SOL - Progetto Chatty

Federico Silvestri 559014

August 31, 2018

Abstract

Questo documento contiene la documentazione relativa al progetto Chatty del modulo Laboratorio del corso A di Sistemi Operativi anno 2017/2018.

Contents

1 Architecture

L'architettura utilizzata per lo sviluppo del progetto è basata sul modello **producer/consumer**, con l'aggiunta di un ulteriore componente **controller** con la responsabilità di pilotare quest'ultimi. Sia producer che consumer sono gestiti unicamente dal controller che offre un meccanismo per lo scambio delle informazioni e ne assicura il sincronismo. Di seguito verranno descritte le funzionalità pincipali di ogni singolo componente.

Il modello Prima di descrivere l'architettura componente per componente è necessario illustrare il modello base che viene seguito. Tecnicamente parlando esso prevede l'implementazione dei seguenti metodi:

name	**description**
component_init()	lettura configurazione, allocazione memoria
component_start()	creazione del thread/thread pool
component_stop()	invio del segnale di stop al(la) thread/thread pool
component_destroy()	iberazione memoria, chiusura file descriptors

Questi quattro metodi garantiscono la robustezza e la sicurezza in quanto ognuno di essi racchiude tutte le operazioni necessarie affinchè la memoria sia allocata e liberata, i file descriptor aperti e chiusi, i thread avviati e stoppati.

Producer Il producer generalmente è il componente che si occupa di produrre il lavoro da far eseguire al consumer: in questo caso la produzione di lavoro è intesa come la ricezione di eventi sulle socket aperte dai client.

Per fare un esempio: quando un client invia un messaggio di REGISTER_OP, quest'ultimo provoca l'attivazione del Producer che dovrà occuparsi di gestire tale richiesta in modo più veloce e *fair* possibile, anche se non dovrà essere lui stesso a doverla processare.

Sintetizzando, le responsabilità principali del Producer sono:

- Creazione e binding della socket di ascolto
- Accettazione delle nuove connessioni
- Migrazione della richiesta al producer

Una volta che la richiesta è stata presa in carico dal consumer, il producer non ha nessun altra responsabilità su di essa.

Consumer Le richieste che vengono messe all'interno della coda di attesa, vengono ricevute in modo sequenziale da una delle istanze del componente consumer (in quanto multithread), che è l'endpoint del flusso di trasferimento della richiesta.

Il consumer si occupa dunque di processarla ed eventualmente di:

- rimanere in attesa di un altro evento
- chiudere la connessione
- rilasciare la connessione al producer

Controller Siccome **P e C** non sono in grado di essere autonomi, c'è bisogno di un ulteriore componente che coordina le operazioni principali, ovvero che esegue al momento giusto e nell'ordine giusto i metodi che sono stati descritti nel modello base. In particolare i punti più critici sono lo startup e lo shutdown, in quanto viene gestita la memoria e controllata la correttezza della configurazione passata al sistema.

Ipotiziamo per esempio che non venisse eseguito il metodo *init()*. Questo comporterebbe che la memoria non verrebbe allocata e dunque durante il runtime si genererebbe un errore, molto probabilmente di segmentazione.

Altra ipotesi: se non venisse eseguito il metodo *destroy*, la memoria utilizzata dai componenti non verrebbe deallocata, generando di conseguenza i temuti *memory leaks*, lo stesso discorso vale anche per i file descriptor.

Le fasi di inizializzazione e distruzione non hanno un ordine ben preciso, in generale tutti i componenti possono essere inizializzati o distrutti anche concorrentemente. Ovviamente a differenza di queste le fasi di avvio e stop devono essere eseguite secondo l'ordine prestabilito dalle necessità dei componenti. In questo contesto il producer viene avviato prima del consumer, perchè la gestione delle socket delle connessioni è affidata completamente al producer ed è di fondamentale importanza per il consumer.

Implementazione del producer Il producer viene lanciato come un thread separato dal processo principale, in modo da poterlo rendere indipendente da esso in termini di segnali e di interazione con l'utente. Durante la fase di inizializzazione viene creata e bindata la socket sulla quale arriveranno le richieste da parte dei client, controllando che non sia già stata utilizzata da un altro processo in esecuzione. Infatti un requisito fondamentale per l'avvio del producer è proprio quello appena descritto: la mutua esclusività sulla socket di ascolto. Una volta controllate le condizioni sufficienti all'avvio, con il metodo *producer_init()*, viene lanciato il thread producer, che si occuperà di gestire le connessioni con il seguente algoritmo:

```
while server_status == RUNNING do
    socksMutex.lock(); init_fds();
    socksMutex.unlock();
    select(read_fds, write_fds);
    socksMutex.lock();
    if newConn() then
        manageNewConn();
    else
        manageCurrentConns();
    end
    socksMutex.unlock();
end
```

Algorithm 1: The producer algorithm

La variabile *socksMutex* fa riferimento al meccanismo di sincronizzazione che viene utilizzato per la mutua esclusività dell'accesso al vettore delle socket. Il vettore delle socket, chiamato *sockets* all'interno del codice, contiene n interi ognuno dei quali rappresenta il filedescriptor di una specifica connessione. Tale vettore viene messo sotto controllo da un mutex perchè il consumer stesso lo utilizza per processare la richiesta dell'utente. Una volta che una connessione è stata presa in carico da un consumer in esecuzione, gli viene data la responsabilità di tutti gli eventi relativi a quella socket, fino a che non viene rilasciata dal consumer stesso. Questo meccanismo viene implementato attraverso il vettore $socket_b locks$, che contiene n booleani, dove n è il numero massimo di connessioni.

Implementazione del consumer A differenza del producer, il consumer invece può essere lanciato in modalità multithread. Infatti all'interno del file di configurazione il parametro *ThreadsInPool* specifica quanti thread dovranno essere creati per gestire tutte le richieste. L'obiettivo del consumer è quello di occuparsi dell'evento che è ricevuto nella socket. Per evento si intende:

- la scrittura di un messaggio dal client
- la disponibilità di spazio sul buffer

Nel caso in cui si parli di scrittura da parte del client, il consumer si occuperà di leggere il messaggio ed eseguire tutte le procedure per completare la richiesta. Nel in cui si parli di lettura il consumer invier i messaggi disponibili al client. Tutte queste funzionalità sono state implementate nelle libreria nominata *worker*.