Relazione SOL lab 2018

Remo Andreoli 535485 (corso B)

Chatty e' un server concorrente che fornisce un servizio di chat strutturato in maniera chiara e implementata semplicisticamente, con l'obiettivo di rendere il piu' semplice possibile la comprensione del codice sorgente

Strutture dati principali

Hashtable: Usata per memorizzare gli utenti registrati alla chat; nello specifico, memorizza la history (vedi sotto) del relativo utente. Versione leggermente modificata dell'implementazione fornita a lezione al fine di rendere le istruzioni di inserimento/ricerca/cancellazione concorrenti. L'Hashtable e' stata inoltre arricchita con funzioni ausiliarie per la free della history in fase di chiusura o deregistrazione dell'utente.

HistoryQueue: Usata per salvare gli ultimi messaggi inviati ad un utente; Implementata come una coda circolare a grandezza fissa con meccanismo di replacement dei messaggi piu' vecchii. Mantiene inoltre un flag per ogni messaggio, per indicare se e' stato effettivamente recapitato o meno. Le istanze della history sono corredate da lock allocate dinamicamente per garantire la thread-safety.

Queue: Usata in due istanze diverse: per la comunicazione listener-worker e per memorizzare la struttura contenente i parametri degli utenti connessi (vedi sotto); Versione modificata dell'implementazione fornita a lezione, cosi' da garantire la thread safety.
Sono inoltre state implementate delle funzioni ausiliarie per facilitare le operazioni sugli utenti online, come per esempio la ricerca o la rimozione di un utente connesso.

User: Usata per memorizzare le informazioni relative ad un utente online: nickname, fd del client e index della lock associatagli per la comunicazione; la lock, scelta da un pool, viene consegnata all'utente al momento della connessione.

Listener/WorkerArgs: Usate per memorizzare i parametri di inizializzazione del listener e dei workers, principalmente il pointer alla coda degli fd e la pipe(con relativa lock) per realizzare la comunicazione bidirezionale;

ThreadPool: Usata per gestire i thread worker; correlata con funzioni ausiliare per l'inizializzazione e join dei thread e chiusura del threadpool.

Gestione memoria

Il server fa largo uso di memoria dinamica per mantenere costantemente i pointer alle strutture dati. Il numero di variabili globali e' stato ridotto al minimo.

Strutturazione del codice

Il server e' suddiviso su piu' file: Il file principale e' chatty.c, che include l'implementazione dell'inizializzazione/chiusura del server, delle strutture dati, dei thread e l'implementazione dei servizi offerti ai client;
Da esso vengono inizializzati il thread che gestisce i segnali e i thread che implementano la comunicazione tra server e client: i thread worker, gestiti da un threadpool e responsabili dell'esecuzione delle richieste dei client, e il thread listener, responsabile dell'acquisizione e smistamento (usando una coda concorrente) delle richieste.
Le richieste avvengono sottoforma di un'apposita struct memorizzata in message.h e le funzioni che implementano l'invio sono memorizzate in connections.c.
L' implementazione del thread listener, dei thread workers, del threadpool e delle strutture dati usate per la memorizzazione sono suddivise su file diversi e tutte le funzionalita' implementate da chatty.c sono elencante nell' header file server.h, che appunto definisce le funzioni relative ai servizi e le strutture dati di supporto ai thread.

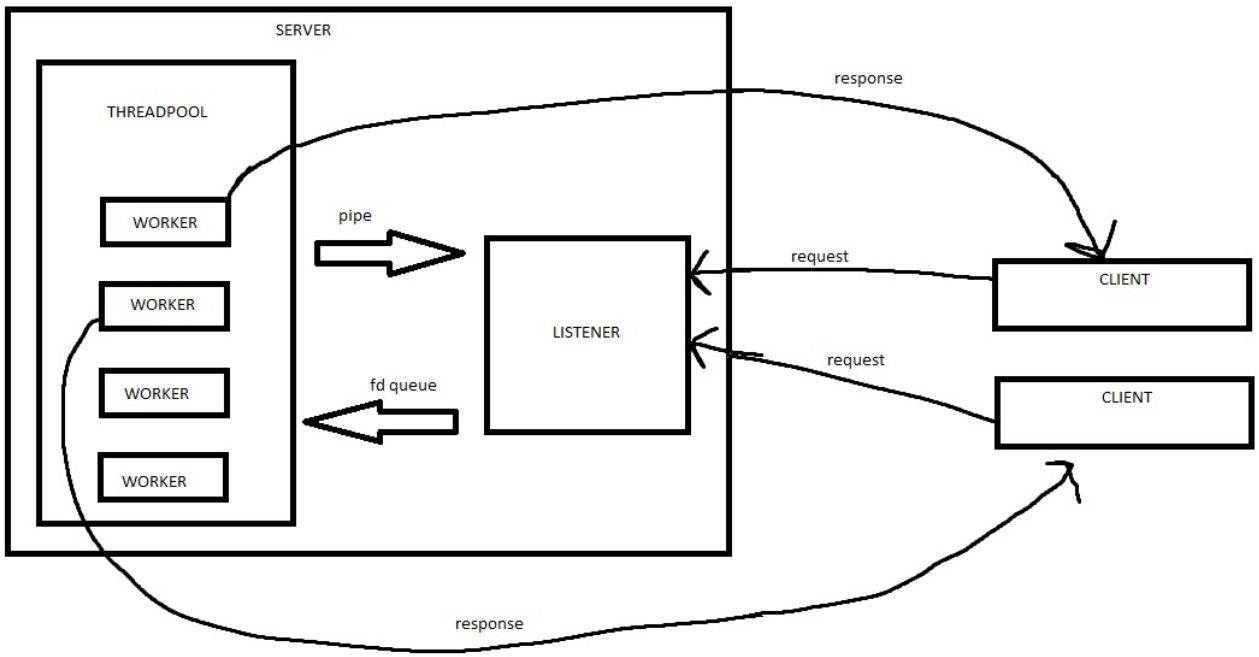
Il server e' riconfigurabile usando i parametri memorizzati nel file di configurazione e le macro memorizzate nel header file config.h; il primo contiene parametri generali di inizializzazione del server, mentre il secondo file contiene valori massimi e i parametri di configurazione della hashtable; tali parametri si trovano in config.h per comodità, ma potrebbero essere benissimo spostati nel file di configurazione modificando poche linee di codice.
Il file header utils.h contiene una macro per verificare la corretta esecuzione delle system call nel processo di inizializzazione del server, la struttura di enum del file di configurazione e la definizione della funzione con cui invocare del parser di tale file, che e' implementato nel file parser.c.
Il file pthreadUtils.h, con la relative implementazione pthreadUtils.c, contiene la definizione delle funzioni per la gestione degli errori sulle operazioni della libreria pthread.
Il file ops.h contiene l'enum usato dal client e dal server per identificare i messaggi inviati e ricevuto.
Il file stats.h contiene la struct delle statistiche raccolte dal server, corredata con la funzione di stampa.

Accesso ai file esterni

Il server accede al file di configurazione all'inizializzazione, appende le statistiche in un file specifico e per ogni scambio di file tra i client ne mantiene una copia; le operazioni nella fase di scambio sono implementate usando systemcall primitive e il file mapping.

Interazioni tra threads

Il thread listener interagisce con i clients e con i thread worker; con il primo interagisce mediante i socket, da cui "raccoglie" le richieste, mentre con i secondi interagisce usando una coda concorrente degli fd dei client che richiedono un servizio; i worker comunicano con il listener attraverso una pipe concorrente, per risettare gli fd dei client nella maschera della select e continuare la comunicazione (coda: listener->worker - pipe: worker -> listener).
Anche i workers interagiscono con i clients, a cui inviano i responsi delle richieste.



Gestione della concorrenza

La concorrenza nella coda degli fd e degli utenti online e' realizzata usando lock e condition variables locali e dinamiche, in modo tale da poter usare la struttura in piu' istanze e casi d'uso diversi.
L' hashtable e' resa concorrente usando N lock dinamiche (con N qualunque definito in config.h con la macro HASHTABLE_REGION), che vanno a dividire la tabella in N porzioni accessibili in maniera concorrente e parallela: all'inizio di ogni operazione, l'hash value dell'utente che richiede il servizio viene diviso per il numero di entry della hashtable (nbuckets), andando cosi' a trovare l'index del lock da occupare; il resto della tabella e' ancora accessibile senza race condition, permettendo anche accessi in parallelo.
Per quanto riguarda la realizzazione della concorrenza nel caso delle connessioni server-client, ad ogni client connesso e' assegnata una lock (presa da una sorta di "lockpool" allocata dinamicamente nella fare di inizializzazione del server), che verra' usata dall'utente per usufruire in sicurezza dei servizi offerti: ogni operazione di invio messaggi richiede l'uso della lock relativa all'utente; la stessa lock viene anche usata per aggiungere i messaggi alla History; La fase di assegnamento della lock e' resa priva di race condition usando un'apposita lock globale.
Sono fornite di lock anche le operazioni sulla struttura che memorizza le statistiche e sulla pipe di comunicazione worker->listener

Gestione segnali

Il server ignora tutti i segnali e rilega ad un determinato thread il compito di attenderli usando la sigwait: SIGPIPE viene semplicemente ignorato da tutti(thread gestore segnali incluso), SIGUSR1 invoca la stampa delle statistiche su file e i segnali di terminazione(SIGQUIT, SIGTERM, SIGINT) causano la terminazione del thread gestore; tale terminazione avvia la fase di chiusura del server, implementata come una reazione a catena di eventi brevemente descritta sotto.

Gestione terminazione del server

Subito dopo la terminazione del thread che gestisce i segnali, viene chiusa la pipe di comunicazione tra listener e worker: nel momento in cui il listener prova a leggere la pipe chiusa, viene settato la flag che termina il loop principale, chiude gli fd e inserisce nella coda concorrente tanti messaggi “di chiusura” quanti sono i thread worker; quest’ultimi, alla ricezione di tali messaggi, terminano il loop principale.
Il threadpool, situato sul thread principale (eseguito da chatty.c), attende quindi i valori di ritorno dei worker.
Successivamente vengono liberate e deallocate la coda degli fd e degli utenti online, le varie mutex usate dal thread principale, e infine l’hashtable degli utenti registrati, che a sua volta provoca la deallocazione delle code history utente per utente.

Note finali importanti

- Il progetto e’ stato scritto usando VIM editor, sono presenti errori di formattazione del codice se aperto con altri editor

- Il progetto e’ stato principalmentetestato su una macchina con sistema operativo Ubuntu; una versione precedente del progetto e’ stata testata sulla macchina virtuale con esiti positivi, ma non quella corrente per problemi a me sconosciuti (non sono riuscito ad accedere ne’ a Dropbox ne’ a MEGA)

- Il progetto richiede l’uso della libreria Math.h per il calcolo del floor al momento della suddivisione della hashtable in partizioni; di conseguenza, il Makefile e’ stato modificato per includere -lm

- Non avendo mai usato Doxygen prima, non sono sicuro della buona riuscita della documentazione, ma il codice e’ comunque sintatticamente ben documentato