ESERCITAZIONE:

Per gli esercizi, seguire le slide di consegna e aprire man mano i file richiesti. La spiegazione per ogni file e/o sorgente di codice si trova come commento dentro ai file stessi.

Per favorire una maggiore comprensibilità e un maggiore matching tra esercizio -> file, i codici sono stati enumerati in ordine con cui sono stati richiesti.

Per l'esercizio 9, più ampio e complesso, è stata creata proprio una sottocartella a parte e, per ogni consegna di esercizio, un relativo codice commentato e un file txt di breve spiegazione.

ESERCIZI da 1 a 9:
aprire i file html seguendo l'enumerazione e provare le varie funzionalità. Non c'è nulla da eseguire di particolare perchè è tutto in frontend classico. Serve ad esplorare le principali funzionalità di html, javascript e CSS. Questa sezione culmina con il file 'Esercizio1.html', dove veniva chiesto di creare un codice html che richiede periodicamente un refresh della pagina principale dell'ansa.

Cartella 9-serverHTTP: ESERCIZIO 1
utilizzando file serverHTTP.c e 9-form-get.html.

1) Avviare l'esercizio: compilare con 'gcc network.c serverHTTP.c -o EXEC' e a seguire ./EXEC per lanciare il server. Poi, aprendo un browser a proprio piacimento, lanciare il client con localhost:8000; in questo modo ci viene restituita la pagina base servita dal server. Se invece si vuole lanciare uno dei file html proposti, basta fare 'apri con' oppure doppio click sul file (NON APRIRE DA VSCode, in quanto apre il live server dell'applicazione sulla porta 5500, che ovviamente è diverso da quello che abbiamo lanciato noi);
2) Analizzare il codice:
    a) il codice contiene una stringa con un messaggio HTTP di risposta positiva, con all'interno il codice HTML di una pagina da visualizzare. Tramite la libreria network.c viene creata una connessione TCP sulla porta 8000 di tipo server che rimane in ascolto di un eventuale client (while(true)). Ne estrae il metodo (GET, POST, ecc.), legge eventuali URL, e nel caso

di un POST legge anche il corpo della richiesta. Questo nella prima riga della richiesta HTTP usando fgets. Estrae il metodo HTTP (solo il primo "token" separato da spazio) usando strtok. Legge le successive righe dell'header HTTP fino a trovare una riga vuota (\r). Durante la lettura degli header, cerca la riga Content-Length: per memorizzare la lunghezza del corpo della richiesta nel caso di un POST, stampandolo sul terminale. Dopo aver eseguito in terminale l'eseguibile, il server è aperto e in ascolto. Basta quindi simulare un client collegandosi al localhost sulla porta 8000 da qualsiasi browser. Il server risponde come detto in base al tipo di richiesta. Nel caso base di aprire localhost e basta, la richiesta è una semplice GET, pertanto viene visualizzata la pagina HTML statica visibile all'inizio del codice.

3) Aprire secondo terminale:
    a) come nell'esercitazione 2, il SO crea un binding tra PID processo e porta diciamo, associando quindi queste due entità. Provando ad aprire un secondo server in contemporanea, il SO vede subito che la porta è già utilizzata e rifiuta la richiesta. Da terminale si viene subito mandati alla riga di richiesta successiva della bash; questa parte di output probabilmente è gestita dalla libreria network.c;
4) Aprire 127.0.0.1 e localhost:
    a) sono la stessa cosa. Il SO interpreta localhost come un dominio noto, e vi associa l'IP 127.0.0.1; Di fatto nel terminale, quando compaiono tutte le informazioni della richiesta HTTP del client, viene mostrato 'localhost' in entrambi i casi;
5) Eseguire il server web serverHTTP.c e con il browser preferito aprire il file form-get.html. Cosa si vede?
    a) Una volta premuto submit sul terminale di esecuzione possiamo vedere i messaggi HTTP GET di richiesta e di risposta. Nella prima, compreso dei parametri nome e cognome inseriti sulla pagina html (con ? direttamente innestati nell’URL, quindi con un massimo di caratteri), nella seconda la relativa risposta (favicon.ico);

6) Provare ad analizzare il contenuto della connessione TCP con Wireshark. Cosa si vede?
    a) mettendo i filtri sull'ip di sorgente e di destinazione possiamo vedere i three-hand shake tcp, e i vari messaggi HTTP (in particolare quello che vediamo anche sul terminale andando nella sezione 'follow TCP stream");

utilizzando serverHTTP.c e form-post.html

1. Eseguire il server web serverHTTP.c e con il browser preferito aprire il file form-post.html Cosa si vede? Provare ad analizzare il contenuto della connessione TCP con Wireshark. Cosa si vede?
    a. In sostanza, i dati non sono più passati come parametri dell'URL stesso, ma fanno ora parte del payload dell'invio dati POST del HTTP. Su WSH infatti sono visibili dentro al pacchetto;

Cartella 9-serverHTTP: ESERCIZIO 2

utilizzando serverHTTP2.c e form-file.html

1. Modificare il file serverHTTP.c in modo che, invece di restituire sempre la solita pagina web di prova, restituisca una delle pagine html usate dal lucido 9 in poi scelta attraverso l'uso del browser. Suggerimenti: Provare a fare la nuova richiesta col browser usando il serverHTTP.c in modo da vedere la richiesta HTTP per capire dove si trova la stringa con il nome del file nella richiesta che fa il browser (aiutarsi anche con Wireshark). Cosa devo scrivere nella barra del browser? Capire come recuperare la stringa con il nome del file dalla richiesta HTTP (si veda esempio-parser.c) Per costruire la risposta riciclare parte del codice usato nell'esercizio del trasferimento di file. Prova finale: cosa succede se chiedo al server di restituire i file form-get.html e form-post.html?
    a. Eseguire serverHTTP2.c con la stessa modalità di prima. Mentre il server è in esecuzione quindi, aprire localhost:8000/<nome_file> sul browser, indicando come <nome_file> il nome di un file html contenuto nella stessa cartella di esecuzione del server. Possiamo quindi provare con '9-form-get.html', '10-form-post.html' e volendo anche '11-form-file'. Provato anche con 'index.html' che contiene dei caratteri speciali per testare la codifica UTF-8. A

differenza del precedente server tuttavia, questo non ‘serve’ la richiesta successiva ai file html (ossia premendo submit nella pagina), ma serve solo a visualizzare il file richiesto.

Cartella 9-serverHTTP: ESERCIZIO 3

1. Modificare il server web serverHTTP.c in modo che accetti con il metodo POST un intero file da salvare nella cartella del server (il cosiddetto “upload sul server”). Suggerimenti: utilizzare il file form-file.html con serverHTTP.c non modificato ed analizzare il contenuto della connessione TCP con Wireshark in modo da capire nella richiesta HTTP dove si trova il nome del file e dove si trova il contenuto del file. Nella lettura della richiesta HTTP sul server aggiungere il codice che salva il file prendendo spunto dal codice usato nell’esercizio del trasferimento di file. Invece la risposta HTTP può essere molto statica come nella versione originale di serverHTTP.c.
    a. Per questo esercizio, lanciare l’eseguibile EXEC3 (o ricompilare con il sorgente serverHTTP3.c). Risponde come prima a qualsiasi html precedentemente usato, riportando però una nuova pagina html di risposta base per le richieste GET. Per le richieste post invece si analizza il contenuto e si cerca il file con l’argomento passato. Se si prova con ‘10-form-post’ la richiesta viene evasa scrivendo nel file di output (‘uploaded_server’) il contenuto della POST (teoricamente, nome -cognome). Se si usa invece ‘11-form-file’, abbiamo due casi. Inserendo un file txt o comunque in caratteri (esempio un file c), il contenuto viene correttamente inserito nel file di output a meno di un header aggiuntivo (che non ha senso togliere per via del caso 2). Inserendo invece un file pdf, il contenuto binario viene comunque correttamente copiato, ma senza l’header precedentemente detto aprirlo non è possibile. Per mantenere il file apribile sia in versione testuale che binaria, si lascia il file senza estensione, in modo da poterlo aprire in entrambi i casi (per il pdf, non si apre da vsc ma da archivio normale). NB: il file può non essere nella stessa cartella dell’eseguibile.

Cartella 9-serverHTTP: ESERCIZIO 4

1. Aprire il file serverHTTP-CGI.c in Esempi-web/ e analizzarne il contenuto. Compilarlo come al solito ed eseguirlo. Aprire il browser preferito e il file sommatrice-web.html. Cosa si vede sul browser? NOTA: Provare con numeri positivi, negativi, con parte decimale. A cosa corrisponde il secondo parametro della funzione sommatrice()?
    a. avviare EXEC4 (o ricompilare con serverHTTP-CGI.c) e aprire file '12-sommatrice.html'. Funziona anche con numeri negativi e decimali, mettendo ovviamente il punto e non la virgola. Attenzione che ho notato dei bug di reindirizzamento se prima di questo html se ne avviano altri che sono 'meno' gestiti da questo server. Il secondo parametro della funzione sommatrice è il file usato dal server per la connessione (connfd).

CARTELLA WEBSERVICE:

N.B: anche qui i file degli esercizi sono enumerati. Quelli che NON lo sono, o sono i server (serverHTTP…) oppure sono dei file secondari non richiesti espressamente negli esercizi.

1. Aprire il file serverHTTP-REST.c e analizzarne il contenuto. Compilarlo come al solito ed eseguirlo. Aprire il file clientREST-GET.c e analizzarne il contenuto. Compilarlo come al solito ed eseguirlo. Che parametri devo passare in linea di comando? Cosa si può vedere analizzando lo scambio di dati tramite Wireshark? Qual è la signature della funzione calcolaSomma() sul server e sul client? Perché ha senso che siano uguali?
    a. i parametri sono nome funzione, op1 e op2;
    b. WSH mette a lato sinistro una piccola barra nera che mostra dove inizia e finisce una determinata connessione, dato che intorno è pieno di pacchetti DNS e ICMP che non riguardano strettamente la connesssione. Poi vediamo il three-hand shake. Continua con il client che invia una richiesta GET al server, dove vediamo chiaramente lo Stub dei dati su cui fare il calcolo. Il server conferma di aver ricevuto con un ACK, e poi risponde con un JSON contenente la somma dei due numeri. Questo avviene in due PDU diversi (TCP segment of a reassemble PDU). Poi vediamo il riassemblamento dei PDU nella

risposta HTTP 200 OK, e infine la chiusura della connessione TCP con la flag FYN;

c. sia sul client che sul server, la signature della funzione calcolaSomma() è float calcolaSomma(float val1, float val2). Devono essere uguali perché definiscono il contratto dell'API, permettendo al client di invocare la funzione remota come se fosse locale.

2. Prendere in considerazione il file ClientREST.java. Dopo aver installato l'ambiente base di Java (già presente in Lab Delta) si può compilare con javac ClientREST.java ed eseguire con java ClientREST. Il fatto che il server sia fatto in C e il client in Java è un problema o un'opportunità? Perché?
   a. Il Client fa la stessa cosa di quello precedente, ma l'esercizio è volto a enfatizzare il fatto che anche se scritto in un linguaggio diverso dal server, funziona ugualmente. Il server implementa una semplice versione della SOA, ossia fornisce un servizio che prescinde dal linguaggio di programmazione del client, dato che il tipo di risposta (JSON) è standard e può essere interpretato da qualsiasi linguaggio di programmazione.

N.B: nel file ClientREST.java è già presente l'implementazione dell'esercizio che segue (3) sui numeri primi. Ai fini del mero esercizio 2, non valutare quella parte. Le funzionalità attese dall'esercizio 2 sono comunque preservate.

3. Estendere il webservice serverHTTP-REST.c in modo che esponga un secondo servizio relativo al calcolo dei numeri primi compresi nell'intervallo [min, max]. Si tragga spunto al programma prime-number-interval.c e estendere il file ClientREST.java in modo da poter chiamare, a scelta, entrambe le funzionalità della nuova API. Provare il client con il calcolo dei numeri primi compresi nell'intervallo [1, 1000000]. Quanto tempo ci mette? E' stato necessario tradurre l'algoritmo dei numeri primi in Java? Perché?
   a. l'esecuzione si avvia come nei precedenti esercizi, questa volta però eseguendo il serverHTTP-REST2.c (IGNORARE IL WARNING) e per il client java, specificando la funzione primenumber con i numeri d'intervallo richiesti dall'esercizio. Ci mette circa un minuto (dipende da altri fattori anche, nell'ultima mia esecuzione 80 secondi per esempio).

b. No, non è stato necessario tradurre l'algoritmo dei numeri primi in Java, perché è il server che implementa questo servizio. In java ho solo aggiunto la gestione del diverso tipo di risposta (stringa in questo caso).

4. Prendere in considerazione il file clientThreadREST.java. Esso invoca calcolaSomma() in 3 thread concorrenti. Però la macchina su cui gira il server chiamato è la stessa e quindi non ci guadagno in prestazioni. Come si potrebbe modificare il codice in modo che le invocazioni finiscano su 3 macchine diverse? Bisogna modificare anche il codice del server? Si riconsideri consideri il servizio che calcola i numeri primi nell'intervallo [min, max] costruito nell'esercizio precedente. Si trovi un modo efficiente, sfruttando diversi server in rete, per calcolare i numeri primi tra 1 e 1000000. Di quanto migliorano le prestazioni? Devo modificare anche il codice del server?
   a. per l'esecuzione prendere in considerazione il client ClientThreadREST.java, che si esegue nello stesso modo del precedente client, e il server finale, ossia serverHTTP-REST_FINALE.c (anche qui ignorare il warning). Una volta creato l'eseguibile, avviarlo su 3 terminali diversi specificando le porte 8000, 8001 e 8002. Poi avviare il client con le specifiche precedenti;
   b. per calcola-somma, come si vede nell'output, il risultato è la solita stampa JSON del risultato. Il calcolo non viene diviso come carico computazionale nei 3 thread, ma appunto passato così com'è a tutti i server. Per evitare ridondanza, viene oscurato il ritorno del server 2 e 3, così da avere il risultato solo una volta. In questo tipo di calcolo quindi, non abbiamo miglioramenti, ad indicare che abbiamo anche un minimo di complessità di elaborazione da dover soddisfare per poter sfruttare questo sistema di load balancing.
   c. per la suddivisione del lavoro per quanto riguarda il calcolo dei numeri primi, è bastato aggiungere al server la richiesta da riga di comando il numero di porta con cui aprire la connessione, e cambiare nel client il numero di porta delle 3 threads (che come detto, è statico). In questo modo, non avviene un'effettiva parallelizzazione del calcolo, ma solo delle richieste. Le threads sono solo il mezzo con cui parallelizziamo il calcolo, per cui stiamo solo istanziando 3/o più server diversi, che si frammentano quindi la complessità del calcolo.

d. il codice del server non va modificato, se non nella porta in cui accetta connessione. A livello di efficienza, il calcolo migliora di molto le prestazioni, dimezzando il tempo di esecuzione (da 80 sec a 40). Chiaramente volendo semplificare ancora il costo del calcolo, basterebbe istanziare ancora più server e suddividerne il calcolo di conseguenza.