PRIMO LABORATORIO: TCP

TCP è un protocollo orientato alla connessione,classificato al livello di trasporto: prima di poter trasmettere dati stabilisce la comunicazione, negoziando una connessione tra i due host, client e server. Possiede quindi le funzionalità per creare, mantenere e chiudere/abbattere una connessione. Il protocollo permette alle due applicazioni di trasmettere contemporaneamente nelle due direzioni, "Full-Duplex", fornendo ai livelli superiori un servizio equivalente ad una connessione fisica diretta che trasporta un flusso di byte, attraverso messaggi di acknowledge e timeout per la ritrasmissione. Inoltre fornisce connessioni multiple attraverso l'uso di diverse porte.

La nostra esercitazione consisteva nell'implementare il lato client di una connessione tcp, completando il codice datoci.

INT MAIN

```
... int main(int argc, char *argv[]) { ...
```

Il nostro programma si aspetta al momento dell'esecuzione un certo numero argc di parametri, memorizzato nell'array puntato da argv di lunghezza argc. Se così non è, verrà notificato il seguente errore: "Incorrect parameters provided. Use: tcp_ping PONG_ADDR PONG_PORT SIZE\n".
Per creare la comunicazione infatti ci servono il nome dell'host "pong", la porta su cui contattarlo e la grandezza del messaggio. Si può accedere a queste info andando a leggere nelle celle dell'array argv.

Prima di tutto si esegue la memset(&hints, 0, sizeof hints), per assicurarsi che la struct hints sia vuota. Verrà poi "riempita" con la getaddrinfo().

```
/*** Initialize hints with proper parameters ***/
hints.ai_family=AF_INET;
hints.ai_protocol= IPPROTO_TCP;
hints.ai_socktype=SOCK_STREAM;
```

"int ai_family" può avere tre valori diversi a seconda delle diverse famiglie di protocolli(IPv4, Ipv6 o Unix); nel nostro caso è AF_INET (Ipv4), che definisce così la struct sockaddr_in.
"int ai_protocol" è ovviamente il TCP, mentre "ai_socktype" è di tipo stream.
"hints.ai_flags = AI_PASSIVE" non è specificato, allora l'indirizzo restituito potrà essere associato ad un socket per connettersi o inviare messaggi.

```
if(getaddrinfo(argv[1], argv[2], &hints, &serverAddr)<0){
//printf("%s\n", gai_strerror(getaddrinfo(argv[1], argv[2], &hints, &serverAddr)));
fail_errno("\nerrore getaddrinfo()\n");
}
```

La funzione getaddrinfo serve per effettuare DNS lookup e service name lookup, e prende come argomenti:
-const char* nodename: l'host name, che nel nostro caso è PONG_ADDR, ovvero argv[1].
-const char* servername: può essere il numero di una porta o un indirizzo "http...", ma nel nostro caso è PORT_NUMBER, argv[2].
-const struct addrinfo *hints: struttura per i parametri in input
-struct addrinfo **res: serverAddr punta ad una linked list di 1 o più struct addrinfo
Questa funzione ritorna 0 in caso di successo, un numero negativo altrimenti. Commentata c'è la funzione gai_strerror() per chiedere la stampa del tipo di errore tornato da getaddrinfo.

```
/*** create a new TCP socket and connect it with the server. Utilizzare le system call socket() e connect() e,
in caso di errore, la funzione fail_errno() definita nel file function.c***/
if((tcp_socketFD= socket(AF_INET, SOCK_STREAM, 0))==-1)
fail_errno("\nerrore apertura socket()\n");
if(connect(tcp_socketFD, serverAddr->ai_addr, serverAddr->ai_addrlen)==-1)
fail_errno("\nerrore connect()\n");
/*** TO BE DONE ***/
```

Per creare un canale di comunicazione chiamiamo la funzione socket, che prende come parametri dominio(AF_INET, ...), tipo(SOCK_STREAM, ..) e protocollo(se 0 viene associato automaticamente il protocollo al tipo definito). Ritorna un socket descriptor oppure -1 in caso di errore. Salviamo questo valore in tcp_socketFD, per un futuro utilizzo.
La funzione connect() cerca di effettuare la connessione tra il socket passato come parametro con il socket in ascolto all'indirizzo specificato. I parametri richiesti sono:
-int sockfd: tcp_socketFD, socket descriptor, ritornato da socket()
-struct sockaddr* serv_addr: contiene info sull'indirizzo remoto
-int addrlen: lunghezza in byte dell'indirizzo.

Questi due ultimi vengono presi dalla struct serverAddr.

```
Sscanf( argv[3], "%d", &msgsz );
if ( msgsz < MINSIZE )    msgsz = MINSIZE;
else if ( msgsz > MAXTCPSIZE )  msgsz = MAXTCPSIZE;
printf(" ... connected to Pong server: asking for %d repetitions of %d Bytes TCP messages\n", REPEATS,msgsz);
sprintf(request,"TCP %d %d\n", msgsz, REPEATS);
```

In questa porzione si legge da argv[ ] la dimensione del messaggio che il client intende spedire e lo si controlla affinché sia compreso tra MINSIZE e MAXTCPSIZE ed infine salvato il msgsz. Si prepara poi la richiesta di ping-pong scrivendo la stringa “TCP” seguita dalla lunghezza dei messaggi in byte e dal numero di ripetizioni, seguiti dal carattere '\n'. La funzione sprintf scrive questa stringa nel buffer 'request'.

```
/*** Write the request on socket ***/
if( write( tcp_socketFD, request, strlen(request) ) > msgsz)  fail_errno("\nerrore write()\n");
```

La funzione “int write(int fildes, const void *buf, int nbyte)” scrive nbyte bytes presi dall'array request nel file associato al file descriptor aperto(tcp_socketFD). Ritorna il numero di byte attualmente scritti nel file associato, numero che non può essere più grande di nbyte; in caso di errore ritorna -1. Nel nostro caso il numero di byte è ritornato dalla funzione strlen().
La risposta restituita dal server Pong viene letta dal socket con la funzione read(). Se la risposta è “ERROR\n” si segnala l'errore, se invece è ”OK\n” si prosegue.

```
if(answer[0]=='E'){ fail_errno("\n errore risposta server \n"); }
```

Abbiamo deciso di controllare soltanto la prima lettera del buffer answer, dato che le possibili risposte del server potevano essere soltanto due. Altrimenti si sarebbero potute confrontare le due stringhe.

```
for ( rep = 1 ; rep <= REPEATS ; rep++ ) {
currentPingTimes[rep-1] = doPing( msgsz, rep, message, tcp_socketFD );
printf("Round trip time was %lg milliseconds in repetition %d\n", currentPingTimes[rep-1],rep);
}
```

Si entra ora nel ciclo for per effettuare le 101 ripetizioni di chiamata della funzione doPing() che invia il messaggio e attende una risposta. I suoi parametri sono:
-int msg_size: lunghezza del messaggio
-int msg_no: numero della ripetizione (scritto nel messaggio)
-char message[msg_size]: buffer
-int tcp_socketFD: socket file descriptor
Ritorna un double.

```
shutdown(tcp_socketFD, SHUT_RDWR);
freeaddrinfo(serverAddr);
close(tcp_socketFD);
```

Shutdown per chiudere totalmente o parzialmente la connessione full-duplez sul socket associato al descrittore.
Freeaddrinfo per disallocare le strutture dati dinamiche che non servono più.
Close per chiudere la comunicazione sul socket passato nell'unico argomento, chiamata da entrambi i socket in comunicazione.

DOUBLE DOPING

.. double doPing(int msg_size, int msg_no, char message[msg_size], int tcp_socketFD) ..

```
/*** write msg_no at the beginning of the message ***/
sprintf(message,"%d\n",msg_no);
```

Nel buffer message viene messo il numero di sequenza seguito da '\n'.

```
/*** Store the current time in sendTime ***/
```

clock_gettime(CLOCK_REALTIME, &sendTime);

Questa funzione inserisce il sendTime il tempo attuale e ritorna un dato di tipo timespec che contiene i secondi (tv_sec) e le frazioni in nanosecondi(tv_nsec) a partire da Jan 1, 1970 00:00:00 UTC.

/*** Send the message through the socket ***/
sentBytes= send(tcp_socketFD, message, msg_size, 0);

Con send() è possibile inviare messaggi dal socket rappresentato dal file descriptor al socket con cui è connesso, ovviamente se è presente una connessione.
Prende come parametri:
-int sockfd: socket file descriptor
-const void *msg: buffer che contiene il messaggio e deve avere una dimensione non inferiore a msg_size
-int len: lunghezza del messaggio
-int flags: se vale 0, allora send() e recv() equivalgono, rispettivamente alle system call write() e read().

/*** Receive an answer through the socket (blocking) ***/
for (offset = 0 ; (offset+(recvBytes = recv(tcp_socketFD,rec_buffer+offset,sentBytes-offset,MSG_WAITALL))) < msg_size ; offset += recvBytes) {
//printf(" ... received %d bytes back\n", recvBytes);
if (recvBytes < 0) fail_errno("Error receiving data");
}

Recv() serve a ricevere messaggi da un socket; il risultato è -1 in caso di errore (condizione che farebbe terminare il ciclo for), oppure il numero di caratteri ricevuti.

/*** Sample the current time in recvTime and return the difference ***/ /*** TO BE DONE ***/
clock_gettime(CLOCK_REALTIME, &recvTime);

Prendo il tempo attuale e lo inserisco in recvTime, così da poter fare la differenza tra tempo iniziale e tempo finale.

return timespec_delta2milliseconds(&recvTime,&sendTime);

La doPing ritorna il double ritornato a sua volta dalla funzione timespec_delta2milliseconds che prende come parametri i due diversi tempi presi in precedenza.

DOUBLE TIMESPEC_DELTA2MILLISECONDS
double timespec_delta2milliseconds(struct timespec * from, struct timespec * to) {

double sendT;
double rcvT;
sendT=(double)(to->tv_sec*1000.0)+((double)to->tv_nsec)/1000000.0;
rcvT=(double)(from->tv_sec*1000.0)+((double)from->tv_nsec)/1000000.0;

return rcvT-sendT;
}

Per entrambi gli orari viene effettuata una trasformazione in millisecondi per avere risultati più “ragionevoli”, facendo la differenza tra il tempo di ricezione del messaggio e il tempo di invio dello stesso.

Confronto con UDP

TCP è un protocollo orientato alla connessione, quindi per stabilire, mantenere e chiudere una connessione, è necessario inviare pacchetti i quali aumentano l'overhead di comunicazione. Al contrario, UDP è senza connessione ed invia solo i datagrammi richiesti dal livello applicativo, non offre nessuna garanzia sull'affidabilità della comunicazione (cioè sull'effettivo arrivo dei datagrammi) e sul loro ordine in sequenza in arrivo. Invece TCP tramite i meccanismi di acknowledgement e di ritrasmissione su timeout riesce a garantire la consegna dei dati, anche se al costo di un maggiore overhead. L'oggetto della comunicazione di TCP è il flusso di byte mentre quello di UDP è il singolo datagramma. L'utilizzo del protocollo TCP rispetto a UDP è, in generale, preferito quando è necessario avere garanzie sulla consegna dei dati o sull'ordine di arrivo dei vari segmenti. Al contrario UDP viene principalmente usato quando l'interazione tra i due host è simmetrica o nel caso si abbiano forti vincoli sulla velocità e l'economia di risorse della rete.

SECONDA ESERCITAZIONE: UDP

Udp è uno dei principali protocollo di rete. È un protocollo a livello di trasporto a pacchetto ed è usato solitamente con il protocollo di livello di rete IP.

Le principali differenze del prtocollo UDP rispetto a TCP sono :

. È un protocollo di tipo STATELESS il che significa che non tiene nota dello stato delle connessione e che deve tenere in memoria meno informazioni relative ad essa quindi per esempio, un server che svolga una determinata applicazione con UDP come protocollo,puo supportare molti più clienti attivi allo stesso tempo.

. Ha una minore affidabilità per quanto tiguarda la gestione dei pacchetti inviati e ricevuti in quanto non gestisce il riordinamento dei pacchetti né la ritrasmissione di quelli persi.

. Ha una maggiore efficienza e velocità in quanto non c'è latenza dovuta al riordino e ritrasmissione. Motivo per cui viene utilizzato spesso per le applicazioni che tollerano la perdita di pacchetti e richiedono un bit-rate di trasmissione minimo per evitare di ritardare la loro trasmissione.

Il protocollo UDP fornisce servizi base dei livelli di trasporto:

. MULTIPLEXING delle connessioni: meccanismo per l'assegnazione delle porte.

. CHECKSUM: verifica degli errori (integrità dei dati) inserita in un campo all'interno dell'intestazione del pacchetto.

Questo genere di protocollo viene usato da:

. Applicazioni elastiche rispetto alla perdita di dati e strettamente dipendenti dal tempo.

. Comunicazioni in BROADCAST: invio su tutti i terminali di una rete locale.

. Comunicazioni in MULTICAST: invio su tutti i terminali iscritti a un servizio.

Datagramma UDP:

+	Bit 0-15	16-31
0	Source Port (optional)	Destination Port
32	Length	Checksum (optional)
64+	Data	

Source port: numero di porta dell'host mittente.
Destination port: numero di porta dell'host destinatario.
Length: lunghezza totale in byte del datagramma UDP (header+data).
Checksum: codice di controllo del datagramma (algoritmo specificato in RFC del protocollo UDP).
Data: dati del messaggio.

FILE UDP_PING.C

Le System Call per stabilire una connessione fra processi,sono basate sulle primitive di base per la gestione dei Socket. Per stabilire la connessione all'interno del main nel file udp_ping.c ,mi comporto esattamente come è stato fatto per il protocollo TCP ad eccezione del fatto che, una volta ricevuta la risposta di OK, il client Ping UDP deve chiudere la connessione TCP, creare un nuovo Socket di tipo DGRAM e cominciare ad inviare datagrammi UDP al server Pong sulla porta precedentemente indicata dal server.

```
112 int prepare_udp_socket(char * pong_addr, char * pong_port) {
113     struct addrinfo hints, *pongAddr = NULL;
114      struct sockaddr_in *ipv4;
115     int pingscktFD;
116     int gai_rv;
117     int np;
118
119     /*** Specify the UDP sockets' options ***/
120     memset(&hints, 0, sizeof hints);
121
122     /*** Initialize hints with proper parameters ***/
123         hints.ai_family=AF_INET;
124         hints.ai_protocol=IPPROTO_UDP;
125         hints.ai_socktype=SOCK_DGRAM;
```

La funzione di memset() inizializza i primi sizeof byte dell'area di memoria con la costante 0 dopodiche ritorna un puntatore a hints.
Una volta allocata e inizializzata a zero l'area di memoria dedicata alle hints è necessaria l'inizializzazione dei suoi campi con i parametri opportuni :

. AF_INET(tipo di dominio del socket usato): formato degli indirizzi dell'host e numero di porta(generalmente usato per la comunicazione dilla rete internet supportata da IPv4).
. IPPROTO_UDP: protocollo usato per la connessione.
. SOCK_DGRAM: tipo di socket.

```
127         if(getaddrinfo(pong_addr, pong_port, &hints, &pongAddr)<0){
128                 fail_errno("getaddrinfo\n");
129         }
```

Getaddrinfo() inizializza il tipo addrinfo definita dal file di sistema nerdb.h.
La struttura addrinfo serve alle funzioni per la gestione dei socket, ci permette di conoscere quale protocollo a livello transport (UDP), livello network(IPv4) e indirizzo IP utilizzare per connettersi e su quale porta.

```
133         if( (pingscktFD= socket(hints.ai_family, hints.ai_socktype, hints.ai_protocol))==-1){
134                 fail_errno("socketT\n");
135         }
136     /*** TO BE DONE
137          Utilizzare la system call socket() e, in caso di errore,
138          la funzione fail_errno() definita nel file function.c   ***/
139
140     /*** change socket behavior to NONBLOCKING ***/
141     fcntl(pingscktFD,F_SETFL,O_NONBLOCK);
142
```

Socket() creo un canale di connessione socket. Poichè stiamo stabilendo una connessione udp, è necessario cambiare i parametri del socket in modo che, prima di iniziare a scambiare messaggi abbia uno stato non bloccante (operazione compiuta dalla dunzione fcntl()).

```
144         ipv4 = (struct sockaddr_in *)pongAddr->ai_addr;
145
146         if((connect(pingscktFD, pongAddr->ai_addr,pongAddr->ai_addrlen))<0){
147                 printf("valore di ritorno della socket()=%d\n", pingscktFD);
148                 fail_errno("connect\n");
149         }
150     /*** TO BE DONE ***/
151
152     return pingscktFD;
```

Per generare le funzioni di libreria, i socket hanno tra i loro argomenti un puntatore alla struttura sockaddr. L'accesso agli indirizzi in struttura ed il loro utilizzo è fatto tramite dei cast (diversi a seconda degli indirizzi usati). Nel nostro caso, addrinfo contine i dati sull'indirizzo della struttura sickaddr nel formato a 14 byte; per conoscere l'indirizzo IPV4 e

il numero di porta per la connessione, sarà necessario usare l'operatore di casting che ci permette poi di accedere ai campi sin_port sin_addr della struttura sockaddr_in.

La funzione di connect() infine, permette di connettere il socket all'indirizzo e porta del server.

Sul nuovo socket devono essere inviati i messaggi concordati.

```
43 double doPing(int msg_size, int msg_no, char message[msg_size], int pingSocketFD,
44               double TimeOut) {
45     char answer_buffer[msg_size];
46     int recvBytes,sentBytes, offset;
47     struct sockaddr_storage actualPongAddr;
48     int actPongAddrLen = sizeof(struct sockaddr_storage);
49     struct timespec sendTime,recvTime;
50     double roundTripTimeMilliSeconds;
51     int re_try = 0;
52
53     /*** write msg_no at the beginning of the message ***/
54         sprintf(message, "%d\n", msg_no);
55     /*** TO BE DONE ***/
56
57     do {
58         /*** Store the current time in sendTime ***/
59         clock_gettime(CLOCK_REALTIME, &sendTime);
60         /*** TO BE DONE ***/
```

Sprintf() contine il tipo di messaggio da inviare:
. message : contiene le stringhe di caratteri "1\n", "2\n" (in testa).
. msg_no : il numero di invio fino a completare il numero concordato (101).

clock_gettime() permette di inizializzare la struttura timespec sendTime che salva il tempo attuale.

```
65         sentBytes=send(pingSocketFD, message, msg_size, 0);
```

Send() é la system call orientata alla connessione In questo caso la flag della send è posta a zero quindi la send si comporta esattamente come si comporterebbe la system call write per l'invio del messaggio.
. PingSocketFD: socket che invia il messaggio.
. message: messaggio da inviare.
. msg_size: dimensione del messaggio.
. 0: flag per il comportamente della funzione.

sentByte contiene il valore del ritorno della funzione: numero di Byte inviati a pong.

```
71         recvBytes = recv(pingSocketFD,answer_buffer,sizeof(answer_buffer),0);
72
```

Recv() è usata per ricevere messaggi da un socket. Come la send(), anche la recv() viene normalmente usata su un socket orientato alla connessione. La funzione ritorna la lunghezza del messaggio ricevuto. Se il messaggio non arriva, la funzione resta in attesa a meno che il socket non sia di tipo non bloccante, in questo caso viene ritornato il valore -1 e ERRNO settata con l'opportuno valore.

```
74         clock_gettime(CLOCK_REALTIME, &recvTime);
75         /*** TO BE DONE ***/
76
77         roundTripTimeMilliSeconds = timespec_delta2milliseconds(&recvTime,&sendTime);
78
```

recvTime viene settato chiamando la funzione gettime che stima il valore del tempo attuale.
Con la chiamata alla funzione timespec_delta2millisecond(), calcolo il round trip time: la differenza fra il tempo di

ricezione e quello di invio in millisecondi.

```
79        for ( offset = 0 ; ( recvBytes >= 0 || errno == EAGAIN || errno == EWOULDBLOCK ) &&
80              (recvBytes+offset) < sentBytes && roundTripTimeMilliSeconds < TimeOut;
81              offset += ((recvBytes > 0)? recvBytes : 0) ) {
82            recvBytes = recv(pingSocketFD,answer_buffer+offset,sizeof(answer_buffer)-offset,0);
83
84            /*** Sample the current time in recvTime and compute difference ***/
85        clock_gettime(CLOCK_REALTIME, &recvTime);
86            /*** TO BE DONE ***/
87
88            roundTripTimeMilliSeconds = timespec_delta2milliseconds(&recvTime,&sendTime);
89        }
90        if ( recvBytes < 0 && errno != EAGAIN && errno != EWOULDBLOCK )
91            fail_errno("UDP ping could not recv from UDP socket");
92        if ( (recvBytes+offset) < sentBytes ) { /*time-out elapsed: packet was lost*/
93            lost_count++;
94            if ( recvBytes < 0 )
95                recvBytes = offset;
96              else
97                recvBytes += offset;
98            printf("\n ... received %d bytes instead of %d (lost count = %d)\n",
99                   recvBytes, sentBytes, lost_count);
100           if ( ++re_try > MAXUDPRESEND ) {
101               printf(" ... giving-up!\n");
102               break;
103           }
104           printf(" ... re-trying ...\n");
105       }
106   } while ( sentBytes != recvBytes );
107
108   return roundTripTimeMilliSeconds;
109 }
```

Una delle principali differenze rispetto al ping-pong TCP deriva dalla inaffidabilità del protocollo UDP: i datagrammi possono andare persi, e quindi occorre gestire un meccanismo di time-out sul client Ping per gestire le situazioni di mancanza di risposta.
Si contano il numero totale di datagrammi persi e si effettuano un numero di tentativi di ritrasmissione prima di passare allo scambio del datagramma successivo, in modo da garantire comunque la terminazione del programma. Nel caso di perdita di un datagramma si ritorna come valore di RTT il tempo effettivamente speso in attesa (>= time-out).

```
262     close(pingSocketFD);
```

Close(), al termine del main() prende come parametri il file descriptor del socket da chiudere e termina la connessione.

TERZA E QUINTA ESERCITAZIONE: SCRIPT BASH/ BANDA-LATENZA
SCRIPT BASH

Siamo partite dal file collect_throughput.bash originario, in cui abbiam notato il primo if, il quale deve garantire che, alla chiamata di questo file, il nome di un protocollo di trasporto venga fornito.
In questo file ci occuperemo di gestire i dati forniti dalle chiamate a Felix rispetto ai due protocolli affrontati, per ottenere: valori di RTT reali (sample), stimati (estimated), variabilità tra questi due (variability), valori di Throughput reali (throughput) e stimati (col modello di banda latenza).

Nel primo for, per tutti i file *.out presenti nella cartella data, vogliamo inserire nell'array Tstring tutte le righe che contengono la parola “Throughput”. Queste righe sono composte da nove 'parole' separate da spazi. In Sstring idem, cercando però le linee contenenti la parola “repetitions” e “Ping”. In Dstring inseriamo invece le linee con “RTT” e “average”, che ci serviranno successivamente per il calcolo del Delay.
In unsorted.dat inseriamo, sempre per ogni file, Sstring[7] (cioè la grandezza del messaggio sul quale è stato calcolato il RTT, etc..), Tstring[8] (cioè il valore dell'overall Throughput per quella grandezza di messaggio) e 0.5*Dstring[11] (cioè il calcolo, attraverso la funzione | bc, di mezzo RTT average, che idealmente corrisponde al Delay di andata per quella grandezza).
Alla fine del for, chiediamo all'algoritmo di sorting di riordinare i nostri dati ed inserirli in un nuovo file: $1_throughput.dat.

Ci preoccupiamo poi di osservare i file del tipo $1_32.out, sempre in data, e, in questi, cercare le righe che contengano le parole “Round” e “repetition”. Queste righe (tutte della forma:

Round trip time was 0.92627 milliseconds in repetition 101, quindi nove 'parole' contate da 0 a 8) vengono poi salvate, una dopo l'altra, nell'array Rtt.
Viene poi creata una variabile var, che ci aiuterà in seguito per il calcolo dei RTT stimati e per il calcolo della variabilità. Qui la inizializziamo a zero per entrambi i protocolli e la inseriamo nel file corrispondente $1_var.txt
Nel for successivo, col l'aiuto di un indice, scorriamo l'array Rtt appena prodotto in cerca dei valori che ci occorrono. Se prendiamo in considerazione una riga singola, si deduce facilmente che le informazioni necessarie, ovvero Rtt e numero di ripetizione, sono alla posizione 4 e 8. Per ottenere le stesse info da tutte le righe, facciamo scorrere il nostro indice da 0 a 909 (9 'parole' per 101 ripetizioni) e ad ogni giro, le variabili ival(indice valore) e irip (indice ripetizione) salvano temporaneamente ciò che è necessario. Un'istruzione echo inserisce questi due valori in $1_unsorted2.dat, subito riordinati col comando di sort e inseriti in $1_sample32.dat.

Come dimensione minima dei nostri messaggi abbiamo scelto 32 byte per entrambi i protocolli, ma per messaggi grandi dovevamo distinguere. Perciò ora ci chiediamo: che tipo di protocollo è stato passato come parametro?
Nel caso sia “tcp” ci preoccupiamo di osservare il file $1_262144.out e seguiamo lo stesso iter seguito per $1_32.out, con l'unica differenza di inserire i dati ordinati nel file $1_sample262144.dat
Nel caso sia “udp” consideriamo il file $1_16384.out e , sempre con lo stesso iter, portiamo i nostri dati ordinati in $1_sample16384.dat.
Con /bin/rm -f rimuoviamo i file unsorted che non utilizzeremo più.

Ottenuti i valori RTT reali, prendiamo il file $1_sample32.dat e cerchiamo la riga con numero di ripetizione=1, inserendola in Rtt. Immediatamente Rtt[1] (cioè il valore di RTT sample alla ripetizione 1 per msg_sz=32) viene inserito in $1_stimato.txt; questa è la nostra inizializzazione per la variabile, definita successivamente, estimated.
Entriamo in un ciclo for con indice da 1 a 101, con incremento di passo 1. La prima operazione sarà estrarre il valore dal file $1_stimato.txt ed inserirlo in estimated (al primo giro questa sarà zero, ma alla fine di ogni giro verrà sostituita).
Cominciamo a costruire il file $1_estimated32.dat inserendo numero della ripetizione e valore RTT stimato.
In Rtt ora inseriamo la riga corrispondente alla ripetizione attuale (al primo giro questa riga sarà identica a quella cercata subito prima di questo for), per salvare in sample il valore dell'RTT reale. In var riprendiamo il valore iniziale della variability (0 al primo giro) e siamo pronti per il calcolo della variabilità. Abbiamo usato la formula: 0.25*| sample(i)-estimated(i)|+0.75*varibility(i-1).
Per il calcolo del valore assoluto abbiamo deciso di sfruttare il concetto a=sqrt(a*a) per assicurarci che il valore fosse sempre positivo. Cominciamo ad inserire in $1_variability32.dat la prima riga: indice e valore di variability nuovo.
Ci ricordiamo poi di aggiornare con lo stesso valore il file $1_var.txt e di calcolare il nuovo valore di RTT stimato da inserire nel file $1_stimato.txt.
Alla fine di questo for avremo RTT reale, stimato e variabilità per entrambi i protocolli con msg_sz=32.

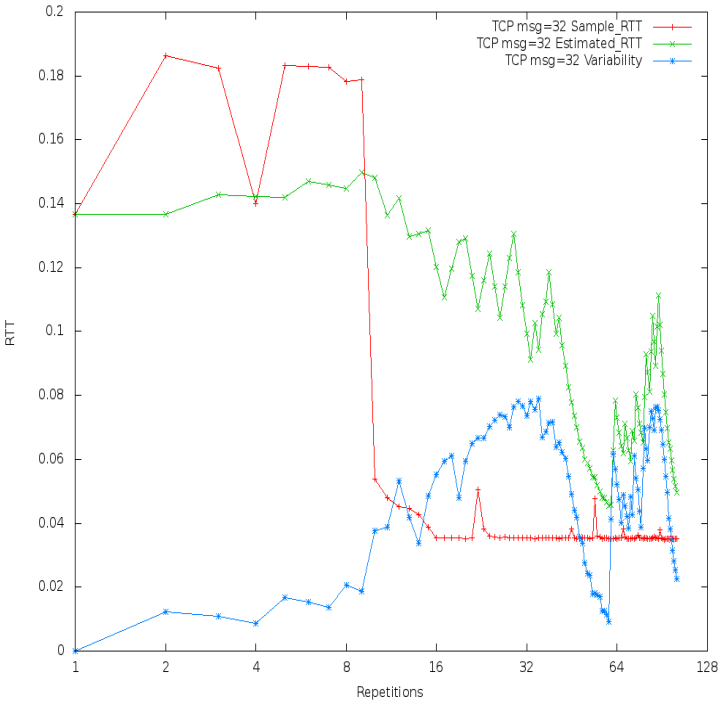
Per le altre grandezze bisogna di nuovo distinguere in base ai due protocolli, quindi, con un if del tutto identico al precedente, ci chiediamo con quale protocollo stiamo lavorando. Se “tcp” ci troveremo ad avere $1_variability262144.dat, $1_estimated262144.dat, altrimenti $1_variability16384.dat, $1_estimated16384.dat, con le stesse identiche operazioni del for precedentemente descritto.
Eliminiamo i file su cui abbiamo salvato valori temporanei che non saranno più utili successivamente.

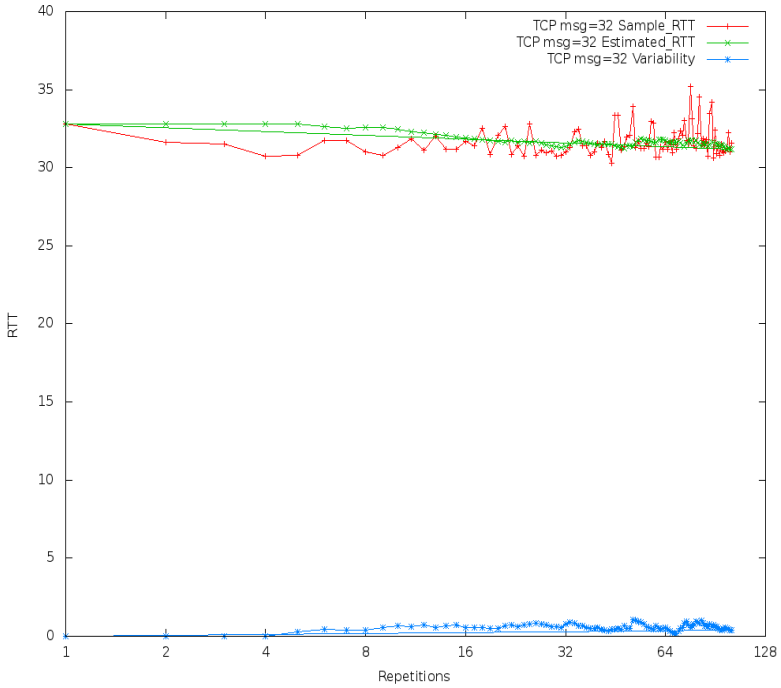
Abbiamo lasciato come ultimo il calcolo del Throughput stimato.
Partendo ovviamente dal file $1_throughput.dat, e distinguendo i due protocolli, prendiamo le righe corrispondenti alle grandezze precedentemente scelte (32-16384 per UDP, 32-262144 per TCP), salvandole in due array diversi.
Ricordiamo che questo file contiene tre colonne: msg_sz thoughput e delay.
Innanzi tutto calcoliamo i valori di B e L0 con le formule tratte dal testo dell'esercitazione.
Per riempire il file $1_modelloBL.dat, vogliamo servirci di un for, ma ovviamente l'indice di stop è diverso per i due protocolli. Perciò, con un ulteriore if, ci accertiamo che le grandezze vengano rispettate. Per il calcolo del Throughput stimato, abbiamo unito la formula T(N)=N/D(N) e D(N)=L0+N/B, ottenendo T(N)= (N*B)/(L0*B + N), così da svolgere tutto in una sola riga.

La grandezza per udp è 16384 e non 32768 perchè quest'ultima grandezza ci dava problemi, creando il file udp_32768.out con soli messaggi del tipo:

... received 0 bytes instead of 32768 (lost count = 1)
... re-trying ...
SEND TIME 1352559338 : 427965298
lunghezza messagio: 32768

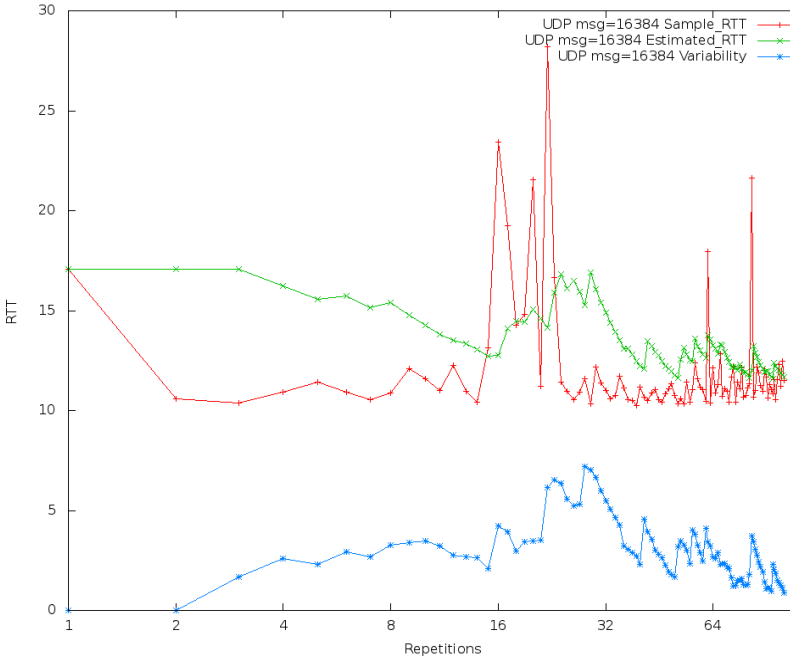
Abbiamo scelto di escluderlo dal calcolo dei valori per i grafici perchè non significativo.
Rendiamo noto che nel cercare di raccogliere dati da una comunicazione con Felix da La Spezia abbiamo avuto molte difficoltà. A volte la connessione veniva chiusa da Felix prima del termine delle ripetizioni, altre volte, con il protocollo udp, molti pacchetti venivano persi. I dati nella cartella data/felix_da_LaSpezia infatti, per udp, hanno valori significativi sono fino alle grandezza 1024, dopodiché abbiamo ricevuto sempre 0.

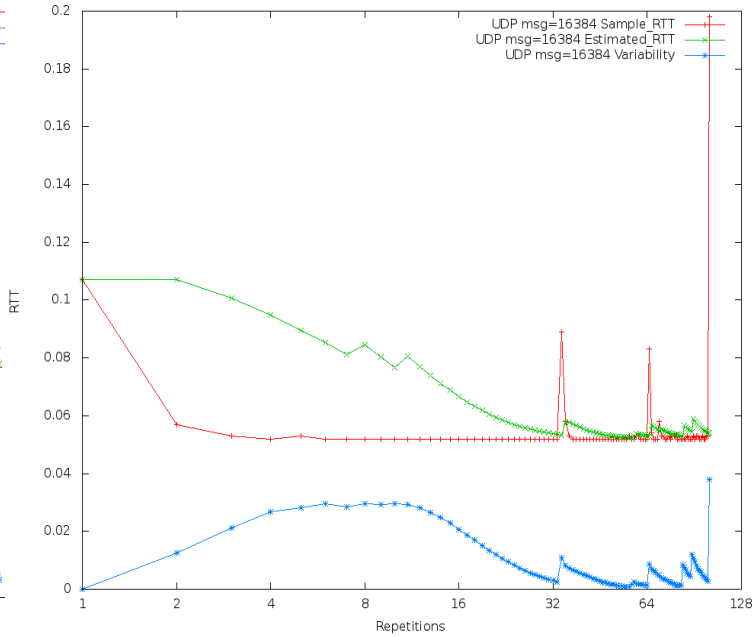




messaggi piccoli

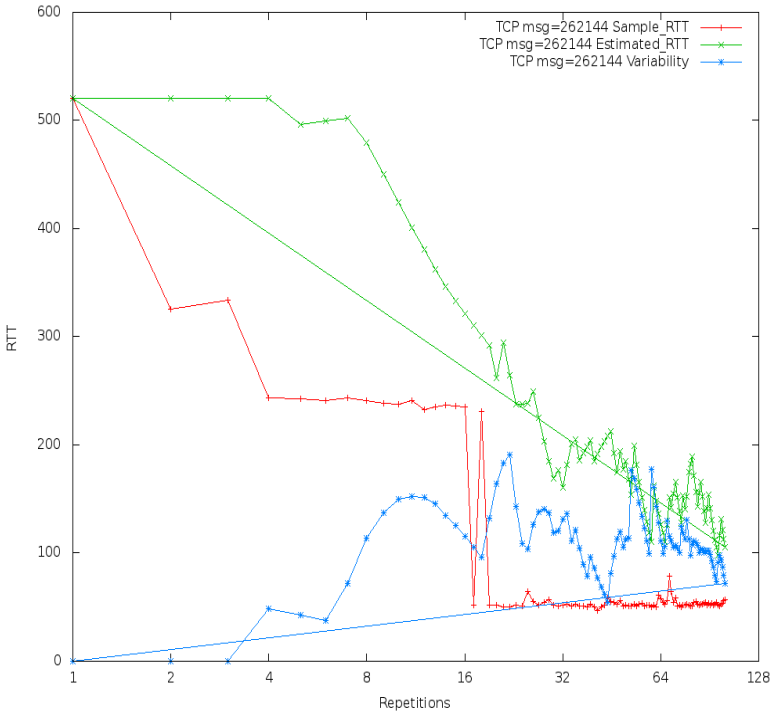
A sinistra abbiamo RTT_32_tcp in localhost, mentre a destra una comunicazione con Felix da La Spezia. Si può notare la grande differenza di variabilità; infatti mentre a sinistra, dopo una prima difficoltà, i valori tendano ad abbassarsi creando molta differenza tra reale(rosso) e stimato (verde), a destra la questione si mantiene abbastanza costante. I valori di RTT però sono differenti di parecchio: mentre a sinistra il valore massimo si ha verso la seconda ripetizione con valore 0.18, a destra i valori sono sempre compresi tra i 30 e 40 ms.

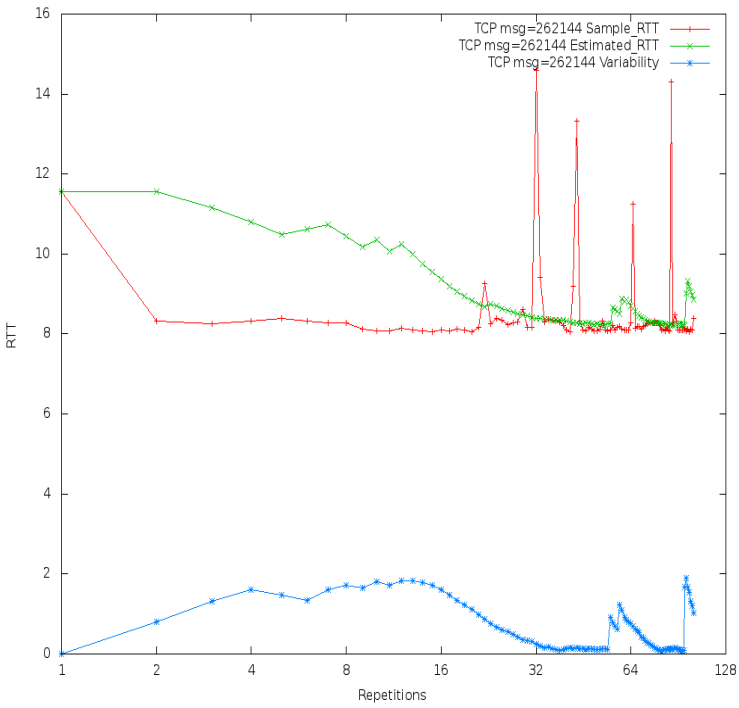




messaggi grandi UDP

A sinistra RTT_16384_udp a Felix dai laboratori del Disi, mentre a destra localhost.
L'udp (o perlomeno il nostro udp) ha sempre qualche difficoltà con i primi messaggi, che hanno sempre un valore più alto rispetto ai successivi, il che fa variare molto i valori stimati e di conseguenza anche la variability.
A sinistra la maggior parte dei valori è compresa nella regione tra i 10 e 15 ms, mentre in localhost, ovviamente, sono più bassi, addirittura tra 0.05 e 0.07 ms.

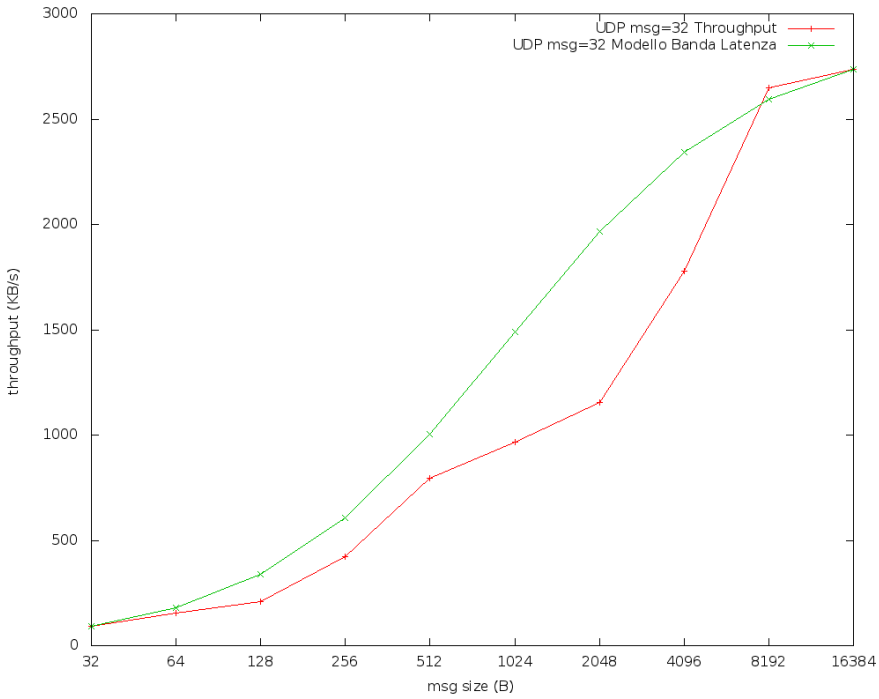




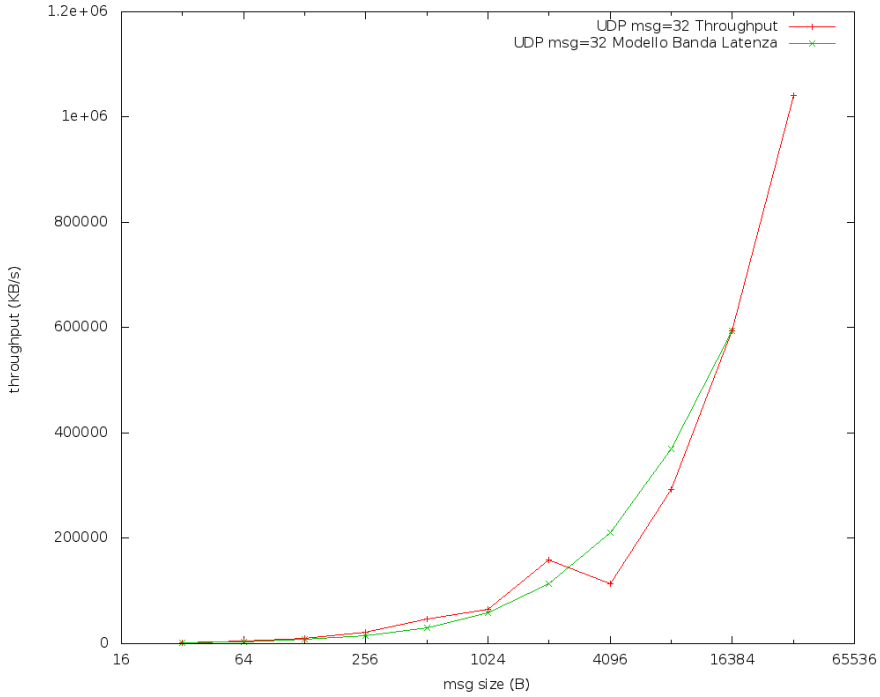
messaggi grandi TCP

Come sopra a sinistra comunicazione dai laboratori, a destra su localhost, per confrontare il comportamento del protocollo TCP con grandezza di messaggi alta (262144).
A sinistra si nota come l'andamento dei valori reali di RTT vada a scendere man mano che le ripetizioni aumentano, portando giù con se i valori stimati ed accrescendo quindi la variability. Non si sa a cosa sia dovuto, magari ad un fattore di congestione della rete, ma di certo in locale non c'è tutta questa variabilità. Infatti la maggior parte dei valori si aggira tra gli 8 e 10 ms, esclusi alcuni picchi.

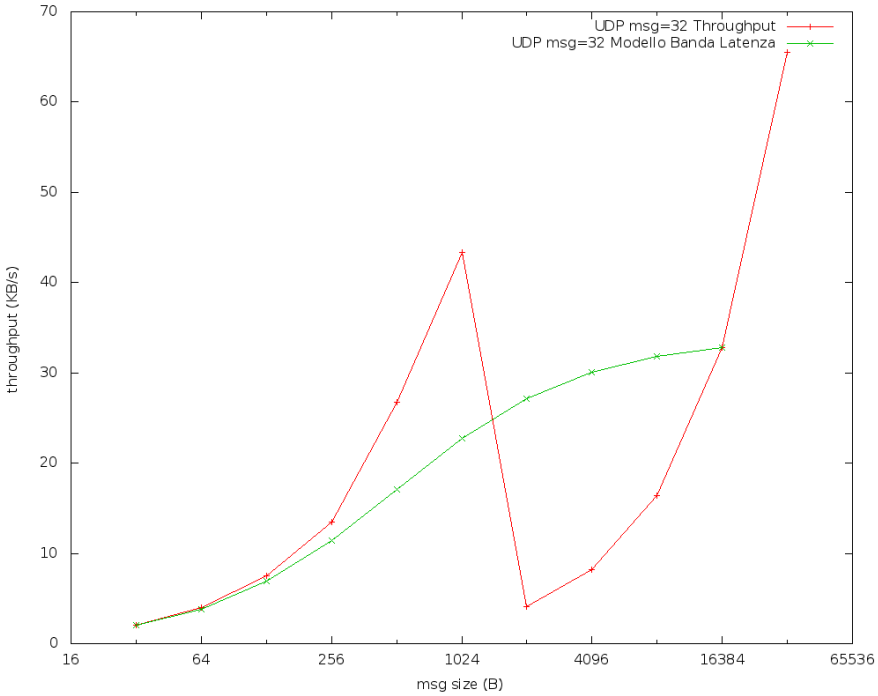
Il througput misura la quantità di dati trasferiti da un punto ad un altro nell'unità di tempo.



Queste sono le due curve di throughput utilizzando udp, quando ci siamo messi in comunicazione con Felix dai laboratori del Disi. Il modello Banda Latenza in verde è una curva 'regolare', mentre la curva rossa di traffico effettivo assomiglia più ad una linea spezzata che rimane sempre al di sotto della stima.



Quest'altra invece rappresenta le curve ottenute in localhost e si può notare come il modello BL abbia una curva diversa da quella precedente.

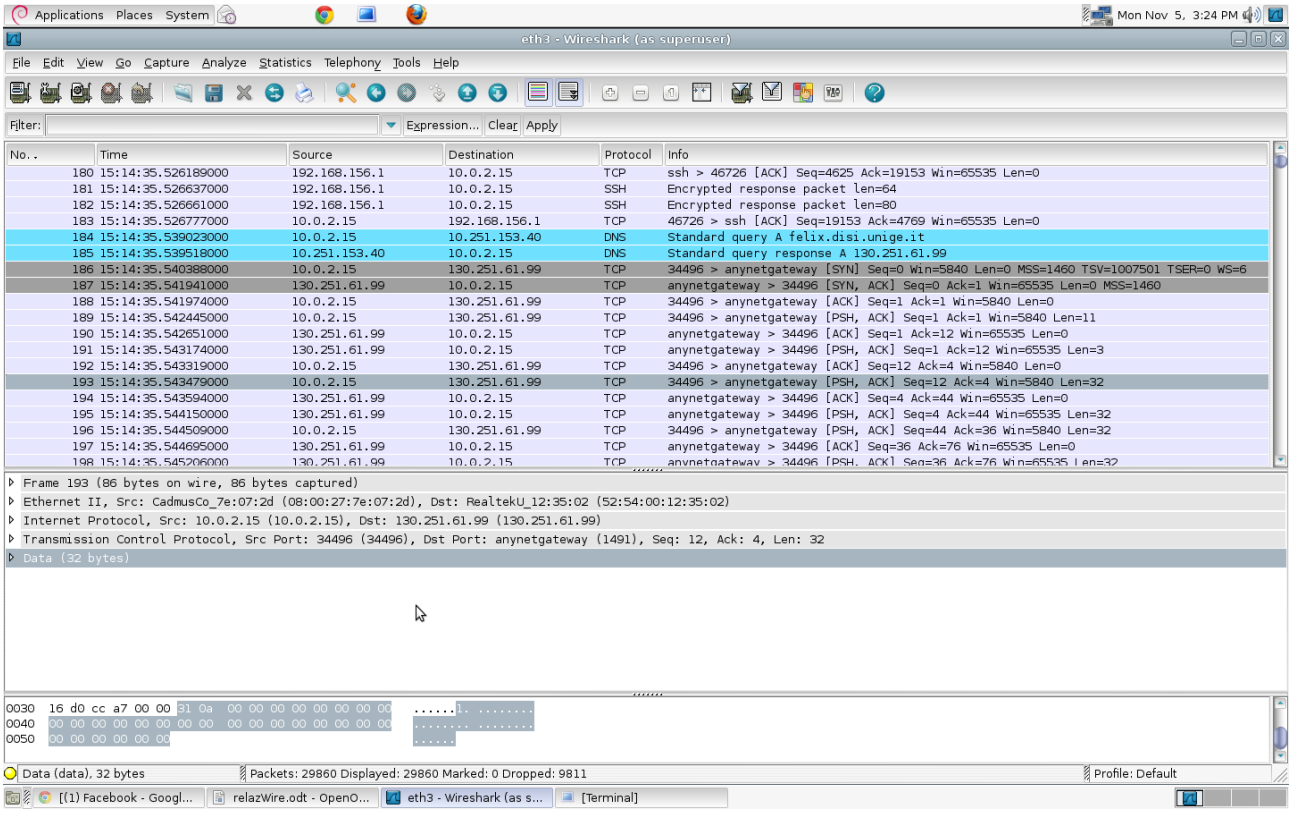


La comunicazione da LaSpezia verso Felix è quella che ha dato più 'scosse' alla nostra curva di throughput reale e quella con valori molto minori rispetto agli altri due grafici.

QUARTA ESERCITAZIONE: WIRESHARK

Wireshark è un software analizzatore di protocollo o “packet sniffer” in grado di analizzare il contenuto di tutti i pacchetti dati in transito sull’interfaccia di rete attiva, fornendo una panoramica dettagliata di tutto ciò che sta accadendo. Wireshark è in grado di individuare i protocolli di rete utilizzati per i vari tipi di comunicazione, il tempo trascorso dall’avvio alla cattura, l'IP sorgente e l'IP destinazione, oltre ad informazioni sui pacchetti stessi.

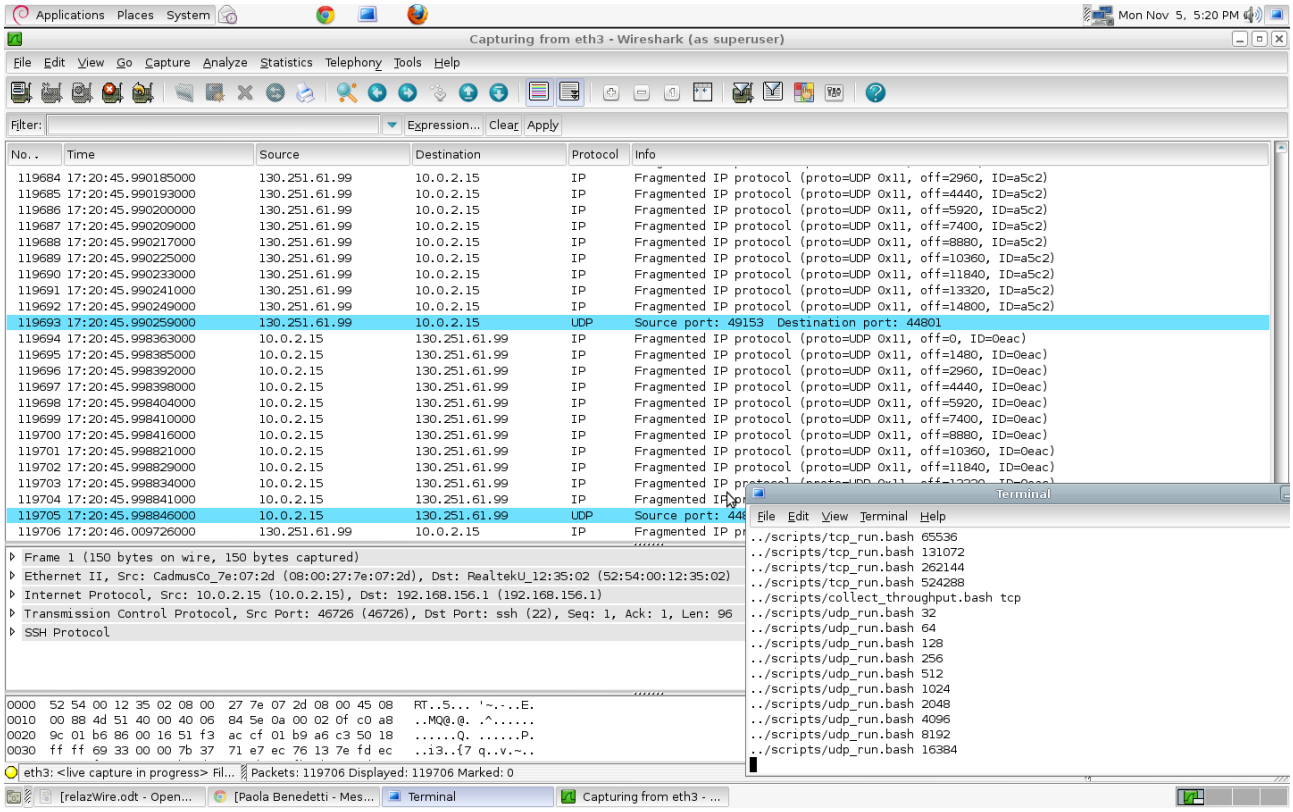
A noi interessa il traffico generato dai nostri client TCP e UDP verso il server Felix, che ha indirizzo ip= 130.251.61.99, mentre noi saremo 10.0.2.15 (macchina della rete privata nei lab).
Abbiamo settato il Time Display Format al Time of the day, nanoseconds, per poter rendere la quantità di tempo trascorsa tra un invio e l'altro.
Nella cartella abbiamo inserito gli screenshot incollati di seguito per motivi di leggibilità.



Nelle due righe in azzurro (No.184/185) e' riportata la risoluzione del nome logico di felix nel suo indirizzo ip dovuta alla prima richiesta fatta nel nostro makefile, cioe' ./tcp_ping felix.disi.unige.it 1491 32. Per ogni successiva chiamata, con il valore della grandezza del messaggio raddoppiata(64, 128, etc..), ci saranno altra righe azzurre.
La riga successiva (No.186), corrispondente alla seq.0, e' un messaggio di SYN dalla nostra macchina (source port 34496) verso Felix (anynetgateway 1491). Subito dopo(No.187) abbiamo la risposta di Felix, che e' un messaggio SYN ACK dove, tra le righe possiamo leggere:
[SEQ/ACK analysis]
  [This is an ACK to the segment in frame: 186]
  [The RTT to ACK the segment was: 0.001553000 seconds]

Fin'ora i messaggi avevano len=0. Il primo pacchetto spedito si trova al No.193, dove si puo' leggere Data (32 byte). Si notano inoltre i numeri di 'Seq=' e 'Ack=' inviati da noi e da Felix. Per esempio alla riga No.193 stiamo dicendo a Felix: ti inviamo la seq che comincia con 12 e ci aspettiamo da te quella che comincia con 4. Alla riga No.194 Felix dice: ti invio la seq che comincia con 4 e mi aspetto quella che comincia con 44 (12+32) e cosi' via.
Il protocollo TCP, dopo l'invio di un messaggio, attende una risposta; questo forma un"’alternanza” visibile tra i nostri messaggi e quelli di Felix, causando ovviamente aumento di tempo. E l'aumentare della grandezza dei messaggi di certo non aiuta, sia per i tempi che per la sicurezza dell'arrivo del messaggio, tanto che si notano linee rosse, che indicano la perdita del messaggio.

Invece per l'invio dei messaggi UDP, i tempi sono minori perché non ci si aspetta la risposta, non ci si accerta della riuscita della ricezione del messaggio.



La riga No.119778 (che nella figura successiva non e' più azzurra perché evidenziata per mostrare i dettagli) e' un nostro messaggio di grandezza 16384 byte. Nella finestra Internet Protocol, oltre agli indirizzi di mittente e destinatario, time to live, checksum etc, si vedono i frammenti in cui e' stato diviso il nostro messaggio per essere spedito, per ognuno la grandezza e il No. Relativo.
Si può notare che la differenza sui tempi e' di circa 6000ns, tempo che il TCP non raggiunge neanche per il messaggio con minor numero di byte.