Reliable UDP File Transfer

RUFT

Relazione del progetto di Ingegneria di Internet e Web
A.A. 2018/2019

Gianmarco Bencivenni
0218205

Cristina Ionne
0224462

Indice

1. Specifiche di progetto

Lo scopo di questo progetto è quello di implementare, in linguaggio C sfruttando l'API del socket di Berkeley, un'applicazione client-server per il trasferimento di file che impieghi il servizio di rete senza connessione, ovvero sfruttando il protocollo UDP a livello di trasporto, ma pur sempre garantendo affidabilità per quanto riguarda lo scambio di dati.

Infatti, questo protocollo, oltre al servizio di multiplexing/demultiplexing e ad una forma molto semplice di controllo (e non di correzione) degli errori non aggiunge quasi nulla ad IP: UDP non fa altro che prendere i messaggi a livello applicativo, vi aggiunge un numero di porta di origine e destinazione e altri due piccoli campi, infine passa il segmento risultante al livello di rete.

Numero di porta di origine	Numero di porta di destinazione
Lunghezza	Checksum
Dati dell'applicazione (messaggio)	

Struttura dei segmenti UDP

Dunque, il nostro obiettivo è quello di progettare un sistema di comunicazione che ci permetta di ottenere insieme **affidabilità**, **velocità di trasferimento** e **semplicità d'implementazione**.

Il software da implementare garantirà le seguenti funzioni:

1. Connessione client-server senza autenticazione;
2. Visualizzazione da parte del client dei file disponibili sul server (comando *list*);
3. Download da parte del client di uno o più di tali file (comando *get*);
4. Upload di file sul server da parte del client (comando *put*);
5. Trasferimento di file affidabile.

La comunicazione tra client e server sarà gestita da un opportuno protocollo in grado di scambiare due tipologie di messaggi:

- messaggi di comando: inviati dal client al server per richiedere l'esecuzione di una delle tre diverse operazioni;
- messaggi di risposta: inviati dal server al client in risposta ad un comando con l'esito dell'operazione.

Funzionalità del Server

Il server, di tipo concorrente e in continuo ascolto su una porta configurabile, sarà in grado di offrire le seguenti funzionalità:

1. Invio del messaggio di risposta al comando *list*: il messaggio conterrà la "filelist", ovvero la lista di nomi dei file disponibili sul server per la condivisione;

2. Invio del messaggio di risposta al comando *get* contenente il file richiesto: se presente, il file sarà inviato al client, altrimenti sarà inviato un opportuno messaggio di errore;

3. Ricezione di un messaggio di richiesta relativo al comando *put* contenente il file da caricare sul server e invio del messaggio di risposta con l'esito dell'operazione (ACK).

Funzionalità del Client

Il client, anch'esso di tipo concorrente, sarà in grado di offrire le seguenti funzionalità:

1. Invio del messaggio di richiesta *list* per visualizzare a schermo la lista di nomi dei file disponibili sul server;

2. Invio del messaggio di richiesta *get* per eseguire il download di uno di tali file;

3. Ricezione di un file richiesto tramite comando *get* o la gestione dell'eventuale errore;

4. Invio del messaggio di richiesta *put* per eseguire l'upload di un file sul server e ricezione del messaggio di risposta con l'esito dell'operazione (ACK).

Trasferimento affidabile

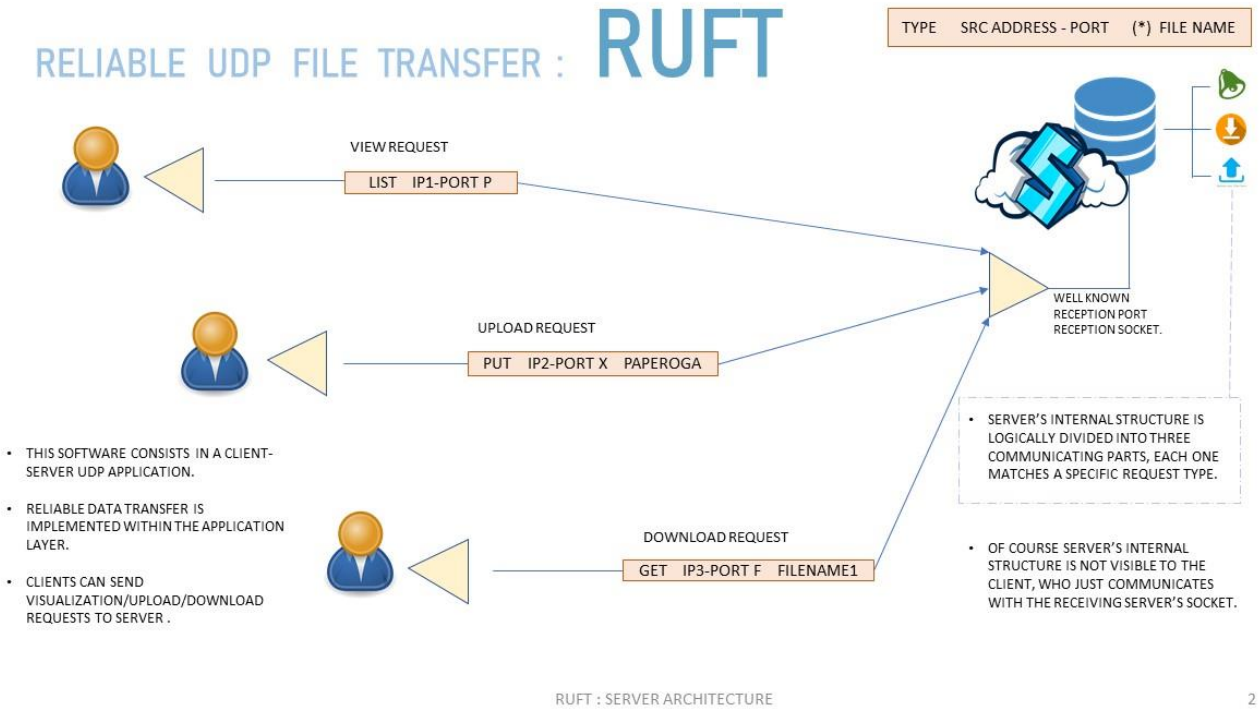
Lo scambio di messaggi avrà luogo tramite un servizio di comunicazione non affidabile, dunque al fine di garantire una corretta trasmissione/ricezione di dati, client e server implementeranno a livello applicativo il protocollo *selective repeat* con finestra di trasmissione di dimensione N e durata del timeout T (cfr. Kurose & Ross "Reti di Calcolatori e Internet", 7° Edizione, sez. 3.4.4, pagg. 212-217).

Per simulare la perdita dei messaggi in rete (evento molto improbabile su rete locale e ancor più improbabile se client e server sono eseguiti sullo stesso host), si assumerà che ogni messaggio sia scartato dal mittente con probabilità p.

La dimensione della finestra di trasmissione N, la probabilità di perdita dei messaggi p e la durata del timeout T, saranno tre costanti configurabili ed uguali per tutti i processi.

Oltre all'uso di un timeout fisso, dovrà essere possibile scegliere un valore per il timeout adattativo calcolato dinamicamente in base all'evoluzione dei ritardi di rete osservati.

2. Architettura del Sistema



Il cuore del progetto risiede proprio nell'architettura del RUFT Server.
Come da specifica, i client possono indirizzare alla porta nota del RUFT Server, configurata manualmente nel codice sorgente, messaggi di richiesta di tipo *list*, *get* e *put*.
Il Server accoglie continuamente le richieste su una "socket di benvenuto" per poi delegare il servizio a specifici processi e thread.

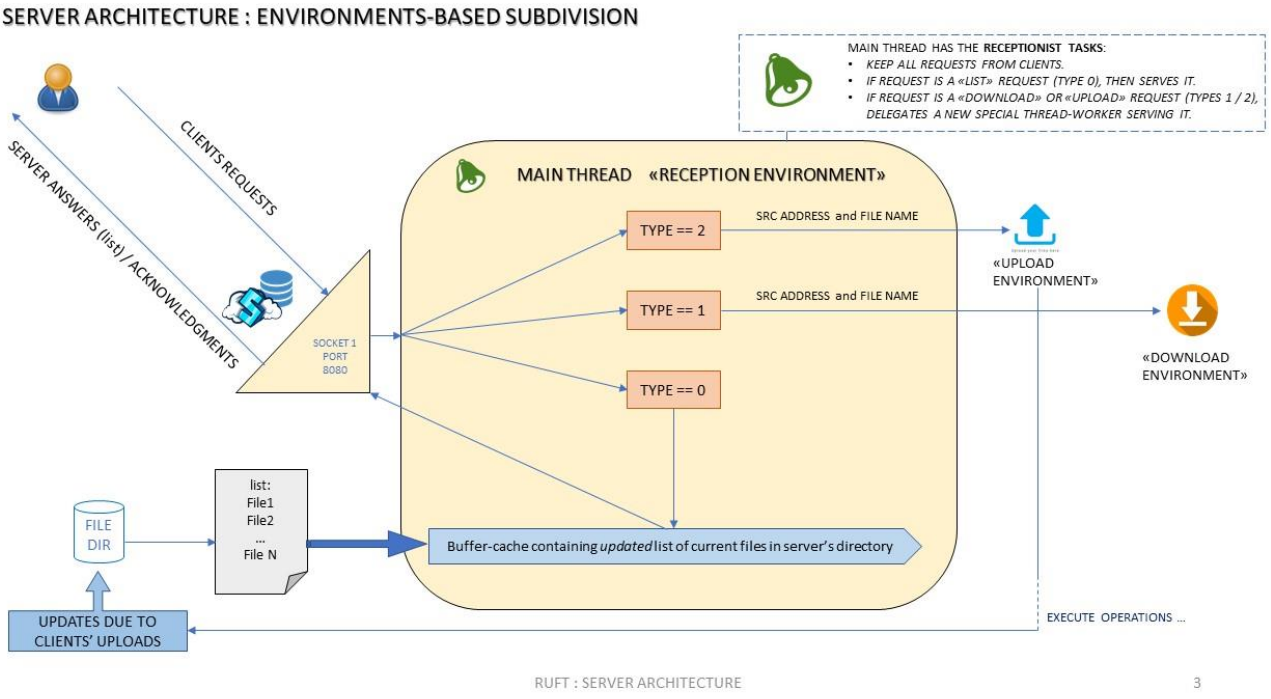
Architettura del Server RUFT

Il RUFT Server si basa su una suddivisione, a livello logico e strutturale, in "Ambienti di Lavoro" (Environments).
L'RDD (*Responsibility Driven Design*) è una tecnica di progettazione, proveniente dal mondo dell'object-orientation, la quale prevede che ogni Classe (o entità) rivesta un ruolo ben definito e svolga i propri compiti in un dominio limitato, minimizzando il più possibile la sua interferenza sui domini coesistenti nell'applicazione.
Sebbene il linguaggio di programmazione, così come l'approccio all'implementazione, siano in questo caso di tipo procedurale, l'applicazione dei princìpi di *progettazione per responsabilità* sull'architettura del software si è rivelata una scelta fondamentale, permettendo una corretta modellazione degli Environments e uno sviluppo software a "compartimenti stagni", indipendenti gli uni dagli altri.

Si presentano a seguire gli Environment costituenti il RUFT Server:

- Reception Environment
- Download Environment
- Upload Environment

Reception Environment



All'arrivo di un nuovo messaggio da parte di un client sulla socket di benvenuto, il Server deve innanzitutto riconoscere la tipologia di richiesta ad esso associata.
Il Reception Environment si occupa della ricezione delle richieste e della loro demultiplazione: nel caso di richieste di tipo *get* (download di file) o di tipo *put* (upload di file), la responsabilità del servizio viene delegata rispettivamente al Download Environment o all'Upload Environment.
Invece, nel caso in cui la richiesta sia di tipo *list*, sarà il main thread dell'Upload Environment (il "Receptionist") ad inviare la filelist aggiornata al client, elencando quindi l'attuale contenuto della directory del server.

Implementazione

Al lancio dell'applicazione, il main thread apre una sessione sulla socket di benvenuto rimanendo in attesa di messaggi di richiesta da parte dei client.

- Il canale di I/O è della famiglia di indirizzi AF_INET, per la comunicazione attraverso indirizzi Internet IPv4.
- Per usufruire, a livello di trasporto, del protocollo UDP (User Datagram Protocol) senza connessione, si è specificato SOCK_DGRAM nel campo "type" della funzione *socket*.

- Il terzo campo di *socket*, posto uguale a 0, indica l'utilizzo del protocollo di default (che è proprio UDP) a partire dai primi due parametri.

In seguito, viene caricato in memoria RAM, tramite la funzione *load_*dir, il contenuto corrente della directory del server, così da permettere il trasferimento rapido dei dati informativi all'arrivo di richieste di tipo *list*.
Ad ogni nuovo upload, tale contenuto viene aggiornato anche in memoria principale, in modo da non generare casi di inconsistenza dei dati sulle richieste di visualizzazione.

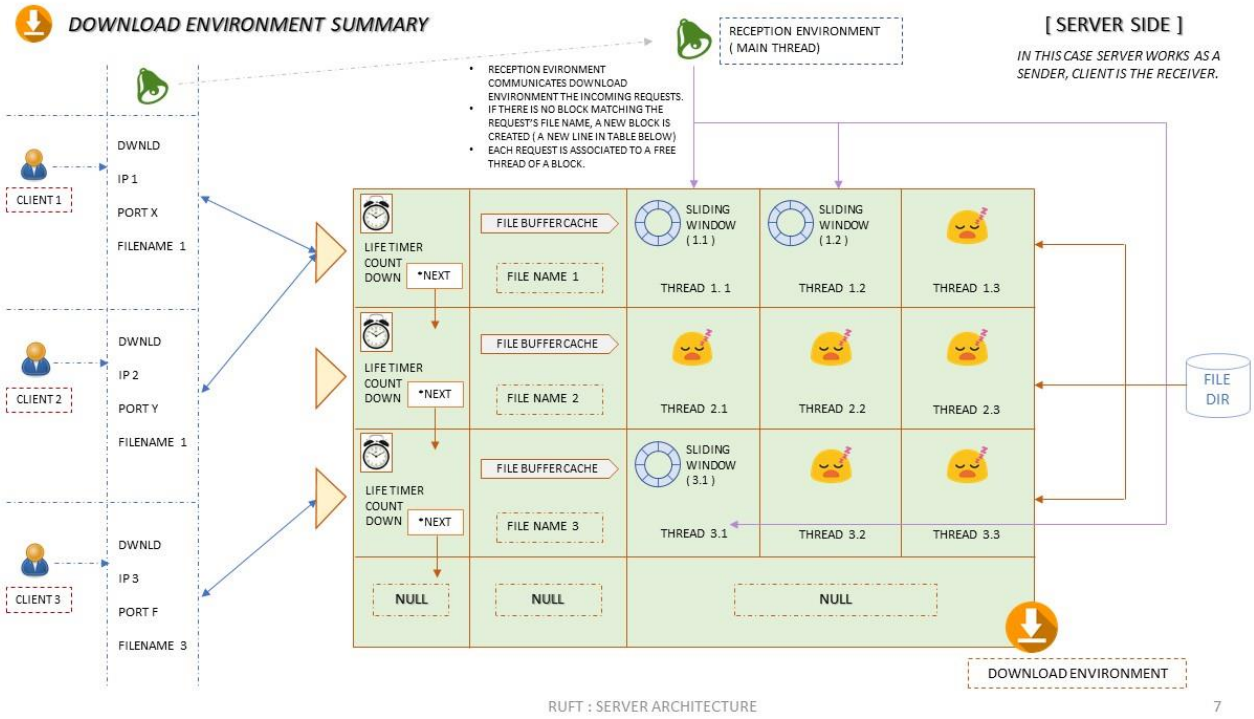
Successivamente, tramite la funzione *bind*, viene eseguita l'associazione "on-demand" dell'indirizzo del server alla socket creata: per la configurazione dell'indirizzo server viene utilizzata una porta (che sarà poi la porta nota del RUFT Server) configurabile direttamente dai file *UDP_Server.c e UDP_Client.c* .

```
/* Load the current directory list on RAM to be ready for using.  */
printf( "\n RUFT SERVER TURNED ON.\n Loading directory file list on buffer cache..." );
int dir_size = current_dir_size();
list = load_dir( dir_size );
printf("Done.\n Current directory content is :\n\n%s\n", list );        fflush(stdout);


/* Creating socket file descriptor */
if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
    perror("socket creation failed");
    exit(EXIT_FAILURE);
}

memset(&servaddr, 0, sizeof(servaddr));

/* Filling server information  */
servaddr.sin_family      =    AF_INET;                    // IPv4
servaddr.sin_addr.s_addr =    INADDR_ANY;                 // 127.0.0.1
servaddr.sin_port        =    htons(PORT);                // Well-known RUFT Server port.

/* Bind the socket with the server address */
if ( bind(sockfd, (const struct sockaddr *)&servaddr, sizeof(servaddr) ) < 0 ) {
    perror("bind failed");
    exit(EXIT_FAILURE);
}
```

Ad ogni nuovo messaggio di richiesta, dunque, il main thread lancia un thread per la gestione del cliente e per direzionare adeguatamente il servizio richiesto.
Dunque, possiamo dire che il Reception Environment, o Ambiente di Ricezione, è formato complessivamente dal main thread e da tutti i threads ("receptionists") concorrenti nel sistema.

Il Reception Environment è responsabile, inoltre, dell'allocazione di dati e metadati delle strutture componenti gli ambienti di Download e Upload, come verrà illustrato nelle prossime trattazioni.

Download Environment



Come già accennato, il Download Environment è l'ambiente di lavoro che, all'interno dell'architettura del Server, si occupa di fornire il servizio di download di file da server a client.

La struttura del Download Environment è la più complessa e caratteristica di tutta l'architettura del Server, è pertanto opportuno soffermarsi sul suo funzionamento, parlando fin da subito dell'implementazione.
I dettagli sul trasferimento dati affidabile verranno esposti nel capitolo 3.

Implementazione

Il Download Environment è una lista collegata formata da blocchi: ogni blocco, strutturato come in figura, viene allocato dinamicamente nel momento in cui viene richiesto il download per un nuovo file appartenente alla directory del server e ad ogni blocco è associato uno e un solo file.

Per comprendere il funzionamento della struttura è conveniente partire dalla descrizione di uno scenario tipico.
Nel momento in cui viene ricevuta una richiesta di *get*, il Receptionist si occupa di eseguire le seguenti azioni:

- dopo aver verificato l'esistenza del file richiesto, controlla se nel Download Environment sia presente o meno un blocco relativo a tale file (funzione *start_download*);

- nel caso in cui non esista alcun blocco residente in memoria e associato al file richiesto, sarà il Receptionist stesso ad allocarne lo spazio necessario per poi attaccare il nuovo blocco all'ultimo esistente e aggiornare la lista a blocchi del Download Environment.

La struttura dati di ciascun blocco (struct *block_*) gioca un ruolo fondamentale nella logica dell'applicazione, pertanto è opportuno soffermarsi a descriverla nel dettaglio, facendo attenzione al significato dei suoi attributi principali.

```
typedef struct block_ {

    char                    *filename;

    char                    *buffer_cache;

    int                     server_sock_desc;

    worker                  *workers;

    pthread_t               ack_keeper;

    pthread_t               volture;

    struct block_           *next;

    int                     BLTC;

    char                    eraser;

    char                    quit;


}               block;
```

```
typedef struct worker_{

    pthread_mutex_t         s_window_mutex;

    pthread_t               time_wizard;

    int                     identifier;

    struct block_           *my_block;

    pthread_t               tid;

    struct sockaddr_in      *client_addr;

    int                     len;

    int                     sockfd;

    char                    is_working;

    sw_slot                 *sliding_window_slot_;

    struct worker_          *next;

}               worker;
```

1. Buffer cache

La ragion d'essere del blocco è nel buffer cache.
Trovato il file all'interno della server directory, questo viene caricato integralmente in memoria principale, nella stringa buffer cache (funzione *mmap*). È una cache, dunque evita inutili accessi alla memoria di massa (più costosi in termini di prestazioni) nel caso in cui arrivino nuove richieste di *get* per lo stesso file.

2. Workers

Workers è un array di strutture dati *worker* contenenti diversi attributi, tra cui *pthread_t tid*, identificatore del *Worker Thread*.
Logicamente, un worker thread è un operatore, interno ad un blocco del Download Environment, che svolge la funzione di trasferimento del file da server a client.
Ogni blocco contiene un numero N (configurabile) di worker threads pronti a soddisfare le richieste provenienti dai client.
Al lancio di ogni worker thread, questo rimane in attesa di ricevere un client target verso cui inoltrare il file caricato sul suo blocco; una volta attivo, il worker thread avrà già ricevuto informazioni circa l'indirizzo del client ed eseguirà il trasferimento affidabile del file (funzione *reliable_file_transfer*) attraverso la socket del blocco, identificata dal socket descriptor (*server_sock_desc*).
Sostanzialmente, caricando un blocco in memoria principale viene creata dinamicamente una "thread pool" di operatori in attesa di essere svegliati, per portare a termine gli eventuali servizi richiesti.
Il task del trasferimento affidabile prevede l'affiancamento, ad ogni worker, di un corrispettivo *Time Wizard* thread adibito alla gestione dei timer e delle ritrasmissioni di pacchetti.
Questo aspetto verrà chiarito più avanti nella trattazione.

3. Socket descriptor

Descrittore della socket del blocco (*int server_sock_desc*). Ogni blocco apre una sessione su un canale attraverso una socket ad hoc.

4. Acknowledgements keeper

L'acknowledgements keeper (*pthread_t ack_keeper*) è il thread che si occupa della ricezione di acknowledgements durante il trasferimento dei file in download. Ogni blocco possiede un singolo ACK

keeper che riceve pacchetti dalla socket del blocco: qualsiasi pacchetto proveniente dalla socket del blocco sarà un ACK destinato a uno dei worker threads che operano (potenzialmente) in parallelo.
L'algoritmo utilizzato per il trasferimento dati affidabile verrà approfondito successivamente.

5. BLTC

La variabile intera *BLTC* (*Block Life Timer Countdown*) indica il valore corrente del timer per la vita residua del blocco in memoria principale.
In sostanza, nel momento in cui un blocco viene caricato in memoria principale, ha un tempo limite entro cui può permanere in questa condizione: di default, alla creazione del blocco, il BLTC viene settato ad un numero di secondi proporzionale alla dimensione del file caricato (file più pesanti da ricaricare potranno restare in memoria più a lungo).
Ogni volta che un worker thread viene svegliato, incrementa di 1 il BLTC del blocco; quando non ci sono più worker threads attivi, il timer inizia a scorrere e se nessuna richiesta ulteriore raggiunge il blocco prima dello scadere del timer, il blocco viene deallocato, altrimenti il valore del timer viene incrementato e resettato.

6. Volture

È il thread incaricato per la deallocazione del blocco dalla memoria RAM. Il *volture* aspetta che l'ultimo thread rimasto attivo gli invii un segnale al completamento di un trasferimento in download: una volta ricevuto tale segnale il *volture* setta un timer di valore pari a BLTC secondi, allo scadere del quale, se nessuna nuova richiesta è stata notificata, il blocco viene deallocato liberando spazio in RAM.

7. Next

Puntatore al prossimo blocco.

Il Download Environment basa il suo funzionamento sul *principio di località temporale*: più grande è il numero di riferimenti allo stesso file, maggiore è il tempo di permanenza del blocco nel sistema. Dunque, maggiore è la frequenza di accesso al blocco, minore sarà la probabilità che questo venga deallocato e più rapidi saranno i trasferimenti.
La presenza della thread pool innestata nel blocco permette di evitare l'allocazione continua di risorse necessarie al lancio di nuovi thread: nel caso in cui la thread pool fosse satura (con tutti i thread attivi) e arrivasse un nuovo riferimento al file, solo allora verrebbe allocato un nuovo blocco.

Una struttura di questo tipo si presta nella sua massima utilità in contesti di richieste multiple dirette verso il download dello stesso file: se, ad esempio, questa applicazione fosse utilizzata per la distribuzione di film, nel giorno di uscita di un film molto atteso è probabile che molti utenti richiederebbero alla stessa ora il download dello stesso file.

Upload Environment

```
struct upload_block {
    int                     sockfd;
    int                     identifier;
    char                    filepath[MAXLINE];
    char                    ACK[ACK_SIZE];
    struct sockaddr_in      *clientaddr;
    int                     addr_len;
    pthread_t               uploader;
    pthread_t               writer;
    rw_slot                 *rcv_wnd;
    char                    uploading;
    int                     sem_id;
    struct upload_block     *next;
};
```

L'Upload Environment, inizializzato dal main thread al lancio dell'applicazione, consiste in una lista collegata di strutture *upload_block*, ognuna delle quali mantiene attiva una coppia di thread, *uploader* e *writer*, che svolgono un lavoro reciprocamente dipendente per la ricezione affidabile dei dati provenienti dal lato client, servendo le richieste di *put* inoltrate dal Receptionist.
Mentre l'*uploader* si occupa della ricezione dei pacchetti e dell'invio dei relativi acknowledgements, il *writer* attende un segnale di notifica (*SIGUSR* definito dall'utente), dopo il quale dovrà scrivere il contenuto dei pacchetti che costituiranno il nuovo file da caricare nella directory del server.

La libreria *<sys/sem.h>,* che permette l'istanziazione di array semaforici per il controllo del flusso d'esecuzione, si è rivelata fondamentale per la gestione dei thread: per ognuno dei blocchi, *uploader* e *writer* rimangono in attesa (azione bloccante) del proprio token.

Implementazione

Nel momento in cui viene ricevuta una richiesta di *put* da un determinato indirizzo client, vengono eseguite a catena le seguenti azioni:

- l'indirizzo del client da cui è arrivata la richiesta di *put* viene impostato come valore della struttura dati *clientaddr* di un blocco in attesa nell'Upload Environment;
- la corrispondente coppia di thread *uploader-writer* inizia il completamento dei propri task e rimane attiva per tutto il periodo di trasferimento del file;
- una volta terminato il trasferimento, la coppia torna allo stato di attesa.

Il limite della struttura dell'Upload Environment risiede nella dimensione della thread pool: il numero di coppie *writer-uploader* è limitato al parametro *POOLSIZE*, configurabile direttamente nel file *UploadEnvironment.c*. Approfondimenti sull'argomento si affronteranno nel capitolo 5 della presente relazione.

Ogni *upload block* è associato ad una socket ad-hoc per il trasferimento di file in ingresso e per l'invio di ACK. La scelta di utilizzare socket multiple per l'upload di file è finalizzata ad aumentare il grado di parallelismo e, come molte altre scelte all'interno del progetto, a mantenere ordine e modularità del codice.

Architettura del Client RUFT

Architettura “a specchio”

L’architettura del RUFT Client è stata pensata e implementata in modo da massimizzare il riutilizzo dei blocchi di codice e delle funzioni utilizzate anche per il lato server.

Notiamo in primo luogo che, intuitivamente, per un client richiedere un download (comando *get*) equivale ad eseguire l’upload di un file presente nella server directory sulla client directory.

Analogamente, una richiesta di upload di file sul server (comando *put*) da parte di un client equivale ad eseguire il download verso il server di tale file.

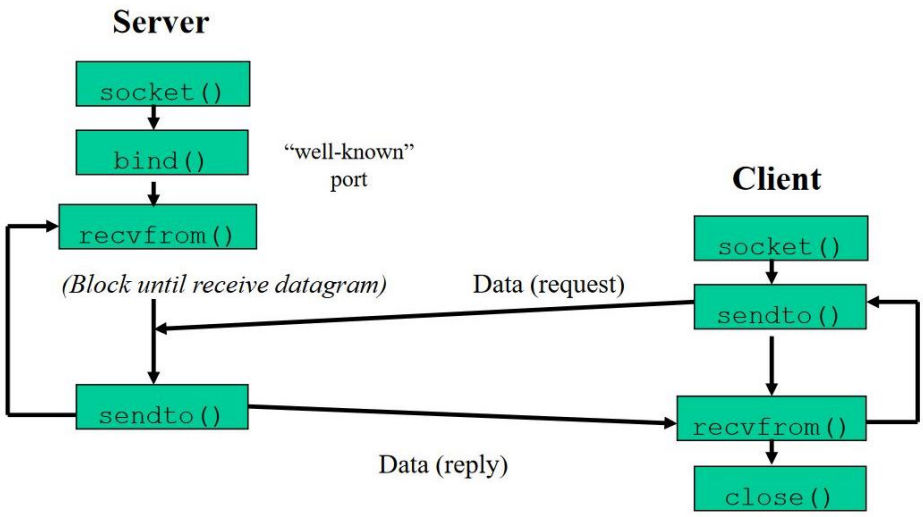
Dunque, essendoci una certa specularità tra le modalità di trasferimento dati client-server e server-client, tale specularità non potrà che concretizzarsi all’interno delle architetture RUFT Client e RUFT Server, sebbene la prima presenti diverse semplificazioni:

- Il Download Environment del lato client è gestito strutturalmente con lo stesso schema logico applicato per l’Upload Environment del lato server, ma il primo presenta semplici coppie di thread *downloader* e *writer*, che cooperano allo scaricamento del file.

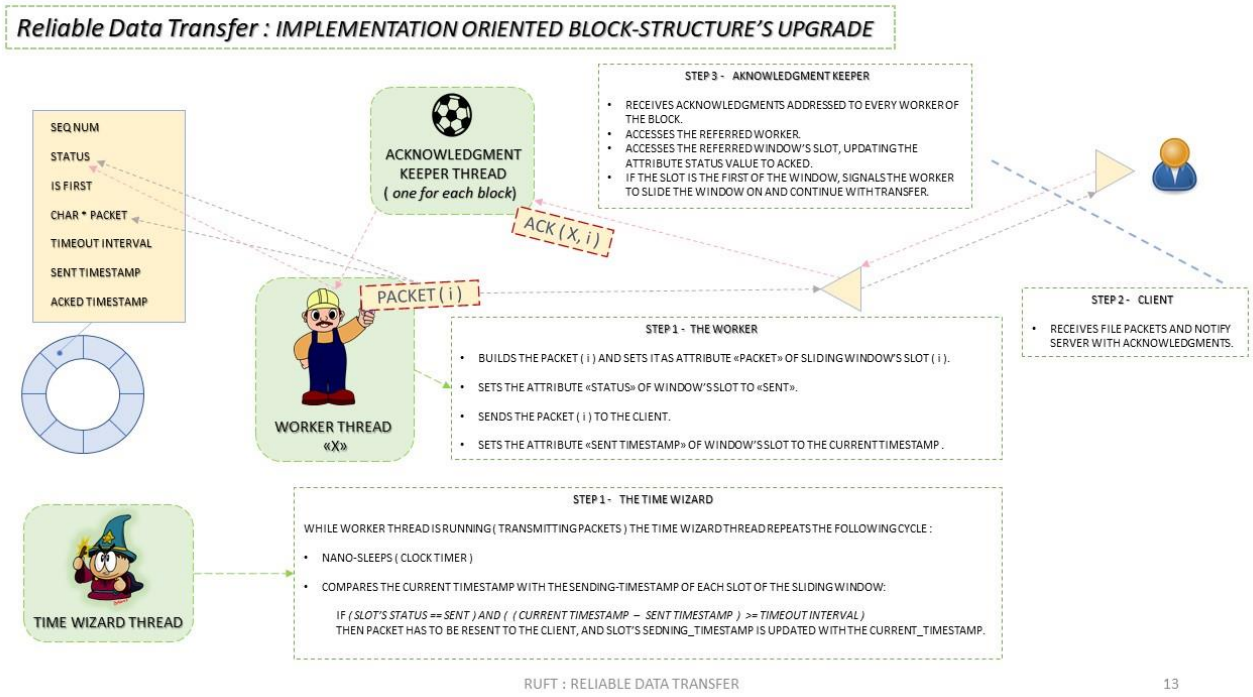
- L’upload di file su server è analogamente gestito come fosse un download per il server, ma eliminando tutta la struttura a blocchi: un *worker thread* coopera con un *ack keeper* e un *time wizard* per il caricamento del singolo file sul server e al successivo upload dovranno essere rilanciati.

La scelta di non mantenere, per quanto concerne il lato client, la struttura permanente di *thread pool* è basata sulla considerazione che, almeno per molte tipologie di applicazione, è abbastanza improbabile che si voglia effettuare l’accesso al server per eseguire download e/o upload in serie: in tal caso, può risultare conveniente allocare le risorse necessarie all’esecuzione di threads solo nel momento in cui questo si riveli necessario.
Ad esempio, quando un utente vuole connettersi al server per la sola visualizzazione del contenuto della directory del server, in tal caso verranno perlomeno risparmiate risorse di memoria.

In linea generale, l’architettura del sistema software rispecchia il tipico pattern client-server per lo scambio di informazioni senza connessione, come mostrato in figura.



3. Trasferimento dati affidabile



Selective Repeat

Il protocollo a livello di trasporto utilizzato nell'applicazione è UDP, il quale, come già menzionato, non assicura un servizio di trasferimento affidabile. Nell'implementazione reale del trasferimento affidabile a livello di trasporto, vengono utilizzati protocolli per il trasferimento in pipeline atti a massimizzare il throughput di una connessione pur mantenendo solida la condizione di affidabilità.
Esempi di questi protocolli sono "*Go Back N*" e "*Selective Repeat*": per la trasmissione dei file in questo progetto software si è scelto di implementare il secondo.

Ma cosa significa "affidabilità" e da cosa è garantita?

Il problema dell'affidabilità nel trasferimento dei dati consiste nella necessità del ricevente di:

a) ricevere tutti i dati trasmessi dal mittente;
b) ricevere tali dati nel giusto ordine;
c) riceverli intatti (senza errori sui bit).

Questo progetto si sofferma sul trasferimento ordinato dei dati, e sulla consegna sicura al destinatario.
Essendo il protocollo di rete IPv4 un protocollo di tipo *best effort*, non assicura in alcun modo la consegna dei dati al ricevente e dunque per tale scopo viene tipicamente utilizzato, a livello di trasporto, il protocollo TCP. Dal momento in cui, al suo posto, si è scelto di utilizzare il protocollo UDP, è stato necessario implementare la logica a ripetizione selettiva a livello applicativo.

Il protocollo in pipeline *selective repeat* prevede l'utilizzo di strutture dati ausiliarie, quali:

- una finestra di scorrimento (*sliding window*) di dimensione W sul lato del mittente;
- una finestra di ricezione (*receiving window*) anch'essa di dimensione W sul lato del destinatario.

Avere una finestra di dimensione W, per un mittente, significa poter inviare consecutivamente fino a un numero di pacchetti W senza che i relativi acknowledgements siano stati recapitati.

I possibili eventi che possono verificarsi dal lato del **mittente** durante il trasferimento di dati sono:

1) **Invio di un nuovo messaggio**
   Il mittente verifica se la finestra è piena: se non lo è, crea ed invia il pacchetto assegnandogli il più piccolo numero di sequenza non ancora utilizzato nella finestra; se invece è piena, restituisce il pacchetto al livello superiore per una successiva ritrasmissione o, eventualmente, lo salva in un buffer.

2) **Timeout**
   Ogni pacchetto ha il suo timer logico, collegato ad un unico timer fisico generale. Allo scadere del tempo di contatore, il singolo pacchetto viene ritrasmesso.

3) **Ricezione di ACK**
   Se il pacchetto relativo all'ACK è presente nella finestra di trasmissione, viene marcato come ricevuto. Se si tratta del primo pacchetto, nella finestra, per ordine di *sequence number*, allora la finestra viene traslata al prossimo pacchetto in attesa di ACK.

I possibili eventi che possono verificarsi dal lato del **destinatario** durante il trasferimento di dati sono:

1) **Pacchetto ricevuto correttamente**
   Quando un pacchetto con numero di sequenza nell'intervallo della finestra di ricezione viene ricevuto correttamente, viene restituito al mittente un ACK selettivo.
   Se il pacchetto presenta numero di sequenza pari alla base della finestra di ricezione attuale, viene consegnato all'applicazione insieme a tutti i pacchetti salvati nel buffer che hanno numeri di sequenza consecutivi al suo.

2) **Pacchetto "doppione"**
   Quando viene ricevuto un pacchetto che era già stato riscontrato e notificato, viene comunque generato e trasmesso un nuovo ACK al mittente.

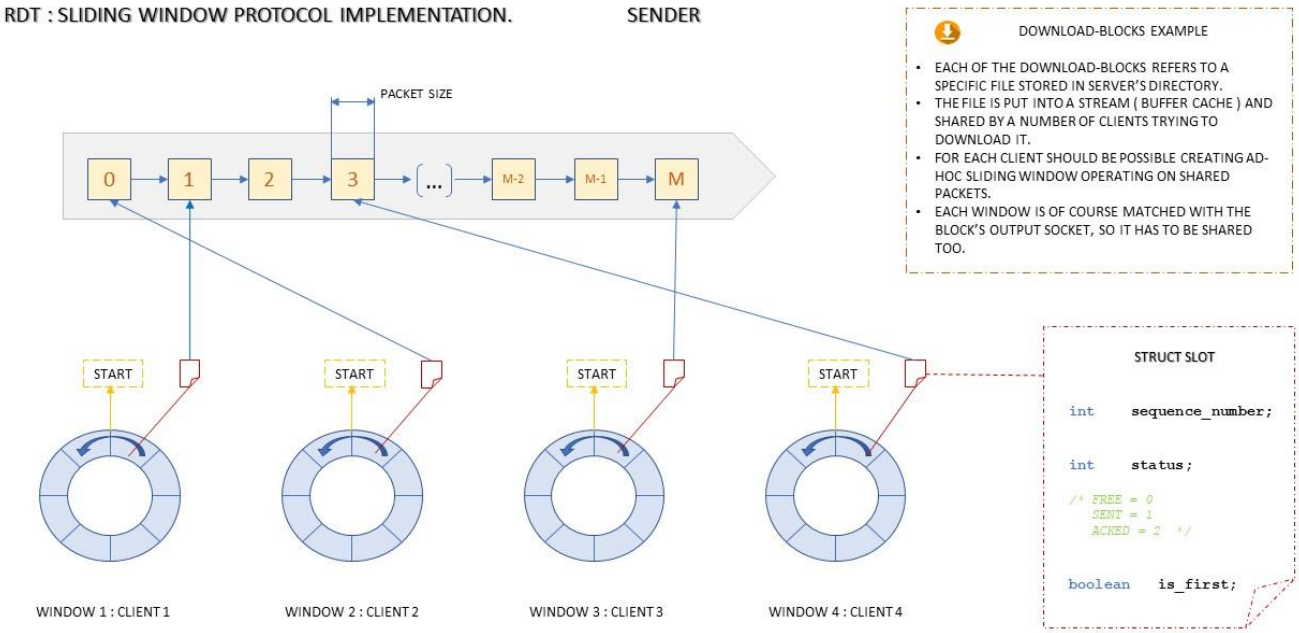
3) **Tutti gli altri casi**
   In un caso qualsiasi diverso dai primi due, il destinatario ignora il pacchetto.

Finestre ed implementazione dell'algoritmo

L'implementazione dell'algoritmo di *selective repeat*, come protocollo a finestra scorrevole, costituisce la parte più intensa del lavoro svolto.

Di fatto, il codice agisce così come è descritto per l'algoritmo teorico, ma si ritiene necessario illustrare dettagliatamente i passaggi effettuati nell'implementazione, le strutture utilizzate, i threads coinvolti nell'esecuzione dell'algoritmo e le funzioni per il trasferimento di dati.

- ### Finestra di scorrimento: lato mittente



La finestra di scorrimento è implementata con la tecnica del buffer circolare. In particolare, la finestra di scorrimento sarà costituita da una lista collegata di strutture *sliding_window_slot*, in cui l'ultimo slot viene collegato al primo.
Ogni slot contiene, come campi principali:

1. La variabile intera *sequence_number*: il numero di sequenza che indica la posizione del pacchetto, temporaneamente associato allo slot, all'interno del file da trasmettere.

2. Il flag *is_first*: indica se lo slot in questione è il primo della finestra di scorrimento.

```
typedef struct sliding_window_slot_ {
    int                     sequence_number;
    int                     bytes;
    char                    is_first;
    char                    retransmission;
    int                     status;
    struct timespec         sent_timestamp;
    struct timespec         acked_timestamp;
    long                    timeout_interval;
    char                    packet[ MAXLINE + 64 ];
    struct sliding_window_slot_     *next;
}               sw_slot;
```

3. Il flag *retransmission*: indica se il pacchetto associato allo slot è stato ritrasmesso.

4. La variabile intera *status*: indica lo stato attuale dello slot e può assumere i valori *FREE* (slot non associato ad alcun pacchetto), *SENT* (slot associato ad un pacchetto inviato, non ancora notificato) e *ACKED* (slot associato ad un pacchetto inviato e notificato).

5. *sent_timestamp* e *acked_timestamp*: istantanee dei momenti di invio e di acknowledgement di un dato pacchetto, necessarie per il calcolo del timeout adattativo e per la gestione delle ritrasmissioni.

6. *packet*: buffer di caratteri contenente la copia del pacchetto inviato, da usare nelle ritrasmissioni.

Di base, prendendo come esempio d'utilizzo il download dal RUFT server, si ha che per ogni istanza di download si avrà una coppia *time wizard* e *worker* associata ad una e una sola finestra di scorrimento, che verrà condivisa con l'acknowledgement keeper dell'intero blocco.

Il file caricato nel buffer cache viene "spezzettato" in *chunks* di bytes, di dimensioni minori o uguali a *PACKET_SIZE*.

Implementazione dell'algoritmo di trasferimento affidabile

- All'interno di un blocco del Download Environment, un *worker* thread, dopo essersi risvegliato tramite segnale definito dall'utente, incrementa il BLTC ed esegue la funzione *reliable_file_forward*. Il *Time Wizard* thread associato viene risvegliato analogamente.

- Il *worker* assume il controllo della sliding window tramite un mutex (onde evitare conflitti in scrittura con l'*ACK keeper* operante sulla stessa finestra) ed inizia ad associare ciclicamente i pacchetti, composti dai *chunks* presenti nel buffer cache, ai relativi slots della finestra di scorrimento. Tali pacchetti vengono man mano inviati al destinatario, con la procedura di *selective repeat* descritta in precedenza, tramite la funzione *sendto*. A finestra satura (tutti i pacchetti sono in stato di *SENT*), il *worker* rilascia il mutex sulla finestra e attende un segnale dall'*ACK keeper* per effettuare lo scorrimento e procedere all'invio dei successivi pacchetti.

- All'istante di invio dei pacchetti viene preso il timestamp corrente e salvato nel relativo campo *sent_timestamp* della struttura *sliding_window_slot*. Tramite questo parametro, il *time wizard* controlla ciclicamente, con impulso *beat* (*nanosleep*), il timer di ritrasmissione di ogni pacchetto inviato, in attesa di acknowledgement o dello scadere del timeout.

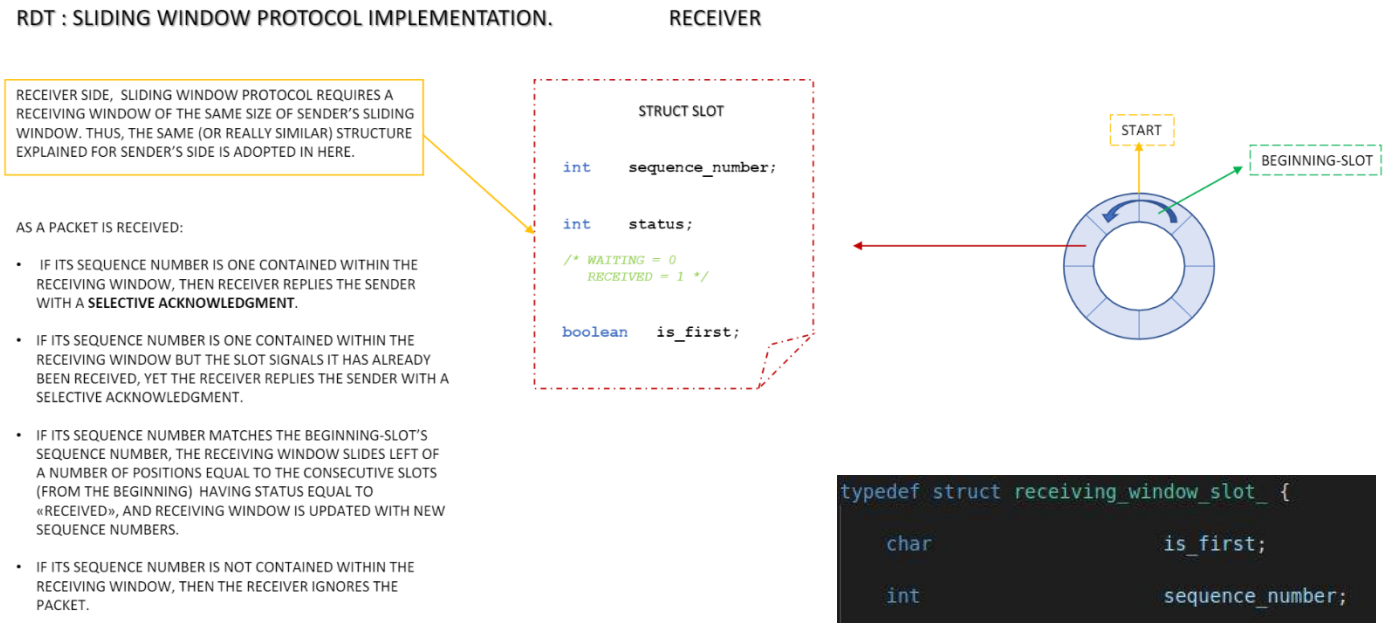
- L'*acknowledgement keeper* rimane in attesa di ACK dal canale di I/O: alla ricezione di un ACK, assume il controllo della finestra di scorrimento, aggiorna lo *status* dello slot da *SENT* ad *ACKED* e registra il timestamp corrente nello slot (*acked_timestamp*). Nel caso in cui lo slot suddetto sia il primo slot della finestra, viene inviata una notifica al *worker* thread per permettere lo scorrimento e l'invio di nuovi messaggi.

- Il *time wizard* si occupa di aggiornare il valore del timeout adattativo ad ogni riscontro di ACK, fatta eccezione per i pacchetti già ritrasmessi.

- Nel caso in cui il timeout sia scaduto prima dell'arrivo del relativo acknowledgment, il *time wizard* ha la responsabilità di ritrasmettere il pacchetto e di segnalare la ritrasmissione sullo slot della finestra.

Al termine della trasmissione, la coppia *worker-time wizard* torna in stato di blocco, in attesa di nuove istanze di download.

- Finestra di ricezione: lato destinatario



La struttura della finestra di ricezione è simile a quella della sliding window, ma subisce un ridimensionamento: alcuni dei campi presenti nella finestra di scorrimento non sono più necessari.

```
typedef struct receiving_window_slot_ {
    char                    is_first;
    int                     sequence_number;
    int                     status;
    char                    *packet;
    char                    *header;
    struct receiving_window_slot_     *next;
}               rw_slot;
```

Implementazione dell'algoritmo di ricezione file

- Il *downloader* thread effettua la ricezione dei pacchetti provenienti dal mittente e condivide la finestra di ricezione con il *writer* thread.

- Il *writer* ha la responsabilità di creare una sessione di I/O su un file (nuovo record di sistema o file troncato se il filepath di riferimento è già presente nella directory di destinazione) e, quando notificato, di scrivervi il contenuto informativo dei pacchetti, caricato sui singoli slot della finestra di ricezione dal *downloader*.

- Il *downloader* scrive il contenuto informativo del pacchetto ricevuto nello slot relativo al numero di sequenza indicato nell'header del pacchetto, e manda un ack selettivo. Se il numero di sequenza è fuori dalla finestra di ricezione attuale, il pacchetto viene ignorato. Se invece il numero di sequenza appartiene ad uno slot già utilizzato, il pacchetto viene ignorato ma viene rimandato un ack al mittente per quel pacchetto.

- Nel momento di arrivo del pacchetto con numero di sequenza pari al primo numero di sequenza della finestra di ricezione, il *downloader* notifica il *writer* che può procedere alla scrittura: il *writer* scrive su file il contenuto informativo di tutti i pacchetti consecutivi inseriti nella finestra di scorrimento a partire dal primo slot. Al

termine della scrittura, sarà il *writer* stesso ad aggiornare la finestra di ricezione e ad effettuarne lo scorrimento.

Lo schema descritto, che prevede lo scambio di dati affidabili tra mittente e destinatario, è applicato in modo speculare su entrambi i lati del software (client e server). Un'analisi dell'efficienza dell'algoritmo al variare dei parametri di configurazione è riportata nel capitolo 4 della trattazione.

Timeout e ritrasmissioni

Come abbiamo già accennato in precedenza nel primo paragrafo, inizialmente l'applicazione utilizza, per la ritrasmissione dei pacchetti, un timeout di valore T configurabile. Tuttavia, affinché la trasmissione di tali pacchetti avvenga in maniera consistente e tale da non consumare risorse inutilmente, si è implementato l'utilizzo di un timeout adattativo nel codice *time_toolbox.c*.

Il timeout adattativo permette di riconfigurare il valore iniziale T in base alla congestione di rete tramite una stima del tempo di andata e ritorno *RTT* (*Round Trip Time*) dei pacchetti.

Per il calcolo di tale stima si è utilizzata la libreria *<time.h>*, in particolare la struttura dati *timespec* e la funzione *clock_gettime()*, grazie alle quali è stato possibile misurare gli intervalli di tempo.

La stima dell'*RTT* è ottenuta tramite la seguente formula:

$$ESTIMATED\ RTT = (0.875 \times ESTIMATED\ RTT) + (0.125 \times SAMPLE\ RTT)$$

la quale, basandosi sul principio di località temporale, pesa maggiormente il valore dell'*RTT* precedentemente stimato.

Per il calcolo del nuovo valore T del timeout, invece, si sfrutta la formula:

$$TIMEOUT\ INTERVAL = ESTIMATED\ RTT \times (4 \times \Delta RTT)$$

dove ΔRTT è la stima della differenza tra valore iniziale e valore stimato dell'*RTT*:

$$\Delta RTT = (0.75 \times \Delta RTT) + (0.25 \times (SAMPLE\ RTT - ESTIMATED\ RTT))$$

In questo modo, ad ogni ricalcolo il valore del timeout si scosta leggermente dal valore stimato precedentemente se la rete non è congestionata, mentre in caso di traffico intenso il valore di T aumenta drasticamente, evitando così di effettuare ritrasmissioni inutili.

4. Analisi delle prestazioni

L'analisi delle prestazioni si focalizza fondamentalmente sulla velocità dell'algoritmo di trasferimento affidabile. Come già anticipato, la modularità del codice ha permesso di utilizzare lo stesso algoritmo affidabile per il download e l'upload di file, a meno di qualche piccola modifica. Pertanto, nello studio delle prestazioni, si è scelto di valutare i tempi di trasferimento in download, che vengono trattati da upper-bound, data la continua allocazione di blocchi in memoria principale.

I parametri delle misurazioni sono:

1. il tipo di timeout di ritrasmissione adottato;
2. il valore del *beat* scandito dal *time wizard*;
3. la probabilità di perdita dei pacchetti;
4. la dimensione della finestra di scorrimento.

In prima analisi, si è scelto di studiare le prestazioni al variare dell'ampiezza della finestra di scorrimento: a seconda del tasso di congestione della rete, simulato attraverso l'incremento della probabilità di perdita p, si sono distinti tre casi.

- Best case scenario: il migliore dei casi è quello in cui la rete non è congestionata, dunque la *loss probability* è pari allo 0%

BEST CASE SCENARIO							
Timeout adaptive		Timeout adaptive		Timeout adaptive		Timeout adaptive	
Beat: 0,07s		Beat: 0,07s		Beat: 0,07s		Beat: 0,07s	
Loss probability: 0%		Loss probability: 0%		Loss probability: 0%		Loss probability: 0%	
Window size: 11		Window size: 20		Window size: 5		Window size: 25	
1	0,00505	1	0,00485	1	0,00494	1	0,00319
2	0,00464	2	0,00480	2	0,00412	2	0,00348
3	0,00447	3	0,00510	3	0,00648	3	0,00397
4	0,03170	4	0,00415	4	0,00730	4	0,00357
5	0,00488	5	0,00339	5	0,00380	5	0,00473
6	0,07076	6	0,00354	6	0,00589	6	0,00468
7	0,00443	7	0,00375	7	0,00408	7	0,00220
8	0,00497	8	0,00348	8	0,07137	8	0,00350
9	0,00561	9	0,00473	9	0,00443	9	0,00459
10	0,00430	10	0,00546	10	0,00354	10	0,00452
11	0,00439	11	0,00439	11	0,00396	11	0,00352
12	0,00371	12	0,00332	12	0,00842	12	0,00408
13	0,00504	13	0,00388	13	0,00448	13	0,00337
14	0,00491	14	0,00353	14	0,00701	14	0,00497
15	0,00471	15	0,00471	15	0,00509	15	0,00402
16	0,00371	16	0,00312	16	0,00402	16	0,00429
17	0,00414	17	0,00662	17	0,00427	17	0,00335
18	0,00435	18	0,00373	18	0,00399	18	0,00337
19	0,00391	19	0,00506	19	0,00512	19	0,00425
20	0,00532	20	0,00406	20	0,00412	20	0,00364
21	0,00356	21	0,00354	21	0,01039	21	0,00320
MEDIA	0,00898	**MEDIA**	0,00425	**MEDIA**	0,00842	**MEDIA**	0,00383
MAX	0,07076	**MAX**	0,00662	**MAX**	0,07137	**MAX**	0,00497
MIN	0,00356	**MIN**	0,00312	**MIN**	0,00354	**MIN**	0,00220

- Scenario 1: in questo caso la probabilità di perdita si alza al 15%, con una “lieve” congestione di rete.

SCENARIO 1							
Timeout adaptive		Timeout adaptive		Timeout adaptive		Timeout adaptive	
Loss probability: 15%		Loss probability: 15%		Loss probability: 15%		Loss probability: 15%	
Window size: 7		Window size: 11		Window size: 18		Window size: 22	
1	0,07195	1	0,07135	1	0,07142	1	1,06059
2	0,14246	2	0,14105	2	0,07176	2	0,07177
3	0,28341	3	0,28356	3	0,21363	3	0,14180
4	0,021236	4	0,14210	4	0,14203	4	0,07211
5	0,21131	5	0,14412	5	0,14175	5	0,14138
6	0,07164	6	0,07147	6	0,14385	6	0,07229
7	0,28344	7	0,35465	7	0,07192	7	0,07111
8	0,21418	8	0,07155	8	0,28251	8	0,28261
9	0,21331	9	0,21253	9	0,07153	9	0,07168
10	0,21209	10	0,07211	10	0,21264	10	0,14230
11	0,14184	11	0,21182	11	0,21220	11	0,07185
12	0,35314	12	0,21256	12	0,14197	12	0,14179
13	0,07142	13	0,07119	13	0,28225	13	0,07166
14	0,35493	14	0,14160	14	0,21186	14	0,14123
15	0,21198	15	0,28363	15	0,35336	15	0,14345
16	0,14310	16	0,14172	16	0,14257	16	0,14299
17	0,21260	17	0,07092	17	0,07119	17	0,71360
18	0,21231	18	0,35376	18	0,07126	18	0,14634
19	0,14379	19	0,07081	19	0,21220	19	0,00604
20	0,21360	20	0,21186	20	0,00408	20	0,07234
21	0,00410	21	0,21279	21	0,28278	21	0,07189
22	0,14320	22	0,14198	22	0,14820	22	0,21254
23	0,14279	23	0,14205	23	0,07151	23	0,00359
24	0,21250	24	0,14184	24	0,07135	24	0,07198
25	0,28198	25	0,07247	25	0,07160	25	0,14177
MEDIA	0,18273224	**MEDIA**	0,166656087	**MEDIA**	0,15774957	**MEDIA**	0,136884348
MAX	0,35493	**MAX**	0,35465	**MAX**	0,35336	**MAX**	0,7136
MIN	0,0041	**MIN**	0,0708	**MIN**	0,0041	**MIN**	0,0036

- Scenario 2: la probabilità di perdita sale al 30%, incrementando la congestione di rete e decrementando la velocità di trasmissione, come, ad esempio, si nota dal confronto tra le colonne con finestra di scorrimento di ampiezza 11.

SCENARIO 2							
Timeout adaptive		Timeout adaptive		Timeout adaptive		Timeout adaptive	
Loss probability: 30%		Loss probability: 30%		Loss probability: 30%		Loss probability: 30%	
Window size: 7		Window size: 11		Window size: 20		Window size: 30	
1	1,62538	1	0,49938	1	2,40619	1	1,31368
2	0,2819	2	0,63802	2	0,42355	2	0,35264
3	0,42457	3	0,35365	3	0,14237	3	0,14184
4	0,42394	4	1,07119	4	0,21838	4	0,21269
5	0,63813	5	0,6328	5	0,28223	5	0,21246
6	0,56505	6	0,89858	6	0,56612	6	0,28503
7	0,42997	7	0,21281	7	0,42430	7	0,35448
8	0,14144	8	0,35418	8	0,14115	8	0,14314
9	0,63417	9	0,99095	9	0,21225	9	0,21411
10	0,35384	10	0,42377	10	0,28333	10	0,14198
11	0,56610	11	0,25728	11	0,42650	11	0,21181
12	0,35404	12	0,53360	12	0,14118	12	0,69171
13	0,63602	13	0,42010	13	0,21243	13	0,21249
14	0,63415	14	0,56469	14	0,28423	14	0,14287
15	0,35349	15	0,21356	15	0,56649	15	0,14290
16	0,49835	16	0,35377	16	0,42543	16	0,21339
17	0,91759	17	0,21094	17	0,63820	17	0,56688
18	0,71032	18	0,37507	18	0,14137	18	0,35428
19	0,77961	19	0,28174	19	0,21228	19	0,28329
20	0,21184	20	0,42407	20	0,28321	20	0,70739
21	0,56633	21	0,42381	21	0,56615	21	0,14163
22	0,28270	22	0,35422	22	0,42249	22	0,14201
23	0,63630	23	0,35379	23	0,14224	23	0,07190
24	0,28270	24	0,42508	24	0,45538	24	0,21257
25	0,42316	25	0,42627	25	0,14174	25	0,21243
MEDIA	0,498426522	**MEDIA**	0,45895304	**MEDIA**	0,31867174	**MEDIA**	0,261447
MAX	0,91759	**MAX**	1,07119	**MAX**	0,6382	**MAX**	0,70739
MIN	0,1414	**MIN**	0,2109	**MIN**	0,1412	**MIN**	0,0719

In ogni caso, l'incremento dell'ampiezza della finestra di scorrimento comporta un aumento delle prestazioni in termini di velocità di trasferimento.

Per effettuare le misure con cui si studia il valore ottimale del timeout T, così come quello della finestra di trasmissione N al variare della probabilità di perdita p, si è scelto di implementare, all'interno del codice, un clock che scandisca il tempo di trasferimento di ciascun file, dall'invio del primo alla ricezione dell'ultimo messaggio, e lo stampi sul terminale.

Sulla base delle velocità medie di trasferimento riscontrate dalle misurazioni precedenti, che sfruttavano un timeout di tipo adattativo per la ritrasmissione, è stato studiato il valore del timeout fisso ("timeout standard").

Procedendo per tentativi e decrementando tale valore, si nota un ovvio decremento del tempo di trasmissione.

Tuttavia, è emerso che per poter massimizzare le prestazioni in velocità di trasferimento, a prescindere dalla tipologia di timeout adottata, è necessario e conveniente scandire il tempo a granularità più fine, diminuendo il valore del *beat* e quindi riducendo la finestra temporale con la quale il *time wizard* scandisce il tempo.

La differenza sostanziale nell'utilizzo del timeout fisso piuttosto che di quello adattativo, risiede nel concetto di congestione di rete: l'utilizzo di un timeout di tipo fisso, in un contesto di congestione altamente variabile, risulta controproducente in quanto potrebbe contribuire all'aumento della congestione stessa a causa delle eccessive ritrasmissioni; al contrario, utilizzare un timeout di tipo adattativo nello stesso contesto, risulta più appropriato in quanto garantisce un flusso di dati adeguato alla congestione di rete corrente. Nel caso studiato, la probabilità di perdita dei pacchetti p è stata fissata al 15%, ragione per cui l'uso del timeout di tipo fisso permette di raggiungere le prestazioni migliori: con un file di dimensione $11{,}3\ \mathrm{kB}$ il throughput medio ottenuto è di $5{,}3\ \mathrm{Mbit/s}$.

TIMEOUT STUDY							
Timeout standard: 1s		Timeout standard: 0,5s		Timeout standard: 0,1s		Timeout standard: 0,07s	
Beat: 0,07s		Beat: 0,07s		Beat: 0,07s		Beat: 0,07s	
Loss probability: 15%		Loss probability: 15%		Loss probability: 15%		Loss probability: 15%	
Window size: 18		Window size: 18		Window size: 18		Window size: 18	
1	1,07081	1	0,50646	1	0,14154	1	0,14104
2	3,01489	2	0,56667	2	0,14344	2	0,14308
3	1,00615	3	1,58694	3	0,42335	3	0,35396
4	3,20595	4	1,08747	4	0,28344	4	0,21182
5	2,10565	5	1,06961	5	0,28581	5	0,21289
6	2,07872	6	1,13347	6	0,28454	6	0,21274
7	2,12471	7	0,50821	7	0,14224	7	0,14386
8	1,07141	8	2,20315	8	0,56413	8	0,42318
9	4,15157	9	0,56921	9	0,14212	9	0,14141
10	1,02978	10	1,66605	10	0,42726	10	0,28352
11	3,12147	11	0,56340	11	0,14179	11	0,07262
12	1,07262	12	1,59487	12	0,14269	12	0,35276
13	3,05599	13	0,57017	13	0,1429	13	0,14253
14	1,01662	14	1,07950	14	0,42474	14	0,21213
15	4,09756	15	2,20640	15	0,4232	15	0,35406
16	3,09082	16	1,65398	16	0,28261	16	0,21437
17	2,07101	17	1,08150	17	0,14255	17	0,14485
18	2,00783	18	1,08333	18	0,14235	18	0,28237
19	0,00356	19	0,00323	19	0,56759	19	0,0037
20	1,06706	20	0,56677	20	0,14317	20	0,14178
21	1,07009	21	0,56688	21	0,4222	21	0,14178
22	3,20145	22	1,64091	22	0,14279	22	0,28237
23	0,00388	23	0,00313	23	0,14189	23	0,00407
24	1,01578	24	0,56487	24	0,14144	24	0,14144
25	3,12632	25	1,63359	25	0,14177	25	0,35276
MEDIA	1,99527	MEDIA	1,02839	MEDIA	0,25526	MEDIA	0,20444
MAX	4,15157	MAX	2,20640	MAX	0,56759	MAX	0,42318
MIN	0,00356	MIN	0,00313	MIN	0,14144	MIN	0,00370

TIMEOUT STUDY										
Timeout standard: 0,01s			Timeout standard: 0,01s			Timeout adaptive			Timeout adaptive			Timeout standard: 0,005s		
Beat: 0,07s			Beat: 0,01s			Beat: 0,07s			Beat: 0,01s			Beat: 0,003s		
Loss probability: 15%			Loss probability: 15%			Loss probability: 15%			Loss probability: 15%			Loss probability: 15%		
Window size: 18			Window size: 18			Window size: 18			Window size: 18			Window size: 18		
1	0,07144		1	0,03293		1	0,07135		1	0,01122		1	0,01125	
2	0,21338		2	0,02211		2	0,14105		2	0,01226		2	0,01217	
3	0,14156		3	0,03258		3	0,28356		3	0,03363		3	0,02338	
4	0,14120		4	0,02201		4	0,14210		4	0,01220		4	0,01491	
5	0,14260		5	0,02228		5	0,14412		5	0,05361		5	0,01558	
6	0,07159		6	0,05214		6	0,07147		6	0,02235		6	0,01513	
7	0,28258		7	0,03307		7	0,35465		7	0,02146		7	0,01146	
8	0,07220		8	0,03360		8	0,07155		8	0,02222		8	0,03259	
9	0,21192		9	0,03367		9	0,21253		9	0,01184		9	0,02704	
10	0,07138		10	0,02219		10	0,07211		10	0,01115		10	0,00848	
11	0,21222		11	0,02127		11	0,21182		11	0,01219		11	0,02458	
12	0,07233		12	0,02199		12	0,21256		12	0,03235		12	0,01099	
13	0,14141		13	0,04238		13	0,07119		13	0,02239		13	0,01790	
14	0,28502		14	0,03383		14	0,14160		14	0,02333		14	0,02858	
15	0,21187		15	0,03292		15	0,28363		15	0,02183		15	0,02066	
16	0,14241		16	0,02274		16	0,14172		16	0,01134		16	0,01910	
17	0,14174		17	0,05673		17	0,07092		17	0,04182		17	0,01818	
18	0,00365		18	0,02177		18	0,35376		18	0,45625		18	0,00411	
19	0,07167		19	0,05184		19	0,07081		19	0,01179		19	0,01156	
20	0,07251		20	0,02215		20	0,21186		20	0,03143		20	0,01223	
21	0,21244		21	0,04354		21	0,21279		21	0,01292		21	0,02269	
22	0,00342		22	0,02309		22	0,14198		22	0,02175		22	0,00478	
23	0,07223		23	0,03252		23	0,14205		23	0,04425		23	0,01103	
24	0,21390		24	0,05310		24	0,14184		24	0,03191		24	0,02469	
25	0,14144		25	0,04307		25	0,07247		25	0,02200		25	0,01129	
MEDIA	0,13623		MEDIA	0,033673043		MEDIA	0,166656087		MEDIA	0,04287		MEDIA	0,016997391	
MAX	0,28502		MAX	0,05673		MAX	0,35465		MAX	0,45625		MAX	0,03259	
MIN	0,0034		MIN	0,0213		MIN	0,0708		MIN	0,0112		MIN	0,0041	

- Worst case scenario: analizzando il caso in cui la probabilità di perdita p è pari all'80%, quindi rete altamente congestionata, e impostando i valori di *beat* e timeout fisso ottimali ricavati in precedenza, il risultato ottenuto, in termini di prestazioni, è comunque soddisfacente.

WORST CASE SCENARIO				
Timeout fisso: 0,005s			Timeout fisso: 0,005s	
Beat: 0,003s			Beat: 0,003s	
Loss probability: 80%			Loss probability: 80%	
Window size: 10			Window size: 22	
1	0,18344		1	0,10735
2	0,21751		2	0,13644
3	0,27569		3	0,17628
4	0,19743		4	0,12516
5	0,25686		5	0,14781
6	0,13974		6	0,20669
7	0,31785		7	0,14989
8	0,32673		8	0,18152
9	0,21773		9	0,11923
10	0,18724		10	0,14446
11	0,17838		11	0,17977
12	0,24927		12	0,16267
13	0,22509		13	0,17304
14	0,20486		14	0,15330
15	0,46581		15	0,12151
16	0,22506		16	0,16165
17	0,21974		17	0,12915
18	0,20250		18	0,15123
19	0,29254		19	0,17446
20	0,18907		20	0,22436
MEDIA	0,238627		MEDIA	0,1562985
MAX	0,46581		MAX	0,22436
MIN	0,1397		MIN	0,1074

5. Limitazioni riscontrate

La principale limitazione dell'applicativo riguarda le caratteristiche di scalabilità intrinseche nell'architettura: se il Download Environment mantiene un'efficace gestione del carico di lavoro all'aumentare delle richieste da parte dei client, attraverso la struttura di pool dinamiche e alla loro auto-distruzione, la stessa cosa non si può dire dell'Upload Environment.

Infatti, l'architettura dell'ambiente di Upload è vincolata alle dimensioni della sua singola pool.

Come accennato nel capitolo 2, il valore *POOLSIZE* è configurabile manualmente all'avvio del server, pertanto basterà adattare le dimensioni dell'ambiente al carico di lavoro in ingresso. Il problema risiede nella dinamicità: allocare un'ingente quantità di metadati può risultare oneroso per il server e comportare l'inutile spreco di preziose risorse di memoria fisica, che potrebbero invece essere investite per il Download Environment.

Allocare una quantità di metadati eccessivamente modesta, al contrario, potrebbe generare, per come è strutturata l'applicazione, troppe negazioni d'accesso al server.

Di fatto, una plausibile, nonché fondamentale soluzione al problema sarebbe quella di introdurre una coda d'attesa, possibilmente con priorità secondo algoritmi di tipo greedy, per gli utenti che richiedano un servizio di *put* e trovino tutti i blocchi di upload occupati.

6. Manuale di installazione e configurazione

L'applicazione è stata sviluppata in ambiente Unix (macchina virtuale con sistema operativo Ubuntu 18.04) su piattaforma di sviluppo Visual Studio Code.
Le analisi delle prestazioni, in particolare le misurazioni in forma tabellare, sono state rappresentate in Microsoft Excel.

Il codice è strutturato nei seguenti file:

- *UDP_Client.c*
- *UDP_Server.c*
- *Upload_Environment.c*
- *Download_Environment.c*
- *time_toolbox.c*
- un file *header* contenente le librerie importate per il progetto
- un *Makefile* contenente le istruzioni per la compilazione

Per l'installazione e la configurazione dell'applicazione basta seguire le istruzioni nell'ordine:

- scaricare dal CD-ROM la cartella "RUFT";
- posizionarsi, da shell, nella cartella scaricata, contenente i file *server* e *client* (gli eseguibili generati compilando con l'istruzione "make all");
- avviare, su terminali diversi, gli eseguibili, rispettivamente con i comandi "./client" e "./server";

All'avvio del programma, il client stamperà a schermo un menù minimale con le operazioni per chiamare i comandi *put*, *get* e *list*.
Per terminare l'applicazione utilizzare il segnale di uscita "CTRL + C" (SIGINT).