Relazione Progetto 1 - Sistemi Operativi 2013/14

Bortoli Gianluca - matricola n. 159993
Dellera Andrea - matricola n. 158365

May 24, 2014

1 Prefazione

Il progetto che abbiamo scelto consiste principalmente in un client ed un server che comunicano tra loro, scambiandosi dei messaggi attraverso delle FIFO. Server e client sono due (o più) processi ben separati e senza alcun antenato in comune; abbiamo perciò scelto di utilizzare delle *FIFO* (dette anche *named pipe*), invece che delle semplici pipe, per non essere vincolati dal legame di "parentela" tra i processi necessari per poter usare le seconde.
Presentiamo ora, separatamente, le funzionalità che abbiamo attribuito alle due entità.

2 Client

Il processo *client* si occupa, come in una generica architettura client-server, di interrogare un altro processo detto server. Il client si limita quindi ad inoltrare richieste al server, il quale risponderà dopo aver eseguito le operazioni richieste. Abbiamo deciso di delegare a questo processo l'incombenza di prendere in input da riga di comando, usando l'apposita funzione getopt, i seguenti parametri:

1. nome del server a cui collegarsi (-n)
2. chiave da utilizzare per la de/codifica (-k)
3. un flag nel caso in cui si voglia prendere l'input da file, il quale esclude il passaggio del messaggio direttamente dalla linea di comando (-f)
4. due flag per indicare se voglio de/codificare (-d e -e rispettivamente)
5. nome del file dove, eventualmente, scrivere l'output (-o)
6. un flag per poter visualizzare i messaggi precedentemente codificati da un certo servet (-s)
7. un indice numerico per richiedere la decodifica di un precedente messaggio (-i)

Una volta eseguito il parsing dei parametri da linea di comando, generiamo un nome sempre diverso per ogni client che viene lasciato, in modo da non aver conflitti con i nomi delle FIFO su cui poi i processi vanno a leggere/scrivere.
Per risolvere ciò, abbiamo deciso di concatenare al termine glient il *pid* corrispondente (es: "client12345").
Infine, la chiamata alla funzione *run_ client* con i parametri corretti. Parleremo più approfonditamente delle funzioni particolari implementate, quando tratteremo la libreria *function.h* che abbiamo appositamente creato.

3 Server

Il processo *server* gestisce le richieste inviate dal client ed offre essenzialmente il servizio di de/codifica di messaggi scelti dal client.
Esso gestisce in modo autonomo tutti gli errori durante il parsing dei parametri da linea di comando, durante la creazione/apertura delle FIFO e per l'eventuale lettura del messaggio da un file esterno.

La chiamata alla funzione *run_server* è stata messa all'interno di un ciclo infinito per far sì che il server rimanga in ascolto finchè non venga terminato volontariamente dall'utente, oppure avvenga un errore per il quale non può continuare.
Alla fine, viene eliminata la FIFO relativa al server con quel nome specifico per permettere l'eventuale creazione di un'altro server con un nome usato in passato.

4 Libreria "functions.h"

Abbiamo deciso di strutturare il codice creando una libreria *functions.h* che contiene tutte le funzioni principali. Ciò ci permettere di non appesantire inutilmente i sorgenti del *client* e del *server* e li rende anche più leggibili.
Le funzioni principali sono la *cript* e la *decript*, la *run_server* e la *run_client* (di cui abbiamo accennato prima) ed infine le *write/read_encoded_message*.

4.1 cript e decript

Queste due funzioni si occupano, rispettivamente, di criptare e decriptare i messaggi inviati dal client al server secondo le specifiche del progetto.

4.2 run_server

Questo metodo si occupa di gestire l'intera parte del server. Apre le fifo necessarie nelle modalità corrette, alloca i parametri utilizzati.
La *run_server* gestisce a tutti gli effetti l'intera comunicazione dal lato del server, facendo, quando serve, gli opportuni controlli prima di aprire le FIFO e/o file di configurazione.

4.3 run_client

Questo metodo si occupa, al contrario, di gestire tutta la parte delegata al client, a partire dall'apertura delle FIFO, fino alla scrittura nella fifo per inviare ad un determinato client ciò che ha richiesto, leggendo i parametri necessari dal file di configurazione creato all'avvio di ogni server.

4.4 write/read_encoded_message

Queste due funzioni sono state scritte per implementare la possibilità di per poter avere una cronologia dei messaggi che un dato server ha criptato, in seguito ad una richiesta di un client.
Per farlo, abbiamo deciso che ogni volta che ad un server arriva una richiesta di codifica (ma non di decodifica), esso si crea un file univoco nel quale memorizza i messaggi in ordine di processamento.
Ciò ci permetterà in un secondo momento di poterli andare a referenziare attaverso un indice e, in questo modo, sarà possibile per il client richiederne la decodifica (data una chiave) senza immettere nuovamente il messaggio.

5 Il log dei server

Abbiamo deciso di tener traccia di tutti i server con i relativi parametri nel file *lista_server.txt*.
Ciò ci permette una migliore e più semplice gestione dei parametri in input per le varie funzioni che li richiedono, ma, al contempo, di dare all'utente una panoramica migliore dei server attualmente in esecuzione.
Nel caso in cui un utente, con molti server in esecuzione in background o come demoni ad esempio, voglia accedere ai loro servizi, ma non si ricorda semplicemente i nomi con cui accedervi, può semplicemente leggere il file di log.
Questa scelta è stata dettata sia dalla comodità di avere un log dei server avviati, ma anche per permettere all'utente una visione d'insieme migliore.

6 Makefile

Oltre ai target richiesti, abbiamo deciso di aggiungerne alcuni per rendere il testing più approfondito e separare la creazione degli eseguibili dalla vera e propria installazione dei programmi *codecclient* e *codecserver*.

1. **install**: sposta gli eseguibili compilati precedentemente dal target *bin* in /usr/bin/ in modo che siano utilizzabili come veri e propri comandi della shell

2. **uninstall**: rimuove *codecclient* e *codecserver* da /usr/bin/

3. **testAssets**: testa gli assets creati appositamente, prendendo i messaggi in input da file (il target test, scrive il messaggio da direttamente linea di comando)