Relazione Progetto Laboratorio di Sistemi Operativi

Nicola Dardanis - 579047

A.A. 2018/2019

Indice

1 Scelte progettuali **3**

1.1 Gestione della memoria 3

1.1.1 Stack e heap 3

1.1.2 Bufferizzazione dei messaggi 3

1.2 Gestione dei segnali 4

1.2.1 Sighandler 4

1.2.2 Soft Shutdown e consistenza dei dati 4

1.3 Gestione degli errori 5

1.3.1 Server - thread principale 5

1.3.2 Server - thread servente 5

1.3.3 Libreria di accesso 6

1.4 Rappresentazione interna delle informazioni 6

2 Makefile e scripts **7**

2.1 Makefile 7

2.2 build.sh 7

2.3 Formato del file di log 7

2.4 Sommario dei tests 8

3 Strutture dati di appoggio **9**

3.1 Lista concatenata 9

3.2 Statistiche del server 9

3.2.1 Informazioni raccolte 9

3.2.2 Gestione della concorrenza 9

3.3 dataBuffer_t 9

1 Scelte progettuali

Di seguito sono illustrate le principali scelte progettuali effettuate durante la realizzazione del progetto.
L'applicazione si compone, come da consegna, di un server multithread che ad ogni richiesta di connessione ricevuta crea un thread servente dedicato, di una libreria statica per la comunicazione con il server e un client di test.

1.1 Gestione della memoria

1.1.1 Stack e heap

Si è preferita una allocazione su stack quando possibile, rispetto a una su heap, in quanto:

1. i varii threads condividono la memoria heap con rischio di saturazione durante lo scambio concorrente di dati di grande dimensione;

2. si sposa bene con la bufferizzazione dei dati scambiati fra client e server (v. 1.1.2, 3.3);

3. il GC è semplificato riducendo il rischio di memory leak e l'ultilizzo di strutture dati dedicate (es. lista concorrente dei puntatori allocati da deallocare nella funzione di cleanup).

In particolare l'allocazione dinamica è impiegata solo nel passaggio del descrittore della socket ai threads serventi (cfr. *server.c*), per le statistiche del server che utilizzano liste concatenate (v. 3.1) e nella libreria di accesso (cfr. *api.h*), per memorizzare i dati ricevuti dal server in un unico blocco da restituire. Il server registra una funzione di cleanup con lo scopo di liberare tutta l'eventuale memoria dinamica allocata e gli eventuali file temporanei.

1.1.2 Bufferizzazione dei messaggi

Il server gestisce la ricezione e l'invio dei messaggi attraverso buffers allocati su stack. In particolare ogni thread:

1. esegue un loop principale nel quale si mette in attesa della ricezione di un header dal client, viene instanziato un buffer che, conformemente allo standard POSIX dei nomi, sia abbastanza grande per contenere comando e parametri;

eventuali dati in eccesso sono salvati su un buffer temporaneo di dimensione tale da assicurare che non si verifichi un errore di BUFFER_OVERFLOW (cfr. *messages.h*, v. 3.3). Le dimensioni dei buffers sono configurabili in *config.h;*

2. esegue un loop interno per le richieste di STORE(RETRIEVE) in cui utilizza un buffer statico per la lettura/scrittura da(a) socket a(da) file, sempre fornendo la lunghezza effettiva dei dati da scrivere e salvando quella dei dati letti (con riuso della memoria v. 3.3). Si noti che i file sono gestiti mediante chiamate POSIX, che rispetto alle funzioni della libreria STDIO non sono bufferizzate in spazio utente.

Questa scelta permette la comunicazione con il server in modo efficiente, configurabile e nel rispetto del protocollo fornito, senza assunzioni sul modo in cui i messaggi debbano essere effettivamente inviati (se in un unico blocco o a pacchetti).
Nella libreria di accesso (cfr. *api.h*) si segue lo stesso modello nel caso di richiesta di RETRIEVE, mentre per la richiesta di STORE, poiché la funzione *os_store* ha come argomento l'intero blocco di dati di cui è richiesta la memorizzazione, per semplicità viene effettuata una sola scrittura su socket.

1.2 Gestione dei segnali

1.2.1 Sighandler

I segnali sono gestiti da un thread detached dedicato (*sighandler*), abilitato alla loro ricezione [1], creato subito dopo l'inizializzazione delle statistiche del server (cfr. *server.c*). Alla ricezione di un segnale la chiamata a sigwait ritorna e viene settata la variabile globale di stop[2][3] (*loop_breaker*). Nel caso si tratti di SIGUSR1 vengono prima stampate le informazioni statistiche del server.

1.2.2 Soft Shutdown e consistenza dei dati

La variabile di stop *loop_breaker* è presente nella guardia dei loops principali del thread principale e dei threads serventi, pur essendo una variabile condivisa non è protetta da mutex poiché è settata solo dal *sighandler.* Per assicurare la ricezione del segnale di stop da parte di tutti i threads attivi, le chiamate bloccanti alla accept e alle read su socket all'interno dei loops principali vengono gestiti con una select con timeout[4] (cfr. *server.c*) in modo da scongiurare il blocco in lettura per un tempo indeterminato.
Il thread principale una volta uscito dal ciclo si mette in attesa della terminazione dei

[1] N.B. Nel thread principale e nei thread serventi per le connessioni sono bloccati.
[2] Per le modalità di shutdown si veda 1.2.2
[3] N.B. Anche il segnale delle statistiche provoca lo stop del server, come da consegna: «Il server quando riceve un segnale termina il prima possibile»
[4] Configurabile in *config.h*

threads serventi, con wait su variabile condizione del contatore (*active_threads_counter*) protetta da mutex. Ogni thread prima di uscire decrementa la variabile contatore dei threads attivi.[5]
La consistenza dei dati è garantita poiché:

- prima della chiusura dell'applicazione si aspetta che tutti i threads attivi concludano le loro operazioni;
- ogni thread nella guardia dei loops interni (che operano sui dati per eseguire STORE e RETRIEVE) non esegue controlli sulle variabili globali di stop. Inoltre, in caso di STORE, la lettura dalla socket è effettuata senza la gestione con select e timeout, quindi continua fino a completa terminazione della lettura dei dati in arrivo dal client o termina per un errore.

1.3 Gestione degli errori

La gestione degli errori è standardizzata attraverso l'uso di macros definite nell'header *myerror.h.* Inoltre per una maggiore chiarezza e gestione sono state definite enumerazioni per tipologie di errori (cfr. *serverapi.h, messages.h*) [1.4].

1.3.1 Server - thread principale

Gli errori che si generano nel server prima di attendere connessioni sono fatali e provocano la terminazione del processo. Un errore restituito invece dalle chiamate per l'accettazione della connessione o la creazione del thread servente dedicato è sempicemente ignorato. Nella guardia del ciclo di accettazione viene verificata anche la variabile booleana di errore *global_failure*, utilizzata per segnalare un errore non gestibile dell'intero processo server. [6]

1.3.2 Server - thread servente

Ricalcando il modello del processo principale, i varii threads serventi dichiarano due proprie variabili booleane di fine ciclo: *own_breaker*[7] e *own_failure.* Queste assieme alla variabili di terminazione del processo costituiscono la guardia del loop principale del singolo thread. In particolare *own_failure* è usata per terminare il thread al verificarsi di errori che compromettono la comunicazione e l'esecuzione di future richieste con il solo client servito. Tutti gli errori di esecuzione del comando richiesto sono notificati al client secondo il protocollo, con brevi messaggi informativi sulla natura del problema quando disponibili.

[5] N.B. Si è preferito l'approccio threads detached + wait su variabile condizione a un approccio con threads su cui chiamare la join per motivi di semplicità della gestione (il secondo caso ha bisogno di una struttura dati di appoggio in cui memorizzare gli ids dei threads).
[6] Es.: la cartella *data* non è accessibile o non esiste al momento della memorizzazione di un file
[7] Usata per terminare il thread in caso di LEAVE o di connessione fallita in casi gestibili.

Se riscontrato un errore compromettente il funzionamento dell'intera applicazione si notifica l'intero sistema server tramite settaggio a true di *global_failure*.

1.3.3 Libreria di accesso

La libreria di accesso (*api.h*) gestisce tutti gli eventuali errori interni, come da consegna, semplicemente ritornando al chiamante i valori che segnalano per le varie funzioni il verificarsi di un errore (aproccio standard per una libreria).

1.4 Rappresentazione interna delle informazioni

All'interno del progetto sono presenti varie definizioni di tipo per la rappresentazioni di informazioni composite (struct) e per le varie tipologie di errori o richieste (enum), nell'ottica di fornire un codice maggiormente leggibile e mantenibile. In particolare si è preferito:

- rappresentare per enumerazione le tipologie di richieste(risposte) che un client(server) può inviare a un server(client) dopo la fase di parsing in modo da rendere più agevole e comprensibile la gestione di operazioni non riconosciute e la selezione delle funzioni preposte allo svolgimento dei relativi compiti;
- raccogliere tutte le informazioni degli header in strutture dedicate, così da differenziare la fase di parsing del messaggio (cfr. *messages.h*) da quella di esecuzione del comando;
- rappresentare gli errori per enumerazioni in modo da fornirne una gestione puntuale e ordinata.

2 Makefile e scripts

Nel progetto sono presenti anche un makefile e tre scripts: lo script di test (*test_script.sh*), quello di analisi (*testsum.sh*) e uno script aggiuntivo di build (*build.sh*).

2.1 Makefile

Il makefile oltre ai target *all*, *clean* e *test* richiesti fornisce anche le opzioni:

cleanall rimozione di eseguibili, cartelle e files generati automaticamente.

debug compilazione del codice in modalità di debug: attiva controlli aggiuntivi e la stampa di informazioni utili al debugging.[1]

production compilazione del codice in modalità produzione (alias di *all*).

testd come *test,* ma compila il codice attraverso il target *debug.*

devtest compilazione di *mytest.c* (in *mytest.exe*) contenente test aggiuntivi di funzionalità del progetto utilizzati in fase di sviluppo.

2.2 build.sh

Script di utilità per la compilazione, chiamato dal makefile per compilare i sorgenti del progetto secondo la modalità specificata (debug | produzione)

2.3 Formato del file di log

Il file *testout.log* contiene l'output di ogni client eseguito in *test_script.sh.* Ogni riga contiene quindi le informazioni raccolte nella struct *client_stats_t* (cfr. *client.c*) codificate in una stringa [chiave]:[valore], in cui ogni coppia è separata da virgole.

[1] N.B. Per visualizzare su STDOUT le notifiche aggiuntive durante l'esecuzione del target *testd* è necessario commentare la redirezione dello STDERR in *test_script.sh.*

2.4 Sommario dei tests

Il sommario dei tests contiene informazioni generali, quali numero di clients lanciati in totale e percentuali di connessioni e disconnessioni avvenute con successo, e informazioni più granulari sia sulla tipologia (batteria) di test, come numero di clients che sono stati lanciati per test, tempo medio di elaborazione delle esecuzioni con successo (utile a stimare le performance dell'intero sistema client - server), sia sulle singole funzioni offerte dalla libreria con conteggio delle chiamate totali e percentuali di fallimento.

3 Strutture dati di appoggio

In questa sezione vengono descritte le principali strutture dati e la loro utilizzazione nel progetto. Per le restanti si rimanda al codice.

3.1 Lista concatenata

I files *liste.h* e *liste.c* contengono l'interfaccia e l'implementazione di una lista concatenata semplice con chiavi puntatori a caratteri. Viene utilizzata esclusivamente nel codice del server per la memorizzazione dei clients online e di quelli presenti nello store. Le stringhe sono mantenute secondo l'ordine lessicografico.

3.2 Statistiche del server

3.2.1 Informazioni raccolte

Le statistiche che il server stampa alla ricezione del segnale USR1 sono memorizzate nella struct *statistics_t* (cfr. *serverapi.h*). Contengono il numero di utenti dello store, di quelli online, di files, la dimensione complessiva di essi e la lista dei nomi dei clients online e complessivi. Le informazioni sono inizializzate dal server subito dopo aver bloccato i segnali nel thread principale e prima di ogni altra operazione (accettazione di connessioni) in modo da assicurare un corretto collezionamento di queste che non possono essere alterate da altrimenti possibili modifiche concorrenti in corso nella directory *data*.

3.2.2 Gestione della concorrenza

Le statistiche sono dichiarate globalmente all'interno del file *serverapi.c* che gestisce internamente l'accesso concorrente per la modifica di tale struttura attraverso funzioni ausiliarie non esportate che operano limitate da una mutex dedicata (anch'essa globale).

3.3 dataBuffer_t

Struct che mantiene al suo interno un buffer di caratteri, la lunghezza di esso e lo spazio occupato. Utilizzata per una gestione coerente, efficiente, in termini di memoria allocata e riuso di essa, e configurabile (cfr. *config.h*) dei buffers.