UNIVERSITÀ DI PISA

# Relazione progetto “Chatterbox”
# Modulo Laboratorio Sistemi Operativi A.A. 2017-2018

**Pietro Libro**
**545559**

## Scelte di progetto

Nella realizzazione del progetto si è resa necessaria per prima cosa la scelta di una struttura dati
adeguata, per la memorizzazione di una mole elevata di utenti. La scelta, come anche suggerito dal docente, è ricaduta su un HashTable, nello specifico icl_hash. Generalmente questo tipo di strutture dati consentono un occupazione spaziale pari ad O(n) ed un accesso in tempo in media O(1+a), ove a rappresenta il fattore di carico della HashTable. La scelta successiva è ricaduta nella struttura dati da memorizzare nella HashTable per conservare le informazioni dell’utente registrato.
A tale scopo nel file “stuffs.h” è definita la struttura dati utente di tipo struct client_struct che contiene al suo interno i diversi campi tra cui anche il nome utente che è inoltre anche la chiave per la ricerca all’interno della HashTable.

```
stuffs.h
31  // Struttura dati per settaggi da file di configurazione
32  struct setting{
33      char *UnixPath;
34      char *DirName;
35      char *StatFileName;
36      int MaxConnections;
37      int ThreadsInPool;
38      int MaxMsgSize;
39      int MaxFileSize;
40      int MaxHistMsgs;
41  };
42  // Istanziazione condivisa
43  struct setting *setupped;
44  // Struttura gestione utente in HashTable
45  typedef struct client_struct{
46      char *nome;
47      int client_fd;
48      message_t **history;
49      int current;
50      int testa;
51  }utente;
52  // Struttura doppiamente linkata per Utente Online
53  typedef struct online_user{
54      char name[MAX_NAME_LENGTH];
55      struct online_user* prev;
56      struct online_user* next;
57  }user;
58
```

La history dell’utente è stata implementata attraverso un puntatore a puntatori di message_t. Per agevolare la gestione di quest’ultima attraverso doppio indice con il solo scopo di individuare rapidamente il primo messaggio.

La history diversamente da quanto richiesto non conserva tutti i messaggi/files, ma solo quelli che devono ancora essere ricevuti dal destinatario, inoltre quando un utente ha una history piena, se non online, non è in grado di ricevere altri messaggi e una notifica di operazione fallita viene recapitata al mittente.
Questa scelta implementativa viene fatta per consentire all'utente di non perdere alcun messaggio non letto che magari poteva risultare di rilevante importanza e al tempo stesso far sapere al mittente che l'utente in questione non ha ricevuto il messaggio. D'altro canto, però, contribuisce a complicare leggermente la leggibilità del codice che gestisce le operazioni POSTTXT/POSTTXALL/POSTFILE.
Per la gestione degli utenti online utilizzo una lista doppiamente linkata, (struttura user nel file "stuffs.h"), che contiene solo il nome dell'utente online e due puntatori, uno per l'elemento precedente e uno per il successivo. L'inserimento in questa lista avviene in testa. Questa implementazione consente di risparmiare tempo, perché piuttosto che scorrere tutta la HashTable con relativa acquisizione di lock, viene scorsa solo una lista con dimensione <= di MaxConnections, in cui vi sono nomi utente di utenti effettivamente online.
Per la gestione dei settaggi contenuti nel file di configurazione, ho implementato la struttura dati struct setting nel file "stuffs.h", queste informazioni dopo il parsing dal file sono disponibili in lettura a tutte le funzioni/thread senza necessita di mutua esclusione.
Si sono rese inoltre necessarie svariate variabili condivise e relative lock per mutua esclusione per consentire, ad esempio, la scrittura su un determinato FileDescriptor, comunicare ai thread la terminazione e più in generale per gestire tutte le potenziali situazioni di RaceCondition e quindi inconsistenza dello stato in cui si sarebbe potuto lasciare la risorsa.
L'accesso all' HashTable è regolato attraverso lock che gestiscono ognuna blocchi di 4 utenti.
Gli utenti da servire vengo messi in "coda" attraverso un vettore circolare di interi (variabile globale toServe), che memorizza i FileDescriptor da servire.

La gestione dei file viene trattata attraverso le funzioni fopen, fclose, fgets, fputs ed un buffer di supporto per la copia effettiva del file.
Per quanto riguarda la gestione della memoria ogni funzione/metodo appena possibile libera la parte di memoria precedentemente occupata e non più necessaria.

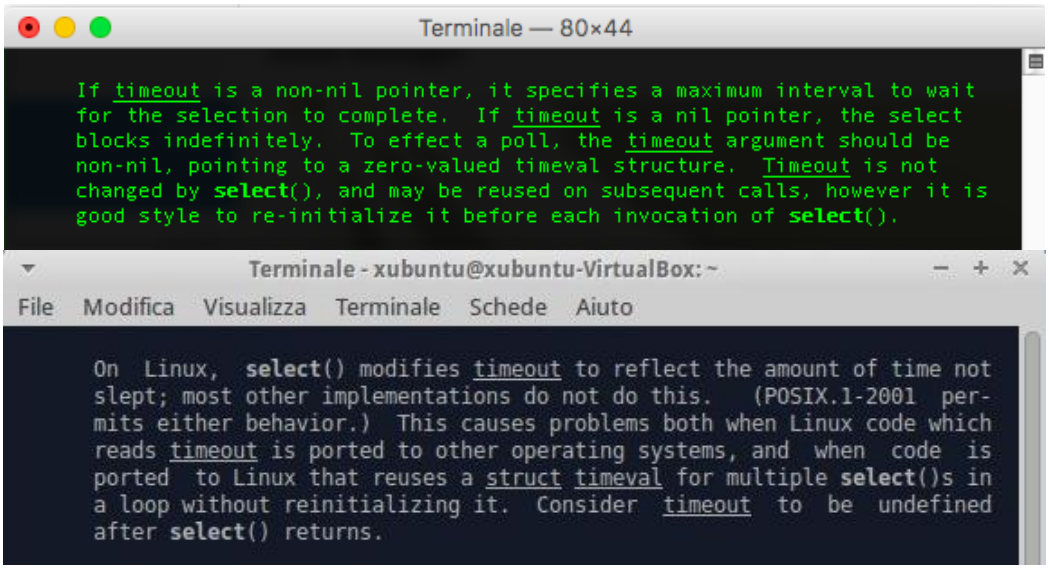
## Struttura dell'esecuzione

Durante l'avvio del server è attivo un solo thread che utilizza la funzione Main per eseguire tutte le procedure necessarie all'inizializzazione del server stesso, come ad esempio il parsing del file di configurazione, l'istanziazione e settaggio di varibili condivise, mutex ecc. Procede inoltre all'avvio dei 2 Thread principali (SignalHandler e Listener) oltre che settare e avviare il pool di Thread, in cui ogni Thread si metterà in wait di una signal su variabile di condizionamento gestita unicamente dal Thread Listener.
Come richiesto da specifica i segnali vengono gestiti da un unico thread che si occupa poi di comunicare l'eventuale terminazione a Listener (variabile globale serverRun), indirettamente ai Thread del pool, e al thread Main. I thread del pool verranno risvegliati del Listener, attraverso una broadcast, ed attenderà la loro terminazione concludendo la sua esecuzione ritornando controllo al Main che terminerà anch'esso.
Una volta avviato gli unici thread attivi saranno il SignalHandler e il Listener.
Il primo resterà per tutta l'esecuzione del server in attesa di segnali da gestire attraverso una sigwait a cui viene data per parametro una maschera settata opportunamente per ascoltare segnali di tipo SIGINT/SIGQUIT/SIGTERM/SIGUSR1 e gestirli.
Il listener attraverso la select sul FD_SET ascolta le richieste in arrivo sulla serverSocket e su una pipe, quest'ultima è l'unica comunicazione diretta, monodirezionale, fra Worker e Listener, e procede ad inserire il fd della comunicazione nel vettore circolare toServe ed a mandare una signal sulla variabile di condizionamento in cui i thread si mettono in wait.
Una volta risvegliato il worker estrae un FD dal vettore circolare e procede alla gestione della richiesta, attraverso funzioni inline (nessun nuovo record di attivazione viene creato).

In seguito, il FD viene inviato al Listener attraverso una pipe.
Nel caso in cui non venga letta alcuna comunicazione sul file descriptor da parte del Worker (nessuna operazione da gestire), la connessione viene chiusa e il file descriptor non viene ricomunicato al thread Listener.
A questo punto il worker si rimette in attesa sulla varibile di conzionamento.

## Osservazioni

Non ostante Linux e MacOs siano entrambi sistemi operativi derivati da Unix e aderenti a Posix sono presenti differenze tra essi, talvolta fondamentali, anche nell' implementazione di quelle che ad una prima occhiata possono sembrare banalità. Un esempio concreto è fornito dalla select, la cui implementazione in MacOS secondo il man non va a modificare la timeval struct. Al contrario di quanto avviene in Linux, dove invece è espressamente indicato di riassegnare il valore al timeout.



# Note

Il progetto è stato sviluppato inizialmente su sistema operativo MACOS X (10.12.6) con l'ausilio di una versione porting di Valgrind (attraverso gestore di pacchetti Brew), per poi proseguire su macchina virtuale Xubuntu 14.10 fornita sul sito del corso.

Per un corretto svolgimento del test3 (testconf.sh), è stato necessario aggiungere nel makefile, il file oggetto chatty.o in modo da ottenere una libchatty.a con peso maggiore di 50KB.

L'esecuzione è stata testata su:

- VirtualMachine Xubuntu 14.10 x64 (2 Core/3GB Ram)
  on Windows 10 x64 (4 Core/8GB Ram
- Ubuntu 18.10 x64 (4 Core/8GB Ram)