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).

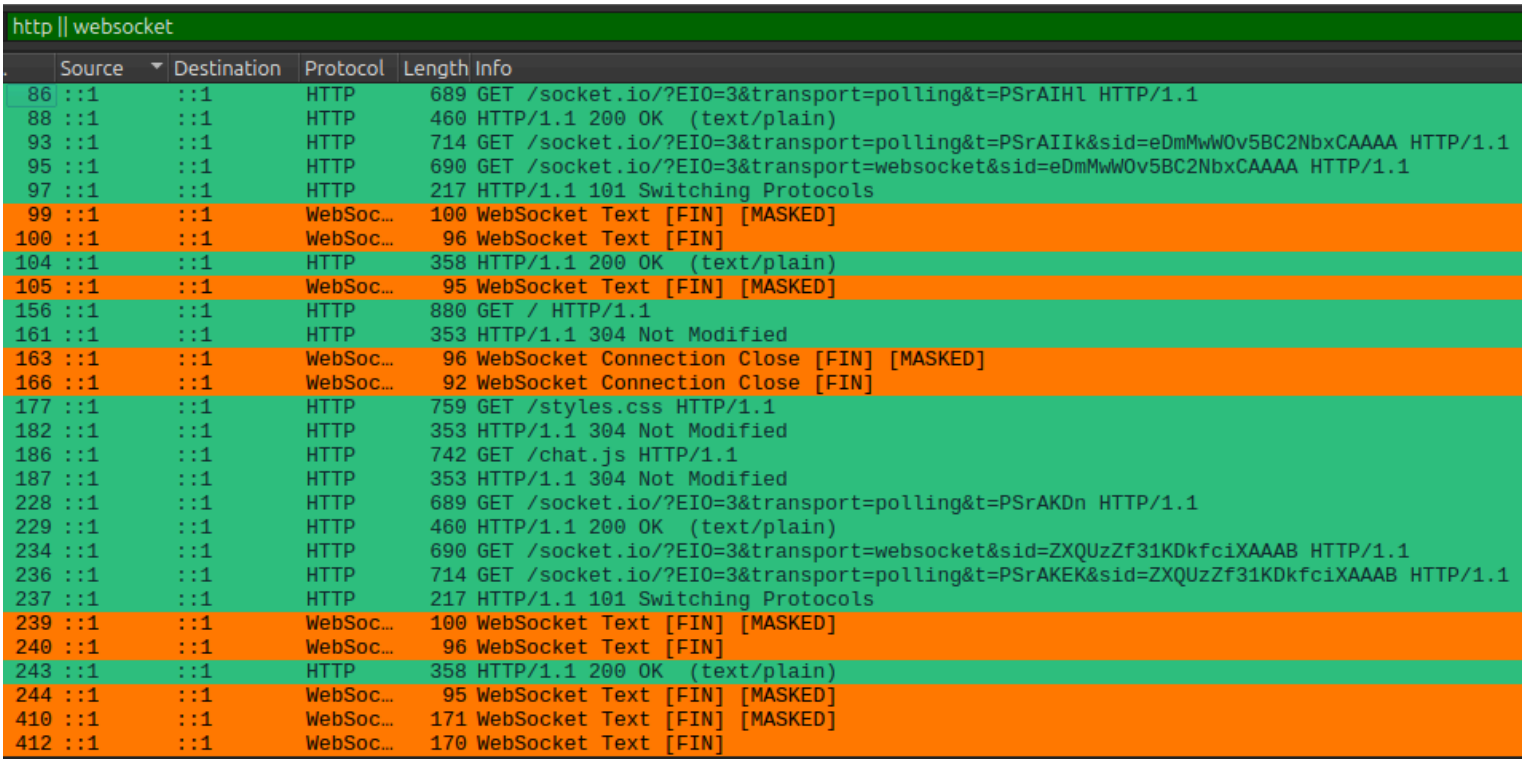
RISPOSTA:
La cattura mostra chiaramente il processo di "handshake" tra client e server, che è il momento in cui la connessione HTTP iniziale viene "aggiornata" a una connessione WebSocket. Questo è cruciale perché, da quel momento in poi, la comunicazione diventa bidirezionale e persistente, permettendo 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:



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

- 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 riaprirla 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, oltre che ai meccanismi di fallback di Socket.io.