SOL - Progetto Chatty

Federico Silvestri 559014

September 2, 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** che ha 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*.

2 Librerie esterne

Per lo sviluppo di questo progetto sono state utilizzate delle librerie esterne, non tanto per semplificare, ma per rendere più efficiente e stabile possibile

l'applicazione.

2.1 RabbitMQ

La gestione delle code è stato uno degli argomenti informatici più richiesti e più studiati nel giro degli ultimi anni. Per questo motivo sono stati sviluppati diversi software che forniscono un servizio di *Message brokering*, ovvero di scambio di messaggi tra componenti di software diversi. La scelta che è stata fatta per questo progetto è stata quella di usare *RabbitMQ*, in quanto è uno dei più semplici e più user friendly con il linguaggio C. RabbitMQ è dunque una dipendenza del progetto che viene installata automaticamente durante la fase di build di chatty, e viene automaticamente disinstallata una volta lanciato lo script di pulizia.

Configurazione La configurazione di RabbitMQ avviene durante le fasi di inizializzazione del producer e del consumer. In particolare vengono fatte due configurazioni diverse, ognuna delle quali è caratterizzata da una connessione TCP dedicata verso il server RabbitMQ.

In generale quello che viene fatto per configurare l'applicazione sono le seguenti azioni:

- creazione dell'exchage di tipo fanout
- creazione della coda per l'exchange
- binding della coda all'exchange

All'interno del file di configurazione possono essere configurati ulteriori parametri, che sono indicati nella sezione configurazione.

Utilizzo RabbitMQ viene utilizzanto fondamentalmente nel progetto, ovvero esso è il mezzo di comunicazione tra producer e consumer. Quando il producer identifica l'arrivo di un evento all'interno della socket, compone un messaggio, compilando un array con la seguente struttura:

- nell'indice 0 viene inserito il tipo di azione, se WRITE o READ.
- nell'indice 1 viene inserito l'indice del file descriptor della socket.

Una volta che il messaggio è stato ricevuto da RabbitMQ, lo inserisce all'interno della coda già preconfigurata durante la fase di init. Ad eseguire l'operazione di *dequeue* è il consumer, che sfruttando un meccanismo ad eventi, sveglia il thread dallo stato di *WAIT* e lo porta in *RUN*. Una volta che RabbitMQ si accorge che tale messaggio è stato preso in carico da un agente (nel nostro caso un'instanza del Consumer), lo rimuove dalle coda, marcandolo come completato. In questo modo il sistema è risultato fair e stabile durante la fase di test.

Attenzione, il mapping tra connessione e thread non è strettamente 1:1

Questo perchè quando il thread ha completato una richiesta, rilascia la socket attraverso il metodo *producer_release_socket()*, riattivando dunque il producer a scansionare gli eventi relativi. Il thread che termina la richiesta, viene rimesso in attesa di messaggi sulla coda di RabbitMQ.

2.2 SQLite3

Un'altra libreria che viene utilizzata è la famosa SQLite3, ovvero una libreria che permette di creare un database relazionale all'interno di un singolo file con estensione .sqlite. Questa scelta è stata fatta affinchè il progetto potesse avere una base solida, veloce e affidabile per la gestione dei dati. In particolare, al posto di creare un sistema "artigianale" in caso per la memorizzazione di strutture dati, sono state create tabelle e query, per poter memorizzare utenti, messaggi e gruppi.

Tables Le tabelle che vengono create sono le seguenti:

table	**Primary key**	**description**
users	nickname	contiene tutti gli utenti che si sono registrati
messages	ID	contiene tutti i messaggi che sono stati inviati dagli utenti

La gestione del database è completamente affidata al worker, ovvero a quel file descritto nella documentazione del codice che contiene tuttte le funzionalitá per la persistenza dei dati. Come **P e C** anche il worker deve essere inizializzato e distrutto ma non implementa un modello asincrono. Le funzioni del worker sono tutte thread safe, dunque non c'è bisogno di implementare meccanismi di sincronismo, a differenza del datastore.

2.3 Libconfig

Questa libreria viene utilizzata dalla maggior parte del mondo opensource per la lettura e il parsing di file di configurazione, infatti è già installata nelle distro come Ubuntu, Debian e Centos. La sua inizializzazione viene eseguita subito dopo il controllo degli argomenti passati al server. Nel caso in cui il file di configurazione non esista, sia illeggibile (parlando di permessi), o contenga un errore di sintassi, viene lanciato immediatamente un errore (*config error*).

Formato del file di configurazione Il formato del file di configurazione, come specificato nella documentazione della libreria, è quello standard, ovvero del tipo:

```
<conf-name> = <conf-value>
<conf-value> := "<string-value>" | <other-types>
```

Gli altri tipi di configurazione messi a disposizione da LibConfig, come per esempio quelli per gli oggetti, non sono stati utilizzati per questo progetto. C'è da fare particolare attenzione alla differenza ai file *.conf1* e *.conf2*, perchè sono stati adeguati al formato richiesto dalla libreria, cioè sono stati aggiunti i doppi apici all'inizio e alla fine di un parametro stringa.

2.4 Log

La libreria di log è quella più semplice che è stata utilizzata. Non richiede l'installazione di file .so, perchè è inclusa nel progetto stesso. Infatti il file *log.c* e il relativo header *log.h* contengono le funzioni per la gestione dei logs. Principalmente ci sono 4 livelli di log che possono essere utilizzati, e sono:

- *FATAL*
- *ERROR*
- *WARN*
- *INFO*
- *DEBUG*
- *TRACE*

È anche possibile usufruire di un ulteriore funzionalità chiamata *QUITE*, che consente di sopprimere tutti i tipi di errori che sono stati lanciati. Questa libreria è anche stata resa dal sottoscritto thread safe, utilizzando un meccanismo di mutua esclusione.

3 Struttura del codice

Il codice consegnato in realtà non è stato sviluppato nella struttura così come si trova, bensì sono state generate le directory

- *src* per tutti i file *.c*
- *include* per tutti i file *.h*
- *scripts* per tutti i file *.sh*
- *makefiles* per tutti i Makefile che sono stati testati
- *DebugGCC/DebugClang/Release* per tutti i file processati dai rispettivi compilatori

Per rendere la struttura del progetto conforme a quella richiesta, è stato creato uno script build_project che esegue la copia dei file dentro le directory nella directory *Release*. Infatti i file consegnati sono esattamente quelli dentro la directory Release. Se ci fosse necessità di ispezionare gli ulteriori file del progetto è possibile scaricare l'intero progetto dalla repository git da: `https://git.com.divisible.net/federicosilvestri/chatty` che però richiede l'autenticazione da parte dello studente.

4 Configurazione del server

Oltre ai parametri richiesti nelle specifiche sono stati aggiunti altri parametri, descritti nella seguente tabella:

name	**type**	**description**
$RabbitMQHostname$	string	l'hostname del server RabbitMQ
$RabbitMQPort$	integer	la porta sulla quale è in ascolto il server RabbitMQ
$RabbitExchange$	string	il nome dell'Exchange da creare per lo scambio dei messaggi
$RabbitBindKey$	string	chiave per proteggere l'exchange da binding sconosciuti.

5 Piattaforme testate

Il progetto è stato sviluppato usando un sistema di Continuos Testing, fornito da Gitlab. Pertanto ogni volta che veniva eseguito un commit nel master branch, seguendo il file di configurazione *.gitlab-ci.yml* (non fornito nella consegna), il sistema buildava il progetto ed eseguiva i test sotto ambienti:

- Ubuntu 18.04
- Ubuntu 16.04
- Debian 9

In caso di test negativo, è stato ribuildato il progetto sulla macchina locale, in modo da debuggare al meglio i problemi incontrati.

5.1 Problemi che possono verificarsi

Il problema che può verificarsi, durante la fase di testing è un errore che poteva essere soppresso, ed è dovuto al problema dell'incompletezza del protocollo usato dal client. In sostanza si verifica perchè non c'è modo di capire il momento in cui il client è in ascolto per nuovi messaggi oppure è in ascolto per altri motivi. Infatti nel protocollo seguito dal client per qualsiasi cosa si voglia fare viene inviata una richiesta al server con il codice dell'operazione, tranne per la ricezione di messaggi. In questo modo il server non è in grado di capire le intenzioni del

client, e dunque utilizza una strategia brutale ma funzionante, cioè utilizza il terzo parametro della funzione *select* per capire quando c'è spazio disponibile sul buffer della socket e se l'utente è autenticato, allora viene inserito nella coda un messaggio di tipo $WRITE$. Alcune volte si verifica il messagio di errore "Broken pipe", ed è riferito al fatto che il server tentando di controllare lo stato dell'utente (ormai disconnesso), riceve dal kernel la disconnessione. Tuttavia questo errore che non è stato possibile evitare, se non:

- modificando il protocollo del client
- aggiungendo una sleep durante la fase di ricezione

non ha prodotto problemi rilevanti durante il runtime del client, dunque anche durante i test.

6 Riferimenti

Gitlab	`www.gitlab.com`
RabbitMQ	`www.rabbitmq.com`
SQLite3	`www.sqlite3.org`
LibConfig	`https://hyperrealm.github.io/libconfig`