RELIABLE UDP

A CIRCULAR BUFFER ADAPTED ON SELECTIVE REPETE PROTOCOL IMPLEMENTATION

Andrea Di Iorio | matricola: 0232032 | Corso: Ingegneria di Internet A.A. 2017/18

CONTENUTI RELAZIONE

Le prossime pagine vanno a comporre la relazione del progetto "trasmissione affidabile con UDP " per la parte pratica dell'esame di Ingegneria di Internet, a.a. 2017/18

Per lo sviluppo ho utilizzato Clion e Vim, supportati da una compilazione semplice basata su makefile e l 'uso gdb per il debugging.

SCELTE PROGETTUALI

1) Ai fini della concorrenza nell'applicazione da entrambi i lati (client e server), date le caratteristiche di non affidabilità delle socket UDP, come i problemi di bufferoverflow, ho preso le seguenti due scelte
    1. Sono stati impiegati diversi processi per gestire le singole trasmissioni e all'interno del server almeno un processo per ogni client connesso cosi da gestire le trasmissioni/ricezioni in modo vicendevolmente indipendente.
    2. Al fine di poter gestire diversi flussi di dati, relativamente allo scambio di file, tra i clients e il server, sono create socket UDP per ogni client che si connette al server e per ogni operazione del client.

2) Per le operazioni di invio e ricezione di un file ho impiegato un approccio basato su buffer circolare acceduto concorrente da più thread con semantica del tipo SPSC adattata alle caratteristiche e vincoli del protocollo Selective Repete, così da usare una struttura dati che occupa una quantità di memoria costante che supporta una logica concorrenziale.
3) Al fine di facilitare il mapping tra Pacchetto e elemento di buffer circolare lo spazio di numerazione impiegato nel esecuzione del protocollo è esattamente il doppio della dimensione del ring buffer dove sono bufferizzati i pacchetti.
4) Nel tentativo di incrementare le performance ho deciso di rendere parallele le operazioni di lettura e scrittura da socket su thread differenti nella logica di invio di un file.
5) Nella trasmissione dei pacchetti di un file, al fine di realizzare la semantica di ritrasmissione basata su timer del protocollo Selective Repete ho impiegato un approccio simile allo *scheme 1* della soluzione proposta nell'articolo: *Hashed and Hierarchical Timing Wheels: Efficient Data Structures for implementing a Timer Facility di George Varghese and Anthony Lauck da IEEE/ACM TRANSACTIONS ON NETWORKING, VOL. 5, NO. 6, DECEMBER 1997*, ottimizzata con un piccolo trucco nel controllo dei timer così da ridurre il potenziale O(n) nel controllo dei timers.
6) Al fine di mantenere modularità all'interno della applicazione, ho dato una separazione netta tra le logiche di scambio messaggi client-server e di trasmissione/ricezione di file e di gestione dei timer, seguendo un approccio assimilabile a Boundary Control Entity dell'Ingegneria del Software.
7) Per la creazione di una connessione tra un nuovo client e il server ho seguito un approccio basato sullo scambio di 3 messaggi (similmente a 3-HandShake)

ARCHITETTURA DELLA APPLICAZIONE

CONCORRENZA NELL'APPLICAZIONE

Come accennato nella sezione precedente l'applicazione è strutturata su diversi processi per gestire singole comunicazioni di file o comunicazione di messaggi di controllo e risposta, e thread all'interno dei processi adibiti a operazione per l'invio e ricezione.

Processi

Segue una rappresentazione schematica dell'architettura dei processi all'interno dell'applicazione realizzata utilizzando un tool basato su UML e con lo scopo di dare una rappresentazione di alto livello dei processi nell'app.
- i rettangoli rappresentano i processi attivi
- la sezione inferiore dei rettangoli contiene le variabili più importanti in possesso del processo
- le dipendenze (frecce tratteggiate) rappresentano le relazioni tra processi remoti

- le associazioni (frecce continue) rappresentano relazioni padre-figlio tra processi

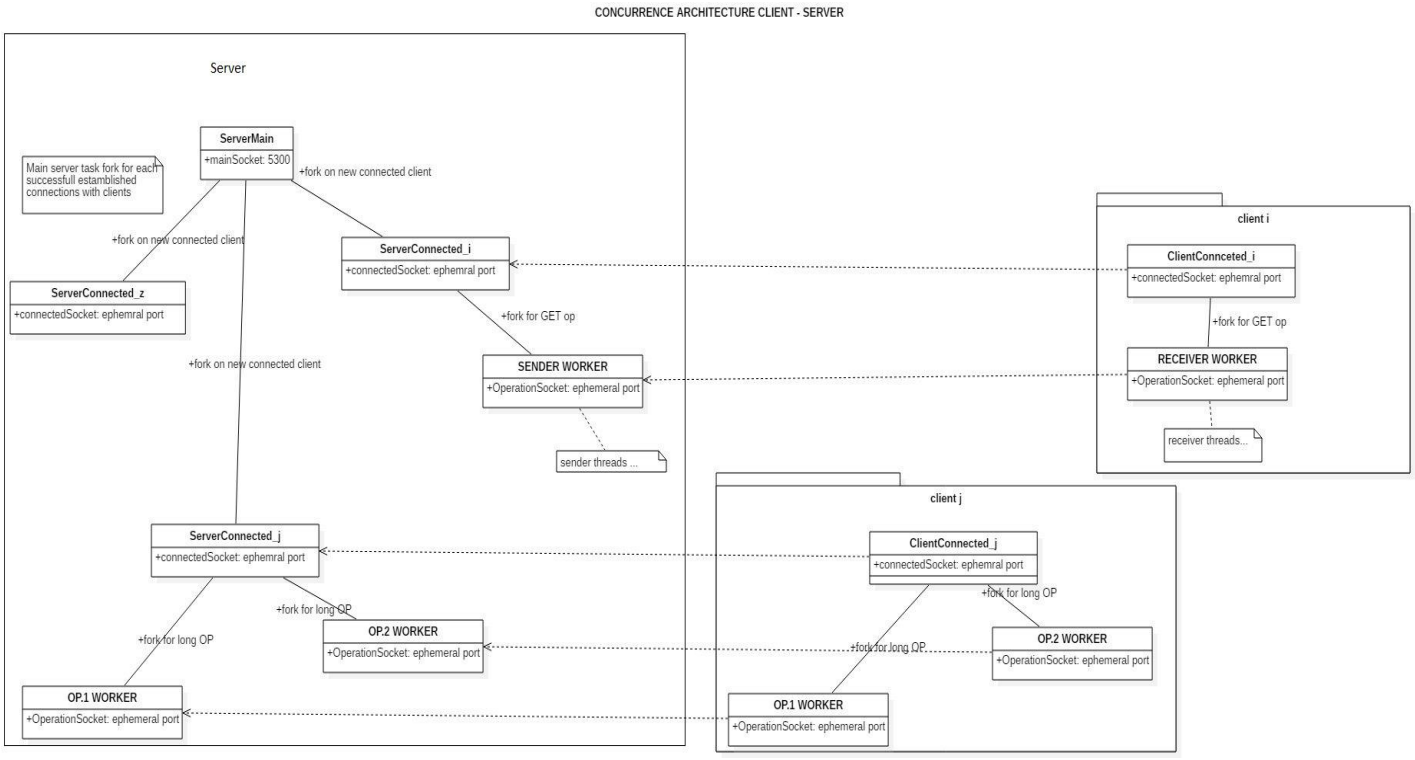

Figura 1

1. LATO SERVER

Il Server, in seguito alla ricezione di una richiesta di connessione da un client, esegue un'associazione basata sullo scambio di 3 messaggi e crea un nuovo processo con una nuova socket associata ad una porta casuale disponibile che viene "connessa" con l'indirizzo di una socket del client. Questa socket "connessa" (nel diagramma e nel codice chiamata "*connectedSocket*") sarà impiegata per lo scambio di richieste dal client al server e messaggi relativi a una richiesta di LIST (operazione non lunga)

Qui il nuovo processo creato si trova nella logica di "Boundary" del server e dopo avere ricevuto i parametri trasmissivi (struttura *tx_config* in app.h) per le sue future trasmissioni di file attende messaggi controllo da parte del client relativi a richieste di operazioni (LIST, GET, PUT).

Per ogni operazione di lunga durata (invio/ricezione di un file) il server creerà un nuovo processo adibito alla singola operazione, dove dividerà i compiti su più thread e notificherà l'esito della operazione sia alla UI e sia al processo padre mediante valore di uscita.

Al fine di mantenere su ogni socket un singolo flusso di dati indipendente, verrà creata e connessa una nuova socket per ognuna di tali operazioni di lunga durata.

2. LATO CLIENT

Il Client dopo una fase di connessione al server, costituita dallo scambio 3 messaggi mediante la socket principale del server (associata alla porta 5300) si trova nella logica "Boundary" del client (file *connectedClient.c*) con una socket "connessa" all'indirizzo della socket di un nuovo processo del server (la quale nel diagramma e nel codice è denominata *connectedSocket*).

Qui vengono prese da UI, i principali parametri per tutte le future trasmissioni (dimensione della finestra di trasmissione e probabilità di simulazione di perdita dei pacchetti) e, successivamente a

questa inizializzazione, i codici dei messaggi controllo relative alle richieste del client, che verranno spedite al server.

Similmente al server, per ogni operazione di invio/ricezione di file viene creato un nuovo processo e una nuova socket "connessa", al fine di mantenere client e server capaci di gestire più operazioni simultaneamente.

- **Creazione e connessione di una socket**
  1. **Connessione iniziale Client <-> Server**
     La connessione iniziale tra un client e nuovo processo del server è realizzata mediante lo scambio 3 messaggi:
     1) Il Client invia un messaggio di richiesta di connessione all' indirizzo noto del server (porta 5300)
     2) Il Server risponde al Client con un valore casuale=x mediante una nuova socket associata ad una nuova porta (risultato di una connect() di una nuova socket all'indirizzo del client estratto dal messaggio di richiesta inziale) che sarà impiegata per servire esclusivamente il Client
     3) Il Client connette la propria socket alla nuova socket del server estraendone il nuovo indirizzo(in particolare la nuova porta) dal messaggio 2) ricevuto, e rinvia lo stesso valore casuale x alla nuova socket del server.
  2. **Connessione relativa a una operazione di lunga durata**
     Per ogni operazione potenzialmente di lunga durata (GET e PUT) nel client viene creata una nuova socket (*opSocket* nel diagramma e nel codice) e associata a una porta casuale disponibile (tramite wildcard binding), questo nuovo indirizzo viene comunicato al server, il quale vi associerà una nuova socket per la trasmissione e invierà una conferma di avvenuta connessione al client.

- **Messaggi scambiati**

Segue una rappresentazione schematica dei messaggi scambiati in seguito alle operazioni di GET e PUT rappresentando i processi coinvolti.

La rappresentazione è effettuata utilizzando i sequence diagram di UML.

Le linee di vita rappresentano i processi, i messaggi continui sono assimilabili a messaggi controllo dal client al server, i messaggi tratteggiati sono assimilabili a messaggi risposta del server.

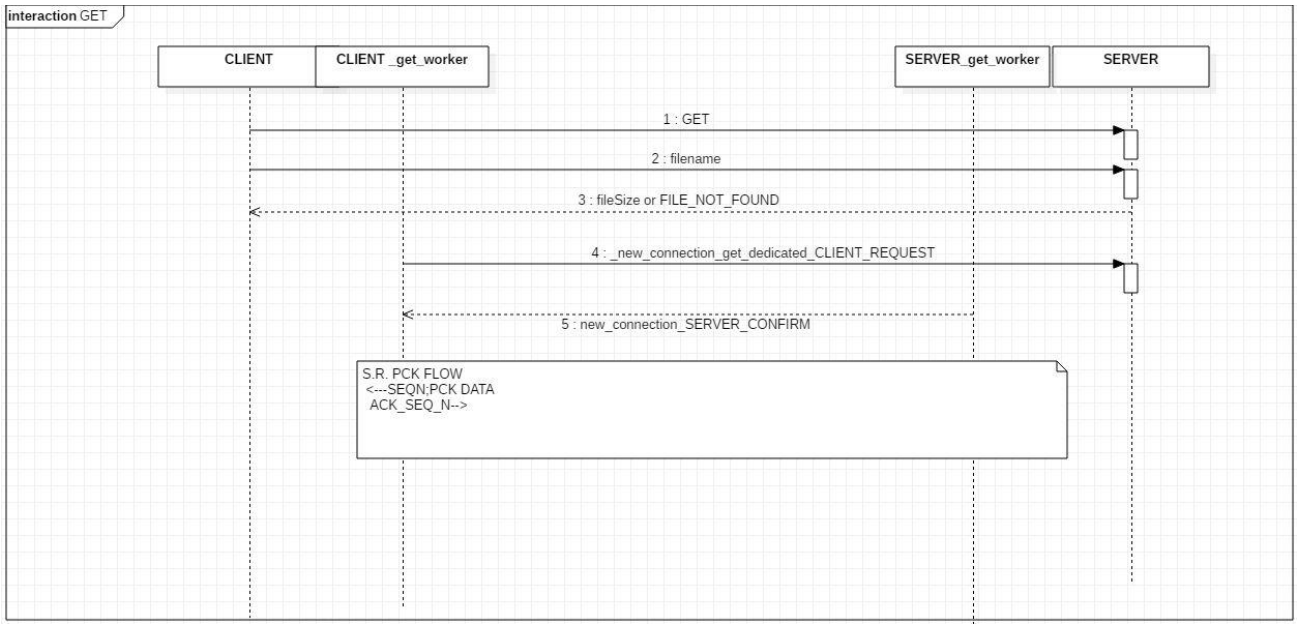

Figura 2

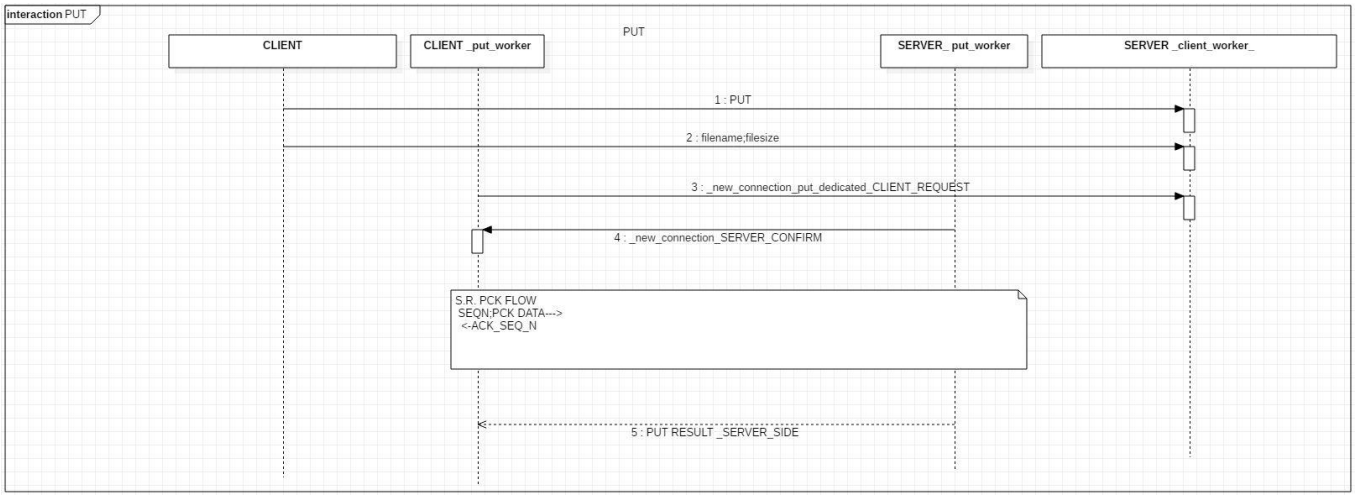

Figura 3

- **<u>Terminazione e propagazione tra processi</u>**

All'interno dell'applicazione la terminazione può essere necessaria in momenti differenti.

- Per ogni operazione bloccante legata a una trasmissione di file è associato un timeout (di durata configurabile tramite MACRO TIMEOUT_CONNECTION in app.h) al termine del quale verrà inoltrato un SIGALRM al processo bloccato causandone la terminazione.
- Nei processi di un livello superiore all'ultimo, una richiesta terminazione o un fallimento causa la propagazione della stessa anche nei processi figli prima di uscire. Questo al fine di evitare l'insorgenza di processi zombi e gestire correttamente la de allocazione delle risorse. La propagazione della terminazione è effettuata mediante l'invio di SIGUSR1 ai processi figli.
- Nei processi principali del client e del server è installato mediante *sigaction* un handler del segnale SIGCHLD al fine di evitare l'insorgenza di processi zombi senza troppe chiamate di wait().

STRUTTURAZIONE DELLA LOGICA APPLICATIVA

Il flusso di esecuzione dell'applicazione segue un percorso simile tra i lati Client e Server. In particolare, nella logica di "*Boundary*" dopo una fase di inizializzazione della connessione (files clientStart.c serverStart.c) si passa alla gestione delle principali interazioni tra client e server (files clientConnected.c e serverConnected.c) .

In seguito a una richiesta di un Client, se ricevuta ed accettata dal server, si passa a una logica di "*Controller* " dell'app dove viene eseguita la richiesta, in un nuovo processo indipendente.

In particolare:

-in seguito a una richiesta di PUT, il client esegue la logica del SENDER mentre il server la logica di RECEIVER

-in seguito a una richiesta di GET, viceversa, il client esegue la logica di RECEIVER e il server la logica di SENDER

IMPLEMENTAZIONE

- **Invio e ricezione di file**

Le operazioni di trasmissione di file sono gestite differentemente nei file *senderScoreboard.c, receiverScoreboard.c* e **le logiche per tali operazioni verranno indicate rispettivamente con i termini *SENDER* e *RECEIVER***.

Similitudini principali tra SENDER e RECEIVER

-Le principali strutture dati di supporto comuni sono il buffer circolare e i pacchetti (definiti in app.h *struct circularbuf* e *struct pck*).

Essenzialmente la struttura del **ringBuffer** è costituita da indici per il produttore/consumatore ed un puntatore a un array di pacchetti denominato ringBuf. Quest'array è costituito da strutture pck, le quali racchiudono le informazioni principali relative ad un pacchetto, dato l'ampio utilizzo di queste è stata definita una macro in app.h *Pck* per la dichiarare una struttura pck più rapidamente.

```
struct pck {
   struct pck* nextPck;                    //point to next pck in cbuf
   int seqNum;
   int pck_index;                          //index  pck in ringbuf (not seqN)
   volatile char flag_internal;            //state of the pck in ringBuf
   char data[PCKPAYLOADSIZE];
#ifdef RETRASMISSION
   struct timer_pck timerPck;          //related timer only if defined MACRO
#endif
};
#define Pck struct pck
```

Nella **struttura pck** sono definiti: un numero di sequenza (in accordo con la semantica del protocollo S.R.), un flag gestito da produttore e consumatore per indicare lo stato del pacchetto, una struttura rappresentativa del timer associato all'interno di una condizione MACRO ed un puntatore al pacchetto successivo nel ring buffer (i quali sono linkati ad anello).

Un possibile impiego di quest'ultima condizione è un client compilato senza tale macro, che potrà solo ricevere file da server, ma risparmierà spazio allocato nel ringbuffer.

-È di utilizzo comune nel SENDER e nel RECEIVER la seguente macro:

```
//set inRangeBool with true if n is in [lowBoundary,highBoundary] MOD circularSize...
#define N_IN_RANGE_MOD_N(lowBoundary,highBoundary,circularSize,n,inRangeBool)\
   if ((lowBoundary) <= (highBoundary)) {              /*range contigue*/ \
      (inRangeBool) = N_IN_RANGE(lowBoundary, highBoundary, n); \
   } else  /*lowst>highest => disjoint range*/ \
   { (inRangeBool) = N_IN_RANGE(lowBoundary, (circularSize)-1, n)|| N_IN_RANGE(0, highBoundary, n);}
```

La quale ritorna vero se un indice cade in tra due estremi del buffer circolare ed è utilizzata essenzialmente per :

-valutare se un pacchetto o un ack è duplicato

-nella logica di spinlock del produttore nel RECEIVER.

-In entrambi i casi di RECEIVER e SENDER è definita una struttura denominata **scoreboard** (con suffisso Sender o Receiver), al cui interno sono presenti oltre a un puntatore al buffer circolare anche altre variabili utili per la trasmissione come file descriptors associati a file e alla socket in uso, costanti, variabili per l'istanzazione ed esecuzione dei thread.

```
struct scoreboardReceiver{
    ////basic logic 4 S.R.
    struct circularBuf* cbuf;
    int maxseqN;
    int winSize;                        //rcv win size
    volatile Pck* rcvbase;              //first missing pck in receive window
    ////io-&-net
    char filename[MAX_FILENAME_SIZE] ;
    unsigned long int fileSize;
    int fileOut_BlockSize;              //max ammount of pck writable from cbuf to file
    unsigned long int fileByteWritten;  //byte writed o
    unsigned long int fileByteReceived;
    int socket;
    int fd;
    char pckTmpBuffer[SERIALIZED_SIZE_PCK];//tmp buffer to receive 1 pck
    ////thread workers
    pthread_t producer;                 //pck_receiver_th;
    pthread_t consumer;                 //file_filler_th;
    pthread_t ackker;                   //ack_sender_th;
};
```

```
struct scoreboardSender{
    struct circularBuf* cbuf;           //circular buffer &co
    volatile Pck* sendBase;             //first pck unackked in send window
    int maxSeqN;                        //max seqN usable in pck header
    int winsize;                        //sending window size...
    ///IO
    char filename[MAX_FILENAME_SIZE] ;
    int fd;
    unsigned int fileIn_BlockSize;       //max # pck readable in cbuf
    unsigned long fileSize;
    int sockfd;
    ///timers
    struct timers_scheduled timersScheduled;
    ///SENDER threads...
    pthread_t  producer;
    pthread_t  consumer_sender;
    pthread_t  consumer_ackker_th;
    ///statistics
    int totPckSent,totPckLosed,totAckRcvd;
#ifdef FAKEPCKLOSS
    double PCK_LOSS_PROB;               //probability for pck loss for testing
#endif
};
```

-**L'esecuzione della logica di invio e ricezione di un file si accomuna nei primi 2 passi**, ovvero nella **allocazione dello Scoreboard** (funzioni initScoraboard...) e **l'esecuzione dei thread worker** (in startReceiver e startSender), queste rappresentano *gl'entry point* per la logica di Controller dell'app.

-Nella logica di produzione del SENDER e RECEIVER **è lasciato libero un pacchetto al fine di differenziare le situazioni di ring buffer pieno e vuoto**. Per tale ragione è necessario sempre che esista almeno un pacchetto extra nel ringbuffer, ulteriore rispetto alla dimensione della finestra di trasmissione(gestito dalla MACRO EXTRARINGBUFSPACE)

LOGICA SENDER

Segue una rappresentazione schematica dell'architettura dei threads per le operazioni di trasmissione e ricezione di file. Tale rappresentazione è stata creata utilizzando un tool basato su UML ed è finalizzata a dare una rappresentazione di alto livello.

In particolare:

- le classi (i rettangoli) rappresentano i thread,

-gli attributi delle classi (sezione intermedia dei rettangoli) rappresentano le variabili più significative che i thread gestiscono,

-le operazioni (ultima sezione dei rettangoli) rappresentano le principali funzioni chiamate dai thread

-le relazioni di dipendenza (le frecce tratteggiate) rappresentano le interazioni principali di un thread con altri, con sopra denominata la variabile utilizzata per lo spinlock

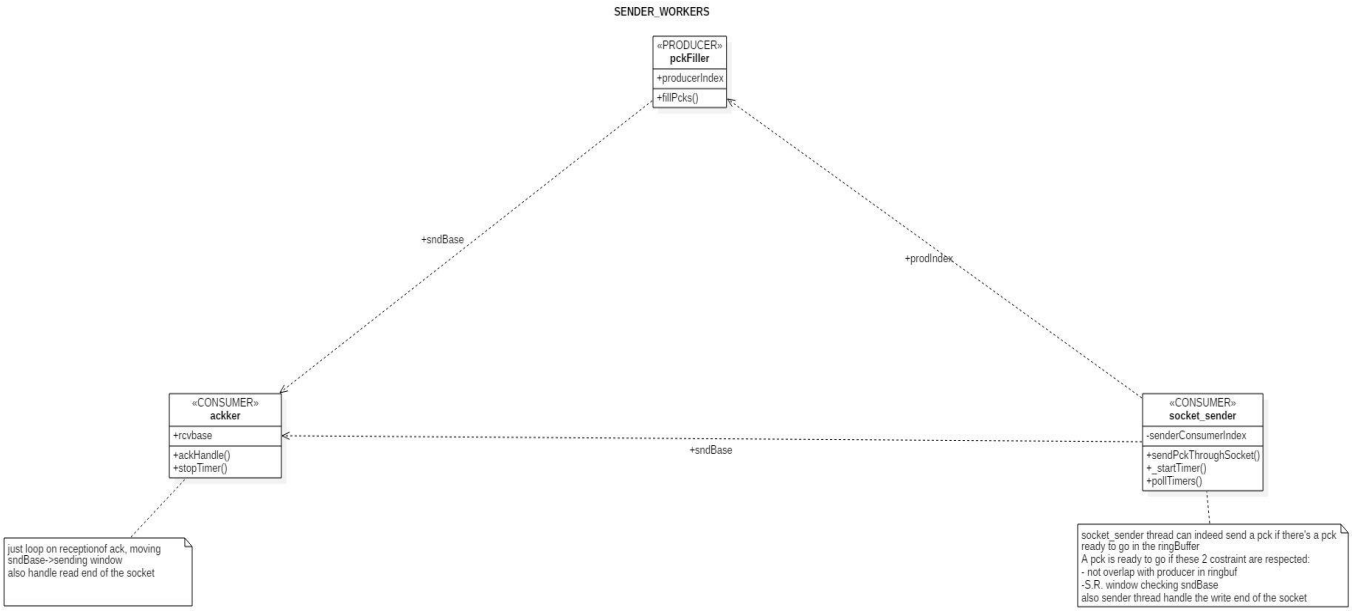

Figura 4

La logica relativa all'invio di un file è suddivisa tra 3 thread specializzati in ruoli differenti:

1) ***pckFiller***:

rappresenta il ***produttore*** nella semantica SPSC del ringbuffer ed essenzialmente legge dati da file, costruisce pacchetti, li copia sul buffer circolare e gestisce l'indice condiviso producerIndex.

Condizione spinlock Producer:

Prima di modificare il buffer circolare valuta la posizione che sndbase occupa nel buffer (nel protocollo S.R. il primo pacchetto non riscontrato nella finestra di invio)

- **Nel caso si trovi oltre la posizione immediatamente successiva all'indice del produttore significa che il thread può produrre nuovi pacchetti nel ringbuffer**, in tal caso valuterà in accordo con l'attuale posizione ricoperta da sndbase un **trade-off** tra il numero di pacchetti producibili nel ring buffer ed un valore massimo (definito dalla macro FILEIN_BLOCKSIZE).
- Viceversa, dovrà attendere fintanto che sndbase non si muoverà oltre, in tal caso **il ringbuffer è pieno.**

Infine il thread termina quando finisce di leggere il file, marcando un ultimo pacchetto nel ring buffer, al fine di notificare agl'altri thread la fine del file in invio. Questo pacchetto sarà trasmesso nella logica di terminazione della trasmissione.

2) ***socket_sender***:

rappresenta il ***consumatore***, essenzialmente invia al destinatario della socket connessa dei pacchetti **pronti**, imposta e controlla i timer associati ai pacchetti.

Questo thread **deve rispettare due vincoli**:

1) non deve spedire pacchetti in produzione, verificando l'indice del produttore

2)non eccedere la quantità di pacchetti non riscontrati inviati, verificando il valore relativo a sndbase (condizione imposta dalle regole del protocollo Selective Repete).

Condizione spinlock Consumer:

Sono possibili 2 condizioni di attesa per il thread, relative al ringbuffer vuoto o alla finestra di trasmissione piena.

Rispettati i due vincoli questo thread può inviare un nuovo pacchetto pronto del ring buffer utilizzando una variabile privata (nel codice *sender_consumerIndex*) per indicare a che punto l'invio dei pacchetti si trova nell' array.

Al fine di **simulare una perdita di pacchetti** anche sull'interfaccia di loopback, all'interno di una condizione MACRO (*FAKEPCKLOSS*) verrà valutata con una probabilità p configurabile se non spedire dei pacchetti, ma avviarne ugualmente i relativi timer, così da innescarne la ritrasmissione.

Periodicamente, verranno controllati i timers dei pacchetti inviati, rinviado un pacchetto se il relativo timer è scaduto.

3) ***ackker***:

legge da socket riscontri dei pacchetti precedentemente inviati, ferma i timer dei pacchetti relativi e gestisce il puntatore a sndBase.

Questo thread, dopo aver ricevuto un nuovo ack da socket (ovvero il numero di sequenza di un pacchetto precedentemente inviato), tramite la funzione *ackHandle*, valuterà se l'ack ricevuto è un duplicato e se comporta lo scorrimento della finestra di invio o meno (spostando il puntatore sndbase). Eventualmente stopperà il timer relativo al pacchetto identificato dall'ack .

In questo modo la lettura e scrittura da socket è concorrente tra i thread 2) e 3) e un riscontro di un pacchetto precedente può essere ricevuto anche durante l'invio, modificando la finestra di invio del protocollo.

Terminazione thread:

il thread 1) termina autonomamente una volta che ha prodotto l'ultimo pacchetto da file,

il thread 3) termina una volta ricevuto l'ack relativo all'ultimo pacchetto (marcato come ultimo dal produttore) e notifica di terminare a 2).

Una volta ricevuto l'ultimo l'ack infatti, nel RECEIVER è stato correttamente ricevuto l'intero file e la trasmissione può terminare con successo

LOGICA RECEIVER

Come sopra valgono le stesse indicazioni di lettura della seguente rappresentazione schematica.

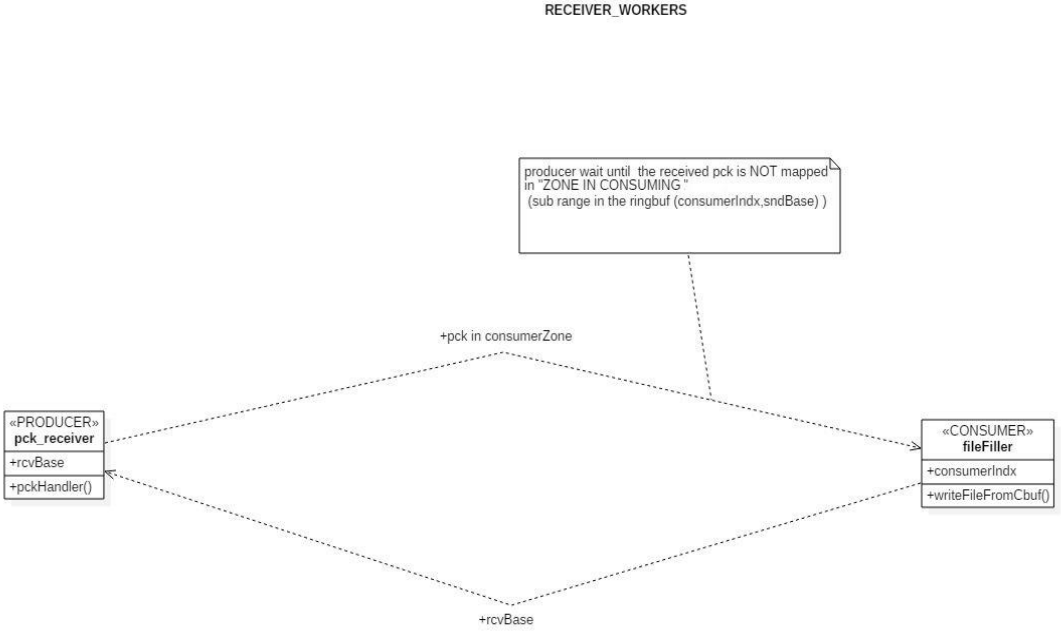

Figura 5

La logica del ricevitore è suddivisa in due thread

1. un **produttore** denomianto ***pck_receiver***, che riempie il buffer circolare con nuovi pacchetti ricevuti, scartando eventuali duplicati e gestendo il puntatore di rcvBase**.**

2. un **consumatore**, che legge pacchetti da buffer circolare e gli scrive su file, chiamato ***file_filler***.

2. Il consumatore semplicemente, prima di scrivere un numero di pacchetti consecutivi dal ring Buffer (in maniera analoga al thread produttore nel SENDER) valuta la posizione di rcvBase nel buffer circolare, se è immediatamente dopo il proprio indice (consumerIndex) dovrà attendere fintanto che rcvBase non si muove in avanti (condizione di ring buffer "vuoto" per il consumatore) al fine di evitare di scrivere su file dati incoerenti (vecchi pacchetti o dati relativi all'inizializzazione del ringbuffer).

Viene definita "***Zona del consumatore***", la zona in cui il consumatore sta leggendo pacchetti dal ringBuffer; coincidente con l'intervallo di posizioni nel buffer circolare: [consumerIndex, rcvbase)

1. Il produttore, quando legge un nuovo pacchetto da socket, prima valuta secondo la semantica del protocollo S.R. in quale sotto insieme dello spazio di numerazione cade il numero di sequenza del pacchetto e in base a questo valuterà se rispondere al Sender con un ack (seqN del pck ricevuto).

Se il pacchetto cade nella ***"zona del consumatore"*** il produttore dovrà attendere fintanto che il consumatore non avanza nel buffer circolare, al fine di non causare scritture di dati inconsistenti .

Infine, il produttore valuta se è necessario far muovere in avanti la finestra di ricezione aggiornando il puntatore condiviso a rcvBase.

GESTIONE DEI TIMER

Al fine di gestire la ritrasmissione basata su timer prevista dal protocollo S.R. sono stati impiegati posix timer, impostati con il flag *SIGEV_NONE* al fine di non mischiare nel contesto di esecuzione segnali classici e segnali per thread.

Le principali dichiarazioni di funzioni e strutture sono presenti nel file timers.h, in particolare è presente una struttura a supporto della gestione dei timer schedulati (*struct timers_scheduled*) nella quale è presente un mutex ed una serie di variabili a supporto del timeout e del suo adattamento.

Dato l'impostazione data ai timer posix, la ritrasmissione dei pacchetti avviene in seguito a un controllo "manuale" dei timer (tramite *gettime()* all'interno della funzione *pollTimers())* ottenuto mediante il timer id presente all'interno delle strutture pck all'interno nel buffer circolare.

Nell'implementazione lo avvio e lo stop di un timer hanno un costo costante data una gestione basata su puntatori, mentre il poll ha potenzialmente costo lineare nella dimensione della finestra di trasmissione, ma il controllo viene sempre limitato a un sotto insieme di interesse [sndbase, sender_consumer_index].

Dato il contesto concorrente di esecuzione nella logica del SENDER, descritto precedentemente, **alcune operazioni sui timer posix sono serializzate tra i thread mediante il mutex sopra citato**.

In particolare, **lo stop e il controllo dei timer** sono eseguiti solo in possesso del mutex, mentre **lo start è lasciato concorrente**. Questo perché mentre le funzionei *pollTimer* e *stopTimer* potrebbero accedere e settare un timer posix in maniera concorrente, la chiamata d'avvio di un nuovo timer è sempre antecedente alla trasmissione del pacchetto e quindi non può mai sovrapporsi a una di tali chiamate.

- **EURISTICA DI ADATTAMENTO DEL TIMER**

Il valore del timer può essere adattivo se non è definita la macro **TIMERFIXED** (configurabile nel makefile). Il valore iniziale del time è configurabile tramite le macro START_TIMEOUT

L'euristica di adattamento del timer è basata su interpretazione di alcuni fenomeni di rete che possono avvenire durante l'esecuzione:

-1 lo stop di un timer (seguente alla ricezione di un ack) può indicare una positiva condizione di rete, è quindi valutabile una diminuzione del timeout.

-2 **lo scadere di un timer indica la perdita di un pacchetto o un valore di timeout troppo piccolo** (che ha causato quindi il prematuro scadere del timer), è quindi valutabile un aumento del timeout.

Date queste possibili interpretazioni delle chiamate effettuate sui timer, l'adattamento del timeout è effettuato mediante un contatore (***time_adaption_counter***) che viene **incrementato o decrementato rispettivamente in seguito all'evento 2 o 1. In particolare, nel caso di un decremento, seguito a uno stop su un timer, la riduzione è proporzionale al rapporto tra il valore di tempo residuo sul timer e il valore corrente sul timer.**

Nel momento che il contatore condiviso (il cui accesso è sempre protetto da mutex) raggiunge una soglia definita (MACRO ADAPTION_UP_THRESHOLD e ADAPTION_DOWN_THRESHOLD) viene effettivamente variato il valore del timeout (di valori configurabili nelle strutture costanti *ADAPTIONUP, ADAPTIONDOWN*).

Il valore del timeout adattivo oscilla sempre al più tra costanti configurabili (MACRO *RAIL_MIN_TIMEOUT, RAIL_MAX_TIMEOUT*).

Al fine di semplificare la logica di gestione dell'adattamento del timer sono state utilizzate le macro per le strutture *timeval*, comportando una riduzione di sensibilità dei timer da nanosecondi a microsecondi .

GUI BINDING

L'associazione della GUI (basata su un semplice script in python) è connessa con il progetto in C tramite l'utilizzo di 2 pipe.

In particolare in seguito alla fase di connessione e inizializzazione del client, se viene data GUI tra i parametri a riga di comando dell'eseguibile, verrà creato un nuovo processo a cui vengono re direzionati i filedescriptor dello standard input e standard output su due pipe e, infine, viene

eseguito lo script tramite la syscall wrapper system. **Tramite le pipe sarà possibile leggere lo standard output del processo su cui è in esecuzione la grafica (su cui verrano passati le scelte dell'utente) e scrivere su standard input i valori da mostrare a grafica**.

Data la significativa differenza tra le performance della logica applicativa in C e la grafica in python le pipe in scrittura sono state estese al massimo consentito per un processo in spazio utente cosi da ridurre la probabilità di saturazione della pipe.

LIMITAZIONI RISCONTRATE

- L'implementazione dello spinlock nel SENDER e nel RECEIVER, può causare una lettura di dati "stantii" da parte dei thread, il che comporta una attesa maggiore del necessario nella produzione e consumazione.
  Al fine di sopperire a ciò sono state dichiarate le variabili utilizzate negli spinlock come *volatile* (evitando ottimizzazioni del compilatore).
  Inoltre, sono state impiegate le primitive atomiche di gcc (le quali da documentazione indicano la necessita di una compatibilità hardware per il loro utilizzo) in versioni ulteriori dei file per la logica di SENDER / RECEIVER (file con suffisso nel nome _ATMOMICS_GCC).
- L'overhead relativo alla gestione dei timer posix non è indifferente, infatti è stato necessario inserire la chiamata di *startTimer* prima dell'invio del pacchetto associato, dato che altrimenti la chiamata a stop (concorrente rispetto all'invio dato che eseguita un differente thread) avveniva spesso prima della chiamata a *startTimer,* causando un inconsistenza per i posix timer.
- I tempi di esecuzione dei lavori sui differenti thread sono significativamente diversi (io da file, in particolare write, contro io da socket). Per tale ragione ho concluso, empiricamente durante lo sviluppo, che si generava un'attesa significativa nel momento in cui scatta la finestra di trasmissione dopo essere stata satura.
  Per tale motivo ho introdotto un margine di bufferizzazione extra rispetto alla dimensione della finestra di trasmissione (macro EXTRA_RINGBUF), cosi da dare più margine di spostamento nel ringBuffer ai diversi thread in alcune situazioni.
- Ho riscontrato performance abbastanza differenti eseguendo il SENDER e simulando il RECEIVER, concludendo che il ricevitore rappresenta il collo di bottiglia dell'app.

CASI DI TEST

Sono allegati al progetto 2 casi di test al fine di verificare:

1) La correttezza del codice relativo alla trasmissione / ricezione di file
2) La concorrenza del client e del server.

Per compilare i test è presente una regola nel makefile (**test**), per l'esecuzione è necessario prima generare dei file da trasmettere (tramite un script bash eseguibile con la regola nel makefile **generate_files**).

Essenzialmente vengono inviati e ricevuti file sull'interfaccia di loopback per poi verificare la coincidenza mediante la chiamata alla funzione delle shell unix *diff* mediante la syscall wrapper *system*.

Nelle prime righe dei codici dei test è possibile configurare i file da scambiare.

È possibile cambiare l'implementazione del RECEIVER e SENDER dalla classica alla versione che utilizza le primitive atomiche di GCC tramite definizione della macro GCCATOMIC_VERSION.

1. **trasmissionTest.c**

Qui viene testata semplicemente la logica di invio e ricezione di file su due processi differenti.

È supposta libera la porta utilizzata nella logica del server (5300).

La socket impiegata viene connessa manualmente nel codice prima della trasmissione del file. Al termine del test verrà verificata la corrispondenza tra i file mediante il comando diff.

Dato che da riga di comando sono impostabili vari parametri trasmissivi (dimensione della finestra, margine extra di bufferizzazione e probabilità di perdita o default se non specificati) questo codice è stato impiegato anche nella analisi delle performance.

2. **multiTX_test.c**

Qui viene testata la correttezza dell'esecuzione in un contesto di trasmissione concorrente.

Per il funzionamento di questo test un processo server deve essere stato lanciato da un'altra shell (in particolare è necessario un server su localhost dietro la porta prefissata dell'applicazione).

Il client viene simulato con il seguente approccio:

-un nuovo processo per l'esecuzione del client viene creato, al quale viene re direzionato il file descriptor dello standard input su pipe.

-viene fatta chiamare una execve sul processo verso l'eseguibile del client con parametri iniziali di default (NO GUI e indirizzo server coincidente con localhost).

N.B. La execve causerà la scrittura sul terminale delle print del client simulato, ma definendo la macro QUIET_OUTPUT ne verranno disabilitati standard output e standard error.

-viene mandato sullo standard input re direzionato del client simulato, una stringa composta sulla base di valori di trasmissione impostati da costanti e, in particolare, vengono scritte due operazioni di scambio file (una GET e una PUT).

Il processo del client simulato leggerà da standard input le operazioni decise e le inizierà consecutivamente, ma ovviamente termineranno in momenti diversi avendo così un periodo di esecuzione contemporanea.

-per notificare una terminazione in seguito all'ultima operazione viene scritto sullo stand input re direzionato un codice speciale(MACRO TERMINATE_AND_DISCONNECT 4) che causerà la terminazione del client in seguito alla terminazione dell'ultima operazione (ovvero ricezione dell'ultimo SIGCHLD da parte del processo padre).

-al termine della simulazione del client viene semplicemente verificata la corrispondenza dei file scambiati mediante diff.

ANALISI PERFORMANCE AL VARIARE DEI PARAMETRI TRASMISSIVI

Sono state valutate le perfomance dell'applicazione mediante l'analisi dei tempi impiegati nella trasmissione di files di dimensioni fissate mediante il caso di test trasmissionTest.c cosi da valutare i tempi impiegati per la trasmissione con diverse configurazioni.

ANALISI TEORICA

Di seguito sono riportate alcune espressioni dell'efficienza del protocollo Selective Repete tratte dal libro *Reti di telecomunicazione di A.Pattavina*.

$$\eta = \begin{cases} 1 & W_s \geq 1 + 2a \\ \dfrac{W_s T_f}{T_f + 2\tau} & W_s < 1 + 2a \end{cases}$$

Figura 6

W rappresenta la dimensione della finestra trasmissiva usata, *Tf* il tempo per trasmettere un pacchetto e *2tau* rappresenta il RTT, *a* rappresenta il rapporto tra il tempo di propagazione e il tempo di trasmissione di un pacchetto.

Le condizioni dei casi del sistema sono relative alla condizione di **trasmissione continua**, ovvero l'utilizzare una finestra che in condizioni ideali permette di mantenere un flusso trasmissivo continuo (in tale condizione l'efficienza è pari a 1)

È possibile notare come aumentando la dimensione della finestra o il tempo per trasmettere un pacchetto è possibile incrementare l'efficienza del protocollo, diminuendo conseguentemente il tempo impiegato complessivamente per la trasmissione.

Nella seguente espressione è confrontata l'efficienza di SR con il protocollo GBN in presenza di errori, con probabilità di perdita P.

$$\eta = \begin{cases} \dfrac{W_s(1-P)}{1+2a} & \text{SR} \\ \dfrac{W_s(1-P)}{(1+2a)[1+(W_s-1)P]} & \text{GBN} \end{cases}$$

Figura 7

In quest'ultima è possibile notare come in presenza di perdita un aumento della finestra di trasmissione migliora l'efficienza del protocollo.

MISURAZIONE EMPIRICA

Al fine di velocizzare la raccolta dei dati è stato utilizzato lo script bash *getTimesPerformance.sh*, il quale comunica anche con il Makefile tramite un file di appoggio (al fine di variare dinamicamente delle MACRO a tempo di compilazione, come la dimensione dei pacchetti). Tutti i dati relativi ai seguenti grafici sono all'interno della cartella PERFORMANCE.

Sono state valutate le performance secondo 3 metriche:

1. **ANALISI TEMPI DI TRASMISSIONE CON VARIAZIONE DELLA DIMENSIONE DEI PACCHETTI**

   Mantenendo una dimensione della finestra costante (abbastanza larga cosi da raggiungere la teorica condizione di trasmissione continua) è stata fatta variare la dimensione dei pacchetti.

PARAMETRI FISSI:	
WINSIZE	33
LOSSP	0
EXTRA RINGBUFF SIZE	3
FileSize	880MB

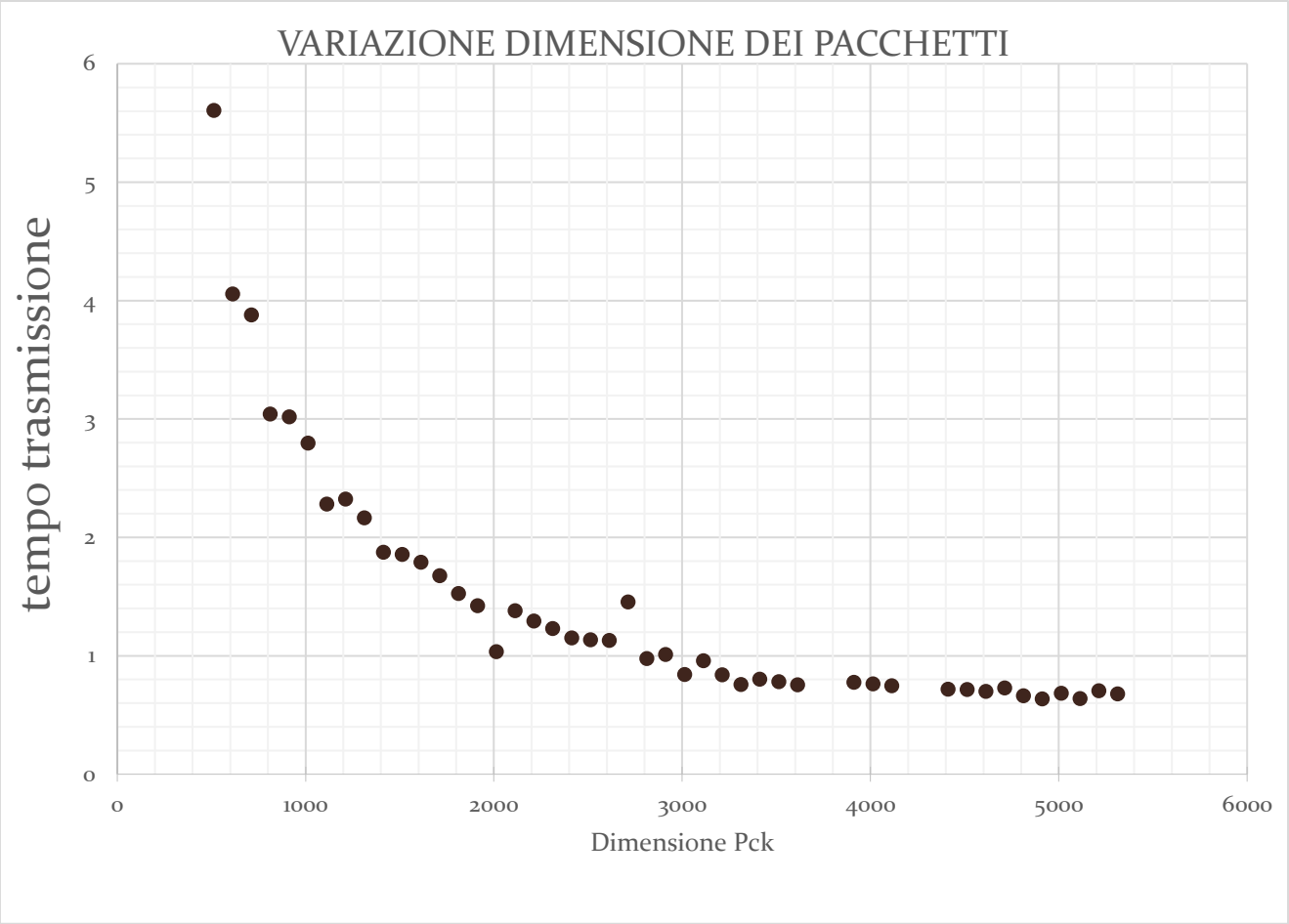

Figura 8

È possibile notare come **incrementando la dimensione dei pacchetti il tempo impiegato per la trasmissione del file tende a ridursi.**

Questo è conseguenza sia del fatto che le read e write da file procedono a passi più grandi e sia dall' incremento della dimensione dell'unita minima trasmessa nell'applicazione.

È inoltre possibile notare come per pacchetti di dimensione oltre 3000 byte il tempo di trasmissione misurato tende a restare pressoché costante.

Questo è in accordo con le considerazioni relative alla figura 6 dato che in questa maniera si aumenta Tf.

2. ANALISI DEI TEMPI DI TRASMISSIONE CON FINESTRA DI DIMENSIONE VARIABILE

Mentendo ora costanti dimensione dei pacchetti è margine di bufferizzazione extra nel buffer circolare è stata fatta variare la finestra di trasmissione misurando il tempo impiegato.

PARAMETRI FISSI:	
PCKSIZE	1024
EXTRA RINGBUFF SIZE	3
PLOSS	0
FileSize	880MB

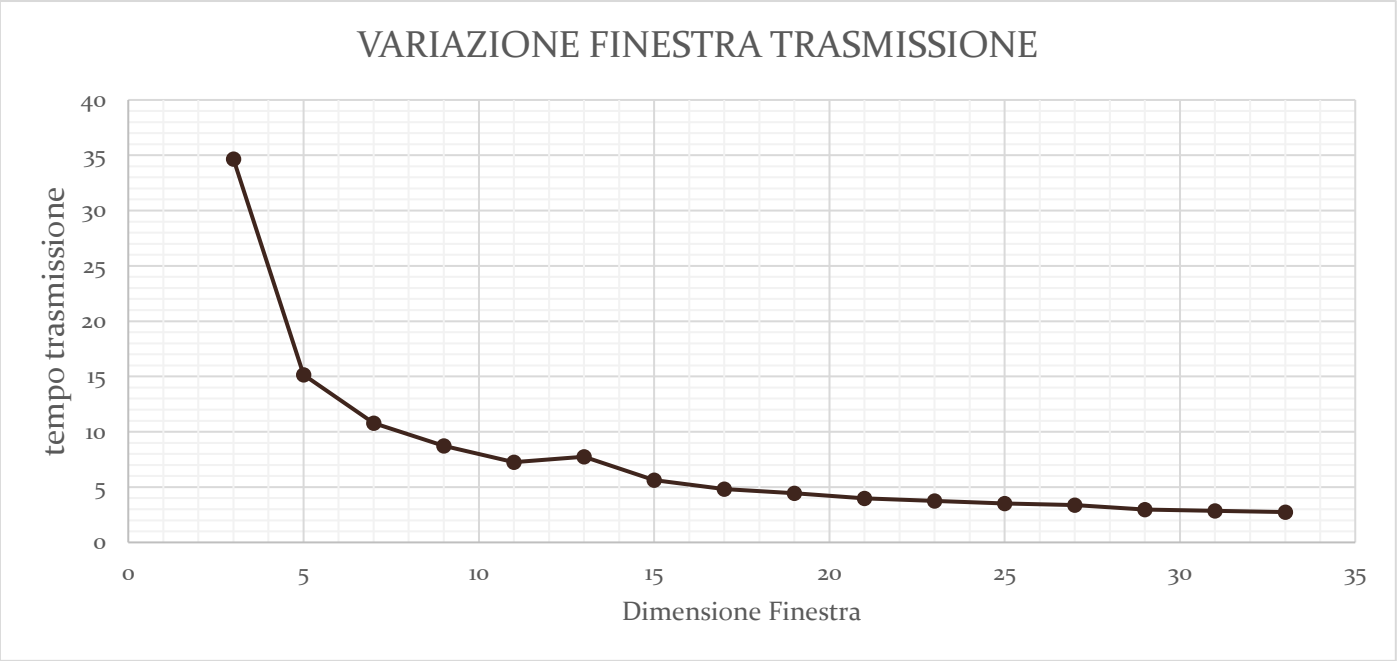

Figura 9

è possibile notare come **aumentando la dimensione della finestra il tempo impiegato tende a ridursi fino a minimo** oltre il quale incrementi ulteriori della finestra non danno significative variazioni. Questo è in accordo con le considerazioni relative alla figura 6 dato che si incrementa Ws e ci si avvicina alla condizione di trasmissione continua.

Inoltre, **una finestra di dimensione maggiore da uno spazio di "movimento" maggiore per i thread che si occupano della produzione / consumazione dei pacchetti, riducendo i periodi di stallo legati allo spinlock**.

3. ANALISI DEI TEMPI DI TRASMISSIONE CON FINESTRA DI DIMENSIONE VARIABILE E PERDITA SIMULATA.

Mantenendo costante la dimensione dei pacchetti e una probabilità di perdita pari a 0.25 sono stati fatti variare la dimensione della finestra di trasmissione e il margine di bufferizzazione extra nel buffer circolare.

Ai fini rappresentativi sono riportati due grafici, con gli stessi parametri costanti ma con incrementi della finestra di trasmissione differenti.

PARAMETRI FISSI:	
PCKSIZE	1024
PLOSS	0,25
FileSize	62MB

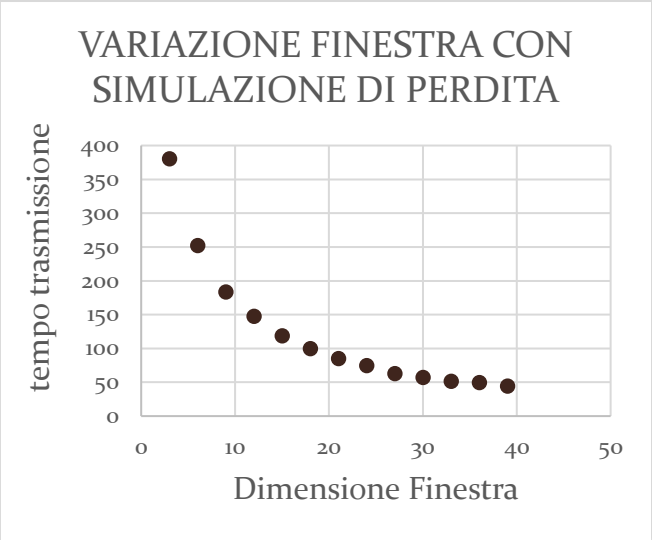

Figura 10

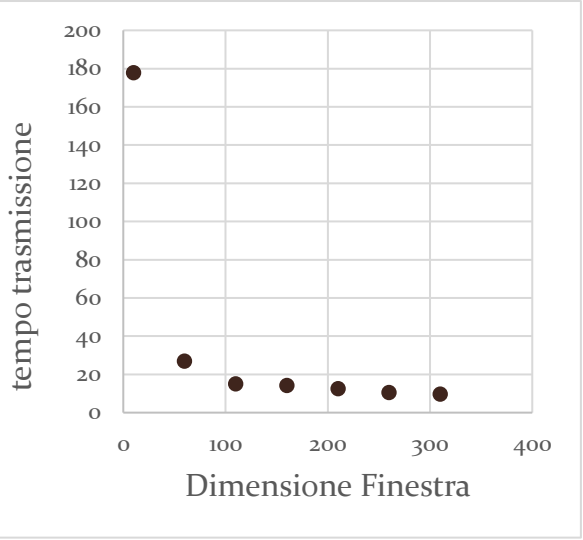

Figura 11

È possibile notare come **il tempo impiegato alla trasmissione si riduce notevolmente in seguito a un aumento della finestra di trasmissione fino a un minimo oltre il quale non si hanno significative variazioni**. Questo è in accordo con le considerazioni relative alla figura 7 dato che si aumenta Ws.

Infatti, dato che con una perdita, la finestra di trasmissione si blocca fino alla successiva ricezione dell'ack, con una finestra più grande è possibile che la ritrasmissione avvenga durante la trasmissione di altri pacchetti, riducendo cosi la quantità di tempo durante il quale REVEIVER e SENDER sono in attesa, rispettivamente, della ritrasmissione e della riconferma.

Inoltre, dato che nel momento di blocco della finestra i thread tendono a rimanere a lungo nello spinlock, **con una finestra maggiorata e un incremento del margine di bufferizzazione extra è possibile ridurre lo stallo dei thread migliorando notevolmente le performance.**

MANUALE DI INSTALLAZIONE

È presente nella cartella sources un Makefile le cui principali regole sono:

all: compila i file per il client e server e posiziona gl'eseguibili nella cartella sources

server.o: compila i file per il solo server

client.o: compila i file per il solo client

test: compila i file per i due casi di test e posiziona gl'eseguibili nella cartella test

generate_files: genera dei file riempiti di zeri nella cartella adibita allo scambio file: files_sendable.

Per l'aggiunta del timeout adattivo alla logica del client e del server è necessario commentare la riga del Makefile relativa alla definizione della macro TIMEOUTFIXED

Per l'interfaccia grafica, eseguibile con un interprete python2.7, può essere necessario un piccolo package ulteriore (se l'installazione di python è la minimale come usuale su linux) denominato python-tk

Su distribuzioni linux debian based: *sudo apt-get install python-tk.*

I file scambiati devono essere contenuti nella cartella files_sendable del progetto.

ESEMPI D'USO

L'avvio del client e del server inizia sempre a partire da un esecuzione da shell degl'eseguibili client.o e server.o.

Configurazione dei parametri trasmissivi

I principali parametri trasmissivi (dimensione della finestra e probabilità di perdita) sono inviati al server a runtime per ogni nuova connessione proveniente da un client. È tuttavia possibile impostare tali parametri con delle macro configurabili a tempo di compilazione se nel makefile viene definita la macro TX_CONFIG_FIXED. In quest'ultimo caso non sarà possibile utilizzare la GUI.

Inoltre per il timeout adattivo non deve essere definita la macro TIMEOUTFIX.

Diversi altri parametri trasmissivi sono definiti tramite MACRO all'interno degl'header file, vengono qui riportati i principali:

- app.h

  - EXTRABUFRINGLOGIC: dimensione di bufferizzazione ulteriore alla finestra di trasmissione all'interno del buffer circolare

  - PCKPAYLOADSIZE: dimensione del payload dei pacchetti all'interno del buffer circolare. Configurabile anche con make se il valore è definito in un file denominato pcksize.

  -TIMEOUTCONNECTION: quantità di secondi di inattività dopo il quale una connessione è considerata chiusa dal server

  -_SERV_PORT: porta utilizzata dal processo principale del server che accetta nuovi clients

- In sender.h

  -FILEIN_BLOCKSIZE: quantità massima di pacchetti prodotti per ogni iterazione

Avvio e uso

L'avvio del server presuppone che nella macchina in uso sia libera la porta definita nella macro _SERV_PORT (di default 5300). Il server si avvia semplicemente da shell a partire dall'eseguibile prodotto dal make (nella cartella sources "./server.o").

L'esecuzione del client prende i seguenti parametri:

<indirizzo IP del server, tipo di UI> (CLI per riga di comando, GUI per interfaccia grafica), di default contatterà il server su localhost con interfaccia testuale.

In seguito all'avvio, se la connessione con il server avviene con successo, **verranno chiesti di inserire la dimensione della finestra di trasmissione e la probabilità di perdita di pacchetti**. (i seguenti parametri trasmissivi saranno comunicati tramite la socket connessa al server, il quale può gestire differenti client con differenti parametri trasmissivi).

A questo punto è possibile inserire uno tra i comandi LIST, GET, PUT, DISCONNECT. Nella esecuzione del client con interfaccia testuale per ogni comando è associato un codice.

Segue una breve descrizione dei comandi in oggetto, dove il numero anteposto ad ogni punto è il codice dell'operazione:

0) DISCONNECT: termina la comunicazione, interrompendo tutte le operazioni attualmente in esecuzione
1) PUT: invia un file presente nella directory files_sendable di cui verrà chiesto il nome
2) GET: richiedi un file presente nella directory files_sendable del server, il cui nome verrà chiesto
3) LIST: richiedi la lista di nomi e dimensioni dei files disponibili nel server

4) TERMINATE_TXs_AND_DISCONNECT: aspetta il completamento delle operazioni in corso e poi termina la comunicazione con il server

In seguito ad ogni operazione richiesta verrà notificata, mediante la UI scelta, il progresso dell'operazione e l'esito dell'operazione. In particolare, gl'aggiornamenti di una trasmissione derivano da print nell'algoritmo di trasmissione **e la terminazione della trasmissione è notificata con la terminazione del processo creato**, incaricato della tale.

Esempio d'uso con interfaccia testuale

```
andysnake@debian:~/Scrivania/net/PRJNETWORKING/sources$ ./client.o 127.0.0.1 CLI
trying connecting to server at addr :127.0.0.1
arrived -93 from 127.0.0.1:57962

correctly connected with :127.0.0.1:57962
input mode cli
:>      insert min trasmission window size for file tx
10
:>      insert pck loss simulation  probability
0
SERIALIZED TX CONFIGS  SRC:      TX_CONFIG: WSIZE 10      MAX SEQN 30     EXTRA RING BUF SPACE 5
LOSS PROBABILITY 0.000000

USAGE
->PUT           1
->GET           2
->LIST          3
->DISCONNECT    0
3
AVAIBLE FILES ON SERVER:
-cli_generateNullFile.sh    149
-enjoy.mp4      62047639
-k.mp4  884016901
-generateNullFile.sh        149
-cli_k.mp4      884016901
-serv_k.mp4     884016901
-serv_enjoy.mp4 62047639
-cli_enjoy.mp4    62047639

2

INSERT FILENAME TO GET
enjoy.mp4
successfully obtained new connected socket for OP WITH :127.0.0.1:47569
GET CONNECTION OBTAINED correctly created a new process to handle GET req on pid :8009
requested file(with prefix) : cli_enjoy.mp4 of size 62047639
```

Esempio d'uso con GUI