Reliable UDP

Simone Falvo

5 gennaio 2018

Sommario

Progetto e realizzazione di un'applicazione FTP distribuita di tipo client-server, utilizzando UDP come protocollo di trasporto. L'applicazione implementa i classici comandi di un'applicazione FTP: LIST, GET e PUT, garantendo la possibilità di eseguire tali operazioni in parallelo per ogni client e l'integrità di messaggi e file scambiati.

Indice

1 Architettura

Il sistema è composto da un server e molteplici client che scambiano dati in parallelo con il server, pertanto è necessario un server che sia in grado di gestire tutti i client contemporaneamente. Per prima cosa quindi, verrà descritto il modello di server scelto, poi verranno descritti i dettagli architetturali dettati dai requisiti funzionali.

1.1 Scelta del modello

La gestione contemporanea di più client viene realizzata da un server di tipo multi-processo.
La scelta di tale modello è stata effettuata tenendo conto dei seguenti criteri:

- Semplicità del codice
- Tolleranza ai guasti
- Efficienza
- Utilizzo delle risorse di sistema

Un'applicazione di questo tipo coinvolge file di grosse dimensioni, che vanno dalle centinaia di megabyte fino all'ordine dei gigabyte. Il trasferimento di tali file richiede tempi consideroveli dell'ordine dei minuti, per questo motivo è di fondamentale importanza che il trasferimento subisca meno intoppi possibili, in particolere è necessario che un fallimento riguardante la connessione con un client non coinvolga le altre connessioni. Si immagini, ad esempio, lo sconforto di un utente che dopo decine di minuti di download, si vede cadere la connessione a seguito di un problema sconosciuto e che non dipende da lui.
Questo fatto rende la gestione dei guasti il criterio dominante, pertanto si è scelto il modello di server a processi in cui le connessioni vengono gestite da singoli processi "isolati", nel senso che il fallimento di un processo non inficia in nessun modo l'esecuzione degli altri processi.

Se, con i processi, da un lato si ottiene tolleranza ai guasti, dall'altro si perde in efficienza, poiché si introduce un overhead significativo per la creazione dei processi ed il cambio di contesto.
Nonostante si sarebbe potuto ridurre l'overhead dovuto alla creazione dei processi tramite lo sviluppo di un modello a preforking, si è scelto di ignorare questa possibilità perché ciò avrebbe reso il codice più complesso a fronte di un miglioramento dei tempi di risposta trascurabile, infatti tali tempi sono dominati dal tempo di trasferimento del file che è di vari ordini di grandezza superiore a quello di generazione del processo.

1.2 Stratificazione del sistema

Per realizzare un'applicazione basata su UDP che garantisca una comunicazione affidabile è stato necessario introdurre a livello applicativo uno strato che si ponesse logicamente tra applicazione e livello di trasporto e che astraesse un protocollo di trasporto affidabile.
Tale strato di "pseudo-trasporto", (o livello di trasporto virtuale), fornisce un'interfaccia all'applicazione per mezzo delle funzioni *rdt_ send* e *rdt_ recv* simile a

quella offerta dalle API *sendto* e *recvfrom* utilizzate per UDP, quindi tutto ciò che riguarda l'implementazione necessaria a garantire affidabilità è trasparente al livello applicativo puro.
Tuttavia la trasparenza non è del tutto completa perché, affinché questo strato di trasporto virtuale funzioni, è necessario inizializzare le strutture che ne fanno parte, sia da un lato che dall'altro della comunicazione con gli stessi parametri. Questo implica che prima di poter comunicare in modo affidabile, client e server devono accordarsi sulla scelta dei parametri, in particolare, vengono impostati all'avvio del server, per poi essere inviati ai client che vogliono essere serviti.
Un'altra caratteristica utile, è che utilizzare la *rdt_recv*, è funzionalmente identico ad effettuare una *read* da una socket in TCP, infatti è possibile effettuare letture successive dello stesso messaggio, senza che gli altri dati che si devono ancora leggere vadano persi.

2 Implementazione

2.1 Simulazione rete inaffidabile

Il progetto è stato testato ed eseguito all'interno di una rete locale, pertanto è stato necessario simulare la perdita dei pacchetti.
Ciò è stato fatto introducendo delle opportune funzioni per l'invio dei dati, le quali inviano se e soltanto se viene estratto un numero casuale maggiore di una data probabilità di perdita.

simul_udt.c

```
ssize_t udt_sendto(int sockfd, const void *buf, size_t len,
            const struct sockaddr *addr, socklen_t addrlen,
            double loss)
{
   double drand = randgen();
   ssize_t retval = len;

   // necessary flow control into a local network
   if (loss < 0.1)
      usleep(50);

   if (drand > loss)
      retval = sendto(sockfd, buf, len, 0, addr, addrlen);

   return retval;
}


ssize_t udt_send(int sockfd, void *buf, size_t size, double loss)
{
   return udt_sendto(sockfd, buf, size, NULL, 0, loss);
}


double randgen(void)
{
   static bool seed = false;
   if (!seed) {
      srand48(time(NULL));
      seed = true;
   }
```

```
    return drand48();
}
```

2.2 Connessione

La comunicazione tra client e server avviene a seguito dell'instaurazione di una connessione senza autenticazione.
Il client quando vuole connettersi al server invia un messaggio di SYN, dopodiché si mette in attesa di un messaggio di risposta (SYNACK) contenente i parametri necessari alla creazione delle strutture che gestiscono la comunicazione affidabile. Questa attesa è limitata dalla presenza di un timer di 5 secondi, allo scadere del quale avviene un nuovo tentativo di connessione reinviando il messaggio di SYN.
Ricevuto il messaggio di risposta vengono create ed inizializzate le strutture di comunicazione e la connessione risulta instaurata, da questo istante è possibile inviare comandi al server in modo affidabile.
Poiché, al momento della richiesta di connessione, non si dispone nemmeno del parametro relativo alla probabilità di perdita di un pacchetto, il messaggio di SYN viene inviato con probabilità di perdita pari al 20%.

client: instaurazione della connessione

```
    ...

/* set receiving timeout on the socket */
timeout.tv_sec = 5;
timeout.tv_usec = 0;
if (setsockopt
    (sockfd, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout))
    == -1)
    handle_error("setting socket timeout");

while (!connected) {

    /* send SYN */
    if (udt_sendto(sockfd, NULL, 0, (struct sockaddr *) addr,
        addrlen, 0.2) == -1)
        handle_error("udt_sendto() - sending SYN");
    fputs("SYN sent, waiting for SYN_ACK\n", stderr);

    /* get SYN_ACK and server connection address */
    errno = 0;
    if (recvfrom
        (sockfd, &params, sizeof(params), 0,
        (struct sockaddr *) addr, &addrlen) == -1) {
        if (errno == EAGAIN || errno == EWOULDBLOCK)
            // timeout expired
            continue;
        handle_error("recvfrom()");
    }

    connected = true;
}

/* turn timeout off */
timeout.tv_sec = 0;
timeout.tv_usec = 0;
```

```
if (setsockopt
   (sockfd, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout))
   == -1)
   handle_error("setting socket timeout");

/* set the endpoint */
if (connect(sockfd, (struct sockaddr *) addr, addrlen) == -1)
   handle_error("connect()");

/* initialize transport layer */
init_transport(sockfd, &params);

   ...
```

Il server, costantemente in attesa di richieste di connessione, alla ricezione di un messaggio di SYN, crea un processo figlio al quale affida il compito di inviare i parametri di connessione e di gestire la connessione con il client.

server: instaurazione della connessione

```
   ...

for (;;) {

   clilen = sizeof(cliaddr);
   memset((void *) &cliaddr, 0, clilen);

   /* wait for connection requests */
   errno = 0;
   if (recvfrom
      (sockfd, buf, MAXLINE, 0, (struct sockaddr *) &cliaddr,
       &clilen) == -1) {

       if (errno == EINTR)
         // signal interruption
         continue;

      handle_error("waiting for connection requests");
   }

   /* create a new proccess to handle the client requests */
   pid = fork();

   if (pid == -1)
      handle_error("fork()");

   if (!pid) {                  // child process

      /* close duplicated listen socket */
      if (close(sockfd) == -1)
         handle_error("close()");

      create_connection(&params, &cliaddr, clilen);
   }
}

   ......
```

Il processo figlio crea una nuova socket e tramite quest'ultima invia al client i parametri del protocollo di comunicazione affidabile. Poi imposta l'indirizzo del

client come destinazione prefissata, inizializza la connessione e rimane in attesa di eventuali comandi da parte del client.

connessione del nuovo processo

```
void create_connection(struct proto_params *params,
                  struct sockaddr_in *cliaddr, socklen_t clilen)
{
   int connsd;

   /* create a connection socket */
   if ((connsd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
      handle_error("socket()");

   /* set the end point */
   if (connect(connsd, (struct sockaddr *) cliaddr, clilen) == -1)
      handle_error("socket()");

   /* send SYN_ACK with protocol parameters */
   if (udt_send(connsd, params, sizeof(struct proto_params),
      params->P / 100.0) == -1)
      handle_error("udt_send() - sending SYN_ACK");

   init_transport(connsd, params);

   server_job();
}
```

Un limite di questa implementazione consiste nel fatto che il server non sa distinguere se un SYN proviene da un nuovo client oppure è un tentativo di riconnessione, per cui, se si verifica il secondo caso, il processo server d'ascolto crea un nuovo figlio quando ce n'era già uno in attesa di richieste di quel client. In questo modo se si perdono molteplici SYNACK destinati ad uno specifico client, verranno creati altrettanti processi server che rimarranno in attesa di richieste che non arriveranno mai.
Una possibile soluzione a questo problema sarebbe quella di far mantenere al server informazioni, di durata limitata, che permettano di capire se per il client che effettua la richiesta di connessione, esiste già un processo dedicato. In tal caso, basterebbe comunicare al processo figlio in questione di inviare di nuovo il messaggio di SYNACK.
Questa soluzione implicherebbe la gestione di una lista di associazioni client-pid, che andrebbe scandita ad ogni ricezione di messaggi SYN, e della comunicazione tra processo padre e processo figlio, quindi per motivi di efficenza e semplicità del codice, si è scelto di non implementarla.
Ad ogni modo un processo di connessione, se non riceve comandi per un tempo pari a 1 minuto, termina la propria esecuzione, liberando così preziose risorse. (La gestione dei processi zombie verrà descritta più avanti).

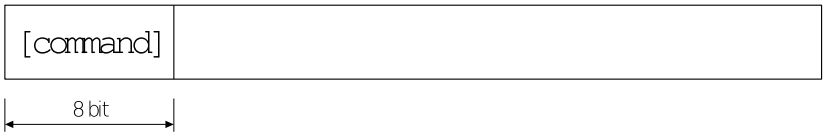
2.3 Protocollo di comunicazione

In generale, il protocollo di comunicazione prevede che il client invii messaggi di comando in cui venga specificato il tipo di comando da eseguire, il server, quindi esegue il comando e invia un messaggio di risposta.
Ogni comando prevede un trasferimento file, quindi, a seconda del tipo di comando, o nel messaggio di comando o in quello di risposta viene allegato il file, con le informazioni necessarie alla sua corretta ricezione.

Ogni messaggio di comando e di risposta è composto da campi di un numero fissato di byte contenenti le informazioni necessarie all'interpretazione del messaggio. Essendo un'applicazione distribuita, i campi vengono distinti utilizzando variabili la cui ampiezza (numero di bit) è indipendente dall'architettura della macchina, a tal proposito ogni campo informativo è composto dalle variabili definite in *stdint.h*.
I primi 8 bit di un messaggio di comando indicano sempre il tipo di comando da eseguire e, a seconda del comando, possono avere o meno ulteriori informazioni.

Costanti comandi

```
// command codes
#define LIST        0
#define GET         1
#define PUT         2

// response codes
#define GET_OK      0
#define GET_NOENT   1
#define PUT_SUCCESS 2
#define PUT_FAILURE 3
```



Una volta impostati i vari campi in un buffer, se non va inviato un file, i dati vengono passati direttamente al livello di trasporto virtuale.
Se altrimenti, va inviato un file, viene chiamata la funzione *send_file*, i campi informativi vengono considerati un header per il file ed il tutto viene frammentato (per non allocare troppa RAM) e passato al livello sottostante.
Lato opposto il ricevente analizza uno alla volta i campi informativi, poi legge il file tramite la funzione *recv_file*.

cmd_commons.c

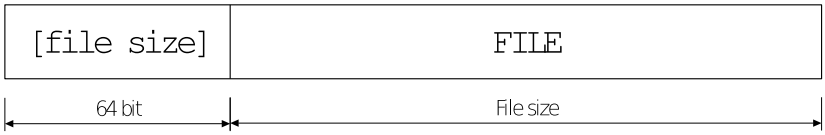
```
void send_file(int fd, void *header, size_t file_size,
            size_t header_size)
{
   int8_t buffer[MAX_BUFSIZE];
   size_t buf_size, total_size;
   unsigned int i, n;

   total_size = header_size + file_size;
   n = total_size / MAX_BUFSIZE;

   memcpy(buffer, header, header_size);

   for (i = 0; i <= n; i++) {

      // calculate last bytes to send
      if (i == n) {
         buf_size = total_size % MAX_BUFSIZE;
         if (!buf_size)
            // total size is a multiple of MAX_BUFSIZE:
            // send only n-1 chunks
```

```
            break;
        } else
            buf_size = MAX_BUFSIZE;

        if (readn(fd, buffer + header_size, buf_size - header_size)
            == -1)
            handle_error("readn() - reading file to send");

        header_size = 0;  // consider header only at the first pass

        rdt_send(buffer, buf_size);
    }
}

void recv_file(int fd, size_t size)
{
    unsigned int i, n = size / MAX_BUFSIZE;
    size_t buf_size;
    int8_t buffer[MAX_BUFSIZE];

    for (i = 0; i <= n; i++) {

        // calculate last bytes to store
        if (i == n) {
            buf_size = size % MAX_BUFSIZE;
            if (!buf_size)
                // size is a multiple of MAX_BUFSIZE:
                // receive only n-1 chunks
                break;
        } else
            buf_size = MAX_BUFSIZE;

        rdt_recv(buffer, buf_size);

        if (writen(fd, buffer, buf_size) == -1)
            handle_error("writen() - writing received file");
    }
}
```

2.4 Comando LIST

Nel messaggio di comando list non sono previsti campi aggiuntivi oltre a quello che specifica il comando.

Il server ricevuto il comando, esegue il comando di sistema *ls* redirigendo l'output sul file *file_ list.txt*, dopodiché viene impostato il messaggio di risposta con la dimensione del file (campo da 64 bit) e a seguire il file contenente la lista dei file disponibili al download.



clicmd.c

```
void cli_list()
{
    uint8_t cmd = LIST;
    uint64_t file_size;
    char *buffer;

    /* send LIST command */
    rdt_send(&cmd, sizeof(cmd));

    /* read list size */
    rdt_recv(&file_size, sizeof(file_size));

    /* allocate buffer */
    buffer = malloc(file_size);
    if (!buffer)
        handle_error("malloc()");

    /* recv file list */
    rdt_recv(buffer, file_size);

    /* print file list and free memory */
    printf("\n%s\n", buffer);
    free(buffer);
}
```

srvcmd.c

```
void srv_list(void)
{
    int fd;
    struct stat st;
    uint64_t file_size;
    uint8_t *header;
    char *filename = "file_list.txt";

    /* execute ls command */
    char *cmd = "ls > file_list.txt";
    if (system(cmd) == -1)
        handle_error("system() - executing ls command");

    /* open the destination file of the list command */
    fd = open(filename, O_RDONLY);
    if (fd == -1)
        handle_error("open() - opening LIST file");

    /* calculate file size */
    if (fstat(fd, &st) == -1)
        handle_error("fstat() - getting LIST file stats");
    file_size = st.st_size;

    /* allocate header buffer */
    header = malloc(sizeof(file_size));
```

```
    if (!header)
        handle_error("malloc() - allocating LIST header");

    /* set the header */
    memcpy(header, &file_size, sizeof(file_size));

    /* send file and free resources */
    send_file(fd, header, file_size, sizeof(file_size));
    free(header);
    if (close(fd) == -1)
        handle_error("close() - closing file list");
}
```

2.5 Comando GET

Nel messaggio di comando get, i primi 8 bit che specificano il comando sono seguiti dal nome del file che si vuole scaricare, questo campo è composto da un numero indefinito di byte, il server lo interpreta come una stringa, pertanto legge fintanto che non trova il terminatore di stringa.

Il server una volta ottenuto il nome del file, controlla se è presente in memoria e, in caso affermativo, prepara il messaggio di risposta così strutturato: i primi 8 bit contengono una costante che indica che il file esiste, i successivi 64 bit la dimensione del file, infine segue l'intero file.

Se il file non esiste, viene inviato un messaggio di soli 8 bit contente la costante che ne indica l'assenza.

2.6 Comando PUT

Nel messaggio di comando put, dopo i primi 8 bit che specificano il comando, segue il nome del file delimitato dal terminatore di stringa, la dimensione del file in un campo di 64 bit ed infine il file stesso.

Il server avvia la procedura di ricezione al termine della quale invia indietro un messaggio di 8 bit con l'esito dell'operazione.

2.7 Trasferimento affidabile

Il trasferimento affidabile è stato realizzato implementando l'algoritmo *selective repeat* descritto nel libro [riferimento].
Tale implementazione è stata "isolata" all'interno di un modulo apposito, creando una sorta di livello di trasporto virtuale, il quale si occupa di suddividere i messaggi in segmenti di giusta misura e di gestirne gli invii, le ritrasmissionie la ricezione.

2.7.1 Interfaccia

L'interfaccia che offre questo stato di trasporto virtuale, come descritto nei primi capitoli, è simile a quella che un programmatore ha a disposizione per un protocollo TCP.
Per richiamare la funzione per l'invio dei dati (*rdt_ send*) basta specificare l'indirizzo e la dimensione del buffer contente i dati da inviare, mentre per la ricezione (*rdt_ recv*) l'indirizzo e la dimensione del buffer che riceverà i dati.
A differenza delle comuni *read* e *write* non va specificato il file descriptor della

socket, perché il servizio di trasporto virtuale si basa su una connessione, per cui la socket è fissata al momento dell'instaurazione della connessione, che avviene quando viene chiamata la funzione *init_ transport*.
Inoltre la funzione *rdt_ read_ string* permette di leggere una stringa di una certa lunghezza massima dal messaggio arrivato, cosa necessaria per poter leggere il nome del file che si vuole scaricare.

transport.h

```
void init_transport(int sockfd, struct proto_params *params);
void rdt_send(const void *buf, size_t len);
void rdt_recv(void *buf, size_t len);
ssize_t rdt_read_string(char *buf, size_t size);
```

2.7.2 Struttura

Lo strato di trasporto virtuale è visto dall'esterno come una black box che prende in ingresso messaggi dal livello applicativo, e restituisce i messaggi di risposta dell'host interlocutore.
Nello specifico un messaggio viene frammentato in segmenti di una misura massima prefissata (MSS), i quali poi vengono inviati e gestititi tramite l'algoritmo di trasferimento affidabile, in questo modo vi è la garanzia che un segmento non venga ulteriormente suddiviso a livello di collegamento.
La dimensione massima del segmento è stata calcolata considerando un MTU relativo ad un collegamento Ethernet standard di 1500 byte, un header UDP/IP di 28 byte ed un header contente i parametri necessari all'esecuzione dell'algoritmo siffatto: il numero di sequenza del segmento pari ad 1 byte e la quantità di byte significativi nel payload pari a 2 byte.

transport.h

```
#define MTU             1500
#define UDPIP_HEADER    28
#define SR_HEADER       (sizeof(uint8_t) + sizeof(uint16_t))

#define MSS             (MTU - UDPIP_HEADER - SR_HEADER)


struct segment {
    uint8_t  seqnum;
    uint16_t size;
    uint8_t  payload[MSS];
};
```

3 Test

4 Esempi di funzionamento

5 Analisi delle prestazioni

6 Installazione e congurazione