*Luca Comparini S4184885*

LABORATORIO 1: Ping-Pong

In questo laboratorio l’attenzione è posta sulle dinamiche primitive delle comunicazioni tramite protocolli TCP e UDP; nonché sull’uso e sullo sviluppo di script bash in grado di produrre risultati dai dati ottenuti empiricamente.

Inizierò con un’illustrazione del codice prodotto esaminando ogni cartella e file messo a disposizione per questo laboratorio.

**tcp\_ping/tcp\_ping.c**

Questo è il file sorgente responsabile della comunicazione tramite TCP: protocollo, connection-oriented a pacchetto, di livello di trasporto che crea una connessione dove possono esser inviati bidirezionalmente numeri arbitrari di byte(stream) consegnati in “ordine”. Questo protocollo prevede una fase preliminare detta 3-way-handshake basata sui valori SEQUENCE e ACKNOWLEDGEMENT mirata a rendere il protocollo affidabile (non è possibile smarrire dati senza che server o client se ne accorgano) inoltre il messaggio può arrivare in modo non ordinato ma potrà sempre essere ricostruito nella forma originaria e ordinata.

* do\_ping

Inizialmente si scrive il message sequence number nel buffer di invio, dopodiché, in questa funzione vengono raccolti i tempi di invio/ricezione dei messaggi per il calcolo del RTT(Round-Trip Time) tramite la funz. clock\_gettime(CLOCK\_BOOTTIME, ). La scelta è ricaduta su CLOCK\_BOOTTIME siccome non è affetto da salti discontinui nel tempo del sistema e comprende il tempo in cui il sistema è sospeso.

Appare in questa funzione la prima primitiva: send() usata per mandare dati a un socket, e adoperata prevalentemente con socket connessi TCP mentre sendto() è più ragionevolmente usata con socket non connessi di tipo UDP.

* main

Il primo passo nell’uso dei socket è la definizione degli hints necessari per specificare le opzioni e il tipo di socket. Viene quindi completata la struct addrinfo gai\_hints dopodiché viene passata alla funzione getaddrinfo() insieme a un hostname, ad un port number ed ad una struct addrinfo “destinazione”. Getaddrinfo() compila la struct “destinazione” in modo che sia utilizzabile per successive funzione come la connect; getaddrinfo() combina le funzionalità offerte da gethostbyname() e getservbyname().

Quindi si procede alla creazione di un socket appropriato passando alla funzione socket i vari campi della struct “destinazione” (eventualmente anche tramite la struttura gai\_hints). Quindi si connette il socket con le “coordinate di rete” tradotte da getaddrinfo nella struct “destinazione”. Per scrivere la richiesta nel socket utilizzo la funzione nonblocking\_write\_all() definita nel directory pingpong/pingpong\_lib/readwrite.c che oltre alle funzionalità della write() implementa un controllo sul numero di byte effettivamente scritti prendendo provvedimenti a riguardo. Infine vi è un controllo sulla risposta letta dal socket attraverso una semplice strncmp.

**udp\_ping/udp\_ping.c**

File sorgente della comunicazione UDP, protocollo a livello di trasporto, connectionless e a pacchetto. Questo protocollo a differenza del TCP non garantisce ne il riordinamento dei pacchetti nell’ordine corretto ne la ritrasmissione dei dati persi, pertanto non è affidabile. Tuttavia la mancanza di 3-way-handshake e dei controlli presenti nel TCP avvantaggiano la velocità di questo protocollo.

* do\_ping

In questa parte del laboratorio ho riscontrato delle difficoltà derivanti dalla scarsa affidabilità del protocollo UDP; in particolare all’aumentare della dimensione dei messaggi (24576 e 32768) la comunicazione non arrivava mai a una conclusione totale.

* prepare\_udp\_socket

Prepara il socket UDP con i parametri appropriati, inoltre compare la funzione fcntl() necessaria per cambiare il tipo di socket in non bloccante. Se il socket rimanesse in modalità bloccante molte delle funzione interessate nel utilizzo del socket come send(), recv(), connect(), accept() etc non andrebbero a restituire il controllo al programma fintanto che l’operazione non è completata.

**pong\_server/pong\_server.c**

Purtroppo non mi è stato possibile, per mancanza di tempo, completare questa parte opzionale del laboratorio. Ho fatto diversi tentativi ma non sono riuscito a identificare quale sia l’errore che sembra generarsi dalla funzione bind(). Teoricamente mi sembra di aver “ordinato” in modo corretto i passaggi del lato server ma l’implementazione è incorretta.

**pingpong\_lib/pingpong.h**  aggiunta la libreria poll.h per la funzione poll().

**pingpong\_lib/readwrite.c**

Completo la funzione nonblocking\_write\_all con un controllo su errno, in modo da identificare i segnali EWOULDBLOCK ed EAGAIN e nel caso far comunque continuare i tentativi di write. Vorrei aggiungere che inizialmente avevo implementato un timeout tramite la funzione poll fino a quando il file descriptoro diventa ready; tuttavia ero incerto sulla sensatezza di questa implementazione e quindi ho deciso di tralasciarla.

**pingpong\_lib/statistics.c**

* timespec\_delta2milliseconds

Questa è una funzione per il calcolo del delta fra due struct timespec.

* print\_statistics

Richiedeva la stampa di statistiche per l’istogramma.

**scripts/print\_average.bash**

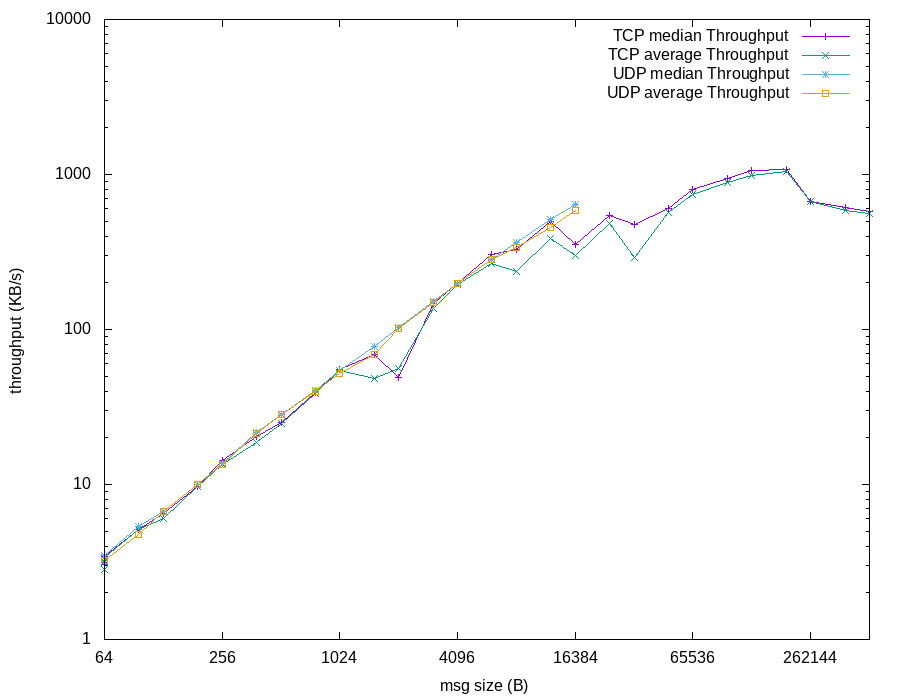
Questo è lo script responsabile dell’estrazione dei dati prodotti e registrati nei file \*.dat nonché della produzione tramite gnuplot dei grafici banda-latenza. Prende in argomento il nome di un protocollo (UDP/TCP) e tramite i comandi head e tail estrae la prima e l’ultima riga del corrispondente \*.dat; dopodiché salva le due righe in due array, procedendo all’estrazione dei throughput medio e mediano. Vengono quindi calcolati i valori di delay, banda e latenza tramite le formule messe a disposizione su aulaweb ed illustrate a lezione; questi calcoli vengono operati con la calcolatrice bc impostata ad una precisione di 10-9. Infine viene passata una funzione rappresentativa di banda-latenza a gnuplot impostando scala logaritmica e le varie etichette del grafico; in aggiunta ho sistemato i range di rappresentazione degli assi a seconda di quanto fosse opportuno per avere una visualizzazione più precisa.

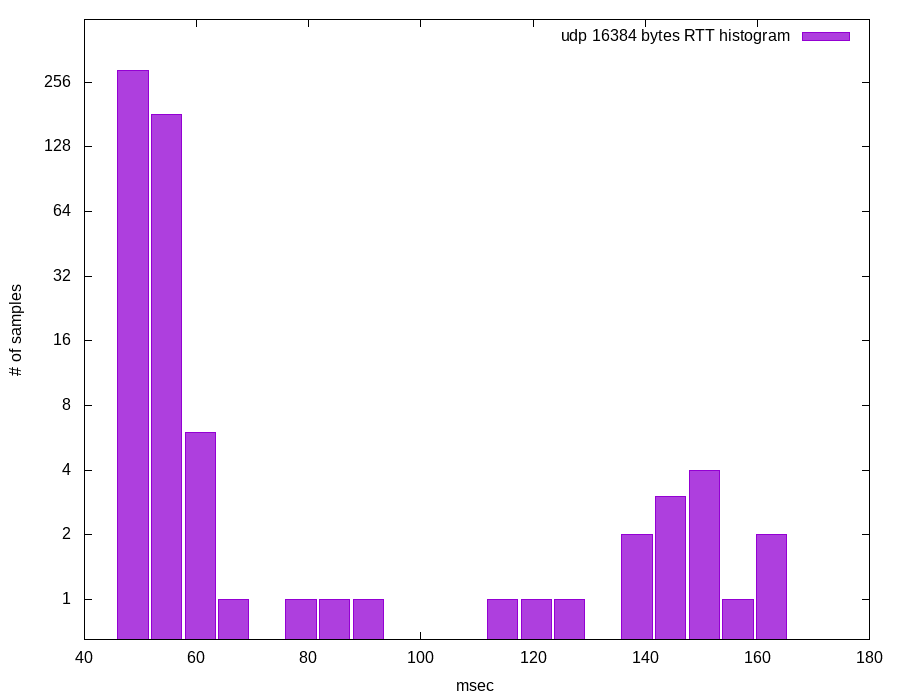
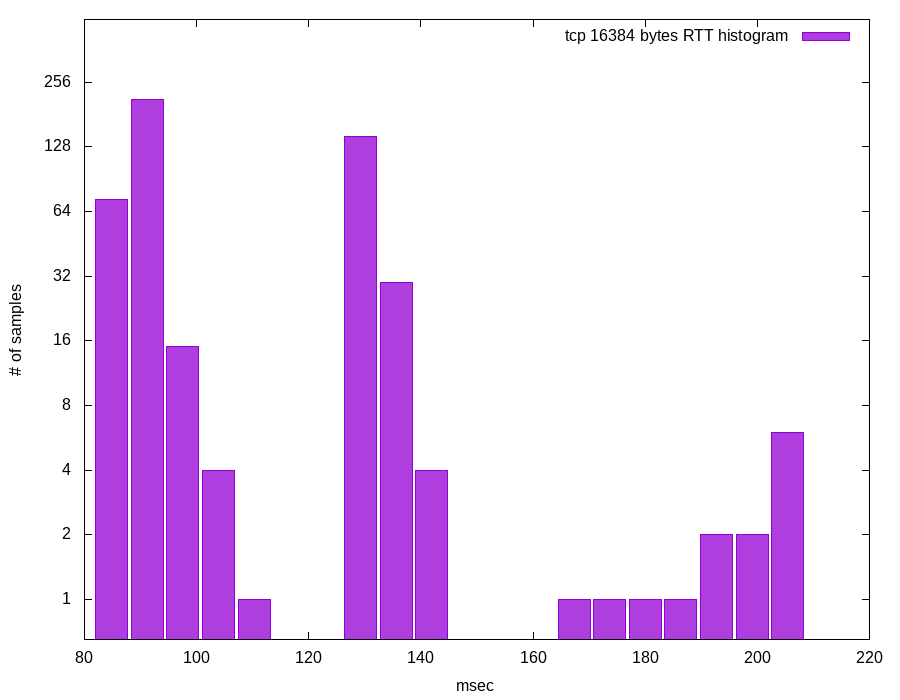
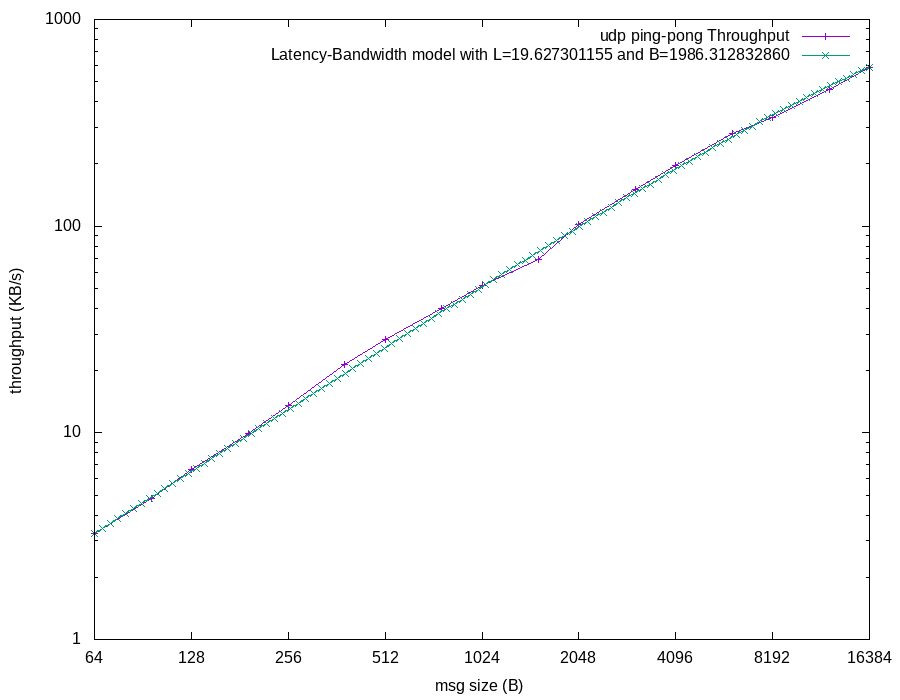
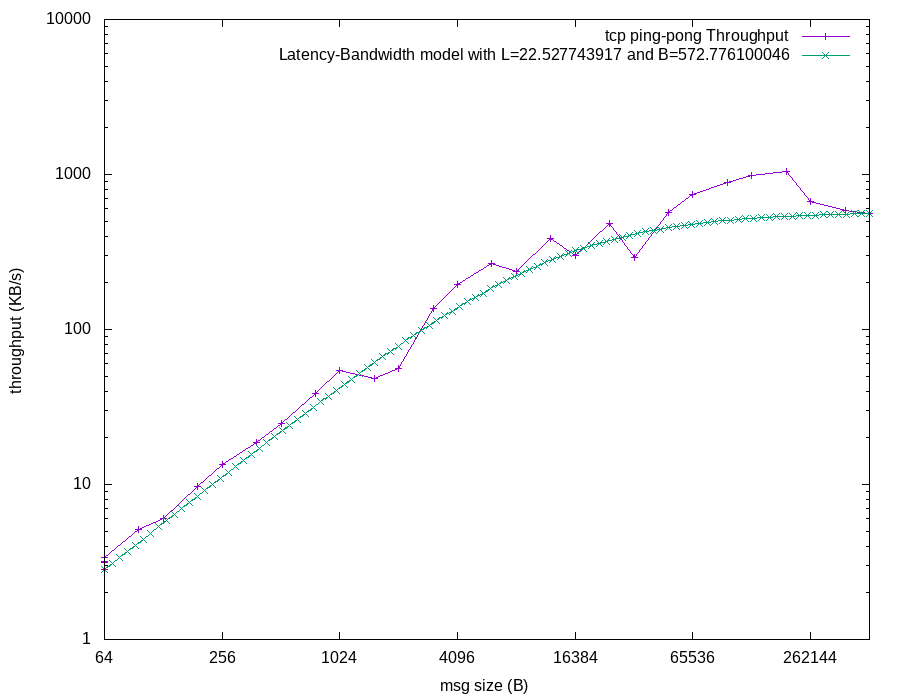
N.B: Per problemi di differente rappresentazione dei file di testo a seconda del S.O. o dell’hardware interessato ho dovuto modificare il file collec\_throughput.bash alla riga 15 in modo da ottenere il corretto funzionamento degli script sulla mia “macchina”.

**RISULTATI GRAFICI**

A causa dei problemi di affidabilità del protocollo UDP descritti precedentemente ho portare a termine la sperimentazione anche tramite due virtual machine (attive sullo stesso computer fisico) messe in comunicazione fra di loro per eliminare la possibilità di avere perdite di pacchetti (i dati non saranno tuttavia realistici).

GRAFICI RELATIVI ALLA COMUNICAZIONE CON SERVER DIBRIS





GRAFICI RELATIVI ALLA COMUNICAZIONE TRA DUE VIRTUAL MACHINE