ASSIGNMENT

Reti di Calcolatori ed Ingegneria del Web - A.A. 2022/23

Progetto B: Trasferimento file su UDP

Lo scopo del progetto è quello di progettare ed implementare in linguaggio `C` usando l'API del socket di Berkeley un'applicazione client-server per il trasferimento di file che impieghi il servizio di rete senza connessione (socket tipo `SOCK_DGRAM`, ovvero UDP come protocollo di strato di trasporto).

Il software deve permettere:

- Connessione client-server senza autenticazione;
- La visualizzazione sul client dei file disponibili sul server (comando `list`);
- Il download di un file dal server (comando `get`);
- L'upload di un file sul server (comando `put`);
- Il trasferimento file in modo affidabile.

La comunicazione tra client e server deve avvenire tramite un opportuno protocollo.

Il protocollo di comunicazione deve prevedere lo scambio di due tipi di messaggi:

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

Funzionalità del server

Il server, di tipo concorrente, deve fornire le seguenti funzionalità:

- L'invio del messaggio di risposta al comando `list` al client richiedente;
- Il messaggio di risposta contiene la filelist, ovvero la lista dei nomi dei file disponibili per la condivisione;
- L'invio del messaggio di risposta al comando `get` contenente il file richiesto, se presente, od un opportuno messaggio di errore;

- La ricezione di un messaggio `put` contenente il file da caricare sul server e l'invio di un messaggio di risposta con l'esito dell'operazione.

Funzionalità del client

Il client, di tipo concorrente, deve fornire le seguenti funzionalità:

- L'invio del messaggio `list` per richiedere la lista dei nomi dei file disponibili;
- L'invio del messaggio `get` per ottenere un file;
- La ricezione di un file richiesta tramite il messaggio di `get` o la gestione dell'eventuale errore;
- L'invio del messaggio `put` per effettuare l'upload di un file sul server e la ricezione del messaggio di risposta con l'esito dell'operazione.

Trasmissione affidabile

Lo scambio di messaggi avviene usando un servizio di comunicazione non affidabile.

Al fine di garantire la corretta spedizione/ricezione dei messaggi e dei file sia i client che il server implementano a livello applicativo un protocollo TCP-like con controllo di congestione.

I candidati possono usare un qualsiasi algoritmo (purché ben motivato) che adotti una finestra di spedizione variabile che aumenta in caso di assenza di perdite e diminuisce in caso di perdita di pacchetti.

Per simulare la perdita dei messaggi in rete (evento alquanto improbabile in una rete locale per non parlare di quando client e server sono eseguiti sullo stesso host), si assume che ogni messaggio sia scartato dal mittente con probabilità p.

La probabilità di perdita dei messaggi p, e la durata del timeout T, sono tre costanti configurabili ed uguali per tutti i processi.

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

I client ed il server devono essere eseguiti nello spazio utente senza richiedere privilegi di root.

Il server deve essere in ascolto su una porta di default (configurabile).

Scelta e consegna del progetto

Il progetto può essere realizzato da un gruppo composto al massimo da 3 studenti.

Per poter sostenere l'esame nell'A.A. 2022/23, è necessario prenotarsi per il progetto, comunicando al docente i nominativi ed indirizzi di e-mail dei componenti del gruppo.

Per ogni comunicazione via e-mail è necessario specificare `[IW23]` nel subject della mail.

Eventuali modifiche relative al gruppo devono essere tempestivamente comunicate e concordate con il docente.

La consegna del progetto deve avvenire almeno dieci giorni prima della data stabilita per la discussione del progetto.

La consegna del progetto consiste in:

- un file `.zip` contenente tutti i sorgenti (opportunamente commentati) necessari per il funzionamento e la copia elettronica della relazione (in formato `.pdf`);
- la copia cartacea della relazione.

La relazione contiene:

- la descrizione dettagliata dell'architettura del sistema e delle scelte progettuali effettuate;
- la descrizione dell'implementazione;
- la descrizione delle eventuali limitazioni riscontrate;
- l'indicazione della piattaforma software usata per lo sviluppo ed il testing del sistema;
- alcuni esempi di funzionamento;
- valutazione delle prestazioni del protocollo al variare delle probabilità di perdita dei messaggi p, e della durata del timeout T (incluso il caso di T adattivo)
- un manuale per l'installazione, la configurazione e l'esecuzione del sistema.

Valutazione del progetto

I principali criteri di valutazione del progetto saranno:

- rispondenza ai requisiti;
- originalità;
- efficenza;
- leggibilità del codice;
- modularità del codice;
- organizzazione, chiarezza e completezza della relazione;
- semplicità di installazione e configurazionedel software realizzato in ambiente Linux.

README

UDP-ReliableFileTransferProtocol per Ingegneria di Internet e Web 2022-2023

di Lorenzo Casavecchia **<** lnzcsv@gmail.com **>**

Presentazione del progetto

Questo progetto consiste nella realizzazione di un sistema per il trasferimento affidabile di file tra un client e un server, utilizzando come protocollo di trasporto UDP e rispettando le linee guida imposte per il progetto previsto nel corso di **Ingegneria di Internet e Web dell'A.A. 2022-2023**

In questo archivio sono presenti:

- la descrizione della soluzione proposta e della sua implementazione (vedere `DESCRIPTION.md` sotto la cartella `doc`)
- i codici sorgenti utili per l'esecuzione di entrambi client e server (sotto la cartella `src`)
- una descrizione dei problemi architetturali e implementativi riscontrati fino la versione corrente del sistema (vedere `ISSUES.md` sotto la cartella `doc`)
- la traccia originale della consegna del progetto (vedere `ASSIGNMENT.md` sotto la cartella `doc`)

Servizi offerti

Come da specifica, l'applicazione permette il trasferimento file tra un client e un server
Le richieste, generate sempre dal client, possono essere di tipo

- `list` per richiedere una lista dei file correntemente posseduti dal server
- `get <nome>` per richiedere il file `<nome>` correntemente posseduto dal server
- `put <nome>` per caricare il file `<nome>` posseduto dal client, nella cartella dei file del server

Installazione

Per la generazione degli eseguibili è necessario:

1. Cambiare cartella di lavoro ad `src`, per esempio eseguendo

```
cd ./src
```

2. Eseguire

   - `make client` o `make clientd` per la compilazione dell'applicazione client
   - `make server` o `make serverd` per la compilazione dell'applicazione server

   dove `client` e `server` corrispondono a versioni del client e server che non gestiscono dinamicamente i timeout di ricezione / spedizione, mentre `clientd` e `serverd` prevedono l'aggiornamento dei timeout in base all'evoluzione dei ritardi misurati da client e server

   È possibile inoltre generare tutti gli eseguibili con `make all`, che risulterà nella presenza dei file `client`, `clientd`, `server` e `serverd` all'interno della cartella `src`

   Infine tutti gli eseguibili possono essere rimossi con `make clean`
3. Avviare gli eseguibili così generati (`./client` o `./clientd` per il client, `./server` o `./serverd` per il server)
   All'interno di una stesso spazio di lavoro è possibile generare ed eseguire ambi client e server in quanto il codice ed i file previsti per il funzionamento del client e del server risiedono rispettivamente nelle cartelle `./src/client-side` e `./src/server-side`
4. Nel caso sia stata avviata una versione del client (`./client` o `./clientd`) sarà allora possibile inviare una richiesta al server eseguendo

   - `list` per ottenere una lista dei file nella cartella del server
   - `get <nome>` per ottenere dalla cartella del server il file `<nome>`
   - `put <nome>` per caricare nella cartella del server il file `<nome>`

   Nello specifico `list` genererà un file `list.txt` nella cartella `./src/client-side/server_files`

   I file (o la lista dei file) richiesti dal client verranno salvati in `./src/client-side/server_files` mentre i file caricati dal client tramite `put` dovranno essere salvati in `./src/client-side/server_files` prima dell'esecuzione del comando

   Questo significa che per il client tutti i file, caricati o scaricati, risiederanno nella cartella `./src/client-side/server_files`

   Un simile ragionamento è applicato al server: qualora venisse eseguita una versione del server

   - tutti gli aggiornamenti dei suoi file dovuti da richieste `put` di un client saranno visibili nella cartella `./src/server-side/server_files`

- tutte le richieste di tipo `get` effettuate da un client dovranno essere precedute dal caricamento in `./src/server-side/server_files` dei file custoditi dal server
- nel caso di richieste `list` in `./src/server-side/server_files` verrà creato un file `list.txt` contenente la lista dei file correntemente posseduti dal server

  Nel caso in cui venga effettuata una richiesta di `get` di un file non posseduto dal server, il server notificherà il client con un codice di errore (per ulteriori dettagli consultare `DESCRIPTION.md`)

5. Per terminare l'applicazione premere `<Ctrl>+C`

Altri parametri di esecuzione come la durata del timeout, la politica di aggiornamento del timeout e molti altri possono essere manualmente modificati dal file `defs.h` nella cartella `./src/client-side` o `./src/server-side` rispettivamente per le applicazioni client e server

I file `defs.h` sono correntemente identici ma logicamente distinti: applicando modifiche alla `defs.h` del client non porterà cambiamenti alla `defs.h` del server (e viceversa)

Sistema

L'applicazione è stata sviluppata e provata con le seguenti specifiche di sistema

- sistema operativo `Xubuntu 22.04.2 LTS x86_6`
- processore `AMD A8-7410 APU with AMD Radeon R5 Graphics (4) @ 2.200GHz`
- compilatore `gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0`
- debugger `GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1`
- memory error detector `valgrind-3.18.1`
- content tracker `git version 2.34.1`
- grafici `gnuplot 5.4 patchlevel 2`

Servizi non previsti

1. Il sistema non prevede la visualizzazione a schermo dei file scambiati: terminata l'applicazione dovrà essere l'utente ad aprire e visionare il contenuto dei file in questione, per esempio eseguendo

   ```
   less ./src/client-side/server_files/<nome>
   ```

   oppure

   ```
   less ./src/server-side/server_files/<nome>
   ```

   rispettivamente per client e server

2. Il sistema inoltre non preve la creazione o il caricamento di file da tastiera a tempo di esecuzione: il caricamento dei file in `server_files` deve essere effettuata prima dell'avvio dell'applicazione per esempio copiando un file da un altra cartella
3. Il sistema non prevede il rimpiazzo dei file originali con le loro copie grezze tantomeno l'eliminazione automatica di file vuoti (possibile risultato di un'istruzione `list`, `get` o `put` fallita)
4. Infine il sistema non prevede il trasferimento di file arbitrariamente grandi e la taglia massima consentiva è sull'ordine dei mega byte (per maggiori dettagli consultare `DESCRIPTION.md`)

DESCRIPTION

Descrizione della soluzione proposta

di Lorenzo Casavecchia **<** lnzcsv@gmail.com **>**

I file

L'header `defs.h`

Nei file `defs.h` vengono definiti i parametri di esecuzione, le variabili di controllo e le funzioni comuni al client e server per la gestione del trasferimento dei file

L'insieme dei parametri di esecuzione è costituito da tutte le macro definite nella prima parte di `defs.h` e comprendono

- l'indirizzo IP e numero di porta di default del server

```
#define UDP_RFTP_SERV_IP
#define UDP_RFTP_SERV_PT
```

- i valori base, minimo e massimo per la taglia della finestra di spedizione

```
#define UDP_RFTP_BASE_SEND_WIN
#define UDP_RFTP_MIN_SEND_WIN
#define UDP_RFTP_MAX_SEND_WIN
```

- la taglia della finestra di ricezione

```
#define UDP_RFTP_MAX_RECV_WIN
```

- i valori base, minimo, massimo del timeout così come il connection timeout (in microsecondi)

```
#define UDP_RFTP_BASE_TOUT
#define UDP_RFTP_MIN_TOUT
#define UDP_RFTP_CONN_TOUT
```

- la politica di aggiornamento del timeout

```
#define UDP_RFTP_MULT_TOUT(t)       (9 * (t) / 8)
#define UDP_RFTP_AVRG_TOUT(t, T)    (1 * (t) / 2 + 1 * (T) / 2)
#define UDP_RFTP_UPDT_TOUT(t, T)    (UDP_RFTP_AVRG_TOUT(t, T))
```

Le variabili di controllo vengono modificate a tempo d'esecuzione e comprendono

- informazioni sullo stato del trasferimento del file
  - `pckt_count`: il numero totale di pacchetti da trasferire
  - `akcd_pckts` ed `ackd_wins`: il numero di pacchetti e finestre riscontrati
  - `file`: identificatore del file
- informazioni sulla finestra
  - `win`: taglia attuale
  - `estimated_win`: taglia stimata dell'altro attore
  - `base_prev_win`, `base_next_win`: i limiti inferiori e superiori della finestra attuale
- i timer per il timeout
  - `set_timer`: impostare il timer
  - `cancel_timer`: interrompere il timer
- altro
  - `rev_progressive_id` e `rel_progressive_id`: gli identificatori progressivi dell'ultimo messaggio (assoluto e relativo alla finestra)
  - `acks_per_pckt`: il numero di porzioni riscontrate nell'ultimo pacchetto

I file `defs-client.h`, `defs-server.h`, `client.c` e `server.c`

`defs-client.h` e `defs-server.h` contengono le funzioni

```
void UDP_RFTP_generate_recv(char* fname);
void UDP_RFTP_generate_put(char* fname);
void UDP_RFTP_handle_put(char* fname);
void UDP_RFTP_handle_recv(char* fname);
```

responsabili della creazione dei processi e della socket con cui verrà gestita la ricezione o spedizione di un contenuto

Le funzioni `generate` vengono usate dal client mentre le funzioni `handle` dal server anche se le logiche del loro funzionamento sono piuttosto simili (un server che gestisca una richiesta `put` si comporterà in modo analogo ad un client che gestisca una `get`)

Di fatto la differenza sostanziale tra `UDP_RFTP_generate_recv` e `UDP_RFTP_handle_put` (similmente tra le altre due funzioni) è la gestione della connessione, quindi la sequenza dei

primi messaggi trasmessi

`client.c` e `server.c` contengono le funzioni main rispettivamente del client e del server, quindi il codice eseguito dal processo padre all'esecuzione dell'applicazione

`client.c` allocherà memoria per le variabili usate dai processi figlio, rimane in attesa per l'immissione di comandi dall'utente e in base al comando inserito invocare la corrispondente `generate`

Prima di mettersi in attesa verifica se uno dei processi generati dovesse aver terminato e, nel caso in cui il numero di richieste attive sia maggiore di `UDP_RFTP_MAXCLIENT` rimarrà in attesa finché uno di essi termini

Il comportamento di `server.c` é simile, con la distinzione nell'attesa di un messaggio di connessione da un client invece dell'immissione di un comando dall'utente

Messaggi

L'interazione tra client e server avviene per scambio di datagrammi UDP il cui campo dati è una sequenza di caratteri separate da punti e virgola `;`

Le stringhe così delimitate costituiscono i campi del messaggio

Client e server dispongono ciascuno di una struttura

- `send_msg` in cui il mittente caricherà i campi del messaggio da inviare al destinatario
- `recv_msg` in cui verranno inseriti i valori dei campi dell'ultimo messaggio ricevuto dal mittente
- `addr` per l'indirizzo del destinatario del messaggio da inviare o dell'indirizzo del mittente dell'ultimo messaggio ricevuto

La generazione, trasmissione e ricezione dei datagrammi è gestita rispettivamente dalle funzioni

```
void UDP_RFTP_send_pckt(int signo);
void UDP_RFTP_recv_pckt(void);
```

dove `UDP_RFTP_send_pckt` converte la struttura `send_msg` nella sequenza di caratteri delimitata da `;` (invocando `void UDP_RFTP_msg2str(UDP_RFTP_msg* msg, char* str)`) e la invia all'indirizzo specificato in `addr`, mentre `UDP_RFTP_recv_pckt` attende la ricezione di un pacchetto e ne salva i campi in `recv_msg` (invocando `void UDP_RFTP_str2msg(char* str, UDP_RFTP_msg* msg)`)

`send_msg` e `recv_msg` sono entrambe strutture `UDP_RFTP_msg` e dispongono di campi

- `port_no` per il numero di porta su cui il mittente vorrà essere contattato dal ricevente (virtualmente usato solo dal server all'inizializzazione della connessione)
- `msg_type` per il tipo di comunicazione in atto (`UDP_RFTP_LIST`, `UDP_RFTP_GET`, `UDP_RFTP_PUT` per `list`, `get`, `put` ed `UDP_RFTP_ERR` qualora il file richiesto non esistesse o fosse inaccessibile infine `0` qualora non sia stato ricevuto nessun nuovo messaggio)
- `progressive_id` per identificare diverse porzioni del file in transito (simile al numero di sequenza in protocolli TCP-like)
- `data` per la porzione del file in transito oppure per specificare altre informazioni riguardo il file trasferito (ad inizializzazione viene usato per dichiarare il nome e la taglia del file)

La verifica dell'indirizzo e il numero di porta del mittente alla ricezione di un pacchetto viene gestita all'esterno di `UDP_RFTP_recv_pckt` confrontandoli con i valori delle strutture `client_addr` e `server_addr` definiti e impostati all'inizializzazione della connessione

Interazione client-server

Messaggio di connessione

Le operazioni di `list`, `get` e `put` vengono generate dal client che, ricevuto il comando dall'utente, genera ed inoltra al server un messaggio di connessione

Un messaggio di connessione è un messaggio generato solamente dal client con

- `msg_type` pari al tipo di servizio richiesto (`UDP_RFTP_LIST`, `UDP_RFTP_GET`, `UDP_RFTP_PUT` per `list`, `get`, `put`)
- `progressive_id` pari a `0`
- `data` contentente specifiche riguardo il servizio richiesto
  - vuoto per comandi `list`
  - il nome del file desiderato per comandi `get`
  - il nome del file ed il numero di messaggi che il server dovrà ricevere per comandi `put`

Inviato un messaggio di connessione il client rimarrà in attesa di una risposta dal server oppure, allo scadere di un timeout impostato prima dell'invio, reinviare la richiesta al server (il timeout è attualmente pari al timeout base impostato su tutti i pacchetti)

Risposta alla connessione

Ricevuta la richiesta di connessione dal client il server dovrà

- creare un processo figlio per la gestione della richiesta
- generare una socket associata a quella richiesta

- comunicare al client del numero di porta della socket generata

Nel comunicare il numero di porta al client il processo figlio del server dovrà inviare sulla socket generata dal padre (con numero di porta di default) un messaggio che nel campo `port_no` abbia il numero di porta della socket del figlio

In questo modo il client riceverà la risposta alla sua richiesta di connessione dalla socket che aveva originariamente contattato e con una direttiva esplicita a chi contattare d'ora in poi per continuare la comunicazione

Gli altri campi della risposta del server dipendono dal tipo di richiesta effettuata e dal valore del campo dati

Per evitare che un client invii un messaggio di connessione senza portarla avanti, il processo figlio generato rimarrà in attesa non oltre una quantità di tempo specificata dal timeout di connessione `UDP_RFTP_CONN_TOUT` (attualmente più grande del valore del timeout base `UDP_RFTP_BASE_TOUT`)

Nel caso in cui il figlio dovesse ricevere una risposta la comunicazione verrà portata avanti, altrimenti il processo figlio terminerà

È importante osservare che per ogni richiesta di un client il server generi una sola risposta quindi se quest'ultima non dovesse raggiungere il client, sarà quest ultimo a dover generarme una nuova (mentre il server si limiterà a terminare l'esecuzione del processo generato)

Riscontri selettivi

Il meccanismo con cui l'attore in ricezione comunichi all'attore in spedizione quali porzioni del file siano state correttamente ricevute è simile al selective repeat: l'attore in ricezione instaura una finestra dei pacchetti ordinati ricevuti o non ancora ricevuti e un buffer della stessa dimensione in cui verranno, di ricezione in ricezione, caricate le porzioni del file

A tal scopo a ciascuna porzione del file viene associato un `progressive_id` quindi un identificatore progressivo della porzione correntemente trasmessa

Il `progressive_id` è, rispetto all'insieme delle porzioni del file, un numero assoluto e non è relativo alla corrente finestra del mittente (il che significa che se il file in questione potesse essere trasmesso in N messaggi, il `progressive_id` potrà variare tra 1 ed N anche se la finestra di spedizione preveda l'invio simultaneo di $n < N$ messaggi)

Il `progressive_id` di una porzione in trasmissione coincide con il `progressive_id` del messaggio contenente quella porzione di file

Questa scelta introduce un limite massimo al valore di `progressive_id`: supponendo `progressive_id` venga rappresentato come `uintx_t` quindi un numero intero senza segno a `x` bit allora il suo valore massimo sarà $2^{\texttt{x}} - 1$

Se `x == 16` (come da implementazione attuale) `progressive_id` potrà valere al più $2^{16} - 1 \approx 64k$, quindi potranno essere trasmessi al più $64k$ messaggi (un messaggio racchiude una porzione di file di $1k$ byte per un totale di circa $64M$ byte)

Un attore in ricezione che riceva una porzione di file dovrà

- verificare che il `progressive_id` non appartenga ad una porzione precedentemente riscontrata (quindi sia maggiore del `progressive_id` della prima porzione di file nella corrente finestra di ricezione e che la relativa porzione non sia stata marcata come ricevuta)
- verificare che il `progressive_id` non appartenga ad una porzione della finestra di ricezione successiva a quella attuale (`progressive_id` dovrà essere minore o uguale al `progressive_id` dell'ultima porzione della finestra)
- inviare un riscontro positivo (ACK) delle porzioni correttamente ricevute dell'attuale finestra

Il riscontro è un messaggio con

- `msg_type` del tipo di richiesta
- `progressive_id` pari a `-1`
- `data` contenente una lista degli identificatori progressivi dei pacchetti riscontrati della finestra corrente separati da `,`

Il riscontro verrà inviato

- allo scadere del timeout di ricezione
- nel caso di `progressive_id` di porzioni di finestre precedenti
- qualora vengano ricevute tutte le porzioni previste dalla finestra
  (in tal senso è simile al meccanismo degli ACK ritardati implementato in alcune versioni di TCP)

D'altro canto la finestra di ricezione potrà essere traslata e ammettere nuove porzioni solo al riempimento di quella precedente

In tal senso possiamo dire che il meccanismo di riscontro delle porzioni dei file è

- di tipo stop and wait nel contesto delle finestre di ricezione
- di tipo selettivo e cumulativo nel contesto delle porzioni in una finestra

Questa scelta seppur permetta un'identificazione più segmentata della trasmissione del file, è più debole del classico riscontro selettivo in quanto non prevede la traslazione della finestra al riscontro delle prime porzioni della finestra e invece attende che la finestra venga ricevuta completamente (a meno di timeout)

Ritrasmissione

La ritrasmissione permette ad un attore in spedizione di inviare porzioni di file non ancora riscontrate a valle di un timeout

Il timeout viene rigenerato

- a valle di un timeout precedente
- a seguito di un evento che lo ha interrotto (la ricezione di nuove porzioni di file, il riempimento della finestra oppure la ricezione di un nuovo riscontro)

Il valore del timeout varia in funzione dei ritardi osservati dalla rete, quindi nell'intervallo tra l'invio di una sequenza di messaggi e la ricezione di messaggi inerenti a quelli spediti (la ritrasmissione di porzioni di file e la ricezione di un nuovo riscontro oppure l'invio di un riscontro con la trasmissione di nuove porzioni)

La misurazione dei ritardi è effettuata dalle funzioni

```
void UDP_RFTP_start_watch(void);
void UDP_RFTP_stop_watch(int update);
```

che avviano e interrompono un timer nei momenti sopra descritti

Se `update == UDP_RFTP_SET_WATCH` allora l'intervallo misurato verrà applicato alla legge di controllo di aggiornamento dei timeout ed impostato come prossimo timeout, altrimenti verranno resettate le variabili che misuravano il tempo passato

La legge di controllo prevede incrementi moltiplicati in timeout e una media pesata tra il timeout precedente e il ritardo misurato, ma comunque entro `UDP_RFTP_MIN_TOUT` e `UDP_RFTP_BASE_TOUT`

$$\texttt{tout}_{k+1} = \begin{cases} q \cdot \texttt{tout}_k & \text{se timeout} \\ a \cdot \texttt{tout}_k + (1-a) \cdot \texttt{rtt}_k & \text{altrimenti} \end{cases}$$

dove $q > 1$ ed $a < 1$ (da corrente implementazione $q = 1.125$, $a = 0.5$)

Questa scelta può essere motivata dal fatto che le finestre vengano traslate interamente quindi se dovesse avvenire un timeout è probabile (a meno di perdite o dimensioni di finestra

sufficentemente grandi) che la risposta non sia ancora arrivata oppure non sia stata inviata (la dinamica di aggiornamento del timeout è uguale per client e server)

La ritrasmissione prevede l'invio di tutte le porzioni di file correntemente non riscontrate comprese nell'attuale finestra

La finestra di ricezione ha taglia fissa mentre quella di spedizione varia entro `UDP_RFTP_MIN_SEND_WIN` e `UDP_RFTP_MAX_SEND_WIN` secondo la legge di controllo

$$\texttt{win}_{k+1} = \frac{\texttt{estimated\_win}_k}{1 + \frac{\texttt{retrans\_count}_k}{\texttt{estimated\_win}_k}}$$

con

- `win` la taglia della k-esima finestra
- `estimated_win` la stimata taglia della finestra dell'altro attore (misurata sul numero massimo di ACK per pacchetto ricevuti)
- `retrans_count` il numero di ritrasmissioni alla finestra k

Notare che `retrans_count == 0` rende `win == estimated_win` (le finestre di ricezione e spedizione si corrispondono), mentre se `retrans_count > estimated_win` allora la rete si considererà congestionata e `win < estimated_win` (decremento moltiplicativo)

Siccome la politica di riscontro delle finestre è stop and wait e la finestra di ricezione è fissa in taglia, l'incremento della finestra di spedizione oltre il valore di `estimated_win` porterà quasi sempre alla ritrasmissione delle porzioni in eccesso (l'unico scenario in cui ciò non accadrà è la ricezione in ordine e senza perdite di tutti i pacchetti)

In tal senso l'incremento della finestra avviene solo

- a seguito di una riduzione da ritrasmissioni
- qualora l'attore in ricezione invii un riscontro che comprenda più pacchetti

Qualora l'architettura dovesse prevedere finestre di ricezioni variabili la legge di controllo proposta proverebbe a dimensionare le taglie delle finestre allo stesso modo

Chiusura della connessione

Un attore che abbia ricevuto tutte le porzioni o tutti i riscontri alle porzioni del file avvierà la sequenza di chiusura alla connessione invocando

```
void UDP_RFTP_bye(void);
```

La chiusura alla connessione avviene semplicemente inviando dei messaggi con `msg_type` pari a quello del servizio e `progressive_id` negativo e attendendo la stessa risposta dal server

Gli altri campi possono contentere niente oppure, nel caso di un attore in ricezione, il riscontro all'ultima finestra

Qualora l'altro attore non dovesse rispondere al messaggio entro un timeout (e dopo un numero `UDP_RFTP_MAXBYE` di tentativi), l'attore in chiusura terminerà

Nel caso in cui un attore dovesse terminare prima di quanto atteso (crash, errore o perché ha ricevuto le porzioni a cui era interessato) è previsto un meccanismo di terminazione per inattività:

- per attori in ricezione la variabile `retrans_count` tiene conto del numero di mancate ricezioni (`msg_type == 0` dalla `UDP_RFTP_recv_pckt`) e termina l'esecuzione dell'attore qualora `retrans_count >= win * UDP_RFTP_MAXRETRANS` dove `win` è la corrente taglia della finestra
- per attori in spedizione `retrans_count` conta il numero di porzioni ritrasmesse a seguito di un timeout (il controllo per la terminazione rimane invariato)

Istanze di esecuzione

Per illustrare l'evoluzione nel tempo dei parametri del sistema all'invocazione di `UDP_RFTP_pckt` corrisponderà anche un'operazione di stampa su file

L'ultimo client eseguito stamperà per colonne i valori dei suoi parametri in `.clientlog` mentre il server su `.serverlog`

I valori in `.clientlog` e `.serverlog` possono essere passati a `gnuplot` che, specificate le colonne interessate, stamperà un grafico nel tempo del loro andamento

Le colonne sono così disposte

```
ackd_pckts  ackd_wins   win estimated_win   retrans_count   acks_per_pckt
set_timer.it_value.tv_sec   set_timer.it_value.tv_usec
```

dove `set_timer` è il timer del timeout, mentre il comando base utilizzato per generare i grafici è

```
plot <nome del file> using <numero della colonna>
```

Nella cartella `doc/plots` sono presenti grafici generati a partire da alcune istanze di esecuzione del sistema

Ciascun file è stato nominato in base alla convenzione

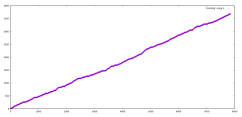
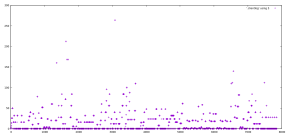
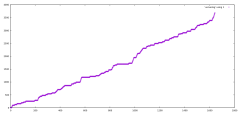
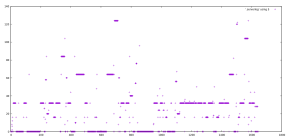
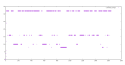
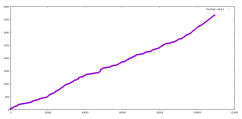
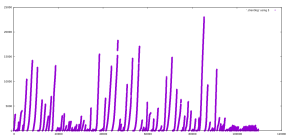
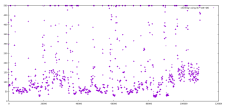
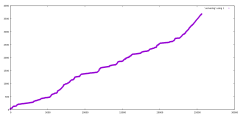
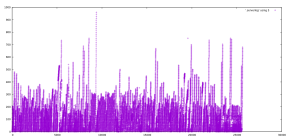
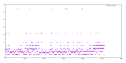
```
<attore>-<tipo di richiesta>-loss<percentuale di perdita>-<parametro>
```

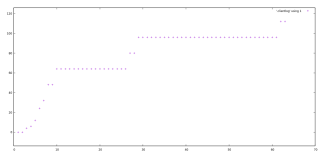
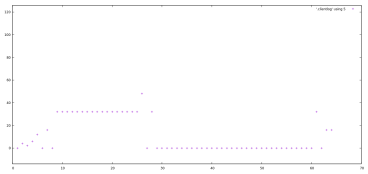
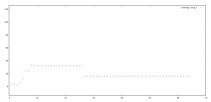
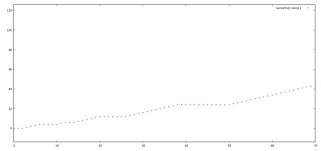
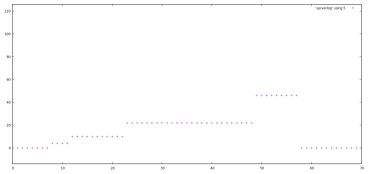
Per tutte le simulazioni sono stati utilizzati i parametri impostati nella versione corrente di questo archivio, eccezione fatta per le simulazioni con tasso di perdita `20` e `70` per cui è stata disattivata la terminazione per inattività

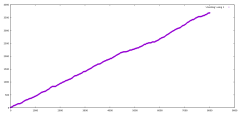
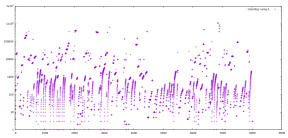
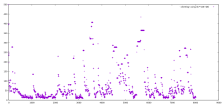
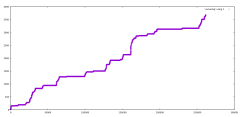
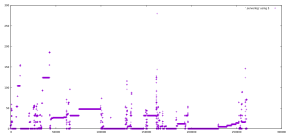
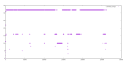
Inoltre il grafico del `retrans_count` di `clientd` della `get` (`UDP_RFTP_LOSS_RATE == 20`) è in scala logaritmica

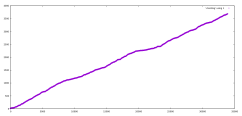
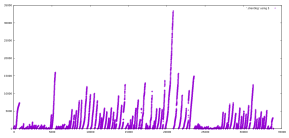
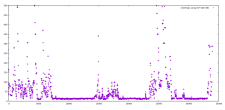
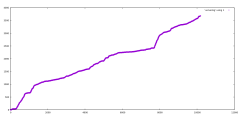
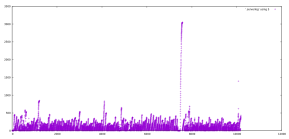
Il file trasmesso in `get` è circa di $3.7M$ byte mentre il file in `put` è di $130k$ byte

Per una corretta visualizzazione dei grafici consultare `LorenzoCasavecchiaIIW22-23.pdf`, aprire i grafici in `doc/plots` oppure aprire questo file in `Obsidian (v1.3.7)`

`get` (`UDP_RFTP_LOSS_RATE == 0`)	**`ackd_pckts`**	**`retrans_count`**	**`win`**	**`set_timer`**
`client`				
`server`				
`clientd`				
`serverd`				

put (UDP_RFTP_LOSS_RATE == 0)	ackd_pckts	retrans_count	win
client			
server			

get (UDP_RFTP_LOSS_RATE == 20)	ackd_pckts	retrans_count	win	set_timer
clientd				
serverd				

get (UDP_RFTP_LOSS_RATE == 70)	ackd_pckts	retrans_count	win	set_timer
clientd				
serverd				

ISSUES

Problemi nella soluzione proposta

di Lorenzo Casavecchia **<** lnzcsv@gmail.com **>**

Problemi architetturali

1. L'idea di suddividere logicamente e inviare un file in porzioni della stessa taglia non è utilizzata al meglio in quanto lo scambio di contenuti tra client e server avviene in modo sequenziale
   L'attore in trasmissione inizia l'invio del contenuto a partire dalla sua prima porzione, seguita dalla seconda, la terza, e così via fino al riempimento della finestra di spedizione

   L'attore in ricezione si aspetterà di ricevere le prime porzioni del contenuto fino al riempimento della finestra di ricezione
   Da qui in poi, per ambi attori, il trasferimento del contenuto avverrà sempre trasmettendo / ricevendo le porzioni corrispondenti alla finestra di trasmissione / ricezione successiva alla precedente

   Sulla base di queste osservazioni si può notare come non sia possibile

   - richiedere specifiche porzioni del contenuto in questione e ignorarne altre
   - interrompere momentaneamente il trasferimento del contenuto e poi riavviarlo in un secondo momento (in quanto richiederebbe di eseguire da capo la sequenza di messaggi per invio e riscontro di tutte le porzioni già ottenute)

   Le modifiche che potrebbero permettere l'implementazione di questi servizi sono
   - il riscontro delle porzioni di un contenuto ricevute utilizzando NACK invece degli ACK
   - una gestione delle finestre e buffer di spedizione / ricezione più dinamica (che permetta il caricamento nei buffer di un sottoinsieme delle porzioni del contenuto logicamente vicine a quella non riscontrata)
   - un meccanismo di logging dello stato del trasferimento del contenuto (quindi uno storico delle porzioni del contenuto in questione correttamente o non ricevute / riscontrate)
2. La gestione concorrente di più client (lato server) o di più richieste (lato client) basata su multi-processamento non è scalabile
   La soluzione proposta prevede che un client possa generare concorrentemente non più di

N richieste e che il server possa gestire concorrentemente non più di M client e per ciascuna delle N o M richieste venga generato un processo

Per valori di N o M arbitrariamente grandi questo porterebbe ad un numero di processi ugualmente grande che potrebbero in un qualsiasi istante risiedere in CPU

Se il tempo di attesa per risiedere in CPU dovessere essere sufficentemente grande l'altro attore potrebbe superare il numero di timeout massimi consentivi e terminare per inattività

Un'idea più scalabile per la gestione concorrente di client e server potrebbe consistere nell'instaurare 3 threads

- uno in ricezione delle richieste / risposte verso l'altro attore
- uno di gestione delle richieste
- uno di trasmissione delle risposte / richieste verso l'altro attore

3. Come evidenziato dalle istanze di esecuzione in `DESCRIPTION.md` le varie parti che compongono il sistema non funzionano bene insieme, specialmente in presenza di timeout dinamici
   Le istanze di esecuzione hanno inoltre evidenziato che anche in assenza di perdite vi siano ritrasmissioni e il motivo è che la gestione del timeout non è adeguata e presenta bug

Problemi implementativi

Ad aggiornamento odierno l'applicazione presenta i seguenti bug

- Gli attori in ricezione non inviano riscontri per l'ultima finestra di ricezione
- Processi figlio tendono a rimanere in stato `defunct` anche a seguito del loro riscontro tramite `wait` o `waitpid`
- Gli attori in trasmissione non impostano correttamente i valori dei buffer dell'ultima finestra di spedizione (se N fosse il numero di pacchetti che dovrebbero essere trasmessi, l'attore in trasmissione potrebbe inviare pacchetti fino ad $N+1$ dove l'($N+1$)-esimo pacchetto ha campo `data` vuoto)
- Anche con probabilità di perdita di pacchetti nulla un client potrebbe inviare più volte una stessa richiesta al server prima che quest ultimo abbia modo di rispondergli (ed è sicuramente dovuto dalla gestione dei timeout all'invio della richiesta)
- Nonostante la politica di aggiornamento del timer sia la stessa del client, si osserva che il timer del server tenda ad assestarsi al valore minimo e a restarci (questo bug credo sia responsabile dei valori terrificanti nel `retrans_count` del client e del server)

Ad aggiornamento odierno l'applicazione non presenta le seguenti funzionalità

- Non sono presenti meccanismi di file locking per attori che agiscono su uno stesso file

Infine seppur funzionale l'implementazione offerta non è ottimizzata, presenta molteplici ridondanze e la maggior parte delle variabili utili al funzionamento sono condensate nei file `defs.h`