2018

Chat Multicanale Linux/Windows

RELAZIONE SULLA TESI DI SISTEMI OPERATIVI

EMANUELE ALFANO
FILPPO BADALAMENTI

UNIVERSITÀ DI TOR VERGATA

Sommario

Consegna della tesi:

Realizzazione di un servizio "chat" via internet offerto tramite server che gestisce un insieme di processi client (residenti, in generale, su macchine diverse). Il server deve essere in grado di acquisire in input da ogni client una linea alla volta e inviarla a tutti gli altri client attualmente presenti nella chat. Ogni client potrà decidere se creare un nuovo canale di conversazione, effettuare il join ad un canale già esistente, lasciare il canale, o chiudere il canale (solo nel caso che l'abbia istanziato).

Si precisa che la specifica richiede la realizzazione del software sia per l'applicazione client che per l'applicazione server.

Per progetti misti Unix/Windows è a scelta quale delle due applicazioni sviluppare per uno dei due sistemi.

Scelta Realizzativa

Il nostro gruppo ha deciso di portare avanti lo sviluppo del progetto su entrambi i sistemi operativi.

Per Linux è stato pensato fin da subito il server, mentre il client è stato deciso fin dall'inizio che sarebbe stato cross Platform.

In prima battuta è stato sviluppato tutto su Linux per permettere un più rapido check potendo avviare rapidamente entrambi gli eseguibili sulla stessa macchina senza passare per una macchina virtuale. Successivamente sullo scheletro del Client Linux è stato scritto il Client Windows, il quale possiede le stesse capacità ma al suo interno le sue system call sono puramente Windows.

Overview dei sistemi

Overview Server Linux

L'idea di funzionamento del server si appoggia molto alla struttura del File-System Linux, esso in input richiede il *Path-Storage*, il quale serve per spostare la directory del processo nel luogo impostato, se non esiste esso provvede alla sua creazione; successivamente valida la cartella con l'ausilio di un file di supporto (vedi anche Creazione Users & Rooms). Se il file è presente e a una versione compatibile con il software tutto procede, se non esiste e la cartella è vuota si inizializza, in caso contrario il processo termina per impossibilità di avvio.

Questo metodo permette di mantenere coerenza e persistenza al server mantenendo le strutture dati anche dopo la terminazione, e ricominciando da quel punto. E potenzialmente la creazione di più server sulla stessa macchina.

Se l'istanziazione del server ha successo si passa all'inizializzazione di 2 alberi AVL globali sui cui nodi saranno presenti le chiavi di comunicazione verso ogni entità (Thread Users e Thread Rooms).

A questo punto si passa alla fase di spawn delle Room (approfondimento su moduli del server successivi), ne viene creata una per ogni directory Room presente, ed essa si carica i propri dati dal File-System. È un assunto del sistema che **se una Room esiste allora è istanziata** e tuttalpiù in stato di Wait.

Terminata la fase di creazione delle room, comincia il setup di rete dell'applicazione, e al suo termine vengono generati N Thread Accept, il quale compito sarà di tirare su un nuovo Thread User ogni qual volta arrivasse una richiesta di connessione. Il login sta al Thread User.

A questo punto tutto è impostato correttamente, si avvia una semplice interfaccia basata su nCurse con la quale si può vedere cosa fa il server, e prelevare qualche semplice informazione tramite prompt.

Il server è ora pronto a rispondere a richieste provenienti da qualsiasi Client.

Overview Client Linux/Windows

Il client all'avvio, come per il server, sposta il proprio processo su una directory, ma non avendo necessità di prelevare dati dai file precedentemente salvati verifica solo la validità del Path.

Parte immediatamente dopo un'interfaccia di Login/Registrazione, la quale in caso di successo riceve dal server l'ID dell'utente (da utilizzare nei successivi login), e la lista delle chat alla quale ha fino a quel momento preso parte (nessuna se in fase di Registrazione).

Si arriva quindi a un'interfaccia di comando che permette di scegliere cosa fare e quali funzionalità attivare. Possiamo decidere di:

- Creare una nuova room; ***Create-Chat***
- Partecipare ad una già esistente (*senza ancora parlare*); ***Join-Chat***
- Entrare ed iniziare la conversazione in una room; ***Open-Chat***
- Eliminarne una già esistente (se precedentemente creata da noi); ***Del-Chat***
- Rimuoverci dalla lista di inoltro della room ***Exit-Chat***
- Abbandonare la room (Variando l'amministratore se ero io). ***Leave-Chat***

Se viene deciso l'ingresso in una room, è inoltre salvata in locale una copia della conversazione della chat fino a quel momento e, tenendo conto dei non trascurabili tempi di setup del sistema, verranno salvati i messaggi in arrivo a parte ed integrati sul file in persistenza non appena disponibile. A quel punto, inizia la fase di interazione con gli altri client presenti nella stessa room.

La ricezione e l'invio di messaggi è delegato a due Thread distinti. Essi lavorano in concorrenza su una struttura dati AVL che si occupa della memorizzazione degli ACK, e ricevuti come conferma di avvenuta consegna agli altri client del messaggio; se per una qualunque ragione la lista di attesa diventa troppo grande, verrà mostrato al client un avviso sulla non affidabilità momentanea del servizio offerto.

In caso si voglia uscire dalla Room basterà un CTRL+C, e un invio per tornare al menu delle opzioni.

Per il segnale CTRL+C si è inoltre preferita una mappatura più canonica nell'interfaccia dei comandi, che causa la chiusura controllata della connessione tra il client e il server.

Discussione sulle scelte realizzative, e delle tecniche usate

Realizzazione della persistenza

La persistenza nel sistema è stata ottenuta creando dei file binari nel File-System, strutturati in base alla tipologia del dato. Queste librerie sono state scritte completamente da noi e esistono anche dei demo-driver utili al loro test, e anche alla visualizzazione fuori progetto dei file prodotto, così da permettere una rapida controllata di debug durante lo sviluppo del software.

Tutte le librerie qui trattate sono state implementate in entrambi i progetti, in maniera identica a meno delle System-Call utilizzate.

La persistenza di queste strutture è correlata a una replica esatta in RAM, questo per permettere un veloce accesso da parte del programma senza però mettere a rischio i dati. Si è cercato di usare ove possibile lo standard C, per rendere più universale il codice, sono infatti presenti lievi differenze solo in fase di apertura dei file poiché era richiesta una modalità di apertura non prevista nella libreria standard C.

Tabelle utenti/chat

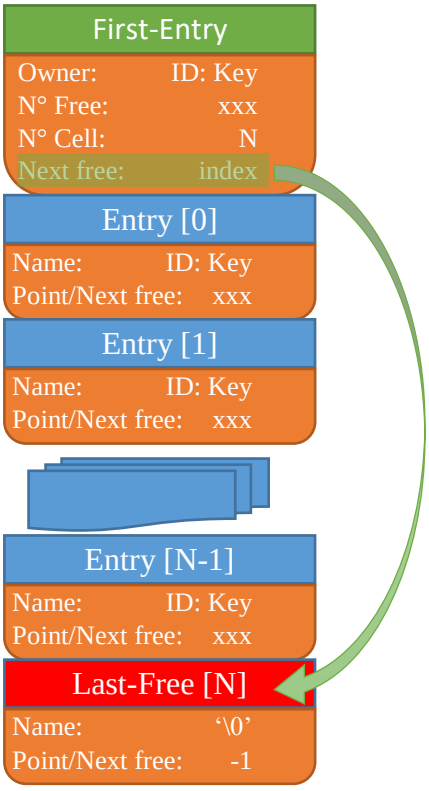
Le entità che usano le tabelle all'interno del sistema sono:

User Thread & Client Process	***Room Thread***
• Ricordare in quale room si fosse effettuato il join.	• Sapere il proprio utente Amministratore • Conoscere gli utenti collegati nella chat

Struttura del sotto sistema:

La tabella ha una struttura di tipo Testa-Data:

1. Il primo campo è un metadato della tabella. È utile in fase di lettura ad accedere correttamente alla tabella, si compone da:
   a. ***N° Free:*** contiene il numero di caselle in *stato "Delete"*
   b. ***N° Cell:*** è il numero di celle contenuto nella tabella, l'informazione è utile quando si passa alla versione in RAM, per poter tenere traccia della lunghezza della lista.
   c. ***Next Free:*** è il parametro più importante della testa, contiene l'indice della prima casella in usabile della lista, nell'esempio accanto la tabella è tutta completa e di conseguenza punta all'ultima casella, ovvero ***Last-Free***
2. Le restanti caselle hanno una struttura fissa dove:
   a. **Name** è una stringa che nel server rappresenta l'entità salvata
   b. **Point/Next free** corrisponde a un numero che varia il significato in base allo stato della cella.



Queste caselle possono essere di 3 tipo:

- **Entry:** Casella informativa valida, nel caso del server contiene un ID: Key, e la posizione del possessore della tabella nella tabella dell'entità salvata in Name. (search nella tabella in O(1))
- **Delete:** Stato in cui la casella risulta riutilizzabile, è presente per permettere a Point di puntare un valore sempre valido, poiché se si dovesse ricompattare la tabella dovrebbero ogni volta essere spostate tutte le caselle che la puntano. È caratterizzata dal NON AVER UN NAME, e avere il Point che funziona con logica Next Free, creando una specie di catena che parte da Firs-Entry e termina in Last-Free
- **Last-Free**: è una casella delete più forte, è il nodo terminale della catena ed è comoda poiché ha la stessa dimensione di ogni altra casella, quindi in caso si necessiti di aggiungere nuove caselle e ingrandire il file basta sovrascriverla e aggiungerne una nuova più avanti.

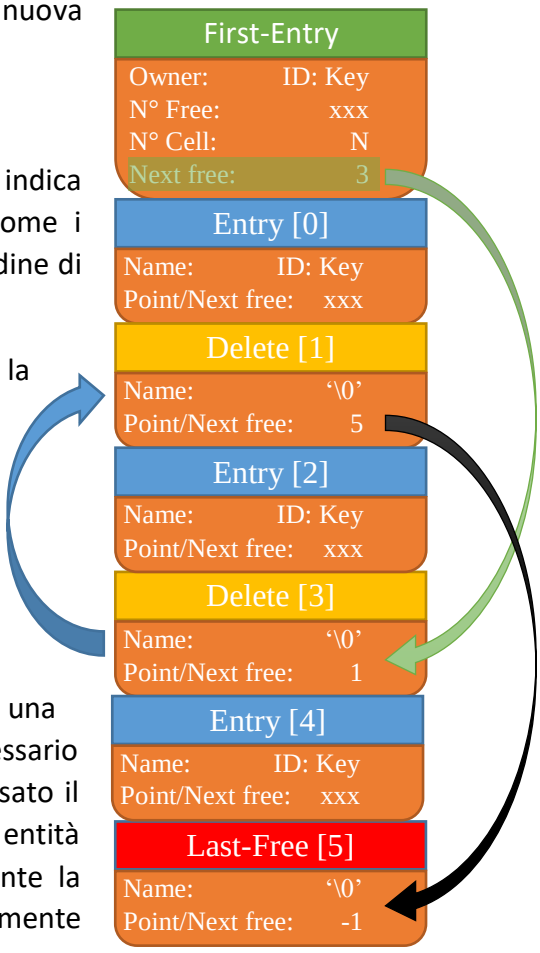
Per attuare quindi la tabella si usa un algoritmo di tipo lazy, ovvero in caso di eliminazione di una entry la tabella non viene automaticamente compattata. Si procede cambia stato in "delete", ed in questa condizione l'indice precedentemente puntato da **Next Free**, viene puntato dalla casella delete in Point, e il campo **Next Free** inizia a puntare questa nuova casella.

Nell'esempio accanto sono presenti 2 caselle eliminate.

La loro caratteristica è di avere il nome vuoto e un indice che indica dove si trova la successiva casella libera, si fa notare come i collegamenti siano a salti e dipendano strettamente dall'ordine di cancellazione e aggiunta.

In questo scenario se si volesse aggiungere qualcosa sarebbe la casella [3] a trasformarsi in una entry e il **First-Free** verrebbe modificato per puntare la [1].

Come si nota la differenza tra Delete e Last-Free sta nel fatto che il **point** risulta essere pari a -1, a indicare che la tabella è finita.



Concorrenza e consistenza

A livello implementativo ogni entità del sistema ne possiede una e solo lei ha gli estremi per accedervi, percò non è stato necessario attuare nessuna sincronia forte nel sistema. Si è tuttavia usato il costrutto flock/funlock per evitare che i 2 thread dello stessa entità potessero accedervi insieme, e si è avuta cura che durante la scrittura/lettura su file i segnali venissero temporaneamente bloccati nel thread in questione.

Operazioni in Ram

Per favorire un discorso di efficienza tutta la struttura descritta fino ad ora in persistenza è mappata 1:1 in RAM, ove l'unica grande differenza è nella lista delle entry dove è un array malloccato in Heap, e nel quale, in caso di espansione della tabella, viene chiamata la system-Call Reallocarray con un elemento in più. */*Possibile miglioramento prestazionale con un array di Puntatori (vedi Conversazione) */*

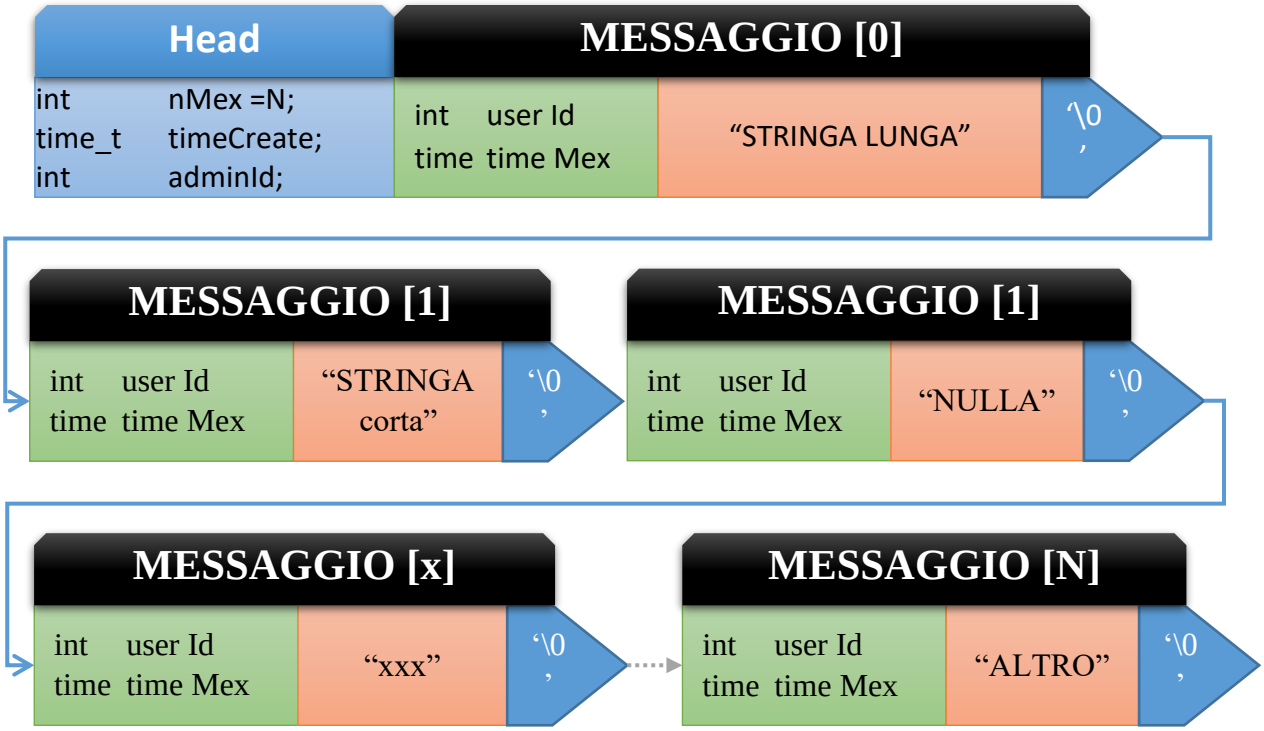
Conversazioni

Idea su persistenza

Le conversazioni sono gestite in maniera simile alle tabelle, ma con 2 importanti cambiamenti:

1. I messaggi non si possono cancellare e di conseguenza il file può solo crescere.
2. Le varie entry sono di lunghezza variabile, sono composte da una testa contenente un metadato fisso, e una seconda parte variabile, contenente la stringa da salvare.

La seconda differenza fa quindi variare in maniera considerevole il funzionamento, ora abbiamo una catena di dati intervallati tra loro con 1 metadato, e un testo variabile. Essendo il metadato fisso per leggere il file è sufficiente leggere la dimensione di ogni metadato e poi cercare il primo *'\0'*, ovvero la terminazione di stringa.



Struttura In RAM

Questa struttura però in RAM non è mantenuta alla stessa maniera essendo il messaggio variabile.

La conversazione è organizzata come un array di puntatori a Messaggi i quali referenziano ognuno un deverso messaggio, in questa maniera è semplice far crescere l'array con i valori "corti" dei puntatori mentre i messaggi (potenzialmente molto lunghi) hanno a loro disposizione tutto lo spazio di cui necessitano.

Connessione tramite Socket

La connessione Client-Server è implementata tramite protocollo TCP. La scelta è dovuta a due caratteristiche dello stesso: l'ordine nell'invio dei pacchetti e il TCP Keepalive.

Ordinamento pacchetti

Nella progettazione del pacchetto si è scelto di inserire:

- Un **metadato** di tipo struct di dimensione fissata, contenente le informazioni sull'azione da effettuare, sul tipo di dato trasportato (enum), oltre che la dimensione del corpo principale;
- Un **corpo principale** di lunghezza arbitraria (non presente se non diversamente specificato), contenente il messaggio di testo inviato, la lista delle chat o la conversazione.

La natura FIFO di TCP permette di leggere uno Stream di byte nell'ordine nel quale era stato inviato dall'altro capo della connessione. Ogni client farà una singola operazione di scrittura su socket, ma in lettura sarebbe impossibile predire la lunghezza massima di ogni pacchetto; pertanto, la funzione readPack, resa **signal-free** grazie all'utilizzo di una sigmask, leggerà prima un numero di byte pari al metadato, e consumerà gli eventuali restanti byte (tipizzati come uint in network order[1] nella struct) all'interno del canale di comunicazione.

TCP Keepalive

(fonte, http://www.tldp.org/HOWTO/TCP-Keepalive-HOWTO/)

TCP mette a disposizione tale strumento che, come si evince dal nome, controlla che la connessione tra server e client rimanga attiva anche in caso di mancanza di interazione umana sul client. Come approfondito nel link, a causa di proxy NAT e firewall, è possibile che la connessione venga interrotta per inattività delle parti dalle due figure in oggetto.

Keepalive risolve questo problema inviando una probe (sonda) allo scadere di un timeout che inizia alla fine di ogni interazione. Se tale sonda non riesce a restituire un ACK di avvenuta consegna entro un intervallo di tempo modificabile, allora verranno mandate con la stessa cadenza un numero di sonde definite dal programmatore, fino alla ricezione di almeno una di esse. In caso negativo, la connessione viene dichiarata interrotta ed inviato il segnale di broken pipe al processo che ha aperto il canale di comunicazione.

Tutti i parametri trattati sono impostabili tramite la funzione setsockopt(), e nell'implementazione del programma tutto è incluso nella funzione keepAlive(socket*).

[1] viene effettuata la conversione dell'intero da host-order a network-order in invio e il viceversa in ricezione per adeguarsi all'endianess della macchina ospitante.

Gestione della Concorrenza in RAM (Thread-Safe)

Nel progetto più di una struttura RAM ha avuto la necessità di gestire gli accessi alla struttura in concorrenza tra più entità, queste strutture avevano in comune la necessità di:

- Leggere/Cercare o in generale attuare delle operazioni che non modificano il sistema.
- Aggiungere/Togliere o in generale modificare la struttura.

Per evitare cali prestazionali, e anche per semplice buon senso, si è deciso di implementare una politica di sincronia di tipo Priority Lock. (Lock/unlock-Read & Lock/unlock-Write)

L'idea a parole

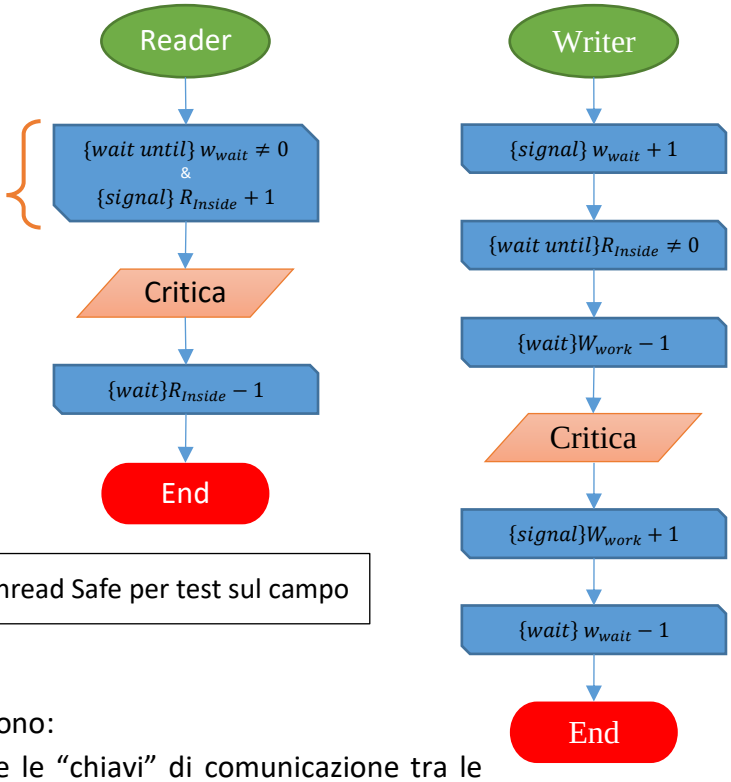
Per capire l'idea di gestione pensiamo a un *museo* dove sono presenti manutentori (Writers) e visitatori (Readers), la politica consiste di dare un "cartellini" a ogni Readers, e di farli accedere senza limiti dentro al museo. Questo finché non ci sono modifiche da fare da parte dei manutentori, ma nel momento in cui un manutentore dovesse entrare, esso impedirà di distribuire nuovi cartellini.

A questo punto ai visitatori non resta che mettersi in coda. Per quanto riguarda il/i manutentore/i in attesa, essi devono aspettare finché tutti i cartellini non rientrano, ovvero sia quando tutti i visitatori hanno finito il proprio giro. A questo punto 1 alla volta entrano nel museo, eseguono il loro lavoro e se ne vanno. Quando tutti i manutentori terminano i visitatori possono ricominciare a fluire all'interno del museo liberamente.

Leggenda:

$$\begin{cases} w_{wait} = Writer\ wait\ [Contarore] \\ R_{Inside} = Reader\ Inside\ [Contatore] \\ W_{work} = Writer\ work\ [mutex] \end{cases}$$

2 Azioni necessariamente atomiche
Per poter entrare quando non ci sono Writers al lavoro, e al contempo segnalare a un possibile Writers di mettersi in fila, senza possibilità di conflitto è Imperativo eseguire queste due operazioni in maniera Atomica, e SISTEM V permette tranquillamente l'operazione.



Vedi Demo AVL-Thread Safe per test sul campo

Dettaglio Implementativo

Strutture che necessitano di questa logica sono:

- Gli alberi AVL, usate per raccogliere le "chiavi" di comunicazione tra le varie entità presenti, il tutto mantenendo un tempo di ricerca molto rapido per le prestazioni. (vedi dopo)
- Le liste doppiamente puntate, utilizzate dentro le Room del server per gestire l'inoltro dei messaggi agli utenti collegati.

A livello implementativo per Linux si sono utilizzate le API di SYSTEM V, questo perché è per attuare una simile politica in maniera semplice e furba è necessario potersi sbloccare allo 0 di un semaforo, e attuare 2 operazioni di sincronia in parallelo.

Nei Client, essendo usati da soli 2 Thread in maniera molto blanda gli AVL, è stata applicata una semplice logica del tipo: Lock/Unlock per ogni operazione.

Da notare come in una simile politica i Writer posseggono sempre Priorità verso i Reader, i quali sono vincolati ad attendere che le manutenzioni siano finite. Ciò però offre un ottimo grado di parallelismo se le strutture non vengono modificate spesso poiché possono entrare sempre un numero imprecisato di visitatori, e *sono garantiti che* una volta entrati hanno tutto il tempo di cui necessitano per svolgere le operazioni mantenendo coerenza dei dati.

Messaggi tra Thread (Server)

Di seguito verranno spiegate come siano state realizzate al loro interno le varie entità all'interno del sistema SERVER, però per le comunicazioni tra le varie entità si è reso necessario la creazione di un metodo di comunicazione tra di loro.

Vettore delle informazioni e Responsabilità

In prima battuta si è deciso di estendere i pacchetti pensati per i Socket anche all'invio di messaggi tra le entità, unica differenza è che il pacchetto che si invia, se oltre il metadato possedesse anche un messaggio, verrebbe inviata direttamente la posizione in memoria (essendo sullo stesso Address Space) e non ogni singolo byte come invece sarebbe stato ragionevole nell'invio OutSide.

Per poter supportare la funzionalità di invio di Puntatori ad aree di memoria ed evitare al contempo dei "Segmentation Fault" è stata definita una regola di invio, o meglio di responsabilità:

- L'inviante ha il dovere di generare uno spazio in memoria **ISOLATO** nell'Address Space (malloc) e riempirlo dei dati necessari.
- Il ricevente ha quindi la garanzia di avere accesso ESCLUSIVO al pacchetto e può quindi consumare con calma i dati e successivamente, se necessario liberare memoria.

Unica eccezione a questo modo di fare è per la condivisione dei messaggi, i quali non possono essere modificati e neanche cancellati, perciò la responsabilità di aggiungere è esclusiva del gestore, e gli altri possiedono se serve il riferimento al messaggio, i quali nell'evoluzione del sistema non variano mai la loro posizione eccetto che sia cancellata la stanza.

Dettaglio implementativo

Nel sistema OGNI entità ROOM è suddivisa in 2 Thread diversi, mentre le Entità USER ne possiedono 3 di cui solo 2 sono raggiungibili (la 3° riceve dal Socket e inoltra all'userRx).

Ogni entità possiede quindi 2 possibili ricevitori "Tx & Rx" e sta a chi invia dire dove mandare per mezzo del Tag.

In Linux si è puntato sulla la struttura "***sys/msg.h - message queue structures***" la quale permette di inviare messaggi nel canale di comunicazione con allegato il "Tag". Ciò permette al ricevente di scegliere a quali pacchetti è interessato ottenendo l'effetto che i messaggi inviati vengono letti solo da chi è interessato a consumarli.

Per mezzo di questo metodo abbiamo che ogni Entità complessiva possiede 1 File-Descriptor con cui venir raggiunta e alla quale si possono inviare messaggi alle sue varie entità semplicemente scegliendo il Tag. Ne segue che il Pack creato per i Socket è "Wrappato" come messaggio della Message Queue con un certo Tag, e il ricevente selezionando il Tag in realtà ottiene il Pack.

Server Linux

Struttura

All'interno del server si possono trovare 2 tipi PRINCIPALI di entità e alcune ausiliarie:

- Principali
  - ***Thread Room***
  - ***Thread User***
- Ausiliari
  - Graphics Thread
    - ***Finto "STDOUT"*** (quello vero è usato dalla libreria nCourse)
    - ***STDERR*** re-direzionato
    - ***fdDebug***
    - ***CMD line control*** per gestire e debuggare il server in caso di bisogno
  - ***N-Thread Accept*** per gestire le connessioni al server e generare nuovi Thread User
- Temporaneo
  - ***Main o setup Thread*** è il Thread di avvio dell'applicazione, si occupa di generare tutte le Room che sono salvate nel Sile-System, avvia la connessione online e genera anche i Thread Accept. A questo punto si trasforma nel ***CMD line control*** ed entra in un ciclo perpetuo con questo scopo

Dei principali ne esistono tanti quanti ne servono per gestire in concorrenza Room e Utenti connessi:

- **Le room sono sempre presenti in stato di wait** nel sistema se hanno almeno un iscritto, non necessariamente collegato in questo momento, e di conseguenza sono sempre reperibili.
- Gli User Thread sono invece generati dal sistema dopo ogni Accept e sono il diretto interlocutore con un client residente in generale su di un'altra macchina. Se il client termina anche il Thread termina

Entità principali

Entriamo ora nel dettaglio di come sono pensati gli Utenti e le Room, la loro interconnessione la potete trovare su Schema di interconnessione.

Thread User

È composto da 3 parti:

- *User Generico*

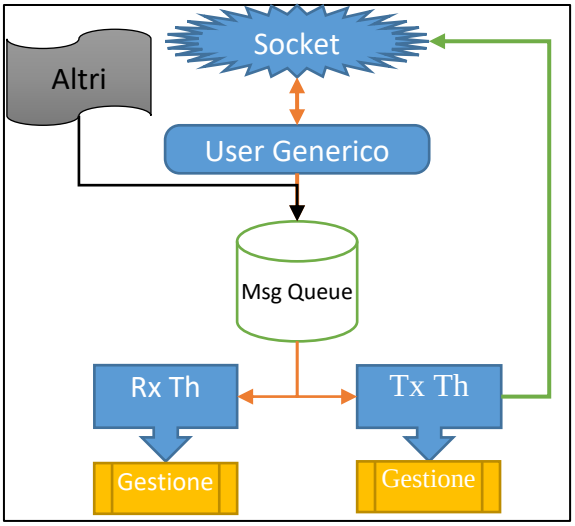
Ha la responsabilità di fare il login, mettere nell'avl l'id dell'utente appena creato e successivamente generare i Thread "Tx & Rx". Durante l'uso normale inoltra nella coda i pacchetti del Client sulla Coda. Alla chiusura della connessione ripulisce la memoria condivisa.

- *Rx Thread*

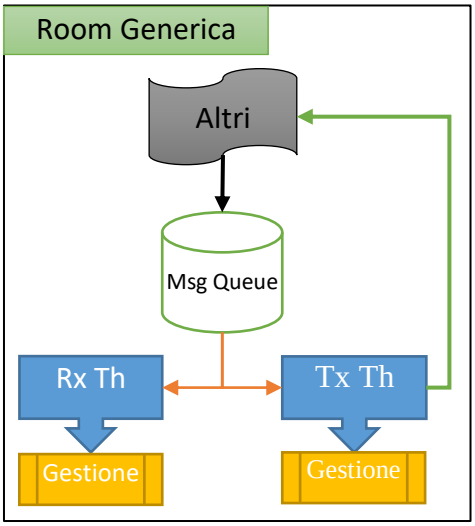
Consuma i pacchetti di "controllo" eseguendo operazioni

- *Tx Thread*

Inoltra ai Client messaggi e altre informazioni "asincrone"



Thread Room



È composto da 2 parti Attive e una di Setup:

- *Room Generico (setup)*

Ha la responsabilità di fare il setup delle infomazioni comuni appena dopo lo spawn, mettere nell'avl l'id della Room appena creata e successivamente generare i Thread "Tx & Rx". Quindi termina senza lasciare traccia.

- *Rx Thread*

Consuma i pacchetti di "controllo" eseguendo operazioni, in caso di eliminazione richiede la chiusura di Tx, attende la sua chiusura, quindi elimina dalla persistenza i file e successivamente dalla ram.

- *Tx Thread*

Inoltra a tutti gli User della room connessi messaggi e altre informazioni "asincrone".

Comunicazioni tra le Entità Principali

Le Room e gli User non sanno a priori come parlare tra di loro, a questo scopo in tutto il sistema sono presenti 2 alberi AVL globali: uno per tenere traccia degli User e l'altro delle Room.

Per le regole appena esposte una Room se **ESISTE** nel sistema allora **DEVE** essere presente nell'albero AVL, e ne consegue che se non presente nell'albero allora non esiste neanche sul File-System. Questa assunzione logica è importante poiché permette a un User di scoprire se una Room è stata eliminata o meno semplicemente cercando nell'albero (quindi anche con tempi contenuti).

Negli alberi sono salvate le coppie "***IdKey:FDEntità***", questo perché l'idKey è stato generato in maniera tale che sia *UNICO IN TUTTO IL SISTEMA* e anche in caso di eliminazione non può mai più essere riassegnato. Il fileDescriptor banalmente può variare in base al momento di creazione e alla situazione.

L'accesso avviene in maniera Sincrona come visto sopra, vedi "Gestione della Concorrenza in RAM"

Entità secondarie

Le altre entità hanno lo scopo di assistere il server e permettere a un operatore umano di vedere cosa sta succedendo a Run-Time.

Monitor Printer

Per scrivere su schermo ogni Thread parla a un File-Descriptor che in realtà è una Pipe, e i consumatori di ogni pipe sono:

- ***Finto "STDOUT"*** (quello vero è usato dalla libreria nCourse)
- ***STDERR*** re-direzionato
- ***fdDebug***

I quali accedono allo schermo anche loro in sincronia ma tramite un semplice mutex essendo nCourse non Thread-safe, in questa maniera occupano la risorsa solo il tempo necessario per trascrivere quello che hanno letto e subito rilasciano, questo unito al fatto che le pipe sincronizzano pacchetti diversi, ma non li mantengono raggruppati può causare una scrittura da parte di diversi Writer uno dietro l'altro perdendo un po' di coerenza, ma per operare dei semplici controlli si è dimostrato molto efficace.

Shell Command

Il ***CMD line control*** è l'unico Thread sul server a leggere da STDIN e per mezzo di strtok_r scompone il testo inviato in un formato uguale all'argc e *argv[], e su questa struttura esegue un piccolo script di driver che permette soprattutto di vedere informazioni riguardo a le varie entità del server, e se servisse a terminare il server, possiede anche un help per permettere di ricordare la giusta sintassi.

Principalmente permette di vedere a run-Time:

- User (quelli connessi) e Room (quelle nel File-System) attualmente presenti sul server.
- Tabelle sia per gli User che per le Room (vedi "Tabelle utenti/chat")
- Printare l'attuale stato degli alberi AVL sia per gli utenti che per le Room Per le room printare l'attuale conversazione salvata della room (vedi "Conversazioni")

Main Temporaneo

In origine è il punto di spawn di tutte le altre entità, imposta il Path di processo del server alle coordinate passate all'avvio, verifica sia una locazione valida e se si procede alla generazione di tutte le room (vedi "Struttura room").

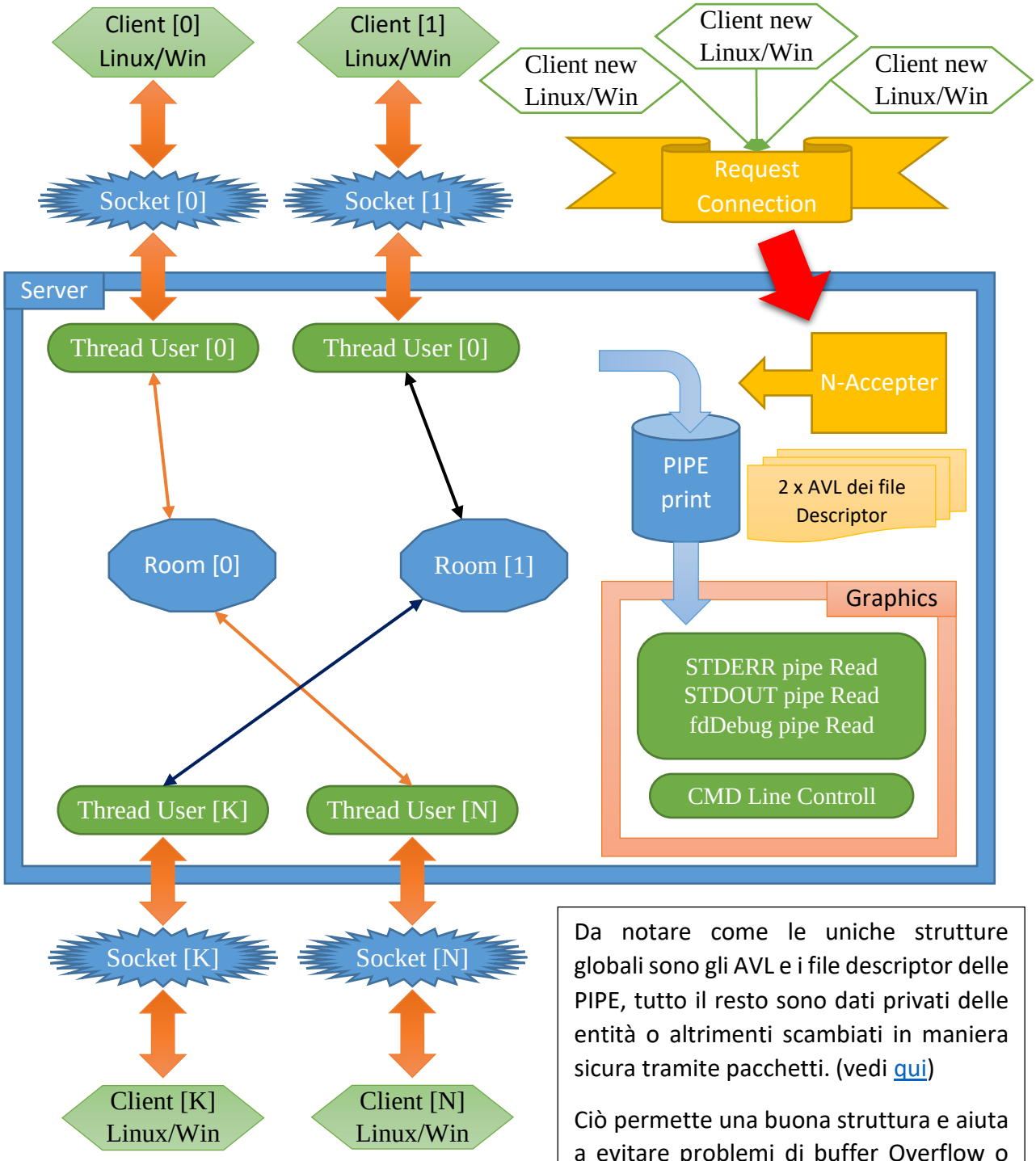
Apre la comunicazione del Socket e se ha successo genera dei Thread Accept pronti a creare Thread Client per ogni richiesta di connessione (vedi "Struttura User").

Se tutto ciò ha successo la fase di setup termina e si trasforma nel ***CMD line control*** visto sopra.

Interconnessioni tra Entità

Schema di interconnessione

Sono qui rappresentate schematicamente le entità presenti nel sistema a Setup ultimato.
Si presti attenzione al gran numero di entità gestite in totale concorrenza.
La gestione della persistenza è responsabilità di ciascuna Entità se presente e per questo non è indicata.



Da notare come le uniche strutture globali sono gli AVL e i file descriptor delle PIPE, tutto il resto sono dati privati delle entità o altrimenti scambiati in maniera sicura tramite pacchetti. (vedi qui)

Ciò permette una buona struttura e aiuta a evitare problemi di buffer Overflow o accesso a dati non consentito.

Breve manuale d'uso dei programmi (compilare & installare)

Linux Client/Server

Usare il Cmake almeno alla versione 3.9 sulla Main directory di entrambi i souce project.

Scrivere sul terminale dentro la cartella di destinazione:

cmake [Source-code-path]
<attendere>
Make

A questo punto sulla directory corrente troverete il file eseguibile compilato e pronto all'uso.
Gli eseguibili richiedono:

- Server: [Path Storage] [Coda Richieste (int)] [PORT]
- Client: [Path Storage] [IP] [PORT]

Client

Il client non potrà creare una connessione con il server se prima non sarà stato avviato, e in ogni caso il client potrebbe fallire se non si impostano correttamente IP e Port.

Se tutto va a buon fine sia arriva alla fase di login/Registrazione dove nel primo caso bisognerà mettere nome e Index dell'utente (stile password) e in caso di registrazione scegliere un Nome e il server comunicherà il proprio Id.

Server

L'IP del server è facilmente ottenibile con il comando su terminale

ifconfig

Prendere questo IP e metterlo come IP target del Client se su una macchina diversa.
Se si sta eseguendo il Client e Server sulla stessa macchina l'IP della localHost è 127.0.0.1.

Il server richiede un Path Storage VALIDO (creato in precedenza da lui), è in grado di rendersi conto se la cartella è adatta o meno, in caso non esistesse la genera lui.

Windows Client

Compilare i sorgenti dentro Windows Studio e andare ad aprire il terminale nella directory di uscita della compilazione.

Valgono tutte le accortezze dette sopra per il client.

Librerie Esterne

L'unica libreria non scritta da noi è stata quella che implementa l'AVL (singolo-Thread)

[1] *https://github.com/jarun/dslib.git*

È possibile trovare un demo dell'albero AVL Thread-Safe scritta da noi all'indirizzo:

[2] *https://github.com/Alfystar/dslib.git*