Assignment #2: Relazione

Programmazione di reti

baiardi martina

lombardini alessandro

# **TASK 1**

## **Descrizione della FSM del Server**

Il server attende nello stato di *wait\_connection* una richiesta di connessione sulla porta designata da parte di un client. A seguito di una richiesta, dopo aver instaurato una connessione, il server passa allo stato di *wait\_hello\_message*, dove attende che venga inviato un hello message dal client.

Quando il client invia il messaggio e il server, a seguito della ricezione, controlla che sia valido:

1. Se l’esito risulta **positivo,** invia un messaggio di conferma al client e passa allo stato di *wait\_probe\_message*, dove attende i messaggi di probe da parte del client.
2. Se l’esito risulta **negativo**, invia un messaggio di errore al client e chiude la connessione. Lo stato del server ritorna quello iniziale di *wait\_connection*.

Nello stato *wait\_probe\_message*, all’arrivo di un nuovo messaggio, il server ne controlla la validità come nello stato precedente:

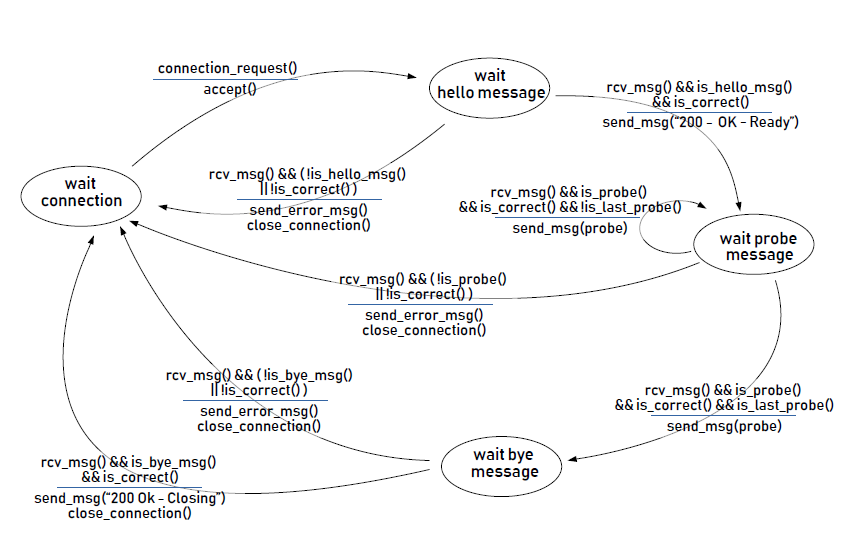
1. Se l’esito risulta **positivo,** invia indietro il messaggio ricevuto e attende di ricevere i messaggi di probe successivi;
2. Se l’esito risulta **negativo**, invia un messaggio di errore al client e chiude la connessione. Lo stato del server ritorna quello iniziale di *wait\_connection*.

Lo stato *wait\_probe\_message* non cambia finché il messaggio ricevuto, oltre ad essere valido, non è l’ultimo fra quelli previsti, ovvero non ne sono stati ancora inviati un numero pari alla quantità specificata nell’*hello message*. Una volta arrivati tutti i messaggi di probe validi, lo stato cambia in *wait\_bye\_message*, ovvero il server si mette in attesa del *bye message.* Quando il server riceve un messaggio, controlla che questo sia effettivamente il *bye message* atteso:

1. Se l’esito è **positivo**, invia un messaggio di conferma al client.
2. Se l’esito è **negativo**, invia al client un messaggio di errore.

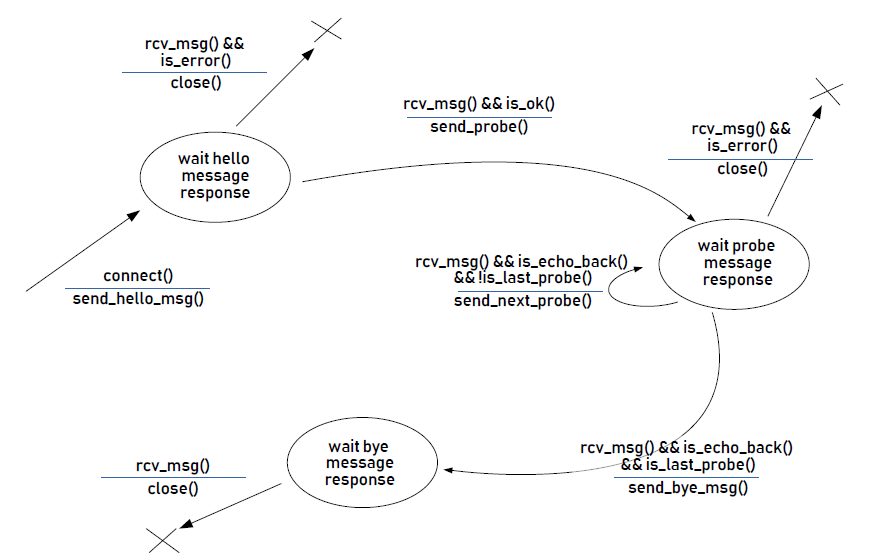
Una volta inviata la risposta, il server procede con la chiusura della connessione e torna in ascolto di nuove richieste nello stato di *wait\_connection*.

**NOTA: NELLA MACCHINA A STATI HA SENSO LASCIAR ALCUNI MESSAGGI IN CHIARO E ALTRI NO? Quando le stavamo facendo tempo fa mi avevi chiesto di mettere solo le stringhe di conferma e di evitare quelle di errore perché troppo lunghe, quindi la soluzione ibrida era questa. Dimmi come devo comportarmi e le modifico come preferisci.**



## **Descrizione della FSM del Client**

Il client instaura una connessione con il server e invia un *hello message*, posizionandosi nello stato di *wait\_hello\_message\_response*. Una risposta **negativa** implica un errore, dunque chiude la connessione e l’applicativo termina; una risposta **positiva** prevede lo spostamento nello stato di *wait\_probe\_message\_response* e il conseguente invio di un primo *probe message.* Se la risposta coincide con il messaggio inviato, il client può procedere con l’invio del probe successivo, altrimenti viene terminato l’applicativo. Non appena sono stati inviati un numero di *probe message* pari al valore indicato nell’*hello message*, e per ciascuno quali ricevuto un echo back valido da parte del server, il client può cambiare stato in *wait\_bye\_message response*, inviando un *bye\_message* al server. In caso arrivi un messaggio di conferma viene chiusa la connessione con successo; nel caso in cui il messaggio sia invece di errore, viene terminato il programma con un errore, chiudendo anche in questo caso la connessione.



# **TASK 2**

## **Client**

Il client si preoccupa di eseguire la richiesta per instaurare una connessione TCP con il server. Devono essere dunque forniti, all’atto dell’esecuzione, l’indirizzo IP e la porta su cui il server è in ascolto.

L’utente, per esprimere le proprie preferenze riguardo il servizio che vuole utilizzare, può modificare il file *init.conf.* Al suo interno devono essere presenti i dettagli riguardanti la misura da effettuare. Il formato della stringa contenuta al suo interno deve essere:

*<measure\_type><sp><n\_probes><sp><msg\_size><sp><server\_delay>*

*measure\_type:* il tipo di misura che su vuole effettuare; può essere ***thput*** o ***rtt***.

*n\_probes:* il numero di probe che desidera vengano inviati al server.

*msg\_size:* dimensione del payload dei messaggi di probe che vengono inviati al server. (in byte)

*server\_delay*: tempo di attesa che deve svolgere il server prima di rispondere ai probe del client (in millisecondi)

Il client legge queste informazioni e ne valuta la correttezza: se corrette, le estrae popolando una struttura dati interna, la quale viene tenuta aggiornata durante le diverse fasi d’esecuzione, se errate, il client non effettua richieste al server e termina.

La struttura dati è la seguente:

/\* Service structure definition \*/

typedef struct node {

char measureType[6]; /\* Service requested - RTT or THPUT \*/

int nProbes; /\* Number of probe messages to send \*/

int messageSize; /\* Size of probe message's payload \*/

int serverDelay; /\* Delay of server probe message echo (in milliseconds) \*/

int serverFD; /\* File descriptor of socket \*/

int phaseNumber; /\* 1: Hello phase

2: Probe phase

3: Bye phase \*/

} serviceNode;

serviceNode service;

Dove *phaseNumber* è una rappresentazione numerica delle fasi in cui si trova il client attualmente.

Se il contenuto del file è valido, il client esegue una richiesta di connessione al server e invia un *hello message,* entrando poi nello stato di attesa di una risposta da parte del server. Se tale risposta è positiva si prosegue nella creazione del messaggio di probe, altrimenti si chiude direttamente l’applicativo segnalando l’errore. Il *probe message* viene ricreato ad ogni ciclo di invio/ricezione, incrementando il suo **sequence number**, per permettere di far sapere al server lo stato di avanzamento dell’invio dei probe. Il payload invece, il quale non è altro che una sequenza dello stesso carattere, viene creato inizialmente della dimensione desiderata e riutilizzato ad ogni invio di probe.

A seguito dell’invio di messaggio di probe viene fatto partire un timer, il quale viene stoppato a non appena si riceve una risposta da parte del server. Se la risposta corrisponde esattamente al probe inviato viene valutato il RTT e stampato; se invece il messaggio di risposta non corrisponde, viene chiusa la connessione e terminato l’applicativo. Una volta inviati tutti i *probe message* viene calcolata una media dei loro RTT. Se il client ha richiesto un calcolo del *througput* allora questo viene calcolato dividendo la dimensione dei pacchetti inviati con la media dei RTT precedentemente calcolata. Come dimensione dei pacchetti viene presa quella relativa all’ultimo messaggio inviato; non viene tenuto conto del fatto che diversi numeri di sequenza comportano una diversa lunghezza del messaggio.

Infine, viene inviato il *bye message* al server e se ne attende la risposta. Una volta ricevuta, sia nel caso in cui questa sia positiva o negativa, la connessione viene chiusa e l’esecuzione dell’applicativo terminata.

## **Server**

Il server è sempre in attesa di connessioni TCP sulla porta indicata dall’utente in fase d’avvio. Possiede a sua volta una struttura dati simile a quella del client, con l’aggiunta di un campo.

/\* Service structure definition \*/

typedef struct node {

char measureType[6]; /\* Service requested - RTT or THPUT \*/

int nProbes; /\* Number of probe messages to send \*/

int messageSize; /\* Size of probe message's payload \*/

int serverDelay; /\* Delay of server probe message echo (in milliseconds) \*/

int connectionFD; /\* File descriptor of socket \*/

int phaseNumber; /\* 0: Ready to accept connection |

1: Ready to accept service request |

2: Ready to echo back |

3: Ready response to bye \*/

int probeSequenceNumberAwaited; /\* Probe sequence number awaited: useful in phase 2 \*/

} serviceNode;

serviceNode service;

Il campo *probeSequenceNumberAwaited* contiene il valore del sequence number che il server attende di ricevere nel prossimo probe. Se questo non corrisponde con quello del messaggio ottenuto viene inviato un messaggio di errore al client. Le informazioni della struttura dati vengono inizializzate nel momento in cui il server riceve un *hello message* da parte di un client, che contiene tutte le informazioni relative al servizio richiesto. Esistono tre pattern di messaggio diversi che il server può accettare: uno per l’*hello message*, uno per il *probe message* e uno per il *bye message.* In funzione del valore di *phaseNumber* il server verifica che il messaggio arrivato soddisfi il pattern relativo allo stato in cui si trova.

Una volta stabilita una connessione con un client, si passa allo stato di attesa di ricezione di un *hello* *message.* Una volta ricevuto, si valuta la sua sintassi e vengono memorizzati i valori dei suoi campi nell’apposita struttura dati. Se il controllo restituisce esito positivo viene inviato un messaggio di conferma al client e viene modificato lo stato, passando in quello di attesa di *probe message*, nel caso in cui invece l’esito sia negativo viene inviato al client un messaggio d’errore e la connessione viene chiusa, tornando allo stato iniziale.

Arrivato un messaggio di probe, questo viene controllato che sia come previsto: se il messaggio non soddisfa il pattern viene segnalato al client, la connessione viene chiusa e il server torna in attesa di una richiesta di connessione, se invece questo soddisfa i criteri del server, tra cui anche la seuquenzialità del sequence\_number, viene inviato al client lo stesso messaggio ricevuto. Una volta terminati i *probe message* previsti, si passa all’ultimo stato: l’attesa di un *bye message*.

Anche per il *bye message* si effettua un controllo quando viene ricevuto, e viene restituito al client un messaggio per fa sapere l’esito del server nella sua valutazione, in questo caso però viene in ogni caso chiusa la connessione.

## *Note*

**Entrambi i codici prevedono una frammentazione, per consentire di renderlo più leggibile. (COSA VUOL DIRE ? Boh, l’avevi scritto tu ma con mille parolone buttate a caso. Posso anche toglierlo per la poca importanza che ha. Forse l’ho scritto male io ma è per dire che non esiste una god-function e ci sono delle funzioni/procedure a cui vengono demandati specifici compiti)**

Per gestire la ricezione di messaggi di dimensione variabile, abbiamo deciso di sfruttare l’allocazione dinamica: in questo modo possono essere ricevuti pacchetti di qualsiasi dimensione. Viene accettato, come valore di payload, un qualsiasi valore positivo maggiore di 0: dati i limiti di memoria assegnata ai processi, non ne è garantito il funzionamento per ordini di grandezza eccessivi.

Per 1K abbiamo inteso 1000 byte, e non 1024.

Anche se non citato nelle specifiche, abbiamo preferito adottare l’uso di un messaggio di errore da parte del server per la fase di bye. In questo modo il client, nel caso di messaggio di bye errato, può terminare ugualemente.

Il termine di un messaggio è definito da un carattere *\n,* diversamente gli applicativi non hanno modo di terminare la ricezione di un messaggio. Per un singolo messaggio è prevista la possibilità di più letture del buffer, e meccanismi per ricostruirlo pezzo per pezzo. Eventuali ritardi del protocollo TCP nella consegna di un pezzo di messaggio non creano quindi problemi.

Non sono previsti timer: se una delle due parti dovesse terminare la propria esecuzione nel mezzo di uno scambio di messaggi, l’altro capo non ha modo di terminare la propria esecuzione autonomamente o, nel caso del server, ritornare nello stato di attesa iniziale.

Il server continua nella lettura fin quando non incontra un carattere *\n,* sempre presente alla fine dei messaggi del client*.* Il server invece invia messaggi che non prevedono l’uso finale del carattere, per questo motivo il client verifica direttamente che il messaggio arrivato coincida con uno dei messaggi previsti dal comportamento del server.

Se viene inviato un probe message in più l’errore viene rilevato: lato utente come probe rifiutato, mentre lato server come bye message errato, in quanto si aspetta proprio un bye message.

## **Grafici delle misure ottenute**

Le misure del RTT prelevate, mostrano la tendenza di aumentare all’incremento del numero di byte inviati. La variazione rimane nell’ordine di grandezza di frazioni di millisecondo in quanto, probabilmente, la misura non è estremamente suscettibile a variazioni di così piccola entità. Non viene qui mostrato un caso in cui, effettuando un aumento consistente al numero di byte trattati, l’RTT tende a crescere nell’ordine dei millisecondi, e non più frazioni di esso.

Il throughput tende invece a raddoppiare ogni volta che viene raddoppiato il numero di byte inviati; essendo il RTT poco suscettibile al variare di piccole quantità di byte, la divisione del calcolo del througput tende, al raddoppiarsi del numeratore, restituire il doppio del valore precedente. E’ dunque proporzionale all’aumento del numero di bytes inviati.

La misura deve ovviamente tenere conto del fatto che è stata realizzata utilizzando due processi in esecuzione sulla stessa macchina virtuale; di conseguenza i RTT analizzati risultano estremamente piccoli rispetto ad una comunicazione TCP nella rete Internet. Trattandosi di variazioni estremamente piccole, dipendono tutte estremamente dal sistema operativo e dallo scheduling dei suoi processi in esecuzione, infatti è possibile che si verifichino, all’aumentare di pochi byte risultati completamente differenti dalle aspettative. La tendenza generale dell’andamento del throughput, tenendo conto delle condizioni descritte, viene evidenziata negli schemi sottostanti.

Dai grafici con delay è possibile notare come l’RTT e throughput mantengono un andamento simile al caso precedente ma spostato di ordine di grandezza. Tanto più è alto il delay tanto più l’ordine di grandezza del throughput tende a diminuire, subendo infatti, a parità di numeratore, un aumento consistente del denominatore. L’RTT, come è ovvio, tende a crescere di pari passo al delay.