

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_rcv(char* fname);
void UDP_RFTP_generate_put(char* fname);
void UDP_RFTP_handle_put(char* fname);
void UDP_RFTP_handle_rcv(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_rcv` 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` è 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_pkt(int signo);  
void UDP_RFTP_recv_pkt(void);
```

dove `UDP_RFTP_send_pkt` 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_pkt` 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` contenente 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 generarne 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 $2^x - 1$

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`

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

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

sufficientemente 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

$$\text{win}_{k+1} = \frac{\text{estimated_win}_k}{1 + \frac{\text{retrans_count}_k}{\text{estimated_win}_k}}$$

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 contenere 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)`

<code>get (UDP_RFTP_LOSS_RATE == 0)</code>	<code>ackd_pkts</code>	<code>retrans_count</code>	<code>win</code>	<code>set_timer</code>
<code>client</code>				
<code>server</code>				
<code>clientd</code>				
<code>serverd</code>				

