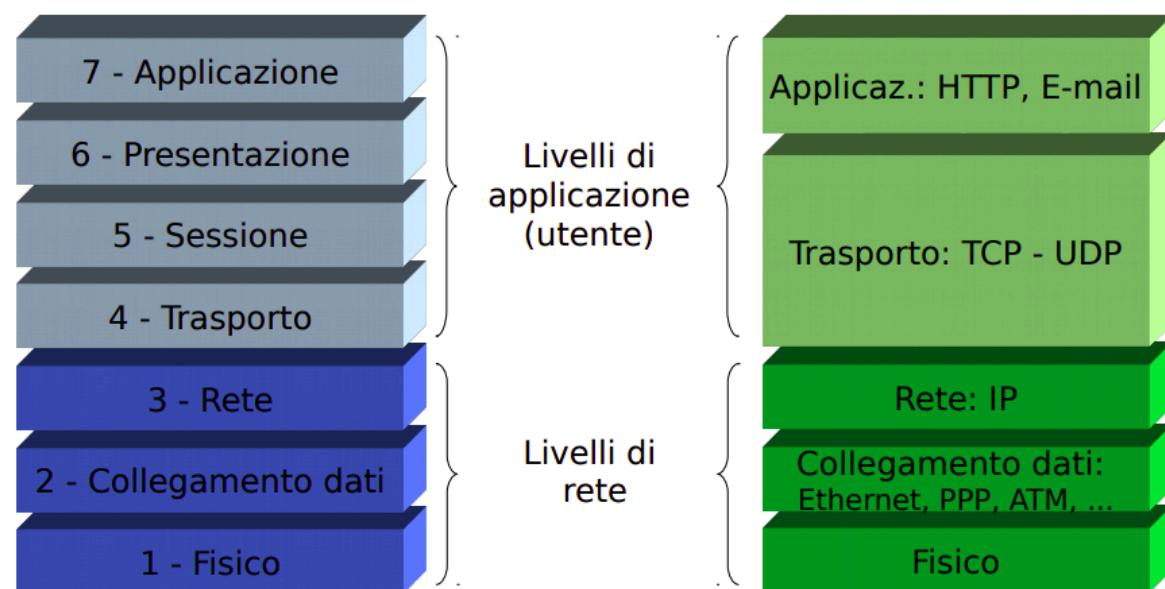


Introduzione ---> Ripasso delle Reti

Stack OSI...

...e Stack TCP/IP



Livelli TCP/IP:

- **Fisico:** trasmissione fisica tramite cavi e bit interpretati da segnali elettrici;
- **Data Link:** scendendo un gradino nella pila protocollare, troviamo il livello data link, il responsabile della comunicazione efficiente e ordinata all'interno della stessa rete fisica. Immaginate questo livello come il sistema che gestisce le conversazioni e gli scambi tra dispositivi che si trovano nella stessa "stanza" o segmento di rete, che sia una LAN tradizionale via cavo (Ethernet) o una rete senza fili (Wi-Fi, Bluetooth). In questo contesto, il modo in cui i dati vengono trasmessi e gestiti è cruciale. Ethernet, ad esempio, un tempo si affidava a meccanismi come il CSMA/CD (Carrier Sense Multiple Access with Collision Detection) per evitare "scontri" di dati sul cavo condiviso. Oggi, con l'avvento degli switch, le collisioni sono quasi un ricordo, poiché ogni dispositivo ha una connessione dedicata al suo interno, rendendo la comunicazione più diretta ed efficiente. Il Wi-Fi, invece, operando via etere, non può "sentire" le collisioni mentre trasmette e continua a usare un approccio come il CSMA/CA (Collision Avoidance), dove i dispositivi "ascoltano" e aspettano il loro turno per parlare, cercando di prevenire le collisioni anziché rilevarle. A questo livello, i dati viaggiano raggruppati in unità chiamate PDU (Protocol Data Unit), che sono fondamentalmente "pacchetti" o "raggruppamenti di bit/byte". Per assicurarsi che ogni pacchetto arrivi al destinatario giusto all'interno di questa stessa rete locale, si lavora

principalmente con gli indirizzi MAC. Ogni dispositivo di rete ha il suo indirizzo MAC, un identificatore unico di 48 bit, solitamente rappresentato come 6 coppie di caratteri esadecimali separati da due punti (es. 00:1A:2B:3C:4D:5E). Possiamo pensare al router come alla "porta" che connette la nostra "stanza" (la rete locale) al mondo esterno. All'interno di questa porta, ci sono delle "maniglie", le interfacce gateway (in particolare quella di default), che sono i punti di ingresso e uscita dalla stanza stessa. Quando un pacchetto viene inviato all'interno della rete locale, il suo primo campo nell'intestazione è proprio l'indirizzo MAC del destinatario. Questo è un meccanismo estremamente efficiente: le schede di rete di tutte le macchine presenti sulla stessa LAN "leggono" solo questo primo campo. Se l'indirizzo MAC non corrisponde al proprio, possono ignorare il pacchetto immediatamente e senza alcuno sforzo ulteriore di lettura o elaborazione da parte del sistema operativo (SO). Questo evita inutili sprechi di tempo e risorse computazionali, garantendo che solo la macchina destinata debba "aprire" e processare quel pacchetto specifico;

- **Rete:** connessione tra diverse reti locali, tramite instradamento e indirizzi IP (protocollo v4 32 bit, v6 128 bit);
- **Trasporto:** esegue un enumerazione per il trasporto di PDU che devono essere divisi in sotto porzioni gestendo, tramite protocolli come TCP (non dire che è sicuro, bensì affidabile) e UDP, la perdita di PDU e il loro recupero;
- **Applicazione:** questo livello ne ingloba 3 dell'OSI perché troppi livelli introducono complessità ed inefficienza inutile (più header, più costi e più tempi). Attraverso il sistema di porte introdotte dal livello di trasporto, siamo in grado di far comunicare i precisi processi (il DNS è un protocollo di questo livello e non del livello di rete, ed è un esempio di tecnologia che utilizza UDP).

N.B: un protocollo è una convenzione; più un pacchetto scende tra i livelli più si gonfia per via degli header.

STRUMENTI DI ANALISI DELLA RETE

Esistono diversi strumenti SW che consentono di analizzare ed eseguire diagnostica di PDU che arrivano sulla propria interfaccia di rete:

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

Noi chiaramente useremo il terzo. Le principali funzionalità di questa libreria sono la possibilità di cercare e trovare interfacce di rete, la gestione avanzata di filtri di cattura e la gestione degli errori e statistiche di cattura.

Catturano fisicamente i PDU e li interpretano. Le due funzionalità (cattura e interpretazione) sono distinte, ossia si può catturare il traffico e analizzare in un secondo momento, allo stesso modo analizzare del traffico precedentemente catturato.

Mentre l'interpretazione avviene ad un livello più astratto e che permette all'utente una visualizzazione e una personalizzazione migliore, la parte di cattura coinvolge elementi di basso livello come le interfacce di rete (ricordarsi che sono molteplici wifi-bluetooth, eth., NFC, USB-C 3.0, ...). In particolare la 'lo' (loopback) è una rete fittizia sempre presente che rende possibile la comunicazione tra processi della stessa macchina come fossero su diverse reti.

In particolare la parte che esegue la cattura (o sniffing) dei PDU è svolta da una libreria a parte scritta in C chiamata libpcap, che scavalca il SO prendendosi una copia di tutte le PDU in arrivo su una determinata interfaccia di rete prima che il SO ignori ed elimini i PDU che normalmente non sarebbero destinati alla macchina in uso. Pertanto necessita i privilegi di root (sudo), dato che deve lavorare a stretto contatto con l'hw (attivazione flag di mod. promiscua -- vedi primo punto). Esistono due macro-casi per questa operazione:

- sniffing all'interno di reti non-switched: in questa tipologia di reti il mezzo trasmissivo è condiviso e, quindi, tutte le schede di rete dei PC ricevono tutti i PDU, anche quelli destinati ad altri. I propri, invece, sono selezionati a seconda del MAC. Lo sniffing, in questo caso, consiste nell'impostare sull'interfaccia di rete la cosiddetta **modalità promiscua** che disattiva il "filtro hardware" basato sul MAC. 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, invece, l'apparato centrale della rete (definito 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 PDU destinati al proprio indirizzo, i PDU multicast e quelli broadcast. L'impostazione della modalità promiscua è, pertanto, insufficiente per poter intercettare il traffico in una rete gestita da switch.

In WSH ci sono due tipi di filtri:

- filtro di cattura: si utilizza associando un'espressione booleana sulla base dei campi dei PDU; avviene prima della cattura effettiva, in quanto lavora ancora una volta insieme alla modalità promiscua volta ad alleggerire il costo della cattura. In questo modo, i PDU che non rispettano l'espressione richiesta vengono completamente ignorati. Per iniziare una cattura basta andare sul menù a tendina 'cattura' e selezionare 'opzioni'; da qui selezionare l'interfaccia desiderata e sulla parte bassa un filtro di cattura per scremare eventuali pacchetti indesiderati;
- filtro di visualizzazione: a differenza del precedente, i PDU ormai sono stati catturati (post cattura), ma serve solo nella mera visualizzazione/ricerca di determinati PDU (non eliminando quindi i PDU che non rispecchiano l'espressione di questo filtro); legato a questo, è possibile anche colorare i PDU per una migliore visualizzazione sulla base di determinate caratteristiche del PDU stesso;

Comandi di utilizzo generale da terminale:

- **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 ICMP (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);
- **traceroute**: il comando traceroute è, invece, 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. 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 (il protocollo è sempre ICMP). Come per le telefonate spam che cercano di capire se il nostro numero di telefono è ancora attivo per venderlo e fare pubblicità, il protocollo ICMP è spesso bloccato dai

network administrator per motivi di sicurezza (in particolare riguardo l'intasamento della rete);

- **nslookup:** il comando nslookup consente di effettuare una interrogazione ai server DNS per poter ottenere da un hostname il relativo indirizzo IP, o viceversa. Si può utilizzare in due modalità interattivo e non interattivo. DNS (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. Nello specifico la 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. La modalità non interattiva invece permette di effettuare una sola interrogazione visualizzandone il risultato, e si abilita ogni qualvolta si specifichi l'host-to-find;
- **ifconfig:** il comando ifconfig 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;
- **route:** il comando route è utilizzato per visualizzare e modificare le tabelle di routing;
- **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 (non incappare nella somiglianza con nslookup, qui si inserisce solo il sito senza www, mentre nell'altro si).

ESERCITAZIONE

PARTE 1 (file capture.cap):

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

RISPOSTA:

Il protocollo utilizzato è Ethernet. Lo possiamo vedere dal primo campo della sezione 'Frame' del (in generale dal livello inferiore a quello interessato è contenuto il campo che specifica il protocollo utilizzato dal livello superiore; in questo modo il campo header-length diventa inutile in quanto è standard dato un determinato protocollo);

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

RISPOSTA:

-----+-----+-----+-----+-----+
MAC Destinazione (6B) MAC Sorgente (6B) Type (2B)
-----+-----+-----+-----+-----+

Destinazione: broadcast (ff:ff:ff:ff:ff:ff);

Mittente: 00:e0:81:24:dd:64;

Type: IPv4 //indica il protocollo del livello superiore;

NB: quando apri un pacchetto visualizzi per righe i vari livelli TCP/IP del pacchetto partendo da quello fisico e andando in giù.

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

RISPOSTA:

Lo ho già scritto prima, è di tipo unicast;

4. Qual è il MAC destinazione? Di che tipo è: unicast o broadcast?

RISPOSTA:

Lo ho già scritto prima, è di tipo broadcast;

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

RISPOSTA:

Il protocollo utilizzato è IPv4. Lo possiamo capire da uno dei campi indicati nel livello inferiore, in questo caso il campo Type del Data-Link;

6. Qual è la lunghezza dell'header IP?

RISPOSTA:

20 bytes;

7. Quali sono gli indirizzi IP sorgente e destinazione?

RISPOSTA:

L'indirizzo mittente è 157.27.252.223 mentre l'indirizzo destinatario è 157.27.252.255;

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

RISPOSTA:

Il protocollo del livello di trasporto è UDP, ed è capibile dal relativo campo nel livello di rete;

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

RISPOSTA:

Porta 631 per entrambe le domande;

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

RISPOSTA:

Fatto;

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

RISPOSTA:

Pacchetti totali 272, visualizzato dopo il filtro sul protocollo ARP 173 (63,6%);

12. Creare un filtro per visualizzare solo i pacchetti che hanno destinazione MAC 00:01:e6:57:4b:e0. (suggerimento: usare l'editor di espressioni; la categoria da selezionare è Ethernet; per 13 l'indirizzo MAC usare la notazione esadecimale con i due punti come separatori; si ricordi di premere su Apply dopo aver creato l'espressione).

RISPOSTA:

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

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

RISPOSTA:

1 su 272, ossia il 0.4%;

14. Creare un filtro per visualizzare solo i pacchetti che hanno destinazione MAC broadcast. (suggerimento: nell'editor di espressioni la categoria da usare è Ethernet; per l'indirizzo MAC usare la notazione esadecimale con i due punti come separatori; si ricordi di premere su Apply dopo aver creato l'espressione).

RISPOSTA:

filtro: eth.dst == ff:ff:ff:ff:ff:ff

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

RISPOSTA:

Sono 228 su 272, ossia il 83,8% del totale. Sono molti perché la maggior parte delle richieste sono volte a scoprire chi ha un determinato indirizzo IP (ARP request);

PARTE 2 (file simpleHTTP.cap):

1. Colorare di rosso tutti i pacchetti che contengono UDP e di verde tutti i pacchetti che contengono TCP. (suggerimento: nell'editor delle regole di colorazione è sufficiente portare in alto due regole già esistenti e modificarle per cambiarne i colori di sfondo).

RISPOSTA:

Su 'Visualizza' trovi la dicitura 'Regole di colorazione', seleziona il tipo di protocollo da colorare e poi in fondo premi su 'sfondo e scegli il colore', poi ok.

2. Cosa contengono i primi due pacchetti della sessione di cattura? IP sorgente, IP destinazione. Tipo di protocollo di trasporto. Tipo di protocollo di livello Applicazione. Come fa Wireshark a capirlo? Messaggio contenuto nel Payload di livello applicazione.

RISPOSTA:

- Pacchetto 1:
 1. Source Address: 157.27.252.202
 2. Destination Address: 157.27.10.10
 3. UDP;
 4. DNS;

- 5. c'è scritto;
- 6. www.polito.it;
- Pacchetto 2:
 - a. Source Address: 157.27.10.10
 - b. Destination Address: 157.27.252.202
 - c. UDP;
 - d. DNS;
 - e. c'è scritto;
 - f. 130.192.73.1

3. Prendere in considerazione il pacchetto n.3. per 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?

RISPOSTA:

Il protocollo del livello di trasporto è TCP, e non sono collegati con i pacchetti precedenti in quanto erano di una connessione con protocollo UDP. L'unico collegamento che si può fare è semantico e a livello di applicazione, in quanto i primi due pacchetti erano una richiesta DNS per l'IP del sito www.polito.it, mentre questo terzo pacchetto è una richiesta di connessione TCP proprio a quel sito (ossia indirizzo 130.192.73.1).

4. Creare un filtro per visualizzare solo i pacchetti TCP (compresi i pacchetti HTTP) e determinare il numero.

RISPOSTA:

Filtro: `tcp && http ---> 134 su 823 (16,3%)` con condizione AND, oppure Filtro: `tcp || http` con OR abbiamo `807 su 823 (98,1%)`;

5. 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?

RISPOSTA:

TCP trasporto, HTTP applicazione. Lo scambio di pacchetti è il three-hands-shake per avviare la comunicazione affidabile. Le flag sono SYN, SYN-ACK e ACK.

5. 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?

RISPOSTA:

Filtro: `tcp && !http --->` 81,8% sul totale, sono tutti three-hands-shake. Se anche la richiesta DNS avesse usato TCP, i pacchetti totali sarebbero stati 5 (3 per l'apertura connessione e i 2 già presenti); no, non sarebbe servito a nulla in quanto la richiesta al DNS di un dominio non richiede l'utilizzo di un protocollo affidabile come TCP;

6. Selezionare il pacchetto 3 e seguire lo stream TCP col comando da menu Analyze/Follow TCP Stream. Cosa si può leggere? Qual è il messaggio contenuto nel payload della PDU di livello Applicazione?

RISPOSTA:

Analizza - Segui - TCP stream, permette di vedere le richieste HTTP (GET) più nel dettaglio. In particolare nel payload è possibile vedere tutto il codice HTML della pagina polito.it richiesta, compresa di immagini e quant'altro.

PARTE 3 (file busyNetwork.cap):

1. Elencare i protocolli di livello Applicazione che entrano in azione in questa cattura classificandoli in base al livello Trasporto utilizzato.

RISPOSTA:

HTTP (usa TCP), DNS (usa UDP), FTP, SSH;

2. Provare ad analizzare diversi stream TCP con sopra diversi protocolli di livello applicazione. Che differenza c'è tra il contenuto trasmesso in una connessione TCP per il protocollo FTP e quello trasmesso per il protocollo SSH?

RISPOSTA:

Per FTP (file transport) la porta usata è la 21 e la maggior parte dei dati trasmessi è facilmente visibile una volta catturato il traffico, mentre per SSH (secure shell) la porta è la 22, e i dati sono tutti crittografati, quindi illeggibili senza chiave di decriptazione.

PARTE 4 (file pingCapture.cap):

1. Individuare le richieste ping (pong) inviate e le relative risposte. Quante sono?

RISPOSTA:

basta mettere nel filtro ‘icmp’ e sono 22;

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

RISPOSTA:

univr;

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

RISPOSTA:

--- www.google.com ping statistics ---

10 packets transmitted, 10 received, 0% packet loss, time 9014ms

Per trovare invece il default Gateway immettiamo il comando ‘ip route’ e vediamo ‘default via 192.168.1.1’. Eseguendo ping --->

--- 192.168.1.1 ping statistics ---

11 packets transmitted, 11 received, 0% packet loss, time 10019ms

PARTE 5 ():

Entrare nel sistema Linux presente in cloud e digitare il comando traceroute www.google.com. Individuare le interfacce dei router attraversati. Individuare i nomi delle organizzazioni a cui sono intestati gli IP delle interfacce dei router attraversati.

RISPOSTA:

Dopo esserci connessi alla vpn dell’uni, ho avviato ‘traceroute www.google.com’ , e come risultato ho ottenuto:

```
1 ***  
2 10.252.10.1 (10.252.10.1) 48.837 ms 48.778 ms 48.657 ms  
3 198.51.100.129 (198.51.100.129) 48.730 ms 51.017 ms 48.602 ms  
4 ru-univr-l1-rl1-vr00.vr00.garr.net (193.204.218.109) 48.478 ms 48.434 ms  
48.397 ms  
5 ***  
6 rs1-mi01-re1-mi02.mi02.garr.net (185.191.180.158) 48.167 ms 48.282 ms  
48.174 ms  
7 142.250.164.230 (142.250.164.230) 48.143 ms 48.071 ms  
142.250.174.46 (142.250.174.46) 48.041 ms  
8 192.178.104.191 (192.178.104.191) 48.976 ms 192.178.104.103  
(192.178.104.103) 48.537 ms *
```

9 142.251.235.177 (142.251.235.177) 48.405 ms 192.178.82.62
(192.178.82.62) 48.461 ms 142.251.235.177 (142.251.235.177) 48.308 ms
10 142.251.235.177 (142.251.235.177) 48.263 ms mil04s51-in-f4.1e100.net
(142.251.209.36) 48.184 ms 142.251.235.179 (142.251.235.179) 49.466 ms
di cui le interfacce sono quelle esplicitate nelle parentesi dopo l'IP di
riferimento del router attraversato. Per le aziende associate basta lanciare l'ip
del gestore con whois;

PARTE 6():

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

RISPOSTA:

Lancio nel terminale 'ifconfig' e vediamo subito le mie interfacce attive, ossia:

- wlo0 ---> connessione wifi;
- lo ---> loopback;
- enx527ac5ca44bc ---> non ne ho idea;
- eno1 ---> connessione ethernet;

Al momento sto utilizzando wlo0 e il mio indirizzo IP è 172.20.10.4, mentre la
netmask corrispondente è 255.255.255.240. Infine con nslookup www.univr.it
è possibile risalire a 157.27.3.60, ossia l'indirizzo IP dell'uni.

N.B: in WSH vediamo sulla dicitura protocol il nome del protocollo di
livello più alto che WSH stesso è riuscito a trovare e individuare. Quindi
se vediamo per esempio TCP, significa che il pacchetto non ha (da solo
magari unito ad altri sì) una parte di livello applicativo interpretabile allo
stato attuale. Questa nota è particolarmente importante per esempio nel
caso in cui leggessimo la flag FYN, comunemente associata al
protocollo TCP, in un pacchetto interpretato come HTTP- Web Socket. In
questo caso non è la flag FYN di chiusura connessione (ossia, lo è del
Web Socket ma non della connessione intera TCP), ma potrebbe proprio
significare completamente un'altra cosa in base al protocollo a cui fa
riferimento.

Interfaccia Socket

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:

1. Client/server;
2. Publisher/Subscriber (Pub/Sub).

Client/Server:

E' la situazione più frequente. Il client fa sempre il primo passo con una richiesta, mentre il server fa il secondo passo e manda la risposta, rimettendosi poi in attesa di altre richieste. La richiesta del client può essere una richiesta di un dato oppure la trasmissione di un dato (cioè la richiesta di prendere in consegna un certo dato).

Quello che determina il ruolo di client e server è l'ordine dei messaggi e non il contenuto. In un'applicazione client/server, il client è quindi colui che fa il primo passo indipendentemente dal fatto che chieda o trasmetta un dato.

Dobbiamo anche fare attenzione ai nomi: client e server sono processi e non host; l'insieme di almeno un client e un server costituisce l'applicazione di rete. In particolare si può dire che un certo processo gioca/interpreta il ruolo di client o server all'interno dell'applicazione.

Durante l'esercitazione non useremo direttamente l'interfaccia socket, bensì una libreria sviluppata da un ex studente durante la sua tesi che incapsula l'utilizzo diretto della socket per non andare eccessivamente a basso livello (function wrapped).

Il programma, prima di usare la rete, deve creare un oggetto di tipo socket. Il socket è l'insieme di chiamate a funzione (punto di comunicazione) identificato da 3 parametri, ossia indirizzo IP locale, porta locale, modalità di trasmissione UDP oppure TCP. Funziona come end-point, ossia in una connessione c'è una socket sia in un host che nell'altro. Il programma server deve decidere esplicitamente il numero di porta locale affinché i client possano saperlo (la porta serve a indicare quale processo, l'IP quale host); i numeri da 0 a 1023 sono riservati a protocolli applicativi noti. Il programma che tuttavia crea un oggetto socket su queste porte deve avere i privilegi di root. Il programma client può deciderlo esplicitamente oppure lasciarlo decidere al proprio sistema operativo.

[UDP:](#)

Ogni pacchetto scambiato tra gli host è logicamente indipendente dai precedenti e successivi. Le perdite di pacchetti non vengono compensate in automatico né dalla rete né dal sistema operativo. Per questo motivo l'UDP è più leggero del TCP come uso di risorse di calcolo e di rete, ma il tipo di applicazione deve essere compatibile con questo tipo di comportamento (esempio, richiesta DNS). Se il server non ha nome occorre chiedere ad un utente sul server di scoprire e comunicarci il suo indirizzo IP, mentre lato server, siccome una risposta parte sempre dopo la relativa richiesta, l'indirizzo IP del client, così come la porta (notare che nel file infatti non viene specificato 20.000 nel server) è contenuto nelle informazioni di mittente della richiesta.

[TCP:](#)

E' byte oriented ossia non c'è correlazione a livello di programmazione tra il taglio di lettura e scrittura dei byte e l'effettivo pacchetto trasmesso. In altre parole, il sender e la sua specifica di quanti byte mandare il pacchetto, è scorrelato dal come il receiver decide di bufferizzarli e riceverli. E' utilizzata quando tra i pacchetti trasmessi c'è una relazione: sono parte di un "messaggio più grande" (ad es. un'immagine o un PDF). L'oggetto socket si preoccupa di numerare i pacchetti appartenenti alla stessa connessione per rilevare eventuali pacchetti persi e poterli ritrasmettere. Come vantaggio, l'utente scrive/legge su un archivio remoto con la stessa naturalezza di quando scrive/legge su un archivio locale come se la rete in mezzo non ci fosse. Come svantaggio invece, gli host trasmettitore e ricevitore devono "lavorare" di più dentro il sistema operativo, e i pacchetti persi e ritrasmessi arrivano in ritardo e questo può essere compatibile oppure no con il tipo di applicazione.

ESERCITAZIONE

Esercizi basati su clientUDP e serverUDP

- 1) Lanciare prima il server e poi il client. Cosa si osserva? Invertire la sequenza di lancio. Cosa si osserva?
- 2) Modificare i sorgenti per mettere il server che **riceve** sulla porta 10000 e il client che **trasmette** dalla propria porta 30000 (ogni modifica dei sorgenti richiede una loro ricompilazione)
- 3) Mettere il server in ascolto sulla porta 100 e osservare cosa succede
 - Bisogna modificare anche il client? Dove?
 - Per chi usa il proprio PC con Linux o una virtual machine Linux, lanciare il server con il comando "sudo ./serverUDP" e osservare cosa cambia
- 4) Sostituire "127.0.0.1" prima con "localhost" e poi con "pippo" e osservare cosa succede
- 5) [Da fare solo se in Lab Delta] Accordarsi per lavorare su coppie di macchine in modo che server e client siano su macchine diverse. Come bisogna modificare i sorgenti?
- 6) Lanciare due volte il server usando due terminali. Cosa si osserva? Funzionano entrambi?
- 7) Modificare il server in maniera che soddisfi 5 richieste prima di terminare
 - E se volessi che non terminasse mai?

RISPOSTA:

N.B: durante il processo, è possibile visualizzare per bene lo scambio di pacchetti desiderato attraverso Wireshark e il filtro "`ip.src === 127.0.0.1 && ip.dst === 127.0.0.1`";

1. se il server è aperto la richiesta viene soddisfatta; nel caso opposto, la richiesta client rimane in attesa e non verrà mai soddisfatta, nemmeno facendo partire il server effettivamente;
2. non avviene connessione dato che la porta è diversa;
3. bisogna modificare il client per far combaciare le porte;
4. localhost è la stessa cosa di quell'IP, pippo non funziona perchè non è una connessione;
5. bisogna mettere stessa porta e l'ip della connessione
6. il secondo server termina subito probabilmente perchè vede che è già aperto;
7. fatto, basta aggiungere un `while(true)` prima della ricezione UDP;

[Esercizi basati su clientUDP_inc.c e serverUDP_inc.c:](#)

- 1) Compilare ed eseguire il secondo esempio;
- 2) Modificarlo per costruire una semplice sommatrice – Il client acquisisce ripetutamente da tastiera un numero intero e lo manda 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;

RISPOSTA:

In sostanza i file originali hanno un client che chiede un numero all’utente, lo manda al server, che come sempre va eseguito prima, che lo incrementa di uno e lo rimanda indietro.

Per modificare il file è stato sufficiente aggiungere un while response != 0, creare una somma cumulativa nel server, o continuare a chiedere input all’utente nel client.

- 3) impossibile da fare senza ethernet;
- 4) Invocare il server della sommatrice con due client diversi (tutti e tre possono anche essere sulla stessa macchina ovviamente su finestre terminali diverse). Che somma leggo da ciascun client? E’ la somma che ciascun client da solo si aspetterebbe?

RISPOSTA:

Il primo client che termina riceve la somma cumulativa ottenuta da entrambi i client, mentre il secondo rimane in attesa infinita.

[Esercizi basati su clientTCP.c e serverTCP.c](#)

- 1) Lanciare due volte il server usando due terminali. Cosa si osserva? Funzionano entrambi?
- 2) Scrivere la sommatrice usando TCP, compilare ed eseguire
- 3) Provare a rifare il caso dell’Esercizio 10 ma con questa nuova versione della sommatoria. Cosa si può osservare? Che soluzione si può trovare? C’è influenza reciproca tra i due client?

Risposta 1:

Il secondo server lanciato si chiude subito probabilmente perché la funzione di creazione del socket per prima cosa fa un controllo per la porta richiesta, e se già aperta killa il processo;

Risposta 2:

Fatto! chiaramente c'era da separare fuori dal ciclo while le richieste di apertura, la sua accettazione, le chiusure TCP e l'invio della risposta dal server al client.

Risposta 3:

Il primo client che si collega riceve effettivamente la somma cumulativa corretta, relativa ai soli dati inviati dallo stesso. Il secondo client invece (verificabile facilmente dagli output del server), invia i dati ma il server non ne salva la somma (ossia proprio non lo "caga"); l'unica cosa che funziona effettivamente è l'apertura della connessione, quindi lo scheletro iniziale del processo (già la chiusura non funziona correttamente). Come si vede dalle img, la risposta al secondo client in realtà è un errore di calcolo interno al client stesso probabilmente.

```
[SERVER] Sono in attesa di richieste di connessione da qualche client  
[SERVER] Connessione instaurata  
[SERVER] Ho ricevuto la seguente richiesta dal client: 2  
[SERVER] Ho ricevuto la seguente richiesta dal client: 2  
[SERVER] Ho ricevuto la seguente richiesta dal client: 2  
[SERVER] Ho ricevuto la seguente richiesta dal client: 0  
[SERVER] Invio la risposta al client
```

```
[CLIENT] Creo una connessione logica col server  
[CLIENT] Inserisci un numero intero:  
1  
[CLIENT] Invio richiesta con numero al server  
[CLIENT] Inserisci un numero intero:  
9  
[CLIENT] Invio richiesta con numero al server  
[CLIENT] Inserisci un numero intero:  
0  
[CLIENT] Invio richiesta con numero al server  
[CLIENT] Ho ricevuto la seguente risposta dal server: 10
```

```
[CLIENT] Creo una connessione logica col server  
[CLIENT] Inserisci un numero intero:  
5  
[CLIENT] Invio richiesta con numero al server  
[CLIENT] Inserisci un numero intero:  
2  
[CLIENT] Invio richiesta con numero al server  
[CLIENT] Inserisci un numero intero:  
0  
[CLIENT] Invio richiesta con numero al server  
[CLIENT] Ho ricevuto la seguente risposta dal server: -518651456
```

[Esercizi basati su clientTCPChar.c e serverTCPChar.c](#)

Per questi file non ci sono esercizi da svolgere. In ogni caso, il client invia un char ‘A’ e il server risponde con un char ‘B’. Estremamente statico.

[Esercizi basati su clientTCPIO.c e serverTCPIO.c](#)

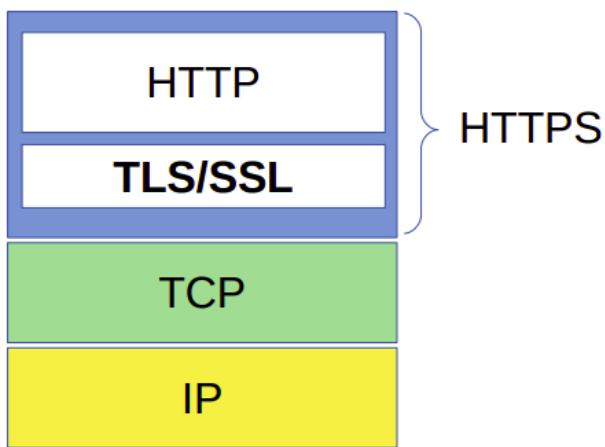
1. Il client richiede al server un file specificandone il nome. Il server lo trasmette un byte alla volta. Il client lo salva in locale con lo stesso nome. Quale protocollo usiamo?

Quando devi trasferire un file come in questo caso, specialmente se è piccolo, hai bisogno che i dati arrivino in ordine, completi, senza duplicazioni, e che l'intera comunicazione sia affidabile. Ho quindi creato un server e un client che sfruttano le funzioni TCPSender e TCPReceiver della libreria network.h/c, nonchè le sys call open/read/write per la gestione del file, per far sì che: client richieda nome file all'utente, lo mandi al server che attende il nome, carica il file e lo legge, mandandolo indietro al client un byte alla volta che a sua volta lo stampa su terminale.

WEBSERVICE

HTTP/HTTPS:

Nato per la fruizione di contenuti in rete (World Wide Web), oggi è usato anche per l'invocazione di funzionalità remote (Webservice). I messaggi che passano nella connessione TCP sono gli stessi dell'HTTP ma vengono sottoposti a cifratura dei dati in transito e autenticazione del server mediante certificato digitale. Il server lavora sulla porta 443 invece che 80.



Esempio di HTTP/HTTPS

- Il client si chiama "web browser" (o semplicemente "browser")
 - Firefox, Chrome, Edge, Safari, Opera, ...
- Il server si chiama "web server"
 - Apache, NGINX, NodeJS
- La comunicazione avviene sul protocollo TCP sulla rete Internet



Questo protocollo è testuale, di tipo client/server formata da:

1. apertura connessione TCP;
2. (se HTTPS) autenticazione del server e negoziazione di una chiave di cifratura;
3. richiesta;
4. risposta;
5. chiusura connessione TCP.

Problema cybersec: nel DNS avviene (solitamente) un controllo IP-nome_host; potrebbe essere infatti che per frode un server si "spacci" per un altro, mentre io client penso di comunicare con qualcuno che invece non è (ma che magari ha scopiazzato la parte estetica del web server originale a posta per truffare);

NB: si può forkare apache per gestire più porte;

HTML:

Il corpo della risposta HTTP può contenere:

- HTML puro;

- HTML + codice Javascript;
- Sequenza binaria che rappresenta un'immagine;
- Cascading Style Sheets (CSS);
- Intere librerie di codice Javascript da eseguire sul browser.

HTML è un linguaggio testuale di descrizione di una pagina, una specializzazione del generico XML (eXtensible Markup Language) , basato su “tag” annidati (costrutto ad albero) eventualmente contenenti attributi.

Il DOM (Document Object Model) è una rappresentazione in forma di albero dell'HTML di una pagina, che il browser crea in memoria quando carica il sito. Ogni elemento HTML (come <div>, <p>, , ...) diventa un nodo di quest'albero e può essere letto, modificato o eliminato dinamicamente tramite JavaScript. L'HTML base invece è semplicemente il codice statico scritto dall'utente.

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

<https://www.univr.it/servizi/studenti/carriera>

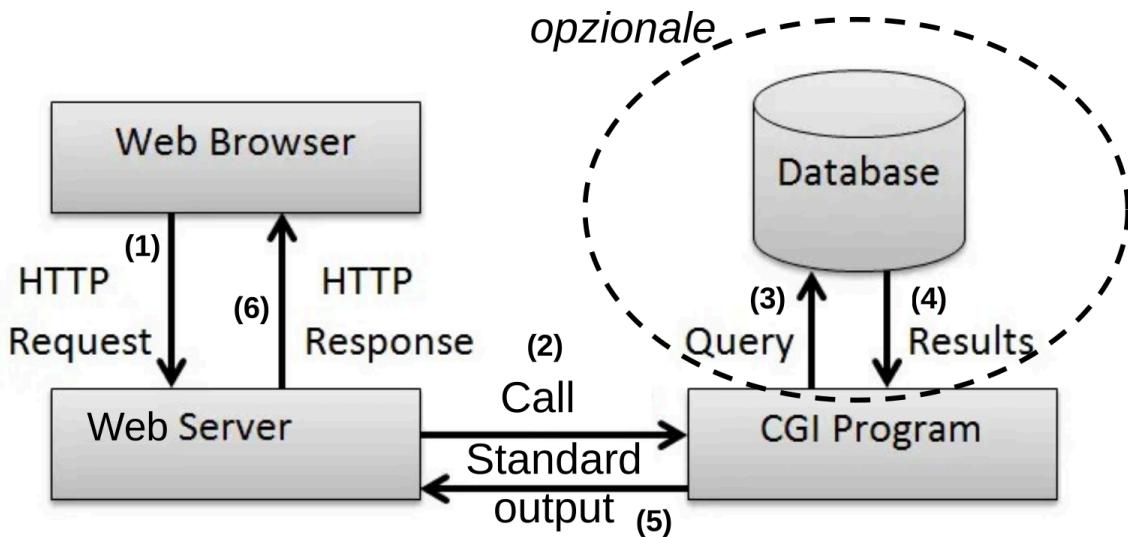
Struttura:

- per prima cosa il protocollo utilizzato a livello applicazione, che di solito include implicitamente anche il protocollo di livello trasporto + la porta utilizzata (per esempio, HTTP →TCP/80, HTTPS→TCP/443);
- segue nome dell'host (o indirizzo IP) che eroga tale risorsa;
- infine nome della risorsa con suo percorso logico completo (non necessariamente fisico).

La porta può essere indicata esplicitamente se non è quella standard:

<https://www.univr.it:8000/servizi/studenti/carriera>

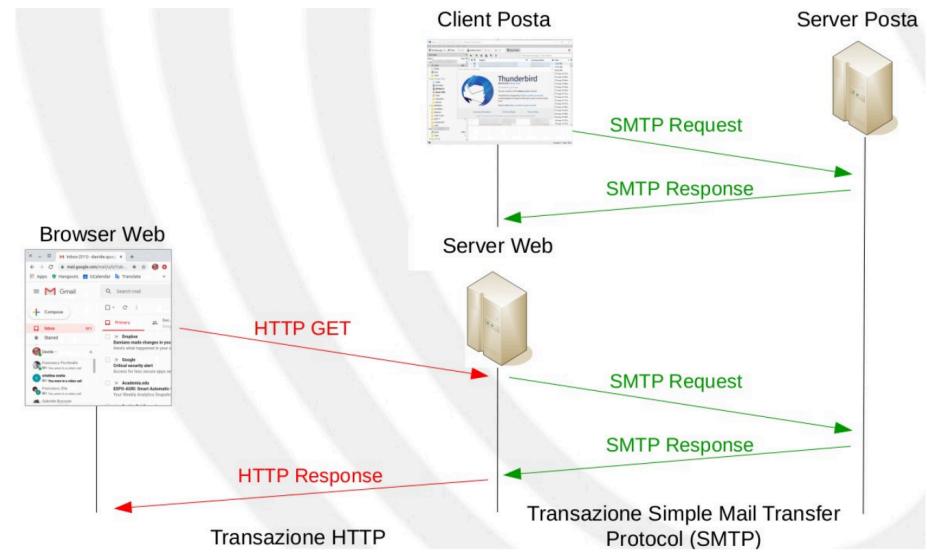
All'inizio il web (detto primo web) era quindi composto da semplici richieste HTTP statiche. Poi, con l'avvento del CGI (common gateway interface) si è aggiunta la possibilità di fare richieste dinamiche. La richiesta del client è sempre la stessa, mentre il server accoglie la richiesta, e vi esegue delle elaborazioni, appunto, dinamiche. Possono essere query in database, elaborazione complessa di dati, ecc. ecc. Il server dunque invoca delle chiamate dietro le quinte (da qui nasce il “backend”) verso qualsiasi cosa (il client comunque non sa nulla), e ritorna il risultato finale al client.



L'esempio più stupido possibile è che il server accumuli un contatore ogni volta che il client si collega al server; quel conteggio è dinamico e non statico, per tanto è una prima forma di backend a tutti gli effetti.

Dunque, il programma CGI viene eseguito “lato server”. Allo stesso modo, per alcune specifiche operazioni del server, esso può diventare a sua volta un client di molte applicazioni di rete, per esempio per:

- Content Management Systems (CMS);
- Posta elettronica;
- Wiki.



Web-socket:

I WebSockets rappresentano un protocollo di livello applicazione che si pone come un'alternativa significativa a HTTP/HTTPS, progettato specificamente per abilitare una comunicazione simmetrica e persistente tra il browser web (client) e il server.

A differenza del modello richiesta-risposta di HTTP, dove il client deve sempre avviare la comunicazione, una volta stabilita una connessione WebSocket, entrambi i processi (client e server) possono prendere l'iniziativa per inviare dati all'altro in qualsiasi momento. Questo elimina il sovraccarico di dover aprire e chiudere una nuova connessione HTTP per ogni scambio di dati, rendendo la comunicazione estremamente efficiente e a bassa latenza.

La connessione WebSocket nasce tipicamente da una sessione HTTP/HTTPS preesistente attraverso un'operazione chiamata "Protocol Upgrade". È come se il browser e il server, dopo un primo "saluto" via HTTP, decidessero di passare a un linguaggio di comunicazione più diretto e continuo per tutto il resto della sessione. Questa capacità di comunicazione bidirezionale e in tempo reale è fondamentale per molte applicazioni moderne.

Un esempio è il refresh asincrono di una pagina web: invece di richiedere al client di "tirare" continuamente per verificare se ci sono nuovi dati, il server può "spingere" (push) proattivamente nuove informazioni o aggiornamenti alla pagina non appena questi diventano disponibili. Questo rende possibili esperienze utente molto più dinamiche e reattive, ideali per chat in tempo reale, giochi multiplayer, dashboard live e notifiche istantanee.

SOA (SERVICE ORIENTED ARCHITECTURE):

Tradizionalmente, le applicazioni venivano concepite come soluzioni "monolitiche". Immaginiamo un unico, grande programma installato direttamente sul computer dell'utente: tutte le funzionalità, dall'interfaccia grafica ai calcoli più complessi, erano strettamente legate e contenute in un singolo eseguibile che "girava" interamente sulla macchina locale. La Service-Oriented Architecture (SOA) ha rappresentato un cambio di paradigma significativo. Invece di un blocco unico, le applicazioni complesse sono ora realizzate combinando diversi programmi che comunicano tra loro, non più all'interno dello stesso sistema, ma attraverso la rete. Con SOA, l'interfaccia utente e alcune funzionalità di base (spesso definite come un "client leggero") risiedono ancora sul dispositivo dell'utente. Tuttavia, il vero cuore dell'applicazione, ovvero le sue funzionalità principali e più elaborate, è fornito da programmi che risiedono su uno o più server remoti. In questo contesto, un Servizio può essere inteso come una specifica funzionalità, concettualmente simile a una chiamata a funzione o a un metodo negli ambienti orientati agli oggetti.

L'adozione di un'architettura SOA porta con sé una serie di notevoli vantaggi. La potenza di calcolo e la memoria non gravano più sul dispositivo dell'utente, ma sono delegate ai

server, permettendo all'applicazione di gestire carichi di lavoro maggiori e di mantenere elevate prestazioni. Questo approccio offre anche una maggiore protezione della proprietà intellettuale, poiché gli algoritmi strategici e il codice critico rimangono residenti sui server. Un altro beneficio cruciale riguarda gli aggiornamenti del software: non c'è più la necessità di distribuire nuove versioni agli utenti ogni volta che le modifiche riguardano solo il codice sul lato server, semplificando enormemente la manutenzione. La SOA ha inoltre aperto la strada a nuovi modelli economici, come il "pay-per-use" (pagamento basato sull'utilizzo effettivo del servizio), e contribuisce significativamente all'eliminazione della pirateria, dato che il cuore dell'applicazione risiede sul server e viene fruito come servizio.

Nonostante i numerosi benefici, l'architettura SOA presenta anche delle sfide intrinseche. Il più evidente è che l'infrastruttura di rete diventa un elemento essenziale e non opzionale. Di conseguenza, se la connessione di rete viene a mancare o è instabile, l'applicazione cessa di funzionare, rappresentando un vincolo significativo per la disponibilità del servizio.

Al centro della SOA risiede un concetto fondamentale, la chiamata di funzione remota. Questo modello è perfettamente conforme al classico paradigma client/server, ma con una complessità e una flessibilità maggiori dettate dalla distribuzione su rete.

Ogni funzione, sia essa locale o remota, è definita da alcuni elementi essenziali:

- nome, che ci indica la sua finalità;
- tipo e il numero dei parametri in ingresso che accetta;
- tipo del valore che restituisce;
- implementazione, ovvero il codice che ne definisce il comportamento.

L'interfaccia di una funzione è proprio la sua "carta d'identità": una descrizione di nome, parametri e tipo di valore di ritorno. Questa interfaccia può essere fornita da una libreria locale, come nelle applicazioni tradizionali, o, nel contesto della SOA, da un server remoto. L'insieme di queste funzioni o metodi esposte da una libreria (locale) o da un server (remoto) prende il nome di Application Program Interface (API).

In un'architettura SOA, il componente server espone una API che descrive una serie di funzioni invocabili da un componente client, che in questo specifico contesto, non è un semplice web browser. L'implementazione vera e propria di queste funzioni risiede interamente sul server. Ciò che rende la SOA così potente è la sua agnosticità tecnologica: il codice che invoca la funzione sul client e quello che la implementa sul server possono essere scritti in linguaggi di programmazione completamente diversi e girare su architetture di calcolo molto differenti senza alcun problema di compatibilità. Per far sì che questa comunicazione avvenga senza intoppi, entrano in gioco due componenti chiave:

- sul lato client, la funzione chiamata dall'applicazione apparentemente realizza la funzionalità desiderata. In realtà, il suo compito principale è codificare i parametri di input in un formato adatto per la trasmissione in rete e, una volta ricevuta la risposta, decodificare il valore di ritorno. Questa funzione speciale sul client, che implementa la stessa interfaccia della funzione remota, è chiamata STUB;
- sul lato server, esiste un componente chiamato SKELETON. Questo si occupa di decodificare i parametri di input che arrivano dalla rete, di passarli alla funzione che contiene l'implementazione logica vera e propria (spesso definita BUSINESS LOGIC), di prendere il risultato generato da quest'ultima, di codificarlo e infine di spedirlo indietro al client.

Per il trasporto effettivo di questi dati codificati attraverso la rete, è necessario un protocollo di rete (come TCP/IP o HTTP). Il processo di codifica dei dati per la trasmissione è spesso definito SERIALIZZAZIONE, mentre il processo inverso di decodifica è la deserializzazione.

Sono quindi state sviluppate diverse tecnologie e standard, ognuno con le proprie caratteristiche e ambiti di applicazione:

- Remote Procedure Call (RPC): RPC è un concetto fondamentale che permette a un programma di eseguire una procedura (o subroutine) su un computer remoto come se fosse locale. Sebbene sia un concetto più generale, è alla base di molte delle tecnologie successive;
- Java Remote Method Invocation (Java RMI): specifico per l'ecosistema Java, Java RMI consente a un oggetto in una macchina virtuale Java (JVM) di invocare metodi su un oggetto che risiede in un'altra JVM, anche se su una macchina diversa. Utilizza TCP come protocollo di trasporto sottostante, sfruttando le capacità native di Java per la serializzazione degli oggetti.
- Common Object Request Broker Architecture (CORBA): CORBA è uno standard indipendente dal linguaggio di programmazione e dal protocollo di trasporto di livello inferiore. Permette a oggetti software scritti in linguaggi diversi e distribuiti su diverse piattaforme di comunicare tra loro. Storicamente, CORBA ha mirato a fornire un interoperabilità universale, spesso utilizzando TCP per il trasporto dei messaggi;
- Web Services: i Web Services rappresentano uno degli approcci più diffusi e standardizzati per l'implementazione della SOA, sfruttando ampiamente i protocolli e le tecnologie del web:

- HTTP/HTTPS vengono utilizzati come protocollo di trasporto principale per scambiare gli elementi della funzione (richieste e risposte). Questo li rende facilmente accessibili attraverso firewall e reti esistenti;
- Per il formato dei dati scambiati, si sono affermati due approcci principali:
 - Protocollo XML SOAP (Simple Object Access Protocol): questo è stato uno dei primi e più robusti standard per i Web Services. Utilizza XML per la messaggistica ed è spesso associato a protocolli più complessi per la descrizione dei servizi (WSDL) e la loro scoperta (UDDI). Sebbene sia molto potente e fornisca funzionalità avanzate (come transazioni e sicurezza a livello di messaggio), è percepito come pesante e talvolta datato a causa della verbosità di XML e della sua complessità;
 - Metodologia REST (Representational State Transfer): attualmente è l'approccio più usato e preferito per la costruzione di Web Services moderni, spesso chiamati API RESTful. REST è uno stile architettonico che sfrutta appieno i principi di HTTP, trattando le risorse come URL e utilizzando i metodi HTTP standard (GET, POST, PUT, DELETE) per operare su di esse. Il formato dei dati più comune per le API REST è JSON, grazie alla sua leggerezza e facilità di parsing, sebbene possa supportare anche XML o altri formati.

I Web Services basati su REST rappresentano oggi l'approccio predominante per la costruzione di API e servizi distribuiti. La loro efficacia deriva dal saper sfruttare appieno la semplicità e la robustezza del protocollo HTTP/HTTPS come veicolo principale per la comunicazione. In questo modello, la chiamata a una funzione remota viene gestita in modo intelligente:

- il nome della funzione non è più un riferimento diretto a una procedura, ma viene mappato su una URL (Uniform Resource Locator), identificando la "risorsa" su cui si vuole operare;
- il passaggio dei parametri avviene in due modi principali: o direttamente nella URL (tipico per metodi GET o DELETE), oppure inclusi nel corpo della richiesta, ovvero dopo l'header HTTP (come accade con i metodi POST o PUT);
- la scelta del metodo HTTP da utilizzare è cruciale e segue una semantica ben definita legata all'azione che si intende compiere sulla risorsa:
 - POST è impiegato per funzioni che creano un NUOVO oggetto sul server;
 - PUT è destinato a funzioni che aggiornano un oggetto ESISTENTE sul server;

- GET è utilizzato per recuperare informazioni di un oggetto ESISTENTE sul server;
- DELETE serve per le funzioni che hanno il compito di distruggere un oggetto esistente sul server.
- Il valore di ritorno della funzione remota viene generalmente inserito nel corpo della risposta HTTP;
- Un'evoluzione significativa rispetto ai protocolli più datati è la sostituzione di HTML come formato di scambio dati. Nei Web Services REST, il contenuto del corpo della risposta è tipicamente testo puro o, molto più frequentemente, JSON (JavaScript Object Notation), grazie alla sua leggerezza e facilità di elaborazione.

Metodi della Richiesta HTTP:

Il protocollo HTTP definisce diversi metodi che indicano l'azione desiderata sulla risorsa identificata dalla URL. I più comuni, già menzionati nel contesto REST, sono:

- GET
- POST
- PUT
- DELETE

L'adozione dei Web Services, in particolare quelli basati su REST, porta con sé notevoli benefici:

- compatibilità con l'Infrastruttura Internet Esistente: L'intera infrastruttura di Internet è già nativamente predisposta all'uso di HTTP/HTTPS. Questo significa che i Web Services funzionano senza problemi attraverso elementi come i firewall e le soluzioni NAT (Network Address Translation), che altrimenti complicherebbero la comunicazione;
- facilità di Debugging: L'utilizzo di contenuti testuali (come JSON o testo puro) nelle transazioni rende estremamente più semplice ispezionare e comprendere il flusso di dati, facilitando notevolmente il debugging delle applicazioni SOA;
- flessibilità del Client: Lo stesso insieme di servizi può essere richiamato e utilizzato sia da un programma client dedicato (come un'applicazione desktop o mobile) sia direttamente da un browser web. Un esempio calzante è quello dei Web Services associati a un'applicazione di Internet Banking: gli stessi servizi possono essere invocati sia dal sito web bancario sia dall'app mobile dedicata, garantendo coerenza e riuso del backend.

JSON:

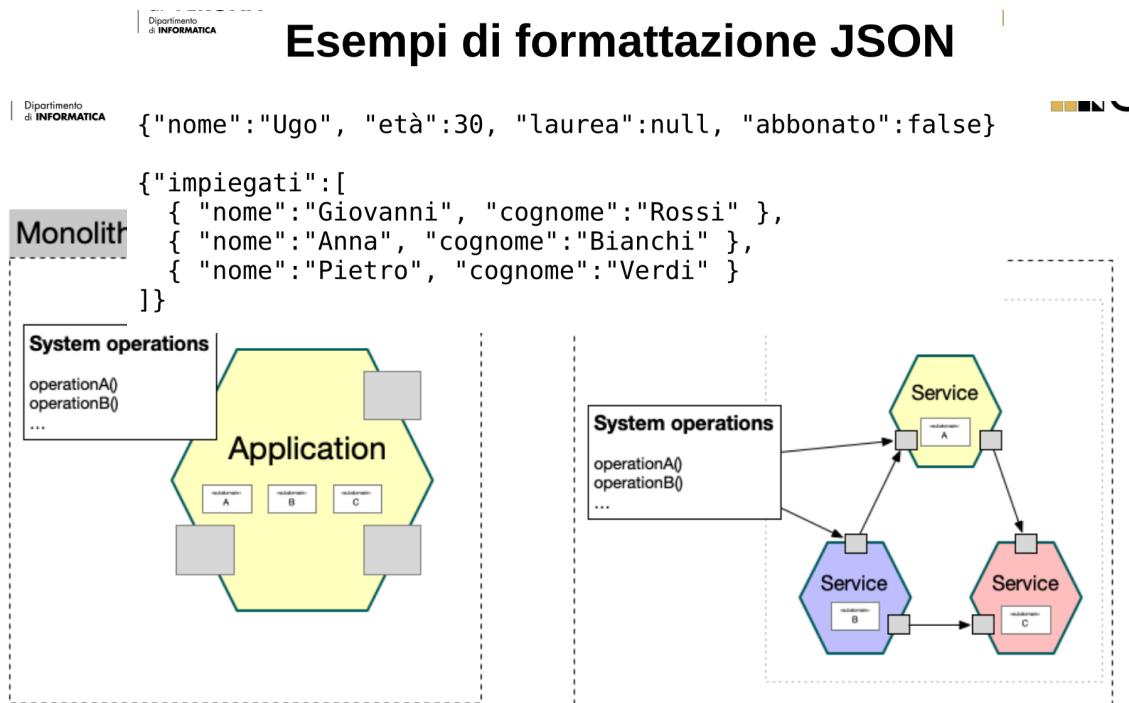
JSON (JavaScript Object Notation) è un formato di dato testuale estremamente versatile, nato in origine con JavaScript ma che oggi gode di un supporto universale in praticamente tutti i linguaggi di programmazione. La sua popolarità deriva dalla capacità di combinare leggibilità umana con una facilità di parsing eccezionale per i programmi.

La sua struttura si basa su una gerarchia intuitiva, fondata sull'elemento base: la coppia attributo : valore. I tipi di valore che JSON può rappresentare sono semplici e diretti:

- stringhe: sempre racchiuse tra virgolette doppie (es. "nome");
- numeri: interi o decimali (es. 123, 45.6);
- booleani: true o false;
- null: per indicare l'assenza di un valore.

Per organizzare dati più complessi, JSON offre due strutture principali:

- array [...]: permettono la composizione di elementi omogenei, rappresentando liste o sequenze di valori (es. ["mela", "pera", "banana"]);
- strutture dati {...} (oggetti): consentono la composizione di elementi eterogenei, mappando attributi a valori per formare oggetti complessi (es. {"nome": "Mario", "età": 30}).



I microservizi rappresentano un'evoluzione delle architetture SOA, spingendo il concetto di modularità al massimo. Anziché un'applicazione composta da pochi servizi di grandi dimensioni, in un'architettura a microservizi, l'applicazione è un insieme di servizi molto piccoli, indipendenti e focalizzati su una singola funzionalità di business.

Vantaggi dei microservizi:

Il principale punto di forza dei microservizi è la loro capacità di rendere estremamente modulare la parte di backend di un'applicazione. Questo porta a numerosi benefici operativi e di sviluppo:

- scalabilità migliorata: ogni micro servizio può essere assegnato a macchine diverse, permettendo di aumentare la scalabilità orizzontale. Se, ad esempio, il servizio di autenticazione è sotto carico elevato, si possono aggiungere più istanze solo di quel servizio, senza dover scalare l'intera applicazione. Questo si lega strettamente al concetto di Load Balancing, che distribuisce le richieste tra le varie istanze per ottimizzare le prestazioni;
- facilità di sviluppo e manutenzione: la dimensione ridotta e l'indipendenza di ogni micro servizio rendono lo sviluppo e la manutenzione molto più agili. I team possono lavorare su singoli servizi in modo indipendente;
- cicli di sviluppo indipendenti: è possibile programmare interventi di manutenzione o l'introduzione di nuove funzionalità su specifici moduli software lasciando attivi gli altri. Questo riduce drasticamente i tempi di inattività e permette rilasci continui e più rapidi;
- riduzione del rischio di effetti collaterali: data l'indipendenza tra i servizi, un errore o un problema in un micro servizio è meno probabile che si propaghi e comprometta l'intera applicazione, isolando così i guasti;
- agnosticismo tecnologico: team diversi possono scegliere le tecnologie e i linguaggi di programmazione più adatti per ogni singolo micro servizio, senza essere vincolati da un'unica scelta per l'intera applicazione.

Svantaggi e complessità dei microservizi:

Nonostante i numerosi pro, l'adozione dei microservizi introduce anche delle complessità significative:

- gestione del database: per massimizzare l'indipendenza dei microservizi, spesso si suddivide il database in più database distinti, uno per ogni servizio o per gruppi di servizi correlati. Questo però elimina la possibilità di eseguire JOIN ottimizzati direttamente a livello di database relazionale, come si farebbe con un unico database monolitico;
- incrocio dati manuale: se i dati necessari per una certa operazione sono distribuiti su database diversi gestiti da microservizi distinti, l'incrocio delle tabelle deve essere implementato manualmente a livello applicativo attraverso chiamate tra i servizi. Ciò richiede un'attenta progettazione per dividere la base dati in punti dove l'esigenza di

incrocio tra diversi database è minima, minimizzando così la complessità e l'overhead di rete;

- complessità operativa e di gestione: un'architettura a microservizi è intrinsecamente più complessa da gestire, monitorare e dispiegare. Richiede strumenti avanzati per l'orchestrazione, il logging, il monitoring e la tracciabilità distribuita;
- comunicazione di rete aumentata: con più servizi che comunicano tra loro, aumenta il traffico di rete e la potenziale latenza tra i componenti.

Realizzare e Mettere Online i Servizi:

Una volta che abbiamo ben chiaro cosa fa ogni servizio e come comunica, possiamo passare alla fase di implementazione. Scegliamo il linguaggio di programmazione che preferisci (Python, Java, Node.js, ecc.) e un buon framework che ti aiuti a creare il server e gestire tutta la logica. Dopodiché, dobbiamo mettere online questi servizi. Questo significa "istanziare" (avviare) i tuoi applicativi server su delle macchine. Pensiamo a come abbiamo fatto con i tre server sulla stessa macchina, su porte diverse (8000, 8001, 8002). In un ambiente reale, questi server potrebbero essere su macchine fisiche diverse, su macchine virtuali o, sempre più spesso, all'interno di container come Docker. L'ottimizzazione iniziale, in questa fase, è più che altro una buona configurazione di base del server per assicurarsi che funzioni senza intoppi. Il backend è progettato e online, ma cosa succede quando arrivano migliaia o milioni di utenti? È qui che l'ottimizzazione diventa cruciale, e il concetto di Load Balancing è il tuo migliore amico. Il Load Balancing è come avere un direttore d'orchestra per il traffico web. Invece di far arrivare tutte le richieste a un unico server, che potrebbe andare in crisi, un Load Balancer le distribuisce intelligentemente tra più istanze del tuo servizio. Immaginiamo di avere tre server che gestiscono gli ordini. Se tutte le richieste arrivano al server A, gli altri due restano a guardare. Un Load Balancer le smista: una al server A, una al B, una al C, e così via. Permette quindi di:

- gestire più utenti: l'applicazione può elaborare molte più richieste contemporaneamente;
- essere più veloce: ogni richiesta viene servita più rapidamente perché il carico è distribuito;
- non andare in crash: nessun server viene sovraccaricato fino al punto di bloccarsi;
- essere sempre disponibile: se un server si guasta, il Load Balancer smette semplicemente di inviar gli richieste e le reindirizza agli altri server funzionanti. Questa è la base dell'alta disponibilità;
- la scalabilità orizzontale: il Load Balancing è la chiave per la scalabilità orizzontale. Invece di comprare un computer sempre più potente (scalabilità verticale, che ha un limite), puoi semplicemente aggiungere più server "normali" e farli lavorare insieme.

Per rendere il Load Balancing davvero efficace in un ambiente dinamico, ci sono due tecnologie fondamentali:

- la containerizzazione (es. Docker): pensa a Docker come a una piccola scatola portatile che contiene la tua applicazione e tutto ciò di cui ha bisogno per funzionare (librerie, configurazioni, ecc.). Questo significa che puoi prendere questa scatola e farla girare su qualsiasi macchina, ed essa si comporterà sempre allo stesso modo. È il modo perfetto per creare repliche identiche e affidabili dei tuoi servizi da distribuire;
- l'orchestrazione (es. Kubernetes): se Docker ti dà le scatole, Kubernetes (o altri orchestratori) è la fabbrica che gestisce quelle scatole. Automatizza il piazzamento delle scatole sui tuoi server, decide quante repliche avviare (anche in base al carico, aggiungendone o togliendone automaticamente), e se una scatola si rompe, la sostituisce subito. È come avere un team di operai robot che gestiscono la tua infrastruttura.

Ci sono diversi modi in cui il load balancer può decidere dove inviare una richiesta:

- servizi diversi su istanze diverse: questo, come accennato, non è "load balancing" in senso stretto, ma piuttosto una divisione del lavoro per servizio. Se il tuo servizio utenti è su un set di server e il tuo servizio prodotti su un altro, il traffico per gli utenti non andrà mai ai server prodotti. Aiuta a distribuire il carico tra i servizi, ma non all'interno di un singolo servizio;
- il load balancer centrale (reverse proxy): questa è la soluzione più diffusa. Hai un unico punto di accesso (un IP o un nome a dominio) che è il tuo Load Balancer. I client si connettono a lui, e lui, come un vigile urbano, smista le richieste ai vari server reali che stanno "dietro le quinte". Il Load Balancer può usare vari criteri per smistare, ad esempio mandare la richiesta al server con meno carico, o distribuire a turno;
 - "session stickiness" (o affinità di sessione): a volte, per applicazioni che mantengono informazioni specifiche dell'utente sul server (come un carrello della spesa o una sessione di login), è utile che tutte le richieste di quello stesso utente vadano sempre allo stesso server. Il Load Balancer può essere configurato per farlo, garantendo che l'utente non perda i suoi dati di sessione. Il rovescio della medaglia è che un server potrebbe ritrovarsi con più "utenti appiccicati" e quindi un carico maggiore;
- load balancing basato su DNS (DNS Round Robin): un metodo più semplice è far sì che il tuo server DNS (quello che traduce nomi come google.com in indirizzi IP) risponda con IP diversi ogni volta che qualcuno chiede l'indirizzo del tuo servizio. Quindi, il primo utente riceve l'IP del server A, il secondo l'IP del server B, e così via.

- il problema del caching DNS: la grande debolezza di questo approccio è che client e altri server DNS in giro per internet tendono a memorizzare queste risposte per un certo tempo (fanno "caching"). Se un server va giù o ne aggiungi di nuovi, i client che hanno in cache la vecchia informazione continueranno a provare a connettersi ai server non più validi o non verranno a conoscenza dei nuovi server fino alla scadenza della loro cache. Questo lo rende meno efficace per la gestione dinamica del carico e l'alta disponibilità.

[Cloud Computing:](#)

Immaginiamo di non dover più comprare, installare e mantenere server fisici nel tuo ufficio o data center. Il Cloud Computing è esattamente questo: un servizio che ti permette di usare risorse di calcolo (server, storage, database, software) tramite internet, senza che tu debba preoccuparti dell'infrastruttura sottostante. È un po' come l'elettricità: non costruisci una centrale elettrica a casa tua, semplicemente attacchi la spina e usi l'energia che ti viene fornita. Nel cloud, paghi per le risorse che consumi, proprio come con la bolletta della luce, oppure con un costo fisso, a seconda del fornitore. Tra i giganti di questo settore troviamo nomi come:

- Amazon Web Services (AWS);
- Google Cloud;
- Microsoft Azure;
- servizi come Dropbox, sebbene non siano piattaforme di calcolo complete, sono un esempio di "archiviazione nel cloud".

Il cloud computing non è magia, ma il risultato di diverse tecnologie mature che lavorano insieme:

- connessione internet costante: è il requisito base. Se non fossimo perennemente connessi, l'idea di accedere a risorse remote sarebbe impraticabile;
- virtualizzazione: questa è la vera star. Permette di far girare più "macchine virtuali" (sistemi operativi completi, ognuno con le sue applicazioni) su un singolo server fisico. In questo modo, i fornitori di cloud possono ottimizzare l'uso delle loro potenti macchine fisiche, affittando "fette" ad n clienti contemporaneamente;
- Web Services: sono il "linguaggio" che permette alle tue applicazioni di sfruttare le risorse di calcolo remote. Tramite API standard (come quelle RESTful che abbiamo visto), il tuo software può chiedere al cloud di eseguire un calcolo, salvare un dato o recuperare un file.

Il cloud non è un servizio unico, ma una scala di opzioni che ti danno più o meno controllo sull'infrastruttura. Vediamole, partendo dal controllo totale fino all'uso puro del software.

- On-site (o On-premise): questo è il punto di partenza, il "non-cloud". Significa che l'hardware è tuo, fisicamente presente nella tua azienda, e devi gestire tutto tu: comprare i server, installare il sistema operativo, le applicazioni, la sicurezza, i cavi;
- Infrastructure as a Service (IaaS): entriamo nel cloud vero e proprio. Con IaaS, è come se affittassi un "terreno" digitale vuoto. Ti viene fornita una macchina virtuale (VM) su cui non c'è nulla, se non l'hardware virtuale. Sei tu a scegliere e installare il sistema operativo (Windows, Linux), e poi tutte le tue applicazioni sopra di esso. Hai molto controllo, ma devi comunque gestire il sistema operativo e ciò che ci gira sopra;
- Platform as a Service (PaaS): Questo è un passo avanti in termini di comodità. In PaaS, non affitti solo il "terreno", ma un "terreno già preparato" con la fondazione. Il fornitore del cloud ti offre già un ambiente di lavoro completo: il sistema operativo è pre-installato, ci sono già i runtime per il tuo linguaggio di programmazione (es. Java, Python) e magari strumenti come Docker. Tu devi solo caricare il tuo codice o le tue applicazioni, senza preoccuparti del server sottostante. È ideale per gli sviluppatori che vogliono concentrarsi solo sul codice;
- Software as a Service (SaaS): questo è il livello con più "chiavi in mano". Qui, è come se affittassi l'uso di un "appartamento già arredato". Non gestisci né l'hardware né il software di base, usi direttamente l'applicazione già pronta. Esempi classici sono Gmail, Dropbox (per l'archiviazione), o servizi cloud che ti offrono direttamente un web server (come Apache o Nginx gestito), un database server (come MySQL o PostgreSQL) o un broker di messaggi (per la comunicazione tra applicazioni), senza che tu debba installarli o configurarli. Li usi e basta.

[Function-as-a-Service \(FaaS\):](#)

Il FaaS è una delle offerte più recenti e interessanti del cloud, simile al SaaS ma con una granularità ancora più fine. Sostanzialmente ti permette di creare e caricare direttamente delle singole funzioni (piccoli blocchi di codice) nel cloud, scegliendo il linguaggio di programmazione che preferisci. Queste funzioni non sono sempre attive, ma vengono associate a degli eventi. Ad esempio, una funzione potrebbe avviarsi quando un'immagine viene caricata su un servizio di storage, o quando arriva un messaggio in una coda.

Vengono eseguite solo quando c'è bisogno, allocando al volo le risorse di calcolo necessarie (CPU, RAM) solo per il tempo di esecuzione della funzione stessa. Il vantaggio enorme è che tu, come utente, non devi preoccuparti nemmeno del processo server che fornisce quella funzione. Non devi creare server virtuali, container, o configurare nulla. Scrivi la tua funzione, la carichi, e il cloud pensa al resto.

Il FaaS è spesso chiamato anche "serverless computing". Questo nome è un po' fuorviante, perché i server ci sono eccome! Semplicemente, tu, l'utente, non li vedi e non li gestisci. È il fornitore del cloud che si occupa di tutta l'infrastruttura sottostante, liberandoti da questo onere.

Tra i servizi commerciali più famosi troviamo:

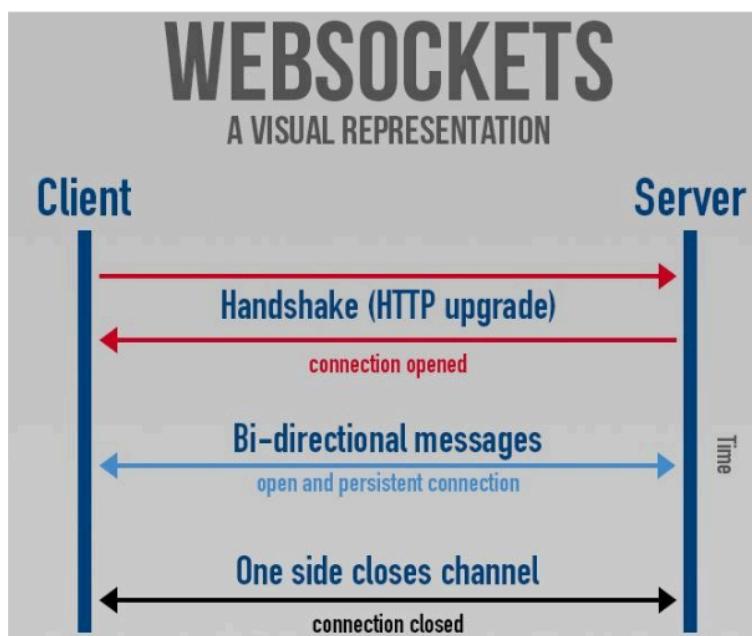
1. AWS Lambda
2. Google Cloud Functions
3. Microsoft Azure Functions
4. IBM Cloud Functions (basato su Apache OpenWhisk)
5. Oracle Cloud Functions

Granularità estrema:

Il FaaS rappresenta una divisione del calcolo ancora più fine rispetto ai microservizi. Con i microservizi hai un intero servizio (es. "gestione ordini"), con FaaS hai una singola funzione all'interno di quel servizio (es. "calcola totale ordine" o "invia email di conferma"). È ideale per gestire task specifici e occasionali con massima efficienza e con costi a consumo molto bassi (paghi solo quando la funzione viene eseguita).

WEB SOCKET

Il WebSocket è un protocollo di livello applicativo che fornisce un canale di comunicazione bidirezionale simmetrico attraverso una singola connessione TCP inizialmente utilizzata per il protocollo HTTP che ha un comportamento bidirezionale ma asimmetrico. Il protocollo WebSocket permette maggiore interazione tra un browser e un server, facilitando la realizzazione di applicazioni web che devono fornire contenuti in tempo reale. Questo è reso possibile poichè i WebSocket permettono al server la possibilità di “prendere l'iniziativa” ed effettuare dei push autonomi di dati verso il browser per aggiornarlo, cosa che non può avvenire con il tradizionale protocollo HTTP.



Un sistema bidirezionale (detto anche full-duplex) permette la comunicazione in entrambe le direzioni simultaneamente. Una buona analogia per il full-duplex potrebbe essere una strada a due corsie con una corsia per ogni direzione. Una connessione TCP è bidirezionale non solo come direzione dei dati ma anche come libertà di entrambi gli agenti coinvolti di trasmettere per primi. Questa possibilità è poi persa nel protocollo HTTP dove è sempre il client a fare il primo passo creando un comportamento asimmetrico. Con i WebSocket si vuole far ritornare simmetrico il rapporto tra i due interlocutori come se fosse una connessione TCP di tipo base col vantaggio che è stata instaurata inizialmente per l'HTTP e quindi è compatibile con molte politiche di sicurezza già presenti su Internet.

I WebSocket sono basati sul protocollo TCP e nascono da una connessione HTTP attraverso una Upgrade request verso il server. Per permettere una compatibilità iniziale tra browser e server, si utilizza l'HTTP Upgrade header' in cui viene richiesto di passare dal

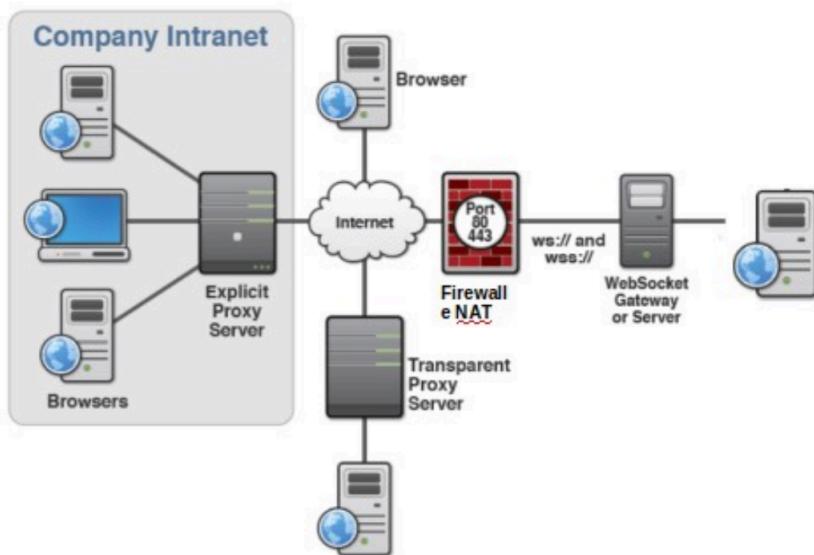
protocollo HTTP a quello WebSocket. Il protocollo HTTP fornisce un meccanismo speciale che permette di stabilire un upgrade del protocollo da usare durante la connessione. Questo meccanismo può essere inizializzato solo dal client, mentre il server, se è in grado di fornire il servizio col protocollo stabilito dal client, decide se accettare o meno questo cambio di protocollo.

Ispezione dell'Upgrade request avvenuto tra browser e server visualizzata attraverso la console di sviluppo del browser (FIGURA):

The screenshot shows the Network tab of a browser's developer tools. A single request is selected, representing an 'Upgrade' event. The request details are as follows:

- General**:
 - Request URL: ws://localhost:4000/socket.io/?EIO=3&transport=websocket&sid=ZGmHcTelj1ENLl-YAAAF
 - Request Method: GET
 - Status Code: 101 Switching Protocols
- Response Headers**:
 - Connection: Upgrade
 - Sec-WebSocket-Accept: CUskFroGUxhoV+mRIGGLDbTo+I=
 - Sec-WebSocket-Extensions: permessage-deflate
 - Upgrade: websocket
- Request Headers**:
 - Accept-Encoding: gzip, deflate, br
 - Accept-Language: it-IT, it;q=0.9, en-US;q=0.8, en;q=0.7
 - Cache-Control: no-cache
 - Connection: Upgrade
 - Cookie: io=ZGmHcTelj1ENLl-YAAAF
 - Host: localhost:4000
 - Origin: http://localhost:4000
 - Pragma: no-cache
 - Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
 - Sec-WebSocket-Key: 3C03ihVDV3u2p8/MzZsjYA==
 - Sec-WebSocket-Version: 13
 - Upgrade: websocket
 - User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.181 Safari/537.36
- Query String Parameters**:
 - EIO: 3
 - transport: websocket
 - sid: ZGmHcTelj1ENLl-YAAAF

Il protocollo WebSocket è supportato attualmente da numerosi browser, inclusi Google Chrome, Internet Explorer, Edge, Firefox, Safari e Opera. Diamo un'occhiata a come i WebSocket interagiscono con gli apparati di rete. Le connessioni WebSocket utilizzano porte HTTP standard (80 e 443). Pertanto, i WebSocket non richiedono l'apertura di nuove porte sulle reti e quindi sono compatibili con le impostazioni di sicurezza dei firewall, proxy web e il meccanismo del NAT. Un'immagine vale più di mille parole. La Figura successiva mostra una topologia di rete semplificata in cui il browser accede ai servizi tramite la porta 80 (o 443) utilizzando una connessione WebSocket full-duplex. Alcuni client si trovano all'interno di una rete aziendale, protetti da un firewall aziendale e configurati per accedere a Internet tramite server proxy che possono fornire memorizzazione e sicurezza del contenuto. A differenza del normale traffico HTTP, che utilizza un protocollo di richiesta/risposta, le connessioni WebSocket possono rimanere aperte per lungo periodo e permettono un scambio paritetico di dati tra browser e server. I firewall, NAT e Proxy eventualmente interposti devono consentire questo tipo di comunicazioni.



Limitazioni:

I WebSockets non rappresentano la soluzione a tutto. L'HTTP riveste ancora un ruolo chiave nella comunicazione tra browser e server come via per inviare e chiudere connessioni per trasferimenti di dati di tipo one-time, come i caricamenti iniziali. Le richieste HTTP sono in grado di eseguire questo tipo di operazioni in modo più efficiente dei WebSocket, chiudendo le connessioni una volta utilizzate piuttosto che mantenendo lo stato della connessione.

Inoltre i WebSocket possono essere utilizzati solo se gli utenti utilizzano i moderni browser con JavaScript abilitato. Questo potrebbe non essere vero in caso di sistemi embedded. Dovrebbero poi essere considerati possibili impatti sull'architettura di rete. WebSocket, essendo una connessione persistente, potrebbe richiedere molte più risorse rispetto ad un server Web standard. Per capire meglio come funziona questa tecnologia si è implementato un breve tutorial in cui si vuole sviluppare una chat online utilizzando la tecnologia offerta dai WebSocket.

Backend e Frontend nello specifico:

L'utilizzo dei WebSocket in un esempio di programmazione event-driven viene mostrato attraverso l'implementazione di una semplice chat in cui ciascun utente, attraverso il proprio client collegato al server, è in grado di mandare messaggi a tutti gli altri. Per la realizzazione di questa chat verranno usati vari strumenti software:

- **JavaScript:** linguaggio di scripting orientato agli oggetti e agli eventi, comunemente utilizzato nella programmazione Web lato client per la creazione di siti web ed applicazioni web;
- **Node.js:** framework event-driven per realizzare applicazioni Web in JavaScript che ci permette di utilizzare questo linguaggio, tipicamente utilizzato nella “client-side”, anche per la scrittura di applicazioni “server-side”;

- HTML: linguaggio di markup. nato per la formattazione e impaginazione di documenti ipertestuali.
- CSS: linguaggio usato per definire la formattazione di documenti HTML, XHTML e XML ad esempio i siti web e relative pagine web;
- Console con ispezione network: monitoraggio da parte del browser dello scambio tra i pacchetti.

Da qui in avanti, quando parliamo del Client intendiamo l'insieme costituito dal Browser e Javascript (spesso delegato quindi alla parte di frontend), mentre quando parliamo del server intendiamo [Node.js](#) (framework per realizzare applicazioni Web in JavaScript, permettendoci di utilizzare questo linguaggio, tipicamente utilizzato nella “client-side”, anche per la scrittura di applicazioni “server-side”).

Approccio Asincrono/Event-driven/Reattivo:

La caratteristica principale di Javascript risiede nella possibilità di accedere alle risorse del sistema operativo in modalità event-driven e non sfruttando il classico modello basato su programmazione sequenziale, processi e thread concorrenti, utilizzato dai classici linguaggi di programmazione. Il modello event-driven, o “programmazione ad eventi”, o “programmazione reattiva” si basa su un concetto piuttosto semplice: si esegue un’azione quando accade qualcosa. Ogni azione quindi risulta asincrona a differenza dello stile di programmazione tradizionale in cui una azione succede ad un’altra solo dopo che la prima è stata completata. Le applicazioni vengono concepite come un insieme di funzioni da eseguire al manifestarsi di certi eventi. Tali funzioni sono chiamate indifferentemente “callback”, “listener” o “event handler”. L’efficienza deriva dal considerare che le azioni tipicamente effettuate riguardano il networking, ambito nel quale capita spesso di lanciare richieste e di rimanere in attesa di risposte che arrivano con tempi che, paragonati ai tempi del sistema operativo, sembrano ere geologiche. Grazie al comportamento asincrono, durante le attese di una certa azione il runtime engine può gestire qualcos’altro che ha a che fare con la logica applicativa, ad esempio.

ESERCITAZIONE

Verrà sviluppata una chat multi-utente a cui collegarsi tramite il browser. In particolare sono stati sviluppati i seguenti punti:

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

E' necessario aprire una shell ed eseguire i seguenti passi:

- preparazione dei sorgenti;
- installare nodejs;

Dentro la cartella con i sorgenti, troviamo già installati nella cartella node_modules i vari moduli necessari all'avvio dell'esercitazione.

Da terminale avviamo quindi con 'nodejs [server.js](#)' per eseguire il server, e apriamo una pagina in '<http://localhost:4000>'.

La spiegazione più dettagliata del codice è lasciata come commento all'interno del codice stesso.

Esercizio 0:

Effettuare una cattura Wireshark relativa al funzionamento della chat. Occorre attivare la cattura prima di aprire la URL dalla finestra del browser e ascoltare non solo sull'interfaccia di loopback ma anche su quella verso l'esterno (opzione ANY).

RISPOSTA:

La cattura mostra chiaramente il processo di "handshake" tra client e server, che è il momento in cui la connessione HTTP iniziale viene "aggiornata" a una connessione WebSocket. Questo è cruciale perché, da quel momento in poi, la comunicazione diventa bidirezionale e persistente, permettendo al server di inviare autonomamente aggiornamenti al client, cosa non possibile con il protocollo HTTP tradizionale. I WebSocket utilizzano porte HTTP standard (80 e 443), rendendoli compatibili con le impostazioni di sicurezza di firewall e proxy esistenti.

Quindi, selezionando come rete 'any' e avviata una cattura su WSH, ho eseguito [server.js](#) e client su localhost:4000, loggato con nome e mandato un messaggio di prova.

I PDU e i vari pacchetti sono veramente tanti quindi ho filtrato (si può benissimo fare anche prima della cattura stessa) con la query

‘http || websocket’

in modo da ricevere solo i pacchetti che interessano questo esercizio. Inoltre ho colorato i pacchetti di websocket di arancione in modo da differenziarli molto rispetto a quelli di semplice HTTP (visualizza ---> regole di colorazione). Il risultato è:

http websocket					
	Source	Destination	Protocol	Length	Info
86	::1	::1	HTTP	689	GET /socket.io/?EIO=3&transport=polling&t=PSrAIHl HTTP/1.1
88	::1	::1	HTTP	460	HTTP/1.1 200 OK (text/plain)
93	::1	::1	HTTP	714	GET /socket.io/?EIO=3&transport=polling&t=PSrAIIk&sid=eDmMwW0v5BC2NbxCAAAAA HTTP/1.1
95	::1	::1	HTTP	690	GET /socket.io/?EIO=3&transport=websocket&sid=eDmMwW0v5BC2NbxCAAAAA HTTP/1.1
97	::1	::1	HTTP	217	HTTP/1.1 101 Switching Protocols
99	::1	::1	WebSoc...	100	WebSocket Text [FIN] [MASKED]
100	::1	::1	WebSoc...	96	WebSocket Text [FIN]
104	::1	::1	HTTP	358	HTTP/1.1 200 OK (text/plain)
105	::1	::1	WebSoc...	95	WebSocket Text [FIN] [MASKED]
156	::1	::1	HTTP	880	GET / HTTP/1.1
161	::1	::1	HTTP	353	HTTP/1.1 304 Not Modified
163	::1	::1	WebSoc...	96	WebSocket Connection Close [FIN] [MASKED]
166	::1	::1	WebSoc...	92	WebSocket Connection Close [FIN]
177	::1	::1	HTTP	759	GET /styles.css HTTP/1.1
182	::1	::1	HTTP	353	HTTP/1.1 304 Not Modified
186	::1	::1	HTTP	742	GET /chat.js HTTP/1.1
187	::1	::1	HTTP	353	HTTP/1.1 304 Not Modified
228	::1	::1	HTTP	689	GET /socket.io/?EIO=3&transport=polling&t=PSrAKDn HTTP/1.1
229	::1	::1	HTTP	460	HTTP/1.1 200 OK (text/plain)
234	::1	::1	HTTP	690	GET /socket.io/?EIO=3&transport=websocket&sid=ZXQUzZf31KDKfcXAAAB HTTP/1.1
236	::1	::1	HTTP	714	GET /socket.io/?EIO=3&transport=polling&t=PSrAKEK&sid=ZXQUzZf31KDKfcXAAAB HTTP/1.1
237	::1	::1	HTTP	217	HTTP/1.1 101 Switching Protocols
239	::1	::1	WebSoc...	100	WebSocket Text [FIN] [MASKED]
240	::1	::1	WebSoc...	96	WebSocket Text [FIN]
243	::1	::1	HTTP	358	HTTP/1.1 200 OK (text/plain)
244	::1	::1	WebSoc...	95	WebSocket Text [FIN] [MASKED]
410	::1	::1	WebSoc...	171	WebSocket Text [FIN] [MASKED]
412	::1	::1	WebSoc...	170	WebSocket Text [FIN]

Analizziamo dunque i pacchetti uno per uno o per gruppi semanticci:

- Pacchetto 86: questa è la richiesta HTTP iniziale fatta dal client a localhost:4000. Questa non è la GET per visualizzare la pagina della chat (quella sarebbe una GET/). Invece, questa è una richiesta specifica di Socket.IO per iniziare la sua connessione utilizzando il trasporto "polling". Socket.IO usa il polling come meccanismo di fallback o come primo passo per stabilire e negoziare la connessione e ottenere un ID di sessione prima di tentare l'upgrade a WebSocket;
- Pacchetto 88: il server risponde al client (browser) fornendo un ID di sessione e altre informazioni;
- Pacchetto 93: il client prosegue con lo scambio di informazioni di socket;
- Pacchetto 95: dopo aver stabilito una sessione con il polling, il client invia una nuova richiesta HTTP GET. Questa richiesta è speciale perché include gli header Connection: Upgrade e Upgrade: websocket. Questo è il segnale esplicito del client

al server per chiedere di "aggiornare" la connessione HTTP esistente (o una nuova connessione TCP per questa richiesta, a seconda del comportamento del browser/Socket.IO) al protocollo WebSocket;

- Pacchetto 97: il codice di stato 101 Switching Protocols indica che il server ha accettato la richiesta del client di passare dal protocollo HTTP al protocollo WebSocket. L'header Sec-WebSocket-Accept è la conferma di sicurezza. Da questo punto in poi, la connessione TCP sottostante non trasporterà più richieste/risposte HTTP standard, ma diventerà un canale per i frame di dati WebSocket;
- Pacchetto 99: i frame WebSocket con [MASKED] sono inviati dal client al server. Il payload di Text: 2, indica un "ping" o un messaggio di controllo/inizializzazione di Socket.IO per verificare che la connessione sia attiva (payload = 2probe);
- Pacchetto 100: questo è un frame di controllo/risposta WebSocket inviato dal server al client, in risposta al pacchetto 99. I frame dal server al client li riconosciamo perché non sono mascherati. Payload = 3probe;
- Pacchetto 104: risposta HTTP 200 OK al pacchetto di polling 93. Il contenuto 1:6 indica che la connessione WebSocket è stata stabilita con successo (il 1 è il codice per "open" da Socket.IO e 6 è il codice per "close" per il trasporto HTTP polling). Questo chiude la connessione di polling HTTP in quanto la WebSocket è ora operativa;
- Pacchetto 105: messaggio WebSocket dal client al server. Il payload è 5. Questo è il primo "ping" heartbeat che il client invia sulla connessione WebSocket per mantenerla viva. Questi ping vengono inviati periodicamente per evitare che la connessione venga chiusa per inattività da firewall o router intermedi;
- SUCCESSIVI: tutti i pacchetti intermedi fino al 410 sono ininfluenti per l'esercizio. Probabilmente da altre connessioni, oppure da una chiusura errata della connessione dovuta a problemi di rete (sono in biblioteca con rete pubblica), il server ha chiuso la connessione, per poi riapirla completamente da capo;
- Pacchetto 410: questo è un messaggio dell'applicazione inviato dal client (Martin) al server. Il prefisso 42 indica un messaggio di tipo "event" in Socket.IO. Il primo elemento dell'array JSON, "message", è il nome dell'evento. Il secondo elemento è un oggetto JSON che contiene i dati associati all'evento, ossia: `{"message": "Ciao, sono entrato con successo", "sender": "Martin"}`. In questo caso, il client Martin sta inviando un messaggio di chat al server, indicando che è entrato con successo;
- Pacchetto 412: questo è un messaggio dell'applicazione inviato dal server al client. Anche qui, il prefisso 42 indica un messaggio di tipo "event", cui nome è "UploadChat". Secondo quanto raccontato nei lucidi, qui il server dovrebbe spedire lo

stesso messaggio ricevuto dal client Martin nel pacchetto precedente, a tutti gli utenti collegati alla chat;

N.B: WSH non riesce a distinguere WebSocket da HTTP in quanto usa le stesse porte, per cui non vi è alcuna differenza per lui da questo punto di vista. Invece, riesce comunque a interpretare ciò perché “ha letto” in uno dei pacchetti TCP precedenti al passaggio tramite la flag ‘PROTOCOL UPDATE’ sull’HTTP.

Esercizio 1:

Modificare a piacimento il contenuto del file public/index.html e valutarne l’impatto grafico.

RISPOSTA:

Fatto. Ho modificato un po’ i colori della pagina per renderla milanista, cambiando il titolo di chat-window e aggiungo il logo del Milan all’interno del riquadro della chat. Ora è a tutto e per tutto una chat di un gruppo milanista.

Esercizio 2:

Modificare il sorgente del codice in modo da far ascoltare il server sulla porta 80 invece che 4000. Quali altri accorgimenti sono necessari per farlo funzionare: lato server? Lato client?

RISPOSTA:

Lato server, è bastato modificare la porta indicata per la creazione della socket di connessione a riga 22-23, specificando 80 anzichè 4000. Lato client invece, aprendo localhost, è bastato modificare la porta da 4000 a 80 e la connessione è avvenuta con successo. Chiaramente un SO come Linux non lascia all’utente i permessi necessari ad occupare una delle Well-known-Port come la 80, pertanto si rende necessario aggiungere il comando ‘sudo’ durante l’esecuzione del server (‘*sudo nodejs [server.js](#)*’). In realtà ho successivamente ri-modificato da 80 a 443 in modo che eventuali hacker principianti non decidesse di hackerarmi vedendo l’insicura porta 80; chiaramente con 443 non parte da solo HTTPS, però a prima vista magari stento qualche malintenzionato essendo che sono in rete pubblica al momento.

Esercizio 3:

Modificare il sorgente del codice in modo da cambiare nome ai seguenti eventi:

1. message → messaggio;
2. UploadChat → aggiornamento.

RISPOSTA:

E' bastato modificare i nomi degli eventi cui la websocket rimane in attesa, in fondo a [server.js](#), e il nome degli eventi per cui è stato aggiunto un listener in [chat.js](#).

Esercizio 4:

Se si dispone di due PC in grado di dialogare sulla stessa rete IP (oppure un PC per il server e uno smartphone per il client sempre in grado di dialogare tra loro a livello IP) provare ad accedere al server che è su Linux mediante diversi tipi di browser e di sistemi operativi. Cosa si può notare?

RISPOSTA:

Accedendo alla chat dal server Linux con vari browser (Chrome, Firefox, Safari) e sistemi operativi (Windows, iOS), la cosa più evidente è che si dimostra completamente fluido e compatibile. Indipendentemente dalla piattaforma, la chat funziona identicamente: tutti i client si connettono, inviano e ricevono messaggi in tempo reale. Questa adattabilità è garantita dalla standardizzazione del protocollo WebSocket e dal ruolo di Socket.IO. I browser moderni implementano lo standard WebSocket, permettendo una connessione bidirezionale stabile. Socket.IO, inoltre, assicura compatibilità estesa con fallback automatici (come il long-polling HTTP) se WebSocket non fosse pienamente supportato o bloccato. Dato che la comunicazione inizia tramite un handshake HTTP su porte standard, aggira spesso ostacoli come firewall. In pratica, il server Node.js non si preoccupa del client specifico, finché questo parla il protocollo WebSocket/Socket.IO, cosa che fanno tutti i sistemi moderni.

Esercizio 5:

Modificare il sorgente del codice per fare in modo che ad ogni utente connesso alla chat arrivi nella console il messaggio "l'utente sta scrivendo..." . NOTA: Lato client, bisogna spedire al server un evento apposito (ad es. "typing") quando l'utente scrive sulla tastiera (catturando l'evento di sistema "keydown"). 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 occorre infine gestire la ricezione del messaggio "typing" che arriva dal server (vedere gestione del messaggio "UploadChat").

RISPOSTA:

E' bastato aggiungere un nuovo div di notifica in cui inserire il messaggio, catturare nel client l'evento keydown e spedire al server una notifica. Il server, ricevuta la notifica, inoltra

a tutti i restanti client escluso il mittente un messaggio di evento chiamato ‘typing’ (tramite funzione broadcast.emit). Il client a sua volta gestisce questo evento inserendo nel div di notifica la scritta ‘<nome_utente> sta scrivendo’, dove una flag regola il fatto che venga scritto una ed una sola volta questa frase, e nome_utente è un dato passato dal server, a sua volta ricevuto dal client che lo ha notificato. Il campo div notifica viene coerentemente pulito ad un qualsiasi aggiornamento (ossia alla ricezione di un messaggio o un 3o client che si mette a scrivere).

Esercizio 6:

Pensando all'esercizio sulla chat dell'esercitazione sulla programmazione mediante socket, che differenze/vantaggi/svantaggi si hanno con le tecnologie impiegate in questa esercitazione?

RISPOSTA:

Per una chat web, usare WebSockets e Socket.IO è infinitamente più pratico e conveniente dei socket TCP/IP che abbiamo usato durante le esercitazioni precedenti. I WebSockets nascono per il browser, gestendo l'handshake e la connessione bidirezionale in modo standard, e superando facilmente firewall e proxy. Socket.IO poi semplifica tutto, gestendo riconnessioni, eventi e persino i fallback se WebSockets non è disponibile. Con i socket normali invece, ci siamo dovuti inventare da zero (chiaramente usufruendo della libreria network.c/h) tutti questi meccanismi (come riconoscere i messaggi, gestire le disconnessioni, etc.) e, soprattutto, non potevamo connetterti direttamente da un browser, rendendo l'approccio impraticabile per una chat web moderna. In breve, WebSockets/Socket.IO sono la scelta naturale per il web in tempo reale, mentre i socket (con C in particolare) sono per scenari di rete di basso livello molto specifici, non per le chat web.

Un ulteriore vantaggio che ho notato durante la stesura dell'esercizio 5 è che nonostante il server chiuda, se non vengono chiusi anche i client, riaprendo il server i client si ricollegano subito e da soli, senza necessità di refreshare la pagina. Probabilmente questo meccanismo è dovuto ai cookies che ho potuto notare nei vari pacchetti HTTP/WebSocket, utili a mantenere dati sulla connessione e i vari utenti collegati.

PARADIGMA PUBLISHER/SUBSCRIBER

Il modello Publisher/Subscriber, o Pub/Sub, è un paradigma di messaggistica asincrona e disaccoppiata in cui i mittenti di messaggi (i "publisher") non inviano direttamente i messaggi a specifici destinatari (i "subscriber"), ma piuttosto a categorie di messaggi, chiamate "topic" o "canali". I destinatari che sono interessati a certi tipi di messaggi si "iscrivono" (subscribing) a questi topic, e ricevono automaticamente tutti i messaggi pubblicati su di essi. Un'applicazione client/server si può trasformare sempre in un'applicazione pub/sub dove i client e server originari diventano più leggeri e la complessità viene scaricata sull'host che fa da broker. I componenti chiave del modello Pub/Sub sono:

1. Publisher (editore):
 - è l'entità che crea e invia messaggi;
 - non ha conoscenza dei subscriber; non sa quanti ce ne sono, né chi sono.
 - si limita a "pubblicare" un messaggio su uno specifico topic o canale.
2. Subscriber (iscritto):
 - è l'entità che riceve i messaggi;
 - dichiara il proprio interesse per uno o più topic specifici (si "iscrive");
 - non ha conoscenza dei publisher; non sa chi sta inviando i messaggi;
 - riceve tutti i messaggi pubblicati sui topic a cui è iscritto.
3. Topic/Channel (argomento/canale):
 - è una categoria o un "nome" astratto al quale i publisher inviano i messaggi e i subscriber si iscrivono;
 - rappresenta il "mezzo" attraverso cui i messaggi vengono filtrati e distribuiti.
4. Message Broker/Event Bus (broker di messaggi/bus di eventi):
 - questo è il componente fondamentale che rende possibile il disaccoppiamento;
 - è un intermediario che riceve i messaggi dai publisher e li inoltra ai subscriber appropriati;
 - è responsabile della gestione dei topic, delle iscrizioni e della distribuzione efficiente dei messaggi.

Come Funziona, a livello di flusso di lavoro:

1. un Subscriber si connette al Message Broker e si iscrive a un Topic X;
2. un Publisher si connette allo stesso Message Broker e pubblica un messaggio sul Topic X;
3. il Message Broker riceve il messaggio dal Publisher;
4. il Message Broker identifica tutti i Subscriber iscritti al Topic X;
5. il Message Broker inoltra il messaggio a tutti i Subscriber interessati.

Vantaggi:

1. Disaccoppiamento (Decoupling):
 - temporale: Publisher e Subscriber non devono essere attivi contemporaneamente. Il broker può conservare i messaggi (o distribuirli appena un subscriber torna online, a seconda dell'implementazione);
 - spaziale: Publisher e Subscriber non devono conoscersi reciprocamente o sapere dove si trovano. Il broker fa da intermediario;
 - tecnologico: possono essere implementati in linguaggi e su piattaforme diverse, purché comunichino con lo stesso broker. Questo disaccoppiamento rende il sistema più robusto, flessibile e facile da scalare e mantenere.
2. Scalabilità:
 - è facile aggiungere nuovi Publisher o Subscriber senza modificare quelli esistenti;
 - se un topic ha molti subscriber, il broker gestisce la distribuzione a tutti loro;
 - se un publisher invia molti messaggi, il broker può gestirli e metterli in coda;
3. Affidabilità: il broker può garantire la consegna dei messaggi (a seconda della configurazione, es. "almeno una volta", "esattamente una volta");
4. Flessibilità: permette facilmente di implementare notifiche one-to-many (un publisher, molti subscriber) o molti-a-molti (molti publisher, molti subscriber). Di fatto, va ricordata come una delle differenze più prepotenti rispetto al modello client-server, che invece lavora necessariamente con un sistema uno-a-molti (un server, n client).

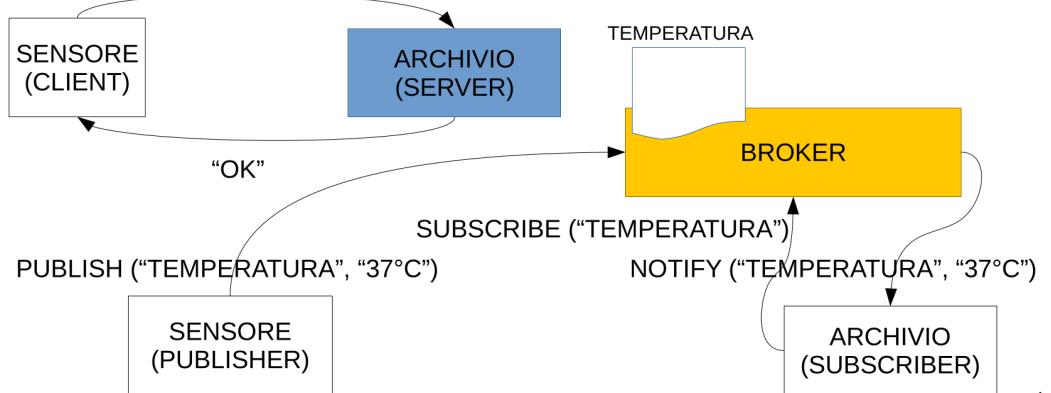
Svantaggi:

1. Complessità Aggiuntiva: l'introduzione di un Message Broker aggiunge un componente in più all'architettura, aumentando la complessità operativa (installazione, configurazione, monitoraggio del broker stesso);
2. Debugging: tracciare il flusso di un messaggio può essere più difficile a causa del disaccoppiamento e della natura asincrona. I messaggi non vanno da A a B direttamente, ma passano per un intermediario;
3. Latenza: in alcuni casi, l'introduzione del broker può aggiungere una piccola latenza rispetto alla comunicazione diretta (anche se broker moderni sono estremamente veloci).

Esempio:

Un esempio reale può essere un feed di notizie: un giornale (Publisher) pubblica articoli su vari topic (Sport, Politica, Tecnologia). Gli utenti (Subscriber) si iscrivono ai topic che li interessano e ricevono solo le notizie pertinenti.

Esempio di trasformazione (una domanda esame potrebbe essere di convertire un'applicazione client-server in pub/sub, quindi studiare e seguire questo modello di esempio):



4

Con il riquadro blu vediamo il modello classico client-server, mentre con il riquadro giallo possiamo vedere il modello pub/sub.

Alcuni esempi di protocolli (elenco puntato nero) e implementazioni specifiche dello stesso (elenco puntato bianco) sono:

- Message Queuing Telemetry Transport (MQTT):
 - Mosquitto (C/C++);
 - Paho (Python).
- Advanced Message Queuing Protocol (AMQP):
 - RabbitMQ;
- Apache Kafka: piattaforma distribuita di gestione di stream di eventi.

MQTT:

MQTT è un protocollo di messaggistica leggero, progettato per la comunicazione efficiente tra dispositivi, specialmente quelli con poche risorse o su reti inaffidabili, come nell'Internet of Things. Funziona sul modello Publisher/Subscriber come descritto. Ciò che lo rende potente sono funzionalità come i diversi livelli di qualità del servizio (QoS) per garantire la consegna, i messaggi "retained" per mostrare l'ultimo stato e il "Last Will and Testament" per notificare disconnessioni improvvise. È l'implementazione ideale del Pub/Sub per il mondo dell'IoT.

N.B: la parte vulnerabile è il broker!!!

ESERCITAZIONE

Esempio di partenza:

Apriamo due terminali nella cartella che contiene il file .perm per avere l'autorizzazione per collegarsi al servizio offerto dall'università. In uno digitiamo

```
'mosquitto_sub -t temperatura -h 90.147.167.187 -p 8883 -u univr-studenti -P  
MQTT-esercitazione2024 --cafile ISRG_Root_X1.pem --insecure -d'
```

mentre nell'altra

```
'mosquitto_pub -t temperatura -m "25" -h 90.147.167.187 -p 8883 -u univr-studenti -P  
MQTT-esercitazione2024 --cafile ISRG_Root_X1.pem --insecure -d'.
```

Il primo è il subscriber, che ritorna il seguente output:

- Client null sending CONNECT
- Client null received CONNACK (0)
- Client null sending SUBSCRIBE (Mid: 1, Topic: temperatura, QoS: 0, Options: 0x00)
- Client null received SUBACK
- Subscribed (mid: 1): 0
- Client null received PUBLISH (d0, q0, r1, m0, 'temperatura', ... (4 bytes))
- halo

il terminale rimane in esecuzione in attesa di pubblicazioni da parte del publisher sul topic desiderato. In tutto questo, null è l'id del subscriber (in questo caso non ne ha uno, per questo null). Con Connect si connette, con connack si conferma la connessione (ACK, 0). Nella terza riga ci si iscrive al topic 'temperatura', con id iscrizione (Mid:1) e QoS a 0, ossia quality of service di livello 0 (non è richiesta la conferma di ricezione da parte del broker). Con SubACK il broker risponde ACK alla richiesta di sottoscrizione. Segue poi una notifica di pubblicazione da parte del broker, che avverte il subscriber che è stato pubblicato qualcosa nel topic; d0 indica che il messaggio non è un duplicato, q0 che viene inviato con metodo QoS impostato a 0, e r1 indica (IMPORTANTE) che il messaggio NON rappresenta una nuova pubblicazione, ma una precedente al collegamento del subscriber e che è stato programmato per essere pubblicato all'avvio. In particolare, si tratta di un messaggio di peso 4 bytes, opportunamente stampato nella riga seguente ('halo'). Alla terminazione forzata del processo (kill -9 o ctrl^C), avviene la disconnessione con relativo messaggio sul terminale ('Client null sending DISCONNECT').

Per il publisher invece abbiamo quanto segue:

- Client null sending CONNECT

- Client null received CONNACK (0)
- Client null sending PUBLISH (d0, q0, r0, m1, 'temperatura', ... (8 bytes))
- Client null sending DISCONNECT

Il processo a terminale viene terminato una volta pubblicato il dato richiesto. Nel terminale del subscriber riceveremo il dato “64”, mentre qui, oltre a CONNECT E CONNACK, abbiamo l’effettiva pubblicazione del dato (da notare flag r0 dato che si tratta di una nuova pubblicazione), e la successiva disconnessione del publisher.

N.B: vedere in automatico nel terminale del subscriber righe come ‘Client null sending PINGREQ’ oppure ‘Client null received PINGRESP’ è del tutto normale. Il primo è un messaggio dal subscriber verso il broker per fargli sapere che è ancora attivo e connesso, mentre il secondo è la relativa risposta del broker, che gli fa sapere che anche lui è ancora bello arzillo.

Esercizio 1:

Partendo dall’esercizio mostrato nelle istruzioni, cosa succede se pubblico una temperatura prima di aver lanciato il subscriber? Provare con l’opzione --retain

RISPOSTA:

Se avviamo prima il publisher e poi il subscriber, come detto prima il publisher chiude subito dopo la pubblicazione degli argomenti. Il subscriber quindi, una volta iscritto, non vedrà la nuova pubblicazione perché antecedente la sua stessa iscrizione.

Aggiungendo la flag --retain al publisher e sempre eseguendolo per primo, il messaggio di pubblicazione di “benvenuto” che ho descritto durante l’esempio di prova viene rimpiazzato da questa effettiva pubblicazione.

Esercizio 2:

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?

RISPOSTA:

Per poter fare questa cosa, ci basta creare due diversi publisher (quindi un terminale a processo), e modificare il topic in cui pubblicano i dati. In uno specifichiamo il topic ‘temperatura/sensore1’, mentre nell’altro ‘temperatura/sensore2’. In questo modo siamo sempre sotto lo stesso macro-topic, ossia la temperatura, ma differenziamo le pubblicazioni

riguardanti due sensori diversi. Sempre mantenendo un senso logico quindi, è possibile modularizzare a piacimento le pubblicazioni e le sottoscrizioni dei vari client.

Quindi, per avere due topic diversi servono chiaramente due publisher, e quindi 2 terminali. Poi sulla base di quanti subscriber vogliamo avere, aggiungiamo altre istanze del terminale. Nel mio caso ho fatto 2 subscriber, uno per publisher, quindi altri 2 terminali (4 in totale).

Esercizio 3:

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?

RISPOSTA:

Risposta banale. Invece di specificare il percorso esatto che porta al topic desiderato come nell'esercizio precedente, andiamo a usare un pattern di percorso che ci permette di indicare la sottoscrizione all'intero macro-topic. Quindi, nel subscriber, invece di scrivere 'temperatura/sensore<N-esimo>' andiamo a scrivere 'temperatura/#'.

In questo specifico caso di esecuzione ci vengono quindi forniti in output tutti i caricamenti da parte di publisher all'interno del macro-topic 'temperatura'. Questo chiaramente comprende anche topic che non ho creato io, come 'casa', 'cucina' o altro ancora. Se avessimo voluto creare una sessione personalizzata al 100%, ossia con topic creati solo da me, avrei dovuto creare un ulteriore sottocartella in cui inserire i sensori, e usare il pattern # al suo interno.

Come spiegato nell'esempio all'inizio, prima di stampare il contenuto del messaggio del publisher, il broker ci manda anche delle informazioni riguardanti la pubblicazione stessa. Oltre alla natura del messaggio (r0 nuova pub, r1 pub precedente), possiamo vedere in chiaro per quale sotto-topic di temperatura è stata fatta la pubblicazione.

Esercizio 4:

Prova a pubblicare un valore di umidità relativa (topic "UR"); il subscriber interessato alle temperature 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.

RISPOSTA:

Chiaramente il sub iscritto alla temperatura non riceve notifiche riguardanti l'umidità, così come vale il contrario. Per invece avere un sub in grado di ricevere entrambe, ci basta usare la wizard +/UR e +/temperatura, che permette l'iscrizione a due sotto-topic appartenenti a due macro topic diversi. Abbiamo quindi due publisher, uno che pubblica un dato in /UR, e l'altro in /temperatura (molto importanti gli '/'). Con il subscriber invece, andiamo a iscriverci a entrambi con (appunto specificando -t +/UR seguito da -t +/temperatura)

```
mosquitto_sub -t +/temperatura -t +/UR -h 90.147.167.187 -p 8883 -u univr-studenti -P  
MQTT-esercitazione2024 --cafile ISRG_Root_X1.pem --insecure -d
```

Esercizio 5:

Aprire e studiare con Wireshark il file PCAP contenuto nello ZIP:

- quale protocollo di livello trasporto utilizza MQTT? Quali sono le porte?
- trovare le fasi di publish e subscribe. Quante connessioni TCP apre un subscriber? Quante connessioni TCP apre un publisher?

RISPOSTA:

A livello di trasporto, MQTT utilizza TCP, usufruendo in questo caso specifico delle porte 1883 e 60070.

Per un client MQTT (sia esso un subscriber o un publisher), la comunicazione con il broker avviene generalmente tramite una singola connessione TCP persistente. Questa connessione viene mantenuta attiva per tutta la durata della sessione, permettendo scambi bidirezionali efficienti. In WSH, il subscriber apre una sola connessione TCP con il broker per connettersi, iscriversi ai topic e ricevere tutti i messaggi. Il publisher invece apre una connessione TCP con il broker per ogni pubblicazione, dato che, come abbiamo visto negli esercizi precedenti, dopo la pubblicazione, chiude.

Esercizio 6:

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

RISPOSTA:

TOPIC DI PUBBLICAZIONE:

- Docenti:

```
mosquitto_pub -t Università/docenti -m "messaggioDaDocente" -h 90.147.167.187 -p 8883  
-u univr-studenti -P MQTT-esercitazione2024 --cafile ISRG_Root_X1.pem --insecure -d
```

- Studenti:

```
mosquitto_pub -t Università/studenti -m "messaggioDaStudente" -h 90.147.167.187 -p  
8883 -u univr-studenti -P MQTT-esercitazione2024 --cafile ISRG_Root_X1.pem --insecure  
-d
```

Assunto, poiché non richiesto, che segreteria e rettore solo leggano in questa applicazione.

TOPIC DI ISCRIZIONE:

- Rettore:

```
mosquitto_sub -t Università/# -h 90.147.167.187 -p 8883 -u univr-studenti -P  
MQTT-esercitazione2024 --cafile ISRG_Root_X1.pem --insecure -d
```

- Segreteria:

```
mosquitto_sub -t Università/docenti -t Università/studenti -h 90.147.167.187 -p 8883 -u  
univr-studenti -P MQTT-esercitazione2024 --cafile ISRG_Root_X1.pem --insecure -d
```

Per come è stata specificata l'applicazione, il rettore e la segreteria leggono gli stessi gruppi alla fine. Per come sono state scritte le sottoscrizioni tuttavia, nel caso venisse aggiunto un nuovo gruppo di pubblicazione, il rettore ne sarebbe iscritto automaticamente.

- Docenti:

```
mosquitto_sub -t Università/docenti -t Università/studenti -h 90.147.167.187 -p 8883 -u  
univr-studenti -P MQTT-esercitazione2024 --cafile ISRG_Root_X1.pem --insecure -d
```

- Studenti:

```
mosquitto_sub -t Università/studenti -h 90.147.167.187 -p 8883 -u univr-studenti -P  
MQTT-esercitazione2024 --cafile ISRG_Root_X1.pem --insecure -d
```

Volendo provare altro, basta usare i due pub aperti per provare questa parte (2 per pub, 4 per sub) e pubblicare in una cartella diversa da docente/studente per vedere che il rettore riceve comunque.