Chatterbox meets SQLite

Marco Costa
<mcsx97@gmail.com>

30 agosto 2018

Indice

Parte I
Scelte progettuali

1 Database

> «SQLite is a self-contained, high-reliability, embedded, full-featured, public-domain, SQL database engine. SQLite is the most used database engine in the world.» da `sqlite.org`

A causa della natura del progetto (semplificando: il server di una chat) è stato scelto di assegnare la memoria ad SQLite, una libreria software che implementa un DBMS SQL di tipo ACID incorporabile all'interno di applicazioni. Utilizzato da applicazioni come Firefox, Skype, Dropbox e molte altre, fornisce in modo semplice, consistente e veloce la disponibilità di un database stabile e performante. Alcune delle funzionalità messe a disposizione (da `sqlite.org/features.html`):

- Transactions are atomic, consistent, isolated, and durable (ACID) even after system crashes and power failures.
- Zero-configuration - no setup or administration needed.
- Full-featured SQL implementation with advanced capabilities like partial indexes, indexes on expressions, JSON, and common table expressions.
- A complete database is stored in a single cross-platform disk file.
- Supports terabyte-sized databases and gigabyte-sized strings and blobs.
- Small code footprint: less than 500KiB fully configured or much less with optional features omitted.
- Simple, easy to use API.
- **35% faster than direct filesystem I/O**
- Well-commented source code with 100% branch test coverage.
- **Available as a single ANSI-C source-code file that is easy to compile and hence is easy to add into a larger project.**
- Cross-platform: Android, *BSD, iOS, Linux, Mac, Solaris, VxWorks, and Windows (Win32, WinCE, WinRT) are supported out of the box. Easy to port to other systems.
- No memory-leaks.

La semplicità di utilizzo dell'interfaccia ha permesso la scrittura di un codice scalabile e consistente per l'interfacciamento al database. Infatti, ad esempio, l'aggiunta dei gruppi e di tutte le operazioni annesse ha richiesto semplici operazioni di reingegnerizzazione sullo schema del database [fig. 1] (e.g. l'aggiunta della voce *creator* nella tabella _Chat ad indicare l'utente con permesso di eliminazione) e sul flusso del codice. È stato sufficiente strutturare la funzione *manage_postmessage* (in *queries.c)* affinché gestisse il destinatario del messaggio come utente o gruppo.

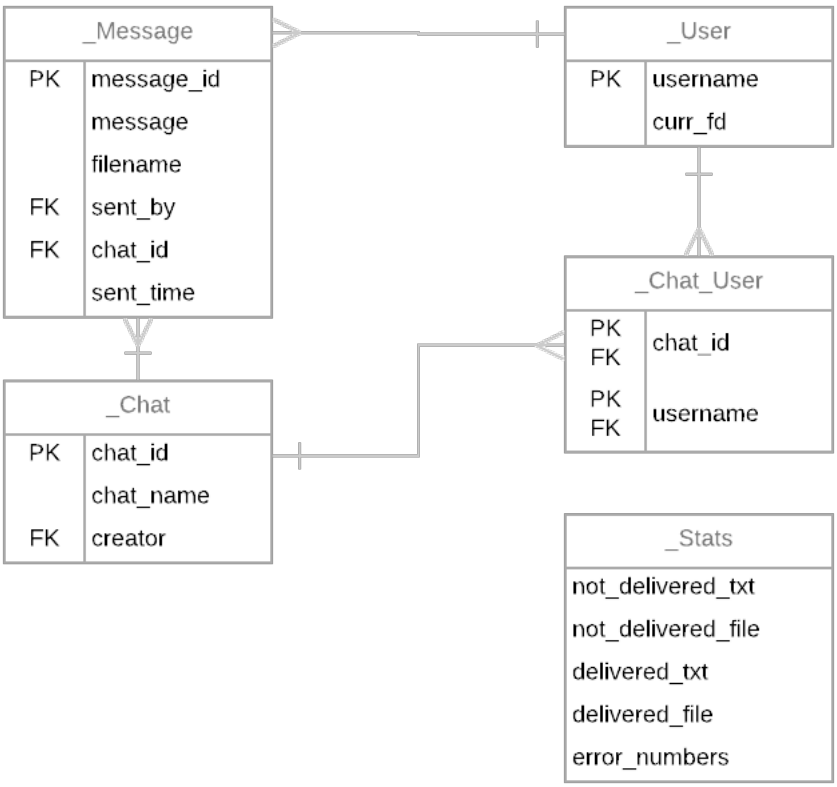

Figura 1: schema del database

NOTA: il campo *chat_name* indica: se NULL una chat tra due utenti, altrimenti un gruppo ed il suo nome.

1.1 Database API

- Apertura del database[1]:

```
int sqlite3_open_v2(
  const char *filename,   /* Nome del database */
  sqlite3 **ppDb,         /* OUT: SQLite db handler */
  int flags,              /* Flags */
  const char *zVfs        /* Nome del modulo VFS da usare */ );
```

- Esecuzione di una query:

```
int sqlite3_exec(
  sqlite3*,                               /* Handler del database */
  const char *sql,                        /* Query SQL */
  int (*callback)(void*,int,char**,char**),  /* Funzione di callback */
```

[1]il modulo VFS fornisce opzioni diverse per la portabilità tra S.O. diversi, è stata lasciata l'opzione di default

```
  void *,                          /* 1° argomento del callback */
  char **errmsg                    /* Eventuale messaggio di errore */
);
```

- Chiusura del database:

```
int sqlite3_close(sqlite3*);
```

2 Server

La parte server, invece, è stata implementata mediante un server sequenziale multi-client tramite l'utilizzo della funzione *select* per il monitoraggio di più descrittori contemporaneamente attivi.

Il thread principale si occupa unicamente di monitorare i descrittori, leggere i messaggi ed inserirli in coda. Alcuni controlli sulla correttezza della richiesta possono essere eseguiti anche dal server (e.g. messaggio troppo grande) tuttavia quest'ultimo non risponde mai direttamente all'utente ma inserisce in coda un'operazione «speciale» la quale verrà gestita da uno dei thread slaves.

2.1 Terminazione del server

Uno speciale descrittore (si veda «*man eventfd*») è utilizzato per la comunicazione tra il thread «signal handler» e il thread server. Il monitoraggio di questo descrittore «fasullo» permette, successivamente ad una scrittura su di esso, il risveglio del server dalla funzione *select()* e l'inizio delle routine di chiusura e pulizia.

3 Gestione della coda

Per implementare la coda di messaggi è stata utilizzata la classe di macro *BSD: STAILQ (Singly-linked TAIL Queue, si veda «man queue»)* la quale fornisce le interfacce per la creazione e manipolazione di un oggetto di tipo *tail queue* (pila con inserimento in coda) mediante le quali sono state implementate le interfacce *queue_push() e queue_pop()* per le operazioni sincronizzate di push e pop sulla coda. È garantito costo in tempo O(1) per entrambe le operazioni.

3.1 Terminazione della coda

La terminazione della coda è invocata dalla routine di pulizia del server. La quale, dopo aver svegliato i possibili slaves in attesa di messaggi sulla coda e settata la variabile di terminazione condivisa tra gli slaves, pulisce i possibili messaggi ancora presenti e distrugge le variabili di sincronizzazione.

È stata predisposta un'operazione speciale, la quale, restituita dalla funzione *queue_pop()* in seguito alla richiesta di terminazione eseguita dal thread server, indica al singolo thread slave di eseguire la propria pulizia e chiusura.

4 Gestione della sincronizzazione tra gli slaves

Per la sincronizzazione interna alla pool di thread slave sono state utilizzate tre strutture principali. La prima garantisce che più thread non possano scrivere sullo stesso canale di comunicazione contemporaneamente.

La seconda, chiamata *critic_ zone*, garantisce la «consistenza» delle operazioni. Si immagini la seguente situazione:

> Ciccio invia al server le seguenti richieste: connessione, invio di un messaggio, disconnessione. Degli n thread slave, slave_1 estrae dalla coda la richiesta di connessione, slave_2 la richiesta di invio messaggio e slave_3 la richiesta di disconnessione. Slave_1 esegue la richiesta, dopodiché lo scheduler di sistema mette in pausa slave_2 prima di aver servito la richiesta mentre consente l'esecuzione di slave_3.

La struttura garantisce che non possano essere eseguite operazioni «incompatibili per precedenza» quali la disconnessione prima di altre operazioni e operazioni generiche prima della registrazione.

Nel caso si verifichi una situazione come quella descritta precedentemente il thread «in anticipo» riposizionerà la richiesta in fondo alla coda e ne estrarrà una nuova.

La terza, invece, garantisce l'accesso esclusivo al database in caso di scrittura. Infatti, SQLite non permette sempre l'esecuzione parallela di alcuni tipi di query «conflittuali» di lettura e scrittura sulla stessa tabella. Perciò, non potendo utilizzare meccanismi di temporizzazione (previsti dalla documentazione di SQLite per la gestione di questo tipo di errori) è stata implementata una politica di sincronizzazione «pseudo-parallela», la quale permette l'esecuzione parallela di query in lettura e la singola esecuzione di query in scrittura.

4.1 Terminazione e chiusura delle strutture di sincronizzazione

Il server, dopo aver chiuso e pulito la coda di messaggi si assicura che nessuno dei thread slaves possa trovarsi in una condizione di attesa su una di queste tre strutture e, come nel caso della coda di messaggi, imposta una speciale variabile utilizzata per indicare ad ogni thread slave ancora in attesa di eseguire la propria routine di pulizia. Al termine di questa operazione viene liberata la memoria di queste tre strutture e, il thread server, attende la terminazione della pool di thread prima di poter eseguire la propria terminazione.

5 Gestione dei segnali

Un thread specifico è stato adibito alla gestione dei segnali. Esso, rimane in attesa (mediante funzione *sigwait()*) dei segnali previsti dall'assegnamento e, alla ricezione di uno di essi, esegue l'operazione specifica.

Gli altri thread semplicemente ignorano la ricezione dei segnali.

6 Difficoltà incontrate

Per quanto la scelta di utilizzare un DBMS è, a mio modesto avviso, una scelta vincente per questo genere di situazioni, non è stata un'implementazione semplice. Studiare e comprendere (sommariamente, il necessario per poterlo utilizzare) il funzionamento di SQLite è, grazie all'importante documentazione presente online, abbastanza inuitivo. Le vere prolematiche sono state altre, la difficoltosa gestione delle stringhe di testo del linguaggio C (cuore dell'esecuzione delle query) e, soprattutto, l'aver dovuto adattare l'accesso al database alla mole di query eseguite durante lo Stress Test. Infatti, per quanto siano state abilitate tutte le modalità di concorrenza sul database, quest'ultimo è sì concorrente ma non a ritmi così alti. Il suo scopo primario è più quello di un database interno all'applicazione che altro.

Tuttavia, credo che il risultato ottenuto sia soddisfacente (sulla mia macchina, lo stress test non ha mai superato il 3% di uso della CPU e i 20MB di memoria fisica), il quale mi è costato (probabilmente) almeno una dozzina di giorni di lavoro in più rispetto all'utilizzare la struttura messa a disposizione.

Parte II
Funzionalità del server

Cosa può fare:

- Conservare il database ad ogni esecuzione (rimuovendo l'opzione di compilazione *-DRESET_DB*[2])
- È possibile cambiare il percorso del database modificando il campo *DB_NAME* nel MakeFile. Rimuovendo questa opzione verrà utilizzato il percorso di default.
- Serializzazione automatica delle query sul database mediante l'opzione *-DSQLITE_THREADSAFE=1* (funzionalità di SQLite)
- Fornire messaggi (in chat singole e gruppi) inviati da utenti non più registrati/membri dei gruppi
- È possibile eseguire il server senza l'utilizzo del file di configurazione ma utilizzando i parametri predefiniti (si veda *config.h*)
- È possibile inserire un file di configurazione «parziale», le voci mancanti saranno sostituite dai parametri di default
- File diversi ma con lo stesso *filename* inviati da utenti diversi non vengono sovrascritti

[2] ovviamente l'esecuzione consecutiva dei test non può funzionare

- È possibile attivare in fase di compilazione la modalità WAL *(Write-Ahead Logging)*[3] per il database tramite l'opzione *-DWAL_MODE* la quale permette:
  - Commutazione automatica su disco dopo ogni query
  - Maggiore sequenzialità delle operazioni su disco
  - Significativamente più veloce
  - Fornisce maggiore concorrenza in quanto permette letture e scritture contemporanee sul database
- Gestire la parte opzionale relativa ai gruppi (creazione/aggiunta/rimozione/eliminazione[4])
- È possibile mantenere aggiornati i file sorgente di SQLite scaricando, quando possibile, una nuova release dal sito web, sostituire i file *sqlite3.h* e *sqlite3.c* e ricompilare.
- È possibile scaricare ed installare il programma *sqlite3* per poter aprire il database ed eseguire interrogazioni SQL sul database da riga di comando. Ad esempio, per ottenere i nomi di tutti i gruppi registrati successivamente all'esecuzione di *testgroups.sh*:

```
[mc@mc-inspiron ~] $ sqlite3
SQLite version 3.22.0 2018-01-22 18:45:57
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open␣FILENAME" to reopen on a persistent database.
sqlite> .open /tmp/chatterboxdb
sqlite> SELECT chat_name FROM _Chat WHERE chat_name IS NOT NULL;
gruppo1
gruppo2
sqlite> .exit
[mc@mc-inspiron ~] $
```

Cosa non può fare:

- Non è stata implementata una routine di pulizia per i messaggi non aventi più destinatari disponibili
- Accettare un file di configurazione sintatticamente non corretto o con parametri non previsti
- Salvare lo stato interno del database in caso di interruzione imprevista del programma/cali di tensione[5] (a meno che non sia abilitata la modalità WAL)

[3] **NOTA**: questa opzione è stata poco testata in quanto non è ancora ben supportata dal file system presente sulla mia macchina di sviluppo. Tuttavia sembra funzionare bene con le opzioni di virtualizzazione di VirtualBox.

Non è garantito il superamento dei test con questa modalità (soprattutto dello Stress Test).

[4] per l'eliminazione di un gruppo da parte del suo creatore è stata aggiunta l'operazione *UNREGISTER_GROUP*.

[5] a causa della mole di query da eseguire durante lo Stress Test non è possibile impostare il commit automatico su disco (a causa delle limitate velocità di I/O di quest'ultimo) dopo ogni query e devono perciò essere gestite con commit sporadici

- Il risultato di messaggi non inviati mediante il programma *client* può essere impredicibile
- Eseguire routine di pulizia nel caso di chiusura per errori imprevisti (e.g chiusura del programma a causa di problemi con la *select*)

È possibile trovare all'interno della cartella ***doxygen*** l'html generato a partire dal codice sorgente.

Parte III
Strumenti di sviluppo e debug

- Macchina di sviluppo: *Dell Inspiron 7370*
- Sistema operativo: Fedora 28
- Ambiente di sviluppo: Visual Studio Code
- Ambiente di debug: GDB con plugin di integrazione per VS Code
- Altri strumenti: oltre all'utilizzo di Valgrind sono state utilizzate le librerie *Sanitizers* di Google (`https://github.com/google/sanitizers`), in particolare:
  - *AddressSanitizer* per la rilevazione di errori quali:
    * use after free
    * heap buffer overflow
    * stack buffer overflow
    * memory leaks
    * ecc.
  - *ThreadSanitizer* per la rilevazione di condizioni di corsa
  - *MemorySanitizer* per la rilevazione di uso di memoria non inizializzata

Parte IV
Esecuzione dei test

IMPORTANTE: poiché i file utilizzati come invio nei test (e.g libchatty.a, client.o, ...) sono molto più grandi dei valori limite presenti nei file di configurazione è stato necessario cambiarli. Con le configurazioni di default i test non vengono superati.

IMPORTANTE: Le opzioni di compilazione salvate nel MakeFile garantiscono il superamento dei test. L'opzione -DMAKE_TEST_HAPPY è mandatoria.

Non sono state riscontrate problematiche (oltre a quelle già descritte) con l'esecuzione dei test.