Relazione progetto **CHATTERBOX**

per il modulo di laboratorio SOL a.a. 2018-2019

STUDENTE : **EMILIO PANTI**
MAT: **531844**

INDICE

1 - INTRODUZIONE

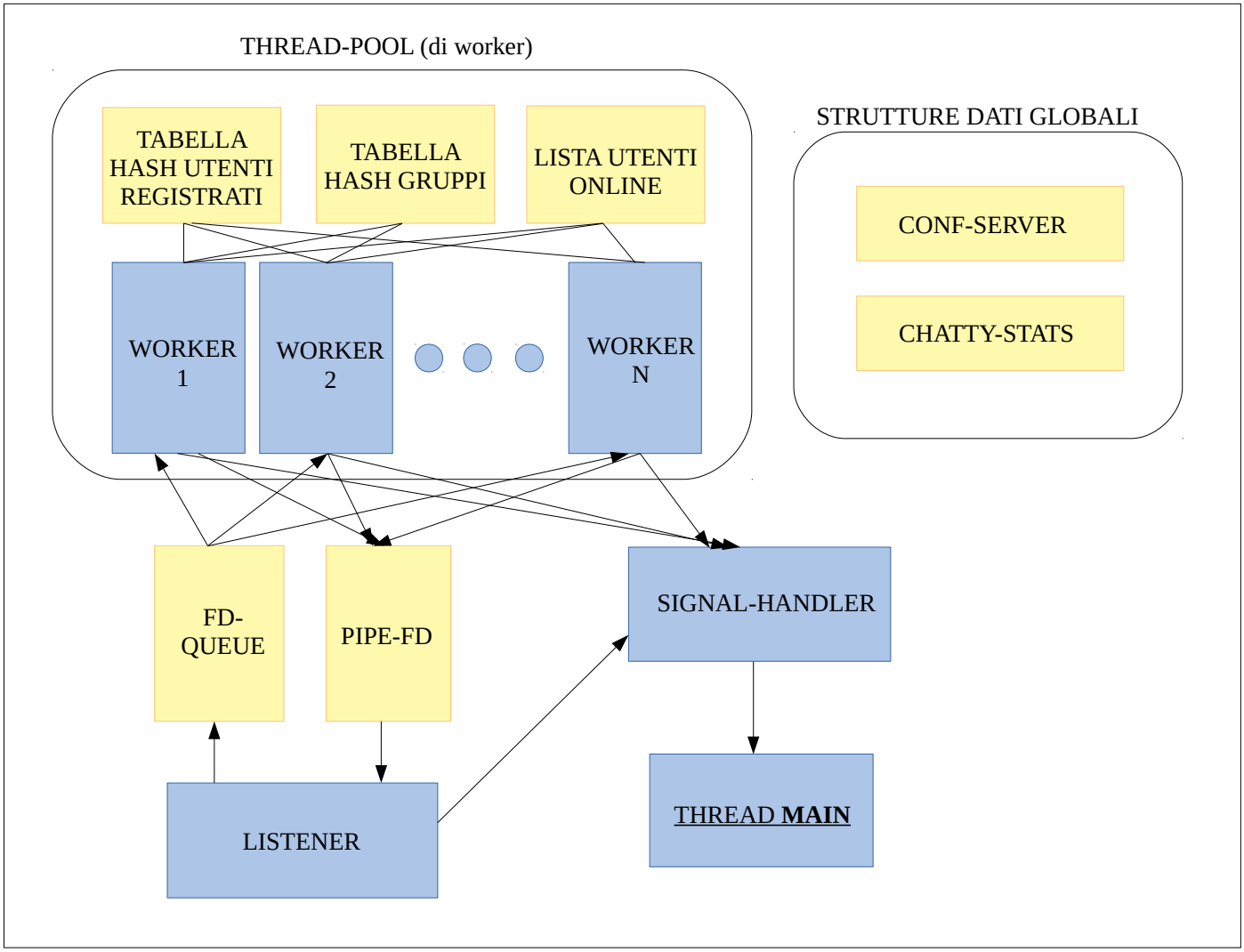
In questa relazione presenterò le scelte effettuate per la realizzazione del progetto e per le quali è stata lasciata libera iniziativa allo studente.

1.1 - SISTEMA OPERATIVO e TESTING:

Il progetto è stato sviluppato e testato completamente sul sistema operativo Ubuntu 18.04.1 LTS. E' stato testato anche sulla macchina virtuale (xubuntu) fornita durante il corso, sia avviandola con 2 processori che con 4.

1.2 - PANORAMICA DEL SERVER IN ESECUZIONE

Una volta terminata la fase di avvio del server ci troviamo nella seguente situazione (in giallo sono visibili le strutture dati, in celeste i threads attivi):



SERVER CHATTERBOX

2 - STRUTTURE DATI

Breve descrizione delle principali strutture dati usate dal server.

2.1 - CONF-SERVER (`typedef struct` **`configs_t`**)

E' la struttura dati destinata a contenere le configurazioni del server. E' globale e visibile a tutti. Questa scelta è dovuta al fatto che in più punti del progetto è necessario reperirne le informazioni all' interno. La struttura non prevede nessun meccanismo di mutua esclusione in quanto (una volta creata e definita dal thread main) è usata in sola lettura.

Per maggiori dettagli vedere i file `chatty.c` , `config.h` e `config.c`.

2.2 - CHATTY-STATS (`struct` **`statistics_t`**)

E' la struttura dati destinata a contenere le statistiche del server. E' globale e visibile a tutti. Il thread main la inizializza, il thread signal-handler vi accede in lettura per stampare le statistiche e i vari thread worker vi accedono in scrittura per aggiornarne i valori. Per evitare situazioni di inconsistenza è previsto un accesso in mutua esclusione (la relativa mutex è dichiarata nel file `chatty.c`).

Per maggiori dettagli vedere il file `stats.h`.

2.3 - FD-QUEUE (`typedef struct` **`fd_queue_t`**)

Struttura dati che contiene la coda degli fd (relativi ai client connessi al server) pronti a fare nuove richieste. Il thread listener si occupa di inserire nuovi fd mentre i thread worker si occupano di prelevarli ed eseguirne le richieste. Oltre ad un meccanismo di mutua esclusione tramite mutex è presente anche una variabile di condizione per segnalare l'inserimento di un nuovo fd da parte del listener. Così facendo i thread worker si bloccano quando trovano la coda degli fd vuota, evitando sprechi di risorse e lavoro.
Tale struttura è inizializzata nel thread main e poi passata (al momento della loro creazione) al listener e al gestore thread-pool (che la passerà ai vari worker).

NOTA: questa struttura è usata anche per la terminazione dei thread worker (leggere il paragrafo sulla terminazione del server per ulteriori spiegazioni).

Per maggiori dettagli vedere i file `fd_queue.h` e `fd_queue.c`.

2.4 - PIPE_FD (`typedef struct` **`pipe_fd_t`**)

Struttura dati che contiene gli fd (in lettura e in scrittura) di una pipe per effettuare comunicazioni extra al listener. Una comunicazione è composta da due parti: una operazione (che identifica lo scopo del messaggio) e un fd (per specificare a quale fd si riferisce l'operazione). Per evitare situazioni di inconsistenza è previsto un meccanismo di mutua esclusione per le letture/scritture sulla pipe.
I thread worker utilizzano questa struttura per comunicare al thread listener gli fd dei client che devono essere chiusi (se si sono disconnessi durante la elaborazione della loro richiesta) o che devono essere riaggiunti al set dei descrittori attivi (se la loro richiesta è stata portata a termine con successo).
Questa struttura è inizializzata nel thread main e poi passata (al momento della loro creazione) al listener e al gestore thread-pool (che la passerà ai vari worker).

NOTA: questa struttura è usata anche per la terminazione del thread listener (leggere il paragrafo sulla terminazione del server per ulteriori spiegazioni).

Per maggiori dettagli vedere i file `pipe_fd.h` e `pipe_fd.c`.

2.5 - THREAD-POOL (`typedef struct` **`hl_thread_pool_t`**)

E' destinato a contenere le informazioni che riguardano i thread worker e le strutture dati necessarie alla loro esecuzione. L' inizializzazione di questa struttura avviene nel thread main ed ha i seguenti effetti:

- creazione delle strutture dati condivise esclusivamente tra i workers, ovvero: la *tabella hash utenti registrati*, la *tabella hash gruppi* e la *lista utenti online,*
- avvio di N thread worker (N è preso dalla struttura globale *conf-server*). Ad ognuno di essi sono passati come argomenti il tid del signal_handler, la *fd-queue* , la *pipe-fd*, la *tabella hash utenti registrati*, la *tabella hash gruppi* e la *lista utenti online.*

Per maggiori dettagli guardare i file `threadpool.c` e `threadpool.h`.

2.6 - USER (`typedef struct` **`user_t`**)

Gli utenti registrati sono rappresentati da questa struttura, nella quale abbiamo:

- nickname dell'utente,
- fd: descrittore aperto verso il relativo client,
- status: ovvero se è online, offline o in fase di cancellazione,
- puntatore ad una mutex: da usare per operare in mutua esclusione su questa struttura,
- lista di messaggi (history),
- lista gruppi: contiene i puntatori ai gruppi a cui è iscritto l'utente.

NOTA1: un utente può inviare messaggi anche a sé stesso.

NOTA2: nella history sono presenti gli ultimi N messaggi inviati all'utente (indistintamente che li abbia ricevuti o meno), dove N è una informazione presa da *conf-server.*

Per maggiori dettagli guardare i file `user.c` e `user.h`.

2.7 - GROUP (`typedef struct` **`group_t`**)

I gruppi (di utenti) sono rappresentati da questa struttura, nella quale abbiamo:

- groupname: nome del gruppo,
- creatore: nickname dell'utente che ha creato il gruppo,
- status: ovvero se è attivo o in fase di cancellazione,
- puntatore ad una mutex: da usare per operare in mutua esclusione su questa struttura,
- lista dei membri: contiene i puntatori agli utenti iscritti al gruppo.

Per maggiori dettagli guardare il paragrafo in fondo alla relazione sulla parte facoltativa e i file `group.c` e `group.h`.

2.8 - HASH-TABLE (`typedef struct` **`hash_table_t`**)

Tabella hash (basata su liste di trabocco) per la gestione di dati generici. Al momento della creazione è necessario fornire due parametri importanti:

- **num_mutex** : ovvero il numero delle mutex che si vuole utilizzare per la mutua esclusione dei dati contenuti nella tabella
- **factor**: cioè quante liste di trabocco devono essere assegnate ad ogni mutex

NOTA1: gli elementi generici contenuti nella tabella hash ereditano la mutex dalla lista di trabocco in cui sono stati inseriti.

NOTA2: la **dimensione finale** della tabella hash è data dalla moltiplicazione (num_mutex * factor)

L'astrazione, rispetto agli elementi base contenuti, comporta la necessità di ulteriori informazioni al momento della creazione di queste tabelle hash (oltre ai due parametri visti sopra). Devono infatti essere fornite alcune funzioni per la gestione dei dati, in particolare:

- una funzione di comparazione,
- una funzione hash (per ricavare la posizione di inserimento),
- una funzione per settare la mutex di un nuovo elemento (che sarà la stessa della lista di trabocco in cui verrà inserito),
- una funzione di pulizia per liberare la memoria occupata da un elemento base.

Per maggiori guardare i file `abs_hashtable.c` e `abs_hashtable.h`.

TABELLA HASH UTENTI REGISTRATI

Per questa struttura è stata una usata una tabella hash generica come sopra descritta, i cui elementi base sono puntatori a strutture *user.* Invece di darle una dimensione fissa ho ritenuto opportuno creare una dipendenza con le configurazioni del server. Per questo motivo alla sua creazione (che avviene al momento della inizializzazione del gestore del thread-pool) ho fornito i seguenti parametri:

- **num_mutex** = numero massimo delle connessioni (dato presente nella struttura *conf-server)*
- **factor** = 10

Per le funzioni necessarie alla gestione degli utenti (passate anch'esse al momento della creazione di questa tabella) vedere i file `user.c` e `user.h`

TABELLA HASH GRUPPI

Per questa struttura è stata una usata una tabella hash generica come sopra descritta, i cui elementi base sono puntatori a strutture *group.* Invece di darle una dimensione fissa ho ritenuto opportuno creare una dipendenza con le configurazioni del server. Per questo motivo alla sua creazione (che avviene al momento della inizializzazione del gestore del thread-pool) ho fornito i seguenti parametri:

- **num_mutex** = numero massimo delle connessioni (dato presente nella struttura *conf-server)* / **2**
- **factor** = 10

Per le funzioni necessarie alla gestione dei gruppi (passate anch' esse al momento della creazione di questa tabella) vedere i file `group.c` e `group.h`

2.9 - LIST (`typedef struct` **`list_t`**)

Lista per la gestione di dati generici, che permette di personalizzare vari aspetti:

- politica di ordinamento,
- possibilità che operi in mutua esclusione fornendole una mutex,
- lunghezza massima (ha anche un valore di default).

L'astrazione rispetto agli elementi base contenuti comporta la necessità di alcune particolari informazioni al momento della creazione. Devono infatti essere fornite delle funzioni per la gestione dei dati, in particolare:

- una funzione di comparazione (per attuare la politica di ordinamento desiderata),
- una funzione di pulizia per liberare la memoria occupata da un elemento base.

NOTA: tutte le liste di dati utilizzate nel progetto si basano su questa struttura:

- liste di trabocco utilizzate nelle *tabelle hash*,
- lista dei messaggi e dei gruppi d'iscrizione nelle strutture *user*,
- lista dei membri nelle strutture *group*,
- *lista degli utenti online* (creata dal gestore thread-pool ed utilizzata dai worker).

Per maggiori guardare i file `abs_list.c` e `abs_list.h`.

3 - DESCRIZIONE THREADS

Il server chatterbox utilizza i seguenti 4 'tipi' di thread:

3.1 - THREAD MAIN

Il thread main (presente nel file `chatty.c`) si occupa della fase di avvio del server. In particolare ha la funzione di:

- leggere e salvare le variabili di configurazione (presenti nel file al momento del lancio del server) nella struttura globale *conf-server* (maggiori dettagli disponibili nei file `config.c` e `config.h`),
- creare le strutture condivise tra listener e worker, ovvero la *fd-queue*l e la *pipe-fd*,
- creare una maschera di segnali (`sigset_t`) da passare successivamente al thread signal-handler (che ne sarà il gestore). La maschera prevede i seguenti segnali:
  - SIGUSR1 – per stampare le statistiche
  - SIGINT, SIGTERM e SIGQUIT – per far terminare il server in modo corretto ed intenzionale
  - SIGPIPE – da ignorare, per evitare che una operazione di `write` possa determinare una terminazione del processo non voluta

    - SIGUSR2 – inviato dal listener o da un worker in caso di errore durante la loro esecuzione
- bloccare i segnali presenti nella maschera sopra citata (questo blocco viene ereditato anche dai threads creati successivamente),
- avviare il thread signal-handler, a cui viene passata la maschera dei segnali e la funzione clean-all() (presente in `chatty.c`) da chiamare nel caso riceva una richiesta di terminazione del server (sia voluta che causata da errore),
- creare il gestore del thread-pool, che avvierà un numero di workers definito dalle configurazioni ed inizializzerà le strutture dati condivise tra essi (*tabella hash utenti registrati*, la *tabella hash gruppi* e la *lista utenti online*). Al gestore del thread-pool viene passato il tid del signal-handler, la *fd-queue* e la *pipe-fd.* Queste informazioni saranno trasmesse a cascata anche ai vari workers,
- Avviare il thread listener, passando anche ad esso il tid del signal-handler, la *fd-queue* e la *pipe-fd*,
- Mettersi in attesa della terminazione del thread signal handler e salvarne lo stato finale,
- Terminare il processo con successo o con fallimento in base allo stato finale trasmesso dal signal-handler.

Se uno di questi passaggi non dovesse andare a buon fine viene chiamata la funzione clean-all() (definita in `chatty.c`) per ripulire l'ambiente fino a quel momento creato ed il processo del server viene terminato con errore.

3.2 - SIGNAL-HANDLER

Il signal-handler viene avviato dal thread main. Gli argomenti necessari al suo funzionamento sono una maschera di segnali da gestire (`sigset_t`) e una funzione di pulizia del server (da usare nel caso in cui venga richiesta la terminazione).
In particolare si occupa di:

- aprire il file delle statistiche in modalità 'append' per poter successivamente riportarne gli aggiornamenti (quando richiesti). Il nome del file per le statistiche è preso dalla struttura globale delle configurazioni *conf-server.* Se nelle configurazioni del server non è stato specificato nessun nome per tale file le eventuali richieste di stampa statistiche verranno ignorate,
- mettersi in **attesa ciclica di nuovi segnali** (tramite la chiamata della funzione `sigwait`) per poi gestirli. In base al segnale ricevuto ha comportamenti diversi:
    - SIGUSR1: appende al file delle statistiche (se è stato definito nelle configurazioni) i dati riportati nella struttura globale *chatty-stats* (vedere il file `stats.h` per maggiori dettagli)
    - SIGINT, SIGTERM o SIGQUIT: chiude il file delle statistiche (se era stato aperto) , chiama la funzione di pulizia del server e termina la sua esecuzione con successo
    - SIGPIPE: non fa niente, è solo da ignorare
    - SIGUSR2: chiude il file delle statistiche (se era stato aperto) , chiama la funzione di pulizia del server e termina la sua esecuzione con errore

Nel caso in cui, durante la sua esecuzione, si verifichi un errore il signal-handler chiama la funzione di pulizia del server e termina la sua esecuzione con errore.

NOTA: Lo stato di terminazione di questo thread è importante al fine di capire (nel thread main) se il server abbia dovuto cessare la sua esecuzione a causa di un errore o per volontà dell' utente gestore.

Per maggiori dettagli guardare i file `signal_handler.c` e `signal_handler.h`.

3.3 - LISTENER

Il listener viene avviato dal thread main. Gli argomenti necessari al suo funzionamento sono il tid del signal_handler, la *fd-queue* e la *pipe-fd.* Si occupa di:

- prendere dalla struttura dati *conf-server* il socket-path ed il numero massimo di connessioni,
- creare un socket AF-UNIX per l'ascolto di nuove connessioni (utilizzando il socket-path preso sopra),
- creare un set di fd (**`fd_set`**) per la gestione dei file descriptor relativi ai vari client. A questo insieme vengono aggiunti anche l'fd del socket in ascolto (per ricevere nuove connessioni) e l'fd aperto in lettura della *pipe_fd* (per ricevere gli aggiornamenti dai vari workers)

Fatto ciò (se non è occorso nessun errore) chiama **ciclicamente** la funzione **`select()`** su una copia di `fd_set` per selezionare gli fd pronti a comunicare. In base al tipo di fd pronto esegue operazioni diverse, infatti:

- se è l'fd relativo al socket in ascolto vuol dire che abbiamo una nuova richiesta di connessione. Se non è stato raggiunto il numero massimo di connessioni viene accettata e l'fd relativo al client viene aggiunto all' `fd_set`, altrimenti viene scartata (per maggiori informazioni leggere la parte relativa al protocollo listener/client che segue),
- se è l'fd relativo alla *pipe-fd* vuol dire che abbiamo una nuova comunicazione (composta da un operazione **op** ed un fd che chiamiamo **fdx**) da leggere sulla pipe. Se:
  - op = UPDATE: significa che è stata eseguita con successo la richiesta del client con fd = fdx da parte di un worker e che quindi tale fd deve essere ri-aggiunto nuovamente all' `fd_set` per poter ascoltare nuove richieste
  - op = CLOSE: significa che il client con fd = fdx si è disconnesso durante l'esecuzione della richiesta fatta in precedenza. In questo caso il listener chiude fdx e diminuisce di uno il numero delle connessioni
  - op = **TERMINATE**: significa che il listener deve terminare la sua esecuzione (prima di ciò chiude tutti gli fd relativi ai client ancora aperti)
- se invece è un fd relativo ad un client già connesso vuol dire che esso è pronto a fare una nuova richiesta al server. L'fd in questione viene rimosso dall' `fd_set` ed inserito nella *fd_queue* affinché un thread worker si occupi della sua richiesta.

TERMINAZIONE LISTENER:

Un thread worker termina:

- (con successo) quando legge dalla *pipe-fd* l'operazione **TERMINATE**.
- (con fallimento) se durante la sua esecuzione si dovesse verificare un errore. In questo caso il thread listener invia un segnale (del tipo **SIGUSR2**) al thread signal-handler (di cui conosce il tid perché passatogli come argomento) e poi termina.

In entrambi i casi prima di terminare chiude tutti gli fd aperti relativi ai client connessi.

Per maggiori dettagli sul thread listener guardare i file `listener.c` e `listener.h`.

3.4 - WORKER

I threads worker sono attivati al momento della creazione della struttura del gestore thread-pool. Gli argomenti di cui hanno bisogno per svolgere la loro attività sono il tid del signal_handler, la *fd-queue* , la *pipe-fd*, la *tabella hash utenti registrati*, la *tabella hash gruppi* e la *lista utenti online.*
Un generico worker si occupa **ciclicamente** di:

- prelevare un fd relativo ad un client pronto a fare una nuova richiesta dalla *fd-queue* (nel caso in cui non ci fossero fd pronti rimane in attesa che ne arrivi uno),
- controllare se a tale fd corrisponde un utente online facendo una ricerca nella *lista utenti online.* Se la ricerca ha successo la struttura di tale utente viene lockata. Questo è per far sì che la lettura della richiesta (passo successivo) avvenga in mutua esclusione e che nessun altro worker scriva o legga dal solito fd in contemporanea, creando così situazioni di inconsistenza,
- leggere la richiesta del client,
- rilasciare la mutex relativa alla struttura dell' utente se al passo precedente era stata lockata,
- eseguire la richiesta (se lecita),
- comunicare sulla *pipe-fd* (collegata al thread listener) l'esito della esecuzione della richiesta. Se tutto è andato bene viene inviata una operazione UPDATE, altrimenti (se il client si fosse disconnesso) una operazione di CLOSE. Oltre all' operazione viene inviato anche l'fd relativo al client che aveva fatto la richiesta,

TERMINAZIONE WORKER:
Un thread worker termina:

- (con successo) quando preleva un **fd == -1** dalla coda *fd_queue.*
- (con fallimento) se durante la sua esecuzione si dovesse verificare un errore. In questo caso il thread worker invia un segnale (del tipo **SIGUSR2**) al thread signal-handler (di cui conosce il tid perché passatogli come argomento) e poi termina.

GESTIONE FILES:

I threads worker sono responsabili dei files che vengono inviati/scaricati dagli utenti. Per gestirli utilizzano le funzioni presenti nei files `files_handler.h` e `files_handler.c`.

COMUNICAZIONI CON I CLIENT

Per leggere/scrivere messaggi da/verso i client i thread worker utilizzano le funzioni presenti nei file `message.c`, `message.h`, `connections.c` e `connections.h`.
Inoltre le comunicazione riguardanti un client a cui è associato un utente online avvengono lockando la mutex della relativa struttura *user* per poter garantire scritture/letture consistenti.

Per maggiori dettagli sul thread worker guardare i file `worker.c` e `worker.h`.

4 - TERMINAZIONE DEL SERVER

Il processo del server chatterbox può terminare per due motivi:

- l'utente 'amministratore' decide di terminarlo inviando un segnale **SIGINT, SIGTERM o SIGQUIT**. In questo caso tale segnale viene 'catturato' dal thread signal-handler che si occupa di chiamare la funzione di pulizia del server e terminare la sua esecuzione (la cui fine che è attesa dal thread main per terminare l'intero processo)
- si verifica un errore durante l' esecuzione di un thread interno al server:
  - thread main: se avviene durante la fase di avvio del server è lo stesso thread main a chiamare la funzione di pulizia del server e a far terminare l'intero processo
  - signal-handler: in caso di errore chiama la funzione di pulizia del server e termina la sua esecuzione (la cui fine è attesa dal thread main per terminare l'intero processo)
  - listener/worker: in caso di errore inviano un segnale **SIGUSR2** al signal-handler e terminano la propria esecuzione. La gestione di tale segnale prevede che il signal-handler chiami la funzione la funzione di pulizia del server e termini anch 'esso la sua esecuzione (la cui fine è attesa dal thread main per terminare l'intero processo)

Lo stato di terminazione del processo del server (**successo/fallimento**) dipende direttamente da quello del thread signal-handler; il quale:

- è uguale a 0 se è terminato per volontà dell'utente amministratore
- è diverso da 0 quando è occorso un errore

FUNZIONE DI PULIZIA SERVER

La funzione in questione (chiamata **clean_all()**) è definita nel file `chatty.c` e si occupa di:

- far terminare i threads worker: per fare ciò inserisce N volte un fd == **-1** nella *fd-queue*, dove N è il numero di worker attivi (informazione presente in *conf-*

server). Quando un worker preleva **-1** da tale coda capisce di dover cessare la sua esecuzione,
- far terminare il thread listener: per raggiungere tale scopo scrive nella *pipe-fd* una comunicazione con operazione == **TERMINATE**. Quando il lestener legge tale messaggio capisce di dover cessare la sua esecuzione,
- deallocare tutte le risorse usate dalle strutture dati del server.

5 - PARTI OPZIONALI ED AGGIUNTIVE

5.1 - CANCELLAZIONE GRUPPI

Oltre alla gestione dei gruppi e l' implementazioni delle operazioni base previste per essi (CREATEGROUP_OP, ADDGROUP_OP e DELGROUP_OP) , ho aggiunto anche la funzione per la cancellazione di un gruppo.
Le assunzioni che ho fatto sulla gestione dei gruppi sono che:
- solo il creatore di un gruppo può richiederne la cancellazione,
- se il creatore di un gruppo si rimuove da quest' ultimo il gruppo stesso viene cancellato,
- se un utente si de-registra vengono cancellati anche tutti i gruppi di cui è il creatore.

Nel file `ops.h` ho aggiunto due operazioni :
- **CANCGROUP_OP:** per la richiesta di cancellazione gruppo,
- **OP_NO_CREATOR**: messaggio di errore nel caso in cui un utente richieda l'eliminazione di un gruppo di cui non è il creatore.

Ho ritenuto opportuno aggiungere anche un altra statistica nella struttura *chatty-stats* (file `stats.h`) per tenere il conto dei gruppi utenti attivi nel server.

5.2 - TESTGROUP2.SH

Script usato per testare la funzione di cancellazione gruppi ed assicurarsi che le operazioni riguardanti i gruppi non generino memory leaks. Dato che il client non prevede l' operazione di cancellazione ho testato questa funzionalità in modo indiretto, eliminando utenti che erano creatori di gruppi o rimuovendo da un gruppo il suo creatore.

5.3 - MAKEFILEPLUS

Ha tutte le funzionalità del Makefile normale con l'aggiunta di due test aggiuntivi:
- **test6** per eseguire lo script testgroup.sh fornito dai docenti,
- **test7** per eseguire lo script testgroup2.sh descritto sopra.

5.4 - CLEANER_DIRFILE.SH

E' lo script bash (richiesto dalle specifiche) per pulire ed archiviare i file che vengono inviati dagli utenti.