UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

CORSO DI LAUREA TRIENNALE IN INFORMATICA

RELAZIONE CONCLUSIVA
DI
LABORATORIO DI SISTEMI OPERATIVI

Sistema di gestione di una videoteca multi-utente

Tutor:
Prof. Alessandra Rossi

Candidati:
Giandomenico Iameo N86002856
Mirko Esposito N86004055

ANNO ACCADEMICO 2024-2025

Indice

1 Panoramica del progetto

1.1 Descrizione sintetica del progetto

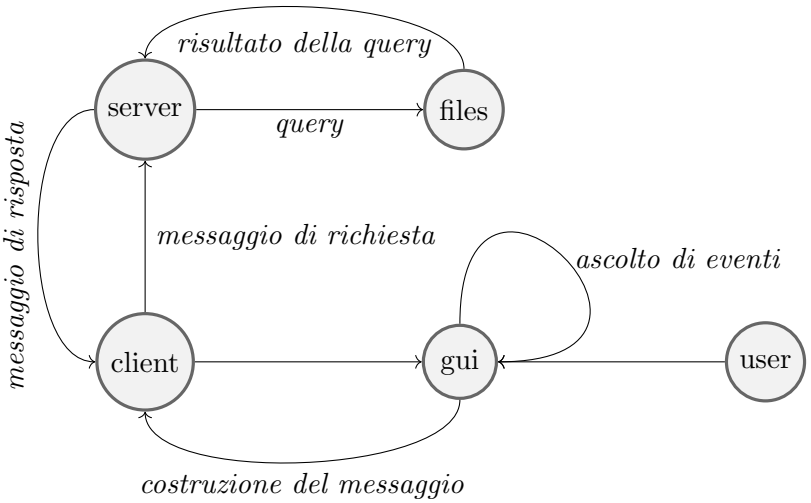
Il progetto consiste nello sviluppo di un'applicazione di rete basata sull'architettura client-server che modella una videoteca per un numero non precisato di utenti. Gli utenti devono potersi registrare e loggare al server. Il negoziante ha una lista di film disponibili nella videoteca, con il numero di copie disponibili, numero di copie prese in prestito, e per ognuna delle copie prese in prestito un nome o un numero identificativo del client che le ha prese in prestito, un timestamp del giorno in cui è stato preso in prestito, e una data per la restituzione del libro. I client possono connettersi al server per cercare, noleggiare o restituire film. Gli utenti possono prendere un numero k di film in prestito, dove k è deciso dal venditore/negoziante. Gli utenti possono mettere nel carrello tutti i film presi in prestito e poi possono effettuare il check-out.

1.2 Modalità d'uso dei programmi client e server

Per la modalità d'uso dei programmi si consiglia la lettura del file README.md, dove sono stati riportati i comandi, le dipendenze necessarie, nonché le tecnologie adottate per l'applicativo. Per la consultazione dei sorgenti si rimanda alla repository Github: Repository

2 Funzionamento dell'applicativo

2.1 Descrizione dell'applicazione



Il funzionamento dell'applicazione è dettato proprio dal grafo rappresentato nella figura in alto. Il client e il server comunicano scambiandosi *messaggi di richiesta* e *messaggi di risposta*.

Il client, oltre a comunicare con il server, interagisce con una interfaccia grafica che si presenta all'utente all'avvio dell'applicazione. Ogni operazione che l'utente effettua all'interno della finestra viene tradotta in un messaggio di richiesta per poter essere inviato e compreso dal server. Ogni messaggio di risposta che il client riceve dal server viene tradotto in una forma comprensibile all'utente e mostrato nell'interfaccia grafica. In sintesi, l'interfaccia grafica funge da *ascoltatore di eventi*. Essa è in grado di catturare gli eventi generati dall'utente e rimanere in attesa di una risposta dal server per notificare poi l'utente dell'avvenuta ricezione.

D'altra parte, il server interagisce con una base di dati *file-based*. Ogni informazione riguardante gli utenti, come i film noleggiati, i film restituiti, i nomi dei film, ecc. è memorizzata nella base di dati per poter essere recuperata, modificata o letta quando si presenta l'esigenza. Dal momento che la base di dati è stata organizzata in archivi di file, sono stati utilizzati i linguaggi *Sed*, *Awk* e *Bash* per la manipolazione dei dati.

2.2 Protocollo di rete MTP

Il protocollo di rete utilizzato per questa applicazione è implementato in due programmi, client e server, in esecuzione su punti terminali che comunicano tra loro scambiandosi messaggi. Il protocollo definisce sia la struttura dei messaggi sia la modalità con cui client e server si scambiano i messaggi, che chiameremo protocollo di trasferimento dati focalizzato sul traferimento dei film (oppure, per semplicità, MTP: *Movie Transfer Protocol*).
Il trasferimento dei messaggi da un client MTP a un server MTP (e viceversa) presenta molti aspetti in comune con i protocolli attualmente in vigore, come l'utilizzo del protocollo TCP a livello di trasporto. In primo luogo, il client MTP in esecuzione costituisce una connessione TCP sulla porta 8080 con il server MTP, anch'esso in esecuzione. Una volta stabilita la connessione, MTP procede in tre fasi: *autenticazione*, *sessione* e *rilascio*. Durante la prima fase (*autenticazione*) il client invia nome utente e password (in chiaro) per autenticare l'utente. Durante la seconda fase (*sessione*) il client effettua le operazioni di noleggio, restituzione e ricerca di un film; inoltre, può anche aggiungere o rimuovere film dal carrello della spesa virtuale.
La fase di *rilascio* ha luogo dopo che il client effettua una richiesta di disconnessione dall'account, il che implica l'invio di una richiesta di cancellazione

dell'account oppure di uscita da quest'ultimo.
Durante una qualsiasi delle fasi, il client invia *messaggi di richiesta MTP* mentre il server risponde con *messaggi di risposta MTP*. La forma e la struttura di tali messaggi sono alla base della compresione delle volontà dell'uno e dell'altro.

2.2.1 Formato dei messaggi di richiesta MTP

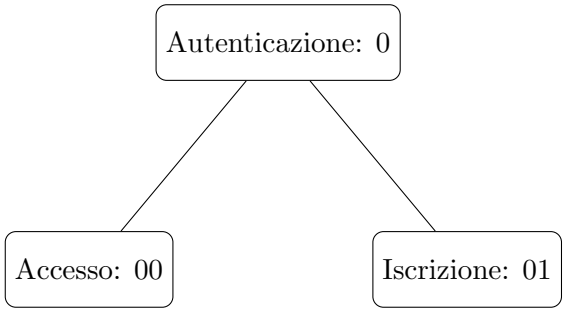
Tipo	Azione	Corpo

Tabella 1: Formato generale dei messaggi di richiesta MTP

In generale, i messaggi di richiesta presentano tre sezioni: la sezione *tipo di richiesta*, *intenzione della richiesta* (*Azione*) e il *corpo dell'entità*. Le prime due sezioni costituiscono l'*intestazione della richiesta* mentre il corpo rappresenta il fulcro del messaggio: vengono specificate le credenziali di accesso all'account, il nome del film, il numero di film da noleggiare, restituire, aggiungere o rimuovere dal carrello e la data di restituzione.
L'intestazione della richiesta è concretamente un numero di due cifre. La prima cifra identifica il *tipo di richiesta* e dà un indicazione di alto livello, la seconda identifica l'*intenzione della richiesta* e fornisce più dettagli.

Ecco un elenco dei principali codici per ogni classe.

- 0 (*Autenticazione*): Il client mediante questa richiesta desidera autenticarsi verso il server.
  - 00 (*Accesso*): Richiesta di accesso all'account.
  - 01 (*Iscrizione*): Richiesta di creazione di un account.



- 1 (*Sessione*): Tale tipologia di richiesta specifica le operazioni da parte del client che può effettuare durante una sessione sull'account.

  - 10 (*Noleggio*): Richiesta di noleggio di un film.
  - 11 (*Aggiornamento*): Richiesta di aggiornamento di un articolo nel carrello.
  - 12 (*Restituzione*): Richiesta di restituzione di un film.
- 2 (*Rilascio*): Si tratta di una richiesta di disconnessione dal'account.
  - 20 (*Uscita*): Richiesta di uscita dall'account.
  - 21 (*Cancellazione*): Richiesta di cancellazione dell'account.

Il corpo del messaggio varia in base all'intestazione specificata. Per le richieste di tipo *Autenticazione* nel corpo vanno specificati l'*username* e la *password*, indipendentemente dal tipo di intenzione specificata. Nel caso delle richieste di tipo *Sessione* il corpo, viceversa, varia a seconda dell'intenzione specificata. Ad esempio, nelle richieste di *Noleggio*, il corpo del messaggio è vuoto, mentre per le richieste di *Aggiornamento* nel corpo vanno specificati il nome del film, la quantità di film e la data di restituzione.

2.2.2 Formato dei messaggi di risposta MTP

Un messaggio di risposta presenta invece due sezioni: *messaggio di notifica* e *codice di stato*.

Notifica	Codice di stato

Tabella 2: Formato generale dei messaggi di risposta MTP

Questo protocollo di dettatura di messaggi utilizza *notifiche* (*acknowledgment*) *positive* o *negative*.

In uno scenario del genere, il server che raccoglie il messaggio di richiesta risponde con un messaggio di *notifica positiva* se ha compreso e memorizzato il messaggio, mentre risponde con un messaggio di *notifica negativa* chiedendo di ripetere il messaggio che non ha compreso. Le notifiche possono assumere una delle seguenti forme: OK (per le notifiche positive) e Bad Request (per le notifiche negative).

Ad esempio, se il client specifica come intestazione il codice 34 riceverà un messaggio di tipo Bad Request se il server non ha implementato nessuna procedura in grado di gestire quel tipo di richiesta.

Il *codice di stato*, invece, indica il risultato di una richiesta. Ad esempio, per una richiesta di *Aggiornamento* il client potrebbe ricevere uno dei quattro

risultati, espressi in codici numerici: *Articolo aggiornato*, *Data non valida*, *Quantità non noleggiabile* oppure *Film non trovato*.

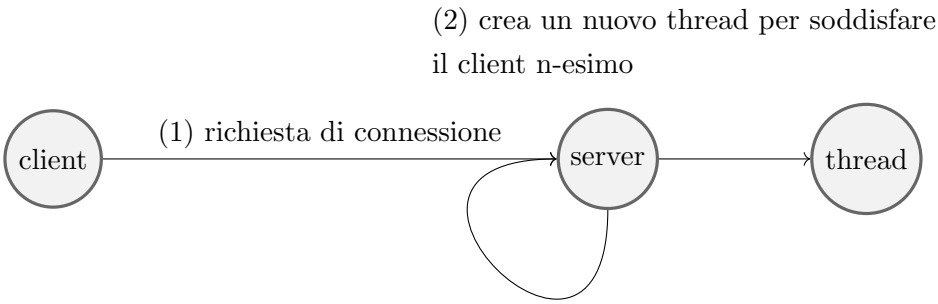
3 Dettagli implementativi

Nei seguenti paragrafi verranno descritti i programmi client e server che comunicano "scrivendo in" e "leggendo da" delle socket TCP messe a disposizione dallo strato di trasporto, con particolar riguardo alla comunicazione multi-client.

3.1 Il programma Server

3.1.1 Gestione della comunicazione multi-client

Il programma server può essere descritto mediante il seguente automa.



La soluzione adottata per gestire la moltitudine di client che accedono in modo concorrente al server si basa sul modello *multithreading*. Tale modello è generalmente più conveniente rispetto a eseguire il server come un singolo processo che crea processi figli separati per gestire i vari client.

Secondo tale schema, quindi, il server crea un thread distinto per soddisfare le richieste dei client; in presenza di una richiesta, anzichè creare un altro processo, viene creato un altro thread per servirla.

Il seguente brano di codice descrive l'automa in figura:

```
while( 1 ) {

        address = sizeof( client );
        wait( &lock );

        if( ( sda = accept( listener,
                            ( struct sockaddr * )&client,
                            ( socklen_t *)&address ) ) < 0 ) {
                perror( "Errore ricevuto dalla primitiva accept" );
                exit( EXIT_FAILURE );
        }
        if ( pthread_attr_setdetachstate( &attributes,
                                          PTHREAD_CREATE_DETACHED ) ) {
                    perror( "pthread_attr_setdetachstate()" );
                    exit( EXIT_FAILURE );
        } pthread_create( &tid, &attributes, runner, &sda );
}
```

Listing 1: Segmento di codice della funzione main

La primitiva `accept()` viene eseguita dal thread principale (*listener*) che blocca temporaneamente l'esecuzione del programma finché un client non bussa alla porta di benvenuto. Nel riprendere l'esecuzione viene creata una nuova socket nel server, chiamata `sda`, e dedicata allo specifico client che ha busssato. Arrivati a questo punto, client e server portano a compimento la procedura di *handshaking*, creando una connessione TCP tra processo client e thread principale. Successivamente, viene creato un nuovo thread tramite `pthread_create()` che avrà la responsabilità di onorare ogni richiesta da parte del client, assegnatogli dal thread principale. Infine, il *listener* potrà tornare in ascolto di ulteriori richieste dei client.

Come è possibile notare, prima della creazione del figlio, il thread principale chiama `pthread_attr_setdetachstate()`. Il suo scopo è quello di configurare gli attributi del thread figlio in modo che venga creato in uno specifico stato, ovvero in stato *distaccato*. Ciò significa che il thread principale non sarà più responsabile della gestione del thread creato. In questo modo, il `listener` può continuare a ricevere nuove richieste senza dover aspettare che un client completi le operazioni.

Tuttavia, il thread principale può eseguire la primitiva `accept()`, cioè mettersi in ascolto di richieste di connessioni, solo dopo che il thread figlio ha eseguito la funzione `post()`, ovvero solo dopo che ha duplicato il descrittore di file `sda`. Questo passaggio risulta necessario perché la variabile `sda` verrà sovrascritta, ad ogni passo del ciclo, con un nuovo descrittore di file.

Il seguente brano di codice mostra le prime istruzioni della funzione `runner()`, da cui parte l'esecuzione di ogni thread creato, mettendo in evidenza il processo di duplicazione del descrittore di file e lo sblocco del lucchetto.

```
void *runner( void *sda ) {

    int sdb = dup( *( int * )sda );
    post( &lock );

    ...
}
```

Listing 2: Segmento di codice della funzione runner

Ora, thread e client possono finalmente scambiarsi *messaggi di richiesta* e *messaggi di risposta* senza alcun tipo di vincolo e in maniera indipendente.

3.1.2 Controllo della concorrenza

Anche se i client vengono gestiti in maniera indipendente, i thread associati, molto spesso, si trovano in situazioni in cui accedono contemporaneamente a spazi condivisi (o *regioni critiche*). Più nello specifico, questo accade quando i thread si trovano ad accedere agli stessi file della base di dati. Alcuni thread posso richiedere la *sola lettura* del contenuto della base di dati mentre altri sono interessati *non alla sola lettura*.

I thread, quindi, sono stati distinti in *thread lettori* interessati alla sola lettura e *thread scrittori* gli altri. Naturalmente, se due lettori accedono nello stesso momento alla base di dati non si ha alcun effetto negativo; viceversa, se uno scrittore e un altro thread (lettore o scrittore) accedono contemporaneamente alla stessa base di dati: ne può derivare il caos.

Per impedire l'insorgere di difficoltà di questo tipo è stato necessario garantire agli scrittori un *accesso esclusivo* alla base di dati condivisa. Qui di seguito verrà illustrata la soluzione al problema dei *lettori-scrittori* in generale. L'approccio descritto è applicato a qualsiasi file condiviso tra i thread.

La soluzione prevede la condivisione da parte dei thread lettori delle seguenti strutture dati:

```
unsigned int numlettori = 0;
semaphore mutex     = PTHREAD_MUTEX_INITIALIZER;
semaphore scrittura = PTHREAD_MUTEX_INITIALIZER;
```

Listing 3: Segmento di codice del file sorgente session.c

I semafori `mutex` e `scrittura` sono inizializzari a non bloccati; `numlettori` a 0. Il semaforo `scrittura` è comune a entrambi i thread (lettori e scrittori). Il semaforo `mutex` viene utilizzato per assicurare la mutua esclusione: garantisce che i lettori non accedino nello stesso momento durante l'aggiornamente della variabile `numlettori`. Il semaforo `scrittura` viene utilizzato per garantire la mutua esclusione per i soli thread scrittori. La variabile `numlettori` contiene il numero di thread che stanno attualmente leggendo nel file.

La figura qui sotto illustra la struttura di un processo lettore, mentre quella di seguito mostra la struttura di un processo scrittore.

```
unsigned char reader( char *command, semaphore mutex,
                      semaphore scrittura, unsigned int numlettori ) {
    unsigned char res;
    pthread_mutex_lock( &mutex );
    if ( ++numlettori == 1 ) {
            pthread_mutex_lock( &scrittura );
    } pthread_mutex_unlock( &mutex );

    res = WEXITSTATUS( system( command ) ); /* Regione critica */

    pthread_mutex_lock( &mutex );
    if ( --numlettori == 0 ) {
            pthread_mutex_unlock( &scrittura );
    } pthread_mutex_unlock( &mutex );

    return res;
}
```

Listing 4: Funzione reader del sorgente metadata.c

```
void writer( char *command, semaphore mutex ) {

    pthread_mutex_lock( &mutex );
    system( command ); /* Regione critica */
    pthread_mutex_unlock( &mutex );
}
```

Listing 5: Funzione writer del sorgente metadata.c

In questa soluzione, il primo lettore che ottiene l'accesso al file blocca il semaforo `mutex` (esecuzione di `pthread_mutex_lock`) per ottenere l'accesso esclusivo alla variabile `numlettori`. Dopodiché blocca il semaforo `scrittura` per impedire ai processi scrittori di accedere alla regione critica.

Come conseguenza di questa strategia, finché c'è un regolare arrivo di lettori, questi avranno immediatamente accesso al file, incrementando il contatore quando accedono ad essa e decrementandolo quando escono dalla regione critica. Lo scrittore, invece, rimarrà sospeso finché ci sono ancora lettori.

Forse l'unico svantaggio di questa soluzione è che viene data priorità ai thread lettori. Si potrebbe dunque pensare che tale soluzione porti a uno stato di attesa indefinita degli scrittori. Tuttavia, dal momento che le operazioni di scrittura nella base di dati sono molto più frequenti delle letture, ciò non dovrebbe condurre a nessuno stato di *starvation* né per i lettori (considerato che è stata data loro priorità) né per gli scrittori (che non vengono bloccati spesso).

3.2 Il programma client

Il programma client è l'implementazione *lato client* del protocollo MTP. Esso presenta una struttura molto più semplice rispetto alla sua controparte (programma server). E' costituito prevalentemente da un insieme di metodi utilizzati per costruire e inviare messaggi di richiesta.

Una volta avviato, il processo inizializza una connessione TCP verso il server. Ciò viene fatto creando una socket. Al momento della costituzione, il client specifica l'indirizzo della socket di benvenuto sul server, ossia l'indirizzo IP del server e il numero di porta relativo al processo. Il seguente brano di codice riassume ciò che è stato detto:

```
def __init__( self, address, port, filename ):

        self.address   = address
        self.port      = port

        self.client = socket( AF_INET, SOCK_STREAM )
        self.client.connect( ( self.address, self.port ) )
```

Listing 6: Costruttore dell'applicazione

Dopo la creazione della socket nel costruttore, TCP avvia un handshake a tre vie e stabilisce una conessione TCP con il server. Dopodiché, client e server possono scambiarsi *messaggi di richiesta* e *messaggi di risposta*.

La costruzione e l'invio di un *messaggio di richiesta* richiedono sostanzialmente l'invocazione di un metodo *omonimo*. Ecco un esempio:

```
def release( self ):

        method = 2
        action = 0

        head = str( int( method ) ) + str( int( action ) )

        # il corpo del messaggio rimane vuoto
        message = head + ''

        self.client.send( message.encode() )
        self.client.recv( 1024 )

        # Quest'ultima istruzione è necessaria
        # per svuotare il buffer ricezione associato
        # alla connessione.
```

Listing 7: Messaggio di disconnessione

L'invocazione di tale metodo provoca la costruzione e l'invio di un messaggio di disconnessione dall'account. Il messaggio passerà attraverso la socket instanziata sul lato client fino a giungere alla socket del server, il quale si preoccuperà di gestire e servire tale richiesta.