatto *header* della PDU di livello *Data-link*, alcune altre interfacce di rete non-Ethernet (come il WiFi) potrebbero non farlo a meno dell'utilizzo di *plugin* specifici per Wireshark (o per la scheda di rete). Nel caso in cui tale *header* non venga fornito, Wireshark visualizzerà un *header* Ethernet “autocostruito”, oppure, un *header* con la dicitura `linux cooked capture`.

3.2.8 Cattura di traffico di rete all'interno di una macchina virtuale

Particolarmente interessante è la cattura del traffico di rete all'interno di una macchina virtuale. La figura seguente mostra un sistema *Host* che ospita un sistema *Guest* attraverso il meccanismo della virtualizzazione.



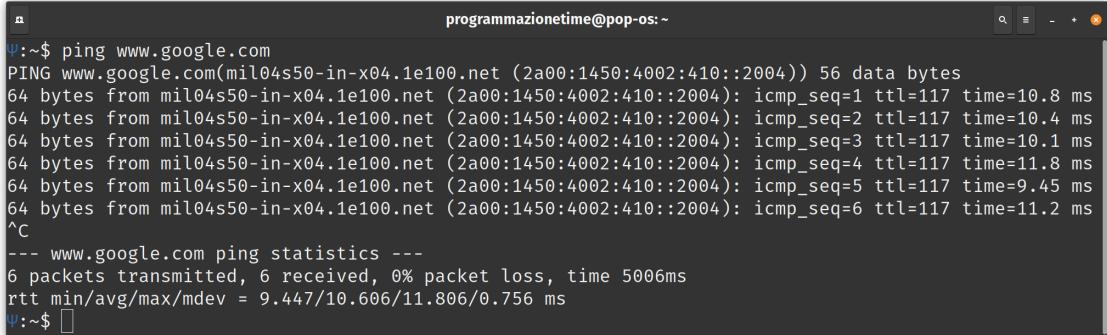
Il traffico di rete catturabile dall'istanza *B* di Wireshark nel sistema *Guest* dipende dalla configurazione del gestore di rete del virtualizzatore che prevede le seguenti tre alternative principali:

1. **Host-only:** l'unico traffico di rete possibile catturabile nel *Guest* è quello tra *Guest* e *Host* (per esempio l'istanza *B* del *browser* comunica con l'istanza *A* di *HTTP server* sull'*Host*).
2. **NAT:** l'unico traffico di rete nel *Guest* è quello verso l'esterno (per esempio tra l'istanza *B* dei *browser* e l'istanza *B* di *HTTP server* su *Internet*).
3. **Bridged:** il gestore di rete del virtualizzatore simula il comportamento di uno *switch* Ethernet a cui sono collegati sia l'*Host* sia il *Guest*. In questo scenario, l'istanza *B* di Wireshark può catturare sia il traffico legato al *Guest* sia quello legato all'*Host* di tipo *broadcast*.

3.3 Comandi di rete

3.3.1 Comando ping

Il comando **ping** è un semplice strumento per verificare la raggiungibilità di un *computer* connesso alla rete e il relativo *Round Trip Time* (RTT), ossia il tempo che intercorre dalla partenza del pacchetto inviato al ritorno della risposta. Per questa operazione viene utilizzato il protocollo ICMP¹.

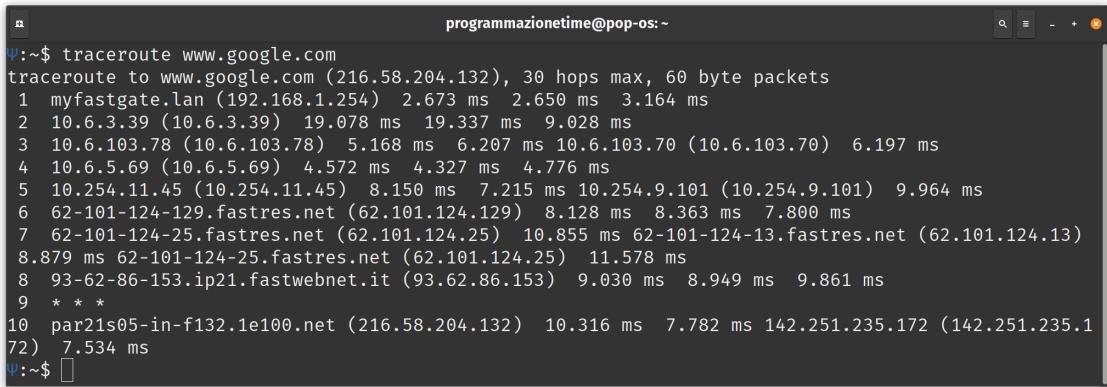


```
programmazionetime@pop-os: ~
ψ:~$ ping www.google.com
PING www.google.com(mil04s50-in-x04.1e100.net (2a00:1450:4002:410::2004)) 56 data bytes
64 bytes from mil04s50-in-x04.1e100.net (2a00:1450:4002:410::2004): icmp_seq=1 ttl=117 time=10.8 ms
64 bytes from mil04s50-in-x04.1e100.net (2a00:1450:4002:410::2004): icmp_seq=2 ttl=117 time=10.4 ms
64 bytes from mil04s50-in-x04.1e100.net (2a00:1450:4002:410::2004): icmp_seq=3 ttl=117 time=10.1 ms
64 bytes from mil04s50-in-x04.1e100.net (2a00:1450:4002:410::2004): icmp_seq=4 ttl=117 time=11.8 ms
64 bytes from mil04s50-in-x04.1e100.net (2a00:1450:4002:410::2004): icmp_seq=5 ttl=117 time=9.45 ms
64 bytes from mil04s50-in-x04.1e100.net (2a00:1450:4002:410::2004): icmp_seq=6 ttl=117 time=11.2 ms
^C
--- www.google.com ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5006ms
rtt min/avg/max/mdev = 9.447/10.606/11.806/0.756 ms
ψ:~$
```

Dopo aver avviato il comando **ping** da terminale è possibile visualizzare lo scambio dei messaggi tra il calcolatore e la destinazione. Per fare ciò è sufficiente avviare una cattura da Wireshark applicando il filtro **icmp** per la visualizzazione.

3.3.2 Comando traceroute

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.



```
programmazionetime@pop-os: ~
ψ:~$ traceroute www.google.com
traceroute to www.google.com (216.58.204.132), 30 hops max, 60 byte packets
1 myfastgate.lan (192.168.1.254) 2.673 ms 2.650 ms 3.164 ms
2 10.6.3.39 (10.6.3.39) 19.078 ms 19.337 ms 9.028 ms
3 10.6.103.78 (10.6.103.78) 5.168 ms 6.207 ms 10.6.103.70 (10.6.103.70) 6.197 ms
4 10.6.5.69 (10.6.5.69) 4.572 ms 4.327 ms 4.776 ms
5 10.254.11.45 (10.254.11.45) 8.150 ms 7.215 ms 10.254.9.101 (10.254.9.101) 9.964 ms
6 62-101-124-129.fastres.net (62.101.124.129) 8.128 ms 8.363 ms 7.800 ms
7 62-101-124-25.fastres.net (62.101.124.25) 10.855 ms 62-101-124-13.fastres.net (62.101.124.13)
8.879 ms 62-101-124-25.fastres.net (62.101.124.25) 11.578 ms
8 93-62-86-153.ip21.fastwebnet.it (93.62.86.153) 9.030 ms 8.949 ms 9.861 ms
9 * *
10 par21s05-in-f132.1e100.net (216.58.204.132) 10.316 ms 7.782 ms 142.251.235.172 (142.251.235.1
72) 7.534 ms
ψ:~$
```

¹ICMP (*Internet Control Message Protocol*) è un protocollo di servizio per trasmettere informazioni riguardanti malfunzionamenti, informazioni di controllo o messaggi tra i vari componenti di una rete di calcolatori.

Si noti la presenza di alcuni asterischi in corrispondenza di determinate tappe. Questi sono dovuti al fatto che, certe interfacce di specifici *router*, non forniscono alcuna informazione. Questa scelta viene presa dagli amministratori di rete per evitare di svelare la topologia di rete a possibili *malware*. In tal caso **traceroute** non può mostrare tali passi del percorso.

Dopo aver avviato il comando **traceroute** da terminale è possibile visualizzare in Wireshark lo scambio dei messaggi tra il calcolatore e la sorgente di destinazione. Per fare ciò è sufficiente avviare una cattura da Wireshark applicando il filtro **icmp** per la visualizzazione.

3.3.3 Comando nslookup

Il comando **nslookup** consente di effettuare una interrogazione ai *server DNS*² per poter ottenere da un *hostname* il relativo indirizzo IP, o viceversa. Si può utilizzare in due modalità: interattivo oppure non interattivo.

Modalità interattiva

Permette di effettuare più interrogazioni e visualizza i singoli risultati. Viene abilitata in maniera automatica quando il comando non è seguito da alcun argomento.

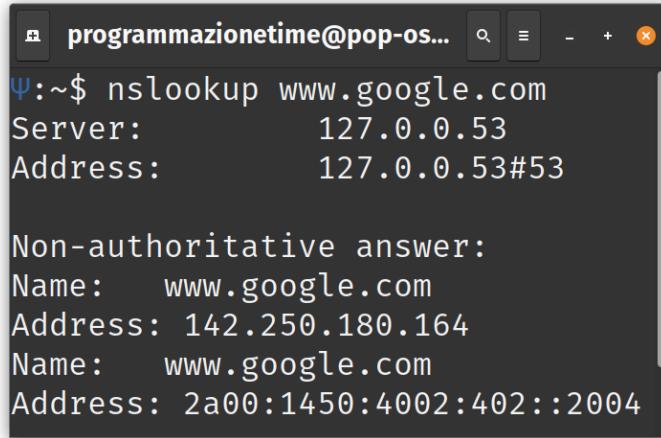
```
programmazionetime@pop-os: ~
Ψ:~$ nslookup
> www.google.com
;; communications error to 127.0.0.53#53: timed out
Server:          127.0.0.53
Address:         127.0.0.53#53

Non-authoritative answer:
Name:   www.google.com
Address: 142.250.180.164
Name:   www.google.com
Address: 2a00:1450:4002:402::2004
> 
```

Modalità non interattiva

Permette di effettuare una sola interrogazione visualizzandone il risultato. Abilitata ogni qualvolta si specifichi l'*host-to-find*.

²DNS (*Domain Name System*) è un sistema di server organizzato gerarchicamente, per la gestione del *namespace* (*Domain Name Space*). Il compito principale di questo servizio è quello di rispondere alle richieste della risoluzione del nome di dominio, ovvero la conversione dei nomi di dominio in indirizzi IP.

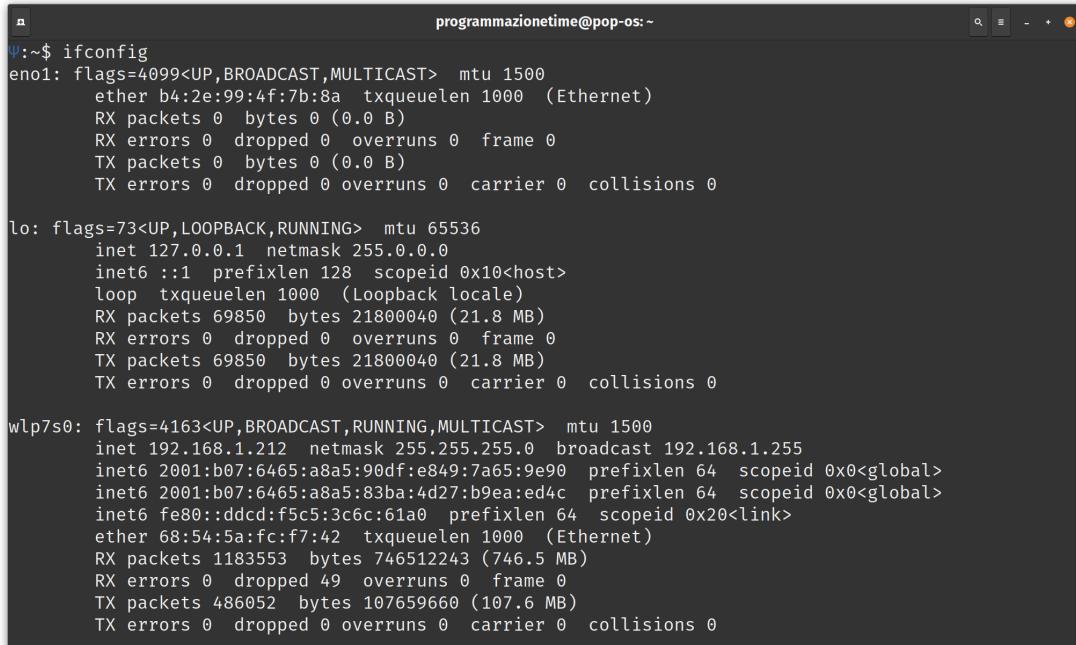


```
programmazionetime@pop-os... ~
Ψ:~$ nslookup www.google.com
Server:          127.0.0.53
Address:         127.0.0.53#53

Non-authoritative answer:
Name:   www.google.com
Address: 142.250.180.164
Name:   www.google.com
Address: 2a00:1450:4002:402::2004
```

3.3.4 Comando ifconfig

Il comando **ifconfig** (**ipconfig** in Windows) è un utilizzato per configurare e controllare un'interfaccia di rete TCP/IP da riga di comando. L'esecuzione del comando con l'opzione **-a** mostra a video le informazioni di tutte le interfacce di rete.



```
programmazionetime@pop-os: ~
Ψ:~$ ifconfig
eno1: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
      ether b4:2e:99:4f:7b:8a txqueuelen 1000  (Ethernet)
      RX packets 0 bytes 0 (0.0 B)
      RX errors 0 dropped 0 overruns 0 frame 0
      TX packets 0 bytes 0 (0.0 B)
      TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
      inet 127.0.0.1 netmask 255.0.0.0
      inet6 ::1 prefixlen 128 scopeid 0x10<host>
      loop txqueuelen 1000  (Loopback locale)
      RX packets 69850 bytes 21800040 (21.8 MB)
      RX errors 0 dropped 0 overruns 0 frame 0
      TX packets 69850 bytes 21800040 (21.8 MB)
      TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

wlp7s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
      inet 192.168.1.212 netmask 255.255.255.0 broadcast 192.168.1.255
      inet6 2001:b07:6465:a8a5:90df:e849:7a65:9e90 prefixlen 64 scopeid 0x0<global>
      inet6 2001:b07:6465:a8a5:83ba:4d27:b9ea:ed4c prefixlen 64 scopeid 0x0<global>
      inet6 fe80::ddcd:f5c5:3c6c:61a0 prefixlen 64 scopeid 0x20<link>
      ether 68:54:5a:fc:f7:42 txqueuelen 1000  (Ethernet)
      RX packets 1183553 bytes 746512243 (746.5 MB)
      RX errors 0 dropped 49 overruns 0 frame 0
      TX packets 486052 bytes 107659660 (107.6 MB)
      TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

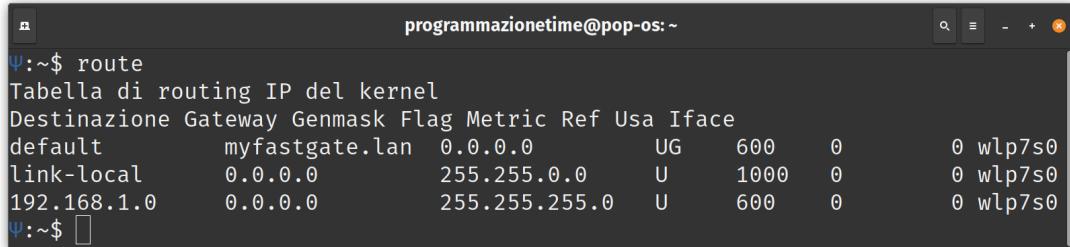
- **eth0** è l'interfaccia Ethernet (ulteriori interfacce, se presenti, sono nominate come **eth1**, **eth2**, ecc.).

- `lo` è l’interfaccia di *loopback*, la quale è sempre presente. È un’interfaccia di rete speciale che il sistema utilizza per comunicare con sé stesso.
- `wlan0` è il nome dell’interfaccia di rete *wireless* del sistema. Ulteriori interfacce *wireless* saranno denominate `wlan1`, `wlan2`, ecc.

Queste elencate sono la vecchia convenzione dei nomi per le interfacce di rete all’interno del sistema Linux (altri OS potrebbero avere differenti convenzioni).

3.3.5 Comando route

Il comando **route** (**route PRINT** su Windows) è utilizzato per visualizzare e modificare le tabelle di *routing*. L’esecuzione permette di visualizzare la tabella di routing dell’*host* come nell’esempio seguente:



```
ψ:~$ route
Tabella di routing IP del kernel
Destinazione Gateway Genmask Flag Metric Ref Usa Iface
default      myfastgate.lan  0.0.0.0      UG      600    0          0 wlp7s0
link-local   0.0.0.0       255.255.0.0   U        1000   0          0 wlp7s0
192.168.1.0  0.0.0.0       255.255.255.0  U        600    0          0 wlp7s0
ψ:~$
```

3.3.6 Comando whois

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

Whois si può consultare tradizionalmente da riga di comando, anche se ora esistono numerosi siti *web* che permettono di consultare gli archivi dove sono contenute tali informazioni.

26 CAPITOLO 3. ANALISI DI RETE CON WIRESHARK E DA LINEA DI COMANDO

```
programmazionetime@pop-os:~
$:~$ 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
```

```
programmazionetime@pop-os:~
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

$:~$ 
```

Dal Web ai Webservices

4.1 Protocolli HTTP/HTTPS

HTTP (*HyperText Transfer Protocol*) è un protocollo di comunicazione di base¹ utilizzato per il trasferimento dati tra un *client* e un *server*.

- È nato per la fruizione di contenuti di rete.
- Viene utilizzato anche per l'invocazione di funzionalità remote, ossia i *Webservice*.

HTTPS (*HyperText Transfer Protocol Secure*) è una versione sicura dell'HTTP: aggiunge una crittografia ai dati trasmessi tra il *client* e *server*. In genere utilizza il protocollo **TLS** (*Transport Layer Security*), oppure il suo predecessore **SSL** (*Secure Sockets Layer*), per stabilire una connessione crittografata. Di conseguenza i dati sono criptati e più sicuri da intercettazioni o attacchi durante il viaggio tra *client* e *server*.

Il *client* prende il nome di **web-browser** oppure **browser**, mentre il *server* viene detto **web-server**. La comunicazione avviene tramite il protocollo TCP sulla rete *Internet* e le fasi dei messaggi scambiati sono i seguenti:

1. apertura di una connessione TCP;
2. [HTTPS] autenticazione del *server* e negoziazione di una chiave di cifratura;
3. richiesta e risposta;
4. chiusura della connessione TCP.

I messaggi HTTPS che passano nella connessione TCP sono gli stessi dell'HTTP, ma sono sottoposti a:

- cifratura dei dati in transito;
- autenticazione del *server* mediante certificato digitale.

Il *server* nel caso dell'HTTPS lavora sulla porta 443 anziché 80 (dedicata all'HTTP).

¹Inventato da Tim Berners-Lee al CERN di Ginevra nel 1989.

Certificato digitale

Il **certificato digitale**, che si fa uso nell'HTTPS, consiste in un *file* contenente informazioni legate all'identità del proprietario del sito *web* e alle chiavi pubbliche e private utilizzate per crittografare e decrittografare i dati trasmessi durante le comunicazioni sicure. In genere gli elementi contenuti sono:

- **Chiave pubblica:** è utilizzata per crittografare i dati inviati al *server*.
- **Chiave privata:** è mantenuta segreta dal proprietario del certificato ed è impiegata per decrittografare i dati ricevuti.
- **Informazione di identità:** contiene informazioni sull'entità proprietaria del sito *web*, come il nome del dominio e l'organizzazione. Queste vengono poi verificate dall'autorità di certificazione (CA) durante l'emissione del certificato.
- **Firma digitale:** autentica l'origine del certificato e garantisce che le informazioni contenute non siano alterate.
- **Dettagli sulla validità:** include le date di validità, dopo la scadenza il certificato deve essere rinnovato.

4.1.1 Richiesta HTTP

È un messaggio inviato dal *client* al *server* per chiedere una risorsa o eseguire un'azione. La richiesta si suddivide in:

- **Linea di richiesta:** è la prima linea e specifica:
 - Metodo HTTP: indica l'azione da eseguire sul *server*. Per esempio GET fa richiedere al *client* una risorsa, POST permette di inviare dati, PUT serve per aggiornare una risorsa esistente e DELETE per rimuovere una risorsa.
 - URI (Uniform Resource Identifier): è l'indirizzo specifico della risorsa richiesta.
 - Versione HTTP: indica la versione del protocollo HTTP utilizzata per la richiesta.
- **Intestazione:** contiene informazioni aggiuntive:
 - User-Agent: indica quale *browser* e sistema operativo sta facendo la richiesta. Questo aiuta il *server* a fornire risposte ottimizzate.
 - Host: specifica il dominio/indirizzo IP del *server* di destinazione.
 - Accept: indica i tipi di contenuto che il *client* accetterà dalla risposta.
 - Authorization: è utilizzato per inviare credenziali e *token* di autenticazione al *server*.
 - Cookie: contiene eventuali dati dei *cookie* da inviare al *server*.

- **Corpo:** (se presente) possiede dati aggiuntivi, come possono essere i dati di un modulo per una richiesta POST.

Esempio

Di seguito è fornita una richiesta HTTP:

```
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
```

4.1.2 Risposta HTTP

È un messaggio inviato dal *server* al *client* in risposta alla richiesta HTTP. Anch'essa è composta da vari elementi:

- **Linea di stato:** è la prima linea della risposta ed è suddivisa in:
 - Versione HTTP: indica la versione del protocollo HTTP utilizzata, per esempio HTTP/1.1.
 - Codice di stato: è un codice numerico che rappresenta lo stato del risultato della richiesta. Per esempio 200 OK indica che la richiesta è stata completata con successo.
 - Descrizione di stato: è una breve descrizione testuale del codice di stato.
- **Intestazione:** contiene varie informazioni aggiuntive sulla risposta e può includere:
 - Content-Type: indica il tipo di contenuto restituito dal *server*.
 - Content-Length: specifica la lunghezza del corpo della risposta in *byte*.
 - Server: indica il nome e la versione del *server web* utilizzato.
 - Date: indica la data e l'ora in cui è stata generata la risposta.
 - Set-Cookie: viene utilizzata per inviare *cookie* al *client*.
 - Cache-Control: specifica le istruzioni sulla memorizzazione nella *cache* da parte del *client* o dei *proxy* intermedi.
- **Corpo:** possiede il contenuto effettivo della risposta, come il testo di una pagina *web* oppure dati JSON.

Esempio

```
Di seguito è fornita una risposta HTTP: 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

<html>
...
</html>
```

4.2 Linguaggi per le pagine web

4.2.1 Linguaggio HTML

L'**HTML** (*HyperText Markup Language*) è un linguaggio testuale utilizzato per creare e strutturare il contenuto delle pagine *web*. Le pagine HTML vengono visualizzate dal *browser web* e consentono di rappresentare testo, immagini, collegamenti ipertestuali e altri elementi multimediali in modo organizzato e interattivo.

Questo linguaggio prevede una serie di **tag** o **etichette** annidate (e contenenti eventuali attributi) che definiscono il contenuto di una pagina. Questi **tag** sono aperti e chiusi rispettivamente dalle parantesi angolari < > e alcuni possono appunto specificare ulteriori informazioni sulle loro proprietà mediante gli **attributi**.

Di seguito è fornito un breve codice HTML:

```
1 <!-- My document -->
2 <html>
3   <head>
4     <title>My Document</title>
5   </head>
6   <body>
7     <h1>Header</h1>
8     <p>
9       Testo <b>grassetto</b>
10    </p>
11    <p>
12      Testo <i>corsivo</i>
13    </p>
14  </body>
15 </html>
```

Tag per richiamare le immagini

Per le immagini si utilizza il tag `` e può avere i seguenti attributi:

- ***src*** (obbligatorio): specifica il percorso dell'immagine da visualizzare.
- ***alt*** (obbligatorio): fornisce un testo alternativo all'immagine che viene visualizzato quando non può essere caricata o è disabilitata.
- ***width***: specifica la larghezza dell'immagine in *pixel*.
- ***height***: specifica l'altezza dell'immagine in *pixel*.
- ***title***: fornisce un testo che appare quando l'utente tiene il cursore sopra l'immagine.
- ***style***: permette di applicare stili CSS direttamente all'elemento immagine.
- ***class***: assegna una classe CSS all'elemento immagine, che può essere utilizzata per applicare stili da un foglio di stile esterno.
- ***id***: fornisce un identificatore unico all'elemento immagine, utile per selezionarlo con JavaScript oppure CSS.

La lista degli attributi non obbligatori non è completa; sono stati riportati quelli più utili.

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

Tag per il collegamento ipertestuale

Il tag `<a>` è utilizzato per creare un collegamento ipertestuale (detto *link*) tra una pagina *web* e un'altra risorsa, come un'altra pagina *web*, un'immagine, un *file* scaricabile o qualsiasi altra risorsa. Questo tag può includere diversi attributi:

- ***href*** (obbligatorio): definisce l'URL della risorsa di destinazione a cui il *link* deve puntare.
- ***target***: definisce come la risorsa di destinazione deve essere visualizzata quando l'utente clicca sul *link*.

- **rel**: definisce la relazione tra la pagina corrente e la pagina di destinazione.
- **title**: fornisce un testo che appare quando l'utente posiziona il cursore sopra il *link*.

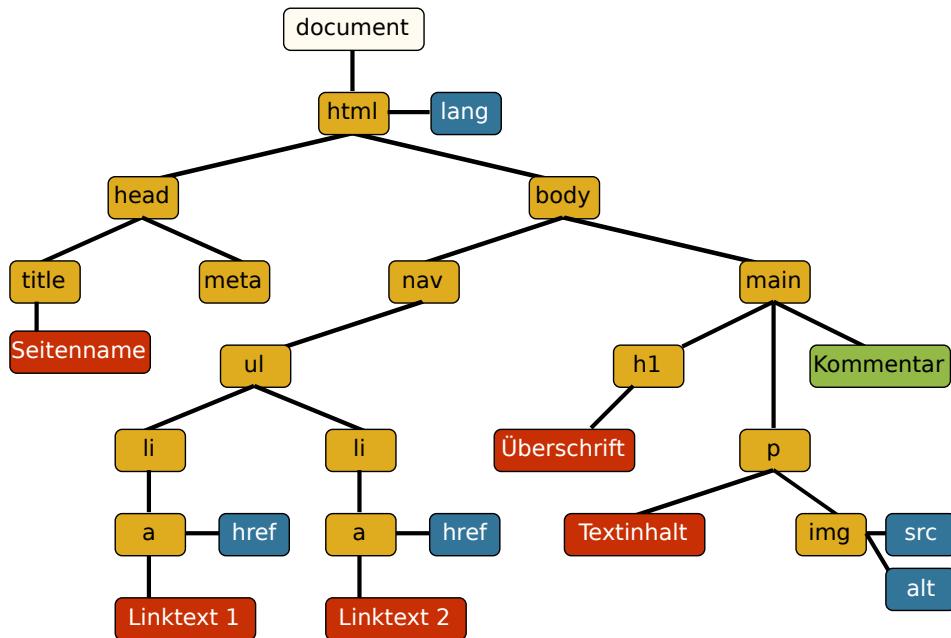
La lista degli attributi non obbligatori non è completa; sono stati riportati quelli più utili.

```

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

Document Object Model (DOM)

Il **Document Object Model** (DOM) è una rappresentazione gerarchica e struttura di un commento HTML o XML all'interno di un ambiente di programmazione, come un *browser web*. Il DOM consente di manipolare la struttura, il contenuto e lo stile di un documento *web* utilizzando linguaggi di *scripting* come JavaScript.



4.2.2 Linguaggio CSS

Il **CSS** (*Cascading Style Sheets*) è un linguaggio utilizzato per definire la presentazione e l'aspetto visivo di una pagina *web* scritta in HTML. In pratica permette di controllare il *layout*, i colori, i tipi di carattere e altri aspetti visivi dei contenuti su una pagina *web*. Questo separa la struttura dei contenuti (definita dall'HTML) dalla loro presentazione (gestita dal CSS). Di seguito viene fornito un codice HTML e CSS:

Codice HTML

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <link rel="stylesheet" href="styles.css">
5   </head>
6   <body>
7     <h1>Header</h1>
8     <p>Paragraph</p>
9   </body>
10 </html>
```

Codice CSS (styles.css)

```

1 body {
2   background-color: powderblue;
3 }
4
5 h1 {
6   color: blue;
7 }
8
9 p {
10   color: red;
11 }
```

Il tag `<link>` permette di importare un file `styles.css`.

4.2.3 Linguaggio JavaScript

Il **JavaScript** è un linguaggio di programmazione utilizzato nelle pagine HTML per aggiungere interattività, dinamicità e funzionalità avanzate. È possibile integrarlo direttamente nelle pagine HTML attraverso il tag `<script></script>`:

```

1 <script>
2 document.getElementById("demo").innerHTML = "Hello JavaScript!";
3 </script>
```

Altrimenti è possibile includere interi file di codice richiamati nell'HTML e poi scaricati dal browser:

```
1 <script src="myscripts.js"></script>
```

Il codice JavaScript può essere associato a degli **eventi** che l'utente genera interagendo con la pagina: pressione di un bottone, selezione di una casella, ecc.

Di seguito è fornito un codice avente del JavaScript che lavora con eventi:

```

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

Javascript e Document Object Model (DOM)

Il DOM trasforma una pagina *web* da documento statico a *Graphical User Interface* (GUI). Il codice JavaScript ottenuto nella pagina HTML ed eseguito dal *browser* può agire sull'oggetto che rappresenta la pagina e modificarla: una pagina si automodifica come reazione a certe azioni scatenate dall'utente. Il comportamento è simile a un'applicazione (*web application*).

javascript-load.html

```

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

javascript-timer.html

```

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

javascript-timer2.html

```

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

```

4.3 Web server

Un *web server* è un *software* o un'applicazione che fornisce contenuti *web* ai *client* che ne fanno richiesta attraverso *Internet* oppure una rete locale. In pratica è responsabile di servire pagine *web*, *file*, immagini e altri contenuti ai dispositivi degli utenti quando questi accedono a un sito *web*.

In generale ecco illustrato il funzionamento nei seguenti punti:

1. **Richiesta del *client*:** quando un utente digita un URL nel *browser* oppure clicca un collegamento, allora il *browser* invia una richiesta HTTP al *server web* corrispondente.
2. **Richiesta al *server web*:** il *server web* riceve la richiesta e cerca il *file* corrispondente all'URL nella sua struttura di *directory*.
3. **Elaborazione della richiesta:** se il *file* richiesto è **statico**, come un documento HTML oppure un'immagine, allora il *server* lo recupera e lo invia al *client*. Diversamente se il *file* è **dinamico**, come una pagina generata da uno *script*, allora il *server* esegue lo *script* in modo da generare il contenuto.
4. **Generazione della risposta:** il *server web* crea una risposta HTTP, la quale include l'*header* (informazioni sulla risposta) e il corpo (contenuto effettivo).
5. **Invio della risposta:** il *server* invia la risposta al *client* attraverso *Internet*. Questa viaggia attraverso una serie di *router* e dispositivi di rete fino a raggiungere il *client*.
6. **Visualizzazione sul *browser*:** il *browser* del *client* riceve la risposta. L'*header* HTTP contiene le informazioni sullo stato della risposta e il tipo di contenuto. Il *browser* interpreta l'*header* e il corpo per visualizzare correttamente il contenuto.
7. **Interazione utente:** il *browser* visualizza il contenuto ricevuto dal *server*.

Uniform Resource Locator (URL)

L'**URL** (*Uniform Resource Locator*) è una stringa che permette di identificare in maniera univoca una risorsa HTTP in qualsiasi parte della rete mondiale.

`https://www.univr.it/servizi/studenti/carriera`

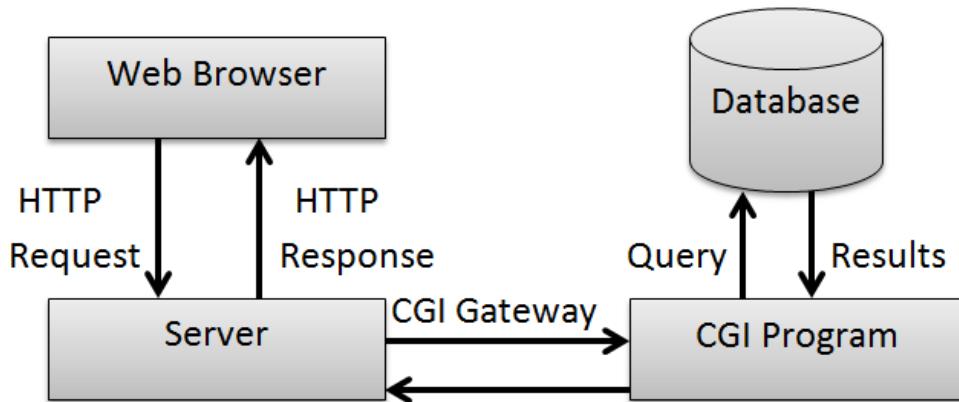
La struttura è caratterizzata dalle seguenti parti:

- protocollo utilizzato a livello di applicazione; protocollo di livello di trasporto + porta utilizzata (HTTP → TCP/80, HTTPS → TCP/443);
- nome dell'*host* (o indirizzo IP) che eroga tale risorsa;
- nome della risorsa definita tramite il suo percorso logico completo.

Se la porta utilizzata non è quella *standard*, allora può essere indicata esplicitamente in questo modo: `https://www.univr.it:8000/servizi/studenti/carriera`.

4.3.1 Common Gateway Interface (CGI)

Il **CGI** (*Common Gateway Interface*) è uno *standard* che consente ai *server web* di interagire con programmi o *script* esterni per generare contenuti dinamici e interattivi sulle pagine *web*.



Il suo funzionamento è espresso nel dettaglio nei seguenti punti:

1. **Richiesta del client:** un *browser* effettua una richiesta a un *server web* per una risorsa che coinvolge l'esecuzione di uno *script* CGI. Questa risorsa potrebbe essere una pagina *web* dinamica oppure un'azione che richiede l'elaborazione dei dati.

2. **Riconoscimento dello *script CGI***: il *server web* riconosce la richiesta come coinvolgente lo *script CGI* in base alla configurazione del *server*. Di solito i *file CGI* sono collocati in una *directory* specifica o sono contrassegnati con un'estensione particolare.
3. **Variabili d'ambiente**: il *server* passa variabili d'ambiente allo *script CGI*. Queste variabili contengono informazioni sulla richiesta, come il metodo HTTP (GET o POST), l'URL richiesto, i parametri della *query*, l'indirizzo IP del *client*, ecc.
4. **Comunicazione con lo *script***: lo *script GCI* legge le variabili d'ambiente per ottenere informazioni sulla richiesta d'invio. I dati inviati dal *client*, come i parametri del modulo in un metodo POST, vengono trasmessi allo *script* attraverso lo *standard input*.
5. **Generazione del contenuto**: lo *script* elabora i dati ricevuti e genera il contenuto dinamico desiderato. Questo contenuto può essere un documento HTML, XML, JSON o qualsiasi altro formato supportato.
6. **Creazione della risposta**: lo *script* restituisce il contenuto generato al *server web*. Il *server* trasforma il contenuto in una risposta HTTP da inviare al *client*.
7. **Invio della risposta al *client***: il *server* invia la risposta HTTP, che include il contenuto generato dallo *script*, al *client* che ha effettuato la richiesta.

▷ Cos'è lo *script CGI*?

È un programma scritto in un linguaggio di programmazione che viene eseguito da un *server web* per generare contenuti dinamici in risposta a una richiesta HTTP da parte di un *client*. Gli *script CGI* sono chiamati dal *server web* ogni volta che viene richiesta una risorsa che coinvolge lo *script*.

Il risultato della richiesta del *browser* non è più un documento statico.

Grazie all'interfaccia CGI il *web* diventa sia dinamico e sia una porta a tutti i servizi remoti, come può essere per esempio la posta elettronica, Wikipedia, *Content Management Systems* (CMS), ecc.

4.3.2 WebSocket

I **WebSocket** sono un protocollo di comunicazione bidirezionale, *full-duplex* e persistente che consente una comunicazione interattiva in tempo reale tra un *client* e un *server* attraverso una connessione TCP; sono un'alternativa a HTTP e HTTPS.

All'inizio viene instaurata una sessione HTTP/HTTS e poi attraverso l'operazione di *protocol upgrade* parte una connessione TCP. Da questo momento in poi entrambi i processi possono mandare dati dall'altra parte.

4.4 Scrittura di applicazioni che usano la rete

Le **applicazioni “monolitiche”** sono quelle in cui l’interfaccia utente richiama delle funzionalità fornite da una serie di librerie collegate in un unico programma, il quale viene eseguito sulla macchina dell’utente.

4.4.1 Architetture orientate ai servizi SOA

Il ***service-oriented architecture*** (SOA) è un approccio architettonico per la progettazione e la distribuzione di sistemi *software* che incoraggia il riutilizzo², la modularità³ e l’interoperabilità⁴ dei servizi. In un’architettura SOA, le diverse funzionalità del sistema sono organizzate come servizi indipendenti che possono essere sviluppati, implementati e distribuiti in modo separato. Questi servizi possono comunicare tra loro attraverso standard di comunicazione ben definiti.

L’idea centrale di SOA è la creazione di servizi autonomi e riusabili, che svolgono funzioni specifiche e possono essere richiamati da altri servizi o applicazioni per realizzare processi complessi.

Le caratteristiche principali di SOA sono:

- **Decomposizione:** si suddividono le funzionalità complesse in servizi separati, ciascuno con una responsabilità specifica.
- **Riutilizzo:** i servizi possono essere utilizzati in più contesti e applicazioni, riducendo la duplicazione dello sviluppo.
- **Interoperabilità:** i servizi comunicano attraverso interfacce standardizzate, consentendo a sistemi diversi di collaborare.
- **Scalabilità:** i servizi possono essere sviluppati, distribuiti e scalati indipendentemente dalle esigenze.

Vantaggi di SOA

I vantaggi di SOA sono descritti e approfonditi nei seguenti punti:

1. *La potenza di calcolo e la memoria sono delegate ai server.*

In un’architettura SOA, i servizi possono essere distribuiti su *server* più potenti. Questo consente di sfruttare le risorse di calcolo e memoria di tali *server*, migliorando le prestazioni e la scalabilità delle applicazioni.

²I servizi atomici una volta sviluppati possono essere riutilizzati in più contesti e applicazioni.

³È la suddivisione delle funzionalità in unità distinte e ben definite, ovvero i servizi. Ogni servizio è responsabile di un compito specifico e può essere sviluppato, aggiornato o sostituito in modo indipendente dagli altri.

⁴È la capacità dei servizi di comunicare e collaborare tra loro in modo efficace, indipendentemente dalla tecnologia o dalla piattaforma su cui sono stati sviluppati.

2. Protezione della proprietà intellettuale su algoritmi strategici.

L'algoritmo risiede nel *server* e di conseguenza non viene dato direttamente in mano agli utenti.

3. Annullamento della necessità di distribuire aggiornamenti del software.

Le modifiche in un servizio specifico non richiedono necessariamente la distribuzione dell'intero *software*.

4. Nuovo modello “pay per use”.

Si fa pagare agli utenti l'effettiva parte utilizzata dell'applicazione, come può essere un abbonamento. Ne consegue da questo l'eliminazione della pirateria, in quanto ciascun utente paga per sé.

5. Time to market più rapido e maggiore flessibilità.

Il riutilizzo dei servizi rende molto più semplice e rapido l'assemblaggio di applicazioni, riducendo il carico di lavoro degli sviluppatori che non devono cominciare da zero come nel caso delle applicazioni monolitiche.

6. Manutenzione semplice.

Tutti i servizi sono autonomi e indipendenti, inoltre è possibile modificarli e aggiornarli in base alle esigenze senza influire sugli altri servizi.

7. Maggiore affidabilità.

Eseguire il *debug* di servizi più piccoli rispetto a codici di grandi dimensioni è più semplice, le applicazioni create tramite il modello SOA sono più affidabili.

Le tecnologie presenti in SOA

L'infrastruttura di rete è un elemento essenziale; senza questa l'applicazione non può funzionare. Come accennato in precedenza SOA dispone di varie tecnologie:

- **Servizio:** è una funzionalità, ovvero una chiamata a funzione. Questa è composta da:

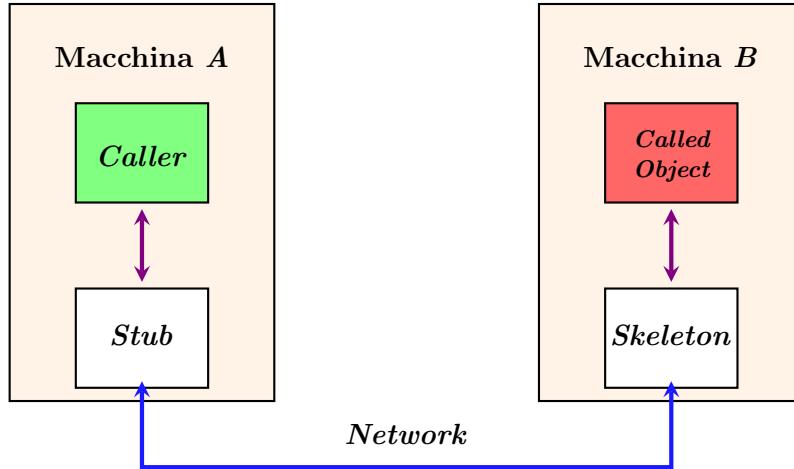
- un nome che indica cosa fa la funzione;
- un tipo e numero di parametri in ingresso;
- un tipo e valore restituito;
- la sua implementazione.

- **Interfaccia:** descrizione di nome, parametri e valore di ritorno delle funzioni erogate da una libreria che è locale in caso di applicazioni tradizionali oppure remota in caso di SOA.

- L'**API** (*Applicazion program interface*) è un insieme delle funzioni esposti da una certa libreria locale o da un *server* remoto.

- La **chiamata di funzione remota** assume una somiglianza concettuale conforme al modello *client/server*.
 - Il componente *server* espone un API che descrive una serie di funzioni che il *client* (non è un *browser*) può invocare.
 - L'implementazione delle funzioni sta nel *server*. Questo componente è responsabile dell'esecuzione delle operazioni richieste dal *client*.
 - Il codice che chiama la funzione sul *client* e quello che implementa la funzione sul *server* possono essere scritti con linguaggi diversi ed eseguiti su architetture di calcolo molto differenti. Entrambe le funzioni implementano la medesima interfaccia.
 - La funzione chiamata dal *client* apparentemente realizza la funzionalità. In realtà *codifica* i parametri per essere trasmessi in rete e *decodifica* il valore di ritorno. Tale funzione prende il nome di **STUB**.
 - Sul *server* esiste un componente chiamato **SKELETON** che *decodifica* i parametri di *input*. Successivamente li passa alla funzione che contiene l'implementazione vera e propria, detta **BUSINESS LOGIC**, che prende il risultato e lo *codifica* spedendolo al *client*.

Ci vuole un protocollo di rete per il trasporto dei dati codificati. La codifica a volte si definisce **serializzazione**.



Tecnologie/standard per la chiamata di funzione remota

- *Remote Procedure Call* (RPC) (C su TCP);
- *Java Remote Method Invocation* (JAVA RMI) (Java su TCP);
- *Common Object Request Broker Architecture* (CORBA);

- *Standard* indipendente dai linguaggi di programmazione e dal protocollo del livello di trasporto.
- *Webservice*:
 - HTTP/HTTPS come protocollo di trasporto degli elementi della funzione.
 - Il formato dei dati segue il protocollo XML SOAP (pesante e datato) oppure la metodologia REST (attualmente la più usata).

4.4.2 Webservice basati su REST

I *webservice* basati su **REST** (*Rapresentational State Transfer*) sono un tipo di architettura di servizi *web* che seguono i principi di REST per la progettazione e l'implementazione delle interazioni tra *client* e *server*. REST è un approccio architetturale che si basa su un insieme di vincoli e principi che mirano a creare servizi *web* scalabili, interoperabili⁵ e ben strutturati.

Di seguito sono riportate alcune caratteristiche chiave dei *web service* basati su REST:

- *Mapping del nome della funzione sull'URL*.
Le funzionalità sono mappate sugli URL e REST si concentra sulle operazioni eseguite sulle risorse.
- *Utilizzo di HTTP/HTTPS come protocolli*.
I *web service* utilizzano i protocolli HTTP/HTTPS per veicolare le richieste e le risposte tra *client* e *server*.
- *Passaggio dei parametri sull'URL o nell'header*.
I parametri possono essere passati nell'URL attraverso i metodi GET o DELETE, oppure nel corpo della richiesta attraverso metodi POST o PUT.
- *Valore di ritorno nel corpo della risposta*.
Il valore di ritorno dell'operazione viene spesso incluso nel corpo della risposta. Questo può essere nel formato di testo puro, JSON oppure altri formati, a seconda delle esigenze e delle preferenze.

Metodi nella richiesta HTTP

Nel contesto dei servizi REST i metodi impiegati nella richiesta HTTP possono essere:

- **GET**: viene utilizzato per recuperare i dati dal *server*. Quando viene effettuata una richiesta GET, il *server* restituisce una rappresentazione della risorsa richiesta.
- **POST**: viene utilizzato per inviare dati al *server* per l'elaborazione. In genere è impiegato per la creazione di nuove risorse sul *server*.

⁵È la capacità di diverse applicazioni o sistemi di comunicare e collaborare senza problemi.

- **PUT:** viene utilizzato per aggiornare o creare una risorsa specifica sul *server*. A differenza di **POST**, il metodo **PUT** è idempotente, ossia se si esegue la stessa richiesta **PUT** più volte allora ha lo stesso effetto di eseguirla una volta sola. Se la risorsa specificata non esiste, allora **PUT** può anche crearla.
- **DELETE:** viene utilizzato per rimuovere una risorsa dal *server*. Tale azione è irreversibile e anche questo metodo dovrebbe essere idempotente.

File di tipo JSON

È un formato di dato testuale nato con Javascript, ma oggi supportato in tutti i linguaggi di programmazione. La struttura gerarchica è facilmente leggibile da un umano e parserizzabile da un programma.

- **Elemento base:** coppia **attributo:valore**.
- **Tipi base:** stringa (da indicare sempre tra “ ”), numeri, booleani e *null*.
- **Composizione di elementi omogenei:** sono i vettori [...].
- **Composizione di elementi eterogenei:** struttura dati {...}.

```

1  { "nome": "Ugo", "anni": 30, "laurea": null, "abbonato": false}
2  { "impiegati": [
3    { "nome": "Giovanni", "cognome": "Rossi" },
4    { "nome": "Anna", "cognome": "Bianchi" },
5    { "nome": "Pietro", "cognome": "Verdi" }
6  ]}

```

Vantaggi dei web service

I benefici chiave associati all'uso dei *web service* sono:

- *Infrastruttura Internet predisposta*.
L'infrastruttura *Internet* è già configurata per supportare protocolli come HTTP/-HTTPS, i quali sono comunemente utilizzati per i *web service*. Questo semplifica l'integrazione dei servizi e la comunicazione tra applicazioni su reti diverse.
- *Compatibilità con Firewall e NAT*.
I *web service*, specialmente quelli basati su protocolli *standard* come HTTP, possono superare le restrizioni dei *firewall* e dei dispositivi NAT.
- *Utilizzo di contenuti testuali e facilità di debugging*.
I *web service* spesso utilizzano formati di dati testuali come XML o JSON per la rappresentazione dei dati scambiati. Questo rende più facile il *debugging* delle transazioni, poiché i contenuti testuali sono leggibili e possono essere analizzati senza l'uso di strumenti specializzati. Questo è particolarmente vantaggioso quando si sviluppano applicazioni SOA complesse.

4.4.3 Differenze tra SOA e REST

Sono due approcci architetturali distinti utilizzati per progettare e implementare sistemi software distribuiti e servizi web, ma hanno alcune differenze chiave:

- **Principi e filosofia:**

- REST è un approccio architetturale che si basa su un insieme di vincoli e principi focalizzati sulla modellazione delle risorse e sulla loro rappresentazione attraverso URL. REST promuove l'uso di metodi HTTP *standard* (GET, POST, PUT, DELETE) e la mancanza di stato nel *server*.
- SOA è un paradigma architetturale che mette l'accento sull'organizzazione dei servizi come unità modulari e riusabili all'interno di un sistema. SOA promuove l'interoperabilità tra i servizi e l'accesso a funzionalità attraverso interfacce standardizzate.

- **Flessibilità VS. standardizzazione:**

- REST è spesso associato a un approccio più leggero e flessibile. Non esistono *standard* rigidi per la definizione delle risorse o delle interfacce, il che significa che l'implementazione può variare più ampiamente.
- SOA, al contrario, è spesso associato a una maggiore standardizzazione attraverso la definizione di interfacce ben definite e servizi ben strutturati, il che può contribuire a una maggiore interoperabilità.

- **Comunicazione e formato dei dati:**

- REST utilizza spesso formati di dati come JSON o XML per rappresentare le risorse e le loro rappresentazioni.
- SOA può utilizzare vari protocolli di comunicazione e formati di dati, come SOAP (*Simple Object Access Protocol*) per la comunicazione tra servizi.

- **Focalizzazione sulle risorse VS. servizi:**

- REST si concentra sulla modellazione delle risorse e delle loro rappresentazioni, con un'enfasi sulla semantica delle operazioni (GET, POST, PUT, DELETE) applicate alle risorse.
- SOA si concentra sulla definizione, organizzazione e gestione dei servizi, con un'enfasi sulla creazione di interfacce standard per l'accesso a queste funzionalità.

4.5 Cloud computing

Il **cloud computing** è un paradigma tecnologico che offre risorse come *server*, archiviazione, *database*, rete e *software*, tramite *Internet*. Invece di possedere fisicamente e gestire

questi componenti, le organizzazioni possono noleggiarli o utilizzarli su richiesta attraverso *provider* di servizi *cloud*. Alcuni esempi di servizi *cloud computing* sono: Amazon Web Service, Google Cloud, Microsoft Azure, Dropbox.

Quindi in altre parole il *cloud computing* è un gruppo di *host* che forniscono servizi di calcolo e memorizzazione senza essere indirizzati o gestiti individualmente dagli utenti. L'intero insieme di *hardware* e *software* è gestito dal fornire del *cloud* che si fa pagare con una politica a consumo oppure forfettaria.

Le tecnologie abilitanti per il *cloud* sono:

- utenti perennemente connessi in rete;
- virtualizzazione per ottimizzare l'uso delle risorse *hardware*;
- *webservice* per sfruttare le risorse di calcolo da remoto.

I servizi offerti dal cloud

Vengono messi a disposizione diversi modelli di distribuzione del *cloud computing*:

- ***On-site*** oppure ***on-premise***: l'*hardware* e l'infrastruttura sono di proprietà e gestiti direttamente dall'organizzazione stessa all'interno delle proprie strutture. Questo richiede un controllo completo, ma comporta anche una maggiore responsabilità per la manutenzione, la sicurezza e l'aggiornamento dell'infrastruttura.
 - ***Infrastructure as a Service (IaaS)***: le risorse *hardware* virtuali vengono noleggiate dai fornitori di servizi *cloud*. Gli utenti possono configurare e gestire il proprio sistema operativo, le applicazioni e i dati su queste macchine virtuali. Questo offre maggiore controllo rispetto ai livelli superiori del *cloud*.
 - ***Platform as a Service (PaaS)***: oltre alle risorse *hardware*, il fornitore di servizi *cloud* offre anche l'ambiente di sviluppo, il sistema operativo e l'infrastruttura *runtime*. Gli sviluppatori possono concentrarsi sull'applicazione senza doversi preoccupare dell'infrastruttura sottostante. Strumenti come Docker possono anche essere integrati per la gestione dei contenitori.
 - ***Software as a Service (SaaS)***: in questo modello gli utenti accedono e utilizzano già applicazioni installate e ospitate sul *cloud*. Queste applicazioni possono essere per esempio *server web*, *database server* o servizi di messaggistica. Gli utenti non devono gestire l'infrastruttura o le attività di manutenzione.
 - ***Function as a Service (FaaS)***: è un modello in cui gli sviluppatori possono creare, eseguire e gestire singole funzioni o pezzi di codice senza doversi preoccupare dell'infrastruttura sottostante o della gestione dei *server*. Invece di dover configura-re un'intera applicazione o un *server*, gli sviluppatori creano direttamente funzioni che rispondono a specifici eventi.
- Quindi è possibile scegliere un linguaggio preferito, associare le funzioni a degli eventi, eseguire le funzioni allocando dinamicamente e sul momento le risorse di

calcolo necessarie e infine il committente non deve nemmeno creare il processo *server* che le fornisce.

- Diversi fornitori *cloud* offrono servizi FaaS come: AWS Lambda, Google Cloud Functions, IBM Cloud Functions based on Apache OpenWhisk, Microsoft Azure Functions, Oracle Cloud Functions.

Utilizzo di Docker

Docker è un popolare *software* libero progettato per eseguire processi informatici in ambienti isolabili, minimali e facilmente distribuibili chiamati **container Linux** (o anche soltanto **container**), con l'obiettivo di semplificare i processi di *deployment* (sviluppo) di applicazioni *software*.

▷ Da quali bisogni è nato Docker?

Ci sono due motivazioni principali che danno il via alla sua nascita:

1. Data la grande varietà di sistemi operativi, CPU, architetture, come è possibile adattare semplificare la diffusione di un *software*? La soluzione è Docker che consente la distribuzione semplice di tali *software*.
2. Le più grandi applicazioni al mondo sfruttano delle apparecchiature performanti per consentire l'esecuzione del loro applicativo:
 - **Cluster**: gruppi di decine di *computer* connessi da una LAN ad alte *performance*.
 - **Data centers**: gruppi di migliaia di *computer* connessi da un insieme gerarchico di LAN ad alte *performance*.

Per gestire questi gruppi di *computer* è comodo utilizzare Docker.

Distribuire un *software* tramite *cointainer* significa forzare il programmatore a utilizzare una determinata versione di: distribuzione del SO, *framework*, varie dipendenze come librerie o moduli. In questo modo, il creatore è certo che qualsiasi utilizzatore potrà utilizzare il *software* con le risorse corrette.

Lo sviluppo di un *container* ha portato ad una serie di benefici:

- aumento della sicurezza grazie all'isolamento di ogni parte dell'applicativo;
- scalabilità;
- portabilità attraverso vari sistemi operativi e piattaforme *hardware*;
- stabilità evitando conflitti tra applicazioni di terzi e prevenendo eventuali aggiornamenti di sistema che potrebbero impattare sull'applicazione.

5.1 Virtualizzazione e containerizzazione

La **virtualizzazione** sfrutta un *software* chiamato Hypervisor che consente di eseguire molteplici sistemi operativi a fianco sullo stesso *hardware*. Quindi si dice che la virtualizzazione virtualizza l'*hardware* fisico. Il risultato è che ogni macchina virtuale contiene:

- un sistema operativo completo;
- una copia virtuale dell'*hardware* (essa è richiesta dal sistema operativo per essere eseguito);
- Almeno un'applicazione e le sue relative librerie e dipendenze (*libraries and dependencies*).

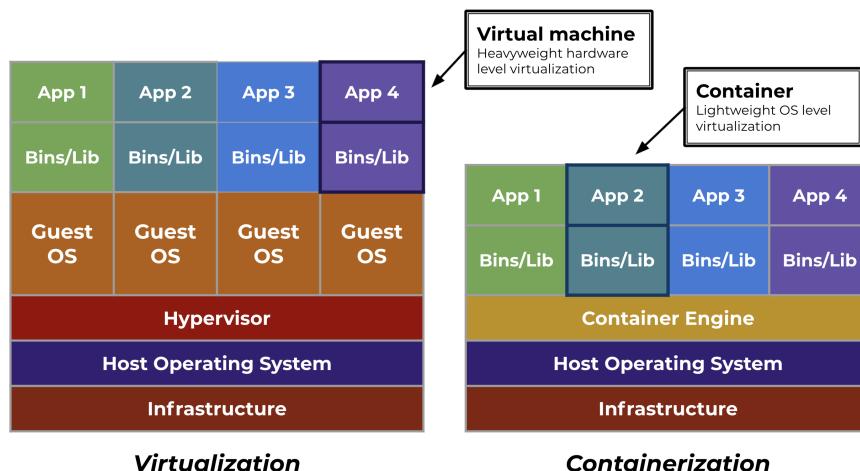
Per esempio un *computer* Windows può scaricare un *software* di virtualizzazione (come VM VirtualBox, VMware) e ospitare al suo interno una o più macchine virtualizzate con sistemi operativi uguali o differenti.

La **containerizzazione** è un sistema più intelligente e invece di virtualizzare l'intero *hardware*, esso esegue una virtualizzazione solo del sistema operativo. In questo modo, ogni *container* conterrà solo le sue applicazioni e le sue dipendenze.

Infine la containerizzazione consente agli sviluppatori di migliorare l'utilizzo della CPU e della memoria. Inoltre possono essere abilitati una serie di architetture di microservizi che facilitano lo sviluppo granulare delle parti di un'applicazione.

Quindi la virtualizzazione virtualizza l'intero *hardware* comprendendo SO, copia virtuale HW, applicazioni (più librerie e dipendenze). Mentre la containerizzazione virtualizza solo il sistema operativo.

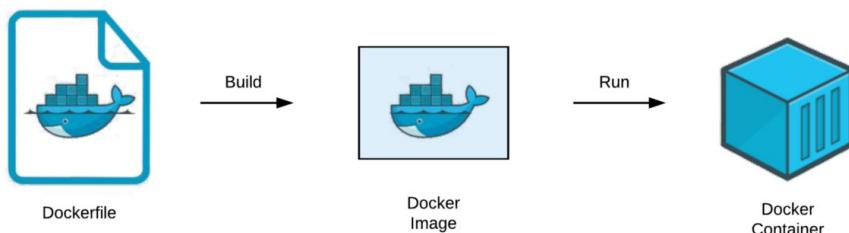
- **Pro:** la containerizzazione è più leggera sia dal punto di vista delle risorse sia dal punto di vista dell'installazione. In altre parole si può dire che è *small, fast and portable*.
- **Contro:** la virtualizzazione è più pesante, perché sono necessarie una grande disponibilità di risorse sul proprio *hardware*.



5.2 Struttura tecnica di Docker

Il ciclo vita di una containerizzazione è il seguente:

1. creazione di un **Dockerfile** (esecuzione della *build*);
2. una volta eseguita la *build*, all'interno del *computer* sarà presente la **Docker Image**;
3. all'esecuzione della **Docker Image**, il Docker *container* sarà attivo e in esecuzione.



5.2.1 Struttura di un Dockerfile

Un **Dockerfile** è una lista di istruzioni utilizzati dal motore della containerizzazione per eseguire il processo di *build* di un'immagine che conterrà l'applicazione che si vuole programmare. In altre parole viene definito lo specifico ambiente in cui si vuole l'applicazione.

```

1 #extending base image
2 FROM ubuntu:18.04
3
4 #maintainer email address
5 LABEL maintainer="john.doe@example.com"
6
7 #updating libraries
8 RUN apt-get -y update
9
10 #installing java8
11 RUN add-apt-repository ppa:webupd8team/java && \
12 apt install -y oracle-java8-installer
13
14 #copying target application package
15 COPY . /app/
16
17 #setting the working directory
18 WORKDIR /app/
19
20 #running the application
21 CMD ["java", "-jar", "myapp.jar"]
  
```

5.2.2 Struttura di un Docker Image

Un **Docker image** è un *template* fondamentale per eseguire un Docker *container*. Una volta eseguita la *build* di un’immagine è possibile utilizzarla attraverso diversi sistemi per eseguire lo stesso *container*.

5.2.3 Struttura di un Docker container

Un **Docker container** contiene una serie di processi isolati dal sistema operativo ospitante. Tuttavia è possibile far comunicare i vari *containers* tra di loro tramite l’apertura delle porte di rete.

5.2.4 Comandi utili

Comando	Descrizione
docker ps	Restituisce una lista di tutti i <i>container</i> in esecuzione nel sistema. Ogni <i>container</i> ha un <i>id</i> e un <i>link</i> all’immagine dal quale è stato generato. Con l’opzione -a è possibile mostrare sia i <i>container</i> in esecuzione che non, cioè tutti.
docker build	Esegue la <i>build</i> di un’immagine da un Dockerfile. Come argomento deve essere specificato il <i>path</i> in cui si trova il Dockerfile (con . si indica l’attuale cartella).
docker pull	Scarica un’immagine <i>pre-built</i> dal Docker Hub (una sorta di GitHub).
docker images	Lista di tutte le immagini disponibili.
docker run	Crea un <i>container</i> da un’immagine e lo avvia (esegue).
docker start	Avvia un <i>container</i> esistente.
docker stop	Mette in pausa un <i>container</i> esistente.
docker restart	Riavvia un <i>container</i> esistente.
docker kill	Uccide un <i>container</i> esistente, ma non lo elimina.
docker rm	Elimina un <i>container</i> .

5.3 Introduzione a Docker Compose

Quando un progetto ha bisogno di una **Docker image** che fornisce una serie di servizi, spesso gli sviluppatori tendono ad utilizzare uno strumento chiamato **Docker compose**. La sua installazione, su Linux, avviene con il comando `sudo apt install docker-compose`.

Con questo strumento, i programmatore possono scrivere un file `.YAML`, il quale descrive tutti i servizi dell'applicazione. Così facendo, con l'esecuzione di un semplice comando, sarà possibile mandare in esecuzione tutti i *container* che forniscono i servizi necessari all'applicazione.

Per esempio, viene utilizzato Docker compose per i progetti che richiedono due servizi: *front-end* e *database*.

Paradigma Publish/Subscribe

6.1 Modello Pub/Sub

Il **modello *Pub/Sub*** (*Publish/Subscribe*) è un’architettura che fa parte dei *design pattern*, i quali vengono utilizzati nei sistemi distribuiti per comunicazioni asincrone tra molteplici componenti o servizi. Nonostante il modello *Pub/Sub* sia basato su dei *design pattern* molto giovani, quali le code di messaggi (*message queuing*) e gli *event brokers*, risulta ancora oggi molto flessibile e scalabile. Questo perché l’architettura consente lo scambio di messaggi tra componenti diversi di un sistema, senza che essi siano a conoscenza dell’identità degli altri componenti. In altre parole, si può dire che sono disaccoppiati.

La nascita di questo modello risale alla necessità di espandere la **scalabilità** dei sistemi informativi. Prima di *Internet*, i sistemi erano largamente scalabili staticamente. Tuttavia, con l’espansione di *Internet*, delle *web-application*, dei dispositivi mobili e soprattutto dei dispositivi IoT (*Internet of Things*), i sistemi hanno necessità di avere una scalabilità dinamica.

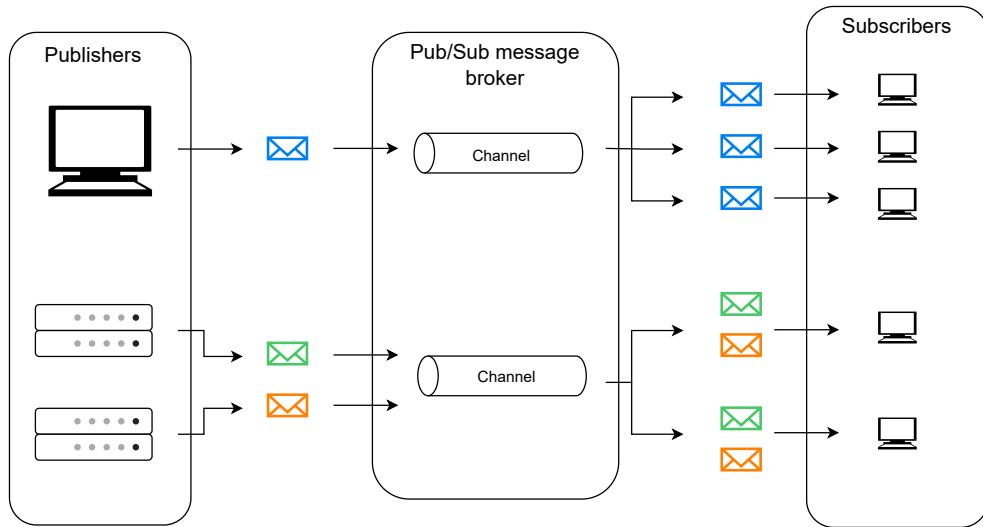
Il vantaggio principale di avere un’**architettura disaccoppiata**, ovvero di un sistema in cui i componenti non siano a conoscenza dell’identità degli altri, risulta essere un’ottima soluzione per tutti quei sistemi dinamicamente scalabili. Se ne deduce che al giorno d’oggi, con l’avvento del IoT, questo modello si stia diffondendo.

Inoltre, il modello consente di gestire la scalabilità senza sovraccaricare (o in gergo *overloading*) la logica dei programmi dei componenti di un sistema.

Nel dettaglio, l’architettura *Pub/Sub* fornisce una struttura (*framework*) di scambio di messaggi tra i ***publishers*** (componenti utilizzati per creare e inviare messaggi) e ***subscribers*** (componenti che ricevono e leggono messaggi).

Osservazione

I mittenti (*publishers*) non inviano messaggi a uno specifico destinatario (*subscribers*) in modalità *end-to-end*. Al contrario viene utilizzato un intermediario, chiamato ***message broker***, il quale raggruppa i messaggi in entità chiamate canali (*channels*) o argomenti (*topics*).



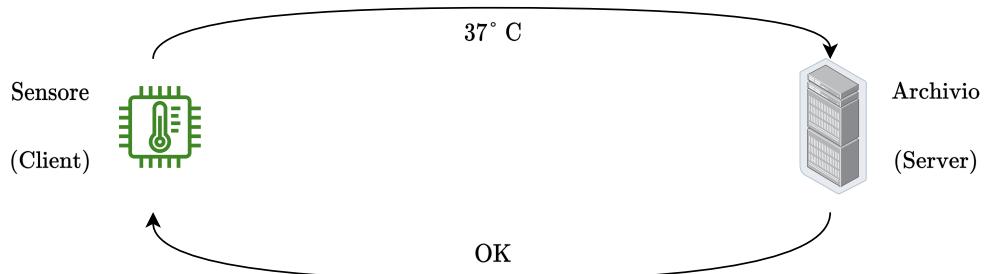
6.2 Confronto tra Client/Server e Pub/Sub

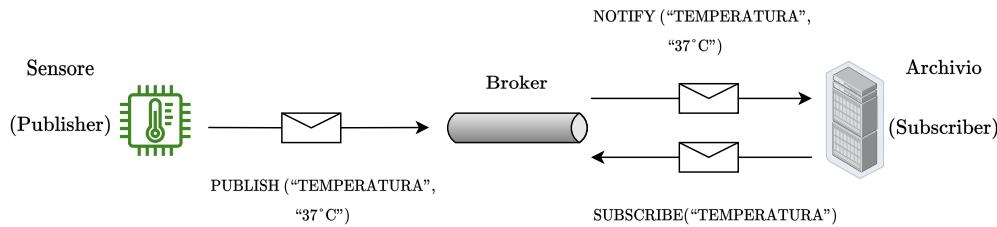
Le differenze tra il modello *Client/Server* e *Pub/Sub* sono molto nette. Nel **modello client/server**, l'*host* su cui vengono ospitati i programmi per usarlo come *server*, ha requisiti particolari:

- Deve rimanere sempre acceso per garantire l'accesso a tutti gli utenti.
- Deve essere sempre raggiungibile tramite *Internet*, perciò deve avere un indirizzo IP pubblico senza misure di sicurezza (NAT e *Firewall*) troppo restrittive.
- Deve essere disponibile a ricevere una grande mole di richieste da parte di *client*.

Al contrario l'**architettura Pub/Sub** ha un **intermediario** (*broker*) che si assume tutte le responsabilità di un *server* (in un modello *client/server*). Inoltre, cercando di fare un'analogia, i *publisher* e i *subscribers* possono essere visti come dei *client*.

Dato che il modello *Pub/Sub* è un *design pattern*, è sempre possibile trasformare un'applicazione *client/server* in un modello *Pub/Sub*. Questo consente di ricevere una serie di vantaggi: i *server* originari diventano più leggeri e la complessità di calcolo viene delegata all'*host* intermediario (*broker*).





Il modello *Pub/Sub* è ***data centric*** poiché evidenzia principalmente chi sono i produttori/consumatori di un certo tipo di informazione. Al contrario il modello *client/server* è ***host centric***. Un'applicazione basata su *Pub/Sub* è più facilmente estendibile nel caso in cui: devono variare il numero di produttori/consumatori ed è necessario aggiungere diversi tipi di informazione.

Infine *Pub/Sub* è più leggero a livello di *byte* trasmessi a confronto del modello *client/server* che utilizza HTTP.

Differenze	Spiegazione
Requisiti	Nel modello <i>client/server</i> , i requisiti che necessita un host per essere un <i>server</i> , sono molteplici: <ul style="list-style-type: none"> • sempre accessibile; • sempre raggiungibile tramite <i>Internet</i>; • sempre disponibile a una grande mole di richieste. Al contrario <i>Pub/Sub</i> ha un intermediario (<i>broker</i>) che si assume le responsabilità di un <i>server</i> .
Trasformazione	Data un'applicazione:
	<i>Client/Server</i> $\xrightarrow{\text{trasformazione}}$ <i>Pub/Sub</i>
	Tuttavia non è possibile ottenere il contrario.
<i>Data Centric</i>	Il modello <i>Pub/Sub</i> è considerato <i>data centric</i> poiché i produttori/consumatori sono ben identificati. Al contrario, il modello <i>Client/Server</i> è considerato <i>host centric</i> .
Estendibilità	<i>Pub/Sub</i> è molto estensibile e non sono necessarie grandi cambiamenti all'infrastruttura. Il modello <i>Client/Server</i> non ha questo vantaggio.
Byte inviati	<i>Pub/Sub</i> invia molti meno <i>byte</i> rispetto al modello <i>client/server</i> che utilizza HTTP.

6.2.1 Aspetti positivi dell'architettura Pub/Sub

Gli aspetti positivi di un'architettura *Pub/Sub* sono molteplici. I più importanti sono i seguenti:

- **Scalabilità:** i sistemi *Pub/Sub* sono molto scalabili e la rottura delle funzionalità non è una preoccupazione, poiché la logica di comunicazione e la logica di “*business*” sono due entità ben separate. Gli sviluppatori possono ridisegnare l'intera struttura degli argomenti (*topics*) senza la preoccupazione di rompere la logica di comunicazione.
- **Architettura multi-a-molti:** i sistemi *Pub/Sub* possono avere una grande mole di dati da parte dei *publisher* e dei *subscriber* senza creare criticità al sistema.

6.3 Protocollo MQTT

Il **protocollo MQTT** è un protocollo gestito da OASIS *standards organization* e riconosciuta internazionalmente da ISO. Risulta essere il primo protocollo utilizzato dai dispositivi e dalle applicazioni per comunicare con i servizi presenti sulle piattaforme (*Platform Service*).

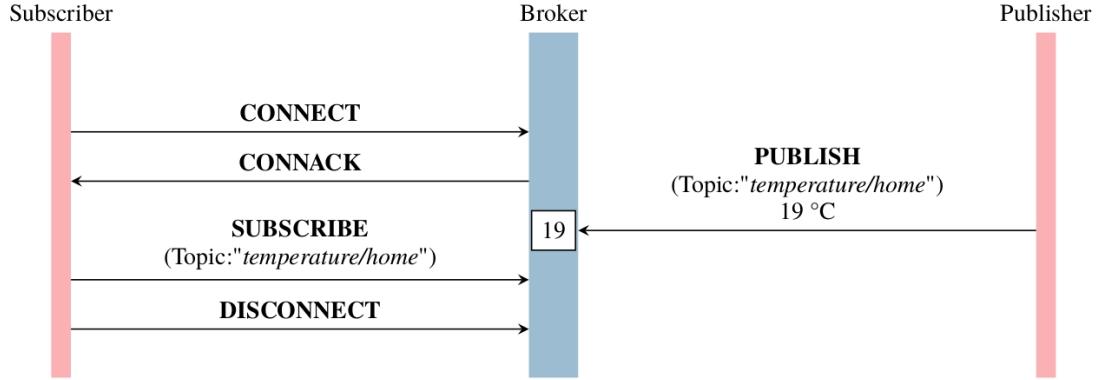
Il protocollo MQTT fornisce tre livelli di qualità del servizio riguardanti la consegna dei messaggi tra *clients* e *servers*:

- **QoS0 - QoS Livello 0** *at most once* (al più una volta)
- **QoS1 - QoS Livello 1** *at least once* (almeno una volta)
- **QoS2 - QoS Livello 2** *exactly once* (esattamente una volta)

Un livello maggiore non significa un aumento di qualità. Ogni livello ha le sue caratteristiche che devono essere considerate caso per caso.

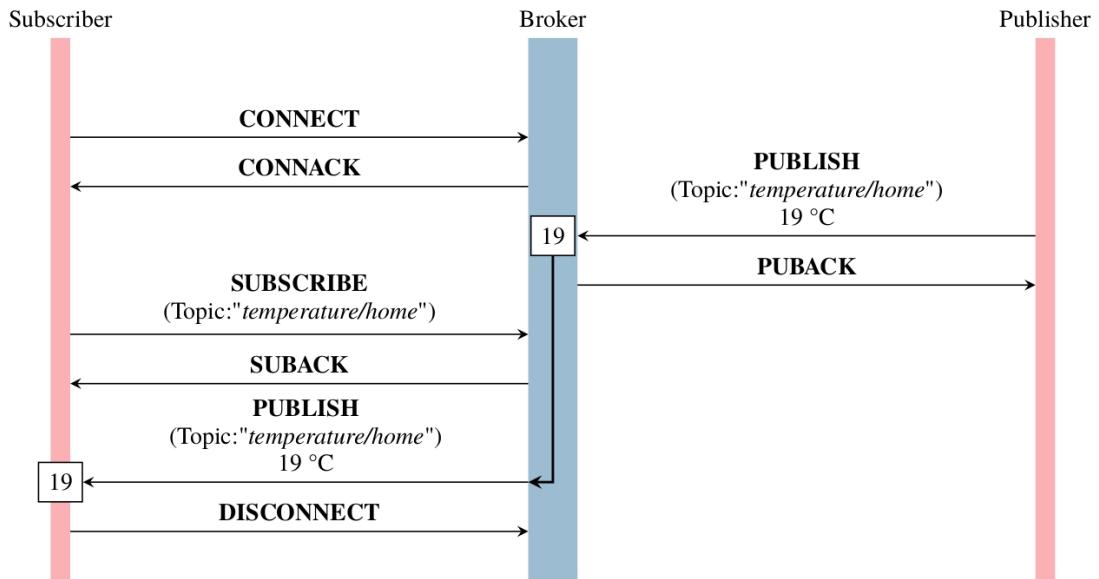
6.3.1 QoS livello 0

La qualità **QoS livello 0**, *at most once*, è il servizio più rapido per quanto riguarda il trasferimento di dati. Spesso viene chiamato *fire and forget*. Il messaggio viene consegnato almeno una volta, in caso contrario la consegna non viene effettuata per niente. Inoltre il messaggio non viene salvato e questo provoca una possibile perdita nel caso in cui il *client* si disconnetta o il *server* fallisca qualche operazione.



6.3.2 QoS livello 1

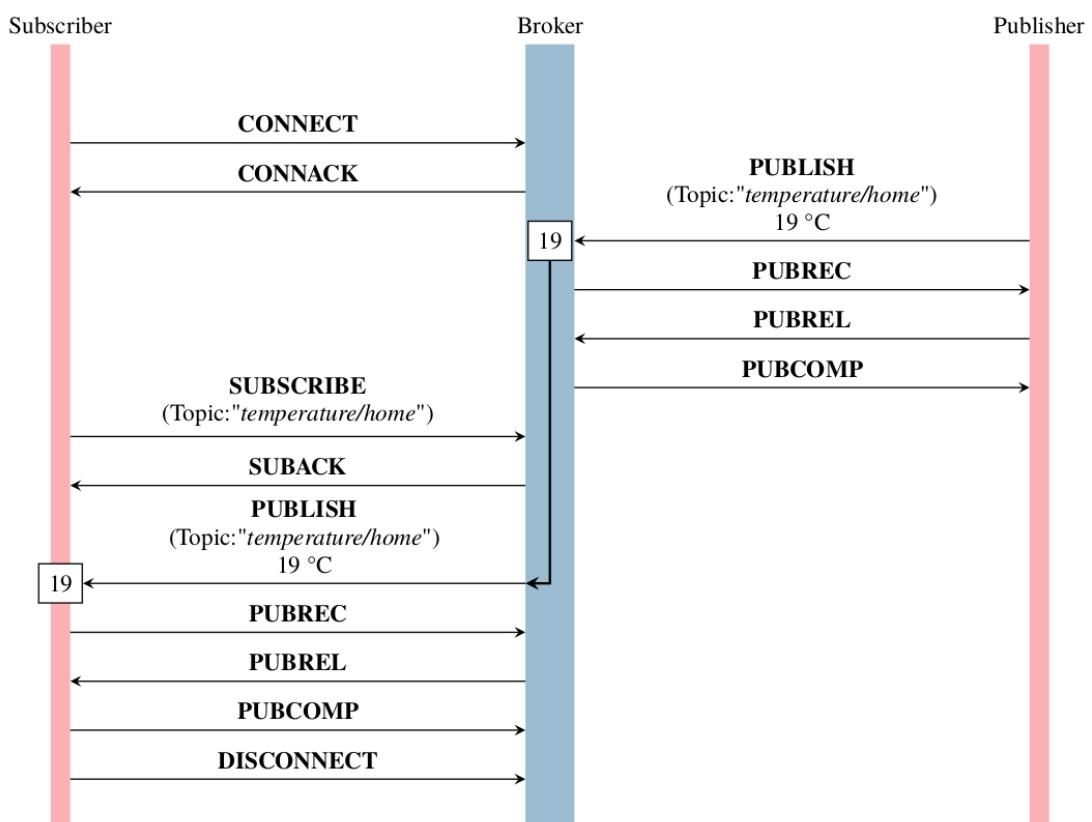
Con la qualità **QoS livello 1**, *at least once*, il messaggio viene consegnato almeno una volta. Nel caso in cui si manifesti un errore e il mittente non riceva l'ACK dal destinatario, un messaggio può essere consegnato più volte. A differenza di QoS0, il messaggio può essere salvato localmente dal mittente finché il destinatario non riceve la notifica di avvenuta pubblicazione (da parte del mittente). In altre parole finché il mittente non è certo che il destinatario sappia che c'è un suo messaggio nel *broker*, esso tiene in memoria il messaggio. Il messaggio viene salvato localmente dal mittente anche nel caso in cui il messaggio debba essere inviato più volte.



6.3.3 QoS livello 2

Con la qualità **QoS livello 2**, *exactly once*, si ha il livello di sicurezza più alto a discapito della modalità di trasferimento che è la più lenta. Il messaggio viene sempre consegnato soltanto una volta e viene anche salvato localmente dal mittente finché il destinatario non ha ricevuto la notifica di avvenuta pubblicazione (come per QoS1).

Per evitare duplicazioni, vengono implementati un *handshaking* e una sequenza di ACK più sofisticati.



6.4 Vari approfondimenti

6.4.1 Struttura del topic

I **topic** (argomenti) servono a etichettare i dati per indicarne la tipologia. Solitamente sono stringhe di testo in cui ci sono spazi, anche se è preferibile utilizzare trattini “-” o “_”. Inoltre è possibile creare gerarchie di argomenti utilizzando lo *slash* “/”. Come per esempio:

- casa/sensori/temperatura/cucina;

- casa/sensori/temperatura/soggiorno.

Qualsiasi *topic* compreso tra “/” si chiama **livello**.

I **subscriber** possono utilizzare i caratteri “#” e “+” come sostituti di parti di *topic*.

6.4.2 Approfondimento Wildcards

I **wildcards** sono simboli utilizzati dai *subscribers* al posto dei *topic* per avere risultati più completi:

- Il carattere “#” viene utilizzato per sottoscriversi a tutti i *topic* che si ottengono sostituendo # con una sequenza di livelli di suffisso reali.

Per esempio invece di sottoscriversi ai *topic*:

- level1/level2/level3;
- level1/level2/level3bis.

Basta sottoscriversi a:

- level1/level2/#.

- Il carattere “+” viene utilizzato per sottoscriversi a tutti i *topic* che si ottengono sostituendo + con una sequenza di livelli di prefisso reali.

Per esempio invece di sottoscriversi ai *topic*:

- level1/level2/level3;
- level1/level2/level3bis.

Basta sottoscriversi a:

- +/level2/level3.

6.4.3 Approfondimento Broker

Il **broker** è l’intermediario nel modello *Pub/Sub* ed è l’unico che implementa connessioni TCP lato *server* (porta 1883: trasmissioni in chiaro; porta 8883: trasmissioni TCP con sicurezza TLS/SSL). Inoltre, il **broker** può trovarsi su *hardware* e sistemi operativi diversi da quelli dei *Pub/Sub*. Anche il linguaggio di programmazione utilizzato può essere differente. Questo perché è il protocollo MQTT che consente di coordinare tutto.

Sicurezza delle reti

7.1 Principi fondamentali della sicurezza

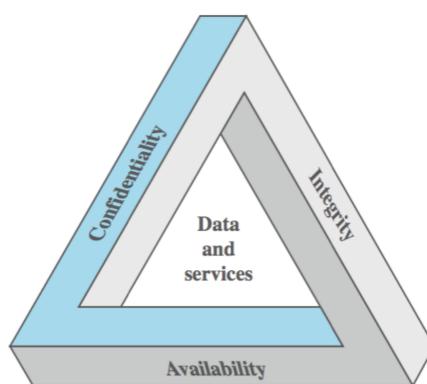
7.1.1 Quali risorse si vogliono proteggere?

Si vogliono proteggere le seguenti risorse:

- **Hardware**: sistemi, componenti e dischi, pertanto una sicurezza “fisica”.
- **Software**: sistema operativo e applicazioni.
- **Dati**: *file* e *database*.
- **Rete**: collegamenti e apparati.

Con il termine “**proteggere**” si indica l’azione tale per cui vengono garantite le seguenti proprietà:

- confidenzialità;
- integrità;
- disponibilità.



In aggiunta ci sono anche l’autenticità e la tracciabilità.

Confidenzialità

Nessun utente deve poter ottenere o dedurre dal sistema informazioni che non è autorizzato a conoscere. Di seguito sono riportati due concetti fondamentali:

- **Riservatezza dei dati:** le informazioni confidenziali non devono essere rivelate o rivelabili da utenti non autorizzati.
- **Privacy:** l'utente controlla o influenza quali informazioni possono essere collezionate e memorizzate.

Integrità

Impedire l'alterazione diretta o indiretta delle informazioni, sia da parte di utenti e processi non autorizzati, che a seguito di eventi accidentali. Se i dati vengono alterati è necessario fornire strumenti per poterlo verificare facilmente.

Di seguito sono riportati due concetti fondamentali:

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

Disponibilità

Rendere disponibili a ciascun utente abilitato le informazioni alle quali ha diritto ad accedere, nei tempi e modi previsti: in determinate condizioni, in un preciso istante, in un intervallo di tempo.

Nei sistemi informatici, i requisiti di disponibilità includono prestazioni e robustezza.

Autenticità

Ciascun utente deve poter verificare l'autenticità delle informazioni: messaggi, mittenti e destinatari. Quindi si richiede di poter verificare se un'informazione è stata manipolata; ciò è valido anche per le informazioni non riservate.

Tracciabilità

Le azioni di un'entità devono essere tracciate in modo univoco così da poter supportare la non ripudiabilità e l'isolamento delle responsabilità.

Esempio

Nessun utente deve poter ripudiare o negare in tempi successivi messaggi da lui spediti o firmati.

7.1.2 In che modo le risorse sono minacciate?

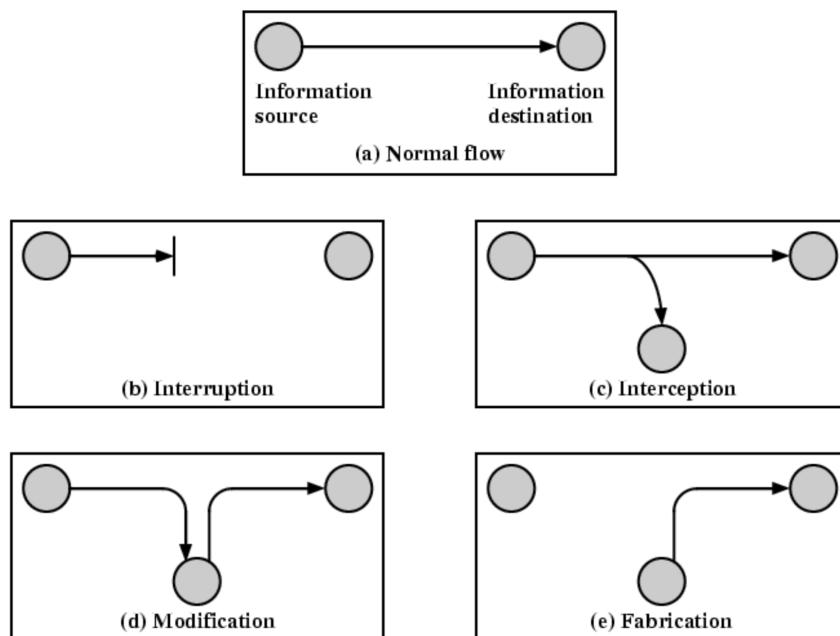
Le minacce compromettono le proprietà di:

- confidenzialità;
- integrità;
- disponibilità.

In tabella vengo riportati alcuni casi:

	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 la **violazione effettiva** è chiamata **attacco**.



Gli attacchi possono essere:

- **Attivi**: tentativi di alterare le risorse o modificare il funzionamento dei sistemi.
- **Passivi**: tentativi di catturare 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.

Classi di minacce/attacchi

- ***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.

7.1.3 Cosa bisogna fare per contrastare le minacce?

Questa risulta essere la domanda più difficile, poiché non esiste una risposta unica e le risposte cambiano nel tempo. Le risorse da proteggere sono sistemi composti da sottosistemi.

- **Attacchi potenziali**: nella progettazione di sistemi occorre considerare i possibili attacchi.
- **Soluzioni contro-intuitive**: nello sviluppo dei meccanismi di sicurezza, dovuto alla complessità del sistema e alle possibili minacce.
- **Dove utilizzare i meccanismi di sicurezza**: sia a livello fisico sia a livello logico (protocollare).
- **La sicurezza dipende anche dagli utenti**: le informazioni possedute (come le *password*) e la creazione, distribuzione e protezione di tali informazioni.

Per gli **attaccanti** basta sfruttare una sola vulnerabilità, mentre agli **amministratori** spetta il compito di prevederle ed eliminarle tutte. Spesso la sicurezza e i suoi meccanismi vengono considerati come elementi addizionali e che rallentano od ostacolano il normale funzionamento dei sistemi. Infatti la percezione della sicurezza come vantaggio emerge solo quando si verifica un incidente di sicurezza.

Nonostante anni di ricerca, è difficile progettare sistemi che prevengano completamente le falliche nella sicurezza. Tuttavia insiemi di pratiche e regole sono state codificate:

- analogamente a quanto avviene per l'ingegneria del *software*.

- *Aspetti economici dei meccanismi.*
 - La progettazione delle misure di sicurezza deve essere il più semplice possibile: da implementare e verificare.
- *Fail-safe default.*
 - Comportamenti non specificati devono prevedere un *default* sicuro, come per esempio i permessi d'accesso.
- *Progettazione aperta.*
 - Preferibile rispetto il codice segreto.
- *Tracciabilità delle operazioni.*
 - Qualsiasi operazione può essere ricostruita e il sistema ripristinato.
- *Separazione dei privilegi.*
 - Differenziazione degli accessi: alle risorse create da ciascun utente (*file*) e alle risorse critiche.
- *Separazione delle funzionalità.*
 - Distinzione dei ruoli nei diversi punti del sistema fisico e logico.
- *Isolamento dei sottoinsieme.*
 - Un sistema compromesso non dovrebbe compromettere gli altri.
- *Modularità.*
 - Meccanismi di sicurezza e indipendenti, sostituibili e riusabili.

Politiche di sicurezza e meccanismi

Una **politica di sicurezza** è un'indicazione di cosa è e cosa non è permesso fare. Le regole possono riguardare i dati (protezione), le operazioni possibili (controllo) e gli utenti e singoli profili (controllo).

Un **meccanismo di sicurezza** è un metodo (strumento/procedura) per garantire una politica di sicurezza. Data una politica che distingue le azioni sicure da quelle non sicure, i meccanismi di sicurezza devono prevenire, scoprire o recuperare da un attacco.

- **Prevenzione:** il meccanismo deve rendere impossibile l'attacco. Spesso sono pesanti e interferiscono con il sistema al punto da renderlo scomodo da usare. Per esempio la richiesta di una *password* come metodo di autenticazione.

- **Scoperta:** il meccanismo è in grado di scoprire che un attacco è in corso. È utile quando non è possibile prevenire l'attacco, ma può servire a valutare le misure preventive. Si utilizza in genere un monitoraggio delle risorse del sistema, cercando eventuali tracce di attacchi.
- **Recupero:** si può svolgere in due modi. Il primo è quello di fermare l'attacco e recuperare/ricostruire la situazione pre-attacco, per esempio attraverso copie di *backup*. La seconda è continuare a far funzionare il sistema correttamente durante l'attacco (*fault-tolerant*).

Meccanismi e livelli

▷ In quale livello del *computer* conviene inserire un determinato meccanismo?

- **Livelli bassi:** meccanismi generali, semplici, ma dimostrabili corretti.
- **Livelli alti:** meccanismi *ad hoc* per gli utenti, sofisticati e difficili da dimostrare corretti.

Alcuni esempi di meccanismi specifici di sicurezza, legati a un specifico livello OSI, sono:

- **Crittografia:** trasformazione dei dati in un formato non intellegibile.
- **Firma digitale e integrità dei dati:** usata per provare la sorgente e l'integrità di dati o messaggi.
- **Autenticazione e controllo degli accessi:** gestione dei diritti degli utenti rispetto le risorse.

I meccanismi generali riguardano invece il rilevamento degli eventi, la gestione degli *Audit*¹ e il *recovery*.

7.2 La crittografia

La **crittografia** è la scienza che si occupa di proteggere l'informazione rendendola sicura, in modo che un utente non autorizzato che ne entri in possesso non sia in grado di comprenderla. Di contro la **crittoanalisi** è la scienza che cerca di aggirare o superare le protezioni crittografiche, accedendo alle informazioni protette.

L'insieme composto dalla crittografia e dalla crittoanalisi è detto **crittologia**.

¹È un processo formale e sistematico di valutazione, esame e verifica di sistemi, processi, procedure, controlli e risorse per determinare la conformità, l'efficacia e l'adeguatezza rispetto a determinati *standard*, regolamenti o obiettivi prestabiliti.

7.2.1 Elementi del processo crittografico

L'**algoritmo crittografico** svolge un ruolo importante nella protezione dei dati attraverso la cifratura e la decifratura. Nel dettaglio è una funzione che prende in ingresso un messaggio e un parametro detto **chiave**, producendo in uscita un messaggio trasformato.

- **Cifratura:** l'algoritmo prende in ingresso un messaggio in chiaro (*plaintext* oppure *cleartext*), e attraverso una serie di operazioni matematiche, il messaggio viene trasformato in un messaggio cifrato (*ciphertext*) che appare non riconoscibile.
- **Decifratura:** l'algoritmo utilizza una chiave di decifratura, che può essere la stessa chiave di cifratura o una diversa, ed esegue una serie di operazioni applicate al messaggio cifrato per ottenere il messaggio in chiaro originale.

È importante distinguere i casi in cui le chiavi di cifratura e decifratura siano uguali o diverse:

- se sono uguali, allora l'algoritmo è detto **simmetrico** e la chiave deve essere segreta.
- se sono diverse, allora l'algoritmo è detto **asimmetrico** e si ha una chiave pubblica (non segreta) e una chiave privata (segreta).

Robustezza crittografica

La **robustezza crittografica** si riferisce alla capacità di un algoritmo crittografico di resistere agli attacchi e di proteggere efficacemente i dati cifrati.

- Non deve essere possibile (facilmente):
 - dato un testo cifrato ottenere il corrispondente testo in chiaro senza conoscere la chiave di decifratura;
 - dato un testo cifrato e il corrispondente testo in chiaro ottenere la chiave di decifratura.
- In generale nessun algoritmo crittografico è assolutamente sicuro, di conseguenza si afferma che sia **computazionalmente sicuro** se:
 - il costo necessario a violarlo è superiore al valore dell'informazione cifrata;
 - il tempo necessario a violarlo è superiore al tempo di vita utile dell'informazione cifrata.

Crittoanalisi

La **crittoanalisi** tenta di ricostruire il testo in chiaro senza conoscere la chiave di cifratura. L'attacco più banale è quello della “forza bruta”, ossia:

- tentare di decifrare il messaggio provando tutte le chiavi possibili;

- applicabile a qualunque algoritmo, ma la sua praticabilità dipende dal numero di chiavi possibili;
- è comunque necessario avere informazioni sul formato del testo in chiaro, per riconoscerlo quando si trova la chiave giusta.

Teorema 7.2.1: Principio di Kerckhoffs

Nel valutare la sicurezza di un algoritmo crittografico si assume che il crittoanalista conosca tutti i dettagli dell'algoritmo. La segretezza deve risiedere nella chiave e non nell'algoritmo.

7.2.2 Tipologie di crittografia

Crittografia a chiave simmetrica

La **crittografia simmetrica** (oppure **crittografia a chiave segreta**), utilizza una chiave comune e il medesimo algoritmo crittografico per la codifica e la decodifica dei messaggi. In pratica due utenti che desiderano comunicare devono accordarsi su di un algoritmo e su di una chiave comune; la chiave deve essere scambiata su un canale sicuro.

Esempio

Il cifrario più iconico di sempre appartenente a questa categoria è il **cifrario di Cesare**. Il funzionamento di questo consiste nel sostituire ogni lettera del testo in chiaro con quella che si trova a K posizioni più avanti nell'alfabeto. Quindi se si considera la chiave $K = 3$, si ottiene:

- In chiaro: A B C D E F G H I L M N O P Q R S T U V Z.
- Cifrate: D E F G H I L M N O P Q R S T U V Z A B C.
- Messaggio in chiaro/cifrato: CIAO/FNDR.

In questo cifrario le chiavi possibili sono soltanto 20.

Esempio

Il **cifrario monoalfabetico** agisce sostituendo ogni carattere da un altro (permutazione), secondo un certo alfabeto che costituisce la chiave.

- In chiaro: A B C D E F G H I L M N O P Q R S T U V Z.
- Cifrate: M Z N C B V L A H S G D F Q P E O R I T U.

- Messaggio in chiaro/cifrato: CIAO/NHMF.

In questo cifrario le chiavi sono pari al numero di permutazioni possibili, ossia $21! = 5.1 \cdot 10^{19}$.

L'**analisi delle frequenze** è una tecnica crittoanalitica utilizzata per decifrare testi cifrati, specialmente quando si tratta di cifrari monoalfabetici o cifrari a sostituzione. Questa tecnica si basa sull'idea che le lingue naturali seguano modelli prevedibili di frequenza delle lettere e delle combinazioni di lettere.

Ecco spiegato il funzionamento:

1. **Frequenza delle lettere:** in una specifica lingua, alcune lettere sono più comuni di altre. Gli analisi delle frequenze prendono un testo cifrato e contano quante volte appare ogni lettera.
2. **Creazione di un profilo di frequenza:** dopo aver contato le occorrenze di ciascuna lettera, si crea un “profilo di frequenza” delle lettere nel testo cifrato. Questo profilo rappresenta le frequenze relative di ciascuna lettera.
3. **Confronto con la frequenza tipica:** si confronta il profilo di frequenza del testo cifrato con le frequenze tipiche delle lettere nella lingua in cui si presume sia scritto il testo in chiaro. Per esempio se il profilo di frequenza mostra che la lettera più comune nel testo cifrato è “Q”, allora potrebbe essere una candidata per corrispondere alla “E”.
4. **Individuazione delle corrispondenze:** si cerca di individuare le corrispondenze tra le lettere più comuni nel testo cifrato e le lettere più comuni nella lingua in chiaro. Queste corrispondenze potrebbero rivelare alcune lettere della chiave di sostituzione.
5. **Sostituzione e congettura:** con le prime lettere della chiave di sostituzione ipotizzate, si cerca di decifrare altre parti del testo. Man mano che più lettere vengono decritte, diventa più facile identificare altre parole e porzioni di testo.

Cifrari a blocchi nella cifratura simmetrica

I **cifrari a blocchi** sono una classe di algoritmi crittografici simmetrici che operano su blocchi di dati con una dimensione specifica, solitamente espressa in *bit*. A differenza dei cifrari di flusso, che cifrano un *bit* alla volta o un flusso continuo di *bit*, i cifrari a blocchi trattano gruppi di *bit* come unità indipendenti.

Il suo funzionamento è definito dai seguenti passi:

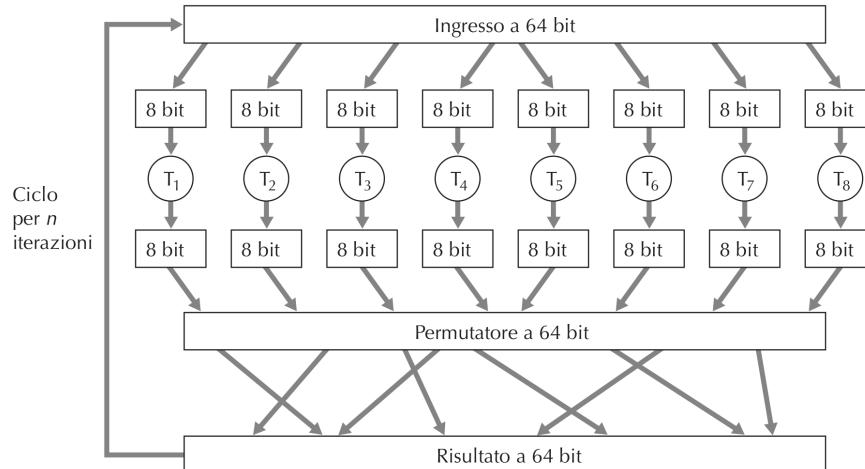
1. **Divisione in blocchi:** i dati in chiaro vengono divisi in blocchi di dimensioni fisse, come 64 bit oppure 128 bit.
2. **Cifratura del blocco:** ciascun blocco viene cifrato separatamente utilizzando l'algoritmo di cifratura a blocchi. L'algoritmo utilizza una chiave segreta per mescolare i *bit* del blocco in *input* e produrre un blocco cifrato.

3. **Concatenazione dei blocchi cifrati:** i blocchi cifrati vengono concatenati in sequenza per creare il testo cifrato complessivo. Ogni blocco cifrato è indipendente dagli altri blocchi e può essere decifrato separatamente.
4. **Decifratura:** per decifrare il messaggio, il processo è quello inverso. I blocchi cifrati vengono decifrati uno alla volta utilizzando la chiave segreta per ripristinare il testo in chiaro originale.

Dati k bit, i possibili 2^k ingressi vengono permutati. Con $k = 3$ si ottiene la seguente tabella:

Ingresso	Uscita
000	110
001	111
010	101
011	100
100	011
101	010
110	000
111	001

Inoltre le permutazioni possono essere combinate tra loro per creare schemi più complessi:



Esempio: DES (Data Encryption Standard)

È il più noto algoritmo crittografico simmetrico moderno, nato negli anni '70 a seguito di un progetto di IBM. È stato adottato ufficialmente nel '77 come *standard* dal governo americano. Utilizza chiavi di 56 bit, ormai da considerarsi obsoleto.

Esempio: Triplo-DES

Per aumentare la sicurezza del DES lo si applica tre volte con chiavi diverse. Esistono due varianti:

- con chiave da 112 bit (56×2);
- con chiave da 168 bit (56×3).

Esempio: AES (*Advanced Encryption Standard*)

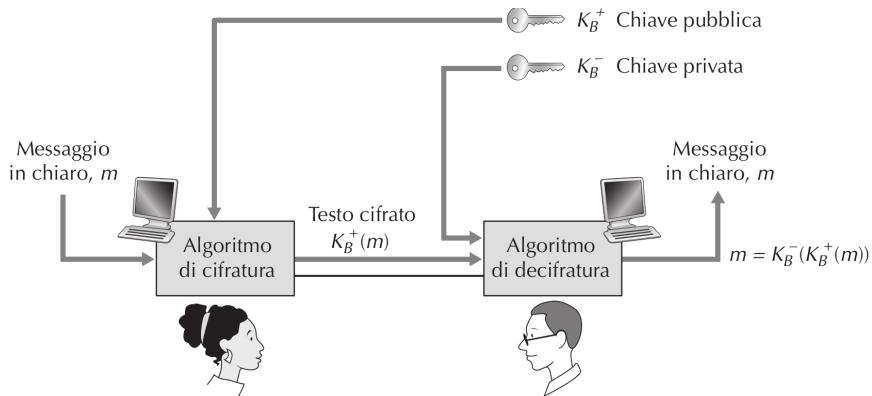
È un algoritmo simmetrico impiegato per sostituire DES e pian piano anche il triplo DES. Nel dettaglio può utilizzare chiavi di 128, 196 oppure 256 bit (rispettivamente AES-128, AES-192, AES-256).

Il problema degli algoritmi simmetrici sta nel fatto che la distribuzione della chiave deve avvenire su canali di comunicazione sicuri.

Crittografia asimmetrica

Nella crittografia asimmetrica ogni utente ha una coppia di chiavi, costituita da una **chiave pubblica** e una **chiave privata**. La prima viene resa nota al pubblico, mentre la seconda deve rimanere privata/secreta.

Il dato viene cifrato con la chiave pubblica del destinatario, il quale può decifrare il messaggio con la propria chiave privata.



I vantaggi della crittografia asimmetrica sono i seguenti:

- Non è più necessario incontrarsi per scambiare chiavi.

- La stessa chiave pubblica può essere usata da più utenti.

I requisiti necessari di questo approccio sono:

- Deve esser semplice la generazione di una coppia di chiavi pubblica/privata.
- Deve essere semplice l'operazione di cifratura e decifratura se si è a conoscenza della relativa chiave.
- Deve essere computazionalmente impraticabile ricavare la chiave privata da quella pubblica.
- Deve essere computazionalmente impraticabile ricavare il testo in chiaro avendo il testo cifrato e la chiave pubblica.

L'algoritmo RSA prende il nome dalle iniziali dei suoi inventori Rivest, Shamir e Adleman. Si basa sulla difficoltà di scomporre un numero in fattori primi.

La chiave RSA ha generalmente dimensioni di 2^{10} bit. Un attacco a forza bruta contro RSA consiste nel fattorizzare il prodotto di due numeri primi.

La cifratura e decifratura con RSA richiede la matematica a modulo. Scelti due numeri primi p, q si calcola:

- $n = p \cdot q;$
- $z = (p - 1) \cdot (q - 1);$
- un numero $1 < e < n$ relativamente primo a z ;
- un numero d tale che $(e \cdot d - 1)$ sia multiplo di z .

La chiave pubblica è (n, e) e per cifrare m si effettua $c = m^e \bmod n$. Invece la chiave privata è (n, d) e per decifrare c si esegue $m = c^d \bmod n$.

Esempio

Si considerano $p = 5$ e $q = 7$, allora si ottengono:

- $n = p \cdot q = 35;$
- $z = (p - 1) \cdot (q - 1) = 24;$
- $e = 5$ è relativamente primo a 24 (andavano bene anche 7, 9, 11, ...);
- $d = 29$ in quanto $5 \cdot 29 - 1 = 144$ che è multiplo di 24.

Il messaggio da cifrare è la parola “dove” e si suppone di rappresentare le lettere con numeri da 1 a 26.

Lettere in chiaro	m : rappresentazione numerica	m^e	Testo cifrato $c = m^e \text{ mod } n$
l	12	248832	17
0	15	759375	15
v	22	5153632	22
e	5	3125	10

Testo cifrato c	c^d	$m = c^d \text{ mod } n$	Lettere in chiaro
17	17^{29}	12	l
15	15^{29}	15	o
22	22^{29}	22	v
10	10^{29}	5	e

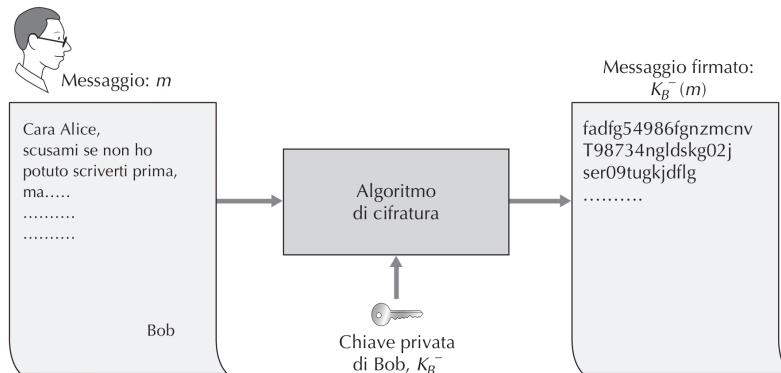
Alcune considerazioni finali sugli algoritmi asimmetrici sono:

- Richiedono molte risorse computazionali. Sono da 100 a 1000 volte più lenti degli algoritmi simmetrici.
- Vengono utilizzati per scambiarsi una chiave di sessione. Questa poi verrà utilizzata con un algoritmo simmetrico sicuro e computazionalmente più efficiente.
- Con RSA ciò che viene cifrato con la chiave pubblica si può decifrare con la chiave privata, ma vale anche l'inverso: ciò che è cifrato con la chiave privata si può decifrare con la chiave pubblica.

7.3 Meccanismi di autenticazione e autorizzazione

7.3.1 Firma digitale

Un **firma digitale** è l'equivalente informatico di una firma convenzionale, di conseguenza una firma di questo tipo non è in generale ripudiabile. Si sfrutta RSA in modo inverso a quanto fatto per cifrare, ossia l'algoritmo di cifratura diventa l'algoritmo di verifica e l'algoritmo di decifratura diventa l'algoritmo di firma.



A livello computazionale firmare l'intero documento, ovvero cifrarlo con la chiave privata, è molto oneroso e dunque l'approccio più efficiente è quello di combinare RSA con le funzioni *hash*.

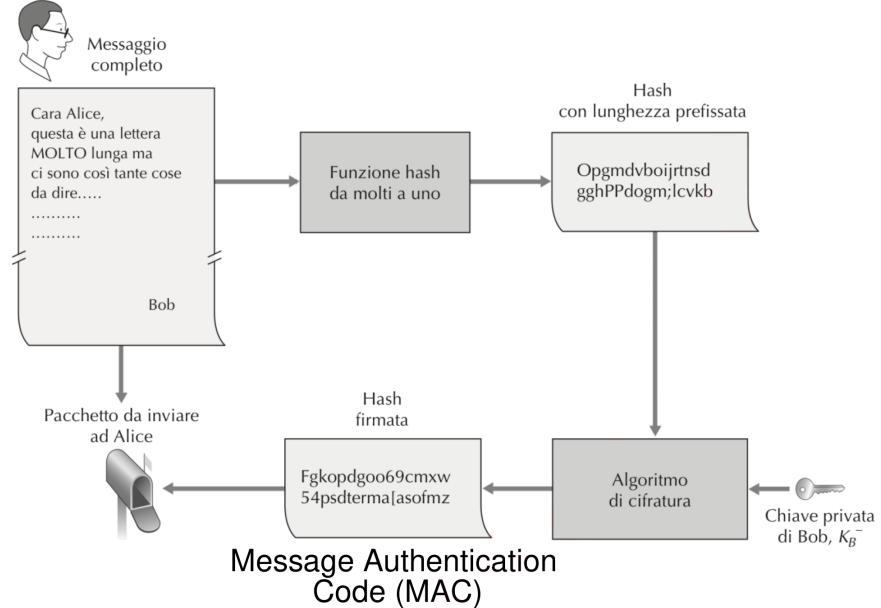


Figura 7.1: Firma di un documento tramite *hash*.

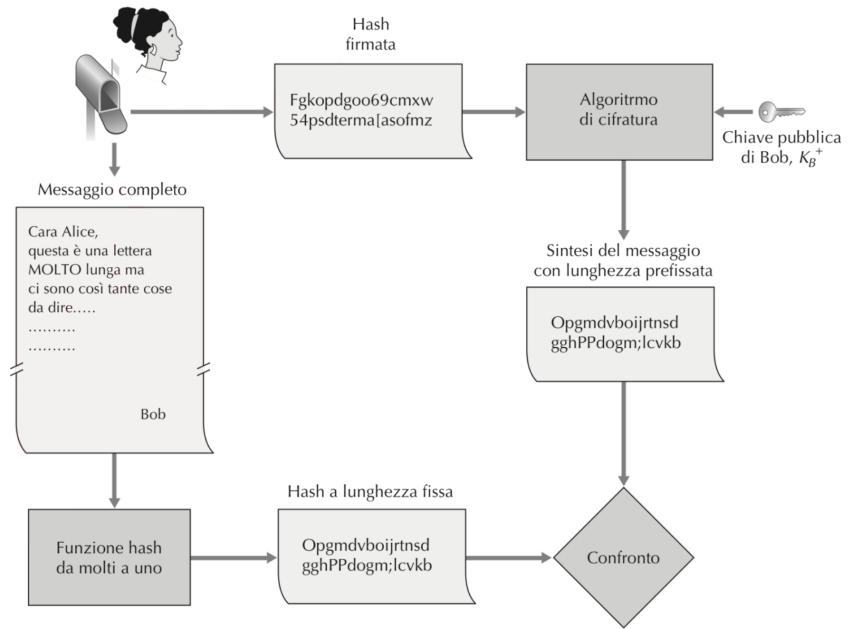


Figura 7.2: Controllo dell'integrità del messaggio firmato.

Per soddisfare le condizioni di sicurezza stabilite per le funzioni *hash*, gli algoritmi devono essere:

- **Coerenti**: *input* uguali corrispondono a *output* uguali.
- **Capaci di stravolgere la statistica dei simboli in ingresso**: per impedire l'interpretazione accidentale del messaggio originale.
- **Univoci**: la probabilità che due messaggi generino il medesimo *hash* deve essere virtualmente nulla.
- **Non invertibili**: risalire al messaggio originale dall'*output* deve essere impossibile.

Funzioni hash

Le funzioni *hash* non invertibili vengono normalmente utilizzate per assegnare “un’impronta digitale” a un messaggio o a un *file*.

- Un’impronta *hash* è univoca e costituisce una prova dell’integrità e dell’autenticità dei messaggi.
- Se *A* e *B* vogliono accertarsi che nessuno sia intervenuto sul contenuto del messaggio in fase di transizione, utilizzano proprio una funzione *hash* non invertibile.

Le funzioni *hash* più comuni sono quelle dell’algoritmo MD5 (*Message Digest 5*) e l’algoritmo SHA (*Secure Hash Algorithm*). Il sistema UNIX utilizza funzioni *hash* per gestire le *password*: piuttosto che memorizzare la *password* si memorizza la sua *hash*. Quando un utente effettua il *login*, UNIX esegue l’*hash* della *password* digitata e lo confronta con il valore in memoria.

Problemi e soluzioni per la firma digitale

Per quanto due utenti *A* e *B* possano scambiare un segreto, non è garantita l’**autenticità**. Infatti *A* potrebbe scambiare la chiave con uno sconosciuto che si spaccia per *B*.

Dunque è necessaria la certificazione della chiave pubblica. Utenti, *browser*, *router* e così via, devono avere la certezza che la chiave pubblica sia proprio quella del corrispondente.

- **Autorità di certificazione** (CA, *Certification Authority*). Convalida l’identità ed emette certificati.
- ITU (*International Telecommunication Union*) e IETF (*Internet Engineering Task Force*) hanno sviluppato *standard* per le autorità di certificazione.

Esempio

Il certificato X.509 è uno *standard* di formato per i certificati digitali utilizzati in crittografia asimmetrica. I certificati X.509 vengono comunemente utilizzati per garantire l’autenticità, l’integrità e la riservatezza delle comunicazioni su reti digitali come *Internet*.

I campi essenziali di un certificato X.509 sono:

Nome campo	Descrizione
Versione	Numero di versione della specifica X.509
Numero seriale	Identificatore unico del certificato fornito dalla CA
Firma	Specifica l'algoritmo utilizzato dalla CA per firmare il certificato
Nome dell'emittente	Identificativo della CA che rilascia il certificato, in formato DN [RFC 4515]
Periodo di validità	Inizio e fine del periodo di validità del certificato
Nome del soggetto	Identificativo dell'entità la cui chiave pubblica è associata al certificato (in formato DN)
Chiave pubblica del soggetto	Chiave pubblica del soggetto e indicazioni dell'algoritmo da utilizzare

Occorre un'infrastruttura per gestire i certificati:

- PKI: *Public Key Infrastructure*;
- gerarchia per gestire la mole di richieste.

7.3.2 Analisi dell'autenticazione

L'**autenticazione** è il servizio di sicurezza che permette di garantire l'**identità** degli interlocutori. Questi ultimi possono essere utenti oppure *computer*:

- *computer-computer* (stampa in rete, delega, ...);
- utente-utente (protocolli di sicurezza, ...);
- *computer-utente* (autenticare un *server web*, ...);
- utente-*computer* (per accedere a un sistema, ...).

Le tipologie di autenticazione si possono distinguere in quattro categorie di sistemi di autenticazione:

- **Locale**: l'utente accede in locale al servizio che effettua l'autenticazione;
- **Remota/diretta**: l'utente accede da remoto al servizio che effettua direttamente l'autenticazione;
- **Remota/indiretta**: l'utente accede da remoto a diversi servizi, i quali si appoggiano su un servizio di autenticazione separato (per esempio RADIUS, Kerberos);
- **Off-line**: i servizi possono prendere decisioni autonome anche senza dover contattare ogni volta l'autorità di autenticazione (per esempio PKI).

Modalità di fattori di autenticazione

I fattori di autenticazione sono basati su qualcosa che l'utente:

- **conosce**: una *password* oppure un PIN;
- **possiede**: elementi fisici o elettronici, come chiavi convenzionali, carte magnetiche o *smart card*;
- “**qualcosa che si è**”: sono le caratteristiche biometriche come l'impronte digitali, dell'iride o il tono di voce.

Per ottenere un'autenticazione più forte si possono combinare i diversi fattori appena elencati.

Generalmente il metodo più semplice per autenticarsi è quello basato su ***username*** e ***password***: l'utente inserisce un nome che lo identifica (*username*), solitamente non segreto, e una parola segreta (*password*). I vantaggi consistono nella semplicità d'uso per l'utente, è economico e non richiede di immagazzinare un segreto nel lato *client*. Mentre gli svantaggi sono la scelta di *password* deboli oppure che i metodi di autenticazione basati su *password* siano deboli.

Possibili attacchi alle *password* possono essere l'**intercettazione** se la *password* è trasmessa in chiaro, oppure i ***guessing/cracking*** come un attacco *bruteforce* o più spesso un “attacco a dizionario” (si provano parole di senso compiuto o le loro minime variazioni).

Una *password* dovrebbe essere abbastanza lunga, non essere una parola di senso compiuto e cambiata frequentemente.

Alcuni sistemi invece generano una nuova *password* a ogni accesso da parte dell'utente per risolvere il problema dell'intercettazione. Queste prendono il nome di ***one-time password*** e vengono generate sulla base di un contatore (esiste di conseguenza una sequenza di *password* successive) o più spesso sulla base dell'istante temporale. I sistemi *one-time password* si appoggiano su:

- ***token***: dispositivi *hardware* che forniscono all'utente la *password* da inserire;
- **SMS**: inviato sul dispositivo personale dell'utente.

Questi *token* appartengono alla categoria dell'autenticazione basata sul **possesso**, poiché fornisce una prova dell'identità. Ovviamente l'autenticazione dimostra solo l'identità del *token* e non quella dell'utente, ecco perché di solito si combina il possesso e la conoscenza; per esempio il bancomat richiede la carta e il PIN.

Il possesso di caratteristiche univoche forniscono una prova dell'identità:

- **Fisiche**: impronte digitali, forma della mano, impronta della retina o del viso, ...
- **Comportamentali**: firma, timbro di voce, scrittura, *keystroke dynamic*, ...

I punti deboli possono essere:

- l'autenticazione si basa su una misura e un confronto con un *template*. Le misure possono essere imprecise anche quando viene creato il *template* oppure ci possono essere dei falsi positivi e dei falsi negativi;
- non sostituibili se compromesse, come impronti digitali falsificate.

Autenticazione remota/diretta

Le tipologie di autenticazione viste fino adesso si possono utilizzare senza modifiche per l'autenticazione locale. Se l'autenticazione avviene da **remoto** (autenticazione diretta) sorgono altri problemi:

- Un intruso potrebbe registrare e replicare le informazioni.

La soluzione consiste nell'eseguire l'autenticazione su un canale sicuro oppure si aggiunge la cosiddetta sfida².

Osservazione

La procedura che garantisce l'integrità di un messaggio, ma non garantisce sull'autenticità del mittente. L'attacco **replay** è un tipo di attacco informatico in cui un attaccante registra una serie di comunicazioni tra due parti legittime e successivamente riproduce o “*replay*” queste comunicazioni registrate per ottenere accesso non autorizzato o causare un impatto indesiderato

Autenticazione remota/indiretta

Quando molti sistemi/applicazioni condividono gli stessi utenti, allora si ricorre a sistemi di **autenticazione indiretta**. Le informazioni sugli utenti vengono centralizzate su sistema di autenticazione e gli altri sistemi si appoggiano su di esso.

Due esempi di sistemi di autenticazione indiretta sono:

- **RADIUS** (*Remote Authentication Dial In User Service*), nato per l'accesso remoto *dial-up*.
- **Kerberos**, usato per l'autenticazione e il *single sign-on* (SSO) tra applicazioni all'interno di un “dominio” amministrativo (per esempio all'interno di un'azienda).

Autenticazione off-line

Questa tipologia di autenticazione è basata sui certificati, i quali sono emessi da un'**autorità di certificazione** (CA) e distribuiti da un'infrastruttura a chiave pubblica (*Public Key Infrastructure*, PKI).

²È una domanda o una richiesta che viene inviata all'entità che sta cercando di autenticarsi

Differenze tra:

Certificato reale	Certificato digitale
Cartaceo	Elettronico
Emesso da un'autorità riconosciuta	Emesso da una CA riconosciuta
Associa l'identità di una persona al suo aspetto fisico	Associa l'identità di una persona a una chiave pubblica

7.3.3 Compiti della CA

La CA deve svolgere i seguenti compiti:

1. Identificare con certezza la persona che fa richiesta della certificazione della chiave pubblica.
2. Rilasciare e rendere pubblico il certificato.
3. Garantire l'accesso telematico al registro delle chiavi pubbliche.
4. Informare i richiedenti sulla procedura di certificazione e sulle tecniche per accedervi.
5. Dichiarare la propria politica di sicurezza.
6. Attenersi alle norme sul trattamento di dati personali.
7. Non rendersi depositario delle chiavi private.
8. Procedere alla revoca o alla sospensione dei certificati in caso di richiesta dell'interessato oppure venendo a conoscenza di abusi o falsificazioni, ecc.
9. Rendere pubblica la revoca o la sospensione delle chiavi.
10. Assicurare la corretta manutenzione del sistema di certificazione.

Quindi l'utente genera sul proprio PC una coppia di chiavi³ (i browser offrono il servizio). La chiave privata è memorizzata localmente in un file nascosto. L'utente invia alla CA una richiesta di certificato, insieme alla chiave pubblica generata (a meno che non si alla CA a generare la coppia di chiavi per l'utente). A questo punto la CA autentica il richiedente, di solito chiedendogli di recarsi di persona a uno sportello di LVP (*Local Validation Point*) collegato con la CA. Una volta verificata l'identità, la CA emette il certificato, lo invia al richiedente tramite posta elettronica e inserisce la chiave certificata nel registro delle chiavi pubbliche.

L'intera procedura accade nell'ambito di una PKI (*Public Key Infrastructure*).

³Generare la coppia di chiavi tramite *SmartCard* collegata al PC garantisce una maggiore sicurezza in quanto la chiave privata non esce mai dalla *SmartCard* protetta da PIN.

La struttura minima è CA più LVP:

- Sono ammesse più LVP.
- LVP è uno sportello per l'autenticazione classica dell'utente.
- LVPO è il suo operatore.

La struttura gerarchica permette ad alcune CA di certificare altre, ottenendo una “catena di fiducia”:

- struttura ad albero;
- la *root* CA certifica le CA di primo livello;
- il primo livello certifica le CA di secondo livello;
- le CA di ultimo livello certificano il singolo utente.

Utilizzo e problemi dei certificati

Quindi un utente *A* invia a un utente *B* il proprio certificato, firmato dalla CA. L'utente *B* verifica la firma della CA sul certificato dell'utente *A* e se è corretta, allora estrae la chiave pubblica di *A* dal certificato:

- l'utente *B* deve già avere il certificato della CA per poterne verificare la firma;
- il certificato della CA è autofirmato.

A questo punto l'utente *B* ha ottenuto la chiave pubblica dell'utente *A*, la cui identità è garantita dalla CA.

Tuttavia è necessario ottenere in qualche modo sicuro il certificato della CA. Il problema della distribuzione delle chiavi pubbliche rimane, ma su scala molto più ridotta. Un certificato può essere revocato, per esempio se il proprietario si accorge del furto della chiave privata corrispondente:

- la verifica della firma della CA su un certificato revocato va a buon fine;
- la CA pubblica una lista dei certificati revocati, da essa firmata, che andrebbe controllata per accertarsi della validità di un certificato.

7.3.4 Autorizzazione (controllo degli accessi)

Il servizio di **controllo dell'accesso** (detto anche **autorizzazione**) garantisce che l'accesso alle risorse limitato ai soli utenti che ne hanno il diritto. Soggetti diversi possono avere diritto a diverse modalità di interazione con le risorse. I soggetti sono in possesso di privilegi sugli oggetti in accordo con le politiche definite sul sistema:

- **Soggetti**: utenti, applicazioni, altri sistemi . . .

- **Privilegi:** lettura, scrittura, esecuzione, proprietà.
- **Oggetti:** file, funzioni, applicazioni, altri sistemi ...

Le **politiche** di controllo dell'accesso definiscono l'attribuzione di privilegi d'accesso dei soggetti agli oggetti. I **meccanismi** di controllo dell'accesso specificano come le relazioni tra i soggetti e gli oggetti (i privilegi) sono rappresentate. Sono utili i seguenti due principi:

- **Privilegio minimo:** a un soggetto dovrebbero essere concessi solo i privilegi minimi necessari a compiere l'azione che deve compiere.
- **Separazione dei compiti:** nessun soggetto dovrebbe avere abbastanza potere per sovvertire il sistema.

Meccanismi di controllo dell'accesso

La **matrice di controllo dell'accesso** è composta da righe che contengono i soggetti, colonne che possiedono gli oggetti e nelle caselle sono rappresentati i permessi.

	<i>File 1</i>	<i>File 2</i>	<i>Programma 1</i>	<i>Programma 2</i>
Alice	rwx	rwx, own	x	rwx
Bob	rwx, own	r	x	rwx
Programma 1	rw	rw	-	x

Soggetti e oggetti possono essere numerosi e sorgono problemi di scalabilità:

- si può memorizzare la matrice per righe/colonne;
- si possono effettuare raggruppamenti per gestire i privilegi in modo omogeneo (gruppi/ruoli).

Access Control List (ACL)

Si memorizza la matrice di controllo dell'accesso per colonne. Ciascuna risorsa viene memorizzata con la lista dei soggetti che possono interagire con questa assieme ai relativi permessi (per esempio il *filesystem* Unix).

Sono adatte a contesti in cui la protezione è **orientata ai dati**:

- È semplice gestire i permessi associati a un oggetto.
- Non sono adatte se si vogliono gestire centralmente i permessi di ciascun soggetto e/o se si vogliono introdurre meccanismi di delega temporanea.

Capabilities

Si memorizza la matrice di controllo dell'accesso per righe. A ciascun soggetto è associato l'insieme degli oggetti con cui si può interagire, si memorizza la lista di relazioni che il soggetto ha con gli oggetti e i relativi permessi. È concessa la gestione in modo efficiente dei permessi associati a un singolo utente; rende semplice anche un meccanismo di delega temporanea. Per determinare tutti i soggetti che hanno diritto di interagire con un oggetto si deve scorrere la lista.

Politiche di controllo dell'accesso

Si distinguono due diversi approcci:

- **DAC** (*Discretionary Access Control*): i singoli utenti possono a loro discrezione concedere e revocare permessi su oggetti che sono sotto il loro controllo. In genere ci si basa sul concetto di proprietà (*ownership*): ogni oggetto ha un proprietario, cioè il soggetto che ne definisce i diritti di accesso. Eventualmente il proprietario può assegnare la proprietà a un altro soggetto.
 - DAC è un modello flessibile, utilizzabile in molti ambiti, come i sistemi operativi Unix, Windows, ecc.
 - DAC non permette di controllare la diffusione dell'informazione. Chi ha i permessi in lettura su un *file* potrebbe inviarlo a chi non ha permessi.
- **MAC** (*Mandatory Access Control*): la politica di controllo dell'accesso è determinata centralmente dal sistema e non dai singoli utenti. Viene utilizzato per ambiti militari, basandosi su una classificazione degli oggetti e dei soggetti. Per esempio nel modello di Bell-LaPadula il sistema forza il rispetto delle seguenti regole:
 - **No read up**: non è possibile leggere informazioni classificate a livelli più alti del proprio.
 - **No write down**: non è possibile scrivere informazioni classificate a livelli più bassi del proprio.

In generale è meno flessibile, ma più robusto del modello DAC.

Gli approcci possono essere combinati con una suddivisione degli utenti in gruppi e ruoli. Un **gruppo** è una lista di soggetti, mentre un **ruolo** è un insieme prefissato di permessi di accesso che uno o più soggetti possono acquisire per un certo periodo di tempo, spesso corrispondente a una funzione all'interno di un'organizzazione. Inoltre i gruppi permettono di gestire insiemi di soggetti/oggetti in modo omogeneo:

- L'accesso alle risorse è basato sui permessi dei gruppi.
- Diversi gruppi possono avere proprietà sovrapposte.
- Modificare i diritti di un gruppo permette di cambiare direttamente quelli di tutte le entità appartenenti.

Mentre i ruoli definiscono insiemi di proprietà e responsabilità solitamente associate alla struttura organizzativa a cui fa capo il sistema (RBAC, *Role Based Access Control*).

7.4 Firewall e IDS

7.4.1 I firewall

I *firewall* di rete sono apparecchiature o sistemi che controllano il flusso del traffico tra due reti con differenti livelli di sicurezza:

- prevenire accessi non autorizzati alla rete privata;
- prevenire l'esportazione di dati dall'interno verso l'esterno;
- schermare alcune reti interne e nasconderle agli altri;
- bloccare alcuni accessi a servizi o a utenti;
- monitorare tramite *log function*.

Tuttavia si deve assumere che gli attacchi avvengano dall'esterno. Il *firewall* non difende contro nuovi bachi non ancora documentati nei protocolli. I filtri sono difficili da impostare e da mantenere, perché è difficile trovare il compromesso tra libertà e sicurezza. Infine può degradare le *performance* della rete.

Esistono due filosofie per quanto riguarda il *firewall*:

- ***Default deny***: tutto ciò che non è espressamente ammesso viene proibito. I servizi sono abilitati caso per caso dopo un'attenta analisi e gli utenti sono limitati.
- ***Default permit***: tutto quello che non è espressamente proibito viene ammesso. *System administrator* deve reagire prontamente ogni volta che un nuovo baco su un protocollo viene scoperto. I servizi sono rimossi/ridotti quando vengono scoperti pericolosi e gli utenti sono meno ristretti.

I *firewall* si suddividono in tre tipologie:

- *Packet-Filtering router* (1^a generazione);
- *Stateful Inspection* (2^a generazione);
- *Gateway* a livello di applicazione (o *Proxy Server*) (3^a generazione).

Filtri a livello 3

- *Source address* del pacchetto (IP address);
- *Destination address* del pacchetto (IP address);
- Tipo del traffico (IP, ICMP, IPX se a livello 3, o anche protocolli di livello 2);

- Possibilmente alcune caratteristiche del livello 4 (porta sorgente e destinazione);
- Talvolta informazioni interne al *router* (quali informazioni circa le interfacce sorgente e di destinazione del pacchetto, utile per *router* con più interfacce).

I vantaggi sono quelli di essere disponibile in molti *router*, il costo d'acquisto è contenuto, la trasparenza (ossia non lavora a livello applicativo, perciò non ostacola il normale utilizzo della rete) e la velocità (effettua meno controlli). Tuttavia sono presenti diverse limitazioni: le regole sono difficili da configurare e può avere bachi (più frequenti nel *packet filtering* rispetto al *proxying*).

Stateful Packet Filtering

Quando viene stabilita una connessione, se le regole di filtraggio non la bloccano, allora le informazioni relative a essa diventano *entry* di una tabella di stato. I pacchetti successivi in ingresso vengono valutati in base all'appartenenza a una delle connessioni consentite presenti nella tabella. Quando la connessione è conclusa, la *entry* nella tabella viene cancellata, per evitare che questa si riempia completamente.

Di seguito sono elencate le informazioni riguardanti la connessione che vengono memorizzate:

- Identificatore univoco del collegamento di sessione.
- Stato di connessione (*handshaking* se siamo in fase iniziale ovvero dove si raccolgono info e si mettono in tabella stato, *established*, *closing*).
- Informazioni di sequenzialità dei pacchetti.
- Indirizzi IP dell'*host* sorgente e di destinazione
- Interfacce di rete utilizzate.

<i>Source address</i>	<i>Source port</i>	<i>Dest. Address</i>	<i>Dest. Port</i>	<i>Connection state</i>
192.168.0.199	1051	192.168.1.10	80	<i>Handshaking</i>
192.168.0.212	1109	192.168.1.23	25	<i>Closing</i>
192.168.3.105	1212	192.168.0.111	80	<i>Established</i>

I vantaggi riguardano tutti quelli che possiede il *packet filtering* (essendone una evoluzione ne ereditano tutti i fattori positivi), dopodiché si ha un buon rapporto tra prestazioni e sicurezza (tipologia *firewall* con più alte *performance*, perché è quella che effettua meno controlli di connessione) e infine la protezione da IP *spoofing* (il controllo non si limita al singolo IP o alla porta, è molto più difficile aggirare il *firewall*). Mentre le limitazioni riguardano la mancanza di servizi aggiuntivi (non potendo agire a livello di applicazione non sono disponibili servizi come la gestione delle autenticazioni) e il *testing* complesso (verificare la corretta configurazione del *firewall* non è facile).

Filtri a livello 7

Routing tra le due interfacce effettuato a livello di applicazione dal *software* del *firewall*; in caso di malfunzionamento del *software*, il *routing* viene disabilitato. È possibile l'autenticazione tramite:

- *User Id e password*;
- *HW/SW token authentication*;
- *Biometric authentication*.

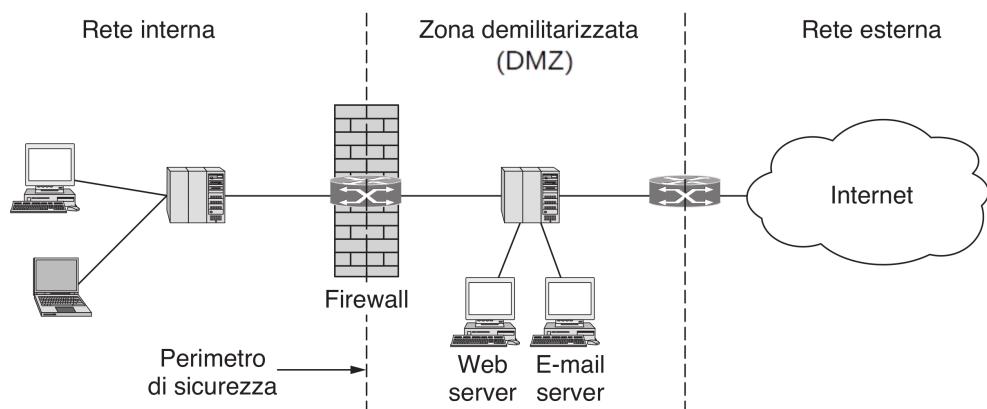
I filtri sono specifici su comandi, come per esempio permettere GET ma non PUT.

I vantaggi stanno nella maggiore sicurezza dei *packet filters*, si deve solo controllare un numero limitato di applicazioni (http, ftp, posta) ed è facile il *log* e il controllo del traffico. Invece gli svantaggi stanno nel *processing overhead* su ogni connessione e può solo controllare un numero limitato di applicazioni.

Proxy server dedicati e firewall personali

I *proxy server* dedicati sono specifici per ogni applicazione, aiutano l'*application proxy gateway* nel lavoro di *contents inspection* e il tipico utilizzo consiste in: antivirus, *malicious code* (applets java, activix, javascript, word) e sono usati spesso per *outbound connections* (*web cache proxy* ed *email proxy*).

I *firewall* personali proteggono solo la macchina dove sono installate ed è necessario specialmente per *mobile users*.



7.4.2 Sistema di rilevamento delle intrusioni (IDS)

Strumento *software* o *hardware* che automatizza il processo di monitoraggio impiegato per individuare eventi che rappresentano intrusioni non autorizzate ai *computer* e alle reti locali. Si può fare a livello di *host* (HIDS) e a livello di *network* (NIDS).

Un IDS può essere considerato come un antifurto, cioè cerca di rilevare eventuali intrusioni. Diversamente un *firewall* può essere considerato come una porta blindata, ovvero serve per bloccare le eventuali intrusioni.

Un IDS perfetto dovrebbe riuscire a individuare tutte le reali intrusioni. Tuttavia i principali problemi sono i falsi positivi (IDS rileva un'anomalia, ma non è successo niente) e falsi negativi (IDS non rileva un'intrusione avvenuta). I requisiti per un IDS reale sono:

- Scoprire un'ampia gamma di intrusioni, sia già note che non note.
- Scoprire le intrusioni velocemente, ma non necessariamente in tempo reale.
- Presentare i *report* delle analisi in formato semplice e facilmente comprensibile.
- Essere accurato nell'analisi.

I principi di base sono saper distinguere situazioni normali da quelle anomale. L'utente in condizioni normali si comporta in modo più o meno prevedibile, non compie azioni atte a violare la sicurezza e i propri accessi compiono solo azioni permesse.

I modelli per l'IDS sono i seguenti:

- ***Detection* di anomalie:** sequenze di azioni non usuali possono essere intrusioni.
- ***Detection* di uso malevolo:** si conosce quali sequenze di azioni possono essere intrusioni.
- ***Detection* in base a specifiche:** si conoscono le situazioni derivanti da intrusioni.
- I modelli possono essere statici o adattivi.

Detection di anomalie

Si analizzano insiemi di caratteristiche del sistema confrontando i valori con quelli attesi e segnalando quando le statistiche non sono paragonabili a quelle attese.

- **Metriche a soglia:** si conta il numero di volte che un evento si presenta, aspettandosi tra m e n occorrenze e se il numero cade al di fuori allora c'è un'anomalia. Un esempio pratico è su Windows il blocco dopo k tentativi di *login* falliti, dove appunto il *range* di tentativi disponibili sono $[0, k - 1]$ e dopo k -esima volta diventa sospetto. Le problematiche riguardano la difficoltà nel trovare un intervallo corretto, talvolta si possono creare situazioni in cui l'intervallo diventa più grande.
- **Momenti statistici:** l'analizzatore calcola la deviazione *standard* (i primi due momenti) o altre misure di correlazione (momenti di ordine superiore). Se i valori misurati di un certo momento cadono al di fuori di un certo intervallo vi è un'anomalia. Le problematiche in questo caso consistono nell'evoluzione dei profili nel tempo, ovvero si possono pesare opportunamente i dati o alterare le regole di *detection*.

- **Modelli di Markov:** l'ipotesi è che la storia passata influenzi la prossima transazione di stato. Le anomalie sono riconosciute da sequenze di eventi e non sulle occorrenze di singoli eventi. Il sistema deve essere addestrato per riconoscere le sequenze valide:

- l'addestramento è svolto con utenti non anomali;
- l'addestramento produce migliori risultati con una quantità maggiore di dati;
- i dati dovrebbero coprire tutte le sequenze normali del sistema.

Detection di uso malevolo

Si controlla se una sequenza di istruzioni da eseguire è già nota per essere potenzialmente dannosa per la sicurezza del sistema. La conoscenza è rappresentata mediante regole e il sistema controlla se la sequenza soddisfa una di queste regole. Non si possono scoprire intrusioni non note precedentemente.

Detection in base a specifiche

Si determina se una sequenza di azioni viola una specifica di come un programma o un sistema dovrebbe funzionare.

7.4.3 Architettura di un IDS

È essenzialmente un sistema di *auditing* sofisticato in cui sono presenti tre attori principali:

- **Agente:** ottiene le informazioni e le invia al direttore. Può mettere le informazioni in altre forme (*preprocessing* dei *record* per estrarre parti rilevanti). Può cancellare informazioni non necessarie. Il direttore può richiedere all'agente ulteriori informazioni. Infine si distinguono in agenti *host* e agenti *network*.
- **Direttore:** colleziona le informazioni inviate dagli agenti (elimina i *record* ridondanti o non necessari). Analizza le informazioni rimanenti per determinare se si è sotto attacco. Infine viene eseguito su un sistema operativo (non influenza le *performance* dei sistemi monitorati).
- **Notificatore:** ottiene i risultati e le informazioni dal direttore. Prende le decisioni appropriate: notificare messaggi agli amministratori, riconfigurare gli agenti e rispondere all'attacco.

Combining Sourcers DIDS

I monitoraggi di *host* e di *network* non sono generalmente sufficienti da soli a scoprire alcuni tipi di attacchi:

- un attaccante prova a fare *telnet* con vari *login*: gli IDS di rete lo possono scoprire, ma non gli IDS di *host*;

- l'attaccante prova a entrare senza la *password*: gli IDS di *host* lo rilevano, ma non quelli di rete.

DIDS utilizza gli agenti sugli *host* da monitorare e un *monitor* di rete.

Risposta alle intrusioni

Prevenzione: l'attacco deve essere scoperto prima del completamento. Una tecnica è il *Jailing*: far credere all'attaccante che l'intrusione è andata a buon fine, ma confinare le sue azioni in un dominio in cui non può fare danni (o causarne pochi): far scaricare *file* corrotti o falsi e imitare il sistema vero.

Introduzione alla programmazione web

8.1 Modello client/server

Il *client* potrebbe essere una qualsiasi applicazione, ma al momento si considera un semplice *browser*. I passaggi di una tipica connessione a una pagina *web* sono:

1. il *browser* invia la richiesta;
2. il *server* elabora la risposta eseguendo *script* e programmi con un tipico linguaggio “lato *server*”: PHP, Java, Microsoft .NET, Python, Node.js, ecc.;
3. alla fine dell'esecuzione lato *server*, è stata prodotta una risposta composta da tutto ciò che un *browser* può interpretare ed eseguire: HTML, CSS, Javascript e risorse varie;
4. il *server* invia la risposta al *browser* che la interpreta e ne mostra i risultati all'utente. Il codice JavaScript al suo interno rappresenta l'intelligenza attiva di quella pagina, ciò che gestisce l'interazione utente.

Ogni volta che una pagina ha bisogno di richiedere elaborazioni o dati presenti sul *server* deve necessariamente avviare una nuova richiesta.

8.1.1 AJAX

AJAX è l'abbreviazione di ***Asynchronous JavaScript and XML*** e indica una combinazione di tecnologie di sviluppo usate per creare pagine *web* dal contenuto dinamico.

Nello specifico la maggior parte delle implementazioni AJAX usa l'oggetto XMLHttpRequest, il quale include un elenco di funzioni di richiesta verso il *server web* che possono essere richiamate all'interno del codice JavaScript e di conseguenza dentro la pagina *web*.

Ciò che caratterizza AJAX è che gli *script* vengono eseguiti lato *client*, consentendo di visualizzare immediatamente sulla pagina i dati ricevuti dal *server*, senza la necessità di ricaricare la pagina stessa per visualizzarne i contenuti.

8.1.2 JSON

JSON (*JavaScript Object Notation*) è un semplice formato di testo per lo scambio di dati, completamente indipendente dal linguaggio di programmazione, che utilizza convenzioni presenti in molti linguaggi e questo semplifica gli algoritmi di *parsing* automatico. JSON è basato su due strutture:

- un insieme di coppie nome/valore. In diversi linguaggi, questo è realizzato come un oggetto, *record*, *struct*, dizionario, *array* associativo e via così, a seconda del linguaggio utilizzato;
- un elenco ordinato di valori. Nella maggior parte dei linguaggi questo si realizza come un *array*.

Il formato di scambio dati viene definito nell'*header* della richiesta HTTP: `Accept: application/json`.

8.2 Strumenti del browser

I *browser* come Mozilla Firefox e Google Chrome mettono a disposizione una serie di strumenti utili per lo sviluppo *web*. Gli strumenti interessati in questa sezione sono:



- Il primo pulsante a sinistra permette di selezionare un elemento della pagina caricata.
- **Analisi pagina:** permette di visualizzare e modificare il codice HTML e CSS.
- **Console:** contiene le informazioni di *log* associate alle pagine *web* (richieste di rete, errori/*warning*, ...) e altri messaggi registrati durante l'esecuzione del codice JavaScript. Permette inoltre di interagire con la pagina *web* eseguendo espressioni JavaScript nel contesto della pagina (per esempio se nel codice JavaScript si esegue `console.log("messaggio")`, si può vedere questo messaggio nella *console* del *browser*).
- **Archiviazione:** permette di ispezionare vari tipi di archiviazione che la pagina *web* può utilizzare.
- **Rete:** mostra tutte le richieste HTTP che vengono effettuate con i relativi dettagli.
- Il terzo pulsante a partire da destra permette di testare la pagina *web* per dispositivi con diverse dimensioni (*tablet/smartphone*).

WePlant

9.1 Configurazione su Linux

La configurazione di WePlant prevede due requisiti fondamentali: docker e docker-compose. La configurazione segue i seguenti passi (ignorare eventuali *warnings*):

1. Aprire un terminale, posizionarsi nella cartella di lavoro preferita e clonare la repository `we-plant-api-es`:

```
1 git clone https://github.com/DavideTonin99/we-plant-api-es
```

2. Entrare nella cartella `docker` tramite:

```
1 cd we-plant-api-es/docker
```

3. Modificare il file `.yml` inserendo alla terzultima riga una *email*: `email@email.com`.

4. Eseguire il `docker compose` digitando:

```
1 docker compose up -d
```

5. Aprire un terminale all'interno del `container we_plant_api`. Per farlo si ottiene prima il codice univoco del `container` digitando:

```
1 docker ps
```

6. Una volta copiato l'`id` del `container`, si avvia il terminale dentro il `container` digitando nel terminale Linux:

```
1 docker exec -it id_container bash
```

7. Nel terminale del `container` eseguire il comando:

```
1 yarn install
```

8. Nel terminale del *container* eseguire il comando:

```
1 mvn clean install -DskipTests
```

Dopodiché si esegue la seconda installazione:

1. Aprire un terminale, posizionarsi nella cartella di lavoro preferita e clonare la *repository* `we-plant-app-es`:

```
1 git clone https://github.com/DavideTonin99/we-plant-app-es
```

2. Entrare nella cartella *docker* tramite:

```
1 cd we-plant-app-es/docker
```

3. Eseguire il *docker compose* digitando:

```
1 docker-compose up -d
```

4. Aprire un terminale all'interno del container `we_plant_app`. Per farlo si ottiene prima il codice univoco del container digitando:

```
1 docker ps
```

5. Una volta copiato l'id del *container*, si avvia il terminale dentro il container digitando nel terminale Linux:

```
1 docker exec -it id_container bash
```

6. Nel terminale del *container* eseguire il comando:

```
1 npm install
```

7. Nel terminale del *container* eseguire il comando:

```
1 npm install -g @ionic/cli
```

Per eseguire `we-plant-api-es` si scrivono i seguenti comandi:

1. Aprire un terminale all'interno del *container* `we_plant_api`. Per farlo si ottiene prima il codice univoco del container digitando:

```
1 docker ps
```

2. Una volta copiato l'id del container, si avvia il terminale dentro il container digitando nel terminale Linux:

```
1 docker exec -it id_container bash
```

3. Nel terminale del container eseguire il comando:

```
1 mvn -DskipTests
```

4. Aprire un altro terminale Linux, aprire un terminale del container (eseguendo il primo punto) e digitare:

```
1 yarn start
```

Per eseguire `we-plant-app-es` si scrivono i seguenti comandi:

1. Aprire un terminale all'interno del *container we_plant_app*. Per farlo si ottiene prima il codice univoco del container digitando:

```
1 docker ps
```

2. Una volta copiato l'id del *container*, si avvia il terminale dentro il *container* digitando nel terminale Linux:

```
1 docker exec -it id_container bash
```

3. Nel terminale del *container* eseguire il comando:

```
1 npm run start
```

9.2 Struttura dell'applicazione

L'applicazione WePlant è stata sviluppata utilizzando JHipster: piattaforma di sviluppo per generare, sviluppare e distribuire rapidamente applicazioni *Web* e microservizi. È un generatore di applicazioni, gratuito e *open-source*. Il suo nome deriva da “Java Hipster” poiché l'obiettivo iniziale degli sviluppatori era creare una piattaforma che potesse fornire sia strumenti moderni che strumenti in “*hype*”. Al giorno d'oggi è utilizzato da aziende di grandi dimensioni come: Google, Adobe, HBO, Bosch e tanti altri.

JHipster supporta molte tecnologie di *frontend* e di *backend*. Quindi gli sviluppatori hanno avuto molta libertà sugli strumenti. L'applicazione si struttura nel seguente modo:

- **Lato server** è stato utilizzato come DBMS PostgreSQL e come linguaggio per sviluppare il *backend* Java insieme a qualche *framework*: Spring e Spring Boot;
- **Lato client** è stato utilizzato Angular con JHipster, pertanto un codice autogenereato, e inoltre è stato utilizzato anche Ionic.

9.2.1 Lato server

È stato utilizzato DBMS PostgreSQL, cioè un DBMS orientato agli oggetti e pubblicato con licenza libera. Utilizza il linguaggio di dichiarativo SQL per eseguire le *query* sui dati.

In lato *server* l'applicazione è stata sviluppata utilizzando il linguaggio di programmazione Java. Inoltre è stato sfruttato il *framework* Java Spring il quale è fondamentale per sviluppare applicazioni utilizzando come linguaggio di programmazione Java. Infine è stato aggiunto Java Spring Boot per aggiungere alcune funzionalità aggiuntive al *framework*.

9.2.2 Lato client

In lato *client*, sono stati usati 7 linguaggi di vario tipo:

- **HTML** (HyperText Markup Language): non è un vero linguaggio di programmazione ma viene identificato come linguaggio di *markup*. Viene utilizzato per creare le pagine *web* e consente di formattare, impaginare e creare la struttura di una pagina *web*;
- **CSS**: è un linguaggio di stile utilizzato per definire la formattazione e lo stile di vari tipi di documenti, tra cui HTML;
- **JavaScript**: è un linguaggio di programmazione che consente di rendere dinamiche le pagine *web*, definendo dei comportamento in seguito a determinati eventi;
- **TypeScript**: è un linguaggio di programmazione che estende JavaScript. Il codice viene convertito in codice JavaScript per poter essere interpretato dal *browser*;
- **Angular**: è un *framework open source* per lo sviluppo di applicazioni *web*, *mobile* e *desktop* multipiattaforma;
- **Ionic**: è una piattaforma che consente di creare applicazioni native multipiattaforma e applicazioni *web*, utilizzando tecnologie *web*.
È stato progettato per integrarsi con alcuni *framework frontend* tra cui Angular, React, Vue o semplicemente nessun *framework*, utilizzando JavaScript Vanilla;
- **Swagger**: è un insieme di specifiche e di strumenti *software* che mirano a semplificare e standardizzare i processi di documentazione di API RESTful.
Swager UI consente allo sviluppatore di visualizzare le API tramite un'interfaccia *web* e offre inoltre la possibilità di provare il funzionamento delle API.

Si noti che gli strumenti come Angular, Ionic e Swagger, servono allo sviluppatore per creare l'applicazione. L'utente finale utilizzerà l'applicazione tramite un *client* (*browser*), senza doversi preoccupare del modo in cui è stata sviluppata.

9.3 Sistemi di sicurezza

9.3.1 Autenticazione password

WePlant come metodo di sicurezza per codificare le *password* utilizza la libreria di Spring `BCryptPasswordEncoder`, la quale utilizza l'algoritmo `BCrypt`. Esso genera una stringa lunga di 60 caratteri così composta:

```
$2<a/b/x/y>$[cost]$[22 character salt][31 character hash]
```

Per esempio, se viene inserita la *password* `abc123xyz`, con un costo di 12 (cioè $2^{12} = 4096$ *rounds*) e l'algoritmo *hash*, il risultato sarà:

\$2a\$12\$	R9h/cIPz0gi.URNNX3kh2O	PST9/PgBkqquzi.Ss7KIUgO2t0jWMUW
	Salt	Hash

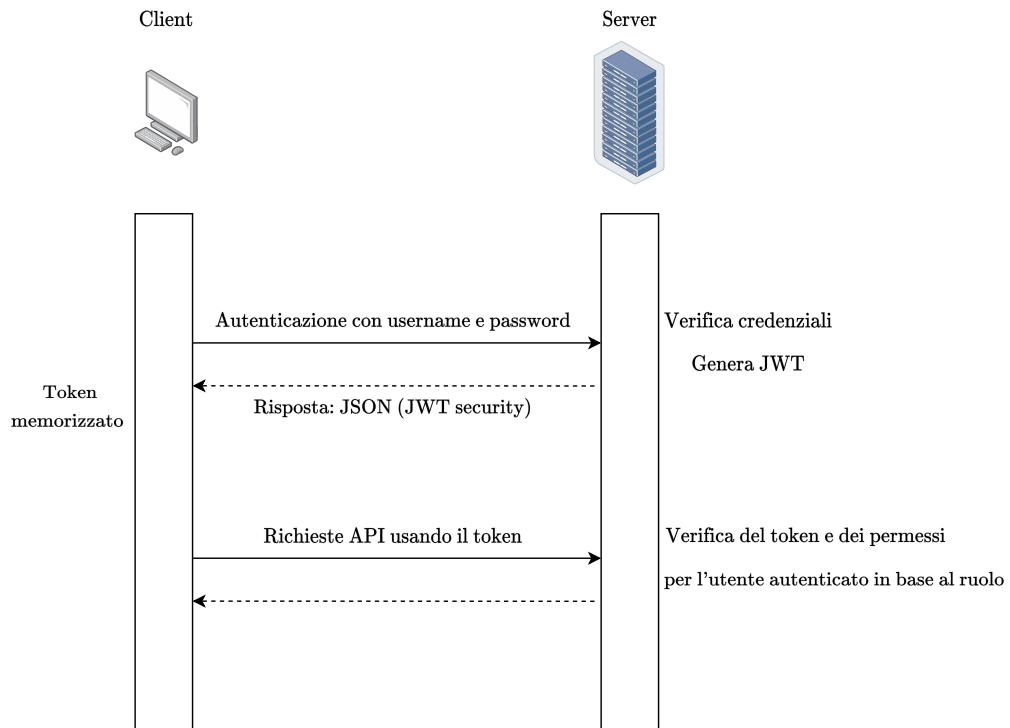
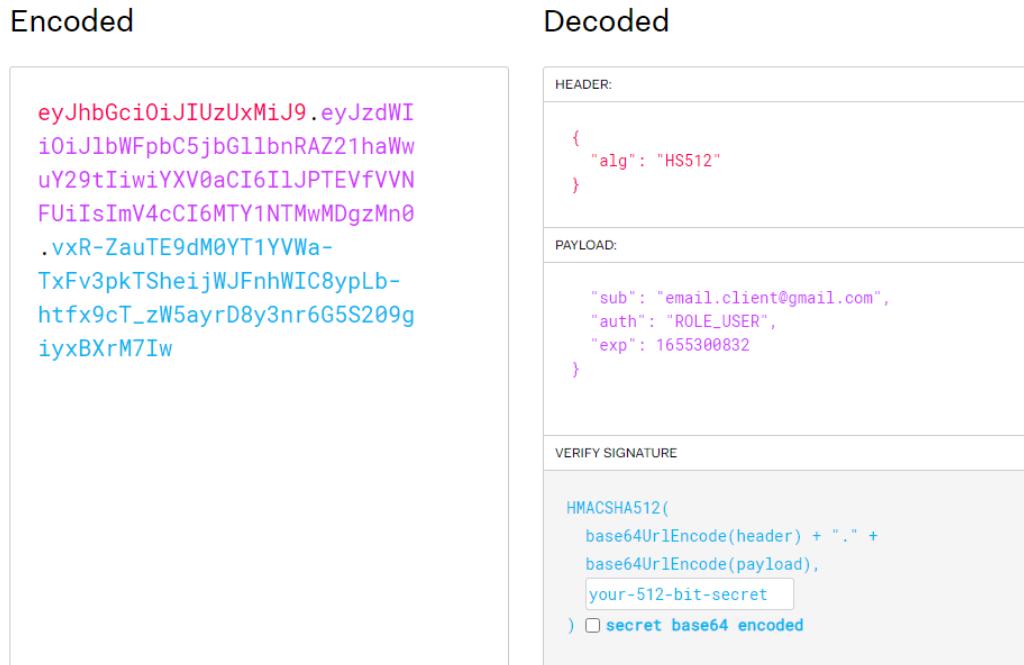
- 2a è l'algoritmo di *hash* (`bcrypt`);
- 12 è il costo di *input* ($2^{12} = 4096$ *rounds*);
- R9h/cIPz0gi.URNNX3kh2O è la codifica in base-64 del *input salt*;
- PST9/PgBkqquzi.Ss7KIUgO2t0jWMUW è la codifica dei primi 23 *bytes* dei 24 *byte* computati tramite *hash*.

9.3.2 Autenticazione e autorizzazione API REST: JWT

Nell'applicazione WePlant viene utilizzato un metodo per l'autenticazione e l'autorizzazione nelle richieste API REST. Il motivo è per evitare che un attaccante esegua delle richieste provenienti dall'esterno, di conseguenza senza autorizzazione.

Il metodo utilizzato è **JWT** (*Json Web Token*) che è un *open standard*, certificato da [RFC 7519](#), e definisce un metodo di sicurezza compatto e auto contenuto, per trasmettere informazioni tra più utenti con semplici *file JSON*. Tale *standard* si fonda sulla firma digitale tramite la coppia di chiavi pubblica/privata usando RSA oppure ECDSA.

In WePlant il *token* viene firmato utilizzando una ***secret-key*** (chiave simmetrica) presa dal *file* di configurazione. Quindi ogni richiesta API contiene questo *token*, nell'*header* di richiesta (*Authorization*), per verificare i permessi dell'utente.



9.4 Archiviazione

Della classe `Window`, esistono due oggetti da osservare attentamente:

- `Window localStorage`;
- `Window sessionStorage`

L'oggetto `Window localStorage` consente di memorizzare coppie chiave/valore nel *browser*. I dati memorizzati non hanno una scadenza e non vengono eliminati quando il *browser* viene chiuso, quindi sono disponibili per sessioni future. Il salvataggio avviene nella sessione del *browser*.

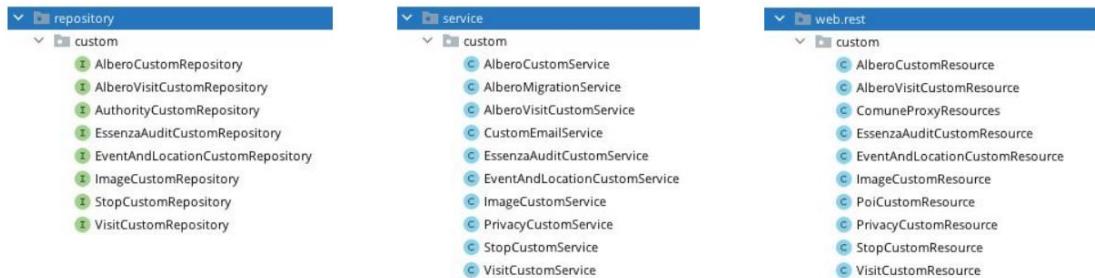
L'oggetto `Window sessionStorage` consente di memorizzare coppie chiave/valore nel *browser*. I dati sono memorizzati per una sola sessione, ovvero finché il *browser* non viene chiuso, essi sono validi. Nel momento in cui un documento viene caricato in una *tab*, una sessione di pagina unica viene creata e assegnata a quel *tab* specifico. Dunque, aprire una pagina in un novo *tab* o finestra, crea una nuova sessione.

9.5 Documentazione API

In WePlant esistono due gruppi di API:

- `default`, che sono auto generate;
- `external`, che sono API *custom* create *ad-hoc* per funzionalità specifiche.

La documentazione è disponibile nella sezione Administration → API.



9.6 Design Responsive

L'applicazione WePlant utilizza una configurazione ***Design Responsive***.

Con *Design Responsive* si intende una configurazione in cui il *server* invia sempre lo stesso codice HTML a tutti i dispositivi e il codice CSS viene utilizzato per modificare il *rendering* della pagina sul dispositivo. I vantaggi sono i seguenti:

- **Creazione di siti/applicazioni *web* che si adattano al dispositivo su cui vengono visualizzate** (PC, *tablet*, *smartphone*) in base alla dimensione dello schermo.
- **Ottimizzazione dei motori di ricerca**, ovvero dell'indicizzazione delle pagine (SEO, *Search Engine Optimization*). Le pagine *responsive* ottengono un punteggio più alto e vengono visualizzate come primi risultati.
- **Tempo di progettazione inferiore** per la manutenzione di più pagine per gli stessi contenuti.
- **Nessun reindirizzamento per offrire agli utenti una visualizzazione ottimizzata** in base al dispositivo in uso.

Per segnalare al *browser* che la pagina è *Design Responsive*, viene utilizzato un *meta tag* nell'intestazione chiamato ***meta tag viewport***. Nel caso in cui mancasse, i *browser* per dispositivi mobili applicano l'impostazione predefinita, ovvero eseguono il *rendering* della pagina alla larghezza utilizzata per gli schermi *desktop*.

Esercizi del corso

10.1 Esercizi sulle applicazioni di rete con interfaccia socket

10.1.1 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*, si osserva che 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*, allora le due parti non riescono a comunicare. Questo perché il client tenta di raggiungere un *host* non esistente.

10.1.2 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.

Listing 10.1: Codice del *client*

```
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 = createUDPIInterface(30000);
11
12    printf("[CLIENT] Spedisco messaggio al server\n");
13    printf("[CLIENT] Contenuto: %s\n", request);
```

```

14     UDPSSend(socket, request, strlen(request), "127.0.0.1", 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);
20 }
```

Listing 10.2: Codice del *server*

```

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

10.1.3 Esercizio 3 - UDP

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

1. Bisogna modificare anche il *client*? Se sì, dove?
2. 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 a essere eseguito per il semplice fatto che la porta 100 fa parte delle *well-known port* e non può essere utilizzata per altri scopi.

10.1. ESERCIZI SULLE APPLICAZIONI DI RETE CON INTERFACCIA SOCKET101

1. 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*.
2. Compilando il *server* con la modalità **sudo** è possibile forzare l'ascolto e comunicazione attraverso la porta 100. Di conseguenza il *server* si metterà in ascolto sulla porta 100 e il *client*, che eseguirà l'invio su tale porta, riuscirà a trasmettere il messaggio.

10.1.4 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*, pertanto 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.

10.1.5 Esercizio 5 - UDP [In laboratorio Delta]

► **Accordarsi per lavorare su coppie di macchine in modo che *server* e *client* siano su macchine diverse. Come bisogna modificare i sorgenti?**

► Soluzione. Supponendo che i due *host* siano connessi sulla stessa rete, perciò non per forza la rete universitaria, ma anche tramite un *hotspot* attraverso il telefono, è necessario modificare nel seguente modo i codici:

- Server: non effettua alcuna 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 che 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.

10.1.6 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.

Listing 10.3: Codice del *server*

```

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

Viceversa il codice del *client* non viene modificato.

10.1.7 Esercizio 7 - UDP

▷ **Compilare ed eseguire il secondo esempio.**

► Soluzione. Si esegue il secondo esempio che riguarda il `clientUDP` e `serverUDP` nella versione `_inc`.

10.1. ESERCIZI SULLE APPLICAZIONI DI RETE CON INTERFACCIA SOCKET103

Listing 10.4: 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 = createUDPIInterface(20000);
11
12    printf("Inserisci un numero intero:\n");
13
14    scanf("%d", &request);
15
16    UDPSend(socket, &request, sizeof(request), "127.0.0.1", 35000);
17
18    UDPReceive(socket, &response, sizeof(response), hostAddress, &port);
19
20    printf("[CLIENT] Ho ricevuto un messaggio da host/porta %s/%d\n",
21           hostAddress, port);
22
23 }
```

Listing 10.5: 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    printf("[SERVER] Sono in attesa di richieste da qualche client\n");
13
14    UDPReceive(socket, &request, sizeof(request), hostAddress, &port);
15
16    printf("[SERVER] Ho ricevuto un messaggio da host/porta %s/%d\n",
17           hostAddress, port);
18
19    printf("[SERVER] Contenuto: %d\n", request);
20
21    response = request + 1;
22
23    UDPSend(socket, &response, sizeof(response), hostAddress, port);
24 }
```

10.1.8 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.

Listing 10.6: 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 = createUDPIInterface(20000);
11    do {
12        printf("Inserisci un numero intero:\n");
13        scanf("%d", &request);
14
15        UDPSend(socket, &request, sizeof(request), "127.0.0.1", 35000);
16    } while (request != 0);
17
18    UDPReceive(socket, &response, sizeof(response), hostAddress, &port);
19
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 }
```

- [Righe 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* [Riga 15].
- [Righe 18-21] Nel momento in cui il numero inserito è zero, il programma termina aspettando il risultato che verrà inviato dal *server*. Alla ricezione, verrà stampato.

10.1. ESERCIZI SULLE APPLICAZIONI DI RETE CON INTERFACCIA SOCKET105

Listing 10.7: 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        printf("[SERVER] Sono in attesa di valori da qualche client\n");
15        UDPReceive(socket, &request, sizeof(request), hostAddress, &port);
16
17        printf("[SERVER] Sommo il numero %d...\n", request);
18        somma += request;
19    } while (request != 0);
20
21    UDPSend(socket, &somma, sizeof(somma), hostAddress, port);
22
23    printf("[SERVER] Ho ricevuto il valore di terminazione da host/porta %s
24          %d\n", hostAddress, port);
25    printf("[SERVER] Contenuto: %d\n", request);
}
```

- [Righe 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 all'interno di una sua variabile locale. Il ciclo continua finché non riceve un valore pari a zero.
- [Righe 21-24] 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.

10.1.9 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, per esempio:

- Digitare “2345” + INVIO
- Digitare “5187” + INVIO
- Staccare il cavo
- Digitare “2” + INVIO
- Riattaccare il cavo e aspettare 30 secondi che il sistema operativo si riassetti

- Digitare “1” + INVIO
- Digitare “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, motivato nei seguenti punti:

- 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.

10.1.10 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), per esempio:

<i>Client A</i>	<i>Client B</i>
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?

10.1. ESERCIZI SULLE APPLICAZIONI DI RETE CON INTERFACCIA SOCKET107

► Soluzione. Si mantengono gli stessi schemi dell'esercizio 8, eseguendo due *client A* e *B*. Ricordando di eseguire prima un'operazione del *client A*, poi un'operazione del *client B*, poi *A*, poi *B* e così via, si ottiene che:

- Nel *client B* viene letta la somma corretta, ovvero quella eseguita prima che il *client B* inviasse il valore 0 e richiedesse la terminazione del *server*.
- 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.

Listing 10.8: Codice del *server*

```
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 = createUDPIInterface(35000);
10
11    int somma = 0;
12    do {
13        printf("[SERVER] Sono in attesa di valori da qualche client\n");
14        UDPReceive(socket, &request, sizeof(request), hostAddress, &port);
15
16        printf("[SERVER] Sommo il numero %d...\n", request);
17        somma += request;
18    } while (request != 0);
19
20    UDPSend(socket, &somma, sizeof(somma), hostAddress, port);
21
22    printf("[SERVER] Ho ricevuto il valore di terminazione da host/porta %s
23          /%d\n", hostAddress, port);
24    printf("[SERVER] Contenuto: %d\n", request);
25
26    // Aggiorna i client che si collegano
```

```

27     do {
28         UDPReceive(socket, &request, sizeof(request), hostAddress, &port);
29
30         UDPSend(socket, &somma, sizeof(somma), hostAddress, port);
31     } while (true);
32 }
```

10.1.11 Esercizio 11 - Sommatrice TCP

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

► Soluzione.

Listing 10.9: 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
10    if (connection < 0) {
11
12        printf("[CLIENT] Errore nella connessione al server: %i\n",
13               connection);
14    } else {
15
16        do {
17            printf("[CLIENT] Inserisci un numero intero:\n");
18            scanf("%d", &request);
19
20            printf("[CLIENT] Invio richiesta con numero al server\n");
21
22            TCPSend(connection, &request, sizeof(request));
23        } while (request != 0);
24
25        TCPReceive(connection, &response, sizeof(response));
26
27        printf("[CLIENT] Ho ricevuto il seguente risultato dal server: %d\n",
28               response);
29
30        closeConnection(connection);
31    }
31 }
```

10.1. ESERCIZI SULLE APPLICAZIONI DI RETE CON INTERFACCIA SOCKET109

Listing 10.10: 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", socket);
11    } else {
12
13        printf("[SERVER] Sono in attesa di richieste di connessione da
14 qualche client\n");
15        connection = acceptConnection(socket);
16
17        printf("[SERVER] Connessione instaurata\n");
18
19        int somma = 0;
20        do {
21            TCPReceive(connection, &request, sizeof(request));
22
23            printf("[SERVER] Ho ricevuto la seguente richiesta dal client: %d\n",
24                   request);
25            somma += request;
26        } while (request != 0);
27
28        TCPSend(connection, &somma, sizeof(somma));
29
30        closeConnection(connection);
31    }
32 }
```

10.1.12 Esercizio 12 - Sommatrice TCP e influenze reciproche

► Provare a rifare l'esercizio 10, ma con questa nuova versione della sommatrice. Cosa si può osservare? Che soluzioni si possono 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.

10.1.13 Esercizio 13 - Sommatrice TCP e perdita dei pacchetti

► Riprovare l'esercizio 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.

10.1.14 Esercizio 14 - Trasferimento di un file

▷ Eseguendo i seguenti passi:

- 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.

Listing 10.11: Codice del *client*

```

1 #include "network.h"
2
3 int main() {
4     FILE *fptr1, *fptr2;
5     char filename[100], c;
6
7     printf("[CLIENT] Inserire il nome del file:\n");
8     scanf("%s", filename);
9
10    fptr1 = createTCPConnectionFD("localhost", 35000);
11
12    for(int i = 0; filename[i] != '\0'; i++)
13        fputc(filename[i], fptr1);
14        fputc('\0', fptr1);
15
16    // apertura del file destinazione in scrittura
17    fptr2 = fopen(filename, "w");
18    if (fptr2 == NULL) {
19        printf("[CLIENT] Errore apertura file %s\n", filename);
20        return 1;

```

10.1. ESERCIZI SULLE APPLICAZIONI DI RETE CON INTERFACCIA SOCKET111

```
21 }
22
23 // lettura primo byte dalla sorgente
24 c = fgetc(fptra);
25 while (c != EOF) {
26     // scrittura del byte nella destinazione
27     fputc(c, fptra2);
28     c = fgetc(fptra);
29 }
30
31 printf("[CLIENT] Contenuto trasferito su %s\n", filename);
32
33 fclose(fptra2);
34 fclose(fptra);
35 return 0;
36 }
```

Listing 10.12: Codice del *server*

```
1 #include "network.h"
2
3 int main() {
4     char filename[MTU], c;
5     socketif_t server;
6     FILE *fptra, *fptra2;
7     int i;
8
9     server = createTCPServer(35000);
10    if (server < 0) {
11        printf("[SERVER] Error: %i\n", server);
12        return -1;
13    }
14
15    while(1) {
16        fptra = acceptConnectionFD(server);
17
18        i = 0;
19        filename[i] = fgetc(fptra);
20        while(filename[i] != '\0') {
21            i++;
22            filename[i] = fgetc(fptra);
23        }
24
25        printf("[SERVER] Nome del file richiesto: %s\n", filename);
26
27        // apertura file fptra2 in lettura
28        fptra2 = fopen(filename, "r");
29        if (fptra2 == NULL) {
30            printf("[SERVER] Errore apertura file %s \n", filename);
31            fclose(fptra);
32            continue;
33        }
34    }
```

```
35     // lettura primo byte dalla fptr2
36     c = fgetc(fptr2);
37     while (c != EOF) {
38         // scrittura del byte nella fptr1
39         fputc(c, fptr1);
40         fflush(fptr1);
41         c = fgetc(fptr2);
42     }
43
44     printf("[SERVER] File inviato\n");
45
46     fclose(fptr2);
47     fclose(fptr1);
48 }
49 return 0;
50 }
```

10.2 Esercizi dal Web ai Webservices

10.2.1 Esercizio di HTML e JavaScript

▷ 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          <iframe id="area" height="2000" width="1000"></iframe>
5          <script>
6              document.getElementById("area").src = "https://www.ansa.it/
7                  sito/notizie/topnews/index.shtml";
8              function autoRefresh() {
9                  window.location = window.location.href;
10             }
11             setInterval('autoRefresh()', 2000);
12         </script>
13     </body>
14 </html>
```

10.2.2 Esercizio su un semplice web server

▷ Si eseguono i seguenti passaggi:

- 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. Si apre il file `serverHTTP.c`:

```

1 #include "network.h"
2
3 int main(){
4     char *HTMLResponse = "HTTP/1.1 200 OK\r\n\r\n<html><head><title>An
Example Page</title></head><body>Hello World, this is a very simple
HTML document.</body></html>\r\n";
```

```

5     socketif_t sockfd;
6     FILE* connfd;
7     int res, i;
8     long length = 0;
9     char request[MTU], method[10], c;
10
11    sockfd = createTCPServer(8000);
12    if (sockfd < 0) {
13        printf("[SERVER] Errore: %i\n", sockfd);
14        return -1;
15    }
16
17    while(true) {
18        connfd = acceptConnectionFD(sockfd);
19
20        fgets(request, sizeof(request), connfd);
21        printf("%s", request);
22        strcpy(method, strtok(request, " "));
23        while(request[0] != '\r') {
24            fgets(request, sizeof(request), connfd);
25            printf("%s", request);
26            if(strstr(request, "Content-Length:") != NULL) {
27                length = atol(request+15);
28                //printf("length %ld\n", length);
29            }
30        }
31
32        if(strcmp(method, "POST") == 0) {
33            for(i = 0; i < length; i++) {
34                c = fgetc(connfd);
35                printf("%c", c);
36            }
37        }
38
39        fputs(HTMLResponse, connfd);
40        fclose(connfd);
41
42        printf("\n\n[SERVER] sessione HTTP completata\n\n");
43    }
44
45    closeConnection(sockfd);
46    return 0;
47 }
```

- [Righe 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*.
- [Righe 11-15] Viene creata la *socket* sulla porta 8000 e in caso di errore il programma termina.
 - [Riga 18] Si attende una richiesta di connessione per l'invio di un *file descriptor* (FD).
 - [Righe 20-30] Al collegamento di un *client*, il *server* stampa l'intero *header* della richiesta sul terminale.
 - [Righe 32-37] Se viene eseguita una richiesta POST, viene stampato il contenuto del *file* passato.
 - [Righe 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* “<http://127.0.0.1:8000/>” oppure “<http://localhost:8000/>” (l'URL è lo stesso perché cambia solo l'*alias*).

10.2.3 Passare dei dati al server web col metodo GET

▷ 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. Avviando il *server web* e aprendo la pagina *web*, è possibile visualizzare lo stesso *form* visualizzabile nella pagina precedente.

Eseguendo il metodo GET, quindi cliccando su “Invia” e 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:

http						
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	0.078455666	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 indentazione 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.

10.2.4 Passare dei dati al server web col metodo POST

- ▷ 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. Avviando il *server web* e aprendo la pagina *web*, è possibile visualizzare lo stesso *form* visualizzabile nella pagina precedente.

Eseguendo il metodo POST, quindi cliccando su “Invia” e 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:

No.	Time	Source	Destination	Protocol	Length	Info
+ 4	0.000229816	127.0.0.1	127.0.0.1	HTTP	647	POST /action HTTP/1.1 (application/x-www-form-urlencoded)
+ 8	0.000601424	127.0.0.1	127.0.0.1	HTTP	66	HTTP/1.1 200 OK
▶ Frame 4: 647 bytes on wire (5176 bits), 647 bytes captured (5176 bits) on interface eth0, Src: 00:0c:29:b8:16:00, Dst: 127.0.0.1 (00:00:00:00:00:00)						
▶ Ethernet II, Src: 00:0c:29:b8:16:00 (00:0c:29:b8:16:00), Dst: 127.0.0.1 (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: 46576, Dst Port: 80						
▶ Hypertext Transfer Protocol						
+ HTML Form URL Encoded: application/x-www-form-urlencoded						
+ Form item: "fname" = "John"						
Key: fname						
Value: John						
+ Form item: "lname" = "Doe"						
Key: lname						
Value: Doe						
0000	00:00:00:00:00:00	00:00:00:00:00:00	00:00:00:00:00:00	HTTP	647	POST /action HTTP/1.1 (application/x-www-form-urlencoded)
0001	02:79:8b:04:40:00	00:00:00:00:00:00	af:78:7f:00:00:01	HTTP	66	HTTP/1.1 200 OK
0002	00:01:b5:f0:41:fd	00:00:00:00:00:00	9d:f5:66:b8:1e:80	HTTP	66	HTTP/1.1 200 OK
0003	02:00:00:0e:00:01	00:00:00:00:00:01	08:a0:c8:e8:21:14	HTTP	66	HTTP/1.1 200 OK
0040	21:14:50:4f:53:20	00:00:00:00:00:00	61:63:74:69:6f:20	HTTP	66	HTTP/1.1 200 OK
0050	54:5a:50:2f:31:2e	00:00:00:00:00:00	04:a8:4f:73:74:3a	HTTP	66	HTTP/1.1 200 OK
0060	6f:63:61:6c:68:6f	00:00:00:00:00:00	73:3a:38:30:30:0d	HTTP	66	HTTP/1.1 200 OK
0070	63:65:72:2d:41:67	00:00:00:00:00:00	65:6e:74:3a:20:4d	HTTP	66	HTTP/1.1 200 OK
0080	6c:61:2f:35:2e:20	00:00:00:00:00:00	28:58:31:3b:29:55	HTTP	66	HTTP/1.1 200 OK
0090	6e:74:75:3b:20:4c	00:00:00:00:00:00	69:6e:78:78:20:36	HTTP	66	HTTP/1.1 200 OK
00a0	34:3b:20:72:76:3a	00:00:00:00:00:00	31:39:39:2e:30:29	HTTP	66	HTTP/1.1 200 OK
00b0	6b:6f:2f:32:30:31	00:00:00:00:00:00	30:31:30:21:20:46	HTTP	66	HTTP/1.1 200 OK
00c0	66:6f:78:2f:31:32	00:00:00:00:00:00	32:0d:00:41:63:63	HTTP	66	HTTP/1.1 200 OK
00d0	74:3a:26:74:65:78	00:00:00:00:00:00	42:2f:6d:6c:2c:61	HTTP	66	HTTP/1.1 200 OK
00e0	6c:66:63:61:74:66	00:00:00:00:00:00	6e:78:6d:74:6d:2c	HTTP	66	HTTP/1.1 200 OK
00f0	6d:66:2c:61:70:70	00:00:00:00:00:00	69:63:61:74:69:6f	HTTP	66	HTTP/1.1 200 OK
0100	6c:63:3b:71:3d:30	00:00:00:00:00:00	2e:39:2c:69:66:61	HTTP	66	HTTP/1.1 200 OK
0110	76:66:66:2c:69:6d	00:00:00:00:00:00	61:67:2f:77:65:62	HTTP	66	HTTP/1.1 200 OK
0120	2f:2b:3b:71:3d:30	00:00:00:00:00:00	3e:38:0d:04:41:63	HTTP	66	HTTP/1.1 200 OK
0130	2d:4c:61:6e:67:75	00:00:00:00:00:00	67:65:3a:20:69:74	HTTP	66	HTTP/1.1 200 OK
0140	24:69:74:3b:71:3d	00:00:00:00:00:00	30:2e:38:2c:65:66	HTTP	66	HTTP/1.1 200 OK
0150	71:3d:30:2e:35:2c	00:00:00:00:00:00	65:6e:3b:71:3d:30	HTTP	66	HTTP/1.1 200 OK
0160	41:63:63:65:70:74	00:00:00:00:00:00	45:6e:63:6f:64:69	HTTP	66	HTTP/1.1 200 OK
0170	20:67:74:69:70:2c	00:00:00:00:00:00	64:66:6c:61:74:65	HTTP	66	HTTP/1.1 200 OK
0180	62:72:8d:0a:43:6f	00:00:00:00:00:00	76:6e:74:2d:54:79	HTTP	66	HTTP/1.1 200 OK
0190	3a:20:61:70:70:6c	00:00:00:00:00:00	63:61:74:69:6f:6e	HTTP	66	HTTP/1.1 200 OK
01a0	77:77:77:2d:66:6f	00:00:00:00:00:00	72:6d:2d:75:72:6c	HTTP	66	HTTP/1.1 200 OK
01b0	64:65:64:0d:04:45	00:00:00:00:00:00	6f:6e:74:6d:24:4c	HTTP	66	HTTP/1.1 200 OK
01c0	67:74:68:3a:20:32	00:00:00:00:00:00	0d:04:4f:72:69:67	HTTP	66	HTTP/1.1 200 OK
01d0	20:66:75:6c:66:0d	00:00:00:00:00:00	04:6e:66:65:63:74	HTTP	66	HTTP/1.1 200 OK
01e0	6e:3a:26:6b:65:70	00:00:00:00:00:00	43:6e:61:69:76:65	HTTP	66	HTTP/1.1 200 OK
01f0	70:67:72:61:64:65	00:00:00:00:00:00	2d:49:6e:73:65:63	HTTP	66	HTTP/1.1 200 OK
0200	52:65:71:75:65:73	00:00:00:00:00:00	74:73:2a:30:01:00	HTTP	66	HTTP/1.1 200 OK
0210	2d:4c:65:74:63:68	00:00:00:00:00:00	2d:44:65:73:74:3a	HTTP	66	HTTP/1.1 200 OK
0220	75:6d:65:6e:74:0d	00:00:00:00:00:00	53:65:2d:46:65:74	HTTP	66	HTTP/1.1 200 OK
0230	2d:4d:67:64:65:2a	00:00:00:00:00:00	26:0e:61:76:69:67	HTTP	66	HTTP/1.1 200 OK
0240	0a:53:65:63:2d:46	00:00:00:00:00:00	54:68:2d:53:69:74	HTTP	66	HTTP/1.1 200 OK
0250	20:63:72:6f:73:2d	00:00:00:00:00:00	73:69:74:65:0d:0a	HTTP	66	HTTP/1.1 200 OK
0260	2d:46:65:74:63:68	00:00:00:00:00:00	2d:55:73:65:72:3a	HTTP	66	HTTP/1.1 200 OK
0270	0a:00:0a:56:68:61	00:00:00:00:00:00	5d:34:6f:68:2e:26	HTTP	66	HTTP/1.1 200 OK
0280	51:6d:65:3d:44:67	00:00:00:00:00:00	65:6e:6c:6e:6d:6e	HTTP	66	HTTP/1.1 200 OK

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

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

10.3 Esercizi di modifica del web server

10.3.1 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 si deve scrivere nella barra del *browser*?
- Capire come recuperare la stringa con il nome del *file* della richiesta HTTP.
- 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*.

```

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
16    if (sockfd < 0) {
17        printf("[SERVER] Errore: %i\n", sockfd);
18        return -1;
19    }
20

```

```

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

```

```

72         exit(-1);
73     }
74     fclose(fptra);
75     // check errors and send to client
76     if (buffer)
77     {
78         // send OK code
79         fputs("HTTP/1.1 200 OK\r\n\r\n", connfd);
80         // send web page
81         fputs(buffer, connfd);
82         fputs("\r\n", connfd);
83     }
84     else {
85         printf("Error fread\n");
86         exit(-1);
87     }
88     free(buffer);
89 }
90
91     // close File Descriptor
92     fclose(connfd);
93     printf("\n\n[SERVER] sessione HTTP completata\n\n");
94 }
95
96     closeConnection(sockfd);
97     return 0;
98 }
```

- Righe [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*.
- Righe [24-28] 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.
- Righe [45-48] Come visto 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*.

- Righe [50-57] Il *server* tenta di aprire il *file* specificato dal *client*. Nel caso in cui il puntatore **fptra** rimanga nullo, il *file* non esiste oppure non può essere aperto. Di conseguenza il *server* risponde al *client* con un 404 Not Found e stampa l'errore.

¹La funzione **strtok** esegue una divisione di stringhe a seconda del delimitatore impostato.

- Righe [50-82] In questa parte di codice l’obiettivo è leggere il contenuto del *file* richiesto dall’utente e inviarglielo come pagina *web*.
 - Riga [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*.
 - Riga [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.
 - Riga [65] Dopo aver calcolato il numero di caratteri dentro il *file*, il puntatore del *file stream* viene riposizionato all’inizio (`SEEK_SET`).
 - Righe [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`;
 - Righe [76-82] Se durante la lettura da *file* non ci sono stati errori, allora viene inviata una risposta affermativa al *client*, cioè un codice 200 del protocollo HTTP; viene inviato l’intero contenuto del *buffer*, quindi del *file*; infine, vengono inviati dei caratteri di indentazione (non necessari).

10.3.2 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 e 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).

```
1 #include "network.h"
2
3 int main() {
4     char *HTMLResponse = "HTTP/1.1 200 OK\r\n\r\n<html><head><title>An
5         Example Page</title></head><body>Hello World, this is a very simple
6         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, " "));
24        while(request[0] != '\r') {
25            fgets(request, sizeof(request), connfd);
26            printf("%s", request);
27            if(strstr(request, "Content-Length:") != NULL) {
28                length = atol(request+15);
29                //printf("length %ld\n", length);
30            }
31        }
32
33        if(strcmp(method, "POST") == 0)
34        {
35            // filename to rcv
36            char filename[100] = "";
37            // flag used to obtain the filename string
38            bool flag = 1;
39
40            // read MIME part
41            while(true)
42            {
43                // take a letter
44                c = fgetc(connfd);
45                printf("%c", c);
46
47                // verify if it's filename
48                if (flag == 1 && c == ';')
49                {
50                    // read (space)
51                    c = fgetc(connfd);
52                    printf("%c", c);
```

```

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 is an 'e'
63             if (c == 'e')
64             {
65                 // read =
66                 c = fgetc(connfd);
67                 printf("%c", c);
68
69                 c = fgetc(connfd);
70                 printf("%c", c);
71
72                 /* read name of file */
73                 while (true)
74                 {
75                     c = fgetc(connfd);
76                     printf("%c", c);
77                     // read until there is a letter or other
78                     if (c == '"')
79                     {
80                         flag = 0;
81                         break;
82                     }
83                     else
84                         // build filename
85                         strncat(filename, &c, 1);
86                 }
87             }
88         }
89     }
90
91     // verify if it's the start of file contents
92     if (c == '\r')
93     {
94         // read \n
95         c = fgetc(connfd);
96         printf("%c", c);
97         if (c == '\n')
98         {
99             // read another \r
100            c = fgetc(connfd);
101            printf("%c", c);
102            if (c == '\r')
103            {
104                // read another \n

```

```
105         c = fgetc(connfd);
106         printf("%c", c);
107         if (c == '\n')
108         {
109             // INSIDE THE FILE
110             // Create local file
111             fptra1 = fopen(filename, "w");
112             if (fptra1 == NULL)
113             {
114                 printf("Cannot open file %s \n", 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, fptra1);
126                 fflush(fptra1);
127                 c = fgetc(connfd);
128             }
129             fclose(fptra1);
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 }
```

- **Righe [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*.
- **Righe [10-29]** Il codice è lo stesso e la spiegazione è possibile trovarla nel codice sorgente (Sezione 10.2.2).
- **Riga [31]** Se il metodo è di tipo POST, allora il codice accoglie il *file*, altrimenti invia la pagina di *default* classica (`HTMLResponse`) e chiude la connessione di *file descriptor*.
- **Righe [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`. Pertanto se sia effettivamente il nome del *file* e non ci si trovi in un altro punto.
- **Righe [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: "-----"
  [Type: multipart/form-data]
  First boundary: -----3063446421998555818734781761\r\n
  -> Encapsulated multipart part: (application/x-shellscrip)
    Content-Disposition: form-data; name="myfile"; filename="update.sh"\r\n
    Content-Type: application/x-shellscrip\r\n\r\n
  -> Media Type
    Media type: application/x-shellscrip (175 bytes)
  Last boundary: \r\n-----3063446421998555818734781761--\r\n

```

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, cioè uno *script* della *shell* (***shellscrip***) e ovviamente il contenuto del *file* in *bytes*.

A fine di ogni linea ci sono alcuni *escape characters*. Essi sono molto importanti.

- **Righe [41-89]** Per leggere il nome del *file*, si esegue questo pezzo di codice:
 - * **Righe [41-46]** Viene effettuata la lettura del contenuto. Nel caso in cui la `flag` sia a 1, ossia ancora non è stato trovato il `filename`, e il carattere

letto corrisponde al punto e virgola ;, allora 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.

- * **Righe [48-61]** Supponendo che la lettura di ogni singolo carattere sia vicino a `filename`, eseguendo due letture (**Riga 49** e **Riga 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 (**Riga 55**). Se la parola inizia con la `f` vengono eseguite 7 letture 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`.
- * **Righe [62-71]** Se la sequenza è corretta, allora l'ultima parola letta dovrebbe essere una `e` e successivamente viene verificata tale condizione (**Riga 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.
- * **Righe [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 (**Riga 76**) e la concatenazione nella stringa risultato (**Riga 86**) continuano finché non viene trovato il doppio apice (**Riga 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.
- **Righe [91-148]** Per leggere il contenuto del `file` che si trova all'interno dei `boundary`, si esegue questo pezzo di codice:
 - * **Righe [91-92]** Osservando la struttura del MIME è possibile notare che ogni riga del protocollo termina con un \e e \n. Perciò è possibile sfruttare questa caratteristica per capire quando inizia il contenuto del `file`.
 - * **Righe [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 sia questa alternanza, allora il puntatore è all'interno del `file`.

- * Righe [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.
- * Righe [118-129] Inizia la lettura/scrittura vera e propria del *file*. Finché 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.
- * Righe [131-148] Avvengono una serie di letture della *pipe* per consentire di svuotarla.
- Righe [149-157] Il *server* invia una risposta di positiva di avvenuta ricezione, inviando la pagina di *default*. Infine la connessione viene chiusa.

10.4 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.

```

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</title></head><body>Risultato=%f</body></html>\r\n\r\n", somma);
18 }
19
20 int main(){
21     socketif_t sockfd;
```

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

Le uniche differenze degne di nota sono:

- **Righe [53-61]** Viene controllato l'URL richiesto dall'utente. Se si tratta della pagina sommatrice, viene invocata la funzione (**Riga 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`.
- **Righe [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.

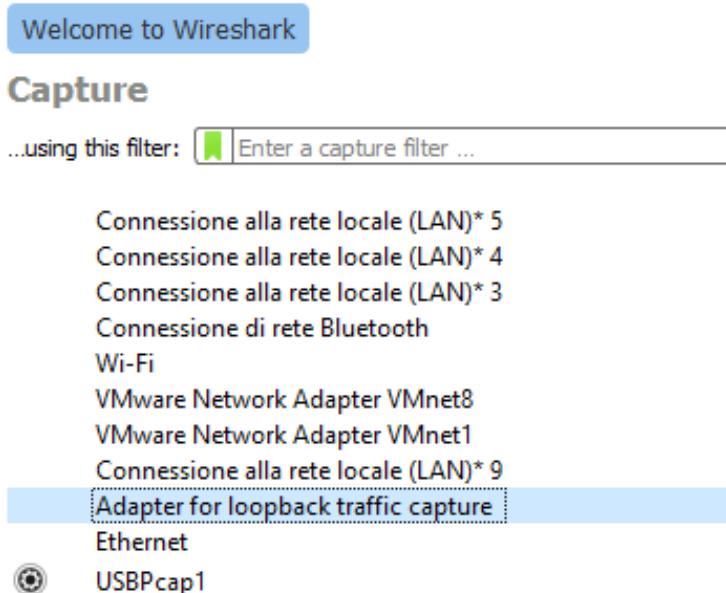
10.5 Esercizi sulla WebSocket Chat

10.5.1 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 sere tale procedura?

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

► Soluzione. 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 3/4 *client* e 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. 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	71	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: 0xfcfd9 [unverified]
  [Checksum Status: Unverified]
  Urgent Pointer: 0
  > [Timestamps]
  > [SEQ/ACK analysis]
  TCP payload (12 bytes)
  [PDU Size: 12]
```

Adesso che è chiaro quali siano i *client* (quelli “marchiati” con MASKED e il motivo per cui i messaggi siano mascherati è dovuto ad una questione di sicurezza) e quale sia 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 (analogamente per gli altri). Si deduce che il *client* faccia sapere al *server* che è ancora connesso ogni 25 secondi circa ($284 - 259 = 25$).

10.5.2 Esercizio 2

▷ Modificare il sorgente del codice per fare in modo che a ogni utente connesso alla *chat* arrivi nella *console* il messaggio “l’utente sta scrivendo...”. **NOTA:** lato *client*, bisogna spedire al *server* un evento apposito (per esempio “*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.

Listing 10.13: Server

```

1 var express  = require('express');
2 var socket  = require('socket.io');
3
4 //Chat setup
5 var app  = express();
6
7 // in questo momento il server e' in attesa delle connessioni
8 // HTTP sulla porta 4000
9 var server = app.listen(4000, function(){
10   console.log('waiting for HTTP requests on port 4000,');
11 });
12
13 // Static files

```

```

14 // con questa funzione viene specificato a Nodejs che
15 // una volta ricevuta una connessione deve andare a
16 // cercare nella cartella public il file html da fornire
17 // al client
18 app.use(express.static('public'));
19
20 // Socket setup & pass server
21 // una volta che la connessione e' stata ricevuta qui
22 // viene effettuato l'upgrade ad una connessione
23 // websocket e il server si mette in attesa degli
24 // eventi ai quali rispondere
25 var io = socket(server);
26 io.on('connection', function(webSocket){
27   console.log('made webSocket connection', webSocket.id);
28
29   // Ricezione di un messaggio da inoltrare ai client
30   webSocket.on('message', function(data){
31     io.sockets.emit('UploadChat', data);
32   });
33   webSocket.on('typing', function(data){
34     webSocket.broadcast.emit('typing', data);
35   });
36 });

```

Il codice è rimasto lo stesso (vedi Sezione 2.2.4), l'unica modifica effettuata è stata dalla Riga 33 e alla Riga 35 in cui si impone al *server* di inoltrare il messaggio ricevuto, con tag *typing*, a tutti gli altri *client* eccetto il *client* mittente.

Listing 10.14: Client

```

1 var name = prompt("What's your name?");
2 while(name == ""){
3   name = prompt("You have to choose a name. \n What's your name?")
4 }
5
6 // Query DOM
7 var message = document.getElementById('message'),
8 sender = document.getElementById('sender'),
9 btn = document.getElementById('send'),
10 output = document.getElementById('output'),
11 feedback = document.getElementById('feedback');
12 sender.innerHTML = name;
13 sender.value = name;
14
15 // Invio richiesta di connessione al server
16 var webSocket = io.connect();
17
18 // Trigger with event key down
19 message.onkeydown = function(e) {
20   e = e || window.event;
21   webSocket.emit('typing', {
22     sender: sender.value
23   })
24 }

```

```

25 // Listen for events
26 btn.addEventListener('click', function(){
27   if (message.value != ""){
28     webSocket.emit('message', {
29       message: message.value,
30       sender: sender.value,
31     });
32     message.value = "";
33   }
34 });
35 );
36
37 // UploadChat event
38 webSocket.on('UploadChat', function(data){
39   feedback.innerHTML = '';
40   var current_date = new Date();
41   output.innerHTML += '<p>' + 'Time: ' + current_date.getHours() + ':'
42   + current_date.getMinutes() + ':'
43   + current_date.getSeconds()
44   + ' - ' + '<strong>' + data.sender
45   + ': </strong>' + data.message + '</p>';
46 });
47
48 // Typing event
49 webSocket.on('typing', function(data){
50   var current_date = new Date();
51   feedback.innerHTML = '<p>' + 'Time: ' + current_date.getHours() + ':'
52   + current_date.getMinutes() + ':'
53   + current_date.getSeconds()
54   + ' - ' + '<strong>' + data.sender
55   + ': </strong>' + 'typing...' + '</p>';
56 })

```

Il codice del *front-end* è rimasto pressoché identico (vedi Sezione 2.2.5) tranne per due modifiche importanti:

- Righe [18-24] 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`).
- Righe [37-56] Alla ricezione dell'evento `typing` da parte del *server*, il *client* stamperà la scritta `typing....`

10.5.3 Esercizio 3

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

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

10.6 Esercizi su Webservice basati su REST

Esercizio 1 - Analizzare server e client REST-GET

▷ Considerare la cartella Webservice/ e aprire il file serverHTTP-REST.c per 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 bisogna 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\n{\r\n    Funzione non
41 riconosciuta!\r\n}\r\n");
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 parametri
50     function = strtok(url, "?&");
51     op1 = strtok(NULL, "?&");
52     op2 = strtok(NULL, "?&");
53     strtok(op1, "=");
54     val1 = atof(strtok(NULL, "="));
55     strtok(op2, "=");
56     val2 = atof(strtok(NULL, "="));
57
58     // chiamata alla business logic
59     somma = calcolaSomma(val1, val2);
60
61     // skeleton: codifica (serializzazione) del risultato
62     fprintf(connfd,"HTTP/1.1 200 OK\r\n\r\n\n{\r\n    \"somma\":%f\r\n}\r
63 \n", somma);
64 }
65
66 fclose(connfd);
67
68 printf("\n\n[SERVER] sessione HTTP completata\n\n");
69
70 closeConnection(sockfd);
71 return 0;
72 }
```

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&param2
8    =%f", val1, val2);
9
10    // chiamata del webservice
11    int res = doGET(request, response, MTU);
12    if (res < 0){
```

```

12     printf("Errore: %i\n", res);
13     return -1;
14 }
15
16 printf("Risposta dal server:\n%s\n", response);
17
18 // stub: de-codifica (de-serializzazione) del risultato
19 return atof(strstr(response,":")+1);
20 }
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(argv[3])));
30 }
31
32 return 0;
33 }
```

Come si può vedere dal codice (Riga 24) che 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 `calcolaSomma()` debba essere uguale. In caso contrario potrebbe manifestarsi un risultato inesatto o un comportamento inaspettato.

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 op1 op2");
12        }
13        else if(args[0].equals("calcola-somma")) {
14            System.out.println("Risultato: " + service1.calcolaSomma(Float.
15                parseFloat(args[1]), Float.parseFloat(args[2])));
16        }
17    }
18
19 class RESTAPI
20 {
21     String server;
22
23     RESTAPI(String remoteServer)  {
24         server = new String(remoteServer);
25     }
26
27     float calcolaSomma(float val1, float val2)  {
28
29         URL u = null;
30         float risultato = 0;
31         int i;
32
33         try
34         {
35             u = new URL("http://"+server+":8000/calcola-somma?param1="+val1+"&
36             param2="+val2);
37             System.out.println("URL aperto: " + u);
38         }
39         catch (MalformedURLException e)
40         {
41             System.out.println("URL errato: " + u);
42         }
43
44         try
45         {
46             URLConnection c = u.openConnection();
47             c.connect();
48             BufferedReader b = new BufferedReader(new InputStreamReader(c.
49             getInputStream()));
49             System.out.println("Lettura dei dati...");
50             String s;
```

```

50     while( (s = b.readLine()) != null ) {
51         System.out.println(s);
52         if((i = s.indexOf("somma"))!= -1)
53             risultato = Float.parseFloat(s.substring(i+7));
54     }
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, 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.

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 tratta spunto dal programma prime-number-interval.c

Successivamente:

- Estendere il ClientREST.java in modo da poter chiamare, a scelta, entrambe le funzionalità della nuova API.
- Provare il *client* con il calcolo dei numeri 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;
```

```
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
51 int main(){
52     socketif_t sockfd;
53     FILE* connfd;
54     int res, i;
55     long length = 0;
56     char request[MTU], url[MTU], method[10], c;
57
58     sockfd = createTCPServer(8000);
59     if (sockfd < 0){
60         printf("[SERVER] Errore: %i\n", sockfd);
61         return -1;
62     }
```

```

63
64     while(true) {
65         connfd = acceptConnectionFD(sockfd);
66
67         fgets(request, sizeof(request), connfd);
68         strcpy(method,strtok(request, " "));
69         strcpy(url,strtok(NULL, " "));
70         while(request[0] != '\r') {
71             fgets(request, sizeof(request), connfd);
72             if(strstr(request, "Content-Length:") != NULL) {
73                 length = atol(request+15);
74             }
75         }
76
77         if(strcmp(method, "POST") == 0) {
78             for(i = 0; i<length; i++) {
79                 c = fgetc(connfd);
80             }
81         }
82
83         if strstr(url, "calcola-somma") == NULL)
84         if strstr(url, "calcola-num-primi") == NULL)
85         fprintf(connfd,"HTTP/1.1 200 OK\r\n\r\n{\r\n        Funzione non
riconosciuta!\r\n}\r\n");
86         else
87         {
88             printf("Chiamata a funzione numero primo\n");
89
90             char *function, *op1, *op2;
91             int val1, val2;
92
93
94             // skeleton: decodifica (de-serializzazione) dei 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 risultato
110            fprintf(connfd,"HTTP/1.1 200 OK\r\n\r\n{\r\n        \"numeri primi\":%s
\r\n}\r\n", primi);
111
112            free(primi);
113        }

```

```

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

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             {
14                 System.out.println("USAGE: java ClientREST tipofunzione op1 op2");
15             }
16         else
17             if(args[0].equals("calcola-somma"))
18                 System.out.println("Risultato: " + service1.calcolaSomma(Float.parseFloat(args[1]), Float.parseFloat(args[2])));
19     }
20 }
```

```

18     else
19         if(args[0].equals("calcola-num-primi"))
20             System.out.println("Risultato: " + service1.calcolaPrimi(Integer.
21                             parseInt(args[1]), Integer.parseInt(args[2])));
22     }
23
24 class RESTAPI
25 {
26     String server;
27
28     RESTAPI(String remoteServer) {
29         server = new String(remoteServer);
30     }
31
32     float calcolaSomma(float val1, float val2) {
33         URL u = null;
34         float risultato = 0;
35         int i;
36
37         try
38         {
39             u = new URL("http://"+server+":8000/calcola-somma?param1="+val1+"&
40             param2="+val2);
41             System.out.println("URL aperto: " + u);
42         }
43         catch (MalformedURLException e)
44         {
45             System.out.println("URL errato: " + u);
46         }
47
48         try
49         {
50             URLConnection c = u.openConnection();
51             c.connect();
52             BufferedReader b = new BufferedReader(new InputStreamReader(c.
53             getInputStream()));
54             System.out.println("Lettura dei dati...");
55             String s;
56             while( (s = b.readLine()) != null ) {
57                 System.out.println(s);
58                 if((i = s.indexOf("somma")) != -1)
59                     risultato = Float.parseFloat(s.substring(i+7));
60             }
61         }
62         catch (IOException e)
63         {
64             System.out.println(e.getMessage());
65         }
66
67         return (float)risultato;
68     }
69
70 }
```

```

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="+
76             val1+"&param2="+val2);
77             System.out.println("URL aperto: " + u);
78         }
79         catch (MalformedURLException e)
80         {
81             System.out.println("URL errato: " + u);
82         }
83
84         try
85         {
86             URLConnection c = u.openConnection();
87             c.connect();
88             BufferedReader b = new BufferedReader(new InputStreamReader(c.
89             getInputStream()));
90             System.out.println("Lettura dei dati...");
91             String s;
92             while( (s = b.readLine()) != null ) {
93                 System.out.println(s);
94                 if((i = s.indexOf("primi"))!= -1)
95                     risultato.add(Integer.parseInt(s.substring(i+7)));
96                 try {
97                     risultato.add(Integer.parseInt(s));
98                 }
99                 catch (Exception e){}
100             }
101         }
102         catch (IOException e)
103         {
104             System.out.println(e.getMessage());
105         }
106     return risultato;
107 }
```

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 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 op1 op2");
10        }
11        else      {
12            RESTAPI service1 = new RESTAPI("127.0.0.1", args[0], args[1], args[2]);
13            RESTAPI service2 = new RESTAPI("127.0.0.1", args[0], args[1], args[2]);
14            RESTAPI service3 = new RESTAPI("127.0.0.1", args[0], args[1], args[2]);
15            service1.start();
16            service2.start();
17            service3.start();
18        }
19    }
20 }
21
22 class RESTAPI extends Thread
23 {
24     String server, service, param1, param2;
25
26     public void run()      {
27         if(service.equals("calcola-somma"))      {
28             System.out.println("Risultato: " + calcolaSomma(Float.parseFloat(
29                 param1), Float.parseFloat(param2)));
30         }
31         else      {
32             System.out.println("Servizio non disponibile!");
33         }
34     }
35 }
```

```

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

Per eseguire il codice su tre macchine differenti, sarebbe necessario modificare l'URL di destinazione alle Righe [12-14]. Al loro posto sarebbe necessario inserire l'indirizzo IP delle tre macchine differenti ed eseguire su quest'ultime il codice del **serverHTTP-REST.c**.

Infine si presenta di seguito un modo semplice, ma efficiente, per calcolare i numeri primi in un dato intervallo. Supponendo che nella rete ci siano diversi *server* che eseguano tale calcolo, si potrebbe eseguire il codice Java (qua sopra), aggiungendo il metodo di richiesta dei numeri primi come nell'esercizio precedente, dichiarando tanti *thread* quanti sono i *server* e a ciascun *thread* imporre di richiedere al *server* un calcolo dei numeri primi in un intervallo diverso dagli altri.

Per esempio si supponga che nella rete ci siano 5 *server* disponibili che hanno in esecuzione il *server* scritto nel linguaggio C. Se viene richiesto al *client* di calcolare un intervallo di numeri primi da 1 a 1'000'000 effettuando una richiesta REST, il programma:

1. dividerà il valore massimo per la quantità di *server* presenti nella rete, cioè 5 ($1'000'000 \div 5 = 200'000$);
2. creerà 5 istanze *thread* diverse assegnando loro un intervallo diverso:
 - (a) 1^o istanza *thread*, intervallo: 1 – 200'000;
 - (b) 2^o istanza *thread*, intervallo: 200'001 – 400'000;
 - (c) 3^o istanza *thread*, intervallo: 400'001 – 600'000;
 - (d) 4^o istanza *thread*, intervallo: 600'001 – 800'000;
 - (e) 5^o istanza *thread*, intervallo: 800'001 – 1'000'000.

In questo modo il tempo per ottenere i risultati sarà ottimizzato di cinque volte rispetto a farlo eseguire su un solo *server* in un unico blocco.

10.7 Esercizi su Wireshark

10.7.1 Esercizio 1 - File capture.cap

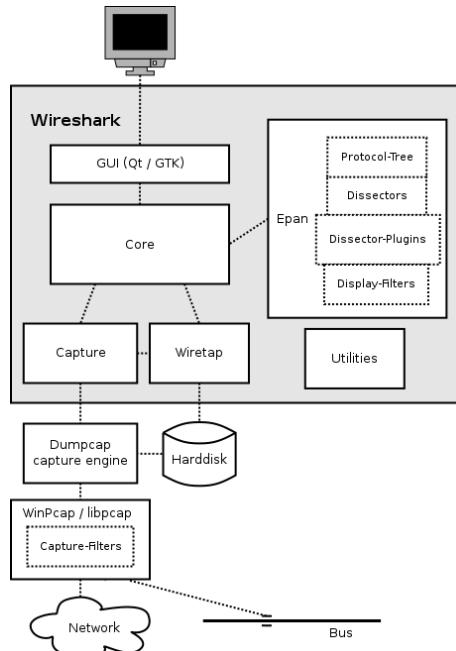
Avviare Wireshark, aprire il menù **File/Apri** e selezionare il *file capture.cap*. Si prenda in considerazione il pacchetto numero 9 e si risponda alle seguenti domande/esercizi.

▷ **Che tipo di protocollo di livello Data-link è utilizzato? Come fa Wireshark a capirlo?**

► **Soluzione.** Con il livello *Data-link* si intende il livello 2, cioè 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
```

Grazie alla libreria **libpcap**, Wireshark riesce a 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:



- 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.

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

- ▷ **Disegnare la PDU di livello *Data-link* indicando il valore dei vari campi.**
- ▶ **Soluzione.** Nella risposta alla precedente domanda, è possibile vedere sia il protocollo che il suo *header*.
- ▷ **Qual è il MAC sorgente? Di che tipo è: *unicast* o *broadcast*?**
- ▶ **Soluzione.** Guardando la figura della prima risposta è possibile visualizzare il MAC sorgente: `00:e0:81:24:dd:64`. Si deduce che è di tipo *unicast*, poiché il bit di indirizzo individuale (quello meno significativo) del primo byte è 0.
- ▷ **Che tipo di protocollo di livello *Network* è utilizzato? Come fa Wireshark a capirlo?**
- ▶ **Soluzione.** 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

```

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

▷ **Qual è la lunghezza dell'header IP?**

► Soluzione. Come si vede dalla foto sopra, il campo **Header Length** comunica che la lunghezza dell'header è di 20 *bytes*.

▷ **Quali sono gli indirizzi IP sorgente e destinazione?**

► Soluzione. Come si vede dalla foto sopra, l'IP sorgente è 157.27.252.223 e l'IP destinazione è 157.27.252.255.

▷ **Che tipo di protocollo di livello trasporto è contenuto in IP? Come fa Wireshark a capirlo?**

► Soluzione. 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.

▷ **Quali sono le porte sorgente e destinazione a livello trasporto?**

► Soluzione. A livello di trasporto il pacchetto contiene i seguenti dati:

```

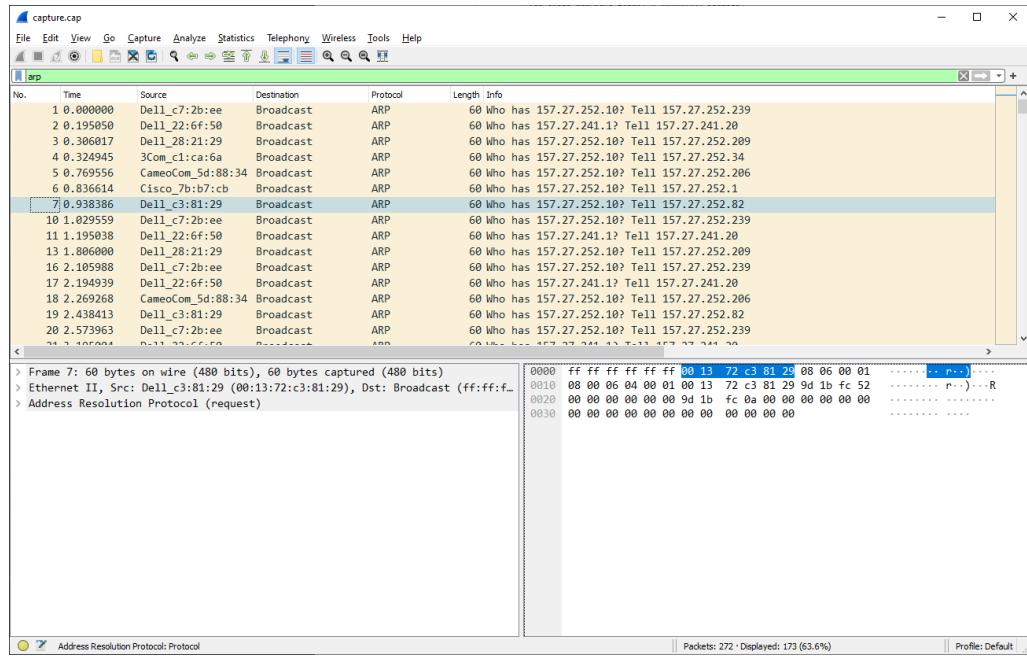
> Frame 9: 227 bytes on wire (1816 bits), 227 bytes captured (1816 bits)
> Ethernet II, Src: TyanComp_24:dd:64 (00:e0:81:24:dd:64), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
> Internet Protocol Version 4, Src: 157.27.252.223, Dst: 157.27.252.255
▼ User Datagram Protocol, Src Port: 631, Dst Port: 631
  Source 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

```

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

▷ **Creare un filtro per visualizzare solo i pacchetti che hanno ARP come protocollo. Suggerimento: basta scrivere arp nella barra Filter sotto la toolbar (si ricorda di premere su **Apply** dopo aver scritto arp).**

► Soluzione. La creazione di un filtro è possibile eseguirla scrivendo nella barra in alto. Il risultato dell'esercizio:



► 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.

► Soluzione. La percentuale di pacchetti che rimangono è pari al 63.6%, cioè 173 pacchetti su un totale di 272.

► 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 ricorda di premere su Apply dopo aver creato l'espressione.

► Soluzione. L'indirizzo MAC è nel protocollo Ethernet. Di conseguenza 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 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.

► 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.

► Soluzione. La percentuale è del 0.4% e cioè 1 pacchetto.

► 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.

► Soluzione. Per visualizzare i pacchetti con destinazione *broadcast* si applica lo stesso filtro della domanda 12, ma con indirizzo di destinazione *broadcast*, ossia:

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

► Dopo aver applicato il filtro precedente, qual è la percentuale di pacchetti che rimangono visualizzati rispetto al totale? Sono molti? Perché?

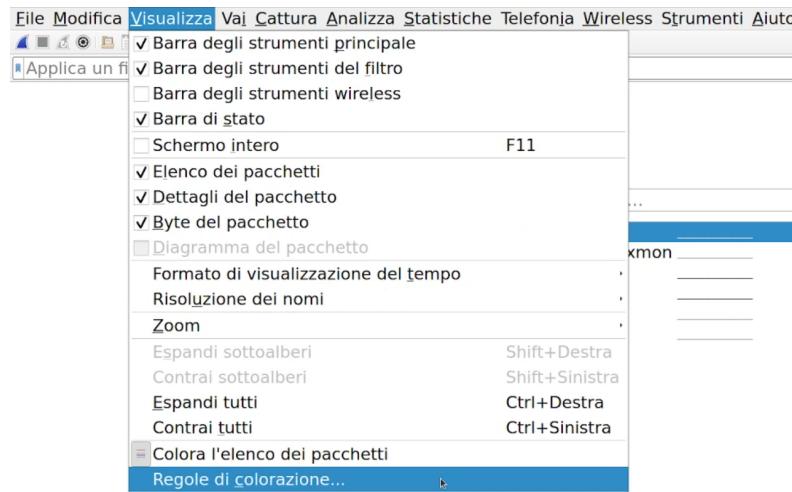
► Soluzione. Il protocollo ARP viene utilizzato per conoscere gli indirizzi MAC degli altri *host* e per conoscere i MAC in una rete privata. Quindi questo protocollo lavora principalmente sul canale *broadcast* dato che deve aggiornare spesso le varie tabelle di ARP. Il **risultato** post filtro è di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

10.7.2 Esercizio 2 - File simpleHTTP.cap

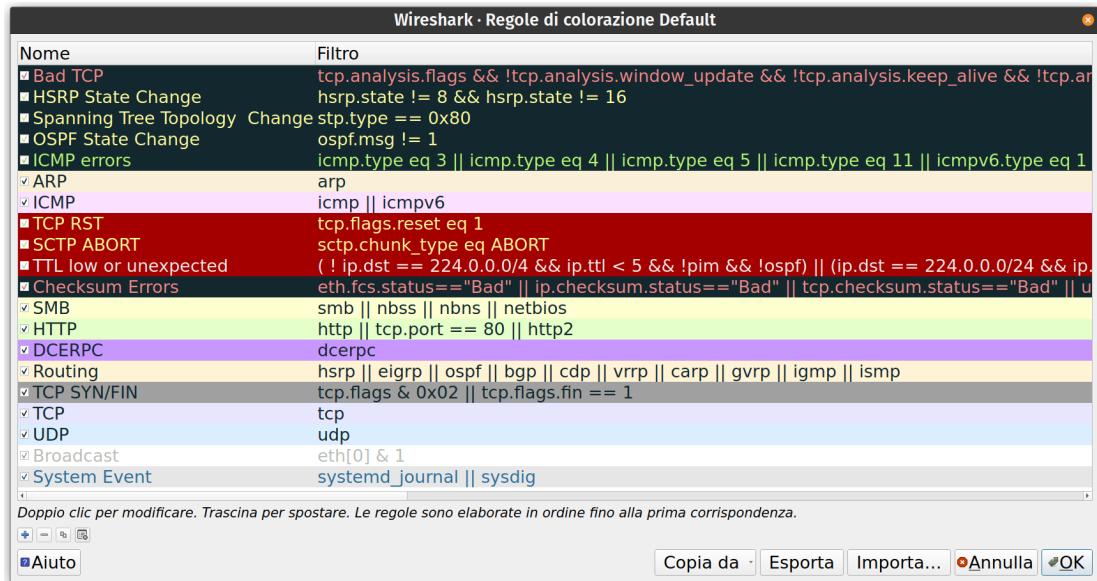
Occorre aprire il menu **File/Apri** e selezionare il file **simpleHTTP.cap**.

► 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.

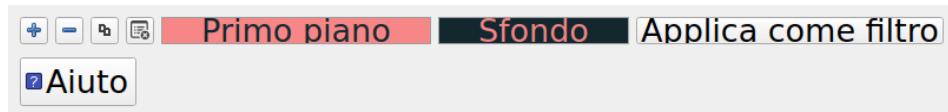
► Soluzione. È possibile modificare il colore dei pacchetti cambiando le regole di colorazione dal menù. Cliccando su **Visualizza/Regole di colorazione...:**



All'apertura della schermata:



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



▷ 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.**

► Soluzione. 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 vasta libreria e alla possibilità di analizzare il pacchetto.

Il primo pacchetto ha all'interno del *payload* una *query* al DNS per capire l'indirizzo IP del sito web www.polito.it. Diversamente il secondo pacchetto ha all'interno del *payload* la risposta alla *query* (si veda l'immagine sotto):

```

    < 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]
```

- *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.
- *Additional records*: informazioni in più utili al mittente per trovare la destinazione.

▷ 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?

► Soluzione. 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*.

▷ 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?

► Soluzione. 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 a 1, il secondo ha le due *flag* SYN e ACK a 1 e infine il terzo pacchetto ha la *flag* ACK a 1.

▷ **Creare un filtro per visualizzare solo i pacchetti TCP (compresi i pacchetti HTTP) e determinarne il numero.**

► Soluzione. 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.

▷ **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?

► Soluzione. Il filtro da applicare è `tcp && !http`. Il numero di pacchetti trovati al punto precedente è di 673 su 823, che corrisponde al 81.8%.

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.

▷ **Selezionare il pacchetto 3 e seguire lo *stream* TCP col comando da menuù Analizza/Segui/Flusso TCP.**

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

► Soluzione. 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.

10.7.3 Esercizio 3 - File busyNetwork.cap

Occorre aprire il menù File/Apri e selezionare il *file* busyNetwork.cap.

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

► Soluzione. 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 **Statistiche/Gerarchie di protocolli** e il risultato che si ottiene è il seguente:

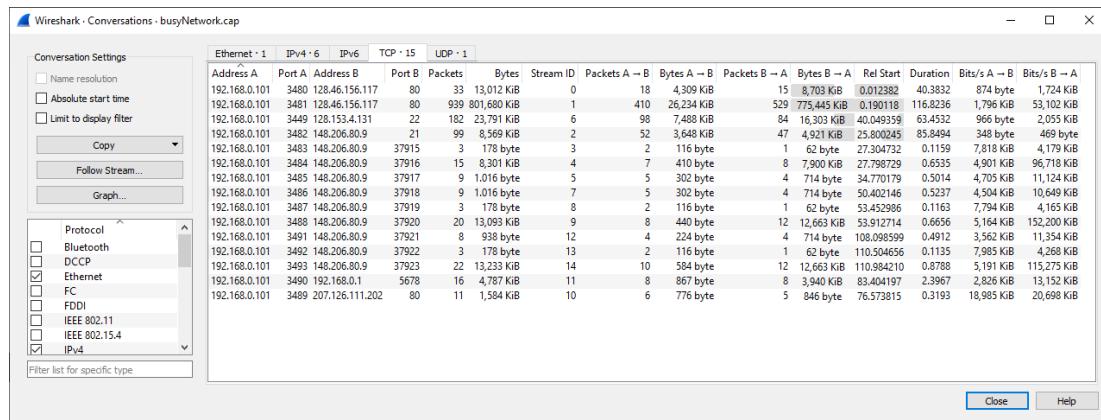
Protocol	Percent Packets	Bytes	Bits/s	End Packets	End Bytes	End Bits/s	PDUs
	Percent Packets	Bytes	Bits/s	End Packets	End Bytes	End Bits/s	PDUs
Frame	100.0	1392	9.8	89337	6107	0	0
Ethernet	100.0	1392	2.1	19634	1342	0	0
Internet Protocol Version 4	100.0	1392	3.0	27840	1903	0	0
User Datagram Protocol	0.3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

▷ **Provare ad analizzare diversi *stream* TCP con sopra diversi protocolli di livello applicazione.**

► Soluzione. 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 **Statistiche/Conversazioni** e apparirà la seguente schermata:



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

► Soluzione. 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*, di conseguenza non è possibile leggerne il contenuto.

10.7.4 Esercizio 4 - File pingCapture.cap

Occorre aprire il menù **File/Apri** e selezionare il file pingCapture.cap.

► Individuare le richieste ping inviate e le relative risposte. Quante sono?

► Soluzione. 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.

▷ Quali sono IP sorgente e destinazione della richiesta ICMP? A quale ente o azienda sono intestati?

► Soluzione. L'IP sorgente dell'*host* che esegue le richieste è 157.27.143.46. Mentre l'IP destinazione dell'*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.

▷ 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 l'RTT medio e la sua variazione. Chi mostra la media più grande? Perché? Chi mostra la variazione più grande? Perché?

► Soluzione. Eseguendo il comando ping www.google.com, si ottiene un RTT medio di 10 ms. Invece eseguendo il comando ping e inserendo l'IP del proprio *default gateway* (ottenibile eseguendo il comando ipconfig su Windows), il tempo medio RTT è di 2 ms. 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*.

10.7.5 Esercizio 5 - Comando traceroute

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

▷ Individuare le interfacce dei *router* attraversati.

► Soluzione. 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.

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

► Soluzione. 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-11-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.

10.7.6 Esercizio 6 - Interfacce di rete

▷ **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?**

► Soluzione. 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*.

▷ **Qual è l'indirizzo IP di `www.univr.it`?**

► Soluzione. 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.

10.8 Esercizi Docker

Dopo la *build* dell’immagine, si esegue il *container* con il comando:

```
docker run -d -p 5000:5000 docker-flask:1.0
```

- `-d` è utilizzato per eseguire l’immagine in un *container* separato.
- `-p` è utilizzato per mappare le porte del *container* con le porte del *host* (*host:container*).
- `docker-flask:1.0` è il *tag* dell’immagine di cui è stata eseguita la *build*.

▷ **Eseguire i seguenti esercizi:**

1. Fermare, riavviare ed uccidere il *container* avviato e visualizzare cosa accade con il comando `docker ps -a`.
2. Rimuovere il *container* e crearne uno nuovo in ascolto sulla porta 8000.
3. Eseguire la stessa immagine in due differenti *container* sulla porta 7000 e sulla porta 8000, rispettivamente.

► Soluzione. Per fermare, riavviare ed uccidere il *container* è possibile farlo grazie ai seguenti tre comandi:

```
docker stop container_id
docker restart container_id
docker kill container_id
```

Dove al posto del `container_id`, deve essere inserito il codice alfanumerico che è possibile ottenere eseguendo il comando `ps -a`, dopo aver eseguito una volta il *container* ovviamente (`run`).

Per il secondo punto è necessario aver capito la sintassi del comando `run`. Prima di tutto si esegue la rimozione del *container* stoppandolo, uccidendolo e infine rimuovendolo:

```
docker stop container_id
docker kill container_id
docker rm container_id
```

È possibile anche eseguire una rimozione forzata senza fermare il *container* (non consigliato):

```
docker rm -r -f container_id
```

Una volta terminato e rimosso il *container*, si procede a rieseguirlo di nuovo ma impostando come porta dell’*host* la numero 8000:

```
docker run -d -p 8000:5000 docker-flask:1.0
```

Accedendo alla pagina `http://localhost:8000` si dovrebbe vedere la pagina *web* d'esempio.

L'ultimo esercizio richiede di eseguire l'immagine in due *containers* diversi in contemporanea. Grazie al punto precedente vi è già in esecuzione un *container* sulla porta 8000. Dunque adesso si provvede a creare un *container* sulla porta 7000 con il seguente comando:

```
docker run -d -p 7000:5000 docker-flask:1.0
```

Così facendo sarà possibile accedere alla pagina web `http://localhost:7000` usando due porte differenti, ovvero la 7000 e la 8000 (`http://localhost:8000`).

10.9 Esercizi MQTT

Installazione di Mosquitto su Linux

Per installare un broker MQTT (Mosquitto) è necessario scrivere la seguente linea di comando:

```
1 sudo apt install mosquitto
```

Per avviare il *broker* si utilizza il comando:

```
1 sudo systemctl start mosquitto
```

Per verificare che il *broker* funzioni si utilizza il comando:

```
1 sudo systemctl status mosquitto
```

Per fermare il *broker* si utilizza il comando:

```
1 sudo systemctl stop mosquitto
```

L'installazione di un Pub/Sub MQTT è eseguito con il comando:

```
1 sudo apt install mosquitto-clients
```

Comandi utili:

Descrizione	Comando
Indirizzo IP broker	-h oppure --host
Topic messaggio	-t oppure --topic
Contenuto messaggio	-m oppure --message

10.9.1 Domande sull'esempio Pub/Sub

Dato l'esempio:

- `mosquitto_sub -t "temperatura";`
- `mosquitto_pub -m "22" -t "temperatura".`

Si risponda alle seguenti domande.

▷ **Partendo dall'esercizio mostrato nelle istruzioni, cosa succede se pubblico una temperatura prima di aver lanciato il subscriber?**

▶ **Soluzione.** Il *publisher* invia il valore 22 con il *topic* temperatura al *broker* attivo, ma la richiesta non viene consegnata a nessun *subscriber* poiché non esistono. Quindi viene pubblicato il messaggio, ma nessuno lo riceve (eccetto il *broker*).

▷ Partendo dall'esercizio mostrato nelle istruzioni, creare un'applicazione Pub/Sub con 2 sensori di temperatura relativi a 2 stanze diverse. Quante finestre di terminale devo aprire?

► Soluzione. Dipende da quanto si vuole isolare ogni componente. Nel caso in cui si utilizzino 3 terminali:

- *Subscribers:*

1. Terminale “archivio” della prima stanza:

```
1 mosquitto_sub -t "temperatura/stanza1"
```

2. Terminale “archivio” della seconda stanza:

```
1 mosquitto_sub -t "temperatura/stanza2"
```

- Publisher:

1. Terminale “sensore di temperatura” delle due stanze. Quindi verrà eseguita una pubblicazione prima di una stanza:

```
1 mosquitto_pub -m "22" -t "temperatura/stanza1"
```

E poi dell'altra:

```
1 mosquitto_pub -m "22" -t "temperatura/stanza2"
```

Altrimenti nel caso in cui si voglia distinguere ogni termostato che pubblica la temperatura, allora:

- Subscribers:

1. Terminale “archivio” della prima stanza:

```
1 mosquitto_sub -t "temperatura/stanza1"
```

2. Terminale “archivio” della seconda stanza

```
1 mosquitto_sub -t "temperatura/stanza2"
```

- Publishers:

1. Terminale “sensore di temperatura” della prima stanza. Quindi verrà eseguita una pubblicazione della stanza numero 1:

```
1 mosquitto_pub -m "22" -t "temperatura/stanza1"
```

2. Terminale “sensore di temperatura” della seconda stanza. Quindi verrà eseguita una pubblicazione della stanza numero 2:

```
1 mosquitto_pub -m "22" -t "temperatura/stanza2"
```

▷ Partendo dall’esercizio precedente, come fare per avere un unico *subscriber* per entrambe le temperature? Come si fa a distinguere da quale stanza proviene la temperatura?

► Soluzione. Per avere un unico *subscriber* è necessario chiudere un terminale dei due *subscriber* attivi e l’unico che rimane, modificarlo con i *wildcards*.

- *Subscriber:*

1. Terminale “archivio unico” di entrambe le stanze:

```
1 mosquitto_sub -t "temperatura/#"
```

- *Publishers:*

1. Terminale “sensore di temperatura” della prima stanza. Quindi verrà eseguita una pubblicazione della stanza numero 1:

```
1 mosquitto_pub -m "22-stanza1" -t "temperatura/stanza1"
```

2. Terminale “sensore di temperatura” della seconda stanza. Quindi verrà eseguita una pubblicazione della stanza numero 2:

```
1 mosquitto_pub -m "22-stanza2" -t "temperatura/stanza2"
```

Come si vede dal messaggio inviato, per capire la stanza dalla quale proviene, una soluzione possibile è modificare il messaggio aggiungendo la stanza di provenienza accanto al valore.

▷ Usare Wireshark per capire quale protocollo viene utilizzato da tutti gli attori a livello 3 e 4. Quali sono le porte di livello trasporto coinvolte?

► Soluzione. Come protocollo viene utilizzato MQTT:

Anche se nella figura viene visto il livello 3 e 4 poiché si ferma solo al livello 2, il protocollo è lo stesso. Le porte utilizzate sono:

- Il *subscriber* utilizza la porta 38120.
- Il *publisher* della stanza 1 utilizza la porta 54474;
- Il *publisher* della stanza 2 utilizza la porta 60688.

No.	Time	Source	Destination	Protocol	Length	Info
12	271.130244883	127.0.0.1	127.0.0.1	MQTT	80	Connect Command
14	271.134996779	127.0.0.1	127.0.0.1	MQTT	70	Connect Ack
16	271.135161276	127.0.0.1	127.0.0.1	MQTT	86	Subscribe Request (id=1) [temperatura/#]
17	271.135218634	127.0.0.1	127.0.0.1	MQTT	71	Subscribe Ack (id=1)
26	303.955968151	127.0.0.1	127.0.0.1	MQTT	80	Connect Command
28	303.956225846	127.0.0.1	127.0.0.1	MQTT	70	Connect Ack
30	303.956355576	127.0.0.1	127.0.0.1	MQTT	99	Publish Message [temperatura/stanza1]
31	303.956450007	127.0.0.1	127.0.0.1	MQTT	68	Disconnect Req
32	303.956531864	127.0.0.1	127.0.0.1	MQTT	99	Publish Message [temperatura/stanza1]
44	318.689325407	127.0.0.1	127.0.0.1	MQTT	80	Connect Command
46	318.689595647	127.0.0.1	127.0.0.1	MQTT	70	Connect Ack
48	318.689752861	127.0.0.1	127.0.0.1	MQTT	99	Publish Message [temperatura/stanza2]
49	318.689810954	127.0.0.1	127.0.0.1	MQTT	68	Disconnect Req
50	318.689911402	127.0.0.1	127.0.0.1	MQTT	99	Publish Message [temperatura/stanza2]

La porta utilizzata dal *broker*, visibile poiché il pacchetto 16, 30 e 48 comunicano con lui e viceversa, è la numero 1883.

► **Prova a pubblicare un valore di umidità relativa (topic “UR”); il subscriber interessato alla temperatura lo riceve? Come si fa a creare un subscriber interessato all’umidità? Costruire un’applicazione Pub/Sub con 4 finestre per produrre e visualizzare sia valori di temperatura sia valori di umidità.**

► **Soluzione.** Il subscriber interessato alle temperature non riceve il messaggio del publisher sull’umidità, poiché il topic a cui è iscritto il subscriber è diverso.

Per creare un subscriber interessato all’umidità, è necessario aprire un terminale e scrivere il comando:

```
1 mosquitto_sub -t "UR"
```

Per costruire un’applicazione pub/sub con 4 finestre:

- *Publishers:*

1. Terminale “sensore di temperatura”. Quindi verrà eseguita una pubblicazione del tipo:

```
1 mosquitto_pub -m "22" -t "temperatura"
```

2. Terminale “sensore di umidità”. Quindi verrà eseguita una pubblicazione del tipo:

```
1 mosquitto_pub -m "5" -t "UR"
```

- *Subscribers:*

1. Terminale “archivio temperature”:

```
1 mosquitto_sub -t "temperatura"
```

2. Terminale “archivio umidità”:

```
1 mosquitto_sub -t "UR"
```

► Si vuole costruire con MQTT un servizio di messaggistica universitaria:

- Il rettore può leggere tutti i messaggi.
- La segreteria può leggere i messaggi dai docenti e dagli studenti.
- I docenti possono leggere i messaggi dai docenti e dagli studenti.
- Gli studenti possono leggere solo i messaggi degli altri studenti.

► Soluzione. Prima di descrivere i comandi da eseguire per creare l'applicazione, si presenta qua di seguito l'albero dei *topic*:

- Livello 1:
 - universita
- Livello 2:
 - universita/personale
 - universita/direzione
- Livello 3:
 - universita/personale/docenti
 - universita/personale/studenti

Adesso si impostano i 4 *subscribers* in 4 terminali diversi:

- Il rettore ha i permessi di leggere qualsiasi messaggio dell'università, quindi deve avere accesso a qualsiasi *topic*:

```
1 mosquitto_sub -t "universita/#"
```

- La segreteria deve avere la possibilità di leggere solo i messaggi provenienti dai docenti e dagli studenti:

```
1 mosquitto_sub -t "universita/personale/#"
```

- I docenti hanno gli stessi permessi della segreteria e quindi il comando sarà:

```
1 mosquitto_sub -t "universita/personale/#"
```

- Gli studenti hanno i permessi più restrittivi di tutti, ovvero possono solo leggere i messaggi provenienti da altri studenti:

```
1 mosquitto_sub -t "universita/personale/studenti"
```