Progetto di Laboratorio di Sistemi Operativi

Emilia Del Re - N86001966

Claudio Cortese - N86001886

Simple File Sharing (SFS)

Descrizione

SFS__ e' stato creato nell'ambito del Progetto relativo all'esame di *Laboratorio di Sistemi Operativi* ed implementa un servizio distribuito di file sharing __peer-to-peer.

SFS__ e' un'applicativo in grado di essere eseguito sia in modalita' di server di *rendez-vous* che in modalita' __peer.

Guida d'Uso

Tutte le informazioni riguardo la compilazione di **SFS__ e l'utilizzo di quest'ultimo, compreso esempi illustrativi, sono presenti nel file *Readme.pdf* all'interno della cartella __doc**/.

Protocollo

In questa sezione si vuole descrivere, analizzandolo, il protocollo adottato all'interno di **SFS** per permettere il corretto funzionamento di quest'ultimo.

Tutti gli scambi di informazioni vengono veicolati attraverso il protocollo **TCP**/**IP**.

Comunicazioni con il server di rendez-vous

Notifica di un nuovo *peer*

All'atto della connessione di un *peer* alla rete, quest'ultimo invia le informazioni riguardanti l'**IP__ e la porta, su cui avverranno le comunicazioni con gli altri *peer*, al server di** .

Piu' propriamente, l'indirizzo **IP__ (_*IPv4*) e' restituito dalla struttura `sockaddr_in`, riempita dalla chiamata alla _system call `accept()`, al momento dell'handshake tra il *peer* appena connesso al suddetto server; sara' solo il numero di porta ad essere inviato tramite la connessione *socket* stabilita.

L'accettazione del nuovo *peer* (nonchè la gestione delle comunicazioni con questo) è demandata interamente ad un *thread* specifico per il client.

Tramite tale *thread*, il server rende disponibili le informazioni riguardanti i *peer* gia' presenti in rete al nuovo connesso mediante il medesimo canale di comunicazione; una volta ottenute le informazioni e storate opportunamente per mezzo di una lista concatenata, il server di *rendez vous* le inviera' poi a sua volta a tutti i *peer* attualmente connessi e, aggiornando di volta in volta, ad i *peer* futuri.

Cambio di stato dei *peer*

Quando un *peer* fa richiesta di terminazione (a seguito di esplicita richiesta utente), questa viene inoltrata al server di , che provvede ad aggiornare le proprie informazioni riguardo i nodi della rete e ad aggiornare conseguentemente i *peer* interessati : se il *peer* richiedente ha *download* od *upload* in corso, la rimozione definitiva viene posticipata fino all'esaurimento dei rispettivi trasferimenti.

In particolare, ai *peer* attualmente connessi vengono inviati indirizzo **IP** e numero di porta del *peer* richiedente la terminazione.

I *peer* riceventi si occuperanno, con opportuni controlli, di registrare il cambiamento notificato; piu' in particolare, se questi sono gia' in possesso delle informazioni appena ricevute ed il rispettivo nodo risulta avere stato "*attivo*", questi lo modificheranno in "*in terminazione*".

Terminazione definitiva del *peer*

Quando un *peer* puo' disconnettersi definitivamente, il server di *rendez-vous* provvede ad inviare nuovamente ai *peer* attualmente connessi le coordinate di networking del *peer* ormai rimosso dalla propria lista di nodi di rete.

I restanti *peer* connessi si occupano quindi di rimuovere definitivamente le rispettive informazioni dalla propria lista personale, confrontando nuovamente **IP** e porta ricevuti con quelli registrati e valutando il rispettivo stato del nodo : se questi risulta gia' "*in terminazione*", allora sara' necessaria la rimozione dei rispettivi dati.

Comunicazione dei nuovi *peer* e dei cambi di stato a tutti i *peer* attivi

Richieste di download

All'atto della richiesta di *download* da parte dell'utente, viene avviato un *thread* specifico per la gestione; cio' per' accade se e solo se il *peer* in questione ha stato ancora attivo; viceversa, le immissioni di nomi di file per il *download* vengono totalmente ignorate.

La richiesta viene inoltrata ai *peer* attivi a cui il nodo risulta collegato, eccetto a quelli in terminazione.

La comunicazione avviene su una specifica *socket*/canale di comunicazione creato appositamente, onde evitare overcluttering di file desciptor dovuti a richieste concorrenti.

Invio e Ricezione di file

Invio

All'atto della ricezione della richiesta di *upload* a seguito di una reciproca di *download*, **SFS** controlla preventivamente l'esistenza o meno del file richiesto all'interno della cartella di condivisione specificata all'avvio dell'applicazione tramite flag `-s`.

In caso in cui il suddetto file esista viene inviato un 'ok' al *peer* richiedente, che a sua volta evitera' di inviare richieste agli altri *peer* della rete.

Si prepara quindi un nuovo canale di comunicazione e si rende disponibile al *peer* richiedente, tramite la creazione di una *socket* temporanea sulla prima porta libera indicata dal sistema operativo stesso, da notificare al *peer*, e successivamente si procede all'effettivo *upload* del file.

Ricezione

Nel momento in cui uno dei *peer* informa che il file cercato dall'utente e' disponibile, il *peer* richiedente il *download* si connette tramite *socket* all'*upload peer* e su questo canale di comunicazione avviene l'effettivo *download* del file, in modo da garantire che successive richieste a questo specifico *peer* vengano gestite in modo corretto.

Il *peer* che ha richiesto il *download* apre un nuovo file, omonimo a quello richiesto, nella directory di *download* specificata all'atto dell'avvio del processo client tramite flag `-d`.
A terminazione del *download*, il file descriptor rispettivo alla *socket* temporanea viene chiuso.

Dettagli Implementativi

In questa sezione si vogliono descrivere i dettagli Implementativi giudicati piu' interessanti.

Utilizzo coerente dello snake_case

Durante la stesura del codice di **SFS__ si e' fatto utilizzo dello __snake_case**

Utilizzo di MakeFile

Per aiutarci durante la fase di testing e di deploy dell'applicativo, abbiamo scelto di utilizzare un Makefile, comprensivo di più flag di compilazione e warning :

```
CFLAGS=-W -Wshadow -Wwrite-strings -ansi -Wcast-qual -Wall -Wextra -Wpedantic -peda
```

In particolare, si noti l'utilizzo dello standard **C__ in versione __99**, scelto perlopiù per semplificare l'inizializzazione delle strutture.

Libreria Statica

Si è scelto l'utilizzo di una libreria statica per rendere lo sviluppo più semplice e veloce.

Utilizzo di Optarg

L'utilizzo della funzione standard `getopt()` ci ha permesso di fornire delle comode `short options` per eseguire in modo corretto il programma, con gli opportuni parametri, unificando inoltre la porzione di codice relativa all'inizializzazione in modalità di *server* di *rendez vous* e rispettivamente di *peer*.

Struttura flags

```
struct flags
{
  unsigned share_dir : 1;
  unsigned download_dir : 1;
  unsigned server : 1;
  unsigned client : 1;
  unsigned ip : 1;
  unsigned wrong_ip : 1;
  unsigned port : 1;
  unsigned wrong_port : 1;
  unsigned listen_port : 1;
  unsigned wrong_listen_port : 1;
};
```

Controllo correttezza flag e passaggio parametri ad `init_peer()`

```
  if(flag.client && flag.ip && flag.port && flag.share_dir && flag._download_dir &&
  {
    printf("Partiamo con il client!\n");
    exit_status = init_peer(&sin, _peer_listen_port);
  }
```

Controllo correttezza flag e passaggio parametri ad `init_server()`

```
if(flag.server && flag.port)
{
  printf("Partiamo con il server!\n");
  exit_status = init_server(&sin);
}
```

Invocazione a `get_help()`

Viene mostrato all'utente un help, specifico per ogni caso d'uso.
In particolare le invocazioni alle chiamate a `get_help()` accadono quando l'utente utilizza in

modo errato le flag con cui avviare il programma.

Esempio di chiamata:

```
get_help(&flag);
```

Thread impiegati nell'applicativo e rispettivo ruolo

N.B Tutti i *thread* utilizzati nell'applicazione sono creati in modalità detached, al fine di parallelizzare i compiti da loro svolti.

Server di *rendez vous*

Il master *thread* del server di *rendez vous* ha il compito principale di creare un canale di comunicazione **TCP**/**IP** per accogliere le richieste dei nodi *peer* della rete; ad ognuno di questi, all'atto della connessione, associa un *thread* specifico.

```
void * client_thread(void * arg)
```

È il *thread* associato dal server di rendez-vous ad ogni client che vi si connette.
Si occupa di gestire tutte le comunicazioni con quest'ultimo, a partire dalla lettura della porta su cui il client si metterà in ascolto per gli altri *peer* della rete; si occupa inoltre di aggiungere alla struttura dati propria del server di *redez-vous* le informazioni inerenti ai *peer* a seguito della loro connessione (**IP**, porta, stato attuale).

Ad ogni *peer* introdotto nella rete, vengono comunicate le informazioni in merito agli altri nodi connessi :

```
send_active_peer_list(*client_info -> active_peer_list, client_addr, client_port, cl
```

Inoltre, agli altri *peer* della rete viene reso noto il nuovo nodo aggiunto :

```
send_last_connected_client(*client_info -> active_peer_list, client_addr, client_por
```

Terminata tale fase di inizializzazione, al *thread* non rimane altro se non attendere la richiesta di modifica di stato da parte del client che questi si occupa di gestire;
abbiamo scelto a tal fine di porre una lettura bloccante :

```
read(client_fd, stop, sizeof stop);
```

Quando il client avrà notificato al *thread* rispettivo per la prima volta, ciò sarà esclusivamente per una transizione dallo stato "attivo" allo stato di "in terminazione";
il *thread* si occuperà quindi di modificare opportunamente la lista condivisa con il master *thread* :

```
change_peer_status(*client_info -> active_peer_list, client_addr, client_port, TERM
```

E di notificarlo a tutti gli altri *peer* della rete :

```
send_changed_status_peer_(*client_info -> active_peer_list, client_addr, client_por
```

Il *thread* si pone poi in attesa di una successiva notifica da parte di quel client, nuovamente tramite lettura bloccante :

```
read(client_fd, stop, sizeof stop);
```

Quando sarà uscito da tale lettura, il *thread* saprà che ciò è dovuto ad una richiesta definitiva di terminazione;
procederà quindi con l'eliminazione del rispettivo nodo dalla propria lista concatenata :

```
*client_info -> active__peer__list = discard_peer_node(*client_info -> active_peer_
```

E conseguentemente notificherà i *peer* della rete :

```
send_peer_to_discard(*client_info -> active_peer_list, client_addr, client_port);
```

Processo *peer*

Il ruolo principale del master *thread* del processo *peer* è quello di instaurare la connessione al server di rendez-vous e di creare opportunamente gli altri *thread*, nonchè di fornire il prompt all'utente, corredato di gestione dei comandi.

In particolare, subito dopo la connessione al server, il *peer* crea uno specifico *thread* per la parte server della comunicazione *peer-to-peer*.

Accept_thread

```
void * accept_thread(void * arg)
```

Il suddetto *thread* si occupa di creare un canale di comunicazione su cui si metterà in ascolto degli altri *peer* attivi.
Dopo ogni connessione stabilita con un *peer*, tale *thread* si occupa di creare un ulteriore *thread*, apposito per la gestione delle richieste di *upload* provenienti dal suddetto *peer*.

Handle_upload_thread

```
void * handle_upload_thread(void * arg)
```

Questo *thread* si occupa, in un while(1), di attendere richieste di *upload* da parte degli altri *peer* attivi nella rete.

Più in particolare, questi attende l'invio del nome di file di cui si desidera effettuare il *download* e rimane in lettura bloccante della lunghezza di suddetto nome e del nome stesso (ciò consentirà alla successiva lettura di sapere esattamente quanti byte attendersi) :

```
read(peer_fd, &filename_len, sizeof filename_len);
read(peer_fd, filename, filename_len);
```

Quando sarà uscito dalle due letture, il *thread* saprà di dover gestire la ricerca del file ed un suo possibile *upload*; di conseguenza, si predispone alla creazione di un nuovo canale di comunicazione, che si occuperà esclusivamente del trasferimento del file.

A tal fine, si occupa della parte server di tale canale, legando il file descriptor alla prima porta disponibile fornita dal sistema operativo, ed attendendosi connessioni su qualunque interfaccia di rete della macchina corrente; comunicherà poi la porta scelta al *peer*, tramite il file descriptor associato precedentemente dall' `accept_thread` :

```
if(!getsockname(sockfd, (struct sockaddr *)&sin, &len))
{
  write(peer_fd, &sin.sin_port, sizeof sin.sin_port);
}

...

else
{
  int flag = -1;

  write(peer_fd, &flag, sizeof flag);
}
```

In tal modo ci assicuriamo, in ogni caso, di scrivere qualcosa al *peer* che attende la lettura, e che riceverà quindi un valore di discrimine per sapere se connettersi o meno.

Se l'assegnazione di una porta al *socket* appena creato ha successo, si procede con l'instaurare una connessione con il *peer* richiedente.
Un *thread* opportuno gestirà SOLO QUESTA richiesta.

upload_thread

Al suddetto *thread* si demanda il compito di cercare il file richiesto all'interno della propria directory di share; inoltre, se questo viene trovato (tramite ricerca ricorsiva, nella directory specificata e in eventuali sue sottodirectory) il *thread* comunica al *download thread* del *peer* rispettivo la dimensione del file da attendere, 0 altrimenti.

Si procederà poi con il trasferimento del file se questi è presente e, ad ogni *upload* che questo *thread* effettua, si incrementerà la variabile che conteggia il numero di *upload* in corso.

```
try_upload_file_to_one_peer(u_info.peer_fd, u_info.filename, &count_upload);
```

Al termine di ciò, il controllo che segue si occuperà di controllare se è stata nel frattempo notificata una richiesta di stop per il *peer* corrente :

```
if(got_stop == 1)
```

```
{
  raise(SIGUSR1);
}
```

Il signal handler seguente si occupa della rispettiva gestione del segnale :

```
void sig_handler(int signal)
{
  if(signal == SIGUSR1)
  {
    got_stop = 2;

    pthread_mutex_lock(&download_list_mutex);
    active_download_list = free_download_list(active_download_list);
    pthread_mutex_unlock(&download_list_mutex);

    pthread_mutex_lock(&_peer_list_mutex);
    active_peer_list = free_peer_list(active_peer_list);
    pthread_mutex_unlock(&peer_list_mutex);

    raise(SIGINT);
  }
}
```

Receive_peer_thread

È il *thread* che si occupa della fase di inizializzazione lato *peer* :

```
write(info.serv_fd, &info.peer_listen_port, sizeof info.peer_listen_port);
```

Il *peer* invia la porta su cui attende connessioni dagli altri nodi della rete; dopodichè, si aspetta di ricevere la lista di *peer* attivi da parte del server di *rendez vous* :

```
receive_active_peer_list(info.active_peer_list, info.serv_fd);
```

E rimane poi in attesa di ricevere le informazioni riguardanti ogni *peer* che si connetterà successivamente :

```
while(1)
{
  receive__peer__info(info.active__peer__list, info.serv_fd);
}
```

download_thread

Il *thread* di cui sopra si occupa strettamente della propagazione della richiesta di *download* ai *peer* attivi, fin quando no se ne trova uno disponibile all'*upload* delle informazioni richieste.

Per ogni *peer* a cui giunge la richiesta di *download* (espressa tramite invio del filename) il *thread* attende la ricezione di un intero (off_t per la precisione) che farà da discrimine; se maggiore di zero, indica la dimensione del file trovato.
Il *thread* si predispone quindi alla ricezione del file da quel *peer*, evitando quindi le richieste ai nodi successivi.

```
_download__peer__info = try__download_file_from_any__peer_(d_info.active_peer__list,
```

La funzione di cui sopra si occupa inoltre di riempire la lista concatenata dei *download* attualmente in corso e di modificare i nodi che subiscono variazioni (avanzamento del *download*).

La rimozione del nodo di *download* è appannaggio del *thread* all'uscita dalla funzione, che poi controlla se c'è stata l'immissione di uno stop da parte dell'utente :

```
if(got_stop == 1)
{
  pthread_mutex_lock(&count_upload_mutex);

  if((_download__count = count_download(*d_info.active_download_list)) == 0 && count
  {
    raise(SIGUSR1);
  }

  pthread_mutex_unlock(&count_upload_mutex);
}
```

Si controlla infatti se c'è stato uno stop, e si vede se non ci sono n'è *download* nè *upload* in corso; in tal caso, la richiesta di terminazione è accordata immediatamente e il segnale è gestito a tal fine dall'handler di cui sopra.