Progetto del modulo di Laboratorio di Programmazione Concorrente di Sistema, corso A, a.a. 2009/10

Federico Mariti

Indice

1 Organizzazione del codice **2**

2 Tipi di Dati Astratti implementati **2**
- 2.1 Generic List . . . 4
- 2.2 Generic Hash Table . . . 4
- 2.3 Blocking List . . . 4
- 2.4 Thread Pool . . . 5

3 Struttura del Server **6**
- 3.1 Strutture dati usate e condivisione delle stesse tra i threads . . . 6
- 3.2 I threads del processo . . . 6
- 3.3 Comunicazione e sincronizzazione tra threads . . . 8
- 3.4 Scrittura dei messaggi di log in logfile . . . 8
- 3.5 Invio di messaggi su una connessione . . . 8
- 3.6 Gestione dei segnali . . . 11
- 3.7 Terminazione del processo e dei thread . . . 11

4 Struttura del Client **12**
- 4.1 I threads del processo . . . 12
- 4.2 Terminazione del processo . . . 12

A Appendice: Guida per l'utente **13**
- A.1 Compliazione . . . 13
- A.2 Installazione e disistallazione . . . 13
- A.3 Esecuzione del Server e del Client . . . 14

B Appendice: Debug e Test **14**

1 Organizzazione del codice

Tutti i file sorgenti dei programmi `msgserv`, `msgcli` e dei programmi di test sono posizionati nella directory `PROJ_BASEDIR/src/`. Oltre alla libreria `msg` sono state implementate altre due librerie: `blockingCollection` e `threadPool`.

La libreria `blockingCollection` dovrebbe contenere un insieme di tipi di dato astratti che realizzano strutture dati di utilità con la caratteristica di fornire, nelle funzioni di manipolazione di tali oggetti[1], sincronizzazione implicita ed esplicita dei threads che ne fanno uso. In realtà tale libreria contine solo l'oggetto "`blockingList`" implementato come elementi (nodi) linkati. Tale struttura dati è molto usata nello svolgimento del progetto come pila/-coda per realizzare l'interazione tra più threads secondo il modello *produttore-consumatore*.

La libreria `threadPool` realizza un oggetto esecutore di un insieme di attività (task) asincrone sottomesse usando un insieme arbitrario di threads. Tale oggetto provvede a risolvere due problemi spesso ricorrenti in un processo "servente":

- Fornisce un limite superiore alle risorse (threads) utilizzate durante l'esecuzione dell'insieme di compiti sottomessi;
- In generale, rispetto ad una gestione esplicita dei thread, migliorano le prestazioni durante l'esecuzione di un numero elevato di task sottemessi, in quanto i threads del pool sono riutilizzati per l'esecuzione di più attività sottomesse in modo da non pagare il costo (elevato) di creazione ed avvio di un thread per ogni esecuzione di un task sottomesso.

Per tali caratteristiche questo oggetto è usato nell'implementazione del programma `msgserv` per sottomettere la gestione di una nuova connessione a nuovi threads. Tuttavia l'implemntazione "da zero" di un tale oggetto è motivata piuttosto da fini didattici che effettiva esigenza in quanto le caratterstiche sopra descritte sarebbero agilmente ottenibili (più semplicemente) con una stutturazione ad-hoc per il programma dei threads e delle strutture dati che compongono un thread pool. Inoltre molte caratterische del thread pool non sono usate e pertanto non implementate o testate, è tuttavia fornita una valida interfaccia (i prototipi delle funzioni e specifiche) e definizioni i tipi necessari (come l'oggetto per memorizzare il risultato pendente di un task).

Per quanto riguarda i tipi di dato astratto implementati e non richiesti dal progetto è stata fornita una implementazione che nasconde l'implementazione fornendo all'utente solo i prototipi delle funzioni pubbiche con cui utilizzare gli oggetti in questione. Ciò non permette una allocazione di uno di questi oggetti in memoria stack in quanto non è nota la dimensione della struttura che descrive l'oggeto. Per tale motivo non sono stati modificati gli oggetti `genList` e `genHashTable` richiesti dal progetto.

2 Tipi di Dati Astratti implementati

I tipi di dati astratti passivi implementati rappresentano strutture dati quali *liste* e *tabelle hash*. È stato inoltre implementato il tipo di dato astratto *thread*

[1] Per "oggettto" si intede tipo di dato astratto, passivo (una struttura dati) o attivo (capacità di autocontrollo)

pool che rappresenta un esecutore di task (o attività) asincroni tramite un insieme di thread opportunamente gestito. Un elenco dei tipi di dato significativi implementati:

list_t lista generica di coppie (chiave, valore)

hashTable_t tabella hash generica

blockingList_t lista generica bloccante

threadPool_t esecutore di task asincroni secondo il modello a pool

Per garantire la proprietà di *information hiding* sui tipi di dato astratto, l'implementazione di questi è strutturata in due file header e in uno o più file sorgenti. Il primo file header prende il nome del tipo di dato astratto e contiene la dichiarazione delle funzioni *pubbliche* che agiscono sull'oggetto e la definizione di macro di utilità. Il secondo file header prende il nome del tipo di dato seguito da "_private" e contiene la definizione delle strutture che implementano il dato ed altre informazioni *riservate* alla sola implementazione del dato e non necessarie all'utente. All'utente vengono forniti il file header pubblico e la libreria, così da nascondere le scelte implementative del tipo di dato. La scelta di garantire tale proprietà e fornire delle valide interfaccie è motivata dal fatto per cui l'uso di un oggeto tramite l'accesso diretto ai campi delle strutture che lo costituisco può provocare comportamenti non attesi/definiti in generale e soprattutto nel caso in cui venga cambiata l'implementazzione del tipo di dato astratto.

Per quanto riguarda le strutture quali liste e tablla hash, per ogni funzione f che agisce sugli elementi della struttura è definita (nell'header pubblico) una macro parametrica che definisce un'altra funzione, wrapper di f, il cui prototipo non ha riferimenti generici `void *` ma vengono specificati i tipi passati come parametri alla macro. Il comportamento della funzione wrapper è identico alla funzione generica. L'utilizzo di tali macro è a discrezione dell'utente e se adeguato permette di garantire alcune proprietà aggiuntive, quali:

- warining di tipo a tempo di compilazione nell'invocazione non corretta di una funzione;
- utilizzo di una lista o tabella hash con elementi aventi valori (e chiavi) di tipo omogeneo;
- utilizzo di un sottoinsieme di funzioni per una lista che offere l'astrazzione di più modelli di manipolazione dati quali LIFO (pile) e FIFO (code).

Ad esempio per la funzione (in blockingList.h):

```
int blockingList_push(blockingList_t * l, void * valuep);
```

sia ha (in blockingList.h) la macro:

```
#define define_blockingList_push(wrapperFunName, valuepType)   \
  int wrapperFunName(blockingList_t * l, valuepType valuep) {  \
    return blockingList_push(l, valuep);                       \
  }
```

l'utente può usare tale macro nel proprio codice per definire una funzione che agisce su una lista bloccante di interi, nel seguente modo:

```
...
#include "headerPath/blockingList.h"
define_blockingList_push(myIntList_push, int)
...
{
  blockingList_t * myList =
    blockingList_new(0, copy_int, cmpr_int);
  myIntList_push(myList, 123);
  ...
}
...
```

2.1 Generic List

Lista di coppie (chiave, valore), entrambi gli elementi della lista sono di tipo generico.

Non è *thread-safe*. Accessi e manipolazioni da parte di più thread concorrenti sulla stessa lista provocano comportamenti non prevedibili.

Usata per implementare le liste di collisione nella tabella hash generica.

2.2 Generic Hash Table

Una tabella hash di chiavi e valori generici. Il probblema delle collisioni di chiavi con stessa indice hash è risolto con le liste di collisione. Non è implementato il ridimensionamento della tabella in base al fattore di carico.

Non è *thread-safe*. Accessi e manipolazioni concorrenti della stessa tabella provocano comportamenti non prevedibili, per tali situzioni è necessario assicurare la mutua esclusione nelle operazioni che modificano la struttura interna della tabella.

Usata per implementare la tabella degli utenti autorizzati nel processo `msgserv`.

2.3 Blocking List

Lista bloccante di elementi con tipo qualsiasi ed eventualmente disomogenei. Con il termine bloccante si intende che le funzionaità di estrazione di un valore da tale struttura sono caratterizzate dall'attesa che la lista sia non vuota, e le operazioni di inserzione di un valore attendono la disponibilità di spazio nella lista. L'implementazione è garantita *thread-safe* ovvero, tutti i metodi che modificano la struttura interna della lista sono eseguiti in modo atomico usando un lock interno o altre forme per controllare l'esecuzione concorrente del codice.

Il modello usato per garantire la correttezza in caso di un uso concorrente dell'oggetto è quello tipico Java: la struttura dati bloccante (la lista in questo caso) contine un lock di tipo `pthread_mutex_t` per garantire l'uso esclusivo dell'oggetto e un singolo monitor di tipo `pthread_cond_t` per realizzare la coda sulle due condizioni "lista non vuota" e "spazio disponibile" e i meccanismi di sospensione e sveglia. Attualmente il lock non è ricorsivo, ne prevede il controllo sull'identificatore del thread, tali caratteristiche non sono necessarie ai fini del progetto, ma possono essere agilmente sviluppate in un secondo momento vista la caratteristica di *information hiding* di tale struttura dati, creando un

nuovo tipo di dato wrapper di `pthread_mutex_t`, che ne utilizzi ed estenda le funzionalità, tale tipo di dato verrà usato per realizzare il lock interno alla lista.

L'interfaccia pubblica ovvero le funzioni e le macro con le quali l'utente adopera la struttura dati, è definita nell'header `blockingList.h`, le strutture che costituiscono la lista sono invece definite nell'header `blockingList_private.h` non visibilie all'utente. L'interfaccia pubblica prevede solo la definizione del tipo di dato `blockingList_t` e la dichiarazione delle funzionini con le quali adoperare la struttura.

L'iterazione degli elementi di tale lista può essere effettuata con il corrispondente tipo di dato iteratore `blockingList_itr_t` definito nell'header pubblico `blockingList_itr.h` il quale definisce i classici metodi per iterare la lista.

Le funzioni `blockingList_lock` e `blockingList_unlock` presenti nell'header pubblico della lista assicurano l'accesso esclusivo alla lista e a tutte le funzionalità per un blocco di codice arbitrario. Occorre porre attenzione al fatto che (attualmente) il mutex della lista non è ricorsivo, quindi l'invocazione di una funzionalità che richieda l'accesso esclusivo alla lista, successiva ad una chiamata del lock esplicito causerà un *dead-lock*. Tali funzioni di lock e unlock sono comunque utilizzabili per accedere esclusivamente alla lista durante una iterazione della stessa.

Tale oggetto è spesso usato per effettuare la comunicazione tra thread dello stesso processo nell'implementazione del programma `msgserv` e del tipo di dato astratto `threadPool_t`

2.4 Thread Pool

Esecutore di attività asincrone, con e senza valore di ritorno, tramite un insieme di threads lavoratori opportunamente autogestito. Tali threads sono in grado di eseguire in sequenza più task sottomessi al thread pool così da minimizzare il costo di overhead dovuto alla creazione di un thread rispetto ad una gestione esplicita dei threads. Le politiche di gestione dell'insieme dei threads sono due:

fixed è fissata la dimensione massima e minima dell'insieme dei thread;

cached il tempo di inattività dei threads è limitato.

I threads che costituiscono tale tipo di dato astratto sono solamente i thread lavoratori (attivi o meno), non esiste un thread dispatcher. L'insieme dei threads lavoratori può variare nel tempo se è stato fornito un tempo massimo di inattività dei threads o in seguito all'invocazione di una funzionalità quali `threadPool_submit`, `threadPool_shutdown*`.

Se fissata una dimensione massima del pool di threads, un task sottomesso non viene eseguito immediatamente ma è inserito in una coda, in attesa che un thread attivo termini l'esecuzione di un task sottomesso precedentemente. Ad ogni modo, indipendentemente dal tipo *fixed* o *cached* del thread pool, l'attività di un thread lavoratore è eseguire ciclicamente l'attesa di un nuovo task dalla coda dei task sottomessi ed eseguire l'attività ed eventualmente (se richiesto) salvare il risultato. La coda dei task è implementata con un oggetto `blockingList_t`, in modo da garantire la correttezza nell'accesso concorrente alla lista e realizzare l'attesa passiva di un nuovo task.

La terminazione del thread pool è implementata in due modi:

- Aggiungendo uno speciale task in coda alla lista dei task sottomessi. Tale task non fa altro che invocare la funzione `pthread_exit((void *) 0)`. In tal modo viene realizzata la terminazione graduale del thread pool (vedi specifiche di `threadPool_shutdown`};
- Cancellando tutti i thread o tutti i thread non attivi. In questo modo viene implementata la terminazione itantanea o la terminazione "finisci l'esecuzione", vedi le specidiche di `threadPool_shutdownNow` e `threadPool_shutdownDoActive`);

3 Struttura del Server

3.1 Strutture dati usate e condivisione delle stesse tra i threads

Vengono elencate e brevemente descritto lo scopo delle strutture dati usate; nei pargrafi successivi vengono approfonditi e motivati gli usi di tali oggetti, in particolare di quelli non menzionati nelle specifiche/consegna del progetto.

Thread Pool per servire le richieste dei clienti connessi;

Tabella degli utenti (autorizzati) mantenere lo stato {"connesso", "non connesso"} degli utenti e la coda dei messaggi in uscita verso l'utente, se quest'ultimo è connesso;

Lista degli utenti connessi per mantenere l'elenco degli utenti connessi, in modo da non dover iterare tutta la tabella degli utenti ad ogni richiesta della lista delgi utenti da parte di un client;

Pila dei messaggi di log

Coda dei messaggi in uscita su una connessione una per utente connesso.

Le strutture {ThreadPool, Tabella utenti, Lista utenti connessi, Pila messaggi log} sono create in fase di inizializzazione del processo e sono riferite da una variabile dichiarata in ambiente globale, quindi tali oggetti sono condivisi tra tutti i thread del processo. La {Coda di messaggi in uscita} invece viene creata solo alla connessione di un utente (il nome) alla chat, tale struttura viene riferita nella Tabella utenti, perciò viene condivisa tra tutti i thread del processo.

3.2 I threads del processo

Di seguito vengono chiamati *worker-thread* i thread del processo `msgserv` la cui attività è caratterizata dal servire le richieste di uno o più utenti connessi. Un worker-thread può essere associato, ovvero servire, una sola o più (tutte) connessioni a seconda della stuttura che si vuole dare al server. Un worker-thread sarà chiamato *scrittore* (mittente) o *lettore* (ricevente) a seconda del servizio che svolge sulla connessione a cui è associato.

Si è scelto di strutturare il server in modo tale per cui ogni connessione ha *associati* due worker-thread, uno lettore e l'altro scrittore; per *associato in lettura/scrittura* si intentde (ora ed in futuro) che la connessione è servita in letttura/scrittura *unicamente* dal worker-thread corrispondente; l'*invio* di un

messaggio unicast/broadcast che un worker-thread lettore riceve dal rispetivo utente è implementato con la comunicazione del thread lettore stesso con il worker-thread scrittore associato alla connessione dell'utente destinatario. Le motivazioni di tale strutturazione sono date nel paragrafo **(3.5)**. Il processo `msgserv` è quindi formato dall'insieme dei worker-thread di cardinalità doppia al numero di connessioni, dal thread scrittore sul file di log e dal thread principale o dispatcher.

La vita dei thread dispatcher e scrittore sul file di log corrsiponde alla vita del processo. I worker-thread sono gestiti dall' oggetto Thread pool quindi possono eseguire più attività nella loro vita, nello specifico possono esssere riceventi o scrittori di più utenti; la vita di un worker-thread dipende dal Thread pool se è prevista un morte a causa di inattività.

Le attività (task) che descrivono i thread di tale processo hanno in comune il modello "servente" che prevede tre fasi:

Inizializzazione di strutture dati e variabili;

Servire la prossima richiesta fino alla terminazione del thread viene reperita una richiesta sottomessa da un cliente e viene servita;

Terminzione vengono liberate le risorse e vengono effettualte le ultime interazioni con altri thread.

Ovviamente la definizione di "terminazione", "cliente" ed "interazione con il cliente o altro thread" varia a seconda dell'attività .

Lo scopo del thread dispatcher è di inizializzare le principali strutture dati, avviare il thread scrittore dei messaggi di log e restare in attesa di richieste di connessione sul socket di ascolto collegato al path prestabilito. Dopo aver stabilito una connessione[2] con un cliente viene sottomesso alla struttura threadPool il compito di eseguire il task del worker-thread lettore con argomento il file descriptor del socket attivo appartenente alla nuova connessione. Tale nuovo worker-thread ha il compito iniziale di attendere il messaggio di login contenente il nome utente del cliente secondo il protocollo definito in `comsock.h` e nelle specifice del progetto, e di verficare l'autenticazione di tale nome. Solo dopo l'autenticazione, l'utente si considera collegato alla chat, il worker-thread lettore procede a creare una coda dei messaggi e impostarla nella Tabella utenti autorizzati al nome utente ricevuto. Tale coda rappresenta i messaggi in uscita verso l'utente ed è passata come argomento, con il file descriptor del socket appartenente alla connessione, ad un nuovo worker-thread scrittore il cui task è sottomesso al threadPool dallo stesso worker-thread lettore. Prima di iniziare a ricevere messaggi dall'utente, il worker-thread lettore deve solo aggiungere il nome utente alla lista degli utenti connessi. Il worker-thread scrittore creato da un lettore dopo l'autenticazione di un utente ha il compito iniziale di inviare la conferma di avvenuta connessione secondo il protocollo stabilito, quinidi si mette in attesa di un messaggio disponibile nella coda dei messaggi in uscita verso l'utente ricevuta come argomento. L'attesa di un nuovo messaggio e' bloccante, una volta disponibile un messaggio vengono effettuati controlli sul tipo del messaggio e sulla lunghezza del buffer, quindi viene spedito all'utente. È quindi compito di ogni worker-thread lettore, una volta ricevuto un messaggio di tipo

[2]Trattasi di connessione logica tra i due processi secondo l'internal protocol di UNIX, non la connessione di un utente (il nome) alla chat.

{TO_ONE, BCAST} creare il o i messaggi da sottmettere al o ai worker-threads scrittore e la o le stringhe da sottomettere al thread scrittore dei messaggi di log. Come avvengono tale sottomissioni di servizi interni al processo, ovvero come i threads comunicano/cooperano, è descritto nei paragrafi successivi.

3.3 Comunicazione e sincronizzazione tra threads

L'interazione tra threads avviene tramite risorse condivise quali pile o code. In generale tali risorse o "canali di comunicazione" hanno $n \geq 1$ threads produttori e $m \geq 1$ threads consumatori, è pertanto necessario imporre vincoli ai threads stessi per quanto riguarda la competizione alle risorse condivise e la cooperazione. Ciò è effettuato (nella maggior parte dei casi) in modo trasparente usando la struttura dati bloccante `blockingList_t` descritta nel paragrafo **(2.3)** che incapsula oggetti lock e monitor così da garantire le proprietà di mutua esclusione sull'oggetto e di cooperazione tra threads. Fa eccezione la risorsa Tabella utenti autorizzati la quale, in generale, è acceduta in scrittura da più worker-threads; tali threads, quindi, necessitano del vincolo di mutua esclusione nell'accesso alla tabella. Tale vincolo non è fornito dalla struttura dati `hasTable_t` che realizza la tabella ma è garantito esplicitamente da un lock di tipo `pthread_mutex_t` dichiarato `static` in una funzione visibile a tutti i thread del processo. Tale funzione permette di inizializzare, bloccare, sbloccare e distruggere il mutex, inoltre garantisce proprietà aggiuntive in modo da avere il ritorno di errori piuttosto che comportamenti non attesi (vedi documentazione).

3.4 Scrittura dei messaggi di log in logfile

Il compito di eseguire la scrittura dei messaggi sul file di log viene effettuta da un unico thread specializzato a fare ciò. Un messaggio letto da un socket vine scritto sul file di log grazie all'interazione tra il worker-thread lettore che ha ricevuto la richiesta di invio del messaggio e tale thread. La comunicazione tra questi due threads avviene tramite una pila (bloccante), l'uso di una struttura LIFO è motivato dalla fatto che non è richiesto l'ordine in cui i messaggi compaiono nel file e dal breve tempo (il minimo possibile) di inserzione di un messaggio da parte di un worker-thread lettore. Si nota che il tempo minimo di inserzione sarabbe disponibile anche con un'ordinazione FIFO su una coda a due teste; implementare un tale struttura dati è un possibile upgrade in quanto torna utile nella comunicazione dei worker-thread lettori e scrittori, dove è necessario un ordinamento FIFO dei messaggi inviati.

3.5 Invio di messaggi su una connessione

L'invio di un messaggio, da parte di un worker-thread, ad un utente richiede la scrittura di tale messaggio nel socket attivo corrsipondente alla connessione dell'utente, tale scrittura non deve essere interrotta da scritture di altri worker-thread che eseguono l'invio di un messaggi. Si può pensare a due strutturazioni dei worker-thread che risolvono tale problema in due modi diversi:

- un qualsiasi socket attivo è scritto da un unico worker-thread. Un worker-thread che esegue l'invio di un messaggio deve interagire con il worker-thread associato alla connessione destinataria. L'interazione tra threads avviene come descritto nel paragrafo **(3.3)**;

- un qualsiasi socket attivo può essere scritto da un numero arbitrario di worker-thread (tutti), ovvero un qualsiasi worker-thread può scrivere direttamente sul socket attivo destinatario. Occorre garantire la mutua esclusione nella scrittura su diun socket da parte dei worker-thread.

Di seguito vengono analizzate alcune soluzioni a tale problema per motivare la scelta fatta.

3.5.1 Unico thread scrittore

Disporre di un solo thread che esegue l'invio di un messaggio, reperito da un worker-thread, in un socket qualsiasi. La comunicazione tra tale thread ed i worker-thread avviene tramite una lista condivisa di coppie (messaggio, file-descriptor socket).

contro: Tale strutturazione pecca in efficenza in quanto si ha un collo di bottiglia nella sottomissione dei messaggi da inviare dai worker-thread al solo thread scrittore, pertanto tale soluzione viene scartata.

3.5.2 Due worker-thread per ogni connessione

Per ogni connessione (socket) sono associati due worker-thread, un ricevente (lettore) ed uno mittente (scrittore) di messaggi. Quando un thread ricevente desidera inviare un messaggio su una connessione qualsiasi (a causa di una richiesta d'invio unicast o bradcast ricevuta dall'utente associato) tale spedizione viene sottomessa al worker-thread mittente associato alla connessione destinataria aggiungendo il messaggio alla coda condivisa dei messaggi in uscita sulla connssione.

pro: reattività del worker-thread lettore nel servire le richieste dell'utente associato. L'intervallo di tempo tra due `receiveMessage` dal socket associato dipende solo[3] dall'accesso esclusivo alla lista condivisa dei messaggi in uscita della connessione e dall'operazione di inserzione del messaggio da spedire, *non* dipende invece dal tempo d'invio del messaggio;

contro: uso di diverse risorse, in particolare il numero di worker-thread è doppio del numero degli utenti connessi.

3.5.3 Un worker-thread per ogni connessione

Ogni connessione (socket) ha associato un solo worker-thread, le scritture e le letture da un socket sono eseguite esclusivamente dal worker-thread associato. L'invio di un messaggio da un worker-thread consiste nell'aggiungere il messaggio stesso nella lista dei messaggi di uscita per la connessione dell'utente destinatario e iniviare un segnale al worker-thread associato a tale connessione, in quanto potrebbe essere bloccato in `receiveMessage`.

[3] In realtà l'intervallo di tempo tra due receive è caratterizzato anche dal tempo di creazione del messaggio da spedire all'utente e da quello per creare la stringa da sottomettere al worker-thread che scrive sul file di log. Si vuole però sottolineare il fatto che la spedizione di un messaggio da parte di un worker-thread lettore non dipende dal tempo di scrittura del messaggio stesso sulla connessione

contro: il server può essere poco reattivo rispetto alle richieste degli utenti, in quanto i worker-thread associati alle connessioni eseguono i due compiti di lettura e scrittura di messaggi, la scrittura di tutti i messaggi destinati ad una connessione puo' rallentare la ricezione dei messaggi dalla connessione stessa, soprattutto se vi sono molti utenti connessi e il tempo di scrittura è significativo.

3.5.4 Mutua esclusione nella scrittura

Esiste una corrispondenza uno-a-uno tra connessioni e worker-thread *solamente* per l'operazione di lettura. L'invio di un messaggio è effettuato da un qualsiasi worker-thread invocando `sendMessage` direttamente sul socket destinatario, dopo aver ricevuto un messagggio `TO_ONE` o `BCAST` dalla connessione associata. In tale strutturazione i worker-thread possono accedere concorrentemente in scrittura ad uno stesso socket, ciò richiede il verificare l'accesso esclusivo al soket per *tutta* la durata di spedizione del messaggio. Alune soluzioni:

Un solo mutex per tutte le connessioni tale soluzione e' implementabile usando una variabile `pthread_mutex_t` statica all'interno della funzione `sendMessage` della libreria `comsock.h` oppure globale ai worker-thread.

- **contro:** un unico punto di attesa per l'accesso esclusivo in scrittura ad una connessione qualsiasi, quindi possibili lunghe attese per la spedizione e di conseguenza bassa reattività del server nel sottomettere le richieste degli utenti.

Write atomica implementare la funzione `sendMessage` della libreria `comsock.h` im modo da effettuare una unica invocazione della funzione `write` sul socket della connessione di un oggetto di dimensioni inferiori di `{PIPE_BUF}` $\geq$ `{_POSIX_PIPE_BUF}` $= 512$

- **pro:** viene usata una proprietà dell'implementazione della system call `write` che solleva il programmatore dall'uso di strutture dati o modelli di interazione di threads per risolvere tale problema;
- **contro:** la dimensione dei messaggi e' fissata;
- **contro:** per garantire l'atomicità dell'invio di un messaggio occore effettuare una sola invocazione della system call `write` quindi si deve disporre dei campi del messaggi memorizzati in modo sequenziale in memoria, cosa che non si ha in quanto (l'eventuale) campo *buffer* è riferito indirettamente. È quindi necessario, nella funzione `sendMessage` una allocazione della memoria heap pari alla dimensione del buffer più 5 byte per gli altri due campi e l'effetture la copia dei *valori* dei tre campi.

Un mutex per ogni connessione ad ogni connessione deve essere associato una variabile `pthread_mutex_t` per garantirne l'accesso esclusivo. Tale associazione può essere implementata con una hash table, ad esempio con la stessa user-table che definisce lo stato degli utenti connessi.

- **contro:** il tempo impiegato da un worker-thread per la scrittura di un messaggio in una conessione puo' influire negativamente nella reattività del server rispetto alla sottomissione delle richieste dei clienti,

in quanto un worker-thread effettua la ricezione del messaggio successivo non prima di aver spedito il messaggio attuale nella connessione destinataria.

3.5.5 La scelta

Si è deciso di non dare un limite superiore alla dimensione di un messaggio e di impiegare il minor tempo possibile per eseguire una richiesta di un utente e servire la prossima. Il secondo aspetto è considerato importante in una chat reale "in rete" dove i ritardi di rete possono influire nella reattività del server rispetto alle richieste di un cliente. Consideriamo ad esempio l'invio di un messaggio broadcast da parte di un utente in una chat che utilizza l'infrastruttura internet e protocolli TCP/IP. Gli utenti connessi alla chat ricevono il messaggio con latenze diverse, in particolare possono esistere utenti per cui il server impiega diverso tempo per inviare un messaggio. In questo scenario è agevole strutturare il server specializzando i worker-thread nei compiti di ricezione da una connessione o spedizione su di una connessione, in modo che il thread che ha ricevuto il messaggio di broadcast non debba eseguire la spedizione a tutti gli utenti connessi e ritardare (indefinitivamente) la ricezione del prossimo messaggio.

3.6 Gestione dei segnali

Tutti i worker-thread del processo `msgserv` e il thread scrittore sul file di log filtrano tutti i segnali, l'unico thread che riceve segnali è il dispatcher, in tal modo un segnale inviato al processo è ricevuto esclusivamente da tale thread. I segnali sono usati da tale processo solo per la terminazione, avere un solo punto di ricezione dei segnali semplifica la gestione della terminazione. Il thread dispatcher,in fase di inizializzazione, ridefinisce il comportamento rispetto al segnale `SIGPIPE` in modo da ignorarlo ed ai segnali `SIGINT` e `SIGTERM` in modo da impostare al valore 1 la variabile globale `leave` dichiarata come `volatile sig_atomic_t`. Tale variabile è usata per la terminazione del processo.

3.7 Terminazione del processo e dei thread

La terminazione del processo `msgserv` avviene unicamente alla ricezione di un segnale `SIGINT` o `SIGTERM` da parte del thread dispatcher. Come già enunciato nel paragrafo **(3.2)** tutti i thread del processo sono caratterizzati dall'eseguire un'attività che per tutta la durata di vita del thread aspetta una richiesta di un "cliente" e la serve. Tale ciclo di vita ha termine quando la variabile globale `leave` assume il valore 1 (vero), infatti tale varibile è inizializzata a 0 (falso) e viene modificata solo dal gestore dei segnali di terminazione.

3.7.1 Terminazione del dispatcher-thread

Il ciclo di vita di tale thread è terminato solo dal valore vero dalla variabile globale `leave`. La fase di terminazione di tale thread prevede, oltre che alla liberazione di risorse, la richiesta di shutdown al Thread pool e la relativa attesa, la cancellazione (richiesta di terminazione) del logwriter-thread e la relativa atesa e la chiusura e unlink del socket.

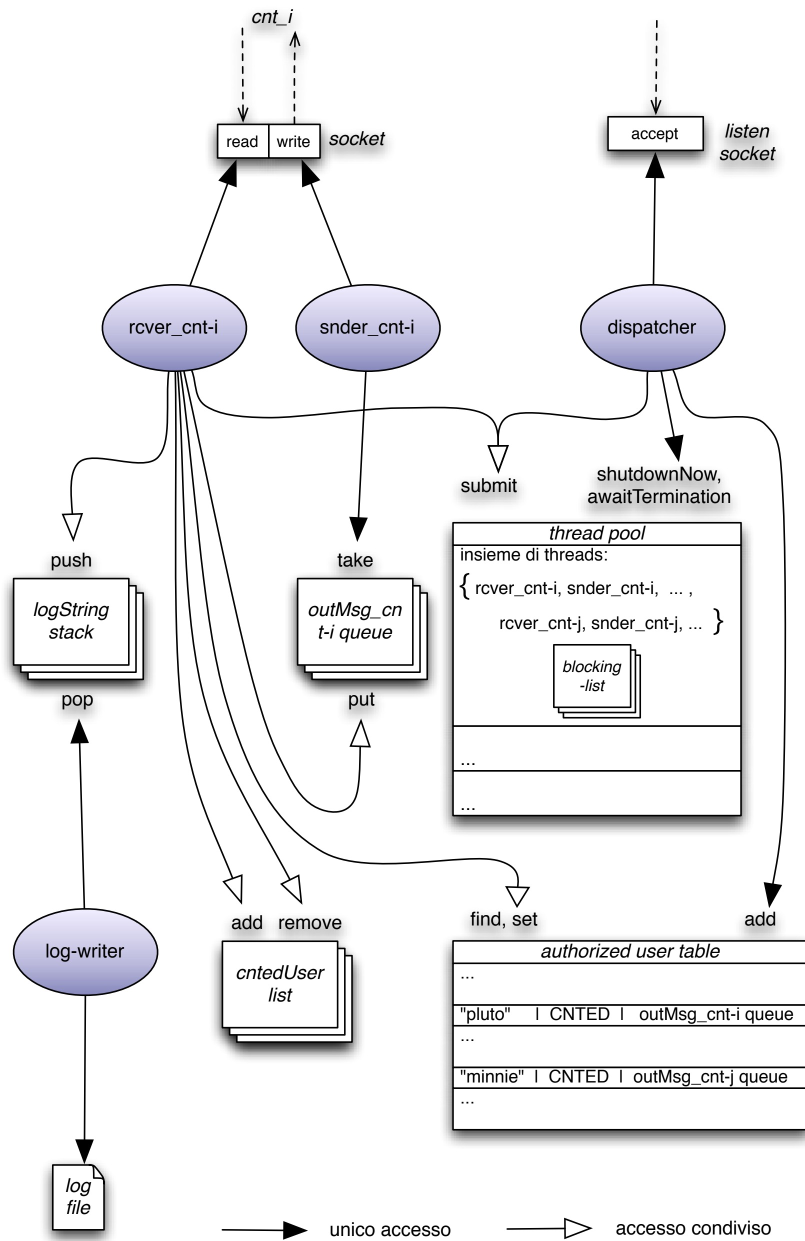

Figure 1: Strutture Dati e Threads usati nel processo `msgserv`. Nota: sono mostrati i threads e le strutture relativi solo alla connessione cnt-i.

3.7.2 Terminazione di un worker-thread

I worker-thread sia lettori che scrittori appartengono e sono gestiti dal Thread pool, quindi le modialità in cui di tali thread possono terminare sono quelle del Thread pool descritte nelle specifiche delle funzioni `threadPool_shutdown*`. In particolare si è detto che il dispatcher-thread usa lo shutdown istantaneo, caratterizzato dalla cancellazione di tutti i thread del pool. Per garantire il completamento di eventuali compiti pendenti il task dei thread riceventi abilita la cancellazione solo durante l'attesa passiva di un nuovo messaggio, mentre il task dei thread mittenti disabilitano la cancellazione all'avvio. Quando un worker-thread ricevente prende atto della richiesta di cancellazione, tra le operazioni di chiusura (cleanup) interagisce per l'ultima volta con il worker-thread mittente associato alla sua stessa connessione inviando uno speciale messaggio `message_t` di tipo `MSG_exit` il cui significato è appunto terminare l'esecuzione. Dato che tale messaggio è inserito in una coda FIFO tutti i messaggi in uscita pendenti saranno inviati prima della terminazione del thread.

3.7.3 Terminazione del logwriter-thread

Anche tale thread abilita la cancellazione solo quando esegue l'attesa passiva di un nuovo messaggio di log dalla pila bloccante. Le operazioni successive all'aver verificato la cancellazione consistono, tra le altre, di scrivere gli eventuali messaggi di log nella pila non ancora prelevati.

4 Struttura del Client

4.1 I threads del processo

Il processo `msgcli` è inizialmente composto da un unico thread (quello che esegue la funzione main) che invia la richiesta di connessione al server e aspetta il messaggio di avvenuta connessione da parte del server. Solo dopo essere connesso al server tale thread crea un altro thread, che ha il compito di ricevere e stampare i messaggi ricevuti dal server, e comincia ad eseguire la funzione che legge da standard input e invia messaggi al server.

4.2 Terminazione del processo

Esistono tre modi in cui il processo può terminare:

Terminazione richiesta dall'utente

L'utente può richiedere la terminazione ed eventuale disconnessione dal server con il comando `%exit` o inviando un segnale di terminazione. Nel primo caso il thread mittente al server legge da standard input la richiesta di uscita e esce dal ciclo di servizio. Nel secondo caso la gestione dei segnali di terminazione imposta ad 1 il valore della variabile globale `leave` di tipo `volatile sig_atomic_t` ciò fa causare nel thread mittente al server l'uscita dal ciclo di servizio. Il thread mittete al server dopo l'uscita dal ciclo di servizio invia il messaggio di uscita `MSG_EXIT` al server dopo di che attende la terminazione del thread ricevente dal server, solo dopo tale tale attesa viene chiuso il socket collegato al server. Il

thread ricevente dal server termina correttamente quando riceve il messaggio `MSG_OK` di conferma avvenuta disconnessione.

Terminazione richiesta dal server (disconnessione)

Il thread ricevente dal server legge il messaggio di disconnessione del server `MSG_EXIT`, in questo caso l'altro thread viene cancellato e si termina l'esecuzione.

Terminazione dovuta ad un errore di un thread di msgcli

In entrambi i thread si terminano i cicli di servizio. Per il thread mittente al server la gestione della terminazione è analoga al caso della terminazione richiesta dall'utente. Per il thread ricevente dal server la gestione consiste nel cancellare l'altro thread e terminare.

A Appendice: Guida per l'utente

A.1 Compliazione

Eseguire `make` in `PROJ_BASEDIR/src/` gli argomenti *msgserv* e *msgcli* per creare i file eseguibili `msgserv` e `msgcli` e compilare tutti i sorgenti e librerie necessarie.

A.2 Installazione e disistallazione

L'installazione causa la compliazione delle librerie degli eseguibili `msgserv`, `msgcli` e la copia di questi, degli headers pubblici e delle librerie nelle directory specificate. Per l'installazione eseguire `make` in `PROJ_BASEDIR/src/` con l'argomento *install*, gli eseguibili dei programmi server e client vengono posizionati nella directory `BIN_DIR` di default "`../bin`", gli headers pubblici vengono copiati in `HEADERS_DIR` di default "`../include`" e le librerie vengono copiate in `LIB_DIR` di default "`../lib`". Per installare in directory diverse modificare il file Makefile nella definizione delle vaiabili `BIN_DIR`, `HEADERS_DIR`, `LIB_DIR` oppure eseguire `make install` con parametri la definizione delle variabili, ad esempio:

```
make install BIN_DIR="/usr/local/bin"
```

La disistallazione causa la rimozione di tutti i file copiati nel sistema con l'installazione. Per la disistallazione eseguire `make uninstall` in `PROJ_BASEDIR/src/`.

Per rimuovere i file oggetto creati nella compilazione delle librerie e degli eseguibili eseguire `make clean`

A.3 Esecuzione del Server e del Client

Per eseguire il server occorre disporre di un file `userAuthList` con l'elenco dei nomi degli utenti autorizzati. Quindi eseguire:

```
BIN_DIR/msgserv userAuthList logfile
```

Al termine dell'esecuzione il file `logfile` contiene tutti i messaggi scambiati dagli utenti, può essere analizzato dallo script `logpro` per statistiche.
Per eseguire un client è sufficente eseguire con un userName autorizzato:

```
BIN_DIR/msgcli userName
```

B Appendice: Debug e Test

Per controllare il corretto funzionamento dei programmi, nei sorgenti esistono alcune stampe su standard error di valori e informazioni ritenuti interessanti in fase di sviluppo, tali stampe vengono effettuate se sono definite particolari macro. Ogni sorgente ha una specifica macro che abilita la stapa delle informazioni di debug. In Makefile esistono variabili che dichiarano macro con la relativa opzione di `gcc`, inoltre il valore della variabile `DBG` è presente in tutte le compilazioni. Ad esempio per abilitare il debug in `msgserv` eseguire i seguenti comandi:

```
make clean
make msgserv DBG=dbg_msgserv
./msgserv userlist logfile
```

Per compilare ed effettuare tutti i test degli oggetti non richiesti dal progetto eseguire `make myTest`