Relazione primo assegnamento

Scucchia Matteo, Ziani Andrea

Agosto 2018

1 Prima richiesta

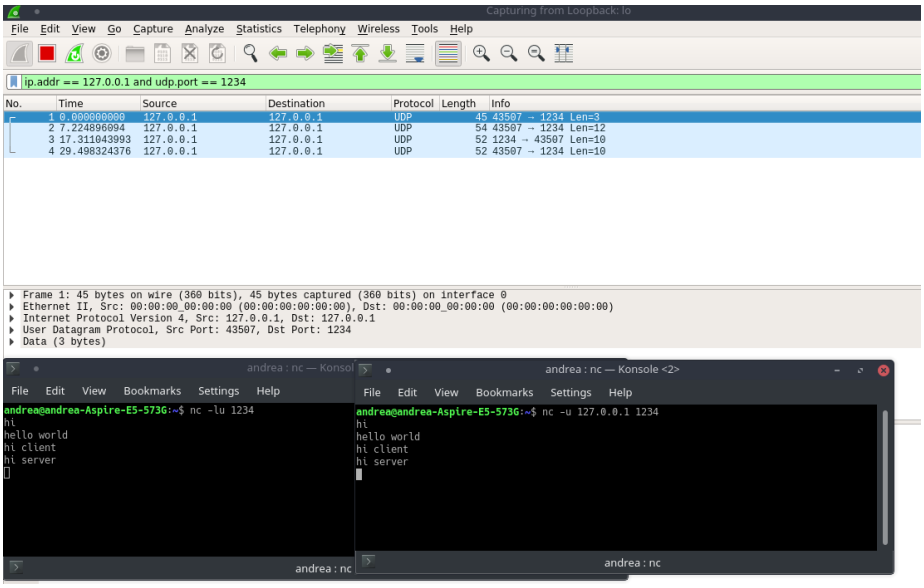
Per compiere la prima richiesta è stato sufficiente consultare il manuale della bash per comprendere il funzionamento di Netcat.

Per svolgere l'esercizio è bastato utilizzare il flag "-u" per utilizzare il protocollo UDP e seguire il manuale alla sezione "Client/Server Model". In questa parte viene spiegato che, per costruire un semplice modello client/server, vanno eseguiti i comandi "nc -l 1234" in una prima console e "nc 127.0.0.1 1234" in una seconda console.

Il primo comando citato mette in ascolto la console sulla socket port 1234, il secondo comando specifica l'ip e la socket port ai quali la console deve inviare i pacchetti. Eseguiti i due comandi aggiungendo il flag "-u" ad entrambi la connessione viene stabilita la connessione con tra client e server con protocollo UDP.

Una volta stabilita la connessione Netcat non si preoccupa più di quale sia il lato client e quale sia il lato server, infatti tutto ciò che viene scritto sulla prima console viene concatenato a ciò che viene scritto sulla seconda console.

Per catturare il traffico generato dal modello client/server di Netcat è stata eseguita una cattura tramite il software Wireshark in "Loopback interface" e filtrandola con "ip.addr == 127.0.0.1 and udp.port == 1234".

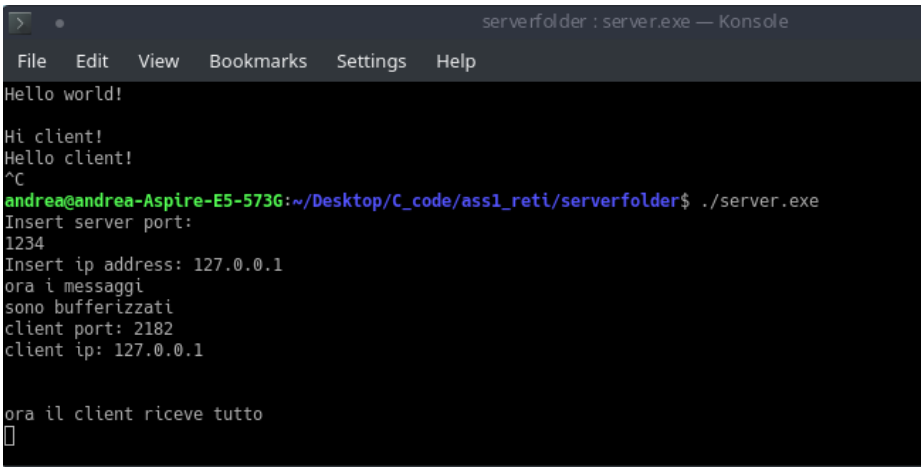


Screen della cattura Wireshark sul funzionamento di Netcat.

2 Seconda Richiesta

Per compiere la seconda richesta è stato analizzato l'esercizio visto in laboratorio, una volta comprese le funzioni utilizzate, è stato riadattato il codice per svolgere la richiesta di emulare il comportamento di Netcat.
Nell'implementazione della nostra soluzione è stata utilizzata la programmazione concorrente facendo uso dei pthread, con annesse funzioni di mutua esclusione. In una struttura di utilità sono state racchiuse tutte le variabili condivise dai thread, mentre abbiamo lasciato variabili globali solamente le mutex e le condition variable.

Netcat bufferizza i messaggi che vengono inviati dal server prima che esso sappia l'indirizzo IP del client; per fare ciò è stata utilizzata una coda nella quale memorizziamo temporaneamente i messaggi per poi inviarli tutti al primo messaggio scritto dopo che si è reso noto l'IP del client.



Screen del funzionamento del buffer del server

Screen del funzionamento della ricezione dei messaggi dal buffer.

2.1 Server:

Per prima cosa il server chiede all'utente su quale port si deve mettere in ascolto e quale sia il suo IP; quindi imposta la famiglia degli indirizzi, il protocollo utilizzato (in questo caso UDP) e la port sulla quale mettersi in ascolto.
Fatto questo il server divide il suo comportamento in tre thread.

Il **primo thread** si mette in attesa di ricevere pacchetti sulla port specificata e ogni volta che riceve un messaggio lo stampa su console insieme all'IP e alla socket port del client dal quale il messaggio è stato inviato.

Un **secondo thread** attende che l'utente inserisca i caratteri a console e manda

il messaggio al client (Si considera un singolo messaggio composto da tutti i caratteri fino all'andata a capo).

Alla ricezione del messaggio "exit" il server non stampa nulla a console, ma risveglia un **terzo thread**, rimasto fino a questo momento in attesa, che manda al client il messaggio "goodbye". Il primo thread che riceve i messaggi continua a svolgere le sue funzioni, mentre il terzo thread che manda il messaggio di goodbye una volta spedito il messaggio si mette in attesa.

Netcat non permette la connessione di più clinet al server; per emulare questo comportamento, viene salvato il primo indirizzo con il quale il server comunica e se gli indirizzi successivi non coincidono con tale, il server non considera questi messaggi.

2.2 Client:

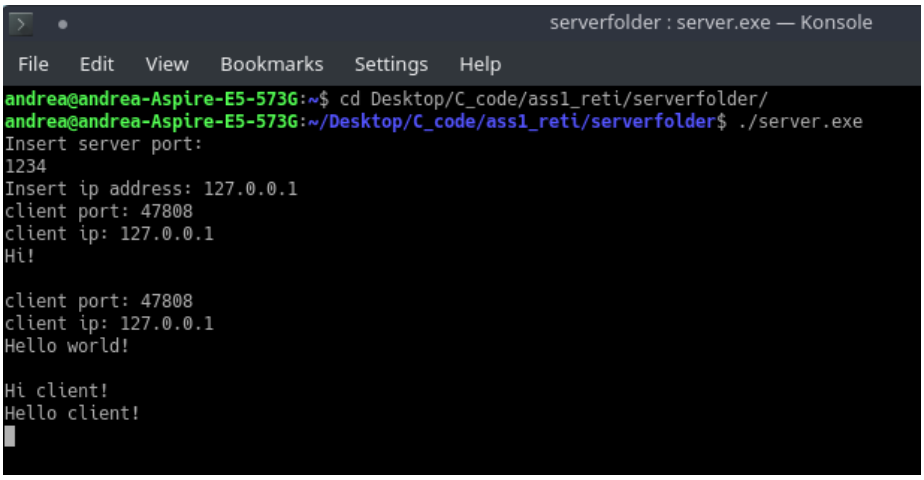
Il client chiede all'utente l'IP address e la port verso il quale deve inviare i pacchetti; quindi imposta la famiglia degli indirizzi, il protocollo utilizzato (in questo caso UDP) e la porta sulla quale inviare i pacchetti.
Fatto questo il client divide il suo comportamento in due thread.

Il **primo thread** attende che l'utente inserisca dei caratteri da console per mandare il messaggio al server.

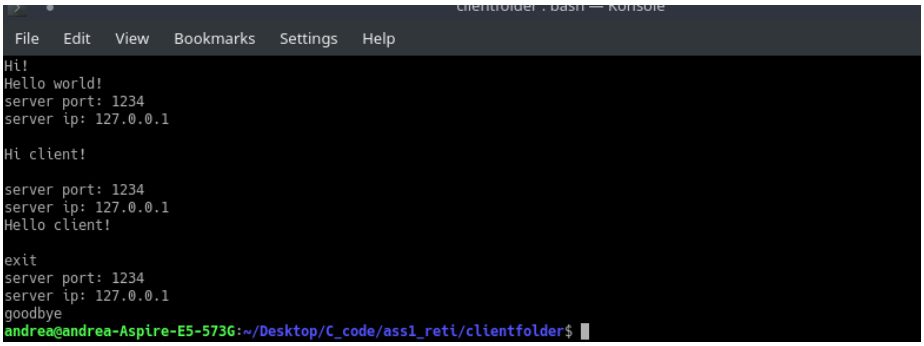
Il **secondo thread** si occupa di stampare a console i messaggi ricevuti dal server.

Per interrompere la connessione tra client e server basterà digitare la stringa "exit" premendo invio e attendere il messaggio "goodbye" dal server.

Il client, ricevuto il messaggio di terminazione dal server, visualizza il contenuto del pacchetto ricevuto su console e termina la connessione.



Screen del funzionamento del server



Screen del funzionamento del client

3 Terza Richiesta

Per rispondere alla terza richiesta è stato preso in esame il codice visto in classe, dato che il nostro modello client server emulando Netcat non accetta i messaggi provenienti da un secondo client.

Scenario 1:

Lanciando un'istanza server e due istanze client e provando a comunicare al server contemporaneamente dai due client, in un primo luogo, il comportamento del server del codice visto in classe potrebbe sembrare concorrente, questo perché le operazioni che deve svolgere non richiedono molte computazioni.

Scenario 2:

Inserendo la funzione "sleep()" nella porzione di codice che processa i messaggi in entrata, il comportamento del server si rivela sequenziale, in quanto è possibile notare le stampe su console dei messaggi in entrata uno alla volta intervallate dal periodo di tempo nel quale il server è in pausa.

La risposta alla domanda è, il server gestisce i messaggi in modo sequenziale.

Soluzione consigliata:

Per poter gestire la ricezione di messaggi in modo concorrente bisognerebbe utilizzare un thread di ricezione per ogni client che contatta il server. Un nuovo thread di ricezione viene fatto eseguire solo quando una nuova connessione viene stabilita ed ogni thread deve essere interrotto quando la connessione viene chiusa. Tale soluzione è efficace ma pericolosa, poichè se un numero molto elevato di client provassero a contattare il server, esso potrebbe non riuscire a gestire un numero così elevato di thread in esecuzione. Per impedire questo inconveniente, bisognerebbe perciò limitare il numero di connessioni che il server puiò accettare.