1. Strutture Dati e Gestione Memoria

Nell' implementazione del server sono utilizzate diverse strutture dati, la prima è la struct *conf_struct* rinominata in *configurazione*, che contiene le informazioni che vengono prese dal file di configurazione il cui nome è passato come argomento al *main* e che serviranno per eseguire il server in diverse configurazioni.
Ad esempio possono variare il numero di thread utilizzati per elaborare le richieste dei client, il nome della cartella in cui memorizzare i file che gli utenti si scambiano o il nome del file in cui stampare le statistiche.
La scelta di riunire tutte queste variabili in un' unica struttura è stata fatta per chiarezza e per poter separare dal server e dagli altri moduli l' estrapolazione di informazioni collocandola in un modulo specifico, dato che viene eseguita solo una volta prima che il server inizi il suo lavoro effettivo ed è quindi indipendente dalle richieste dei client che il server deve soddisfare.
In questo modo inoltre posso passare solo il puntatore alla memoria allocata per la struct alla procedura apposita, implementata in un altro file, che la restituisce contenente tutte le informazioni necessarie.
Al momento dell' inizializzazione della struttura con la funzione specifica viene allocata memoria per le tre stringhe contenenti il path del socket unix, il path del file dove vengono stampate le statistiche e il nome della cartella dove salvare i file temporanei, questa memoria verrà poi liberata alla fine del *main*, cioè alla terminazione del programma.
Per la memorizzazione di utenti e gruppi registrati al servizio sono utilizzate due tabelle hash molto simili.
La prima, quella per gli utenti (*reg_users*), è implementata con una struct i cui campi sono il numero di utenti registrati (*nentries*), quante caselle è lunga (*nbuckets)*, il puntatore alla prima casella (*buckets*) e la funzione hash e viene creata con 256 caselle.
Anche le entry della tabella, che rappresentano gli utenti, hanno una propria struttura dati che contiene il descrittore della connessione usata dall' utente (*fd*), una variabile che indica se l' utente è connesso (*connesso*), la history dei messaggi ricevuti dall' utente (*msgs*), il numero di messaggi nella history (*nmsgcoda*), il nome dell' utente (*data*) e il puntatore alla entry successiva (*next*).
Nel dettaglio:

- il descrittore usato dall' utente è l' ultimo usato, visto che ad ogni connessione viene aggiornato, mentre se l' utente si disconnette è -1;
- la variabile che indica se è connesso assume in questo caso valore 1, altrimenti 0;
- la history è implementata come una coda a cui si appendono nuovi messaggi in fondo e si prelevano dalla testa e i cui nodi hanno un campo che contiene un messaggio completo di tutte le parti (header e body), che è una copia del messaggio originale, e un puntatore al nodo successivo;
- il numero di messaggi nella history serve per sapere quando questa è piena senza dover tutte le volte contarli rallentando l' esecuzione del programma, dato che c' è un numero massimo di messaggi ricordabili per utente ed è specificato nel file di configurazione;
- il nome dell' utente è il suo identificativo unico, perchè due utenti con lo stesso nome non possono essere aggiunti, infatti in caso di richieste di questo tipo il server risponderà con un *OP_NICK_ALREADY*;
- il puntatore alla entry successiva serve per poter scorrere la lista, ad esempio quando si crea la lista utenti connessi oppure quando si invia a tutti un messaggio.

La memoria per la tabella hash viene aumentata ogni volta che si aggiunge un iscritto, allocando lo spazio per il nodo occupato dalla entry e per le strutture dati che contiene.
Anche per la history dei messaggi inizialmente non si alloca niente e lo spazio viene poi occupato man mano che questi vengono aggiunti.
Allo stesso modo man mano che vengono prelevati con una richiesta di tipo *GETPREVMSG_OP* si libera lo spazio che occupano ma, siccome il client può non richiedere questa operazione per tutti i messaggi che vengono scambiati, alla fine del programma con una funzione specifica si procede a eliminare l' intera tabella hash, liberando i vari nomi utente e le varie code dei messaggi che erano rimasti non consegnati.
I messaggi inseriti nella history sono in realtà copie del messaggio originale, questo perché passando il puntatore del messaggio originale si creano problemi quando questo messaggio è rivolto a tutti gli utenti, in questo caso quando veniva prelevato dalla coda di uno dei destinatari e la memoria che occupava liberata, gli altri destinatari non potevano più recuperarlo, ma se non veniva liberato poi non c'era più modo di liberarlo non sapendo quando tutti i destinatari lo avrebbero ricevuto.
Mettendo invece una copia, una volta copiato nella history di tutti viene liberata la memoria occupata dall' originale e tutti ne hanno una copia che verrà liberata quando richiesta oppure alla fine del programma.
Anche una sola entry della tabella hash, rappresentante un solo utente, può essere eliminata se il client richiede una *UNREGISTER_OP*, in questo caso si libera la memoria occupata dalla entry, dalla history dei messaggi e dal nome utente.
Analogamente funziona la struttura dati rappresentante i gruppi (*reg_groups*) che è una tabella hash come quella usata per gli utenti, con i campi modificati per adattarla al contesto e meno caselle visto che i gruppi saranno

probabilmente in numero molto inferiore rispetto agli utenti, ma con lo stesso algoritmo di hashing, quindi la stessa funzione hash.
Ogni volta che viene creato un gruppo si alloca la memoria necessaria.
La singola entry, rappresentante il gruppo, contiene i campi:

- data è il nome del gruppo, il suo identificativo unico;
- admin è il nome dell' utente che ha creato il gruppo e quindi l' unico che può eliminarlo;
- n_users è il numero di utenti facenti parte del gruppo;
- next è il puntatore al gruppo successivo nella tabella;
- u_coda è la lista degli utenti aggiunti al gruppo e quindi che possono inviare messaggi al gruppo o riceverne di indirizzati al gruppo

La lista degli utenti connessi è una lista con puntatori come quella dei messaggi di un utente, in cui ogni nodo contiene un nome utente e un puntatore all' elemento successivo e viene allocato quando l' utente viene aggiunto al gruppo.
Un' altra struttura fondamentale per il funzionamento del server è la coda in cui il thread listener, cioè il *main*, inserisce i descrittori delle connessioni accettate.
I nodi di questa coda sono composti da un campo (*fd*) che rappresenta il descrittore contenuto nel nodo e un puntatore al nodo successivo (*next*) e anche qui i nuovi elementi vengono messi in fondo e quelli da prelevare sono presi dalla testa.
Questa coda viene liberata alla fine del programma con una specifica procedura.
Oltre a queste strutture sono presenti anche un array contenente 64 variabili di mutua esclusione usate per regolare l' accesso alla tabella degli utenti e uno contenente i thread id dei thread del pool.
Anche queste due vengono liberate alla terminazione del programma dal *main*.
Quando viene chiesta, tramite l' invio di alcuni segnali specifici, la terminazione del server, risulteranno inevitabilmente allocate ancora molte zone della memoria, quelle occupate dagli utenti o gruppi ancora iscritti e dai descrittori di file ancora in coda ad esempio.
Per questo il *main* prima di terminare, ma dopo aver atteso che tutti i thread abbiano terminato il proprio lavoro, libera la memoria occupata dall' intera coda dei descrittori, dalla tabelle hash degli utenti e dei gruppi, dall' array contenente gli id dei thread del pool e dalla struttura contenente le informazioni di configurazione.
A parte la struct contenente soltanto le informazioni di configurazione e l' array dei thread id, usati solo dal *main*, tutte le strutture precedenti devono essere condivise da tutti i thread del programma, che possono leggerle oppure modificarle, quindi devono essere accedute in mutua esclusione.
Da questo discorso è escluso l' array delle variabili di mutua esclusione, essendo usato appunto per questo scopo.

2. Suddivisione in files

La struttura del server è divisa su più files, che una volta compilati andranno a formare la libreria *libchatty.a*.
Questa divisione è stata fatta per separare il più possibile i vari moduli del programma, operazioni indipendenti le une dalle altre che vengono eseguite dal server.
Il metodo *main* che descrive come vengono eseguite le varie procedure e gestite le varie richieste e la procedura che utilizzano i thread per farlo sono definiti nel file *chatty.c*, quello che utilizza la libreria *libchatty*, e che è appunto eseguibile una volta terminata la fase di compilazione e poi di linking.
Ogni altro modulo contiene le strutture e le funzioni necessarie a implementare una particolare funzione necessaria al server per funzionare.
Nel dettaglio questi file sono:

- **config.h**

Contiene le varie define, quelle utilizzate dall' algoritmo di hashing, il numero di caselle delle due tabelle hash e il numero di mutex utilizzati per regolare la concorrenza sulla tabella hash degli utenti, infine due macro utilizzate per controllare se le chiamate di sistema restituiscono errori (codificati con -1 o *NULL*).

- **message.h**

Contiene le informazioni riguardanti il messaggio, cioè l' entità scambiabile tra gli utenti, che può contenere caratteri oppure un file.
Quindi qui è definita la struct che rappresenta un messaggio, a sua volta contenente due struct che rappresentano la parte header, che contiene l' operazione da effettuare e il nome del mittente, e il corpo, che contiene il messaggio testuale e a sua volta una struct che rappresenta l' header e contiene nome del destinatario e lunghezza del messaggio testuale.
Inoltre sono dichiarate anche due funzioni utilizzate una per settare l' header del messaggio e una per settarne il corpo.

- **connections.h**

Contiene la dichiarazione delle funzioni che gestiscono la connessione tra il server e i client.
Ci sono la funzione utilizzata dai client per aprire la connessione verso il server e tutte quelle per inviare o ricevere un intero messaggio oppure solo header o solo corpo.

- **connections.c**

Contiene l' implementazione delle funzioni dichiarate in *connections.h*.
Per aprire una connessione verso il server copia il nome del path del socket usato, contenuto nel file di configurazione, nella struttura che contiene l' indirizzo *sockaddr_un* e setta il campo *family* di questa a AF_UNIX, crea il socket e infine prova a connettersi su questo un numero prestabilito di volte, passato come parametro alla funzione che apre la connessione. Fatto questo, se la connessione è stabilita prima di esaurire i tentativi, restituisce il descrittore della connessione al chiamante.
Le funzioni di invio messaggio hanno più o meno tutte la stessa struttura, cioè prendono le varie parti del messaggio e le inviano tramite *write* sul socket una a una, con la differenza che quella che invia solo l' header si ferma dopo aver inviato operazione e nome del mittente, quella solo per il corpo omette questo passaggio e invia direttamente destinatario, lunghezza e messaggio testuale (o file).
E' molto importante che tutte queste però inviino le cose nello stesso ordine poiché nello stesso ordine leggono quelle che si occupano di ricevere, e inoltre a volte un messaggio viene spedito interamente e non letto subito interamente ma in due momenti diversi e quindi le procedure per ricevere solo header e solo corpo insieme devono essere in grado di ricevere un messaggio spedito interamente tramite procedura apposita.
In particolare l' ordine in cui leggere/scrivere è sempre: operazione richiesta, mittente, destinatario, lunghezza del messaggio e infine messaggio testuale o file, che può essere scritto/letto anche in più *write*/*read*, dato che, soprattutto in caso di file, le dimensioni possono essere molto grandi, ma è garantito che tutto il messaggio venga scritto/letto.

- **icl_hash.h**

Contiene le strutture che rappresentano la tabella hash e le entry di questa, cioè gli utenti, e vi sono dichiarate tutte le funzioni usate per interagire con questa tabella, cioè crearla e distruggerla, inserire ed eliminare utenti, sapere se un utente vi appartiene o meno.
E' quella fornita dai docenti del corso con qualche adattamento al contesto per la struct che rappresenta l' utente: sono state aggiunte la coda dei messaggi, una variabile che indica se è connesso o no, una che indica il numero dei messaggi in coda e una che indica il descrittore della connessione che usa. Inoltre dalla struct che rappresenta la tabella hash è stato rimosso il comparatore, perché non usato.

- **icl_hash.c**

Contiene l' implementazione delle funzioni dichiarate in *icl_hash.h* più una utile alle altre che implementa l' algoritmo di hashing.
Le funzioni sono quelle facenti parte del file fornito dai docenti con le modifiche alle funzioni dovute alla modifica della struttura dati che rappresenta l' utente.

- **history.h**

Contiene la dichiarazione delle funzioni che servono per gestire la history dei messaggi, implementata come una lista puntata, e la struttura dati che rappresenta il singolo nodo della coda.
Con queste funzioni si possono aggiungere o togliere nodi, cioè messaggi, alla coda oppure eliminarla completamente.

- **history.c**

Contiene l' implementazione delle funzioni definite in *history.h*, cioè delle funzioni che gestiscono la coda dei messaggi precedentemente inviati al singolo utente.
La procedura di aggiunta di un nodo prende come parametri la testa della coda, che può essere anche *NULL* se è vuota, e il messaggio da inserire, ne crea una copia (il motivo di questa scelta implementativa è spiegato nella sezione (1)), scorre tutta la lista e aggiunge questa copia in fondo, infine restituisce il puntatore alla testa della coda.
Quella che invece ne toglie uno prende come parametro il puntatore a questa coda, elimina il nodo in testa liberando anche la memoria occupata e sposta il puntatore a quello successivo, infine restituisce il puntatore alla testa della coda così modificata.
L' ultima procedura provvede a eliminare tutta la coda e restituire un puntatore a *NULL*, ed è utilizzata solo quando si elimina un utente a causa di una *UNREGISTER_OP* oppure al termine del *main* quando si distrugge l' intera tabella hash.

- **codafd.h**

Contiene la dichiarazione delle funzioni per interagire con la coda dei descrittori delle connessioni e la struttura che ne rappresenta un nodo.
In questa coda il server colloca i descrittori sui quali è stata accettata una connessione con il client per essere poi presi da un thread del pool che ne elaborerà la richiesta.

- **codafd.c**

Contiene l' implementazione delle funzioni dichiarate in *codafd.h*.
La prima è quella che inserisce un nodo in coda (*push*) che prende come parametri la testa della coda, puntatore a *NULL* se vuota, e il descrittore da inserire, scorre tutta la lista e appende in fondo il nodo per poi ritornare il puntatore alla testa della coda modificata.
La seconda (*pop*) prende come parametro il puntatore al nodo in testa alla coda da liberare e libera la memoria occupata da questo nodo, spostando il puntatore alla testa al nodo successivo.
L' ultima (*elimina_coda_fd*) è chiamata dal *main* alla fine della propria esecuzione per liberare la memoria occupata da tutta la coda dei descrittori di file.

- **mutex.h**

Contiene la dichiarazione delle funzioni usate per gestire la concorrenza, cioè la struttura contenente i semafori usati per accedere alla tabella hash degli utenti (l' array di mutex *mtx*), quello per regolare l' accesso alle statistiche (*mtxstats*), quello per la coda dei descrittori (*mtxcoda*), quello per entrare nel recinto di mutua esclusione mentre viene copiato un file, evitando così che altri richiedano operazioni sullo stesso in questo intervallo di tempo (*mtxfiles*), quello per regolare l' accesso alla tabella hash dei gruppi (*mtxgroups*) e infine la variabile di condizione usata dai thread per mettersi in attesa se la coda dei descrittori è vuota (*condcoda*).

- **mutex.c**

In questo file è presente l' implementazione delle funzioni dichiarate in *mutex.h*.
Queste implementazioni si limitano a chiamare le funzioni della libreria *pthread.h*, introdotta in *mutex.h*, riguardanti la concorrenza (lock, unlock, wait, signal, broadcast) e a verificare che non restituiscano errori, stampandoli eventualmente.

- **configurazione.h**

Contiene la dichiarazione della struct *conf_struct* rinominata per comodità e migliore comprensione in *configurazione*.
E' dichiarata qui anche la funzione *parsing* che, chiamata all' inizio del programma, estrae le informazioni utili dal file di configurazione contenuto nella cartella *DATA* e il cui nome è passato come argomento al *main* e le inserisce nei campi di *configurazione*, il cui puntatore era passato come argomento alla funzione, e ne restituisce il puntatore (alla struct modificata).
Le altre due procedure dichiarate qui (*inizializza_conf* e *pulizia_stringa*) servono la prima ad inizializzare la struttura dati di cui è passato il puntatore, la seconda è una funzione ausiliaria usata da *parsing* per rimuovere eventuali caratteri speciali dalle stringhe contenenti il nome del path del socket, il path della directory in cui collocare i file temporanei e il nome del file contenente le statistiche.

- **configurazione.c**

Implementa l' header file *configurazione.h*.
inizializza_conf alloca lo spazio di memoria necessario per le stringhe che conterranno i parametri del file di configurazione *UnixPath, DirName, StatFileName* e inizializza a 0 gli interi contenenti gli altri parametri.
pulizia_stringa si limita a eliminare dalla stringa ricevuta come argomento i caratteri corrispondenti, nella tabella ASCII, ai valori 10 e 9, che creavano problemi nel momento in cui venivano creati i path per la copia dei file che gli utenti si volevano scambiare nella cartella il cui nome è contenuto nel parametro *DirName*.
La procedura *parsing* alloca spazio per un array di stringhe, apre il file di configurazione il cui nome è passato come parametro, lo scandisce riga per riga e controlla per prima cosa se il primo carattere di questa è un # e in tal caso passa direttamente alla riga di successiva dato che è un commento.
Se è una riga valida prende la prima parola di questa, spezzando la stringa fino al primo spazio con la funzione *strtok* contenuta nella libreria *string.h*, e controlla a quale parametro di configurazione corrisponde.
A questo punto continua a spezzare la stringa: se al primo passo trova il nome di uno dei parametri, al secondo trova necessariamente un uguale e al terzo l' informazione cercata, che verrà poi troncata adeguatamente se è una stringa oppure tradotta in intero con la funzione *atoi* della libreria *stdlib.h*.

- **stats.h**

Contiene la definizione della struct contenente le statistiche del server in esecuzione, ogni campo è una di queste, e la funzione che le stampa in un file, il cui nome è specificato nel file di configurazione, in aggiunta all' ora esatta (con precisione al secondo) in cui vengono stampate.
Questa procedura, di nome *printStats* viene chiamata dal thread che si occupa di gestire i segnali, quando riceve il segnale *SIGUSR1*.
Il file viene aperto, viene appesa una riga con tutte le statistiche e infine viene chiuso.
Se non già esistente, il file viene creato dal metodo *main* durante la procedura di inizializzazione del server.

- **ops.h**

Questo file contiene semplicemente una codifica delle varie operazioni associandole ad interi, ad esempio, se il server riceve uno 0 dal client nel campo del messaggio che identifica l' operazione richiesta, saprà, grazie a questa libreria, che il client vuole effettuare una *REGISTER_OP*, e lo stesso vale per tutte le operazioni possibili.
Inoltre sono codificati anche i messaggi di risposta dal server verso il client, che possono significare che tutto è andato a buon fine (*OP_OK*) oppure identificare precisi errori, in modo che il client possa sapere quale è stato il problema.

- **group_hash.h**

Contiene la dichiarazione dei metodi utilizzati per gestire la tabella hash rappresentante i gruppi, con operazioni per aggiungere gruppi, rimuoverli, restituirne uno con un determinato nome se presente, eliminare un user da tutti i gruppi (quando viene deregistrato dal server) e infine liberare la memoria occupata da questa struttura (viene fatto alla fine del programma).
Inoltre qui vengono dichiarate anche le struct che rappresentano l' intera tabella hash e la singola entry della tabella, cioè un singolo gruppo.
La lista degli utenti appartenenti al gruppo, che è un campo della entry, viene dichiarata e gestita nel modulo *usercoda.h*.

- **group_hash.c**

Implementa il file *group_hash.h*, i metodi funzionano come quelli utilizzati per gestire la tabella hash degli utenti, dato che la struttura è molto simile, e l' algoritmo di hashing è lo stesso.
Viene sempre calcolato un valore hash a partire dal nome del gruppo e poi usato per trovare il gruppo da eliminare, ad esempio, da restituire in un' operazione di ricerca, ma anche per sapere dove allocarlo quando è richiesto di aggiungerlo.
Quando viene deregistrato un utente è necessario anche rimuoverlo da tutti i gruppi in cui è presente e rimuovere quelli di cui era il creatore, per evitare che si formino gruppi non più eliminabili, dato che solo il creatore può eliminarli.
Per fare questo viene scandita tutta la tabella hash e, per ogni entry, cioè per ogni gruppo, viene controllato se il nome dell' user, passato come argomento alla funzione, è uguale al campo *admin* del gruppo e in quel caso viene eliminato il gruppo intero, in caso contrario, se l' utente appartiene a un gruppo viene tolto dai nomi degli utenti appartenenti al gruppo.
Quando invece il server termina è importante deallocare tutta la memoria allocata durante l' esecuzione per i gruppi, così come viene fatto per la struttura rappresentante gli utenti, e viene fatto scandendo la tabella ed eliminando per ogni singola entry tutto quello che era stato allocato, in particolare la lista degli utenti che facevano parte del gruppo in questione e i nomi dell' admin e del gruppo stesso.

- **usercoda.h**

Contiene quello che riguarda la lista degli utenti iscritti a un gruppo.
La struct che rappresenta un nodo della lista, qui definita, contiene solo un nome di utente e il puntatore all' elemento successivo.
Vi sono poi le dichiarazioni delle funzioni per aggiungere un utente a una lista, eliminare un utente da una lista, sapere se un utente appartiene o meno ad una lista e infine deallocare tutta la memoria occupata da una lista.

- **usercoda.c**

Implementa le funzioni per gestire la lista degli utenti appartenenti ad un gruppo.
Per eliminare un utente viene scandita la lista e, una volta trovato il suo nome viene eliminato e vengono spostati i puntatori in maniera opportuna.

Al momento di una richiesta di tipo *ADDGROUP_OP* l' utente viene messo in testa alla coda per risparmiare il tempo di scorrere tutta la lista fino in fondo, dato che l' ordine non è importante in questo caso.
Per trovare un utente viene scandita tutta la lista fino a trovare un nome uguale a quello passato come argomento alla funzione, a quel punto viene restituito il valore uno, se invece arriva alla fine l' utente non è stato trovato e viene restituito uno zero.
Anche per l' eliminazione viene scandita tutta la lista ed eliminati i nodi uno ad uno fino in fondo, viene fatto alla fine del programma oppure quando viene eliminato un gruppo.

3. Interazione tra processi/threads

Il thread *main* inizia la propria esecuzione ignorando il segnale *SIGPIPE*, generando il thread che gestirà i segnali e che lavorerà per tutto il tempo in maniera indipendente dal resto del programma, ed estraendo le informazioni necessarie dal file di configurazione per poi creare la cartella dove dovranno essere memorizzati i files e il file che conterrà le statistiche.
Vengono poi create le tabelle hash degli utenti e dei gruppi ed anche una pipe dove i thread inseriranno dei descrittori di client con cui hanno completato un' interazione ma che devono fare altre richieste, in modo che il *main* li riattivi nella select e che vengano serviti nuovamente.
Successivamente viene generato il pool di thread, che saranno quanti specificati nel file di configurazione meno uno, che è quello già creato che si occupa dei segnali.
Il main crea un socket, lo associa all' indirizzo passato come argomento al main e si mette in ascolto per un numero di connessioni il cui massimo è specificato nel file di configurazione.
Per ascoltare le richieste dei client utilizza la funzione *select* che attende in sola lettura da un gruppo di descrittori.
La maschera che usa inizialmente ha settato a 1 solo un primo descrittore su cui ascolta le richieste dei client e quello della pipe, con la quale attende comunicazione dai thread.
La pipe serve ai thread per comunicare al *main* quando un descrittore con per il quale hanno eseguito una richiesta deve fare altre richieste, e quindi va risettato a 1 nella maschera usata dalla *select*.
Per dare la precedenza alle nuove connessioni però il descrittore della pipe è ascoltato solo una volta ogni mezzo secondo, grazie a un timer che ogni volta che trascorre questo intervallo di tempo invia un segnale di tipo *SIGALRM*, catturato dal gestore che setta una variabile globale per comunicarlo.
Dopo aver settato anche il timer il *main* inizia un ciclo che si interromperà solo quando il gestore dei segnali imposterà a 1 la variabile che indica appunto che è arrivato uno dei segnali di terminazione, e lo stesso farà il ciclo eseguito dai thread.
Nel ciclo si mette in attesa sulla *select*, quando è possibile leggere da uno dei descrittori del gruppo li testa tutti per vedere da quale arriva la richiesta, se è quello che usa lui stesso viene accettata la connessione e il file descriptor ritornato dalla procedura *accept* viene settato nella maschera, in modo che sia considerato dalla *select*.
Il descrittore che fa sbloccare la *select* potrebbe anche essere quello usato per la pipe, che viene considerato solo se è anche scaduto il timer nel frattempo, e in quel caso il descrittore specificato da uno dei server e comunicato tramite la pipe viene riaggiunto alla maschera perché ha altre operazioni da richiedere e deve quindi essere ascoltato di nuovo.
Se invece il file descriptor, tra quelli nel gruppo, non è quello che usa il *main* né quello della pipe, allora è uno di quelli con cui la connessione era già stata accettata, cioè usato da un client per fare una richiesta, quindi viene tolto dalla maschera e il suo descrittore viene messo nella coda per poi essere prelevato da uno dei thread che compiono il lavoro.
I thread nel loro ciclo prendono un descrittore dalla coda, ne elaborano le richieste e alla fine se questo non ha più niente da comunicare, disconnettono l' utente che lo stava usando dal server e chiudono la connessione con quel descrittore, altrimenti comunicano al *main* tramite pipe che su quel descrittore di file arriveranno altre richieste.
Se la coda in un determinato momento è vuota i thread si sospendono su una variabile di condizione e attendono che il *main*, quando ne inserisce uno in coda, esegua una *signal* per risvegliarli e consentirgli di prendere un descrittore con cui lavorare.
Quando vengono svegliati comunque la prima cosa che fanno è controllare se nel frattempo è arrivato un segnale di terminazione, visto che possono essere svegliati anche dal gestore dei segnali con una *broadcast*, e in quel caso interrompono devono interrompere subito il ciclo e terminare.

4. Gestione della terminazione

La terminazione del programma può avvenire soltanto tramite la ricezione di un segnale e in particolare con *SIGINT*, *SIGQUIT* e *SIGTERM,* che sono catturati e gestiti, viene terminato in maniera regolare.
Quando arriva uno di questi segnali il gestore dei segnali setta a 1 la variabile globale *segnale_arrivato*, inizialmente settata a 0, e sveglia tutti i thread in modo che sia questi che il *main* interrompano il proprio ciclo.

A questo punto i thread chiudono la pipe che usavano in scrittura e terminano mentre il *main* deve liberare tutta la memoria occupata al momento della chiusura, ma prima di farlo, per evitare errori, attende in un ciclo, con la procedura *join*, che tutti i thread siano terminati.
Una volta terminati tutti i thread viene liberata tutta la memoria occupata dagli utenti e dai gruppi iscritti fino a quel momento con le relative procedure, vengono chiusi i descrittori delle connessioni, viene liberata la memoria occupata dai nomi presi dal file di configurazione, dal nome della pipe e dalla coda dei descrittori e viene eliminato il socket usato per la comunicazione tramite pipe creato dal server nella cartella dove stanno i file temporanei.

5. Gestione della concorrenza

La tabella hash contenente gli utenti iscritti è una struttura dati condivisa a cui tutti i thread devono avere accesso per modificarla oppure per estrarne delle informazioni, quindi deve essere acceduta in mutua esclusione e per farlo non ho voluto usare un numero eccessivo di mutex per non aumentare troppo l' overhead di tempo e memoria dovuto alla gestione della concorrenza.
Ne vengono usati 64, che sono un quarto delle caselle della tabella hash e quindi un semaforo ogni quattro caselle.
Allo stesso modo deve essere acceduta la tabella rappresentante i gruppi attivi, ma con un solo semaforo, come descritto nella sezione dedicata ai gruppi.
Le statistiche vengono aggiornate dopo essere entrati in un recinto di mutua esclusione tramite la variabile *mtxstats*.
Anche se non avrebbe causato problemi nel funzionamento del programma, aggiornandole in questo modo rimangono consistenti in ogni momento.
La variabile di mutua esclusione *mtxfiles* è una sola e vale per qualsiasi file quindi non possono essere eseguite contemporaneamente due o più operazioni su file, che siano invii o ricezioni, anche se su file diversi.
Sarebbe stato possibile evitare in parte questa situazione adottando la stessa strategia usata per l' accesso alla tabella hash degli utenti, in quel caso però valeva la pena introdurre un overhead per consentire più accessi in punti diversi in contemporanea essendo un' operazione eseguita sicuramente una o più volte durante la gestione di una richiesta del client da ognuno dei thread, in questo caso invece si tratta di un' operazione che viene eseguita con una probabilità molto minore e, nei vari test eseguiti, questa scelta non ha causato evidenti rallentamenti.
E' presente anche una variabile di condizione usata dai thread per sospendersi quando non trovano descrittori con cui lavorare in coda, in attesa di essere risvegliati dal *main* quando ne inserisce uno oppure dal gestore dei segnali quando riceve un segnale di terminazione.
Infatti la prima cosa che fa un thread quando viene risvegliato è controllare se il segnale è arrivato e in quel caso interrompere il proprio ciclo di esecuzione.
Infine anche per inserire o prelevare descrittori di connessione dalla coda è necessario accedere, tramite l' apposita variabile *mtxcoda*, in mutua esclusione.

6. Gestione dei segnali

Il segnale *SIGPIPE* viene subito ignorato, in seguito viene settata una signal mask in modo da mascherare solo *SIGUSR1*, *SIGINT, SIGTERM, SIGQUIT* e *SIGALRM*.
A questo punto viene creato un thread la cui funzione sarà soltanto quella di ascoltare i segnali.
Questo thread riceve dal *main* la signal mask alla creazione e si sospende sulla procedura *sigwait*.
Alla ricezione di uno dei segnali precedentemente elencati si sblocca e controlla quale ha ricevuto, se è *SIGUSR1* chiama la procedura per stampare le statistiche del programma, contenute in una struct, su un file nel formato richiesto, se è uno tra *SIGINT, SIGTERM* o *SIGQUIT* setta a uno la variabile dichiarata di tipo *volatile sig_atomic_t* per essere safe durante la gestione del segnale che fa sospendere il ciclo eseguito dai thread (gestore dei segnali e *main* compresi) facendo terminare il programma.
Se invece riceve un *SIGALRM* setta a uno la variabile, anch' essa di tipo *volatile sig_atomic_t*, che invece fa in modo che il *main*, che fa da thread listener, ascolti nella select anche le richieste pervenute dai thread del pool tramite la pipe.

7. Gruppi

La gestione dei gruppi è realizzata mediante una tabella hash come quella contenente gli utenti iscritti.
Le dimensioni sono più piccole di quest' ultima dato che in un utilizzo normale il numero di gruppi sarà molto minore rispetto a quello degli utenti, e per lo stesso motivo viene usata una sola variabile di mutua esclusione per accedere a questa tabella anziché una ogni quattro caselle della tabella come è stato scelto di fare per gli utenti.
Anche se non specificato nel testo, ho scelto di eliminare un gruppo, oltre che quando viene richiesto esplicitamente con un' operazione di deregistrazione, anche quando il creatore di un gruppo viene deregistrato, perché è invece specificato che il creatore di un gruppo sia l' unico con i permessi per eliminarlo e quindi, all' eliminazione di questo il gruppo rimarrebbe attivo senza più la possibilità di essere eliminato.

8. Test

I vari test forniti dai docenti, compreso *testgroups.sh*, sono stati eseguiti da me e hanno avuto un esito positivo, oltre che sulla macchina virtuale del corso con sistema operativo Xubuntu (con 1 o 2 processori assegnati), su questi sistemi:

- Ubuntu 16.04
- Debian 9

9. Script bash

Lo script bash, contenuto nel file *script.sh*, innanzitutto controlla che il numero di parametri passati come argomento al momento dell' invocazione sia accettabile oppure se tra gli argomenti è presente *-help*, opzione che mostra il modo corretto di eseguire lo script e che può essere specificata in qualsiasi punto della lista di argomenti, anche se sono più di quelli necessari al programma per funzionare.
In questa fase controlla anche che il parametro contenente il nome del file di configurazione sia il nome di un file esistente e non vuoto e che il numero passato come secondo argomento sia non negativo (può essere 0).
Fatto questo assegna i parametri passati, che sono il nome del file di configurazione e un intero non negativo, a delle variabili, per comodità e leggibilità del codice.
Inizia a scorrere il file riga per riga considerando solo i primi 7 caratteri e li confronta con i 7 che compongono la parola "DirName", quella cercata.
Una volta trovata sa di essere alla riga giusta, cioè quella che contiene il nome di directory cercato, e che lo conterrà precisamente dal diciannovesimo carattere in poi.
Una volta trovato il nome della cartella questa viene aperta e si controlla che non sia vuota.
A questo punto se il numero passato come argomento era zero il programma stampa il contenuto di questa cartella e termina, altrimenti con una *find* cerca i file che sono più vecchi di alcuni minuti, il cui numero è specificato dal secondo parametro passato al programma, e li mette in un archivio *.tar* per poi rimuoverli.
Finito questo procedimento comprime ancora l' archivio *.tar* nel formato *.gzip*.