Laboratorio di Sistemi Operativi (277AA)
Progetto membox++

Relazione finale

Alessandro Antonelli
(matricola 507264, corso A)

Appello straordinario aprile 2017
A.A. 2015/2016

Indice

1 Come compilare ed eseguire il codice

Il codice deve essere compilato attraverso il target `all` del Makefile, ossia con il comando di shell `make all` . Prima della compilazione è possibile agire sulla variabile CFLAGS del makefile per modificare il comportamento del server: aggiungendo `-DNDEBUG` si disabiliteranno gli assert, con `-DSTAMPETEST` compariranno sullo standard error delle stampe di log che documenteranno il funzionamento interno del server, mentre con `-DDEBUG` compariranno sullo standard error i messaggi di errore nelle occasioni in cui funzioni e chiamate di sistema restituiscano valori diversi da quelli di attesi, prodotti dalle macro `CheckExit` , `CheckMenoUno` e `CheckNULL` definite in `tools.h` (NB: i messaggi non sono necessariamente sintomo di bug in quanto la situazione anomala viene gestita).

Una volta compilato, il server deve essere avviato usando il comando `membox -f membox.conf [-d dumpfile]` . Il codice è stato testato su Ubuntu 14.04 LTS 64 bit.

2 Strutturazione del codice

configurazione.c / .h Contiene prototipo e implementazione della procedura che legge e interpreta il file di configurazione, nonché la definizione del tipo di dato dove si memorizzano i campi letti dal file.

connections.c / .h Contiene i prototipi e le implementazioni delle funzioni utilizzate per la comunicazione tra server e client.

icl_hash.c / .h Implementazione della tabella hash (fornita dai docenti; sono state effettuate alcune modifiche, documentate nella sez. 6).

Makefile Contiene le regole per automatizzare la compilazione.

membox.c È il principale file sorgente del progetto. Contiene il `main` nonché le procedure con cui viene avviato ciascun thread.

message.h Contiene la definizione dei tipi di dato utilizzati per lo scambio dei messaggi tra client e server, e alcune procedure per compilare correttamente i campi delle strutture.

ops.h Contiene i codici delle operazioni richieste e delle risposte.

queue.c / .h Contiene i prototipi e l'implementazione di una semplice coda di elementi interi ad accesso FIFO, con la definizione del tipo di dato `queue_t` e delle operazioni per lavorarci.

stats.h Contiene la funzione per stampare le statistiche e il tipo di dato utilizzato per memorizzarle.

tools.h Contiene alcune macro di utilità generale (controllo dei valori restituiti dalle chiamate di sistema, lock e unlock dei mutex, stampe di test) e la definizione del tipo di dato booleano.

3 Protocollo client-server

Per aprire una connessione con il server, il client utilizza la funzione `openConnection` implementata in `connections.c`, alla quale va passato il path dello stesso socket specificato nel file di configurazione del server (campo `UnixPath`). Il socket rimane in vita per tutto il tempo di esecuzione del server, che rimane sempre in ascolto su di esso.

Stabilita una connessione, ha inizio un ciclo in cui il server rimane in attesa di un messaggio da parte del client (che contiene la richiesta), e successivamente il client rimane in attesa di un messaggio da parte del server (che contiene l'esito ed eventualmente la risposta). Il ciclo va avanti finché uno dei due non chiude la connessione.

Per lo scambio dei messaggi si è scelto di utilizzare strutture dati e funzioni simmetriche: il server, cioè, invia e riceve messaggi con le stesse modalità del client.

Strutture

I messaggi, sia di richiesta che di risposta, sono strutture del tipo `message_t` (definito in `message.h`), che contengono al loro interno un campo `hdr` per l'header e un campo `data` per il contenuto del messaggio. Per la corretta creazione di una variabile di questo tipo, vanno usate le procedure `setHeader` e `setData`.

L'header, di tipo `message_hdr_t`, è a sua volta una struttura che contiene due campi: il tipo di operazione `op` (che è uno dei codici enumerati in `ops.h`) e la chiave `key` dell'oggetto a cui è riferita. Il campo `key` è significativo solo per alcuni tipi di richieste (non lo è nel caso delle LOCK, UNLOCK e DUMP) e non è mai significativo nel caso delle risposte inviate dal server al client.

Il contenuto, di tipo `message_data_t`, contiene un campo `buf` per il dato e un campo `len` per la lunghezza in byte del file. Della parte relativa al contenuto si fa uso solo nei messaggi con cui il client richiede le operazioni di `PUT` e `UPDATE`, e nei messaggi con cui il server risponde alle `GET`; in tutti gli altri casi il campo `buf` è impostato a NULL e `len` a 0.

Procedure

Le funzioni per inviare e ricevere messaggi sono implementate in `connections.c`.

L'invio di un messaggio avviene in un solo colpo, invocando la funzione `sendRequest`, la quale effettua tre `write` consecutive (header, lunghezza file, dati del file); le ultime due scritture non avvengono se il messaggio è privo della parte relativa al contenuto.

La ricezione di un messaggio avviene invece in due tempi, tramite le funzioni `readHeader` (che effettua una singola `read` per l'header) e `readData` (che effettua due `read`, una per la lunghezza del file e una per i dati), da invocare tassativamente in questo ordine. Dal momento che non tutti i messaggi hanno la parte relativa al contenuto, dopo aver invocato la prima funzione, è sempre presente un controllo sul tipo di operazione indicato nell'header, in modo da invocare la `readData` solo se l'operazione in questione prevede di ricevere dei dati.

4 Strutture dati e algoritmi

Le strutture dati principali sono le seguenti:

hashTable È la tabella hash dove vengono memorizzati i file che costituiscono il repository (tipo `icl_hash_t`, definito in icl_hash.h).

coda La coda dove vengono inseriti i file descriptor delle connessioni in entrata, in attesa che un thread del pool le prelevi dalla coda ed esegua le sue richieste. L'accesso in mutua esclusione è garantito da `codaMutex` mentre la variabile di condizione `arrivatoClient` serve a segnalare l'evento in cui un nuovo client viene inserito in coda.

config Variabile che memorizza i dati della configurazione con cui è stato avviato il server (tipo `config_t` definito in configurazione.h)

stats Variabile in cui sono memorizzate le statistiche del server (tipo `struct statistics` definito in stats.h)

4.1 Tabella hash

Gli elementi della tabella hash sono strutture di tipo `icl_entry_t` con due campi `key` e `data`. L'implementazione generica fornita dai docenti non è stata modificata (entrambi sono puntatori di tipo `void*`); nel progetto, tuttavia,

per la chiave viene sempre utilizzato il tipo `membox_key_t` (definito in message.h, alias di `unsigned long`) e per il campo data il tipo `hashTableEntry` (definito in icl_hash.h).

Questo ultimo tipo è a sua volta una struttura (aggiunta dal sottoscritto in icl_hash.h), che possiede i seguenti campi:

data (tipo `char*`), che contiene il contenuto del file

len (tipo `unsigned int`), che contiene la lunghezza del file in byte

detentoreLock, codaRichiedentiLock e rilascioLockObj , campi necessari per l'implementazione dell'operazione LOCK_OBJ. Per il loro significato vedere la sez. 5.5.

4.2 Numero di buckets

Alla creazione della tabella hash, un parametro fondamentale è il numero di buckets, da cui dipende sia l'efficienza del server in tempo (velocità di accesso al repository) che quella in spazio (consumo di memoria).

La calibrazione del numero di buckets è stata guidata dai seguenti princìpi:

- Adattarsi al numero massimo di oggetti specificato nel file di configurazione
- Quando esso non presente: adattarsi all'indicazione, contenuta nelle specifiche, che il numero di oggetti possa essere anche dell'ordine di 10^4
- Il mantenimento, in entrambi i casi, di un load factor $= \frac{\text{oggetti}}{\text{buckets}}$ ottimale anche al caso pessimo (< 1 o ancora meglio $< 0,66$) o almeno ragionevole, per preservare le proprietà di efficienza algoritmica delle operazioni sulla hash table
- La necessità di avere un limite massimo al numero di buckets, costante e noto a tempo di compilazione

Per trovare un compromesso tra tutti i punti precedenti, è stata utilizzata una funzione che assegna i buckets con un comportamento dinamico, al variare di x numero massimo di oggetti:

$$numeroBuckets(x) = \begin{cases} \frac{x}{0{,}000368 \cdot x + 0{,}33} & \text{se } x \leq 2500 \\ 0,09376 \cdot x + 1766 & \text{se } 2500 < x \leq 15000 \\ \texttt{MAXBUCKETS} \text{ ossia } 3172 & \text{se } x > 15000 \end{cases}$$

Il numero max di oggetti x viene letto dal settaggio `StorageSize` del file di configurazione; se manca o è illimitato, al suo posto viene usata una stima $maxOggetti = \frac{\text{dimensione max dello storage in bytes}}{\text{dimensione media di un oggetto}} = \frac{\texttt{StorageByteSize}}{0{,}66 \cdot \texttt{MaxObjSize}}$.

Inizialmente, per un numero di oggetti basso, la funzione assegna molti buckets in modo tale da tenere basso il load factor; questa strategia ha lo scopo di mettere la tabella hash nella condizione di massima efficienza algoritmica, almeno in quei casi in cui questa si può raggiungere con un dispendio di memoria limitato (poche centinaia di buckets), visto anche che, dai test forniti dai docenti, il server sembra essere utilizzato spesso in queste condizioni. Inizialmente si hanno più buckets che file massimi, il che può sembrare un controsenso, ma è utile per minimizzare la probabilità di collisioni e comunque questo avviene solo per numero di file basso. I buckets crescono ovviamente meno del numero di elementi e da un certo punto in poi diventano meno numerosi degli elementi, con il load factor che passa da 0,33 (tre buckets per un elemento) fino ad 1,25 (2000 buckets per 2500 elementi).

Oltre i 2500 elementi il comportamento della funzione cambia e la crescita del numero di buckets si fa molto più lenta: visti i numeri non più bassi, si è scelto di contenere il consumo di memoria (anche se questo comporta un rapido allontanamento dall'efficienza algoritmica), mantenendo comunque una certa gradualità (ad un incremento degli elementi corrisponde comunque un incremento dei buckets, seppur minimo). Il load factor passa da 1,25 (2000 buckets per 2500 elementi) a circa 4,7 (3172 buckets per 15000 elementi); oltre questa soglia è stato considerato prioritario il risparmio di memoria e il server assegna sempre `MAXBUCKETS` = 3172 buckets, in ossequio al principio di avere un tetto massimo noto.

Da qui in poi il load factor cresce liberamente, arrivando fino a circa 25 nel caso limite di 80000 elementi (certamente lontano da quello ottimale ma considerato accettabile visto che il caso dovrebbe essere poco frequente e che 25, in confronto agli 80000 elementi, può comunque essere ritenuto $\mathcal{O}(1)$).

Quando non viene specificato nessun limite al numero di file (mancano o sono settati a infinito sia `StorageSize` che `StorageByteSize` e `MaxObjSize`), il server usa `DEFBUCKETS` = 1024 buckets. In alcuni casi, tra cui diversi test forniti dai docenti, il picco del numero di file è minore e in quei casi c'è certamente uno "spreco" di memoria, ma - in assenza di indicazioni - si è ritenuto di usare una cifra che garantisse una minima efficienza algoritmica anche nei casi estremi. Ad ogni modo si consiglia di inserire sempre l'indicazione del massimo numero di file (anche solo una stima), in quanto la sua assenza non permette al server di adattarsi alla situazione, con cattivi risultati sia in termini di consumo di memoria che in termini di efficienza delle operazioni.

5 Threads e loro interazione

Il server si compone dei seguenti thread:

- Il thread del main
- Un thread che accetta le connessioni (esegue la procedura `AccettaConnessioni`)
- Pool di più thread che espletano le richieste dei client (eseguono la procedura `ServiClient`)
- Un thread che gestisce i segnali e stampa le statistiche (esegue la procedura `attendiSegnali`)

5.1 Accettazione delle connessioni

La gestione delle connessioni in entrata provenienti dai client è implementata con l'interazione tra due thread, che seguono il modello produttore-consumatore comunicando per mezzo di una coda condivisa (la variabile globale `coda`) e una variabile di condizione (`arrivatoClient`, che segnala l'inserimento in coda di un client).

Un thread apposito (che esegue la procedura `AccettaConnessioni`) si occupa di eseguire la `accept` e di inserire il file descriptor della nuova connessione nella coda. Se viene raggiunto il limite `MaxConnections` il thread rifiuta la connessione, dopo aver risposto con codice OP_FAIL.

Dopo l'inserimento in coda, il thread effettua una `signal` per svegliare i thread del pool. Questi ultimi agiscono da *consumatori* e si occupano di estrarre le connessioni pendenti dalla coda condivisa per poi entrare nel ciclo *leggi richiesta-esegui richiesta-invia risposta*.

Non è stata prevista una corrispettiva variabile di condizione atta a segnalare l'evento in cui la coda passa dall'essere piena a disporre di almeno un posto libero: conseguentemente il thread `AccettaConnessioni` rimane in esecuzione senza mai sospendersi, anche quando la coda è piena. Questa scelta è stata obbligata dal fatto che, con un'implementazione del genere, la `openConnection` da parte del client fallirebbe portando alla terminazione del client e al conseguente fallimento dei test (in particolare del test n. 4: alla riga 88 di `testsize.sh` ci si aspetta che, al raggiungimento di `MaxConnections`, il server rimanga in ascolto e risponda con OP_FAIL).

L'assenza della variabile di condizione comunque non porta a uno spreco eccessivo della potenza di calcolo, perché il thread passa la maggior parte del tempo sospeso nella `accept` e solo all'arrivo di una connessione esegue del codice (il minimo indispensabile per inviare il messaggio di fallimento).

5.2 Mutua esclusione della tabella hash

Tra i problemi di sincronizzazione, la questione di maggiore importanza è quella relativa all'accesso della tabella hash da parte dei thread del pool.

All'interno della struttura dati, i campi `nbuckets`, `hash_function` e `hash_key_compare` vengono acceduti da più thread ma non richiedono la mutua esclusione perché l'accesso è in sola lettura e i campi vengono scritti solo una volta, all'atto della creazione della hash table (quando c'è un solo thread in esecuzione). L'unico ad avere problemi invece è `nentries`, perché questo numero cambia durante l'esecuzione del server e la variabile è acceduta sia in lettura che scrittura; allo scopo è stata apportata una modifica a `icl_hash.c`, introducendo un mutex `nentriesMutex` di cui viene acquisita la lock prima di ogni accesso alla variabile.

Ma il problema maggiore è quello dell'accesso agli oggetti, cioè all'array dei bucket. Infatti è necessario evitare che più thread accedano contemporaneamente allo stesso oggetto; tuttavia non è possibile né utilizzare un solo mutex per l'intera struttura, con l'accesso di un solo thread per volta (soluzione che azzererebbe il parallelismo e rederebbe inutile avere più thread) né utilizzare un numero troppo elevato di mutex (soluzione che renderebbe eccessivo l'overhead e l'utilizzo di memoria). Pertanto anche qui si è dovuto trovare un compromesso e si è deciso di calcolare il numero dei mutex con la seguente funzione, al variare di x numero massimo di oggetti:

$$numeroMutex(x) = \begin{cases} x & \text{se } x \leq 10 \\ 10 + \ln^2(x-9) & \text{se } 10 < x \leq 45000 \\ \texttt{MAXMUTEX} \text{ ossia } 125 & \text{se } x > 45000 \end{cases}$$

Fino a quando gli oggetti massimi sono dieci, la funzione dà la priorità al parallelismo e assegna un mutex per ogni potenziale oggetto. Oltre questa soglia, la crescita è più lenta e ai dieci mutex se ne aggiungono altri con un fattore additivo dell'ordine di $\log^2$: questo garantisce che all'inizio la crescita sia aprezzabile, ma che questa diventi progressivamente più lenta con l'aumentare del numero max di oggetti (caratteristiche che garantiscono un buon livello di parallelismo quando gli oggetti sono "pochi" e un overhead limitato quando sono "tanti"). La funzione infatti passa da assegnare 10 mutex per 10 oggetti max ad assegnare 68 mutex per 2000 oggetti max, fino ad arrivare a 125 mutex per 45000 oggetti max. Oltre i 45000 oggetti max il numero di mutex smette di crescere e rimane fisso a 125 (`MAXMUTEX`).

Se `StorageSize` non è specificato, il numero massimo di oggetti viene stimato come già spiegato nel precedente paragrafo sul numero di buckets. Se mancano anche `StorageByteSize` e `MaxObjSize`, il numero di mutex è

`DEFMUTEX` cioè 40. I mutex, in ogni caso, sono memorizzati nell'array globale `hashMutex`.

Mappatura tra oggetti e mutex

Il problema successivo è stato quello di dividere gli oggetti del repository tra i mutex disponibili. La decisione presa a tale proposito è stata quella di effettuare una partizione non degli oggetti ma dei bucket; si è scelto cioè di usare sempre uno stesso mutex per tutti gli oggetti ricadenti in un certo bucket.

La decisione è stata presa per via del fatto che partizioni alternative, ad esempio basate sulla chiave dell'oggetto, avrebbero potuto portare a due oggetti consecutivi nella lista di trabocco gestiti da mutex diversi: questa circostanza avrebbe consentito potenziali accessi in violazione della mutua esclusione (ad esempio nel caso di operazioni come la `insert` che scandiscono l'intera lista di trabocco o la `delete` che non solo eliminano l'oggetto ma modificano anche i campi `next`/`prev` degli oggetti che lo precedono/seguono nella lista). La scelta adottata, inoltre, dovrebbe portare a una distribuzione abbastanza equa degli oggetti tra i mutex, in quanto qualsiasi buona funzione hash distribuisce gli oggetti con una probabilità uniforme tra i bucket, e quindi anche tra i mutex.

Visto che i mutex sono sempre meno dei bucket, ciascuno di essi deve gestirne più di uno solo. Se n sono i mutex totali, la mappatura adottata è la seguente:

```
bucket 1   -> mutex 1
bucket 2   -> mutex 2
  ...
bucket n   -> mutex n

bucket n+1 -> mutex 1
bucket n+2 -> mutex 2
ecc.
```

In pratica, per accedere all'oggetto di chiave k, è necessario calcolare il bucket b in cui ricade ($b = hashfunction(k) \bmod bucketTotali$) e poi acquisire la lock del mutex che nell'array `hashMutex` ha indice $i = b \bmod n$. Tutte queste operazioni sono semplificate dall'utilizzo delle macro `LockHash(key)` e `UnlockHash(key)`, definite in `tools.h`.

5.3 Segnali

Per la gestione dei segnali la scelta è stata di non usare gli handler ma di avere un thread dedicato (`attendiSegnali`) che rimane in ascolto di essi ciclando sulla funzione `sigwait`. La soluzione è stata scelta perché permette di accedere senza problemi alle variabili globali e di invocare funzioni non signal-safe durante la loro gestione.

All'avvio del server (nelle prime righe del `main`) tutti i segnali vengono mascherati, per evitare interferenze finché la loro gestione non sarà completa. Successivamente viene settato di ignorare `SIGSEGV` e `SIGPIPE`, per evitare di essere terminato da una scrittura su un socket chiuso.

A questo punto la maschera iniziale viene rimossa, ma vengono mascherati i segnali che saranno gestiti dal thread apposito: `SIGINT`, `SIGTERM`, `SIGQUIT`, `SIGUSR1` e `SIGUSR2`. Più avanti avviene lo spawn di tutti i thread (compreso quello che gestisce i segnali); in tal modo tutti quanti ereditano la maschera impostata dal main.

5.4 Terminazione

La terminazione del server "dolce" (`SIGUSR2`) e quella "immediata" (`SIGINT`, `SIGTERM` e `SIGQUIT`) sono gestite per mezzo delle rispettive variabili globali booleane `TermDolce` e `TermImmediata` (protette da un unico mutex `TermMutex`). Alla ricezione del rispettivo segnale, il thread `attendiSegnali` setta la variabile a `TRUE` e termina dopo aver risvegliato tutti i thread del pool (con una `broadcast`).

I thread del pool inattivi, risvegliati dalla broadcast, controllano le variabili e terminano. I thread del pool attivi controllano `TermImmediata` frequentemente (tra una richiesta e l'altra) e `TermDolce` meno frequentemente (tra una connessione e l'altra); in entrambi i casi, quando la variabile è TRUE, la connessione viene chiusa e il thread termina.

Dopo aver fatto la `join` di tutti i thread del pool, il `main` uccide il thread che accetta le connessioni con una `cancel`. Questa soluzione è certamente poco delicata, ma si rende necessaria perché il thread potrebbe essere bloccato in una `accept`: in quella situazione l'esecuzione non passerebbe mai alla riga in cui si controlla il valore delle due variabili globali. Dopo la stampa delle statistiche e le opportune deallocazioni, anche il thread del main termina.

5.5 Lock oggetto

La lock dei singoli oggetti è implementata attraverso l'aggiunta dei seguenti campi alla struttura che rappresenta le entry della tabella hash (tipo

`hashTableEntry`, definito in icl_hash.h):

detentoreLock Intero che indica se l'oggetto è in stato di lock ed eventualmente quale client la detiene. Vale -1 se l'oggetto è unlocked, mentre vale > 0 se l'oggetto è locked (in tal caso il valore coincide con il file descriptor usato per comunicare con il client che detiene la lock).

codaRichiedentiLock Eventuale puntatore alla coda dei client che sono in attesa di acquisire la lock sull'oggetto (ma ancora non la detengono).

rilascioLockObj Eventuale puntatore alla variabile di condizione usata per segnalare il rilascio della lock ai client in coda.

Alla creazione di un nuovo oggetto il primo campo vale -1 e gli altri due sono NULL. Quando un client acquisisce la lock viene modificato solo il primo campo; se ulteriori client tentano di acquisirla, allocano la coda e la variabile di condizione (lasciando i puntatori nei rispettivi campi) e si sospendono.

La unlock di un oggetto (procedura `DoUnlock`) rimette `detentoreLock` a -1 nel caso in cui non ci siano altri client in attesa, mentre se ce ne sono estrae il primo dalla coda e gli assegna la lock, modificando il campo detentore, per poi eseguire una `broadcast` sulla variabile di condizione.

Tutti i thread a servizio dei client in coda vengono risvegliati e controllano se il segnale era effettivamente rivolto a loro (testando se `detentoreLock == file descriptor mia connessione`); così rimane attivo solo quello che ha effettivamente ricevuto la lock, che può rispondere al client con un messaggio di successo.

Unlock in caso di disconnessione

Per evitare situazioni di deadlock, è stato necessario sviluppare dei meccanismi che effettuassero automaticamente la unlock degli oggetti, nel caso in cui il client si disconnetta senza farlo.

A tale scopo, ogni thread del pool dispone di una semplice struttura per tenere conto di quali e quanti oggetti sono stati posti in lock dal client attuale: si tratta dell'array `oggettiLocked` (che ad ogni istante contiene le chiavi degli oggetti su cui si è fatta la lock ma non ancora la unlock) e della variabile `numOggettiLocked` (che indica il numero di oggetti nell'array). Quando il client si disconnette, viene fatta d'ufficio la unlock degli oggetti ancora presenti nell'array.

Update di un oggetto locked

Un caso a cui si è dovuto prestare attenzione è quello in cui il detentore della lock esegue una UPDATE, invalidando il puntatore all'elemento posseduto dai client in coda. A tale scopo, ogni volta che si viene risvegliati dall'attesa sulla variabile di condizione, il puntatore all'elemento viene sempre sovrascritto invocando la `icl_hash_find`.

Rimozione di un oggetto locked

Più complessa è stata la gestione di un altro caso particolare, quello in cui il detentore della lock esegue una REMOVE dell'oggetto. Infatti, nel caso in cui ci siano altri client in attesa di acquisire la lock, la rimozione diretta dell'oggetto provocherebbe gravi problemi (con la liberazione della memoria, e quindi della variabile di condizione, diventerebbe impossibile risvegliare i thread sospesi, che andrebbero in stallo).

La situazione è stata risolta introducendo un valore speciale della variabile `detentoreLock`, che viene impostata a -2 nel caso in cui si debba effettuare una REMOVE ma `codaRichiedentiLock != NULL`. Questo valore speciale indica che l'oggetto è "in fase di eliminazione" pertanto, pur essendo presente, viene considerato a tutti gli effetti come oggetto inesistente (non esiste agli occhi del client e non è conteggiato nelle statistiche).

Dopo aver cambiato il valore della variabile, il thread detentore esegue una `broadcast` e risponde al client che la rimozione è avvenuta con successo. I thread in coda, risvegliati, trovano il valore `-2` e si rimuovono dalla lista di attesa, rispondendo al client con codice di errore `OP_LOCK_OBJ_NONE`. L'ultimo thread a uscire dalla coda si occupa di eseguire la rimozione vera e propria dell'oggetto.

Come detto sopra, dal momento in cui il client richiede la REMOVE di un oggetto locked al momento in cui l'ultimo thread esegue la rimozione "vera", l'oggetto viene considerato come inesistente. Questo lasso di tempo è molto breve; tuttavia, per quanto remota, esiste la possibilità che in quello spazio di tempo un altro client richieda la PUT di un oggetto con quella stessa chiave. La scelta, in questo caso, è stata quella di mantenere un comportamento coerente, senza negare la richiesta ma piuttosto ritardando la sua esecuzione fino all'avvenuta cancellazione materiale del vecchio oggetto; l'attesa di questa condizione è implementata con una `sleep` (membox.c, riga 397), sicuramente un cattivo meccanismo di sincronizzazione, che però è stato preferito per la maggiore complessità che avrebbe comportato l'adozione di una apposita variabile di condizione, non giustificata dall'estrema rarità con cui si presenta questo evento.

5.6 Lock intero repository

Per implementare la lock dell'intero repository si è utilizzata una variabile globale intera, `lockRepository`, che è uguale a -1 quando il repository *non* è in stato di lock, e diversa da -1 quando è in stato di lock. In questo secondo caso il valore della variabile è il file descriptor del socket relativo alla connessione che detiene la lock.

Di conseguenza, nell'effettuare qualsiasi operazione, i thread del pool proseguono solo se `lockRepository` è uguale a -1 o al file descriptor della propria connessione.

6 Riassunto delle modifiche apportate ai file forniti dai docenti

Anche se nel README c'è scritto esplicitamente di non modificarli, in alcune parti del codice fornito dai docenti è stato necessario effettuare delle piccole modifiche (necessarie per rimanere fedeli alle specifiche o per correggere dei bug presenti), riassunte qui di seguito:

In icl_hash.h e icl_hash.c:

- definito il tipo `hashTableEntry`, che rappresenta il file del repository (vedi sez. 4.1).
- introdotto `nentriesMutex` (vedi sez. 5.2).
- fixato il memory leak con l'aggiunta delle righe 208 e 209 (come concordato nell'email del 25 settembre 2016)
- fixato bug che impediva alla `update_insert` di restituire il puntatore adatto (come indicato nelle specifiche), rimuovendo le righe 233 e 234 (come concordato nell'email del 25 settembre)
- è stata cambiata la funzione `icl_hash_dump` per renderla conforme alla specifica. Sono state cambiate la signature e l'implementazione: l'argomento è stato cambiato dal tipo FILE* a int, in modo da poter usare le chiamate di sistema e non le funzioni di libreria, e la stampa avviene in un file binario e non testuale.

In client.c:

- eliminato un bug alla riga 106: dopo la liberazione della memoria il puntatore non veniva settato a NULL, causando un bug in caso di riposta OP_FAIL (alla riga 115 veniva letta l'area di memoria appena liberata).