Relazione di Progetto

- Sviluppo di un sistema per la gestione remota -

Corso di: Programmazione di Sistema

Prof. Marco Cianfriglia

Studente: Josef Emanuele Zerpa Ruiz

Introduzione:

Per consolidare le conoscenze acquisite durante il corso di Programmazione di Sistema, si vuole sviluppare un sistema Client-Server, per la gestione remota di un calcolatore.

Per applicare gli studi sulla programmazione di sistema Unix e sistema Windows, i programmi sviluppati saranno eseguibili sia su un tipo sistema, che sull'altro, e anche in soluzioni miste; es. server Linux e client Windows. I programmi saranno quindi 'Cross-Platform'.

In breve, il Client e il Server comunicheranno attraverso una connessione TCP, implementeranno un semplice protocollo di autenticazione, dopo un'autenticazione con successo il Server riceverà comandi dal Client, questi verranno svolti, restituendo al Client l'output dell'esecuzione, o in caso di errore, un codice identificativo. Inoltre il Server svolgerà del logging in locale, registrando ogni richiesta dopo un'autenticazione con successo. Il Server potrà essere configurato attraverso un file di configurazione, opportunamente impostato all'avvio, il quale potrà essere anche usato per aggiornare le configurazioni durante l'esecuzione.

Scelte progettuali:

Mano a mano che si sviluppava il progetto ci si è resi conto che le dimensioni del codice sarebbero state rilevanti. Per mantenere una buona lettura del codice, "readability", e quindi facilitarne in mantenimento, ma anche lo sviluppo in sé, il progetto è stato modularizzato in vari moduli e file di supporto/ utilità.

Inoltre la funzione dei moduli e dei file di utilità è anche quella di mascherare le implementazioni specifiche ad un determinato sistema.

Core: *server.c client.c*

Al centro del sistema naturalmente ci sono il programma Server, *server.c*, ed il programma Client, *client.c*. Questi organizzano il flusso di esecuzione, assumendo l'input dall'utente, fornendo messaggi informativi a video, e eseguendo le funzioni dei moduli: autenticazione, lancio dei thread, esecuzione dei comandi, logging.

Modules: *authentication.h commands.h logging.h thread.h*

Ogni modulo svolge un compito a sé stante, e ricopre sia la parte eseguita dal Server, che la parte eseguita dal Client. Nell'autenticazione, ad esempio, c'è una sequenza di azioni svolte dal server, e una sequenza di risposte fornite dal client, e viceversa. Questa sequenza, o protocollo, è gestita interamente dentro ilo file, o modulo, *authentication.h*, un 'header' che ingloba il compito dell'autenticazione. Allo stesso modo l'esecuzione del comando ha come protocollo l'invio del comando richiesto, la valutazione (parsing) del comando, un'esecuzione diretta in caso di comandi da eseguire sul server o l'apertura di file e attesa di dati in una richiesta di trasferimento dati, l'invio di una risposta all'esecuzione del comando, o l'invio di un messaggio di errore. Anche in questo caso, questo flusso di azioni è stato impachettato nel modulo *commands.h*.

Utilities: *constants.h networking.h security.h synchronization.h timing.h*

I file di utilità invece forniscono funzione di, appunto, utilità. Anche qui, uno dei bisogni principali per la formazione e la pacchettizzazione di questi file era creare un'interfaccia che mascherasse le attuali implementazioni specifiche ad una tipologia di sistema piuttosto che un'altra. Ad esempio il file *networking.h* implementa la creazione di un socket, la quale ha bisogno di un setup in più se ci troviamo in un ambiente Windows (l'avvio del Windows Socket API).

Inoltre nel file *networking.h* vengo implementate funzioni ad hoc per l'invio di interi a 64 bit, la trasformazione da little endian a big endian (network byte order), di interi a 64 bit.

Allo stesso modo il file *security.h* fornisce funzioni ad hoc per la formazione di numeri pseudo-casuali di 64 bit, e l'hashing a 64 bit di una stringa.

Structures: *connection.h queue.h*

Infine, per facilitare certe operazioni, sono state usate due strutture ad hoc: una coda, ed una struttura che astraesse i dati di una connessione.

Connection.h descrive una struttura che impacchetta il valore numerico descrittore della connessione, usato nelle funzioni send() e recv(), e le informazioni sulla connessione, quali indirizzo ip e porta del client, utili alle funzioni di logging.

Queue.h invece implementa una coda di puntatori, utile per fornire una struttura dati generica, che immagazzini un determinato numero di valori. Questa struttura viene infatti usata dal Server per salvare gli identificatori dei thread lanciati (in *threads.h*), e quindi poi terminarli in caso di aggiornamento delle configurazioni, oppure salvare le connessioni ricevute, per poi farle gestire dai thread.

Sincronizzazione:

Sono state implementati meccanismi di sincronizzazione per la condivisione delle connessioni, e quindi l'esecuzione dei vari thread, e l'accesso al file di logging.

Come annunciato sopra, le nuove connessioni ricevute vengono inserite dentro una coda. Questa è condivisa globalmente a tutti i thread, e l'accesso è controllato da un semaforo. È stato scelto un semaforo per replicare la soluzione al paradigma Producer-Consumer. Il semaforo dà la possibilità di "accumulare" le "notifiche di sblocco" in caso risorse vengano "prodotte" più velocemente di quanto queste vengano "consumate". Questo è il caso in cui tutti i thread sono occupati a svolgere qualche richiesta mentre arrivano nuove connessioni. Con l'implementazione scelta le richieste verrano accodate, e risolte appena un thread si libera dalla precedente esecuzione.

Per il file di logging invece è stato scelto un semplice lock, mutex. Dato che l'accesso al file è esclusivo, la risorsa è solo una, un semplice lock assicura l'accesso esclusivo. Per l'ambiente Unix è stato usato un pthread mutex, per l'ambiente Windows una Critical Section. Entrambe le strutture assicurano l'esecuzione esclusiva tra thread.

Dettagli Implementativi:

Nel dettaglio, sono stati scelti i seguenti dettagli implementativi.

- **struct Queue**

Per risolvere il problema dello stocking temporaneo di informazioni, quali le connessioni o gli id dei thread, è stata implementata una struttura a coda. La struttura usa un array a dimensione limitata per salvare le informazioni. L'array viene allocato dinamicamente al momento dell'inizializzazione di una nuova coda, attraverso l'apposita funzione *createQueue()*. *destroyQueue()* si occupa di liberare ogni memoria dinamicamente allocata all'inizializzazione.

Funzioni di convenienza, come *enqueue(), dequeue(),* ecc, si occupano di popolare e gestire la coda.

Essendo costituita da un'array di puntatoti a void, la coda può essere usata per accumulare diversi tipi di dati o strutture.

La dimensione della coda viene definita all'inizializzazione.

- **struct Connection**

Connection è un'altra struttura di utilità che semplifica lo svolgimento di operazioni richieste dalle specifiche del progetto. Nel dettaglio, viene richiesto il logging della richiesta, con l'indirizzo ip del client. In questo caso il thread che prende in carico le richieste oltre a conoscere il descrittore della connessione deve conoscere anche l'indirizzo della connessione, e quindi la struttura *addr_in*. Per risolvere questa necessità si sono impacchettate i due dati dentro la struttura Connection.

- **long_rand()**

Un altro problema riscontrato è stato quello di produrre numeri pseudo-casuali a 64 bit. La funzione rand() produce interi, a 32 bit. Per compiere il nostro scopo, l'implementazione sviluppata sfrutta la funzione rand() ripetutamente, costruendo un intero di 64 bit i primi 8 bit ottenuti da rand(). In questo modo dopo 4 esecuzioni avremo un numero pseudo-randomico da 64 bit.

- **long_send() / long_recv()**

Allo stesso modo, l'invio di valori a 64 bit attraverso una socket non è un'operazione standard. Per risolvere questo problema sono state sviluppate funzioni ad hoc che si occupano della trasformazione dell'ordinamento dei dati, da host-order a network-order (little-endian/big-endian), e il conseguente invio / ricezione e riallocamento dei dati.

- **Commands.h**

Commands.h è il file che si occupa di tutto il parsing, invio e ricezione del comando, dei codici di stato, esecuzione del comando, ed invio e ricezione del risultato.

I comandi vengono specificati da linea di comando. Quindi la tipologia di comando è specificata da "-l" per "LIST", "-s" per "SIZE", "-e" per "EXEC", ecc. La prima cosa da fare è convertire i parametri di linea di comando a comando da inviare al server.

Per fare ciò l'opzione "-X", dove X è la lettera che specifica il tipo di comando, viene convertita nel tipo esteso, es: "-l" in "LIST".

Il comando inviato attraverso la rete quindi prevede come prima parola il tipo di comando, il tag, e successivamente eventuali parametri da usare con il comando. Es: "UPLOAD file.txt destination.txt".

Una volta ricevuto il comando, il Server lo esegue in modi diversi in base al tipo di comando. Se è un comando di trasmissione di un file invia la risposta 200 per far sapere al Client cha ha ricevuto la richiesta ed è pronto a ricever/inviare il file. Altrimenti traduce il comando in un comando eseguibile da linea di comando, es "LIST" in "ls" e lo esegue attraverso la funzione popen(), _popen() per gli ambienti Windows.

Esecuzione:

Per testare il progetto, e verificarne il funzionamento, sono stati aggiunti opportuni comandi nel Makefile.

I comandi lanciati con il make hanno impostazioni impostate a indirizzo 127.0.0.1 e porta 8888.

Per avviare il server lanciare: *make runserver*

Per avviare il client lanciare: *make runclient*

Per avviare il client con un comando differente specificare il comando e i parametri come variabile ARGS del comando make: *make runclient ARGS="-e echo hello world!"*

Per altre modalità di avvio visionare il Makefile.

Conclusioni:

Lo sviluppo di questo progetto è stata un'attività complessa e molto stimolante. I tempi di produzione si sono svelati molto più lunghi rispetto quanto immaginato. Eppure, sicuramente, ho potuto notare una crescita a livello di conoscenze, nell'uso delle funzioni di sistema, nella

manipolazione di variabili a 64 bit, nell'utilizzo delle socket, nella modularizzazione e pulizia del codice, nella scrittura del Makefile, nella scrittura di codice Cross-Platform.

Sono rimasto molto soddisfatto di aver intrapreso questa attività, felice dello sforzo impiegato, l'esperienza e la conoscenza acquisita.