Chatterbox (Chatty)
Progetto SOL 2017/18

Alexandra Bradan [530887]

28 marzo 2019

Indice

1 Strutture dati utilizzate

1.1 HashTable

Struttura dati utilizzata per memorizzare gli utenti registrati alla Chat e la relativa coda dei messaggi (History). Organizzata come un array associativo di dimensione finita (la dimensione è in funzione del numero di connessioni estrapolato dal file di configurazione e moltiplicato per una potenza di due per facilitare l'acquisizione delle mutua esclusione sui buckets, che risolve le collisioni con il metodo del "SEPARATE CHAINING". L'idea dell'indicizzazione dei buckets della HashTable viene ripresa dal "DOUBLE HASHING", con l'indice risultante avente la seguente segnatura:

$$index = (hash_a(string) + i * (hash_b(string) + 1)) mod\ num\_buckets$$

dove:

1. i = è il numero di tentativi (dato che le collisioni vengono risolte con le liste di trabocco, i = 1);
2. string = è la chiave di cui bisogna calcolare l'hashing;
3. num_buckets = dimensione della HashTable;
4. hash_a e hash_b = funzione hash chiamata con due numeri primi, a e b, aventi dimensione maggiore dei caratteri alfanumerici ASCII (caratteri alfanumerici ASCII variano da 0 a 127). Nel mio caso ho scelto arbitrariamente a = HT_PRIME_1 = 151 e b = HT_PRIME_2 = 163.

La funzione hash ha, invece, la seguente segnatura:

```
int ht_hash(const char* s, int a, int m){
long hash = 0;
const int len_s = (int)strlen(s);

for (int i = 0; i < len_s; i++) {
    //s[i] = rappresentazione numerica del carattere
    hash = hash + (long) pow(a, (len_s - (i+1))) * s[i];
    hash = hash % m;
}

return (int)hash;
}
```

Gli step fondamentali della funzione hash sono i seguenti:

- convertire la stringa ad un intero piu' grande;
- ridurre la dimensione di questo intero ad un range fissato (quello dei buckets della HashTable), usando il modulo della dimensione della HashTable.

Così facendo, per i nickname usati nei vari test, le collisioni sono risultate essere nulle.

1.2 UsersQueue

Gli usernames che collidono sullo stesso bucket della HashTable vengono risolti tramite una lista di trabocco. Le liste di trabocco sono implementate come double-linked lists, di cui si tiene il riferimento all'ultimo nodo inserito per efficientare l'inserimento di un nuovo username collidente.

1.3 HistoryQueue

La History di ogni utente della Chat è memorizzata nel nodo della lista di trabocco nella quale l'utente si trova, sotto forma di un array circolare di dimensione finitia (MAXHISTMSGS), con un meccanismo di replacement dei messaggi più vecchi.

1.4 lockItems

Array di dimensione finita (MAXCONNECTIONS) che permette l'accesso in mutua esclusione, per sezioni, alla HashTable.

1.5 OnlineQueue

La coda degli utenti connessi in un dato momento è implementata all'interno della struct che raccoglie la HashTable e le informazione ad essa legate. Viene memorizzata qui perchè tale coda tiene riferimento non solo del file descriptor di un client connesso, ma anche dell'indice del bucket in cui l'utente è memorizzato e del puntatore al nodo della lista di trabocco in cui esso si trova. Questo consente di efficientare sia l'hashing per accedere alla HashTable (l'indice del bucket relativo viene calcolato solo all'atto della connessione dell'utente, per verificare che sia effettivamente registrato alla Chat e memorizzato qui), sia il reperire l'utente nella lista di trabocco (senza dover ogni volta scorrere tale lista) tramite un puntatore relativo. La coda degli utenti connessi è strutturata come un array di dimensione finita (MAXCONNECTIONS*BASE_ONLINEQUEUE=32), indicizzata tramite i file descriptors assegnati dalla Select.

1.6 HibernationQueue

La coda di ibernazione è una struttura ausiliaria utilizzata dal ThreadPool per gestire l'attesa indefinita di uno o piu' messaggi/files da parte di quei clients che si sospendono in attesa di ciò. La coda di ibernazione raccoglie i *file descriptor* di questi clients (oppurtunamente modificati, sommando loro + FD_SETSIZE = 1024, per distinguerli dai clients che non stanno facendo l'operazione "MENO_R_OP"), in un array di dimensione finita (MAXCONNECTIONS*BASE_ONLINEQUEUE). Per comprendere meglio il funzionamento di tale array si invita a leggere la sezione "MENO_R_OP.

1.7 threads

Array di dimensione finita (THREADSINPOOL) che raccoglie gli ID degli worker-threads creati all'atto della creazione del ThreadPool e fatti terminare dopo il soppraggiungere di uno dei segnali di terminazione e di esaurimento delle richieste presenti nell'array di lavoro.

1.8 arguments

Array di dimensione finita (THREADSINPOOL) che contiene gli argomenti da passare all' i-esimo worker-thread, con $0 \leq i <$ THREADSINPOOL. Gli argomenti che ogni worker-thread riceve sono il riferimento al ThreadPool per poter accedere all'array di lavoro, il riferimento alla HashTable per poter accedere agli utenti registrati e all' array dei connessi e l'indice dello slot dell'array "threads" in cui si trova l'ID del worker relativo. Tale indice consente ad ogni worker-thread di accedere all'array delle statistiche e poter aggiornare le sue statistiche locali.

1.9 threadStats

Array di dimensione finita (THREADSINPOOL) che raccoglie in ogni slot le statistiche locali dell' i-esimo worker-thread, con $0 \leq i <$ THREADSINPOOL.

2 Gestione della memoria

L'array associativo che rappresenta la HashTable, le liste di trabocco di ogni bucket con le quali si risolvono le collisioni, gli array circolari di dimensione finita che rappresentano le Histories degli utenti della Chat, l'array di locks che permette l'accesso alla HashTable, l'array degli utenti connessi in un dato momento, gli ID degli worker-threads raccolti nell'array *threads*, gli argomenti da passare agli worker-threads racchiusi nell'array *arguments*, l'array che raccoglie le statistiche locali degli worker-threads, l'array di lavoro dal quale gli worker-threads devono estarrre i tasks da portare a termine e la coda di ibernazione dei clients che stanno facendo *attesa attiva* sono tutti allocati dinamicamente all'atto dell'avvio del listener-thread e deallocati all'arrivo di uno dei tre segnali di terminazione(SIGINT, SIGQUIT o SIGTERM).

3 Struttura del codice e threads usati

Il primo thread che viene messo in esecuzione è il "chatty main" che ha il compito di effettuare il parsing del file di configurazione, creare il listener-thread e metterlo in esecuzione, attedere la sua terminazione e deallocare le variabili globali che sono state allocate dinamicamente all'atto del parsing.

Il listener-thread si occupa di registrare e gestire l'arrivo dei segnali (VEDI SEZIONE SEGNALI), creare il *welcome socket*, allocare le strutture dati necessarie alla Chat (Hashtable e array degli utenti connessi, ThreadPool e array di lavoro, creazione e messa in esecuzione degli worker-threads, array di ibernazione), avviare il ciclo di ascolto (accettare nuove connessioni e nel caso si sia superato il numero massimo di connessioni (MAXCONNECTIONS), rifiutarle) e predisporre la chiusura del Server al soppraggiungere di uno dei segnali di terminazione(VEDI SEZIONE TERMINAZIONE). Compito supplementare da me affidato al listener-thread è quello di gestire l'estrazione di quei file descriptors ibernati in seguito all'operazione di *attesa attiva* di messaggi/files (VEDI SEZIONE MENO_R_OP).

Gli worker-threads hanno il compito di estrarre tasks dall'array di lavoro, leggere le richieste dei clients associati ed invocare le routines per soddisfarle.

Una routine, oltre a portare a termine l'operazione richiesta da un client, ha anche il compito di aggiornare le statistiche locali del worker-thread, deallocare il/i buffer/s dati inviato/i dal client, nel caso in cui il client si fosse disconnesso durante l'operazione, chiudere il socket relativo, altrimenti inviargli un messaggio di esito. Se durante la prestazione del servizio il client non si è disconnesso, il worker-thread ha il compito di reinserirlo nel *working set*, dei file descriptors attivi della Select, per consentirgli di richiedere nuove operazioni.

4 MENO_R _OP

Per ovviare alla problematica di scritture concorrenti sullo stesso socket, nella mia implementazione l'invio dei messaggi non è "istantaneo", ma le operazioni POSTTXT_OP, POSTTXTALL_OP e POSTFILE_OP corrispondono all'inserimento in History del messaggio/file. Ripercorrendo, poi, l'implementazione dell'invio di un file e dell'esplicita richiesta del file da parte del destinatario, allo stesso modo sono gli utenti a richiedere esplicitamente i propri messaggi, grazie all' operazioni di MENO_R_OP (da me introdotta per gestire "l'attesa attiva" di messaggi/files da parte di un client). [1]

La gestione dell'operazione MENO_R_OP è illustrata di seguito. Innanzitutto, per venire a conoscenza di quando un client sta facendo l'operazione "-R", ho implementato due funzioni distinte per la lettura di un messaggio: la "readServerMsg" usata dal Server per leggere le richieste degli utenti e la "readMsg" usata dal client in sole due occasioni: quando richiede la sua History e dopo averne ricevuto la dimensione, si sospende in attesa di leggerne il contenuto e quando fa *attesa attiva*, invocando il flag "-R", sospendendosi in attesa dell'invio di uno o più messaggi/files. La "readServerMsg" si occupa di leggere l'header e il data della richiesta di un client, mentre la "readMsg" legge l'header e il data di risposta del Server, ma antepone a queste letture l'invio di un messaggio con l' intestazione "MENO_R_OP". Nel caso in cui il client abbia richiesto l'operazione GETPREVMSGS_OP, nella routine che gestisce tale servizio, tale messaggio viene ignorato, mentre se il client non ha fatto GETPREVMSGS_OP questo messaggio consente al worker-thread di capire che il client si è sospeso in attesa di uno o più messsaggi/files. Questo comportamento del client si traduce in una vera e propria operazione, gestita nello switchcase delle operazioni con una routine apposita, che controlla se nella History dell'utente ci sono messaggi/files pendenti e glieli invia. Se la History non contiene messaggi/files pendenti o il numero non è ancora sufficiente a sbloccare l'utente, tale client rimane in attesa e viene "ibernato" nell'aray di ibernazione, per consentire al worker-thread di soddisfare altre richieste e dare tempo agli altri clients, eventualmente, di sbloccarlo. All'interno dell'array di ibernazione viene inserito il *file descriptor* del client, opportunamente modificato (sommandogli + FD_SETSIZE = 1024) e consentire ad un'estrazione successiva di capire che stava facendo la MENO_R_OP.

[1] Non è stato possibili implementare un invio simultano dei messaggi a causa di disincresie a me sconosciute con la gestione del flag "-R". A testimonianza di tale tentativo ho lasciato nel codice la routine *getPendingMessages* che controllava ad ogni richiesta di un client se fossero soppraggiunti messaggi (aggiunti nella sua history in caso affermativo) e glieli inviava prima di esaudire la richiesta.

E' compito del listener-thread, ad ogni ciclo di ascolto, estrarre (se ci sono) i *file descriptor ibernati* dall'array di ibernazione. Nel caso in cui l'array non sia vuoto, il listener-thread preleva il *file descriptor ibernato* da più tempo e lo mette all'interno dell'array di lavoro. Il worker-thread che lo estrarrà dalla coda di lavoro saprà che il client ha fatto l'operazione MENO_R_OP proprio dalla modifica del *file descriptor* apportata in precedenza, lo riporterà al valore originale, verificherà se nel frattempo gli sono giunti messaggi/files sufficienti, nel caso affermativo gliegli invierà, altrimenti lo rimetterà in coda di ibernazione ed il ciclo si ripeterà.

5 Gestione della concorrenza

5.1 Concorrenza sulla HashTable

L'accesso alla HashTable avviene tramite un array di locks di dimensione finita (MAXCONNECTIONS per l'esattezza). L'indice "i", con $0 \leq$ i $<$ MAXCONNECTIONS, dell'array di locks da l'accesso al bucket[j] della HashTable, tale per cui $i = j \ mod \ size$, con $0 \leq$ j $<$ size = MAXCONNECTIONS * BASE_HASHTABLE=32. Ciò consente l'accesso alla HashTable per sezioni (massimo MAXCONNECTIONS accessi concorrenti alla volta). Una volta che si e' entrati in possesso della mutua esclusione su un bucket, è possibile andare a scorrere la lista degli utenti che collidono su quel bucket, modificare/eliminare tali utenti e le loro Histories senza ulteriori mutue esclusioni, dato che ad ogni bucket, per quanto detto sopra, puo' accedere un thread per volta.

5.2 Concorrenza sulla UsersQueue

Una volta acquisita la mutua esclusione su un bucket della HashTable è possibile modificare tutti i nodi della lista di trabocco relativa, dato che c'è un solo thread alla volta che ne ha l'accesso.

5.3 Concorrenza sulla HistoryQueue

Una volta acquisita la mutua esclusione su un bucket della HashTable è possibile modificare tutte le Histories della lista di trabocco relativa, dato che c'è un solo thread alla volta che ne ha l'accesso (la modifica di tutte le Histories della lista di trabocco relativa viene fatta solo quando un utente richiede l'operazione POSTTXTALL_OP, per le operazioni POSTTXT_OP e POSTFILE_OP si modifca una sola History nella lista di trabocco).

5.4 Concorrenza sulla OnlineQueue

Siccome la Select assegna ad ogni client un *file descriptor* univoco, le operazioni fatte sulla coda dei connessi sono effettuate senza mutua esclusione esplicita (la mutua esclusione è data appunto dall'accesso univoco tramite file descriptor).

5.5 Concorrenza sulla TaskQueue

La coda di lavoro dalla quale gli worker-threads prelevano le richieste dei clients da soddisfare viene acceduta tramite mutua esclusione esplicita (lockTa-

skQueue) e l'ausilio di due variabili di condizione (emptyTaskQueueCond e fullTaskQueueCond).

5.6 Concorrenza sulla HibernationQueue

La coda di ibernazione nella quale gli worker-threads inseriscono quei clients che stanno facendo *attesa attiva* e dalla quale il listener-thread preleva tali clients per inserirli nella coda di lavoro e permettere agli workers di verificare il soppraggiungere di messaggi/files per sbloccarli viene acceduta tramite mutua esclusione esplicita (lockHibernationQueue) ad una variabile di condizione (fullHibernationQueueCond).

5.7 Concorrenza su threads

L'array che memorizza gli ID degli worker-threads viene acceduto senza mutua esclusione, dato che viene usato solo per registrare gli ID degli workers all'atto della loro creazione.

5.8 Concorrenza su arguments

L'array che memorizza gli argomenti da passare agli worker-threads viene acceduto senza mutua esclusione, dato che ogni worker accede al suo slot, indicizzato dallo stesso indice che racchiude il suo ID nell'array "threads" (tale informazione gli perviene all'atto della sua creazione).

5.9 Concorrenza su threadsStats

L'array che memorizza le statistiche locali degli worker-threads viene acceduto senza mutua esclusione, dato che ogni worker accede al suo slot, indicizzato dallo stesso indice che racchiude il suo ID nell'array "threads" (tale informazione gli perviene nell'atto della sua creazione).

5.10 Concorrenza sull'insieme dei descrittori

E' compito di un worker-thread chiudere il socket di un client che si è disconnesso oppure che ha manifestato problematiche di letture/scritture sul socket. A tale scopo, prima di inserire un *file descriptor* nell'array di lavoro, il listener-thread lo toglie dall'insieme dei descrittori ed è compito del worker che prenderà in carico la sua richiesta, reinserirlo oppure in caso di disconnessione/probemi di connessione, non farlo. L'accesso all'insieme dei descrittori da parte di più threads concorrenti ha reso, pertanto, necessario l'utilizzo di una mutua esclusione esplicita (mtx_set).

5.11 Concorrenza sui files

La scrittura e la lettura concorrente di un medesimo file ha richiesto l'accesso in mutua esclusione per portare a termine tali operazioni da parte di threads diversi (mtx_file).

6 Gestione segnali, statistiche e terminazione

Il listener thread al suo avvio si occupa di invocare la funzione "signals_registration". La funzione "signals_registration" ha il compito di:

- inizializzare le variabili globali "sigStats" e "sigStats" con le quali veniamo a conoscenza del sopraggiungere di uno dei segnali di terminazione (SIGINT, SIGQUIT, SIGTERM) o di stampa su file delle statistiche (SIGUSR1);
- registrare i gestori "termination_handler e "statistics_handler", che si occupano di settare tali variabili all'arrivo dei relativi segnali (a tal scopo le variabili sono state dichiarate "volatile sig_atomic_t" per renderle *signal safe*). Si registra anche un gestore per ignorare il SIGPIPE (ciò evita di far terminare il *welcome socket* se non ci sono client-sockets attivi in un dato momento).

La raccolta delle statistiche è affidata direttamente agli worker-threads, che raccolgono le informazioni sull'esito delle operazioni che hanno gestito. Ogni worker-thread, infatti, ha accesso ad uno slot dell'array delle statistiche (l'indicizzazione avviene tramite lo stesso indice che nell'array degli ID degli worker-thread identifica il worker-thread), nel quale esso memorizza le sue statistiche locali. Sarà, poi, compito del listener-thread, sommare le statistiche locali di ogni worker-thread ed ottenere così le statistiche globali della Chat. La stampa delle statistiche viene gestita esplicitamente, all'interno del ciclo di ascolto del listener-thread, tramite la funzione "handle_stats", che si occupa di stampare le statistiche globali su file, resettando la variabile "sigStats" modificata dal gestore del segnale SIGUSR1. [2]

All'arrivo di uno dei segnali di terminazione si predispone, invece, l'uscita dal ciclo di ascolto del listener-thread e la chiusura del Threadpool, tramite *Graceful shutdown* (non si accettano più nuove richieste, ma si soddisfano quelle presenti nella coda di lavoro e di ibernazione, attendendo infine la terminazione di tutti gli worker-threads). Dopo di che si deallocano le strutture dati utilizzate dal ThreadPool e dalla HashTable, si chiude il *welcome socket* e si fa terminare il listener-thread. Il "chatty main" si occuperà, invece, di deallocare le variabili globali che sono state allocate dinamicamente all'atto del parsing del file di configurazione.

[2] Il listener-thread è l'unico a mantenere il conteggio degli utenti connessi all'interno della variabile globale *numConnectedUsers* (numConnectedUsers++, dopo "accept"). Gli worker-threads, invece, si occupano di sommare gli utenti disconessi nelle loro statistiche locali al posto degli utenti connessi (threadsStats[index].nonline++) e di conseguenza per scoprire quanti utenti sono connessi in un dato momento, bisogna sottrare alla variabile gloable *numConnectedUsers* la somma degli utenti disconessi dagli workers.

7 Testing

Il progetto è stato sviluppato sulla seguente piattaforma:

1. Ubuntu 18.04.2 LTS

E' stato, invece, testatato sulle seguenti piattaforme:

1. Ubuntu 18.04.2 LTS
2. Ubuntu 18.04.5 LTS
3. Linux 4.9.0-8-amd64 1 SMP Debian 4.9.144-3.1 (2019-02-19) x86_64 GNU/-Linux

8 Note

1. Il progetto richiede l'uso della libreria Math.h per il calcolo della funzione hash. Di conseguenza ho modificato il MakeFile per includere -lm. Inoltre ho aggiunto il cflag "-D_POSIX_C_SOURCE=200809L " perchè alcune piattaforme non erano in grado di riconoscere la struct sockaddr_un;
2. Non sono sicura della buona riuscita della documentazione Doxygen da me redatta, vistochè ci sono wornings nella costruzione della medesima. Posizionandosi, però, nella directory del progetto, digitando il commando **doxygen DoxyFile** in shell, si crea la cartella **html**, all'interno della quale si trova il file **index.html** per visualizzare la documentazione sintatticamente corretta del progetto;
3. Per l'implementazione da me redatta del progetto, è possibile eliminare il flag "-t" delle operazione del test5 (flag che rapprensenta i millisecondi che intercorrono tra la gestione di due comandi consecutivi) senza problematiche nel portare a termine l'esecuzione del medesimo.