Relazione progetto chatterbox per il modulo di laboratorio SOL 2017
Autore: Andrea Tosti **Matricola:** 518111 **Corso:** B

Doxygen
Il file di configurazione usato da doxygen per generare la documentazione e' `Doxyfile`.
Il parsing viene fatto su tutti i file sorgenti tranne `client.c` ; alcuni sorgenti (`utlist.h`, `icl_hash.h`, `queue.c`, `get_num.c`, `error_functions.c`) contengono la macro `DOXYGEN_SHOULD_SKIP_THIS` per evitare di fare il parsing di alcune porzioni di codice non documentate e quindi di generare warnings. Per generare una documentazione html visualizzabile da browser basta aprire un terminale, posizionarsi sulla directory principale, usare il comando *`doxygen Doxyfile`*, nella stessa cartella a questo punto troviamo una nuova cartella, `html`, all'interno della quale troviamo ed apriamo `index.html` per visualizzare la documentazione.
Script bash
Lo script bash e altri file utili sono nella cartella `script_bash`; in particolare sono presenti al suo interno:
`script.sh` sorgente dello script bash
`file.conf1` utile a verificare le potenzialita' del parser presente nello script, il quale provvede ad estrapolare il nome della directory `DirName`; nel file ci sono molti formati disponibili riconosciuti correttamente, per poterli vedere basta avviare lo script in questo modo: *`./script.sh file.conf1 0 -v`* cosicche' a schermo compariranno i nomi di tutte le directory riconosciute come valide.
`file.conf2` utile a verificare il funzionamento completo dello script, ad esempio e' possibile avviare lo script cosi': `./script.sh file.conf2 0` al fine di stampare a schermo i nomi dei file (e non delle cartelle) contenuti in `dummy folder`.
Se in un file di configurazione ho piu' di una voce `Dirname="nome directory"` , allora lo script utilizzera' l'ultima voce presente per eseguire le operazioni su essa.
Per poter visualizzare l'help bisogna lanciare lo script senza argomenti oppure con --help. Se viene specificato un tempo (secondo parametro) maggiore di zero, allora verra' chiesta una conferma prima di poter eliminare i file (per confermare premere y oppure Y).
- Concorrenza → Lock sugli insiemi di Bucket
Premessa: in ogni istante un thread comunica con un solo client;
Nella pagina a seguire faccio vedere un quadro generale di cio' che fa il client e introduco il problema riguardante l'invio di messaggi a client destinatari da parte di un thread che in certo istante sta servendo il client mittente.
Dato un threadX che stia servendo un clientX, le zone contrassegnate in azzurro sono quelle per cui altri thread diversi da threadX non possono mandare messaggi al clientX per conto di altri client diversi da clientX.

Quando un threadA sta parlando con un ClientA, decide di inviare un messaggio ad un ClientB per conto di ClientA. Ci sono situazioni in cui un threadB puo' smettere per un po' di parlare con ClientB e lasciare la possibilita' ad altri thread di parlare con ClientB. Cio' puo' avvenire nelle zone contrassegnate in arancione.
Nel sorgente chatty.c queste zone iniziano quando troviamo scritto

```
/*----*//* In questo istante un altro thread potrebbe prendere il lock e mandare un TXT/FILE *//*----*/
```

Solitamente prima del commento sopra c'e' una Unlock del BucketSet, dopo il commento c'e' una Lock del BucketSet; Cosi' facendo permetto ad altri thread di mandare messaggi a client diversi dal client che stanno servendo in un certo istante.

- Strutture dati principali usate

*Una hashtable di utenti e gruppi (`usersTable`)

*Una lista di utenti e gruppi (`listUsers`)

Per quanto riguarda la hashtable, essendo questa composta da `HASH_DIM` buckets, ho pensato di suddividerla in `HASH_BUCKETS_PER_LOCK` insiemi di buckets, in modo da poter fare il lock su un insieme di buckets (`BucketSet`) anziche' un lock per bucket, perche' in quest'ultimo caso sarebbe troppo costoso e inefficiente.

Il lock di un insieme di bucket mi permette soprattutto di gestire le comunicazioni da e verso i client da parte dei thread in mutua esclusione. Riguardo alla pagina precedente, per entrare in una delle zone contrassegnate in azzurro bisogna prima prendere il lock di un BucketSet e quando si esce da tali zone si rilascia il lock. A quel punto un altro thread puo' prendere il lock e mandare un messaggio. Nel frattempo lo stesso thread che aveva rilasciato il lock poco prima cerca di riprendere il lock, in modo da entrare in un'altra zona contrassegnata in azzurro.

In caso di invio di messaggi, un thread terra' il lock, in ogni momento, solo su un BucketSet.

La hashtable usersTable ha come chiavi i nickname degli utenti o gruppi e come valori una struttura `User` ; questa struttura contiene i seguenti campi:

`previousMessages` array circolare dei messaggi ricevuti mentre l'utente era offline

`num_pending_messages` numero dei messaggi ricevuti ancora da leggere

`index_pending_messages` che contiene l'indice del prossimo elemento da rimpiazzare eventualmente nell'array `previousMessages`

Come struttura secondaria, ma non meno importante, c'e' la lista di utenti/gruppi. Questa aiuta principalmente a prelevare informazioni sui gruppi/utenti in fase di invio di messaggi, perche' invece che lockare un'intera hashtable per prelevare nickname e file descriptor, mantengo un lock su un BucketSet e un lock sulla lista.

Ad esempio in fase di invio messaggio a tutti i membri di un gruppo o a tutti in generale, si fa prima una copia degli utenti presenti in lista, poi l'unlock della lista, per ogni utente presente nella copia si prende il lock sul BucketSet relativo al nickname e si ricontrolla se l'utente non si sia deregistrato/disconnesso nel frattempo ricontrollando la lista originale `listUsers`.

La lista `listUsers` contiene elementi di tipo `list_string_el`, che e' una struttura contenente i seguenti campi:

`nickname` nome dell'utente o del gruppo

`fd` file descriptor di un utente se e' connesso, -1 altrimenti

`online` booleano che indica se l'utente e' online

`groupMembers` che assume NULL se si tratta di un utente, altrimenti una lista di utenti appartenenti al gruppo; in tal caso il primo elemento di questa lista e' sempre il fondatore del gruppo.

Concorrenza → altre sezioni critiche
Oltre all'array di mutex utilizzato per la hashtable e la mutex per la lista utenti/gruppi, sono presenti le seguenti mutex:
- `qlock` associata ad una variabile di condizionamento `qcond` per la coda `fdQueue`; i thread workers estraggono in mutua esclusione i file descriptor presenti nella coda (consumatori) mentre il thread main li inserisce, sempre in mutua esclusione(produttore)
- `fdsetlock` per l'insieme master_read_fd dei descrittori, tale set viene utilizzato sia nella Select dal thread main, sia dai worker: il main toglie un file descriptor dal fd_set e lo inserisce nella coda `fdQueue`, i worker reinseriscono il file descriptor nel fd_set solo se hanno servito una richiesta dal client.
- `statslock` per le statistiche, i thread worker aggiornano le statistiche in mutua esclusione
- `numonlinelock` per il numero di utenti online, sia il thread main che i worker aggiornano tale numero in mutua esclusione

- Segnali
Per gestire i segnali e' stata utilizzata una tecnica nota come "Self Pipe Trick", la quale sfrutta una pipe che ha la write-end e la read-end non bloccanti, dopodiche', alla ricezione di un segnale, la funzione `handler` dei segnali provvede a scrivere un byte su questa pipe, nel caso di SIGTERM, SIGQUIT e SIGINT viene scritto `0`, nel caso di SIGUSR1 viene scritto `1`. Infine, la Select, alla ricezione di uno dei segnali istallati, vedra' l'evento di scrittura sulla pipe e quindi il thread Main potra' fare tutto il necessario per gestire il segnale.

- Gestione della memoria
Durante il normale flusso di esecuzione dei thread worker la memoria allocata per strutture dati quali i messaggi o le liste temporanee viene liberata sempre a fine di un'operazione;
Nel caso in cui arrivi un segnale di interruzione, i thread worker vengono informati di finire di gestire la loro ultima richiesta (scrivendo -1 sulla coda dei file descriptor da gestire, `fdQueue`) e, dopo aver atteso il completamento di tutti i thread worker, il main salta ad una etichetta di goto `cleanup` e libera la memoria allocata per la lista utenti/gruppi `listUsers`, la coda dei descrittori `fdQueue`, la hashtable `usersTable` e qualche altra variabile minore.

- In quali macchine Makefile e codice compilano ed eseguono correttamente
*Linux 4.15.10-1-MANJARO (x86_64), GNU C Library version 2.26 (stable), GNU C Compiler version 7.3.0 (GCC)
*Macchine di laboratorio

-Testcase aggiuntivo
E' stato aggiunto il test `testcustom.sh` (eseguibile con *`make testcustom`*); tale test vuole evidenziare i seguenti aspetti:
* richiesta di un'operazione subito dopo la deregistrazione, sapendo che il client non fa il solito controllo "nickneeded=1" perche' il client e' ancora connesso.
* in fase di deregistrazione oppure cancellazione da un gruppo, di un utente appartenente a uno o piu' gruppi di cui non e' fondatore, l'utente venga rimosso da tutti i gruppi cui apparteneva
* in fase di deregistrazione oppure di cancellazione da un gruppo, di un utente appartenente a uno o piu' gruppi di cui e' fondatore, tali gruppi vengano rimossi e quindi gli utenti appartenenti ad essi non ne facciano piu' parte

-Strutturazione del codice e librerie
Vedere la documentazione Doxygen, ad esempio sul browser, nella sezione
Files → File List, viene spiegato nel dettaglio la suddivisione di file e librerie.
Altri files non documentati in Doxygen sono:
`Build_ename.sh` che, se eseguito con *`./Build_ename.sh > "./ename.c.inc"`*,
crea una nuova versione del file `ename.c.inc` che viene utilizzato in `error_functions.c` facendo il parsing dei nomi degli errori presenti in `errno.h`.

- Difficolta' incontrate
Fin dall'inizio sono stati utilizzati Socket non bloccanti (ossia, settati in modo non bloccante dopo una accept()), ma senza successo a causa della difficolta' nel gestire il tutto con la Select. Come soluzione ho ripiegato sui Socket bloccanti.