D atagram
F ile
T ransfer

Trasferimento dati affidabile con UDP

D'Aniello Simone

Frascaria Giulia

January 27, 2017

Contents

1 Introduzione **3**

2 Architettura del sistema **4**
2.1 Visione macroscopica 4
2.2 Analisi di una connessione 5
2.3 Analisi di un processo 6

3 Progettazione del funzionamento **8**
3.1 Requisiti funzionali 8
3.2 Invio dei dati 9
3.3 Gestione dell'affidabilità 10

4 Algoritmi e strutture dati **13**
4.1 Selective Repeat 13
4.2 Timer Wheel 15
4.3 Connessione e Disconnessione 18

5 Comandi e trasmissione dati **20**
5.1 Operazioni 20
5.2 Lettura da socket e pipe 22
5.3 Ritrasmissione 23
5.4 Gestione degli errori 24

6 Esempi **27**

7 Testing **30**

8 Configurazione **33**

9 Limiti e potenzialità **35**

10 Strumenti utilizzati **37**

1. Introduzione

In questa relazione verrà presentato ed esposto il lavoro svolto per la progettazione e l'implementazione in linguaggio C di un'applicazione distribuita, con architettura di tipo client-server, e avente come obiettivo il trasferimento di file attraverso la rete Internet.

Lo sforzo di progettazione è stato rivolto al garantire la comunicazione affidabile tra client e server, secondo il modello offerto da TCP, nonostante l'utilizzo del protocollo di rete UDP a livello di trasporto. Un notevole impegno è stato inoltre dedicato al garantire robustezza architetturale ed efficienza nelle prestazioni, nel rispetto delle richieste di affidabilità.

L'obiettivo di questo documento non è solo quello di effettuare un'analisi approfondita delle scelte progettuali adottate per lo sviluppo dell'applicazione, ma anche di mostrare all'utente una descrizione dettagliata delle prestazioni ottenute, giustificandone ove possibile il risultato, e fornire una guida il più possibile immediata per permettere l'esecuzione del software su un qualsiasi terminale Linux.

Verranno analizzate tutte le scelte progettuali ed implementative, fornendo esempi e controesempi per evidenziare i punti di forza delle soluzioni adottate, così come eventuali loro limitazioni e prospettive di miglioramento per uno sviluppo ulteriore.

2. Architettura del sistema

In questa sezione verrà analizzata la struttura dell'applicazione distribuita, le fasi in cui si articola una connessione corretta e i moduli che compongono un agente dell'applicazione.

2.1 Visione macroscopica

Il sistema software presentato è stato sviluppato secondo i canoni di un'applicazione distribuita di tipo Client-Server. Il server, attivo ed in ascolto su una porta nota, gestisce le richieste di connessione dei client.

Al fine di garantire a più utenti l'accesso concorrente al servizio offerto, si è reso necessario gestire molteplici connessioni mediante un server multiprocesso. Oltre a garantire una continuità di servizio a nuovi client che vogliono iniziare una connessione senza ritardi di accodamento, la scelta di costruire un server multiprocesso fornisce un maggiore isolamento tra diversi utenti, sollevando il server principale dalla gravosa necessità di sincronizzare e gestire le risorse condivise. In questo modo la garanzia di isolamento delle connessioni è invece delegata al normale funzionamento del sistema operativo. Ogni client viene gestito da un processo dedicato creato appositamente dal server principale [figura 2.1], che torna immediatamente in attesa di nuove connessioni in ingresso.

La vita di un singolo processo figlio è limitata alla durata della connessione con il proprio client, la quale può terminare in modo corretto su richiesta dello stesso, oppure in modo anomalo qualora si verifichino errori nell'esecuzione. Tutta la procedura avviene comunque in modo trasparente al server principale, il quale non riceve più messaggi da un client connesso per cui è già stato

avviato un processo servente.

L'applicazione vuole quindi mantenere un server principale dal funzionamento semplice, il cui unico compito è quello di gestire le nuove connessioni in ingresso. Un client viene nel contempo servito attraverso un processo ad hoc creato in modo trasparente. Su tale processo viene posto tutto l'onere di gestione della connessione affidabile con il client.

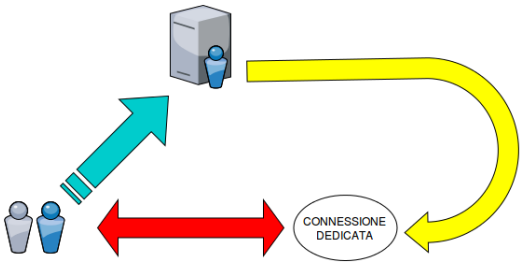


Figure 2.1: Connessione di un client

2.2 Analisi di una connessione

Un utente, al momento del'avvio di un processo client, effettua automaticamente una richiesta di connessione al server noto. Da questo momento in poi viene automaticamente messo in comunicazione con il proprio server dedicato, che ne gestisce le richieste fino al termine del servizio.

All'interno di un singolo processo si è deciso di disporre di un ulteriore livello di parallelismo, mediante un meccanismo di multithreading graduale che avviene durante la procedura di connessione, seguendo i passaggi del three-way-handshake propri del protocollo TCP.

Al termine della connessione, client e server sono connessi attraverso due canali di comunicazione. Entrambi i processi creano due diverse socket unidirezionali, ciascuna delle quali è gestita da un thread che si occupa del flusso di dati rispettivamente in ingresso o in uscita. Si è scelto di seguire per questa organizzazione il modello offerto dal protocollo FTP, creando due flussi di dati (uno di datagrammi con payload informativo e uno di messaggi di controllo) che viaggiano in versi opposti.

Lo scambio di informazioni tra i terminali remoti avviene dunque attraverso due socket, accedute in modo esclusivo da due thread chiamati Listener e

Sender. Questi sono tra loro solidamente sincronizzati per gestire in modo corretto e coerente il flusso di dati in ingresso e in uscita [figura 2.2].

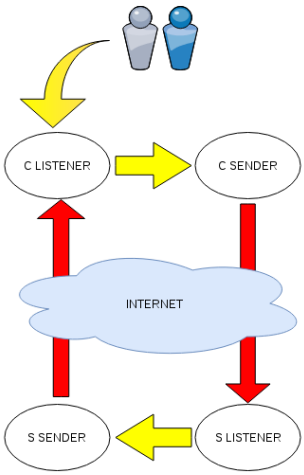


Figure 2.2: Comunicazione tra thread remoti e locali

2.3 Analisi di un processo

La scelta di costruire un'applicazione multiprocesso, e in particolare multithreaded ha posto in essere il problema della sincronizzazione e della gestione delle risorse condivise. Verrà ora analizzato un processo server, analogo nel funzionamento ad un processo client, evidenziando dove opportuno le differenze a livello logico.

Un server, al termine della procedura di connessione con il proprio client, si trova ad avere 3 thread concorrenti che contribuiscono al funzionamento dell'applicazione. Il listener e il sender, già introdotti nella precedente sezione, si occupano dell'accesso alle socket. Oltre a questi, opera in background un thread dedicato alla gestione dei timer dei pacchetti che segnala eventuali necessità di ritrasmissione al sender thread.

Listener e sender sono tra loro gerarchizzati con una logica master-slave. Il thread listener è infatti l'unico ad essere sempre attivo, in attesa di comandi dall'utente nel caso del client, e di pacchetti dalla rete per quanto riguarda il server. Il sender viene attivato a necessità e soltanto fino al completamento di una transazione, ovvero la sequenza di operazioni necessarie per portare a termine l'esecuzione di un comando.

Anche il thread timer è attivo in background soltanto durante le operazioni che coinvolgono l'invio di dati che necessitano di riscontro. Per funzionare correttamente, è collegato in modo diretto al sender mediante una pipe in cui invia messaggi relativi a pacchetti per cui è stato rilevato un timer scaduto.

Una volta attivato, il thread sender opera sempre sotto il controllo diretto del listener, che manda segnalazioni per l'invio di acknowledgement o pacchetti. Nel caso in cui si stiano inviando dati da riscontrare, il thread sender è però controllato in modo più stringente dal thread timer, per permettere ritrasmissioni con diritto di prelazione sulla trasmissione normale. In questo modo non si raggiunge una situazione di stallo nel funzionamento in caso di perdita di pacchetti nella rete.

Si è cercato in questo modo di ottimizzare l'allocazione e l'uso delle risorse, garantendo il funzionamento dell'applicazione con un'attività limitata ai periodi di trasferimento dei file e di comunicazione tra processi. Soprattutto il server trae diversi benefici da queste scelte di progettazione: il mantenimento in parallelo di processi server dedicati non è troppo esoso grazie alla politica di risparmio delle risorse. Inoltre, i diversi canali dedicati ai client rendono possibile l'invio di più flussi di dati in parallelo, velocizzando le connessioni e evitando tempi di overhead per il corretto indirizzamento dei pacchetti.

3. Progettazione del funzionamento

Questo capitolo chiarirà gli obiettivi di funzionamento dell'applicazione e le scelte progettuali effettuate per garantire il soddisfacimento dei requisiti funzionali e di ottimizzazione.

3.1 Requisiti funzionali

L'applicazione in esame ha come obiettivo il trasferimento di dati tra terminali remoti. I comandi fondamentali per permettere questa funzionalità devono permettere di elencare i file presenti nell'host server e di effettuare trasferimenti da e per tale destinazione.

Questo requisito ha reso necessaria l'ideazione un meccanismo di trasferimento che permetta la suddivisione del contenuto in pacchetti inviati separatamente, in modo da poter gestire con flessibilità file di ogni dimensione. Conseguentemente è emersa la necessità di ricostruire correttamente il contenuto a destinazione, nonostante l'eventuale presenza di buchi causati da una ricezione in ordine non sequenziale.

L'utilizzo del protocollo UDP a livello di trasposto ha come conseguenza l'assenza di garanzie di affidabilità. Questo ha obbligato ad adottare un protocollo di comunicazione che effettuasse riscontro e ritrasmissione di pacchetti persi. Ciò è stato conseguito con l'implementazione di un algoritmo di tipo Selective Repeat e conseguentemente di un meccanismo di gestione di timer per singoli pacchetti.

Data la complessità dell'applicazione, si è cercato di progettare ed implementare le scelte con l'obiettivo di ottimizzare l'utilizzo delle risorse e consentire un'esecuzione veloce dei meccanismi di background atti a garantirne il funzionamento.

3.2 Invio dei dati

Nella progettazione è stata rivolta notevole attenzione all'efficienza della fase di scambio dati tra i terminali. Si esporranno in questa sezione le motivazioni che hanno portato all'ideazione della struttura di dati utilizzata come messaggio applicativo e i meccanismi di ricostruzione del file nell'host remoto.

Il meccanismo di comunicazione tra il client e il server prevede l'invio di messaggi aventi struttura e lunghezza noti. Questo comporta la suddivisione di ogni file in messaggi di lunghezza prefissata. Come si vedrà meglio nel capitolo relativo all'implementazione, questa standardizzazione facilita notevolmente le operazioni di lettura e scrittura sulla socket e la ricostruzione del file sul terminale remoto grazie a un meccanismo di puntamento basato sul numero di sequenza del datagramma ricevuto.

Il primo passo della progettazione del messaggio applicativo ha riguardato la scelta della dimensione per il datagramma. Anche in questo caso, si è cercato di ottimizzare in qualche modo l'applicazione. La scelta si è orientata, più che sulla minimizzazione dei datagrammi scambiati, sulla velocizzazione del loro passaggio attraverso la rete internet, nel tentativo di uniformare il più possibile i Round Trip Time, e conseguentemente l'efficacia dei timer. L'obiettivo è stato conseguito scegliendo la misura di datagramma che evitasse il rischio di frammentazione del datagramma a livello IP.

L'RFC 791 indica che ogni host deve essere in grado di gestire, a livello di rete, un datagramma di 576 byte senza bisogno di effettuare frammentazione. Sottraendo a questa dimensione i 20 byte di header IP e gli 8 byte di header UDP rimane uno spazio di 548 byte per il messaggio a livello applicativo. Si è scelto quindi di riservare 512 byte al trasporto di contenuto informativo, lasciando i restanti 36 byte all'header. Questo è composto da campi di tipo int e short int, quindi nelle più comuni architetture moderne è di circa 16 byte [figura 3.1]. I restanti byte sono lasciati come margine nel caso di utilizzo in architetture che hanno interi di dimensione maggiore.

Evitare la frammentazione a livello IP permette di velocizzare il tempo di elabolazione del datagramma all'interno della rete. L'host di destinazione inoltre non va incontro ad un overhead aggiuntivo derivante dall'attesa di tutti i frammenti che permettono di ricostruire il datagramma originario. In questo modo il gestore dei timer gode di condizioni favorevoli in cui operare,

riuscendo a regolare in modo preciso i timer dei pacchetti, evitando inutili ritrasmissioni.

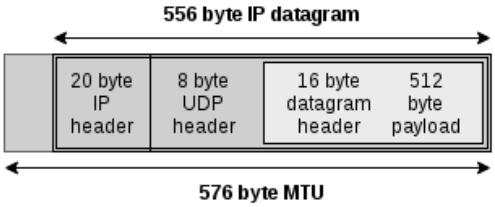


Figure 3.1: Studio per evitare frammentazione

La suddivisione in chunk di 512 byte obbliga inoltre a implementare un algoritmo che permetta di comporre in modo corretto il file nell'host di destinazione, gestendo in modo adeguato sia frammenti ricevuti fuori sequenza che eventuali duplicati derivanti da ritrasmissioni. Il meccanismo adottato si avvale del solo numero di sequenza del pacchetto ricevuto per indirizzare in modo puntuale la posizione nel file dei 512 byte ricevuti rispetto al punto di riferimento, rappresentato dal primo numero di sequenza della trasmissione corrente. Viene in questo modo minimizzato il numero di informazioni aggiuntive necessarie al corretto funzionamento dell'applicazione.

La standardizzazione della fase di trasferimento dei dati che è stata appena introdotta ha permesso di ottenere un funzionamento dell'applicazione rapido e fluido, ma soprattutto flessibile e adattabile a diverse condizioni di operatività, per quanto sfavorevoli.

3.3 Gestione dell'affidabilità

La specifica sull'affidabilità della trasmissione ha imposto il bisogno di implementare algoritmi e strutture dati differenziati ma tra loro coerenti per l'host sender e listener. Verranno esaminate ora le scelte progettuali fatte per implementare l'algoritmo Selective Repeat e l'algoritmo Timing Wheel per la gestione dei timer.

Ci soffermeremo dapprima sull'host sender. La fase di invio dei dati necessita di uno stretto controllo sui dati inviati e sui riscontri ricevuti. Inoltre, qualora per un pacchetto non venga ricevuto un riscontro entro un tempo stabilito e ritenuto significativo, si deve procedere alla ritrasmissione

della stessa porzione di dati, fino alla ricezione del riscontro.

E' quindi necessario per l'host sender inviare dati in modo controllato, monitorare le di ritrasmissioni e tenere traccia dei riscontri ricevuti. Queste tre mansioni sono portate avanti in parallelo rispettivamente dai thread sender, timer e listener. La coordinazione tra thread avviene sia attraverso canali di comunicazione dedicati, che in modo implicito attraverso l'aggiornamento di strutture dati globali.

Il thread sender invia dati rispettando la finestra di spedizione dell'algoritmo Selective Repeat aggiornandola con ogni invio e facendo partire il timer del pacchetto inviato. In background, la finestra è fatta avanzare dal thread listener quando riceve acknowledgement dei pacchetti inviati, che comportano anche lo stop del timer. D'altra parte il thread sender monitora continuamente una coda di messaggi con cui il thread timer segnala i numeri di sequenza dei pacchetti per cui ha trovato un timer attivo.

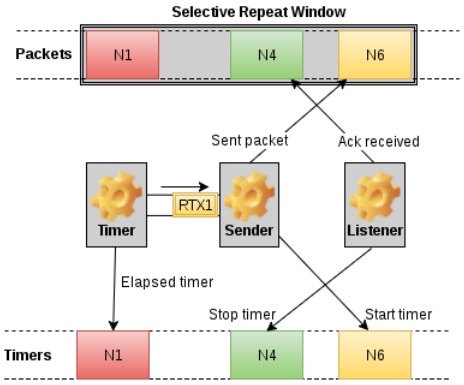


Figure 3.2: Flusso dati in un terminale

Anche l'host che riceve i dati ha bisogno di una sincronizzazione tra i thread che operano. Tuttavia in questo caso il thread timer non è attivo in quanto non si ha bisogno di inviare dati da riscontrare, ma soltanto acknowledgement per i pacchetti ricevuti. Cooperano quindi il thread listener e il thread sender. Il primo è in ascolto sulla socket e, nel momento della ricezione di un pacchetto, scrive il numero del pacchetto di cui mandare il riscontro in una coda di messaggi che il sender legge continuamente. Mentre il sender si limita all'invio di acknowledgement, il Thread listener è contemporaneamente impegnato nella ricostruzione del file a partire dalle porzioni ricevute, Scrivendo i 512 byte di contenuto del pacchetto in una posizione del file univocamente determinata, calcolata mediante la differenza tra il numero di sequenza ricevuto

e il primo della transazione.

I meccanismi di sincronizzazione introdotti permettono ai tre thread in esecuzione di cooperare in maniera fluida al funzionamento dell'applicazione, mediante semplici scambi di messaggi e aggiornamenti di strutture di dati globali. Tutto il tempo di computazione durante l'esecuzione non è quindi perso nella sincronizzazione ma usato in modo costruttivo per trasmettere dati con efficienza.

4. Algoritmi e strutture dati

Questo capitolo si soffermerà sull'effettiva implementazione degli algoritmi introdotti fino ad ora. Si cercherà di giustificare tutte le scelte implementative alla luce dell'ottimizzazione ottenuta.

4.1 Selective Repeat

Per l'applicazione DFT si è implementato il metodo noto come Selective Repeat, ossia è un protocollo a ripetizione automatica (ARQ) che permette l'invio in pipeline di più pacchetti. I pacchetti vengono inviati entro i confini di una finestra di spedizione di dimensione fissata e che parte dal più vecchio pacchetto di cui ancora non si è ricevuto un riscontro. La finestra di spedizione quindi scorre sui numeri di sequenza dei pacchetti man mano che la trasmissione procede.

Il meccanismo di scorrimento della finestra di spedizione ha posto di fronte alla difficoltà di implementare in modo efficiente una struttura altamente dinamica mantenendo però un tempo di accesso ridotto ai dati riguardanti un singolo pacchetto identificato. A livello logico quindi la scelta si è orientata verso l'utilizzo di una struttura di dati circolare avente le dimensioni della finestra di spedizione.

Il modo scelto per implementarla mantenendo nel contempo un tempo di accesso costante alle informazioni dei pacchetti è stato di utilizzare un array di celle aventi una struttura definita. Questo array è popolato ed esplorato indirizzando le celle mediante il numero di sequenza dei pacchetti modulo la dimensione della finestra. In questo modo si è riusciti a simulare una situazione di dinamismo mediante una struttura di dati statica che permette

un accesso efficiente e privo di tempi di ricerca.

Come si è detto, le celle dell'array contengono una struttura dati così definita:

value — A seconda del valore che assume questo intero, il pacchetto corrispondente alla cella può essere identificato come **non ancora spedito**, cella vuota (**0**), **spedito non riscontrato** (**1**) o **spedito e riscontrato** (**2**).

packetTimer — Struttura dati utilizzata dai tre thread dell'applicazione per gestire le ritrasmissioni. Questa struttura dati verrà analizzata in seguito.

cellMtx — Un mutex utilizzato in ogni accesso alla cella della struttura globale, proteggendo i dati da ogni corruzione derivante da race condition.

Nella finestra di spedizione viene inoltre identificata come *sendBase* la posizione nell'array corrispondente al più vecchio pacchetto inviato e non ancora riscontrato. È di fondamentale importanza in quanto il thread Sender, per l'invio di ogni pacchetto, deve controllare che il numero di sequenza del pacchetto (e di conseguenza la posizione nella ruota) rientri all'interno dell'attuale finestra di spedizione che parte dal *sendBase*.

Anche durante lo spostamento di *sendBase*, e quindi della corrente finestra di spedizione, rimane costante la posizione nell'array di una cella corrispondente a un pacchetto inviato.Chiamata *windowDimension* la dimensione della finestra infatti, la *selectCell* corrispondente al pacchetto inviato sarà sempre quella corrispondente al valore *packet.seqnum % windowDimension*.

All'inizio di ogni transazione avviene un'inizializzazione della struttura circolare chiamando un'apposita funzione `initWindow` che, se chiamata per la prima volta, si occupa anche di eseguire la chiamata di funzione `pthread_mutex_init` su ciascun mutex. Viene inoltre impostato il valore di *sendbase* al numero di sequenza del primo pacchetto inviato. A seguito dell'invio di un pacchetto, il thread Sender chiama la funzione `sentPacket` attraverso la quale vengono aggiornati i valori della *selectCell* corrispondente e del relativo timer. In particolare viene impostato a 1 il valore di *value* e *packetTimer.seqnum* viene inizializzato il timer con il numero di sequenza del pacchetto spedito.

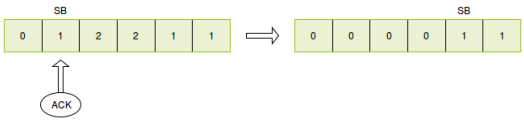


Figure 4.1: Spostamento SendBase

Per permettere la trasmissione continua, la finestra di spedizione ha bisogno di un continuo scorrimento. Questo è operato dal thread Listener, che per ogni ACK ricevuto aggiorna la finestra attraverso la chiamata di funzione `ackSentPacket`, che ferma anche il timer del pacchetto corrispondente. Nel caso in cui l'ACK ricevuto sia relativo al pacchetto nella posizione di *sendbase* nella ruota, la finestra scorre fino a trovare la prima posizione di un pacchetto non ancora riscontrato, che diventa la nuova *sendbase* della finestra.

Utilizzare una struttura statica in modo dinamico, simulando un comportamento circolare, permette un funzionamento tanto semplice quanto efficiente per il protocollo Selective Repeat. Ciascuna operazione sulla finestra di ritrasmissione si limita infatti al cambio di valore di una variabile o al controllo di un valore. E' una struttura minimale ma efficace che rappresenta un canale di comunicazione fondamentale tra i tre thread.

4.2 Timer Wheel

Il thread Timer è un thread che opera esclusivamente in background nell'esecuzione dell'applicazione, ma ha un ruolo fondamentale nel garantire l'affidabilità del protocollo di comunicazione ideato. Per questo motivo ciascuna scelta progettuale ed implementativa ha cercato di rendere il funzionamento semplice e immediato, quanto robusto.

Dovendo implementare una trasmissione di dati con un protocollo di tipo Selective Repeat, gli acknowledgement e le ritrasmissioni sono limitati al singolo pacchetto mandato, e non vi è un meccanismo di cumulazione come nei protocolli che implementano una strategia di tipo Go-Back-N. Per questo motivo i timer utilizzati nell'applicazione non possono essere riferiti ed utilizzati per più di un pacchetto ancora "in volo".

La trasmissione di un file mediante l'applicazione comporta quindi la

presenza in contemporanea di numerosi timer attivi e di un continuo processo di ritrasmissioni e disattivazioni per pacchetti riscontrati. er permettere un funzionamento fluido all'applicazione, è stato quindi necessario sfruttare una struttura di dati e un algoritmo che permettesse una gestione efficiente di numerosi timer acceduti in modo concorrente da più thread.

Poichè un'allocazione dinamica di nuovi timer per ogni trasmissione sarebbe risultata dispendiosa, oltre che di difficile sincronizzazione, si è deciso di assegnare un timer ad ogni cella della struttura di Selective Repeat in modo da poter avere anche in questo caso un tempo di accesso costante per le operazioni di *startTimer* e *stopTimer* effettuate rispettivamente dal thread sender e listener.

Il problema principale per la gestione dei timer ha riguardato invece l'organizzazione di una struttura dati gestita dal timer che identificasse univocamente l'elenco di timer scaduti in un determinato istante dell'esecuzione, evitando per quanto possibile delle ripetute operazioni di ricerca tra tutti i timer attivi. Si è deciso quindi di implementare una struttura dati denominata Timing Wheel, seguendo l'idea riportata nel paper [4].

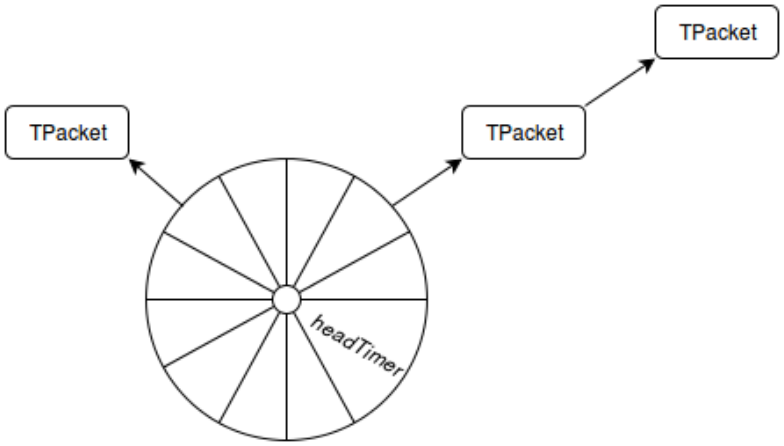


Figure 4.2: Struttura della wheel

Questa struttura ricorda per il suo operato un orologio: anche in questo caso infatti si è utilizzato un array, esplorato in modo circolare. A ogni avanzamento dell'orologio, che avviene ad intervalli di tempo regolari, viene esaminata una cella che contiene soltanto lista dei timer impostati per scadere nell'istante di tempo corrente. È importante notare come il tempo dedicato alle operazioni di ricerca dei pacchetti da ritrasmettere sia in questo modo reso del tutto trascurabile, in quanto si limita alla sola esplorazione della lista

collegata di timer per cui è possibile una ritrasmissione nell'istante corrente. Alcuni di questi timer possono essere non più validi in quando è stato ricevuto un ack epr il rispettivo pacchetto.

Ciascuna cella della Timing Wheel qui introdotta contiene una struttura *headTimer* molto semplice, che funge da testa della lista collegata di timer.

```
1 struct headTimer
2 {
3     volatile struct timer * nextTimer;
4 };
```

Connesse ad *headTimer* attraverso puntatori ci sono le strutture *timer*[1] di cui il thread timer analizza solo il campo di validità ed eventualmente il numero di sequenza. Queste strutture formano una catena che viene esplorata in profondità durante il controllo della cella.

I timer sono posizionati in una delle liste dal thread sender nel momento in cui il pacchetto viene mandato. Il timer della cella della struttura di Selective Repeat è infatti impostato come *nextTimer* della testa della lista collegata opportuna, e ad esso vengono collegati tutti gli eventuali timer già presenti in lista mediante un semplice passaggio di puntatori. All'arrivo di un ACK, il Listener deasserisce *isValid*.

La durata del timer è rappresentata dal numero di celle di offset dalla corrente posizione della "lancetta" della Timer Wheel ed è ricalcolata ad ogni invio in base a misurazioni dei Round Trip Time effettivamente rilevati durante le trasmissioni, senza mai andare però a modificare la lista su cui il thread timer sta attualmente lavorando.

In fase di inizializzazione dello scambio di dati il valore del campo*nextTimer* di ciascun *headTimer* è un puntatore a NULL. La posizione corrente a cui punta la Wheel del thread Timer è definita dalla variabile globale *currentTimeSlot*.

Nel momento dell'ispezione della cella, viene controllata la lista di timer. Se il valore di validità del timer è uguale ad 1, viene notificata al sender la ritrasmissione scrivendo su una pipe il numero di sequenza del datagramma da ritrasmettere. A questo punto si passa attraverso il puntatore *nextTimer* alla struct successiva e si esce dal ciclo while solo nel caso in cui questo punti a NULL.

I timer della ruota sono impostati nella posizione più accurata possibile

[1] TPacket, nella figura

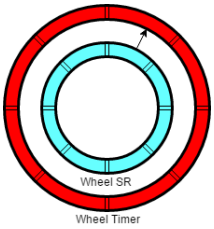


Figure 4.3: Correlazione tra le ruote di timer e selective repeat

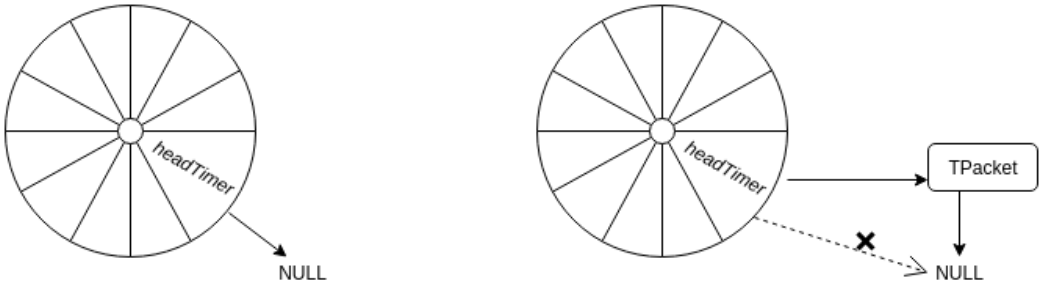


Figure 4.4: Timer wheel

perchè durante l'esecuzione del programma è attivo un meccanismo di campionamento del Round Trip Time. Questo permette di aggiornare continuamente la stima del giusto offset da utilizzare all'interno della timing wheel, e rende l'applicazione adattabile a cambiamenti delle condizioni di rete anche durante l'esecuzione.

Si può notare come, in conclusione, come il thread timer si limiti all'esplorazione di una catena di timer e a scritture su una pipe in caso di pacchetto non riscontrato. Ciascuna operazione di Listener e Sender è pensata per arrecare il minor rallentamento possibile al Timer, accedendo quindi alle strutture già inizializzate soltanto per cambiare il valore di un campo.

4.3 Connessione e Disconnessione

Si conclude questo capitolo analizzando la procedura di connessione di client e server. Questa fase è cruciale perchè in essa si dispiega l'allocazione delle risorse ed avviene il multithreading che permette la corretta esecuzione dell'applicazione.

Si è scelto di seguire il modello dettato dal three-way-handshake proposto

da TCP. Tuttavia, operando il trasferimento di dati su due diverse socket, il procedimento appare più lungo poichè devono avvenire due diversi handshake prima di poter ritenere conclusa la fase di connessione.

Durante la fase di connessione avviene lo scambio di informazioni tra client e server che permette di conoscere il numero di sequenza remoto utilizzato, oltre che di salvare i dati dell'indirizzo di destinazione a cui ogni socket fa riferimento.

A seguito della richiesta di connessione sul server principale (messaggio di SYN), il server effettua infatti una fork per creare il processo servente dedicato. Questo, utilizzando i dati forniti dal processo padre, risponde al client con una socket dedicata creata appositamente. Il primo handshake avviene su questa socket ed è quello deputato a connettere il thread sender del client e listener del server.

Una volta concluso il primo handshake, il thread listener del client comincia una nuova connessione seguendo lo stesso schema, ma usando l'indirizzo della prima socket dedicata al posto di quello del main server. Il server dedicato apre una nuova socket dedicata mediante la quale viene portato a termine anche il secondo handshake. Alla fine del processo le due socket connettono parallelamente i thread listener con i sender remoti.

Durante la fase di connessione viene rispettato il requisito di comunicazione affidabile. Tuttavia, non essendo ancora concluso il processo di allocazione delle risorse e di funzionamento fisiologico, si utilizza un algoritmo leggermente diverso da quello descritto nella sezione precedente. Il thread timer infatti non notifica le ritrasmissioni al sender ma al thread che sta effettuando la connessione nel momento in cui si verifica la scadenza del timer di pacchetto.

A seguito della corretta terminazione della procedura, i due processi remoti si trovano in una condizione di corretta allocazione di risorse. Inoltre è avvenuta correttamente l'impostazione del corretto destinatario della comunicazione e dei numeri di sequenza remoti.

La procedura di disconnessione regolare è avviata dal client con il comando *quit*. il client e il server a questo punto operano un ultimo scambio di messaggi per notificare la ricezione del comando e provvedono alla deallocazione delle risorse, alla chiusura dei file rimasti aperti e all'interruzione del processo tramite un *pthread_join* che permette l'interruzione in contemporanea dei thread listener e sender.

5. Comandi e trasmissione dati

In questo capitolo verranno esaminate progettazione ed implementazione delle funzioni che permettono il trasferimento di file tra terminali remoti. Ogni paragrafo sottostante esaminerà nel dettaglio ciascuna fase dell'esecuzione.

Nell'implementazione dell'applicazione è stata ritenuta di primaria importanza la linearità e la semplicità dell'esecuzione dei diversi thread. Si è tentato di rendere ciascuna esecuzione il meno complessa possibile cercando di rendere asincrona la comunicazione tra thread. Sono state evitate attese non necessarie o non utili ai fini dei comandi rendendo globali le risorse necessarie all'esecuzione e ponendo particolare attenzione alla consistenza e congruenza di queste.

Analizziamo nel dettaglio le fasi dei processi client e server e le scelte implementative.

5.1 Operazioni

L'applicazione permette lo scambio di dati facendo comunicare il client con il server per attraverso tre diverse operazioni :

- LIST
- PULL
- PUSH

I primi due comandi hanno un funzionamento molto simile tra loro, cambiando solamente nella fase di controllo. Per fase di controllo si intende

il primo scambio di pacchetti necessario a impostare correttamente l'ambiente di invio e ricezione (apertura o creazione di file e comunicazione della dimensione dell'oggetto da inviare).

Come si è visto nel capitolo riguardante l'implementazione della trasmissione affidabile, per ogni pacchetto inviato è necessario aspettare un acknowledgement che può essere ricevuto in un pacchetto di tipo *handshake* o attraverso un *datagram* insieme a parte del contenuto richiesto (come nel caso dell'operazione LIST).

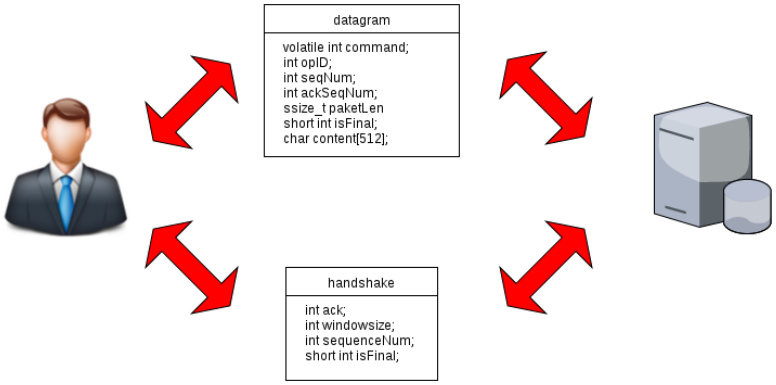


Figure 5.1: Scambio di pachetti

Lato client, il thread Listener è inizialmente in ascolto sullo standard input. Per ciascuna richiesta arrivata, viene eseguito il parsing del comando inserito e, qualora non siano riscontrati errori, viene parzialmente impostato il contenuto del pacchetto da inviare. A seguito di questa operazione viene inviato un segnale `pthread_cond_signal(pthread_cond_t *cond)` al Sender che procede allo svolgimento del comando.

A questo punto il Listener attende l'opportuna risposta dal server, mediante la funzione `listPullListener` o `pushListener`, che monitora la socket in modo continuo fino al termine della trasmissione. L'operazione eseguita in ricezione varia a seconda del comando eseguito. Per ogni datagramma viene aggiornata una pipe contenente i numeri di sequenza dei pacchetti di cui mandare riscontro, e viene riempito il file in modo opportuno a seconda del comando. Per ogni ACK ricevuto viene invece aggiornata la finestra di *Selective Repeat* e fermato il timer relativo al pacchetto riscontrato.

Il funzionamento del lato server è analogo a quello appena descritto, con l'eccezione di un dettaglio: il thread Listener è continuamente in ascolto sulla propria socket, invece che di input dall'utente.

Analizziamo il funzionamento generale delle singole operazioni :

LIST — Riporta l'elenco completo dei file presenti nella directory del server specificata. È identificata dal numero di comando 0.

PULL — Permette di salvare in locale la copia di un file presente nel server. È identificata dal numero di comando 2. Il client richiede un file specificandone il nome e riceve dal server tutti i pacchetti contenenti il file, di cui il client invia riscontri.

PUSH — Consente di salvare una copia di un file locale sul server, dentro la stessa directory impostata per l'operazione LIST. È identificato dal numero di comando 2. Il client invia l'informazione riguardante il nome del file da creare e attende il riscontro. Una volta avvenuto, invia pacchetti contenenti il file in modo speculare all'operazione PULL. Il server in questo caso ha l'unico scopo di salvare il contenuto dei pacchetti e riscontrarli.

Sono disponibili altre due operazioni ausiliare al programma: HELP e QUIT.
La prima agisce in locale stampando a schermo i possibili comandi eseguibili dall'utente e spiegandone brevemente lo scopo. La seconda avvia i procedimenti di chiusura della connessione descritti nel capitolo precedente.

5.2 Lettura da socket e pipe

In questa sezione verrà posto l'accento sul meccanismo mediante il quale i thread accedono in lettura alle socket e alle pipe.

Di default, il sistema operativo ha infatti un'implementazione delle chiamate di lettura sui file descriptor di tipo bloccante. Ciò implica che, in lettura da una socket, il thread listener rimanga bloccato fino alla ricezione di dati utili. Questo accade anche per il thread sender in occasione della lettura sulla pipe di notifiche di ritrasmissioni e/o di acknowledgement da inviare.

Un meccanismo del genere però non è risultato affatto conveniente in un contesto in cui i thread dovevano essere continuamente sincronizzati e in comunicazione tra di loro, oltre che intenti a monitorare più di una fonte di informazioni. Per raggiungere una maggiore robustezza ed integrazione, si è

deciso quindi di modificare l'accesso ai file descriptor detti sopra per renderlo non bloccante. In questo modo i thread non restano bloccati ma effettuano un meccanismo di polling continuo dalle diverse fonti da monitorare.

Questa scelta implementativa ha anche reso più robusta gestione di timer che interrompessero l'esecuzione del programma in caso di inattività prolungata. L'implementazione dei segnali infatti non è thread-safe, come riportato in [Ker10], quindi non è stato possibile gestire in modo corretto un sistema che utilizzasse il segnale *alarm* per interrompere l'esecuzione.

5.3 Ritrasmissione

UDP è un protocollo del livello di trasporto impiegato solitamente per la trasmissione di informazioni real-time, come le chiamate VoIP, dove la velocità della comunicazione è preferita alla affidabilità della stessa. Per questo motivo non viene gestita dal protocollo alcuna forma di riordinamento e ritrasmissione di pacchetti. La principale richiesta che si fa ad una applicazione client-server è tuttavia la correttezza. Ci si aspetta che qualunque file venga trasferito senza errori.
In questa sezione viene spiegato come, a livello software, si sopperisce alle mancanze del protocollo UDP per lo scambio affidabile di pacchetti.

Il thread Sender di un processo deve inviare delle ritrasmissioni solamente nel caso in cui stia trasmettendo dei datagrammi. Prima di inviare ciascun pacchetto, viene controllata la pipe dedicata alle ritrasmissioni che mette in comunicazione Timer e Sender. Nel caso in cui ci sia un pacchetto da ritrasmettere, viene ricostruito il contenuto andato perduto (attraverso la funzione `rebuildDatagram`) e viene inviato nuovamente.
Ciascun datagramma ritrasmesso aggiorna la finestra di trasmissione e fa partire nuovamente un timer legato al pacchetto.

La funzione che ricostruisce il contenuto del pacchetto lo fa in modo efficiente, utilizzando lo stesso meccanismo con cui in remoto avviene la ricostruzione del file dalle porzioni ricevute: anche in questo caso, a meno che non si tratti della ritrasmissione del primo pacchetto contenente il comando da eseguire, è necessario solamente sapere il numero di sequenza del pacchetto da ritrasmettere e del primo della sequenza. Il pacchetto contenente il comando è invece opportunamente salvato per permetterene una rapida ritrasmissione senza alcun tempo di overhead.

Come si evidenzierà dai test di esecuzione, la presenza di notevoli ritrasmissioni non comporta un incremento delle risorse richieste dal programma, grazie all'efficienza dell'implementazione appena esaminata.

5.4 Gestione degli errori

Ciascun errore che avviene durante l'utilizzo del software viene gestito dipendentemente dalla sua tipologia e dallo stato di esecuzione dell'operazione.

Viene definita *errore preliminare* una anomalia che si verifica prima della fase di controllo[1] della operazione.
In questo stadio, il client impedisce all'utente di contattare il server, informandolo del motivo dell'interruzione.
Le più ricorrenti occorrenze di questo tipo di errori sono l'inserimento errato del nome dell'operazione e la richiesta di push per un file che non è presente sul proprio sistema.

```
insert command :
test
command does not exist, enter 'list', 'push', 'pull', 'help' or 'exit'
insert command :
push test
error in open:  No such file or directory
insert command :
...
```

Leggermente più complessa è la gestione degli errori che possono avvenire durante lo scambio di pacchetti tra client e server. Ciascun terminale deve esser pronto a terminare in sicurezza l'operazione senza minare la stabilità della connessione, in modo da poter rieseguire il comando in un secondo momento nella stessa sessione.

In caso di ricezione di un pacchetto con il campo *isFinal* posto a -2, viene avvisato prontamente il sender che interrompe l'invio di qualunque tipo di informazione. Entrambi i thread tornano quindi nello stato di attesa di comando.
Questo perchè si è ricevuto un pacchetto che notifica un errore avvenuto

[1] Si ricorda che "per fase di controllo si intende il primo scambio di pacchetti necessari ad impostare correttamente l'ambiente di invio e ricezione"

nell'altro terminale. D'altra parte, lo stesso comportamento si riscontra dal lato di chi ha inviato il messaggio di errore.

Un pacchetto che ha il compito di informare l'altro terminale di una anomalia nell'operazione e, essendo un messaggio di errore che implica il termine dell'operazione corrente, non deve essere riscontrato.

Può quindi accadere, ad esempio, che il client richieda l'interruzione dello scambio di messaggi e il pacchetto si perda. In questo caso il client tornerebbe a chiedere all'utente ulteriori comandi mentre il server sarebbe ignaro di tutto.
Questo inconveniente è gestito grazie a un timer che monitora il tempo che intercorre tra l'arrivo di un pacchetto e un altro. Nel caso ci fosse un silenzio troppo lungo, si chiude l'operazione esattamente come si farebbe in caso di ricezione di un pacchetto di errore.

Nel caso in cui il server stia inviando pacchetti (come avviene per una pull o una list), il client, ricevendo una ritrasmissione inaspettata, invia nuovamente la richiesta di interruzione notificando l'errore.

```
Request received, command = 2          insert command :
    pull :  test                           pull test
    sprintf result                         | path :
    /home/user/directory/test              /home/user/Directory/test
    1:  error on open file,                | file created with name :
    retransmission:  No such file          test
    or directory                           error
    exit_failure (sendCycle)               exit_failure (sender)
    error
    Waiting for commands                   insert command :
                                           ...
```

Prima di terminare questa sezione è necessario appuntare che esiste un terzo tipo di errore.
gli *errori non bloccanti*, come si può intuire dal nome, sono anomalie che non interrompono lo scambio di pacchetti e che permettono di portare a termine l'operazione, non sempre con i risultati sperati. È un esempio il raro caso in

cui il sistema non riesca a leggere (o scrivere) dal file precisamente 512 byte. Il terminale che riceve un pacchetto che trasporta una quantità inferiore di dati non altera il proprio flusso di esecuzione.

6. Esempi

In questa sezione vengono mostrate alcune esecuzioni del programma, iniziando dalle fasi iniziali di configurazione e portando esempi di ciascun comando[1].

- Avvio del programma

```
  /xxxxxx/DFT$ make all
 /xxxxxx/DFT$ ./server                    /xxxxxx/DFT$ ./client
*----------------------------*
un client vorrebbe                       insert command :
connettersi
*----------------------------*
richiesta dal client
127.0.0.1
Received packet from
127.0.0.1, port 47083
Second data stream from
127.0.0.1, port 51192

Waiting for commands
```

[1]Rispetto alla versione finale della applicazione potrebbero essere variate alcune stampe

- LIST

```
  Request received,
command = 0
isFinal = -1 sent :  3595
Waiting for commands
```

```
  insert command :
  list
----------LIST-----------

file1
file2
file3
.
.
.

--------------------------

insert command :
```

- PULL

```
  Waiting for commands
Request received, command = 2
pull :  d.png
sprintf result
/home/xxxxxx/d.png
isFinal = -1 sent :  3407
Waiting for commands
```

```
  insert command :
  pull d.png
| path :  /home/xxxxxx/d.png
| file created with name :
d.png

--------------------------
| operazione completata in
104 millisecondi
| dimensione del file:  346
kB
| velocità media:  3333.001
MB/s
--------------------------

346708,104,0
insert command :
```

- PUSH

```
  Waiting for commands                    insert command :
Request received, command = 1             push /home/xxxxxx/d.png

push                                      --------------------------
file to open:                             | operazione completata in
/home/xxxxxx/d.png                        108 millisecondi
                                          | dimensione del file:  346
Waiting for commands                      kB
                                          | velocità media:  3210.001
                                          MB/s
                                          --------------------------

                                          346708,108,0
                                          insert command :
```

7. Testing

A seguito della fase di sviluppo del software in esame, una notevole attenzione è stata rivolta all'ideazione di strumenti che permettessero di testare in modo corretto e completo le funzionalità del programma esemplificate nei capitoli precedenti.

Si è quindi scelto di rendere possibile per l'utente la scelta di molti parametri che possono influenzare il rendimento dell'applicazione. In fase di compilazione si può impostare infatti:

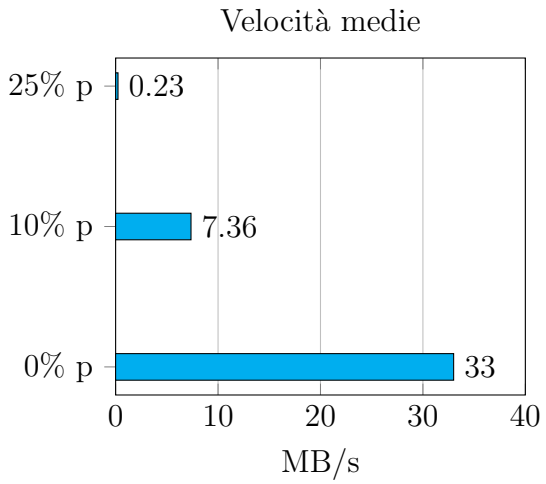
finestra di spedizione Cambiando questo valore viene modificato il numero di pacchetti che possono essere inviati in pipeline prima che ci sia un riscontro

probabilità di perdita Questo valore rappresenta la probabilità, in millesimi, di perdere un pacchetto inviato in rete. In fase di invio vengono generati numeri distribuiti randomicamente tra 1 e 1000 e viene scartato un numero di pacchetti proporzionale alla probabilità impostata

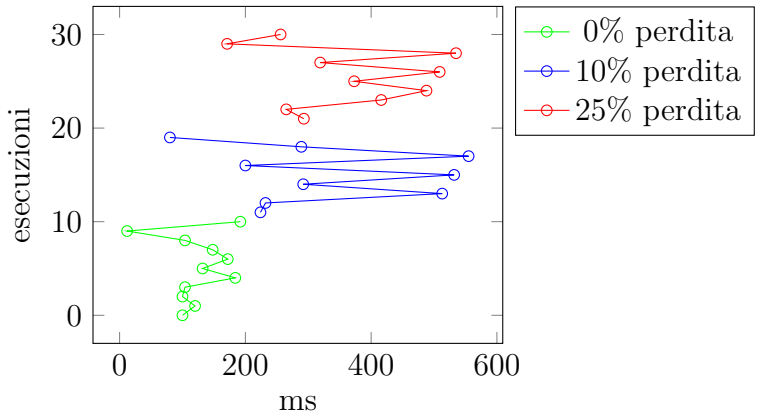
precisione del timer Indica la durata di una singola cella del timer, e comporta di conseguenza la durata di un intero ciclo della timer wheel.

default timer Questo valore rappresenta il minimo valore a cui può essere impostato un timer, espresso in numero di slot della timer wheel. La sua durata è dipendente dalla precisione del timer.

Qui di seguito viene riportato un test eseguito su un campione di 50 operazioni *PULL* eseguite in locale.



Dal seguente grafico vengono invece messi in primo piano i tempi di completamento per 30 esecuzioni effettuate con diverse probabilità di errore.



Tutti i dati presentati sono stati raccolti in modo automatico grazie al processo client, che popola dopo ogni operazione un file in formato CSV. Questo file è di facile utilizzo e permette di poter eseguire autonomamente e rapidamente ulteriori test di funzionamento per l'applicazione. Il file presenta, in quest'ordine, i seguenti campi:

1. Dimensione del file in Kb
2. Tempo di esecuzione in millisecondi
3. Probabilità di perdita in millesimi

4. Dimensione della finestra di spedizione
5. Durata del Round Trip Time misurata in media

8. Configurazione

Al momento è stato possibile testare il software soltanto in loopback oppure all'interno di una rete locale. Per eseguire client e server sullo stesso terminale è quindi necessario accedere a due console differenti e posizionarsi all'interno della directory in cui sono presenti i file sorgenti dell'applicazione.

E' possibile compilare tutti i sorgenti mediante l'esecuzione del makefile presente, chiamando il comando `make all`. Qualora si desideri modificare i parametri dell'applicazione di cui si è parlato nel capitolo precedente, è necessario modificare tale file con i valori opportuni.

Viene riportata qui di seguito la copia del makefile. Sono stati evidenziati in blu gli elementi da modificare assolutamente prima di compilare.
LSDIR corrisponde alla directory di riferimento per il server, utilizzata per le funzioni list e push.
PULLDIR corrisponde invece alla directory di riferimento per il client, utilizzata per la funzione pull.
Sono state inoltre evidenziati in rosso i valori di default relativi alla finestra di selective repeat, timer, porta e probabilità di perdita.

```
CC=gcc
CFLAGS=-Wall -Wextra -pthread -DNANOSLEEP=10000 -DWINDOWSIZE=2048
-DTIMERSIZE=2048 -DLOSSPROB=0 -DBASETIMER=200 -DPULLDIR="/home/pulldir/"
-DLSDIR="/home/lsDir/" -DPORT=4242 -I.

all:  client server clean

client:  client.o dataStructures.o
$(CC) -Wall -Wextra -pthread -DNANOSLEEP=10000 -DWINDOWSIZE=2048
-DTIMERSIZE=2048 -DLOSSPROB=0 -DBASETIMER=200
```

```
-DPULLDIR="/home/pulldir/" -DPORT=4242 -o
client client.o dataStructures.o -I.

server:  mainServer.o dataStructures.o server.o
$(CC) -Wall -Wextra -pthread -DNANOSLEEP=10000 -DWINDOWSIZE=2048
-DTIMERSIZE=2048 -DLOSSPROB=0 -DBASETIMER=200
-DLSDIR="/home/lsDir/" -DPORT=4242 -o server server.o mainServer.o
dataStructures.o -I.

clean:
rm *.o
```

Per l'esecuzione del software non è necessario alcun privilegio di root.

9. Limiti e potenzialità

La progettazione e l'implementazione di questa applicazione client-server è frutto di diversi mesi di lavoro.
La comunicazione tra i thread, gli algoritmi per l'esecuzione dei comandi, le varie strutture e, in generale, ciascun altro aspetto del software, è stato pensato molto prima di scrivere le prime linee di codice.
Anche i passaggi che possono sembrare semplici sono stati messi in discussione più volte al fine di rendere il più semplice o armonioso possibile il funzionamento dell'applicazione.
Questo non significa però che il software sia nella sua forma definitiva. Possono essere implementate numerose migliorie in grado di rendere più intuitiva e utile l'esperienza dell'utente e migliorare notevolmente la stabilità del sistema.

Un primo limite di funzionamento è la gestione efficiente di file di grandi dimensioni. L'eccessiva velocità di trasmissione di grosse moli di dati fa infatti sì che possano esserci lievi difetti nella sincronizzazione tra thread, così che alla fine della trasmissione il file non risulti ricostruito in modo totalmente corretto, nonostante non si siano rilevati errori.

Un secondo limite che potrebbe portare ad un blocco dell'applicazione è la mancanza di adattività della ruota dei timer qualora fossero rilevali Round Trip Time superiori alla sua dimensione totale. Un RTT troppo lento comportarebbe un problema in quanto le ritrasmissioni prenderebbero completamente il sopravvento sulla normale comunicazione.

La stabilità del software è inoltre carente in presenza di un numero eccessivo di connessioni, sia involontarie che derivanti da attacchi informatici di tipo Denial-Of-Service. QUeste numerose richieste di connessione generano un altro possibile problema: all'arrivo di un nuovo client infatti viene prontamente chiamata la funzione `fork()`da parte del server e anche il più semplice attacco DoS può trasformarsi in una sorta di *Fork Bomb*. // Sarebbe necessario,

per questioni di sicurezza, eseguire un controllo dei file inseriti nel server attraverso il comando PUSH.
In caso di ricezione di un file con un nome già esistente nel server, quest'ultimo viene sovrascritto. Questa scelta implementativa, unita allo stato temporaneo del server che prevede una singola directory comune, può permettere ad un client con intenzioni malevole di eliminare ogni contenuto salvato da altri utenti.

Analizzeremo invece ora un possibile scenario di sviluppo dell'applicazione:

Un limite che deve essere superato è la totale assenza di directory diverse per ciascun utente.
Attualmente ogni sessione è indipendente dall'altra e non è previsto alcun tipo di riconoscimento del client. La directory viene semplicemente condivisa tra tutti coloro che utilizzano il software e non è presente alcuna protezione dei dati.
In un possibile aggiornamento dell'applicazione ciascun utente potrà operare su una personale directory protetta mediante meccanismi di autenticazione.

10. Strumenti utilizzati

Nello sviluppo di questa applicazione è stato fatto ricorso a diversi software, non solo per la scrittura del codice, ma anche per la sincronizzazione del gruppo e per la fase di testing e debugging. È stato utilizzato il servizio di hosting **GitHub** attraverso il quale è stata resa semplice la cooperazione tra i membri del gruppo ed è possibile esaminare passo per passo i progressi effettuati nel codice. Per la scrittura dei file C e la loro organizzazione è stata scelta la IDE **CLion**[1], scegliendola proprio per l'integrazione con questo servizio.
Per la scrittura e il testing di funzioni e appunti riguardante il codice si è scelto di utilizzare l'editor **Sublime Text**[2], sfruttando l'enorme manegevolezza e semplicità d'uso che questo software offre.

Il progetto è stato sviluppato interamente in ambiente Linux utilizzando da terminale il software **GCC** per la compilazione sia dell'intero software che delle singole funzioni.
In fase di testing è stata impiegata la funzione **strace**, il debugger **gdb** e il tool **netem** per emulare reti WAN in locale.

La relazione è stata scritta utilizzando il linguaggio **Latex**, e per facilitare la cooperazione tra i membri del gruppo si è scritto attraverso l'editor online **ShareLatex**.

[1] JetBrains CLion versione 2.2
[2] Sublime Text versione 1

Bibliography

[P181a] J Postel and rfcmarkup version 1. *Internet protocol*. Sept. 1981. URL: https://tools.ietf.org/html/rfc791 (visited on 12/07/2016).

[P181b] J Postel and rfcmarkup version 1. *Transmission control protocol*. Sept. 1981. URL: https://tools.ietf.org/html/rfc793 (visited on 12/07/2016).

[Ste90] Richard W Stevens. *UNIX network programming*. 15th ed. New York, NY, United States: Prentice Hall PTR, Jan. 1990. ISBN: 9780139498763.

[Ker10] Michael Kerrisk. *The Linux programming interface: A Linux and UNIX system programming handbook*. San Francisco, CA: No Starch Press San Francisco, CA, Oct. 2010. ISBN: 9781593272203.

[Sar+11] Matt Sargent et al. *Computing TCP's Retransmission timer*. June 2011. URL: https://tools.ietf.org/search/rfc6298 (visited on 12/07/2016).

[KR12] James F. Kurose and Keith W. Ross. *Computer networking: A top-down approach*. 6th ed. Boston: Addison-Wesley Educational Publishers, Feb. 2012. ISBN: 9780132856201.

[VL] George Varghese and Tony Lauck. *Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility*. Tech. rep. URL: https://pdfs.semanticscholar.org/0a14/2c84aeccc16b22c758cb57063fe227e83277.pdf (visited on 12/07/2016).