

Università degli Studi di Roma Tor Vergata



Facoltà di Ingegneria

Corso di Ingegneria del Web

**TORTELLA**

Progetto Chat room P2P

Professoressa:  
Valeria Cardellini

Studenti:  
Ibrahim Khalili  
Simone Notargiacomo  
Lorenzo Tavernese

Anno Accademico 2007-2008

# Indice

<b>Abstract</b>	<b>3</b>
<b>1 Introduzione</b>	<b>4</b>
1.1 Requisiti . . . . .	4
1.2 Sistemi Peer to Peer . . . . .	6
1.2.1 Architettura dei sistemi P2P . . . . .	7
<b>2 Gnutella vs TorTella</b>	<b>10</b>
2.1 Protocollo Gnutella . . . . .	10
2.2 Protocollo TorTella . . . . .	12
2.2.1 Descrittori TorTella . . . . .	12
2.2.2 Pacchetto TorTella . . . . .	15
<b>3 TorTella Application</b>	<b>18</b>
3.1 BootStrap . . . . .	18
3.2 Flooding . . . . .	20
3.2.1 Search e SearchHits . . . . .	20
3.2.2 Join . . . . .	23
3.2.3 Leave . . . . .	24
3.3 Failure Detection . . . . .	24
3.4 Invio e ricezione dei pacchetti . . . . .	25
3.5 Generazione ID . . . . .	26

<b>4</b>	<b>Architettura ed Implementazione</b>	<b>27</b>
4.1	Socket Manager . . . . .	29
4.2	TorTella Protocol . . . . .	31
4.3	Http Manager . . . . .	34
4.4	Packet Manager . . . . .	36
4.5	Route Manager . . . . .	38
4.6	Servent . . . . .	38
4.7	Data Manager . . . . .	45
4.8	Init . . . . .	47
4.9	Controller . . . . .	48
4.10	Utils . . . . .	51
4.11	Conf Manager . . . . .	52
4.12	Common . . . . .	52
4.13	GUI . . . . .	52
<b>5</b>	<b>Ambiente di Sviluppo</b>	<b>53</b>
5.1	Concorrenza . . . . .	53
5.2	Libreria pthread . . . . .	54
5.3	Liste, Hashtable e Code . . . . .	56
<b>6</b>	<b>Installazione ed Esecuzione</b>	<b>57</b>
6.1	Installazione . . . . .	57
6.2	Configurazione . . . . .	60
6.3	Esecuzione . . . . .	63
<b>7</b>	<b>Conclusioni e sviluppi futuri</b>	<b>71</b>

# Abstract

Il progetto scelto come argomento di esame per il corso di Ingegneria del Web prevede la realizzazione di una chat di tipo peer to peer completamente decentralizzata. Scopo della seguente relazione è di fornire una panoramica sulle reti peer to peer ed illustrare le varie scelte progettuali ed implementative esaminate durante la realizzazione dell'applicazione *TorTella* e quelle effettivamente utilizzate, nonché le problematiche incontrate dovute alla impossibilità di raggiungere una decentralizzazione completa. Inoltre, sarà presentata una guida d'installazione e verranno mostrati degli esempi di esecuzione del sistema realizzato.

# Capitolo 1

## Introduzione

### 1.1 Requisiti

Lo scopo del progetto è realizzare in linguaggio C, usando l'API dei socket di Berkeley, una chat room P2P. In un servizio di chat (o instant messaging) la comunicazione tra gli utenti avviene in tempo reale. Ogni messaggio inserito da un utente viene recapitato immediatamente ai partecipanti alla chat, che visualizzano subito il messaggio ricevuto. L'applicazione realizzata deve comprendere le seguenti funzionalità:

- La creazione di una chat room;
- L'ingresso (*join*) e l'uscita (*leave*) da una chat room esistente;
- La lista (*list*) dei partecipanti alla chat a cui si è interessati;
- L'invio dei messaggi a tutti i partecipanti attivi della chat;
- Il servizio di presence information, riguardante lo stato corrente degli altri partecipanti (online, busy, away, ...);
- Un meccanismo di *failure detection*, per decidere che un partecipante non è più attivo se, allo scadere di un periodo di *timeout*, nessun messaggio è stato ricevuto da questo partecipante.

I requisiti salienti dell'applicazione sono elencati di seguito:

- *Architettura P2P* completamente decentralizzata (l'unica forma di centralizzazione può essere eventualmente il servizio di registrazione alla chat).
- La comunicazione tra peer (basata su un protocollo da definire) deve avvenire usando come “protocollo di trasporto” il *protocollo HTTP*.
- Il protocollo di comunicazione della chat deve prevedere lo scambio delle seguenti due tipologie di messaggi:
  1. Messaggi di tipo comando: riguardano la gestione della chat room; includono i messaggi join, leave, list;
  2. Messaggi di tipo informativo: contengono il messaggio inviato a tutti i partecipanti attivi della chat
- L'applicazione deve essere eseguita nello spazio utente senza richiedere privilegi di root.
- L'applicazione deve avere un *file di configurazione*, in cui specificare i valori dei parametri di configurazione (numero di porta, ...).
- L'applicazione deve avere una semplice interfaccia grafica, comprendente una finestra principale usata per scambiare messaggi (per ogni messaggio viene visualizzato il *nickname* del mittente, l'ora in cui il messaggio è stato inviato, il testo del messaggio), una finestra secondaria per visualizzare lo stato dei partecipanti alla chat, una finestra di dialogo con l'utente locale.

La chat P2P può fornire anche le seguenti funzionalità:

- l'invio di messaggi soltanto ad un sottoinsieme di partecipanti attivi alla chat;
- l'invio di messaggi privati (soltanto ad un partecipante attivo);
- la gestione contemporanea di più room.

## 1.2 Sistemi Peer to Peer

I sistemi peer-to-peer (P2P) sono divenuti noti al grande pubblico abbastanza recentemente, intorno agli inizi del nuovo millennio. Le applicazioni basate su questo paradigma hanno consentito ai loro utenti di condividere e scambiare file di vario genere da ogni parte del mondo e di comunicare tra loro in maniera istantanea. Sebbene queste tipologie di applicazioni P2P siano indiscutibilmente le più conosciute ed utilizzate, è noto come molti utenti non siano a conoscenza di cosa esse siano e del loro reale funzionamento. Spesso viene ignorata la presenza di altre tipologie di sistemi P2P più complessi e meno pubblicizzati, ma probabilmente utilizzati almeno una volta in maniera inconsapevole. Lo scopo di questa Sezione è quello di fare chiarezza a tal proposito, offrendo una panoramica sui vari tipi di sistemi P2P esistenti al momento, e descrivendo l'utilità e le funzionalità di ciascuno di essi. In generale, il peer to peer è un modello di comunicazione nel quale ciascuna delle parti ha le stesse funzionalità e ognuna delle parti può iniziare la sessione di comunicazione, in contrasto con altri modelli come il *client/server* o il *master/slave*. In alcuni casi, la comunicazione P2P viene implementata dando ad ognuno dei nodi di comunicazione le funzionalità di server e client. Tale sistema è inoltre in grado di gestire popolazioni transienti di nodi, mantenendo tuttavia un livello accettabile di connettività e di prestazioni, e senza la necessità dell'intervento di alcun intermediario o di alcun supporto da parte di un server centrale o di una qualche autorità. In realtà, tale definizione può essere leggermente rilassata, in particolare sull'ultimo punto, come sarà possibile osservare nella seguente Sezione. I principali sistemi P2P conosciuti sono:

- sistemi per la distribuzione di contenuti (*file-sharing*): come già anticipato, sono i più noti ed utilizzati, e consentono agli utenti la condivisione e lo scambio di intere applicazioni, file audio e video, archivi, ed ogni altro tipo di contenuto trasferibile via rete. Alcuni esempi sono Napster, eMule, DC++, e KaZaA, oltre a molti altri;

- sistemi per la memorizzazione di contenuti (*file-storage*): consentono agli utenti di disporre di un file-system distribuito ed indipendente dalla locazione, efficiente e immune da tentativi di intrusione, al quale poter accedere in maniera anonima. Un esempio di questo tipo è Freenet;
- sistemi per la condivisione di risorse di calcolo: permettono agli utenti di sfruttare risorse di calcolo, come processori o memorie centrali, non locali ma distribuite anche su larga scala, il tutto in maniera trasparente. Alcuni esempi sono SETI@home e Grid;
- sistemi per la comunicazione istantanea: è l'altra *killer-application* oltre al file-sharing, e consente ai suoi utenti l'inoltro di messaggi testuali, o anche vocali, in maniera istantanea. Esempi di questa tipologia sono IRC, Microsoft Messenger e Skype, e molti altri;
- sistemi di basi di dati distribuite: offrono ai loro utenti la possibilità di memorizzare ed accedere a dati che in realtà vengono stipati in multiple locazioni remote, o anche replicati, il tutto in maniera efficiente e trasparente per gli utenti stessi.

### 1.2.1 Architettura dei sistemi P2P

Come affermato nella sezione precedente, la definizione di peer to peer in cui non si fa riferimento né ad un server centrale né ad alcun intermediario, non è propriamente corretta. Questa inesattezza è dovuta al fatto che questi sistemi hanno sperimentato un ampio fenomeno di crescita. In particolar modo, le diverse applicazioni per cui essi sono stati concepiti hanno richiesto la progettazione e la realizzazione di diverse architettura di sistema, alcune delle quali non sono propriamente conformi alla definizione sopra riportata, ma che presentano degli aspetti tali per cui ne è stata proposta ed accettata l'utilizzazione. Si riportano di seguito i vari tipi di architetture utilizzate per sistemi P2P:

- architetture decentralizzate pure: rappresentano la versione originale



dei sistemi P2P, e prevedono la sola presenza di nodi *servent* (server + client), i quali sono gli unici responsabili della ricezione, dell'elaborazione, dell'invio, o dell'inoltro dei vari messaggi scambiati con il resto dei nodi. Questa soluzione è quella che garantisce la massima scalabilità, affidabilità, e sicurezza in termini di rispetto della privacy degli utenti. L'operazione più complicata, tuttavia, è quella della scoperta dei vari *servent* che compongono il sistema, questa dev'essere realizzata grazie a complicati algoritmi distribuiti, dal momento che non esiste nessun nodo in grado di coordinare tale operazione. Un esempio di sistema P2P per file-sharing che adotta questo tipo di architettura è Gnutella 0.4.

- architetture decentralizzate ibride: questo tipo di architettura risolve le difficoltà sorte nel caso precedente introducendo un nodo di livello gerarchico più elevato. Questo nodo ha il compito di semplificare le interazioni tra i vari peer ed è implementato come server centralizzato. Esso si occupa di mantenere una directory in cui vengono memorizzate le informazioni sui peer presenti nel sistema. Questa directory viene interrogata ogni qual volta si necessita di disporre di tali informazioni. L'effetto collaterale di quest'architettura è l'introduzione di un singolo *point-of-failure* nel sistema: il suo intero funzionamento è legato all'operatività del directory server centralizzato, il quale contiene anche delle informazioni potenzialmente lesive della privacy degli utenti. Un esempio di sistema realizzato mediante questa architettura è *Napster*;
- architetture parzialmente centralizzate: il problema del singolo *point-of-failure* sollevato dalla soluzione precedente viene risolto da questa architettura introducendo una serie di *supernodi*. Essi agiscono da rappresentanti dei propri sottoposti, secondo un'organizzazione gerarchica. I supernodi vengono di volta in volta scelti in base alla loro connettività e capacità computazionale, e sono logicamente connessi tra loro. Questa soluzione è, in effetti, una mediazione tra le due precedenti, poiché

è composta da più directory server semicentralizzati, i quali sono gli unici interessati dall'operazione di discovery. In questo modo, si riduce la difficoltà d'implementazione ed il tempo d'esecuzione. Esempi di sistemi P2P per file-sharing realizzati in questo modo sono KaZaA e Gnutella 0.6.

Tutti i sistemi sopra riportati sono molto utilizzati (ad eccezione di Napster che è stato chiuso per motivi legali) e, quindi, almeno in termini di popolarità, non è dato sapere quale delle architetture sia la migliore.

## Capitolo 2

# Gnutella vs TorTella

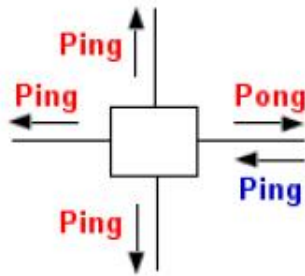
Gnutella è un protocollo per la ricerca distribuita. Sebbene tale protocollo supporti un paradigma di ricerca tradizionale del tipo client/server centralizzato, ciò che lo differenzia da altri protocolli è la sua *architettura decentralizzata*. In questo modello ogni client è un server e viceversa, proprio per questo ogni entità è chiamata *servent*. Questi forniscono un'interfaccia lato client tramite la quale gli utenti possono inoltrare *query* e vedere i risultati della ricerca e allo stesso tempo accettano anche query da altri serventi. Una rete di serventi che implementa il protocollo Gnutella è altamente tollerante ai guasti grazie alla sua natura distribuita, poiché se un sottoinsieme di serventi dovesse subire un guasto, le operazioni della rete non verrebbero comunque interrotte.

### 2.1 Protocollo Gnutella

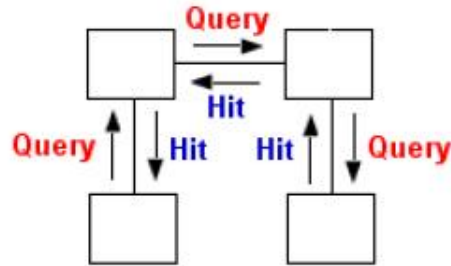
Il protocollo Gnutella<sup>1</sup> definisce il modo in cui i serventi comunicano attraverso la rete. Esso consiste in un insieme di descrittori utilizzati per la comunicazione dei dati attraverso i serventi ed un insieme di regole che governano lo scambio di questi descrittori, che sono:

---

<sup>1</sup>Protocollo versione 0.4



(a) Ping-Pong routing



(b) Query-QueryHit routing

**Ping** è utilizzato per la scoperta di host nella rete. Un peer ricevente un Ping risponderà con uno o più descrittori di tipo Pong.

**Pong** è la risposta ad un descrittore di tipo Ping. Include l'indirizzo di un servente Gnutella connesso e le informazioni riguardanti la quantità dei dati che esso rende disponibile nella rete.

**Query** è il meccanismo primario per la ricerca su reti distribuite. Un peer ricevente un descrittore di tipo Query risponderà con un QueryHit se la ricerca sul data set locale fornirà dei risultati.

**QueryHit** è la risposta ad un descrittore di tipo Query. Il descrittore fornisce al destinatario abbastanza informazioni per acquisire i dati che soddisfano la Query corrispondente.

**Push** un meccanismo che consente ad un servent sotto firewall di contribuire coi suoi dati.

I messaggi di Pong e di QueryHit vengono inviati tramite *backward routing*, ovvero i descrittori seguono lo stesso percorso dei rispettivi descrittori ping/query a ritroso. Il backward routing viene individuato tramite gli identificatori univoci dei messaggi, questo implica che se un servente dovesse ricevere un descrittore di tipo pong/queryhit con id pari a N e questi non abbia ricevuto precedentemente un descrittore di tipo ping/query con id pari a N il pacchetto verrebbe scartato.

## 2.2 Protocollo TorTella

Il protocollo *TorTella* nasce prendendo spunto da Gnutella, a cui sono state apportate delle modifiche per renderlo in linea alle necessità dell'applicativo sviluppato. L'architettura adottata da TorTella è stata inizialmente di tipo decentralizzata ibrida, ma vista la presenza di un point of failure a causa della presenza di un server centrale dedicato alle registrazioni, si è deciso di migrare verso un'architettura di tipo completamente decentralizzata. Di seguito si riportano alcune caratteristiche del protocollo TorTella, le analogie e le differenze con Gnutella:

### 2.2.1 Descrittori TorTella

Il protocollo realizzato eredita da Gnutella alcuni descrittori e ne definisce di nuovi per far fronte alle esigenze dell'applicazione:

**Ping** a differenza del descrittore originario non viene inviato in flooding, ma indirizzato verso un unico peer. Viene utilizzato per stabilire nuove connessioni, per notificare i cambiamenti di stato dell'utente e per gestire il meccanismo di failure detection. Composto da due campi, il primo rappresentante il numero di porta e il secondo rappresentante lo status dell'utente. Un pacchetto con un descrittore di tipo Ping conterrà nel campo dati il *nickname* del peer che lo invia.



Figura 2.1: Descrittore Ping

**Join** viene inviato in flooding per accedere ad una chat creata precedentemente da un generico utente. Contiene dati riguardanti l'utente che ha generato inizialmente il pacchetto, in modo da consentire a tutti i

peer riceventi di conoscere tale server. Contiene il campo status e l'id della chat a cui ci si vuole connettere, id dell'utente che vuole connettersi, porta e ip del peer mittente, ttl e hops. Il campo dati contiene il nickname.



Figura 2.2: Descrittore Join

**Leave** viene inviato in flooding per notificare l'abbandono di una chat. Contiene l'id della chat room, l'id dell'utente mittente, ttl e hops.



Figura 2.3: Descrittore Leave

**Message** utilizzato per identificare i pacchetti di tipo messaggio. Contiene l'id della chat a cui recapitare il messaggio. Nel campo dati c'è il testo del messaggio.



Figura 2.4: Descrittore Message

**Search** simile al descrittore Query di Gnutella. Contiene i campi TTL e hops. Nel campo dati è contenuta la stringa di ricerca.

**SearchHits** simile al descrittore QueryHit di Gnutella; contiene inoltre, la lista degli utenti partecipanti alle chat rispondenti alla richiesta. Contiene il numero di chat trovate sulla base della query inserita. Il campo dati contiene i risultati della ricerca.



Figura 2.5: Descrittore Search

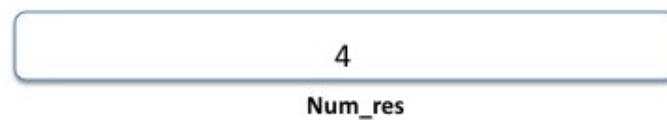


Figura 2.6: Descrittore SearchHits

**List** serve per richiedere la lista degli utenti connessi ad una specifica chat (non più utilizzato perchè gestito dalla Search). Composto dall'id della chat, ttl e hops.

**ListHits** serve per ritornare la lista degli utenti connessi ad una specifica chat (non più utilizzato perchè gestito dalla SearchHits). Composto da il numero di utenti, l'id della chat, ttl e hops. Nel campo dati c'è l'elenco degli utenti della chat.

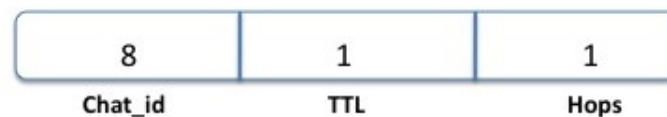


Figura 2.7: Descrittore List



Figura 2.8: Descrittore ListHits

**Bye** serve per sconnettersi da uno specifico peer.

### 2.2.2 Pacchetto TorTella

Il pacchetto TorTella è incluso nel campo dati HTTP ed è composto da 3 campi: un header di 40 byte, un campo descriptor e un campo relativo ai dati di dimensione variabile.



Figura 2.9: Pacchetto TorTella

Si presentano di seguito l'header utilizzato per la realizzazione del protocollo utilizzato e quello Gnutella:

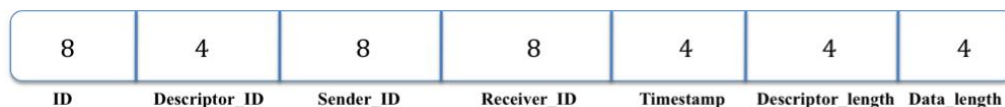


Figura 2.10: Header TorTella

**ID** un unsigned long long (8 byte) che identifica il pacchetto in modo univoco;

**Descriptor\_ID** un unsigned int (4 byte) che specifica il descrittore di pacchetto trasportato;



- 0x01 = Ping
- 0x03 = List
- 0x04 = ListHits
- 0x05 = Join
- 0x06 = Leave
- 0x07 = Message
- 0x09 = Search
- 0x10 = SearchHits
- 0x11 = Bye
- 0x12 = Close (utilizzato localmente)

**Sender\_ID** un unsigned long long (8 byte) che identifica in modo univoco il mittente del pacchetto;

**Receiver\_ID** un unsigned long long (8 byte) che identifica in modo univoco il destinatario del pacchetto;

**Timestamp** un time\_t (4 byte) che indica l'ora in cui il pacchetto è stato inviato;

**Descriptor\_length** un unsigned int (4 byte) che indica la lunghezza del descrittore che segue l'header;

**Data\_length** un unsigned int (4 byte) che indica la lunghezza del campo dati presente all'interno del pacchetto.

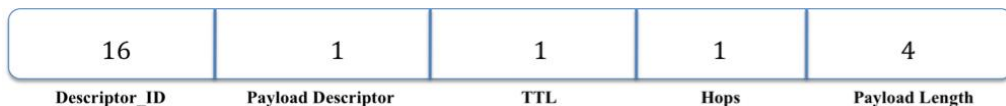


Figura 2.11: Header Gnutella

**Descriptor\_ID:** una stringa di 16 byte che identifica univocamente il descrittore sulla rete;

**Payload Descriptor:** I descrittori sono i seguenti:

- 0x00 = Ping
- 0x01 = Pong
- 0x40 = Push
- 0x80 = Query
- 0x81 = QueryHits

**TTL:** Time to live. Il numero di volte che il descrittore sarà inoltrato dai server Gnutella prima che esso sia rimosso dalla rete. Ogni server decrementerà il TTL prima di inoltrarlo ad un altro server. Quando il TTL è pari a zero, il descrittore non verrà più inoltrato;

**Hops:** Il numero di volte che il descrittore è stato inoltrato. Il TTL e il campo Hops devono soddisfare la seguente condizione:

$$\text{TTL}(0) = \text{TTL}(i) + \text{Hops}(i)$$

**Payload Length:** La lunghezza del descrittore successivo a quest'header. Il prossimo descriptor header si trova esattamente Payload Length byte dalla fine di questo header.

Come si può notare, l'header TorTella non presenta i campi TTL e Hops poiché questi servono solo ed esclusivamente nel caso di invio di pacchetti di flooding, quindi si è pensato di collocare questi due campi all'interno del campo descriptor; inoltre vengono aggiunti i campi sender\_id e receiver\_id per gestire al meglio l'invio e la ricezione dei pacchetti. Il campo descrittore presenta una lunghezza variabile a seconda del descriptor\_id presente nell'header.

## Capitolo 3

# TorTella Application

In questo capitolo si parlerà del funzionamento dell'applicazione realizzata e di tutti i suoi meccanismi. L'argomento è incentrato su: bootstrap, flooding, failure detection, invio e ricezione pacchetti.

### 3.1 BootStrap

Il primo argomento riguarda il *BootStrap* dell'applicazione, ovvero le operazioni effettuate per connettersi alla rete *TorTella*. Per effettuare quanto detto si è deciso di fornire ad ogni peer una lista di coppie *ip port* a cui connettersi inizialmente. Il metodo per reperire tale lista è al momento manuale (vedi 7). Le operazioni effettuate sono:

1. Avvio dell'applicazione;
2. L'Applicazione prende un nodo dalla lista (ip e porta);
3. Avvia un *client thread* che andrà a gestire l'invio di pacchetti verso il peer selezionato;
4. Il *client thread* appena lanciato invierà inizialmente un pacchetto Ping (con fake `recv_ID`), verso il peer a cui è associato, per stabilire la connessione;

5. Il peer ricevente risponde con un pacchetto di conferma e poi invia un Ping con il suo vero ID;
6. Il peer locale riceve il Ping e invia la conferma;
7. Connessione stabilita quindi torna al passo 2;

Da questo elenco di azioni si può notare che avviene un invio del pacchetto Ping con fake recv\_ID, questo viene fatto perchè il peer locale che vuole entrare in TorTella Network non conosce l'ID dei nodi già presenti. Questo meccanismo di connessione consente ai peer esterni alla rete di collegarsi senza problemi (vedi figura 3.1).

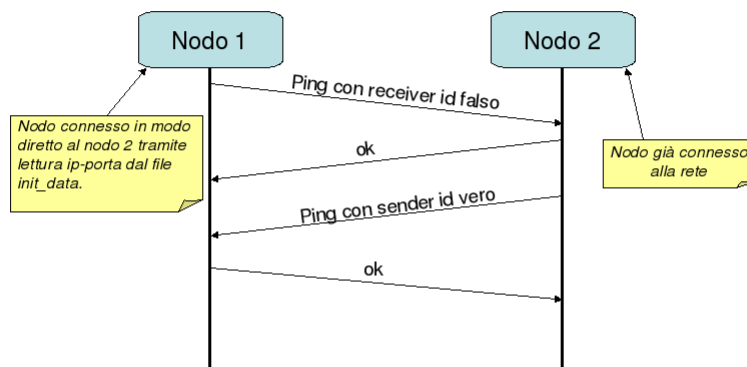


Figura 3.1: BootStrap peer

Il meccanismo di identificazione dell'ID di ogni nodo, connesso alla rete, si basa principalmente su un controllo con un parametro (`gen_start`), opportunamente inizializzato nel file di configurazione. Seguendo l'esempio

presentato in figura 3.1, il Nodo 1 dopo aver ricevuto il Ping dal Nodo 2 con il vero ID, esegue il controllo che il nuovo ID sia di valore maggiore rispetto a quello del parametro `gen_start`.

### 3.2 Flooding

Il meccanismo di *flooding* realizzato si ispira a quello del protocollo *Gnutella*, in cui quasi ogni pacchetto viene inviato in flooding. Nel caso di *TorTella* vengono inviati in flooding i seguenti pacchetti: Search, SearchHits (*backward routing*<sup>1</sup>), Join e Leave. Essendo la rete in questione completamente decentralizzata, l'implementazione del flooding per gran parte dei pacchetti è stata una scelta dettata dal fatto che il protocollo *TorTella* si basa fortemente sul *passaparola*.

#### 3.2.1 Search e SearchHits

L'intero meccanismo di searching delle chat, presenti nella rete, si basa sul principio del flooding, il quale provvede al re-direct di tutti i pacchetti di ricerca inviati da un generico peer, al fine di riuscire a recuperare una lista con tutte le chat attinenti alla query di ricerca. Una forte limitazione che si potrebbe presentare, applicando tale algoritmo, consiste nella congestione dell'intera rete nel momento in cui il numero di peer connessi raggiunga quello di un'ipotetica *web chat*. Al fine di evitare il presentarsi di tale situazione è stato impostato ad un valore finito il campo TTL (*TIME TO LIVE*) del relativo pacchetto di ricerca, il quale viene decrementato ogni volta che raggiunge un nodo, bloccando la possibilità di fare il redirect di un pacchetto nel caso questo abbia un valore pari a zero. Di seguito viene riportato un esempio di possibile ricerca con tre nodi connessi.

---

<sup>1</sup>segue a ritroso lo stesso cammino del rispettivo messaggio di Search

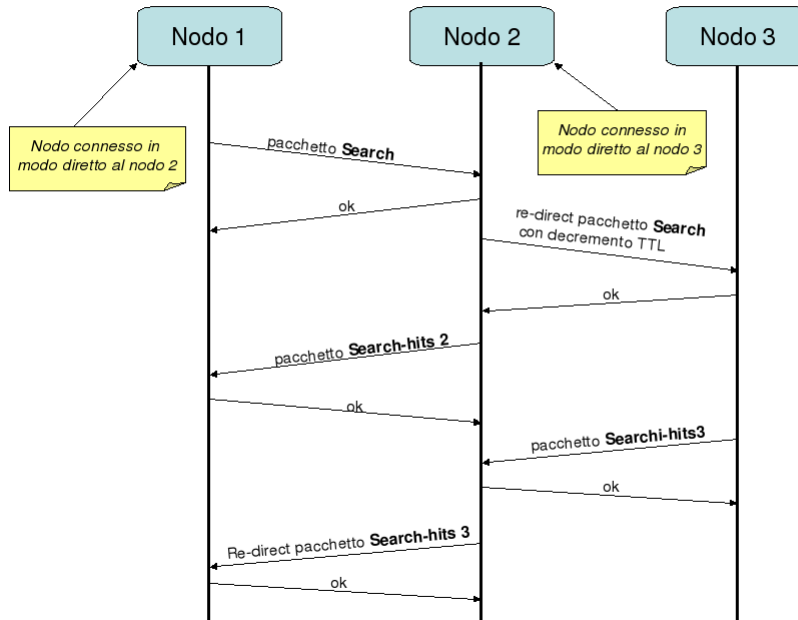


Figura 3.2: Search Flooding

Come è possibile notare osservando l'immagine 3.2, ogni nodo deve aver effettuato l'operazione di bootstrap per potersi connettere direttamente ai suoi vicini. Successivamente il NODO 1 invia un pacchetto Search, che viene inoltrato agli altri nodi diminuendo di un'unità il valore del TTL. Il NODO 2 inoltra a sua volta il pacchetto e provvede a fornire la lista di tutte le chat da lui conosciute in quell'istante (inviando un SearchHits), che rispondano ai requisiti della query effettuata; infine il NODO 3 invierà la sua SearchHits al NODO 2 e quest'ultimo rinverrà tale pacchetto al NODO 1. Il SearchHits sfrutta il *backward routing* che consente ai peer di sapere a quali nodi rinviare il pacchetto quando ne ricevono uno. A tal proposito è stata realizzata una tabella di routing che contiene tutte le regole che ogni pacchetto con uno specifico ID deve seguire. Un esempio di *routing* può essere il seguente:

1. Un peer generico riceve un pacchetto Search;

2. Rinvia il Search ai suoi vicini e aggiunge una regola di routing per ognuno di essi;
3. Ogni volta che riceverà un SearchHits prenderà la regola di routing che il pacchetto deve seguire, ovvero il peer a cui deve essere rinviato.
4. Ogni regola ha un contatore, il quale viene incrementato in base a quanti pacchetti Search con uno specifico ID sono stati rinviati.

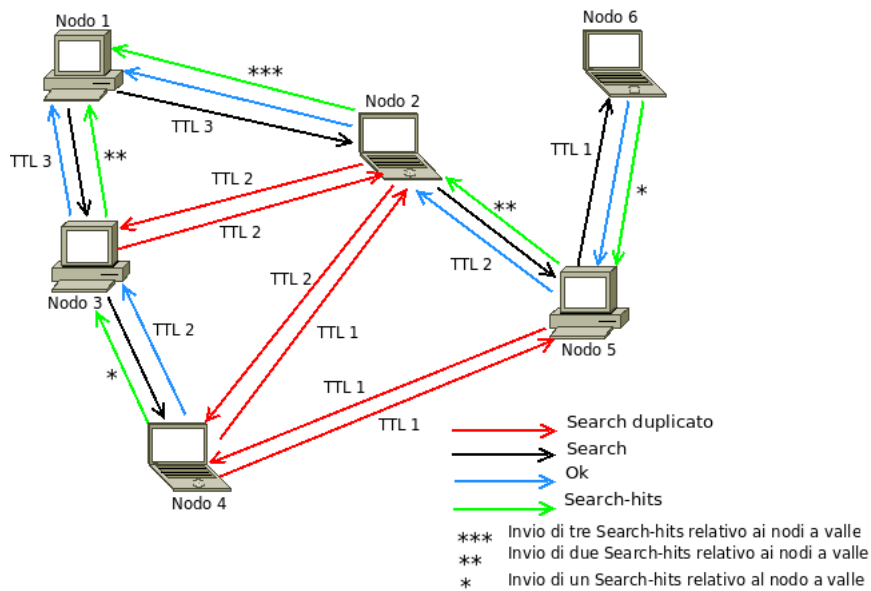


Figura 3.3: Search Flooding

L'immagine in figura 3.3 ha lo scopo di evidenziare che il flooding dei pacchetti prevede un meccanismo di gestione dei pacchetti duplicati, ovvero ogni pacchetto di tipo Search possiede un identificativo univoco per tutta la rete, in questo modo se un peer riceve nuovamente un Search con lo stesso ID provvederà a scartarlo.

### 3.2.2 Join

Inviato nel momento in cui si tenta di accedere ad una chat, il pacchetto di Join segue un percorso di flooding di sola andata, in modo che tutti i peer connessi alla chat a cui si sta accedendo aggiornino le proprie informazioni sui peer connessi. Inizialmente l'invio non doveva essere di flooding, ma in seguito a delle inconsistenze riscontrate nel caso di accessi simultanei alle chat, si è deciso di adoperare questa nuova politica. All'interno di un pacchetto di Join sono racchiuse tutte le informazioni riguardanti il peer che si sta' connettendo alla chat (che per comodità verrà chiamato peerA), poichè potrebbe accadere che alcuni peer raggiunti da tale pacchetto non conoscano il peerA. In questo modo, tutti i peer raggiunti saranno a conoscenza del fatto che peerA si sta connettendo ad una chat specifica e, nel caso uno di questi peer dovesse ricevere una Search, questi includerà nella lista degli utenti connessi (SearchHits) anche il peerA.

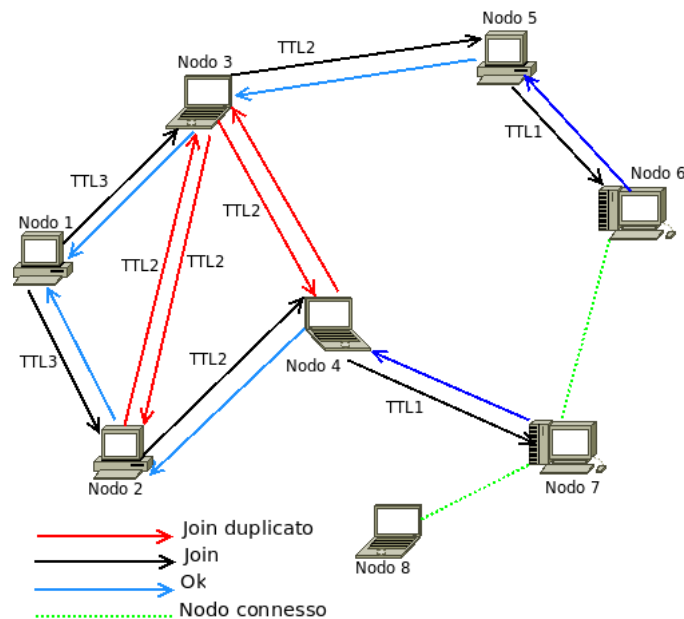


Figura 3.4: Join Flooding



### 3.2.3 Leave

Come il pacchetto Join, anche il Leave non era stato pensato come un pacchetto di flooding, ma viste le problematiche già incontrate con il Join si è adoperata la politica di flooding. Grazie al flooding, è stato possibile evitare problemi quali la presenza nella rete di differenti informazioni riguardanti le chat e i peer. Un tipico problema incontrato era il seguente: un peerA si sconnetteva dalla chat e inviava il Leave solo agli utenti connessi alla chat; questo comportava che qualora un qualsiasi peer effettuasse la ricerca di quella chat nell'istante in cui il peerA si stava sconnettendo, avrebbe ricevuto informazioni errate sulla chat, poichè il pacchetto SearchHits avrebbe contenuto tra gli utenti connessi anche il peerA.

## 3.3 Failure Detection

Tale meccanismo serve a capire quando un peer non è più attivo, a causa di crash o problemi di rete. Per la gestione di questi inconvenienti, si è deciso di inviare ad intervalli di tempo regolari un pacchetto di Ping per rilevare lo stato del peer remoto; in caso di mancato invio o errore nella ricezione del pacchetto, il sistema capisce che il peer remoto non è più attivo e provvederà alla rimozione dei dati che lo riguardano.

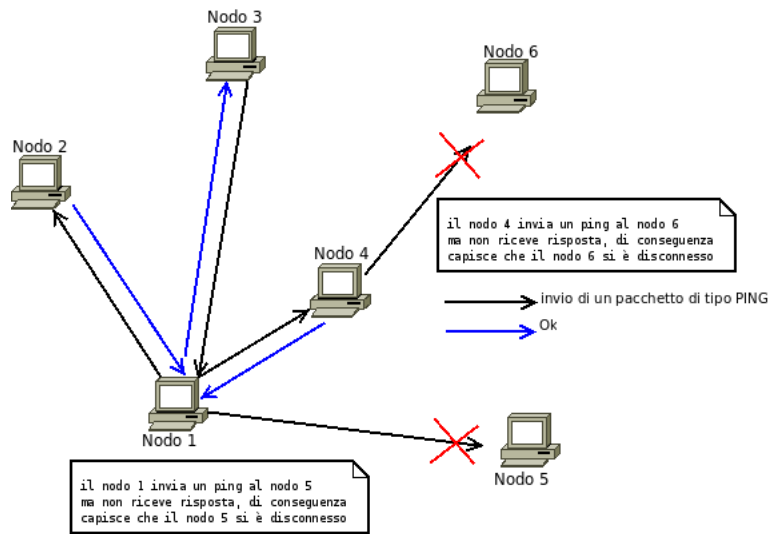


Figura 3.5: Failure Detection

### 3.4 Invio e ricezione dei pacchetti

Per ogni peer con cui comunicare ci sono due thread: un thread per la gestione del client (invio) e un thread per la gestione del server (ricezione). In questo modo se si vuole inviare un pacchetto ad uno specifico peer basta richiedere al *client thread* di farlo, il che viene fatto sfruttando una semplice coda che contiene tutte le richieste di invio pacchetti. Il *client thread* preleva la richiesta dalla coda ed effettua l'operazione desiderata. Il *server thread* invece riceve tutti pacchetti e li elabora in base al tipo (vedere capitolo 4). La comunicazione tra i thread ed il resto dell'applicazione è resa possibile da una struttura dati (una per peer) che contiene tutte le informazioni relative al peer. Tramite questa divisione dei compiti si è raggiunto un livello di efficienza tale da permettere la comunicazione con decine di peer contemporaneamente.

## 3.5 Generazione ID

Si sono scelti degli ID da 8 byte, in modo da rendere il più raro possibile la presenza di ID duplicati. Per la generazione dell'ID è stata usato un algoritmo che sfrutta il *MAC* della macchina e il *timestamp*. Per dettagli vedere il capitolo 4.

## Capitolo 4

# Architettura ed Implementazione

L'applicazione è stata progettata seguendo un approccio modulare, per consentire una migliore leggibilità del codice e una più facile manutenibilità dello stesso. L'applicazione è composta da vari moduli:

**Socket Manager:** gestione delle connessioni TCP.

**TorTella Protocol:** definizione del protocollo utilizzato.

**Http Manager:** creazione dei pacchetti http e relativi parser.

**Packet Manager:** intermediario tra il livello http e quello dei socket.

**Route Manager:** gestione delle regole di routing per i pacchetti di flooding.

**Servent:** strutture e funzionalità relative ai peer.

**Data Manager:** gestione dei dati della chat e degli utenti.

**Init:** modulo che consente il boot iniziale dell'applicazione.

**Controller:** gestione della comunicazione tra la GUI e gli strati sottostanti dell'applicazione.

**GUI:** generazione e gestione dell'interfaccia grafica.

Moduli di supporto:

**Utils:** generazione degli id, dump dei dati.

**Common:** definizione dei tipi di dati utilizzati.

**Logger:** gestione del log dell'applicativo.

**Conf Manager:** gestione del file di configurazione.

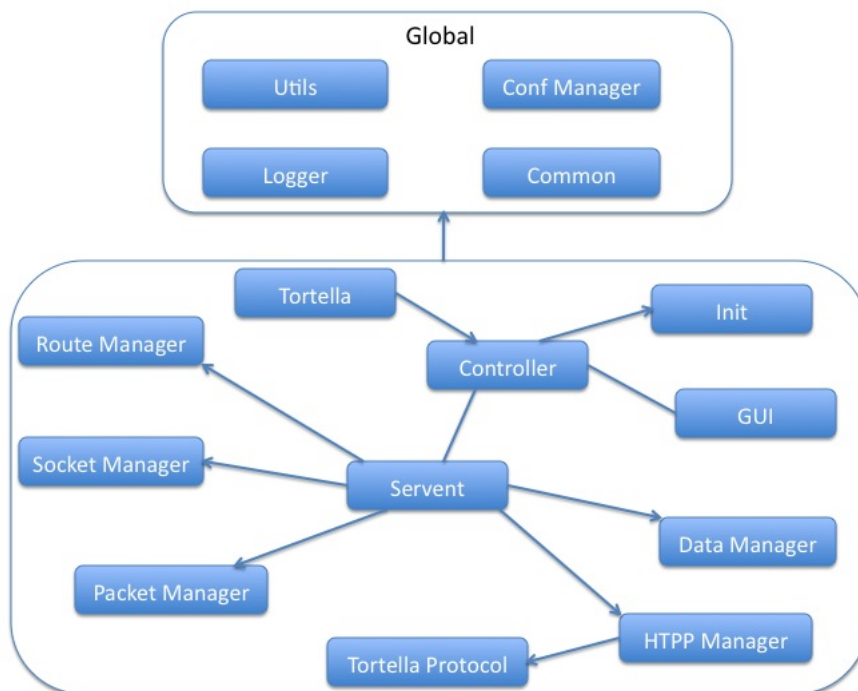


Figura 4.1: Schema Architetture

Nella figura 4.1 si possono notare le interdipendenze tra moduli, in particolare l'importanza del modulo servent, che rappresenta il cuore dell'applicazione. Di fondamentale importanza è anche il controller, che gestisce la comunicazione tra la GUI e il resto dell'applicazione, nonché il bootstrap con le relative connessioni iniziali ai peer.

## 4.1 Socket Manager

Lo strato software che gestisce l'instaurazione di una connessione di tipo persistente, e di tutto il relativo processo di comunicazione tra due o più nodi viene gestito dal file `socketmanager.c`. Poichè un server deve comportarsi sia da client che da server. A tal proposito verranno analizzate i due processi di creazione dei socket.

**create\_tcp\_socket()** La funzione ha il compito di creare un socket di connessione remota, ovvero di instaurare una connessione remota ad un server TCP. Come è possibile capire, il tipo di comunicazione è orientata alla connessione, permettendo il trasferimento di un flusso continuo di dati. Dopo la fase di inizializzazione del socket e l'apertura della connessione verso il server, viene eseguita la chiamata di sistema `setsockopt()` con il flag `SO_KEEPALIVE`, al fine di caratterizzare la connessione precedentemente instaurata, ovvero creando un tipo di connessione persistente<sup>1</sup>.

```
int create_tcp_socket(const char* dst_ip, int dst_port)
```

**create\_listen\_tcp\_socket** Il compito di inizializzare il lato server del peer, creando un socket d'ascolto, è affidato a questa funzione. Dopo la creazione del socket, vengono effettuate le chiamate `bind()` e `listen()`, le quali hanno il compito rispettivamente di inizializzare l'indirizzo IP locale e il relativo processo e di porre il socket dallo stato di `CLOSED` a quello di `LISTEN`. Come nel caso precedente viene utilizzata la syscall `setsockopt()`, la quale non si occupa solo di mantenere la connessione in uno stato permanente, ma anche di poter riutilizzare l'indirizzo locale. Usando il flag `SO_REUSEADDR` fra la chiamata a `socket()` e quella a `bind()` si consente a quest'ultima di avere comunque successo anche se la connessione è attiva o si trova nello stato di

---

<sup>1</sup>Per connessione persistente si intende la presenza di un timer che viene reinizializzato ogni volta che avviene uno scambio di informazioni tra peer. Allo scadere del timer, il socket viene chiuso

TIME\_WAIT. Le operazioni riguardanti la ricezione del flusso di pacchetti e relativo invio sono gestite rispettivamente dalle funzioni: `recv_packet()`, `send_packet()`, le quali attraverso le chiamate di sistema standard della programmazione di rete si occupano di svolgere le funzioni di scrittura e lettura sui sockets d'interesse. Particolare attenzione deve essere rivolta alla funzione di lettura dei dati, in quanto è stato previsto un meccanismo di frammentazione dei pacchetti in blocchi, di dimensione massima (4096 byte) pari a quella impostata nel parametro `buffer_len`, presente nel file di configurazione iniziale (`tortella.conf`).

```
int create_listen_tcp_socket(const char* src_ip, int src_port)
```

Il modulo, oltre alle funzioni di creazione del socket, fornisce le funzioni relative all'invio e alla ricezione dei pacchetti.

**send\_packet()** Tale funzione consente la comunicazione tra i peer, in particolare consente di inviare dati tramite un socket. Per effettuare questa operazione viene utilizzata la chiamata di sistema `write()`.

```
int send_packet(int sock_descriptor, char* buffer, int len)
```

**recv\_packet()** Quest'ultima funzione consente la ricezione dei dati da un peer. E' stata progettata rendendo possibile la ricezione di dati di dimensione illimitata (teoricamente), questo per rendere possibile il trasferimento di grossi file<sup>2</sup>. Il buffer passato come argomento deve essere non allocato per consentire alla funzione di allocare memoria dinamicamente.

```
int recv_sized_packet(int sock_descriptor, char** buf, int max_len)
```

---

<sup>2</sup>funzionalità non implementata

## 4.2 TorTella Protocol

Questo modulo contiene tutte le definizioni, le funzioni e le macro che implementano il protocollo TorTella. Ci sono tutte le definizioni dei pacchetti e dei descrittori trasportati, che sono i seguenti:

```
#define PING_ID          0x01
#define LIST_ID          0x03
#define LISTHITS_ID      0x04
#define JOIN_ID          0x05
#define LEAVE_ID         0x06
#define MESSAGE_ID       0x07
#define SEARCH_ID        0x09
#define SEARCHHITS_ID    0x10
#define BYE_ID           0x11
#define CLOSE_ID         0x12

//Status ID
#define ONLINE_ID        0x80
#define BUSY_ID          0x81
#define AWAY_ID          0x82

struct tortella_header {
    u_int8 id;
    u_int4 desc_id;
    u_int8 sender_id;
    u_int8 rcv_id;
    time_t timestamp;
    u_int4 desc_len;
    u_int4 data_len;
};
typedef struct tortella_header tortella_header;

struct ping_desc {
    u_int4 port;
    u_int1 status;
    //Campo dati: nickname
};
typedef struct ping_desc ping_desc;

struct list_desc {
    u_int8 chat_id;
    u_int1 ttl;
    u_int1 hops;
};
typedef struct list_desc list_desc;
```



```
struct listhits_desc {
    u_int4 user_num;
    u_int1 ttl;
    u_int1 hops;
    u_int8 chat_id;
    //Campo dati: elenco utenti della chat
};
typedef struct listhits_desc listhits_desc;

struct join_desc {
    u_int1 status;
    u_int8 user_id;
    u_int8 chat_id;
    u_int4 port;
    char ip[16];
    u_int1 ttl;
    u_int1 hops;
    //Campo dati: nickname
};
typedef struct join_desc join_desc;

struct leave_desc {
    u_int8 user_id;
    u_int8 chat_id;
    u_int1 ttl;
    u_int1 hops;
};
typedef struct leave_desc leave_desc;

struct message_desc {
    u_int8 chat_id;
    //Campo dati: il msg
};
typedef struct message_desc message_desc;

struct search_desc {
    u_int1 ttl;
    u_int1 hops;
    //Campo dati: stringa ricerca
};
typedef struct search_desc search_desc;

struct searchhits_desc {
    u_int4 num_res;
    //Campo dati: risultati ricerca
};
```

```
typedef struct searchhits_desc searchhits_desc;

struct bye_desc {
};
typedef struct bye_desc bye_desc;

struct tortella_packet {
    tortella_header *header;
    char *desc;      //desc_len nell'header del pacchetto
    char *data;      //data_len nell'header del pacchetto
};
typedef struct tortella_packet tortella_packet;
```

Le varie funzioni sono descritte di seguito:

**tortella\_bin\_to\_char()** Questa funzione ha il compito di eseguire il parser del pacchetto tortella. In particolare prende in input da parametro il pacchetto, memorizzato nella sua struttura dati, e restituisce tutto il suo contenuto in un buffer di caratteri. Inoltre ritorna la lunghezza del buffer generato. Si è cercato di rendere tale operazione di conversione il più efficiente possibile, essendo una funzionalità utilizzata ogni volta che si invia un pacchetto.

```
char *tortella_bin_to_char(tortella_packet *packet, u_int4 *len)
```

**tortella\_char\_to\_bin()** Tale funzione svolge le funzionalità di parser inverso rispetto alla funzione **tortella\_bin\_to\_char()**. La procedura riceve come parametro il buffer, contenente i dati, i quali vengono memorizzati nella struttura dati **tortella\_packet**. Questa funzione viene chiamata ogni volta che si riceve un pacchetto, per estrarre il suo contenuto e renderlo maggiormente accessibile dalle altre funzioni del programma.

```
tortella_packet *tortella_char_to_bin(char *packet)
```

## 4.3 Http Manager

La gestione dei pacchetti HTTP, il cui protocollo è stato definito nel precedente capitolo, viene affrontato all'interno di questo modulo. Di seguito vengono riportate le strutture dati rappresentanti i pacchetti:

```
struct http_header_request {
    char *request;
    char *user_agent;
    u_int4 range_start;
    u_int4 range_end;
    u_int4 content_len;
    char *connection;
};
typedef struct http_header_request http_header_request;

struct http_header_response {
    char *response;
    char *server;
    char *content_type;
    u_int4 content_len;
};
typedef struct http_header_response http_header_response;

struct http_packet {
    u_int4 type;
    http_header_request *header_request;
    http_header_response *header_response;
    tortella_packet *data;
    char *data_string;
    u_int4 data_len;
};
typedef struct http_packet http_packet;
```

Come è possibile notare sono state definite 3 tipi di strutture dati, dove le prime due sono utilizzate per la creazione dell'header a seconda se il messaggio da inviare sia di richiesta o di risposta, invece la terza struttura è utilizzata per definire l'intero pacchetto http. Sia nell'header di request che nell'header di response viene memorizzata, la lunghezza dell'intero pacchetto, la tipologia di dei dati contenuti(html,image..), che nel nostro caso saranno solamente di tipo binario e la versione del tipo di protocollo utilizzato (TorTella 0.1). Le uniche differenze presenti sono dovute al tipo di informazione (richiesta o

risposta) contenuta negli header e all'identificazione nell'header request del tipo di connessione instaurata (di tipo persistente) tra due peer. Al fine di rendere più agevole la creazione di un generico pacchetto sono state definite delle macro all'interno dell' header file:

```
#define HTTP_REQ_GET          0x40
#define HTTP_RES_GET          0x41
#define HTTP_REQ_POST         0x42
#define HTTP_RES_POST         0x43
#define HTTP_STATUS_OK        200
#define HTTP_STATUS_CERROR     400
#define HTTP_STATUS_SERROR     500

#define HTTP_REQ_POST_LINE     "POST * HTTP/1.1"
#define HTTP_AGENT              "User-Agent: "
#define HTTP_REQ_RANGE         "Range: bytes="
#define HTTP_CONNECTION        "Connection: "
#define HTTP_CONTENT_TYPE      "Content-Type: "
#define HTTP_CONTENT_LEN       "Content-Length: "
#define HTTP_SERVER             "Server: "

#define HTTP_OK                 "HTTP/1.1 200 OK"
#define HTTP_CERROR             "HTTP/1.1 400 Bad Request"
#define HTTP_SERROR             "HTTP/1.1 500 Internal Server Error"
```

Tutte le funzionalità relative al parsing dei dati contenuti nel pacchetto http sono svolte dalle seguenti funzioni:

**http\_bin\_to\_char()** La funzione si occupa di leggere i dati memorizzati in una struttura dati per poi salvare gli stessi in un buffer di caratteri. Nel momento in cui vi è la chiamata a tale funzione vengono passati come parametro sia l'intero pacchetto http che la sua lunghezza. L'algoritmo di parsing dopo aver controllato che il pacchetto ricevuto non sia vuoto provvede alla creazione di un buffer, il quale avrà dimensioni differenti in base al tipo di messaggio contenente il pacchetto (REQ\_POST-REQ\_GET-RES\_POST-RES\_GET). Ogni campo dell'header, all'interno del buffer, viene delimitato con due caratteri separatori `\r\n`, in modo da permettere un'identificazione migliore delle varie informazioni, nel caso in cui si abbia la necessità di riconvertire tutti i dati nelle loro strutture d'appartenenza (`http_char_to_bin`).

Dopo di ciò viene fatto uso di un puntatore temporaneo, il quale dopo essere stato inizializzato con l'indirizzo del primo byte del buffer, viene spostato della sua lunghezza, in modo da poter appendere in seguito tutte le informazioni relative al campo dati del pacchetto http (vedi appendice).

```
char *http_bin_to_char(http_packet *packet, int *len)
```

**http\_char\_to\_bin()** La funzione esegue il procedimento inverso della **http\_bin\_to\_char()**, in quanto prendendo da parametro un puntatore a caratteri, che contiene i campi dati del pacchetto http, effettua la conversione di questa nella relativa struttura d'appartenenza. Come nel caso precedente l'algoritmo suddivide il processo di parsing a seconda del tipo di pacchetto da ricostruire. Durante la conversione la funzione si occupa anche di preparare la struttura dati **tortella\_packet**, fornita dallo strato inferiore come contenitore del messaggio del pacchetto.

```
http_packet *http_char_to_bin(const char *buffer)
```

## 4.4 Packet Manager

Il modulo in questione consente di creare ed inviare pacchetti TorTella e HTTP. Si può considerare come un insieme di API, volte a migliorare la leggibilità del codice, altrimenti si sarebbero dovute effettuare tutte le operazioni di invio manualmente. In particolare, le operazioni che ognuna di queste API svolge sono:

1. Preparazione dell'header del pacchetto TorTella;
2. Preparazione del descrittore del pacchetto in base a ciò che si vuole inviare;
3. Preparazione dei dati (qualora presenti) del pacchetto TorTella;

4. Preparazione del pacchetto HTTP;
5. Conversione del pacchetto in un buffer di dati;
6. Invio del pacchetto;

Di seguito si riportano le definizioni delle funzioni:

```
int send_search_packet(u_int4 fd, u_int8 packet_id, u_int8 sender_id, u_int8
    recv_id, u_int1 ttl, u_int1 hops, u_int4 string_len, char *string)

int send_searchhits_packet(u_int4 fd, u_int8 packet_id, u_int8 sender_id,
    u_int8 recv_id, u_int4 num_res, u_int4 res_len, char *res)

int send_join_packet(u_int4 fd, u_int8 packet_id, u_int8 sender_id, u_int8
    recv_id, u_int1 status, u_int8 user_id, u_int8 chat_id, char *nick,
    u_int4 port, char *ip, u_int1 ttl, u_int1 hops)

int send_leave_packet(u_int4 fd, u_int8 packet_id, u_int8 sender_id, u_int8
    recv_id, u_int8 user_id, u_int8 chat_id, u_int1 ttl, u_int1 hops)

int send_ping_packet(u_int4 fd, u_int8 sender_id, u_int8 recv_id, char *nick
    , u_int4 port, u_int1 status)

int send_list_packet(u_int4 fd, u_int8 sender_id, u_int8 recv_id, u_int1 ttl
    , u_int1 hops, u_int8 chat_id)

int send_listhits_packet(u_int4 fd, u_int8 sender_id, u_int8 recv_id, u_int4
    user_num, u_int4 res_len, char *res, u_int8 chat_id)

int send_bye_packet(u_int4 fd, u_int8 sender_id, u_int8 recv_id)

int send_message_packet(u_int4 fd, u_int8 sender_id, u_int8 recv_id, u_int8
    chat_id, u_int4 msg_len, char *msg)

int send_post_response_packet(u_int4 fd, u_int4 status, u_int4 data_len,
    char *data)

int send_get_request_packet(u_int4 fd, char *filename, u_int4 range_start,
    u_int4 range_end)

int send_get_response_packet(u_int4 fd, u_int4 status, u_int4 data_len, char
    *data)
```

## 4.5 Route Manager

Il modulo è stato implementato per la gestione delle regole di backward-routing dei pacchetti di tipo Search-SearchHits. Il peer da cui parte la richiesta deve ricevere un pacchetto SearchHits da ogni peer che ha ricevuto la richiesta; per realizzare tale meccanismo si utilizzano le regole di routing. In particolare, ogni peer che riceve un pacchetto Search deve memorizzare nella tabella di routing un record relativo al percorso che dovrà effettuare il SearchHits di risposta. Per il funzionamento di questo meccanismo è stata utilizzata una hashtable contenente la struttura dati di seguito elencata:

```
struct route_entry {
    u_int8 sender_id;
    u_int8 rcv_id;
    u_int4 counter;
};
typedef struct route_entry route_entry;
```

Le regole vengono gestite attraverso le seguenti funzioni:

```
int add_route_entry(u_int8 packet_id, u_int8 sender_id, u_int8 rcv_id)

int del_route_entry(u_int8 id)

route_entry *get_route_entry(u_int8 packet_id)

u_int8 get_iddest_route_entry(u_int8 id)
```

## 4.6 Servent

L'architettura di un sistema *peer to peer* prevede che ogni entità sia in grado di funzionare sia da client che da server. Ogni peer deve quindi essere realizzato come un servent. La parte server dovrà gestire i messaggi in arrivo dalla rete e quella client dovrà invece provvedere a inviare dati agli altri peer. Questo porta alla necessità di avere un sistema di gestione di tipo concorrente. L'utilizzo di thread, per la gestione concorrente delle richieste, ha permesso

una maggiore agevolazione nella implementazione della comunicazione *client server*.

**servent\_data** Per la comunicazione tra i thread relativi ad un peer è stata predisposta una struttura dati per la condivisione delle informazioni, ovvero la **servent\_data**:

```
struct servent_data {
    u_int8 id;
    GQueue *queue;
    GQueue *res_queue;
    char *ip;
    u_int4 port;
    u_int1 status;
    char *nick;
    time_t timestamp;

    GList *chat_list;

    pthread_rwlock_t rwlock_data;

    char *msg;
    u_int4 msg_len;

    GList *chat_res;
    u_int1 ttl;
    u_int1 hops;
    u_int8 packet_id;

    u_int1 is_online;
    u_int4 post_type;

    //FLOODING
    u_int8 user_id_req;
    u_int8 chat_id_req;
    u_int4 port_req;
    char *nick_req;
    char *ip_req;
    u_int1 status_req;
    char *title;
    u_int4 title_len;
};

typedef struct servent_data servent_data;
```



I campi della struttura presentata sono divisi in gruppi, in base al loro tipo di utilizzo:

- i campi che rappresentano le informazioni del peer, come: ip, porta, nickname, id;
- i campi utilizzati per la comunicazione con il peer, come il tipo di pacchetto da inviare al peer associato alla struttura dati. Ad esempio, nel caso in cui il server thread dovesse ricevere un pacchetto Search da uno specifico peer, tale campo dovrà assumere il valore `SEARCHHITS_ID` per segnalare al client thread di inviare il pacchetto SearchHits di risposta;
- i campi utilizzati per il flooding, ovvero servono alla ritrasmissione dei pacchetti ricevuti (Search, SearchHits, Join, Leave).

Per l'accesso alla struttura `servent_data` di un peer vengono utilizzati dei meccanismi di lock, i quali consentono di evitare inconsistenza dei dati.

All'avvio dell'applicazione un'istanza del controller si occupa di chiamare la funzione `servent_start()`, la quale ha il compito di inizializzare tutti i parametri del servente locale, di istanziare il lato server dello stesso, mettendolo in attesa di connessioni da altri peer, ed infine di avviare il suo lato client. A tale proposito all'interno della funzione vi sono tre chiamate di funzione, le quali hanno rispettivamente il compito di svolgere le attività elencate precedentemente. Di seguito vengono mostrati le dichiarazioni, presenti nell'header file, delle tre procedure con relativa spiegazione.

**servent\_init()** La funzione, come affermato in precedenza, si occupa di inizializzare i parametri del servente locale, identificando il suo indirizzo ip, numero di porta e status iniziale, i quali vengono reperiti dal file di configurazione `tortella.conf`. Inoltre viene inizializzato il mutex relativo alla sua struttura dati

```
int servent_init(char *ip, u_int4 port, u_int1 status)
```

**servent\_start\_server()** La funzione ha il compito di avviare il lato server del peer locale. Inizialmente viene creato il server, con la relativa coppia ip-porta, ricevuta da parametro; in seguito viene lanciato un thread, attivando l'esecuzione della funzione **servent\_listen()**, nella quale è presente un ciclo infinito, in cui viene creato un nuovo thread per ogni richiesta di connessione da nuovi peer.

```
int servent_start_server(char *local_ip, u_int4 local_port)
```

**servent\_init\_connection()** La funzione ha il compito di inizializzare il lato client del servente locale e di avviare una connessione per ogni peer vicino presente nella lista **init\_servent**, la quale viene riempita con i dati relativi ai peer vicini inizialmente conosciuti. Il numero di peer considerati come “vicini” per ognuno dei nodi componenti la rete è stato scelto a priori e i relativi indirizzi ip e numero di porta sono salvati in un file di configurazione (**init\_data.txt**).

```
int servent_init_connection(GList *init_servent)
```

**servent\_connect()** Questa funzione è stata realizzata per consentire la gestione della parte client di un peer, ovvero il *client thread*. E' stata strutturata considerando la modalità di funzionamento a che deve avere. Infatti, essendo il thread che consente l'invio dei pacchetti ad uno specifico peer, sono state collocate le operazioni da svolgere all'interno di un ciclo while infinito. Grazie a questa tecnica è possibile inviare pacchetti ad un peer in qualsiasi momento, in particolare basta inserire in una coda apposita (spiegata più avanti) la richiesta da effettuare a tale thread. Tale funzione soddisfa le richieste sequenzialmente, effettuando l'invio e l'attesa di risposta. Questo *client thread* consente, ai moduli superiori, di rilevare errori di invio e ricezione dei peer destinatari. Infatti, grazie ad un'ulteriore coda di messaggi di risposta si possono controllare eventuali *timeout* dei pacchetti. Tale coda viene riem-

pita dalla `servent_connect()` nel momento in cui riceve una conferma di ricezione pacchetto.

```
void *servent_connect(void *parm)
```

**servent\_listen()** La parte *server* di un peer è gestita da due funzioni (o meglio thread): `servent_listen()` e `servent_respond()`. La prima funzione in questione è la `listen`, la quale viene avviata una sola volta per ogni peer. In particolare, questa funzione non fa altro che accettare le richieste di connessione iniziali, il che si verifica quando un peer remoto invia un pacchetto di PING con fake ID al peer con la `listen` attiva. Tale `listen`, per ogni richiesta ricevuta, lancia un nuovo *servent thread* che viene gestito dal thread `servent_respond()`.

```
void *servent_listen(void *parm)
```

**servent\_respond()** La `servent_respond()` è il cuore del sistema e consente la ricezione di tutti i pacchetti inviati da uno specifico peer. Questi sono i passi generici eseguiti:

1. Il thread `servent_listen()` lancia un nuovo thread che gestisce la funzione `servent_respond()`;
2. La funzione in questione entra in un ciclo while infinito;
3. Attende la ricezione di un pacchetto HTTP sulla porta specificata;
4. Pacchetto HTTP con dati TorTella ricevuto;
5. Riempimento della struttura dati `http_packet` con i valori ricevuti nella stringa binaria;
6. Verifica correttezza pacchetto ricevuto;
7. Salta al flusso che gestisce il tipo di pacchetto ricevuto:

**Ping:** Può ricevere due tipi di Ping: con fake ID e con ID reale. Nel caso di fake ID invia il pacchetto di conferma e lancia un nuovo *client thread* che invia un Ping con ID vero. Nel caso di ID reale invia un pacchetto di conferma ed un Ping.

**Join:** Prima di tutto memorizza l'utente specificato dalla Join nella lista degli utenti della chat indicata. In seguito rinvia il pacchetto a tutti i vicini ed aggiunge delle regole di routing.

**Leave:** Rimuove l'utente dalla lista degli utenti della chat specificata. In seguito rinvia il pacchetto ai suoi vicini.

**Message:** Stampa il messaggio ricevuto.

**Search:** Effettua la ricerca localmente e rinvia il pacchetto ai vicini. In seguito aggiunge delle regole di routing.

**SearchHits:** Memorizza i risultati tornati dal pacchetto e preleva la regola di routing associata all'ID del pacchetto.

**Bye:** Elimina l'utente da tutte le strutture dati e chiude eventuali finestre PM.

8. Invia eventualmente un pacchetto di conferma ricezione, se non è stato già inviato;
9. Torna al passo 3.

Tale modulo comunica principalmente con: *Packet Manager*, *Http Manager*, *Controller*.

**servent\_send\_packet()** Per richiedere l'invio dei pacchetti verso un peer è necessario chiamare tale funzione, la quale si occupa di inserire in una coda la richiesta di invio effettuata. In particolare opera nel seguente modo:

```
void servent_send_packet(servent_data *sd) {  
    if(sd != NULL)  
        g_queue_push_tail(sd->queue, (gpointer)sd);  
}
```

Come si può notare dal codice, la funzione prede come parametro una struttura dati `servent_data`. In realtà viene passata una copia della struttura originale utilizzata per condividere memoria; in questo modo si evitano richieste sovrapposte con eventuale invio di pacchetti errati. Se non si fosse attuata tale tecnica si sarebbe potuto incorrere in errore nel seguente caso: Il peer locale chiede di inviare un pacchetto di tipo Ping ad uno specifico servente; il peer locale chiede, inoltre, di inviare un pacchetto di tipo Join allo stesso servente; la `servent_connect()` che riceve la richiesta di invio potrebbe inviare il pacchetto Ping con i dati del Join. Con il metodo adottato si evitano problemi di questo genere.

**servent\_pop\_queue()** Quest'altra funzione viene utilizzata dai *client thread* per ricevere le richieste di invio pacchetti.

```
servent_data *servent_pop_queue(servent_data *sd) {
    servent_data *servent;
    if(sd==NULL || sd->queue==NULL) {
        logger(ALARM_INFO, "[servent_pop_queue]Queue error\n");
        return NULL;
    }
    //Ciclo utilizzato per attendere la richiesta di invio pacchetti
    while((servent = g_queue_pop_head(sd->queue))==NULL) {
        usleep(100000);
    }
    return servent;
}
```

Viene effettuata una operazione di *pop* sulla coda ad intervalli di tempo precisi, in particolare ogni 100000 microsecondi. Il *polling* potrebbe essere migliorato aggiungendo il supporto ai segnali, infatti basterebbe aggiungere al posto della `usleep()` la `pthread_cond_timedwait()` la quale si sarebbe sbloccata alla ricezione di un segnale o allo scadere del timer. Con quest'ultimo metodo si avrebbe un leggero incremento della prestazioni. Sono state realizzate push e pop anche per la coda dei messaggi di risposta, ma non verranno riportati essendo molto simili (ovvero `servent_append_response()` e `servent_pop_queue()`).

**timer\_thread()** Thread utilizzato per gestire il meccanismo di failure detection e per pulire la lista dei pacchetti ricevuti. L'intervallo di tempo è impostato nel file di configurazione. Tale funzione fa un elevato uso delle routine appena presentate.

```
void *servent_timer(void *parm)
```

## 4.7 Data Manager

Il modulo datamanager si occupa di gestire tutti i meccanismi di accesso e di ricerca di una o più chat. Le principali strutture dati utilizzate sono chatclient e chat; la prima contiene le informazioni di un utente connesso ad una chat, invece la seconda mantiene le informazioni relative ad una specifica chat, considerando l'ID, il titolo, la lista di tutti gli utenti presenti in quel momento ed infine un mutex utilizzato per la gestione dell'accesso concorrente alla chat stessa.

```
struct chatclient {
    u_int8 id;
    char *nick;
    char *ip;
    u_int4 port;
};
typedef struct chatclient chatclient;

struct chat {
    u_int8 id;
    char *title;
    GHashTable *users;
    pthread_mutex_t mutex;
};
typedef struct chat chat;
```

Durante la fase di progettazione del modulo era stata prevista anche la possibilità di reperire o salvare le informazioni relative alle chat da un generico file, per questo motivo si è deciso di mantenere tutte le funzioni che real-

izzavano tale meccanismo, in previsione di ulteriori sviluppi successivi. Di seguito viene riportato l'elenco dei prototipi di tali funzioni:

```
int write_to_file(const char *filename, chat *chat_str, u_int4 mode)

int write_all(u_int4 mode)

int read_from_file(const char *filename)

int read_all(void)
```

Dall'analisi dettagliata del modulo è possibile identificare due sezioni, la prima, come affermato precedentemente, si occupa degli accessi alle varie chat, invece la seconda ha il compito di gestire tutti gli algoritmi di parsing dei dati. Di seguito sono riportate le funzioni principali utilizzate per la gestione di una chat:

```
int data_add_chat(u_int8 id, const char *title)

int data_del_chat(u_int8 id)

int data_add_user(u_int8 id, const char *nick, const char *ip, u_int4 port)

int data_add_users_to_chat(u_int8 chat_id, GList *users)

int data_del_user(u_int8 id)

int data_del_user_from_chat(u_int8 chat_id, u_int8 id)

chat *data_search_chat(const char *title)

chatclient *data_search_chatclient(const char *nick)
```

Tutte le funzioni, sopra riportate, sono state implementate tenendo conto del meccanismo di memorizzazione utilizzato. Nell'header file, in particolare, sono state definite due hashtable, le quali hanno il compito di memorizzare tutte le chat e tutti gli utenti presenti nella rete. In particolare tutte le funzioni adibite al parsing si occupano o di scandire il buffer contenente i dati memorizzandoli nelle relative strutture dati, o di effettuare il processo conversione inverso. Le informazioni relative alle chat sono salvate nel seguente

formato:

```
:  
|chat_id;title;  
user_id;nickname;indirizzo ip;numero porta;  
|chat_id;title;  
user_id;nickname;indirizzo ip;numero porta;  
:
```

Come è possibile notare ogni chat è delimitata dal carattere separatore |, il quale caratterizza l'inizio e la fine di ogni dato relativo alla chat identificata con il campo `chat_id` e `title`. Nelle righe successive sono mostrati tutti gli utenti connessi alla chat con il relativo *ID*, *nickname*, *indirizzo*, *ip* e numero di porta del socket associato. La decisione di suddividere in tal modo ogni informazione nasce principalmente dalla necessità di identificare in modo semplice ogni dato e dalla necessità di sviluppare un parser abbastanza semplice e veloce.

## 4.8 Init

Il modulo in questione consente di effettuare il bootstrap dell'applicazione, poichè permette ad un peer di conoscere i propri vicini (appositamente memorizzati in un file). Il file dei peer vicini è strutturato nel seguente modo:

```
ip;porta;  
ip;porta;  
...
```

Per la memorizzazione dei record presenti all'interno del file è stata utilizzata la seguente struttura dati:

```
struct init_data {  
    char *ip;  
    u_int4 port;  
};  
typedef struct init_data init_data;
```

e per la gestione di questi sono state utilizzate le seguenti funzioni.



**init\_char\_to\_initdata()** Riceve come parametro un buffer contenente un record del file e istanzia la struttura dati settando in modo opportuno ip e porta.

```
init_data *init_char_to_initdata(char *buffer)
```

**init\_read\_file()** Legge il file in cui sono contenuti i record dei vicini e aggiunge tutti i peer presenti all'interno del file in una lista contenente strutture di tipo `init_data`. Si serve della funzione `init_char_to_init_data()` per l'istanziamento delle strutture dati.

```
GList *init_read_file(const char *filename)
```

## 4.9 Controller

Il *Controller* ha il compito di inizializzare ed arrestare il sistema ed inoltre semplificare le interazioni tra la *GUI* e il *Servent*, in modo particolare per gestire al meglio i segnali scatenati da un utente durante l'esecuzione dell'applicazione. A tal proposito, si può operare una suddivisione delle funzioni per renderne più semplice la comprensione:

- funzioni per inizializzazione e chiusura dell'applicazione;
- funzioni per l'aggiornamento corretto dell'interfaccia grafica;
- funzioni per l'invio dei messaggi al servent.

**controller\_init()** Gestisce la fase di inizializzazione, che è composta dai seguenti passi:

1. lettura del file di configurazione;
2. avvio del logger;

3. lettura del file contenente i vicini conosciuti e memorizzazione di questi nell'apposita lista;
4. avvio del server;
5. avvio del timer dedicato alla gestione del meccanismo di failure detection.

```
int controller_init(const char *filename, const char *cache)
```

**controller\_exit()** Contrariamente alla `controller_init()`, la funzione in questione ha il compito di gestire la chiusura dell'applicazione, effettuando i seguenti passi:

1. kill di tutti i thread del server;
2. chiusura del file di log;

```
int controller_exit()
```

Del primo blocco di funzioni fa parte anche la `controller_init_gui()` (si veda a tal proposito l'appendice). Il secondo blocco comprende tutte le funzioni che permettono ad esempio di: visualizzare un messaggio ricevuto, aggiornare l'elenco dei partecipanti alla chat, aprire una nuova finestra in caso di ricezione di un messaggio privato. Di seguito si presentano le funzioni utilizzate:

```
int controller_change_status(u_int1 status)

int controller_manipulating_status(u_int8 user_id, u_int1 status)

int controller_create(const char *title)

int controller_add_user_to_chat(u_int8 chat_id, u_int8 id)

int controller_rem_user_from_chat(u_int8 chat_id, u_int8 id)

int controller_add_msg_to_chat(u_int8 chat_id, char *msg)

int controller_add_msg(u_int8 sender_id, char *msg)
```

Le funzioni facenti parte dell'ultimo blocco sono fondamentali per il corretto funzionamento dell'applicazione, poichè preparano le strutture dati dei server e permettono l'invio dei pacchetti da parte del server, inserendo le strutture dati in un'apposita coda in caso di invio e rimuovendo da un'altra in caso di ricezione di messaggi di risposta. Di seguito si presentano le funzioni utilizzate:

```
int controller_send_chat_users(u_int8 chat_id, u_int4 msg_len, char *msg)

int controller_send_subset_users(u_int8 chat_id, u_int4 msg_len, char *msg,
    GList *users)

int controller_send_pm(u_int4 msg_len, char *msg, u_int8 recv_id)

int controller_leave_all_chat()

int controller_connect_users(GList *users)

int controller_check_users_con(GList *users)

int controller_send_bye()

int controller_receive_bye()

u_int8 controller_search(const char *query)

int controller_join_flooding (u_int8 chat_id)

int controller_leave_flooding(u_int8 chat_id)
```

A tal proposito si presenta la funzione `send_join_packet()` come esempio:

**send\_join\_packet()** Il pacchetto di JOIN viene inviato in flooding, quindi per ogni peer conosciuto vengono effettuati i seguenti passi:

1. recupero delle informazioni associate all'utente a cui si vuole inviare il pacchetto;
2. controllo per evitare l'auto invio del pacchetto e l'invio ai fake id;
3. copia dei dati del peer in una struttura dati temporanea

4. impostazione dei campi della struttura dati temporanea (ad esempio: nickname, id utente, status utente, ip, porta, etc.);
5. inserimento della struttura dati nell'apposita coda che verrà gestita dal server per l'invio del pacchetto
6. attesa della ricezione del messaggio di ok da parte del peer remoto
7. aggiornamento dei dati e della GUI.

**controller\_send\_bye()** Tale funzione viene chiamata dall'applicazione quando viene richiesta la chiusura, in particolare effettua le seguenti operazioni:

1. Invia il LEAVE a tutti i peer vicini per tutte le chat a cui è connesso;
2. Attende le risposte;
3. Invia il BYE a tutti i peer vicini;
4. Attende le risposte;
5. Termina il processo.

### 4.10 Utils

Il modulo Utils contiene la funzione che viene utilizzata per la generazione degli ID, ovvero:

```
u_int8 generate_id(void) {
    char *addr;
    addr = get_mac_addr();
    int i=0, j=10;
    unsigned long res = 0;
    for (; i<6; i++, j*=10) {
        res += addr[i]*j;
    }
    res *= random();
    return (conf_get_gen_start()+((random())^res))*(res/2);
}
```

La generazione dell'identificativo univoco dei pacchetti e degli utenti, di tipo `unsigned long long`, avviene tramite una combinazione del *MAC* della macchina, il valore generato dalla funzione pseudo-randomica `random()`, e dall'ID iniziale presente nel file di configurazione di ogni peer.

## 4.11 Conf Manager

Tale modulo consente la gestione del file di configurazione, consentendo di recuperare i valori desiderati tramite un insieme di semplici funzioni.

## 4.12 Common

Contiene le definizioni dei tipi di dati creati appositamente per *TorTella*.

## 4.13 GUI

Essendo il modulo GUI di poca importanza relativamente al resto, si rimanda la lettura del codice in appendice.

# Capitolo 5

## Ambiente di Sviluppo

L'applicazione è stata sviluppata in ambiente *Unix*, utilizzando come distribuzione Linux: *Ubuntu 8.04* e *Debian Lenny* per l'implementazione e relativa fase di testing. L'ambiente di sviluppo utilizzato è stato *Anjuta 2.4* con il supporto di alcune librerie esterne (*libgtk 2.12.9-3* -*libglib 2.16.3-1*) per l'implementazione dell'interfaccia grafica dell'applicazione, e per l'utilizzo di routine del C quali liste, code ed hashtable.

### 5.1 Concorrenza

Per quanto riguarda la concorrenza si è preferito l'uso dei thread (*programmazione multithreading*) invece dei processi (*multiprocess*), il che comporta dei vantaggi e degli svantaggi. I vantaggi sono:

- La creazione di un nuovo thread è in genere più veloce della creazione di un nuovo processo in quanto la prima utilizza lo stesso *process address space* del processo creatore.
- Il tempo di terminazione di un thread piuttosto che di un processo risulta essere minore.
- L'utilizzo dello stesso process address space da parte di due thread

comporta uno switching tra gli stessi molto più veloce che tra due processi.

- Il tempo di comunicazione tra due thread, per quanto detto, risulta certamente essere minore del tempo richiesto per la comunicazione tra due processi. Questo perchè basta utilizzare delle variabili globali invece dei meccanismi di condivisione di memoria.
- Utilizzando il multithreading si sfruttano maggiormente i *processori multicore*.

Svantaggi:

- Lo spazio di memoria riservato ad un thread è più ristretto rispetto allo spazio di un processo.
- Si devono utilizzare molti meccanismi di accesso esclusivo per evitare l'incoerenza dei dati condivisi.

Lo sviluppo di un software peer to peer necessita dell'implementazione di un sistema che consenta di svolgere più operazioni contemporaneamente. Per questo motivo sono stati presi in considerazione l'utilizzo dei thread e dei processi e sulla base dei pro e contro dei due si è optato per i thread. Per implementare i thread si è utilizzata la libreria pthread.

## 5.2 Libreria pthread

La libreria *pthread* è conforme allo standard *POSIX*, ed offre delle funzioni e dei dati particolari che consentono alle applicazioni di utilizzare diversi flussi di esecuzione (thread per l'appunto) all'interno di un solo processo. Innanzitutto è stata utilizzata la funzione

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start_routine)(void *), void *arg)
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start_routine)(void *), void *arg)
```

che consente la creazione e l'esecuzione di un nuovo thread. La funzione `pthread_create()` accetta come parametro un riferimento ad un dato di tipo `pthread_t`, nel quale viene memorizzato l'identificativo del thread che viene creato. Il secondo parametro, di tipo `pthread_attr_t`, consente di valorizzare i parametri del nuovo thread. Se tale valore è nullo, vengono utilizzati dei parametri di default. Il terzo parametro è un puntatore ad una routine precedentemente definita, che contiene le istruzioni che vengono eseguite in maniera indipendente dalle altre grazie al nuovo thread. Nel quarto parametro possono, infine, essere specificati i parametri che devono essere passati a tale routine per la sua esecuzione, raggruppati in una struttura (se presenti in un numero maggiore di uno). Una volta che più thread sono stati messi in esecuzione, occorre dedicare particolare attenzione alla loro sincronizzazione nel caso di accesso a risorse condivise. A tal proposito sono stati utilizzati due diversi tipi di dati definiti dalla libreria "pthread", che vengono riportati di seguito:

**pthread\_mutex\_t** : semaforo per l'accesso in mutua esclusione ad una risorsa ad esso associata;

**pthread\_rwlock\_t** : semaforo per l'accesso in mutua esclusione che distingue tra accessi in lettura ed accessi in scrittura.

Questi tipi di dati sono stati spesso utilizzati all'interno delle componenti realizzate, poiché hanno consentito la gestione dei conflitti in cui sarebbero potuti occorrere i thread. Per quanto riguarda la `pthread_mutex_t` si è fatto uso solo della funzione `pthread_mutex_init` riportata di seguito:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr)
```

Si è deciso di utilizzare delle funzioni che operano una la distinzione tra access in lettura e quello in scrittura in modo da evitare inutili serializzazioni dei dati:

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```



```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);  
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

La prima funzione applica un lock in scrittura al *rwlock*. Il thread chiamante acquisisce il lock se nessun altro thread possiede il read-write lock *rwlock*, altrimenti il thread è bloccato fin quando non acquisisce il lock. La funzione `pthread_rwlock_rdlock` applica un lock in lettura a *rwlock*. Il thread chiamante acquisisce il lock in lettura se non esiste nessun lock in scrittura bloccati, altrimenti il thread viene bloccato fino all'acquisizione della risorsa. L'ultima funzione viene chiamata semplicemente per rilasciare il lock su *rwlock*. La libreria `pthread` è stata utilizzata, e lo è ancora, in molte applicazioni che necessitano del supporto di multipli flussi di esecuzione al suo interno; a tal proposito, è possibile reperire una grande mole di informazioni a riguardo sul Web, che possono essere utilizzate per meglio comprendere l'utilizzo che se ne è fatto.

### 5.3 Liste, Hashtable e Code

L'utilizzo di diversi tipi di collezioni, al fine di gestire qualsiasi tipo di informazione riguardante o la chat o gli utenti o i diversi tipi di pacchetto, ha caratterizzato l'intero processo di implementazione. In una prima fase si è deciso di utilizzare una versione ad-hoc di tali collezioni, che sono risultate poco efficienti dal punto di vista prestazionale. A tale ragione sono state utilizzate le strutture fornite dalla libreria *glib*, le quali forniscono un'implementazione più performante delle elementari funzioni di manipolazione. In particolare sono state utilizzate le code per gestire il meccanismo di invio concorrente dei pacchetti da parte di più peer; ovvero prima di inviare un pacchetto il server aggiunge questo nella coda, associata alla sua struttura dati, per poi inviare in modo sequenziale tutti i pacchetti presenti nella coda stessa.

# Capitolo 6

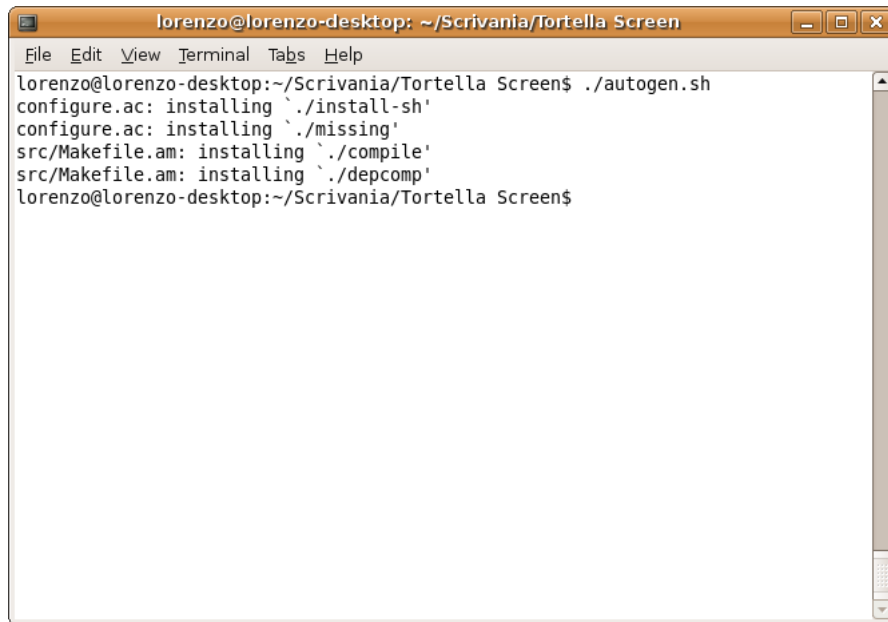
## Installazione ed Esecuzione

Nel presente capitolo verrà fornita una rapida guida d'installazione per l'utente e verrà illustrato un esempio di funzionamento dell'applicazione.

### 6.1 Installazione

Per generare il Makefile bisogna seguire i seguenti passi:

1. Posizionarsi nella radice del progetto;
2. Eseguire il file `./autogen.sh`;

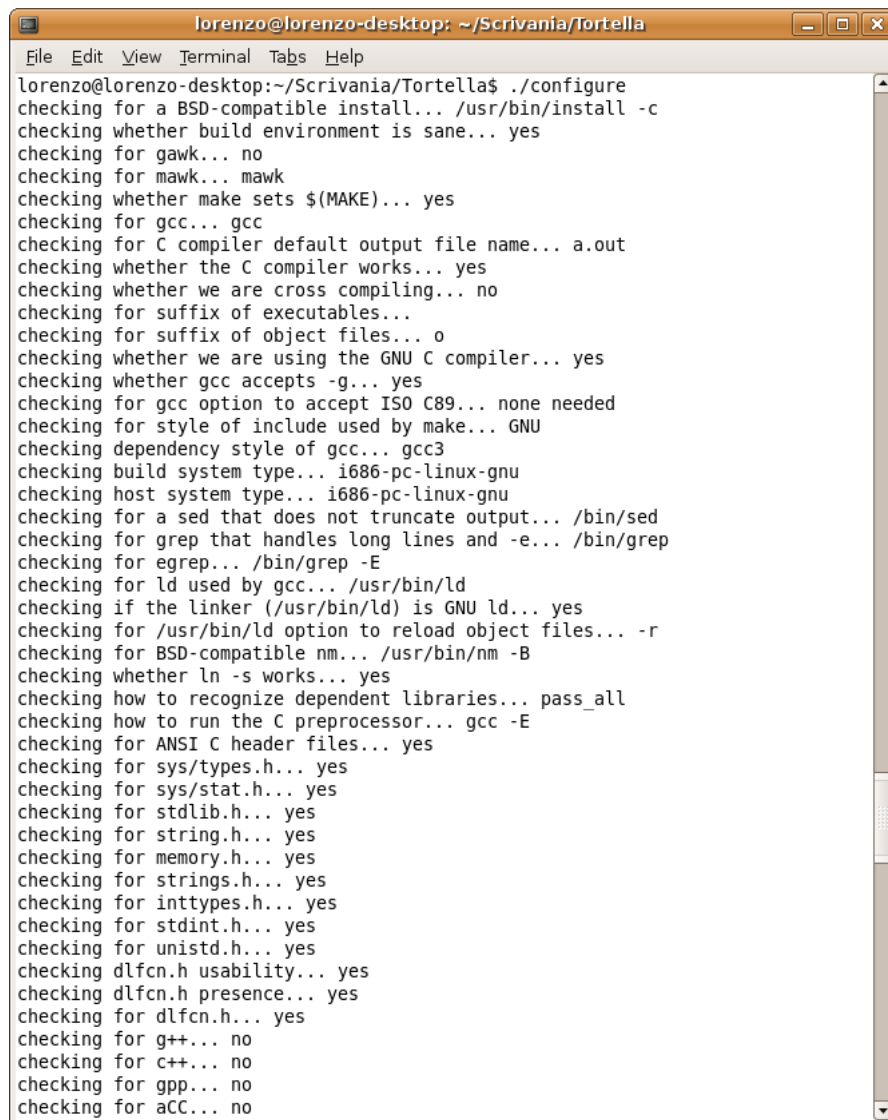


The image shows a terminal window titled "lorenzo@lorenzo-desktop: ~/Scrivania/Tortella Screen". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The terminal content shows the following commands and output:

```
lorenzo@lorenzo-desktop:~/Scrivania/Tortella Screen$ ./autogen.sh
configure.ac: installing `./install-sh'
configure.ac: installing `./missing'
src/Makefile.am: installing `./compile'
src/Makefile.am: installing `./depcomp'
lorenzo@lorenzo-desktop:~/Scrivania/Tortella Screen$
```

Figura 6.1: autogen

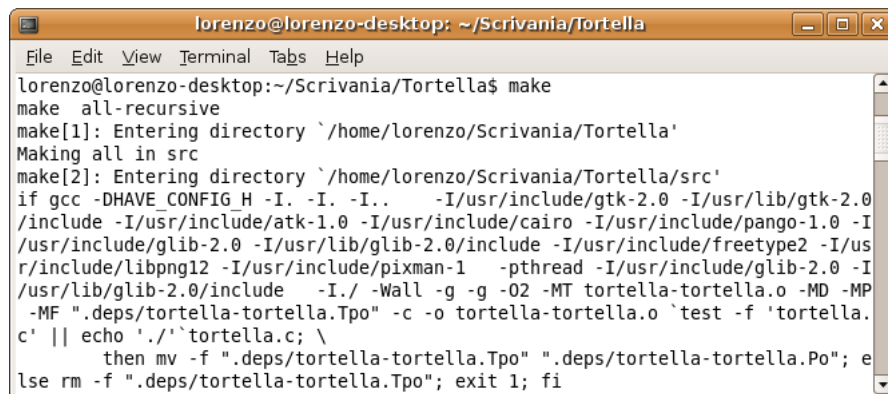
3. Eseguire il `./configure` sempre nella radice del progetto;



```
lorenzo@lorenzo-desktop: ~/Scrivania/Tortella
File Edit View Terminal Tabs Help
lorenzo@lorenzo-desktop:~/Scrivania/Tortella$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking for a sed that does not truncate output... /bin/sed
checking for grep that handles long lines and -e... /bin/grep
checking for egrep... /bin/grep -E
checking for ld used by gcc... /usr/bin/ld
checking if the linker (/usr/bin/ld) is GNU ld... yes
checking for /usr/bin/ld option to reload object files... -r
checking for BSD-compatible nm... /usr/bin/nm -B
checking whether ln -s works... yes
checking how to recognize dependent libraries... pass_all
checking how to run the C preprocessor... gcc -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking dlfcn.h usability... yes
checking dlfcn.h presence... yes
checking for dlfcn.h... yes
checking for g++... no
checking for c++... no
checking for gpp... no
checking for aCC... no
```

Figura 6.2: configure

4. Ora il Makefile è pronto, per compilare basta eseguire make dalla radice del progetto.



```
lorenzo@lorenzo-desktop: ~/Scrivania/Tortella
File Edit View Terminal Tabs Help
lorenzo@lorenzo-desktop:~/Scrivania/Tortella$ make
make all-recursive
make[1]: Entering directory `/home/lorenzo/Scrivania/Tortella'
Making all in src
make[2]: Entering directory `/home/lorenzo/Scrivania/Tortella/src'
if gcc -DHAVE_CONFIG_H -I. -I. -I. -I/usr/include/gtk-2.0 -I/usr/lib/gtk-2.0
/include -I/usr/include/atk-1.0 -I/usr/include/cairo -I/usr/include/pango-1.0 -I
/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include -I/usr/include/freetype2 -I/us
r/include/libpng12 -I/usr/include/pixman-1 -pthread -I/usr/include/glib-2.0 -I
/usr/lib/glib-2.0/include -I./ -Wall -g -g -O2 -MT tortella-tortella.o -MD -MP
-MF ".deps/tortella-tortella.Tpo" -c -o tortella-tortella.o `test -f 'tortella.
c' || echo './'`tortella.c; \
    then mv -f ".deps/tortella-tortella.Tpo" ".deps/tortella-tortella.Po"; e
lse rm -f ".deps/tortella-tortella.Tpo"; exit 1; fi
```

Figura 6.3: make

E' anche possibile generare la documentazione del codice tramite le seguenti operazioni:

1. Posizionarsi nella directory `./docs`;
2. Eseguire il comando `doxygen doxygen.conf`.

La documentazione verrà generata nella directory `./docs/documentation` nei formati *latex*, *html* e *manpages*.

## 6.2 Configurazione

Prima di avviare il programma è necessario che i file vengano creati correttamente. A tal proposito si presenta un esempio di come dev'essere strutturato il file di configurazione e un esempio di come dev'essere strutturato il file contenente i vicini conosciuti:

```
#TorTella Configuration file

#LOGGER

verbose = 3;
```

```
#SOCKET
qlen = 5;
buffer_len = 4096;

#PACKET
path = /tmp/tortella1;

#UTILS
gen_start = 100000;

#SERVENT
max_len = 4000;
max_thread = 20;
max_fd = 100;
timer_interval = 20;
new_connection_counter = 10000;

#SUPERNODE
datadir = ./data;

#SERVENT
local_ip = 127.0.0.1;
local_port = 2110;
nick = simo;
```

I primi due parametri sono relativi alla caratterizzazione del backlog massimo e della dimensione massima di un blocco di dati, di un socket di connessione. In particolare il parametro `qlen` è utilizzato per impostare il numero massimo di connessioni che possono essere gestite da un server, invece `buffer_len` serve per imporre la dimensione massima del buffer che deve contenere una parte del pacchetto ricevuto. Tale frammento sarà poi concatenato con i

restanti frammenti. La variabile `path`, serve invece per memorizzare il percorso relativo in cui vengono salvati i log file, ampiamente utilizzato durante la fase di debugging e testing dell'intera applicazione. Il parametro `gen_start`, viene adottato nel meccanismo di identificazione univoco dell'utente, ovvero il valore impostato nel file serve come criterio di comparazione per la validazione del identificativo, associato all'utente stesso. In particolare distingue i fake ID con gli ID reali. Il parametro `timer_interval` indica l'intervallo di tempo che il meccanismo di failure detection deve far passare tra i controlli. Il parametro `new_connection_counter` indica il valore iniziale dei fake ID. Il parametro `verbose` indica il livello di dettaglio delle stampe. Infine gli ultimi tre parametri sono necessari nella fase di inizializzazione di ogni peer componente la rete, in quanto forniscono il relativo indirizzo ip, numero di porta del socket e nickname da assegnare all'utente. Si consiglia all'utente inesperto di cambiare solamente gli ultimi tre parametri. Per quanto concerne il file di inizializzazione, questo dovrebbe essere strutturato in questo modo:

```
127.0.0.1;2110;  
127.0.0.1;2120;
```

Una volta che i file sono stati creati, è necessario posizionarli nelle directory corrette:

- Il file di esecuzione nella directory `src`;
- file di configurazione in `src/conf`;
- file di init in `src/data`.

Se i file sono nelle posizioni corrette si può avviare l'esecuzione. Per l'esecuzione su una sola macchina è necessario avviare almeno due peer seguendo questi semplici passi:

1. Posizionarsi in `src`;

2. Eseguire `./ tortella ./conf/ tortella .conf;`
3. Eseguire `./ tortella ./conf/ tortella1 .conf ./data/init\_data.`

## 6.3 Esecuzione

L'esecuzione qui presentata viene eseguita su macchina locale simulando la presenza di tre peer nella rete. I tre peer, che per comodità verranno chiamati `simo`, `lore`, `ibra` saranno inizialmente connessi nel seguente modo: `simo` non conosce alcun peer vicino, `lore` e `ibra` invece conosceranno come unico vicino `simo`. I tre peer avviano l'applicazione e `simo` crea una chat dal nome `prova`:

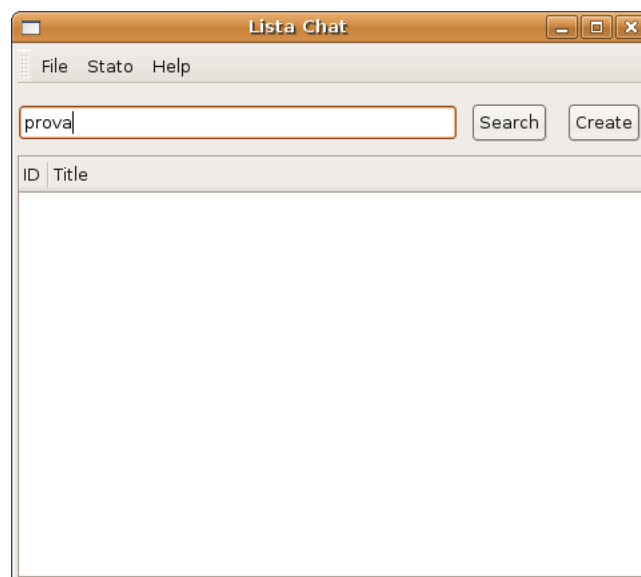


Figura 6.4: Creazione della chat



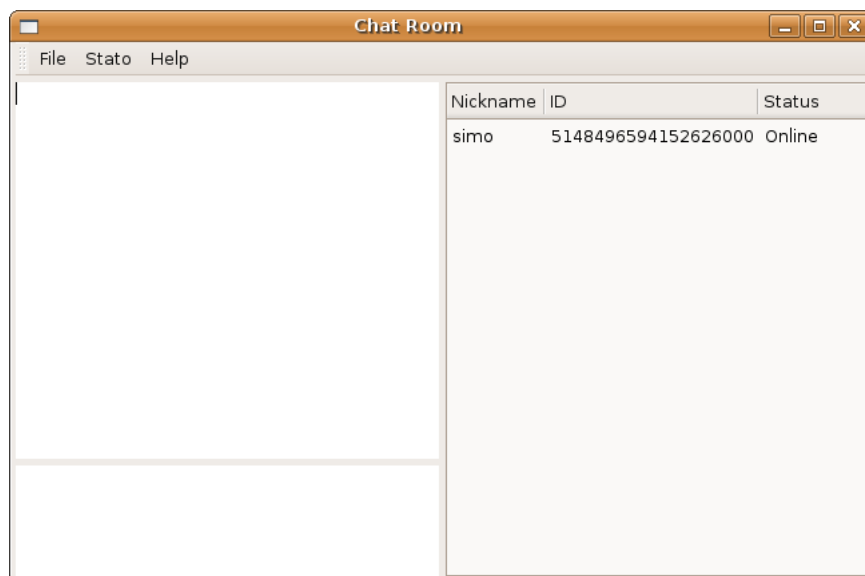


Figura 6.5: Apertura della chat

Lore cerca la chat prova e si connette:

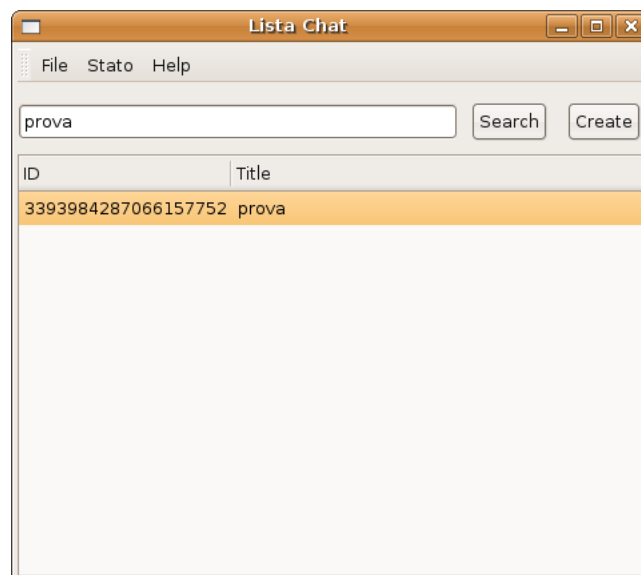


Figura 6.6: Ricerca della chat

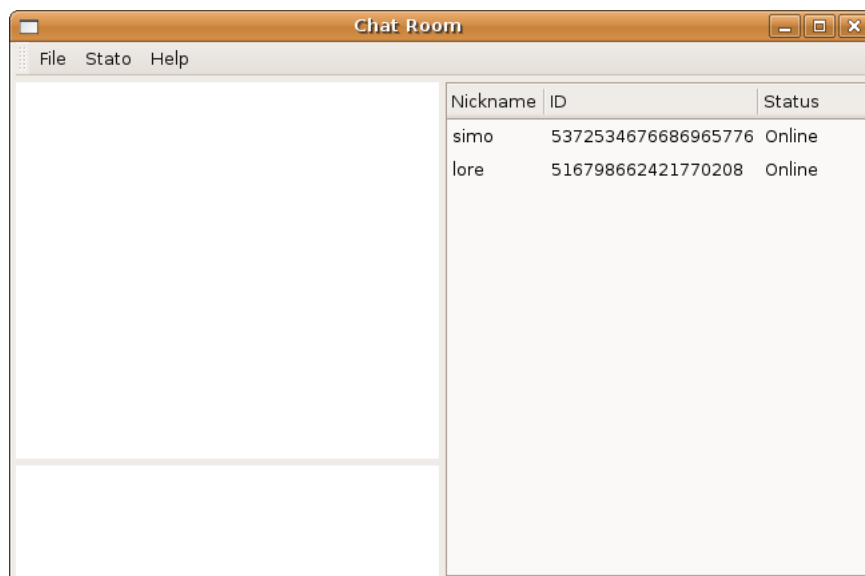


Figura 6.7: Join di lore

ibra crea una chat di nome pippo

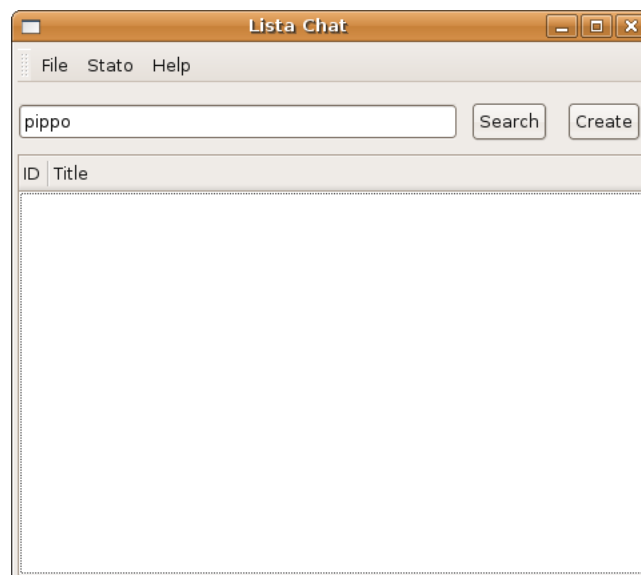


Figura 6.8: Creazione di una nuova chat

Nell'attesa che qualcuno si connetta alla chat appena creata, ibra cerca la chat prova e si connette

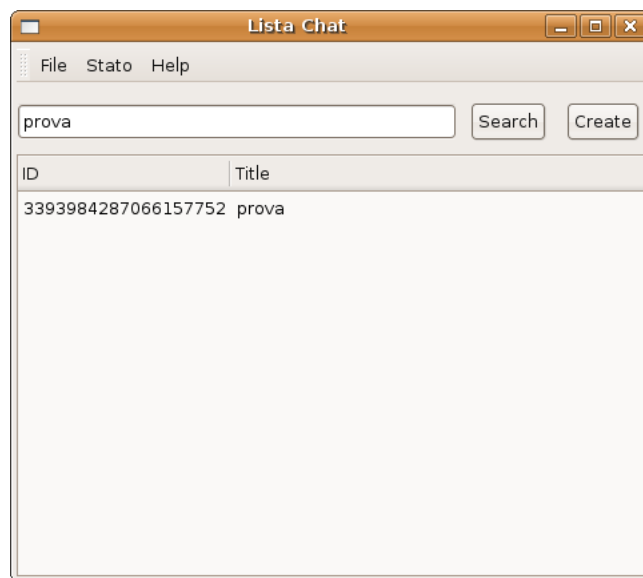


Figura 6.9: Ricerca della chat prova

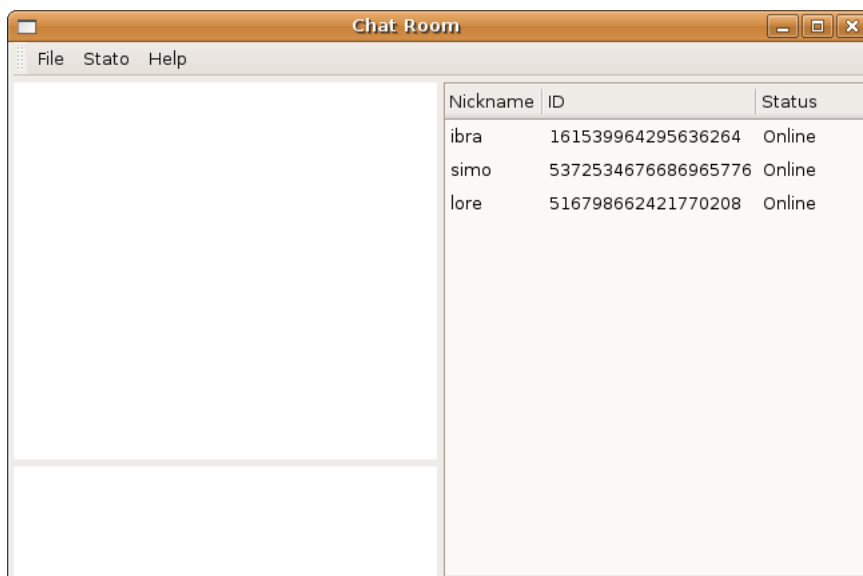


Figura 6.10: Join di ibra

I tre utenti comunicano tra di loro

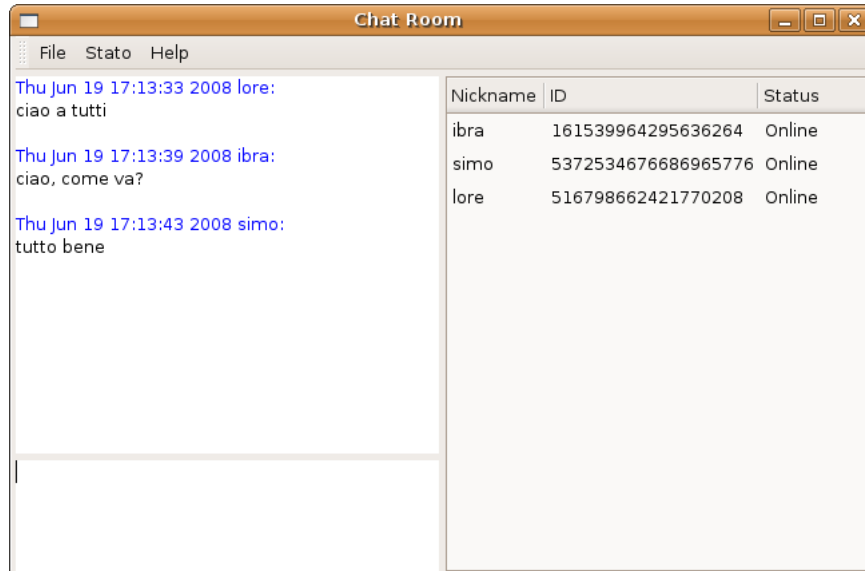


Figura 6.11: Conversazione tra gli utenti

lore invia un messaggio solo a simo (in realtà avrebbe potuto mandarlo a più utenti, ma essendo la chat composta da soli 3 utenti non avrebbe avuto senso mandare il messaggio a tutti)

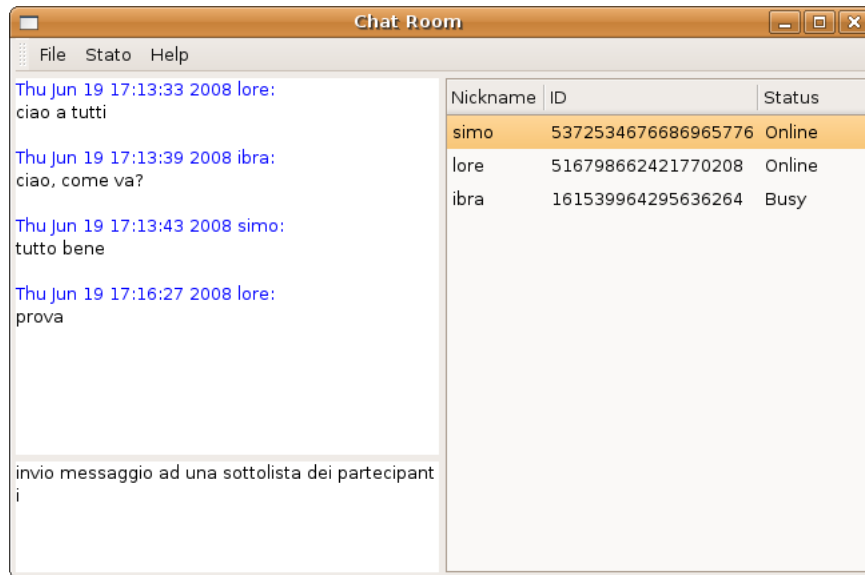


Figura 6.12: Invio di un messaggio ad un sottoinsieme degli utenti

lore invia un messaggio privato a ibra, che viene ricevuto correttamente da quest'ultimo



Figura 6.13: Invio di un messaggio privato

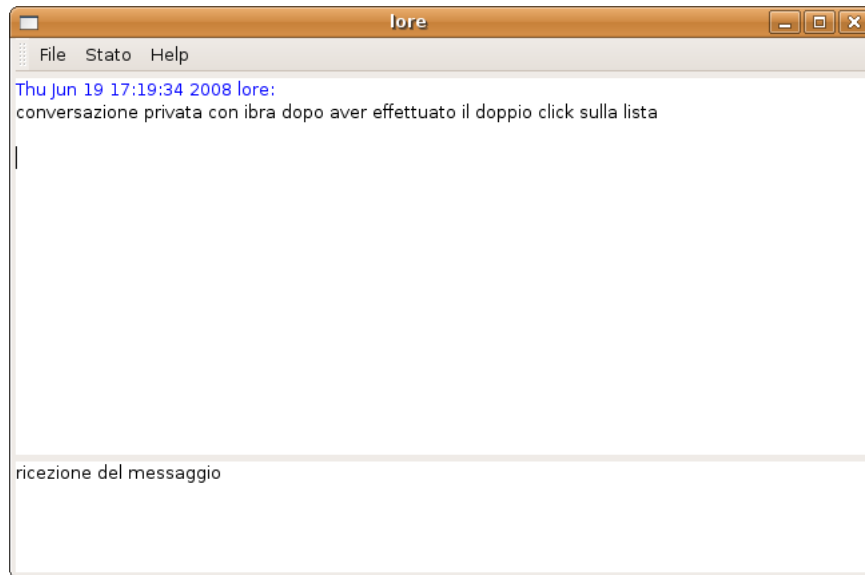


Figura 6.14: Ricezione di un messaggio privato

infine lore effettua il leave dalla chat e in seguito simo chiude l'applicazione inviando un bye ad ibra

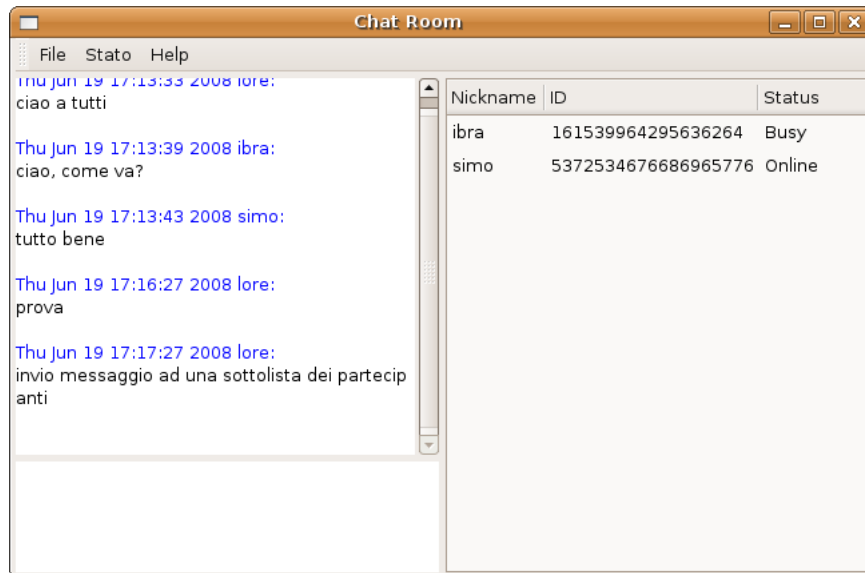


Figura 6.15: Leave dalla chat prova

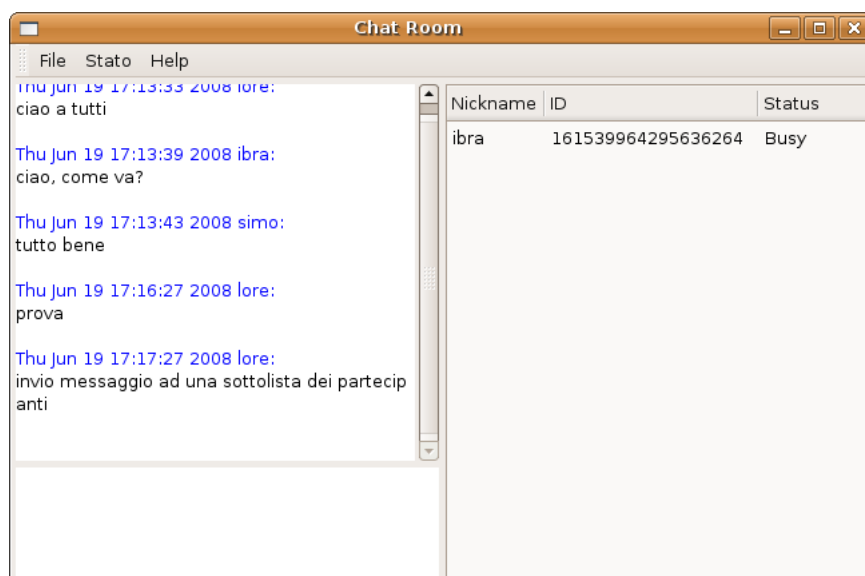


Figura 6.16: Ricezione di un bye

## Capitolo 7

### Conclusioni e sviluppi futuri

E' stato sviluppato un software *P2P* per lo scambio di messaggi tra due o più utenti facendo uso del linguaggio C e delle API del socket di *Berkeley*. Le API dei socket di Berkeley sono state utilizzate per la gestione delle connessioni tra peer, permettendo l'instaurazione di connessioni di tipo persistente; inoltre è stato inserito un livello software ex-novo, il quale si comporta come un'interfaccia d'accesso per la comunicazione tra lo strato TCP e quello HTTP su cui si basa l'intero processo di scambio di pacchetti. I vari test che sono stati eseguiti hanno dimostrato che il sistema riesce a mantenere un buon grado di stabilità e velocità, senza mai trovarsi in uno stato di inconsistenza. Nonostante ciò effettuando un'analisi più dettagliata si possono notare alcune inefficienze, soprattutto per quello che riguarda la gestione dell'invio concorrente dei pacchetti. A tale proposito è stato deciso di adottare un meccanismo di sequenzializzazione dei pacchetti basato sull'utilizzo di particolari strutture come le code e su un algoritmo di polling basato sullo sleep dei *client thread*. In questo modo ogni client non rischierà di perdere alcun segnale di notifica, proveniente dal lato server, per l'invio di un pacchetto. Una possibile risoluzione al fine di evitare lo sleeping del thread potrebbe essere l'introduzione di un `pthread_cond_timedwait()`, ovvero una funzione che gestisce la sospensione del thread solo nel caso di ricezione di un segnale o allo scadere del timer. Per quello che riguarda lo sviluppo di funzionalità



aggiuntive si potrebbe pensare alla possibilità di effettuare il trasferimento di file tra gli utenti di una chat, funzionalità in parte già sviluppata. Un'ulteriore miglioria all'applicazione potrebbe essere fatta aggiungendo il supporto ad una sorta di *WebCache*, ovvero la possibilità di reperire la lista dei peer (necessari al *bootstrap*) da un server centralizzato.

# Elenco delle figure

2.1	Descrittore Ping . . . . .	12
2.2	Descrittore Join . . . . .	13
2.3	Descrittore Leave . . . . .	13
2.4	Descrittore Message . . . . .	13
2.5	Descrittore Search . . . . .	14
2.6	Descrittore SearchHits . . . . .	14
2.7	Descrittore List . . . . .	14
2.8	Descrittore ListHits . . . . .	15
2.9	Pacchetto TorTella . . . . .	15
2.10	Header TorTella . . . . .	15
2.11	Header Gnutella . . . . .	16
3.1	BootStrap peer . . . . .	19
3.2	Search Flooding . . . . .	21
3.3	Search Flooding . . . . .	22
3.4	Join Flooding . . . . .	23
3.5	Failure Detection . . . . .	25
4.1	Schema Architetturale . . . . .	28
6.1	autogen . . . . .	58
6.2	configure . . . . .	59
6.3	make . . . . .	60
6.4	Creazione della chat . . . . .	63

## *ELENCO DELLE FIGURE*

---

6.5	Apertura della chat . . . . .	64
6.6	Ricerca della chat . . . . .	64
6.7	Join di lore . . . . .	65
6.8	Creazione di una nuova chat . . . . .	65
6.9	Ricerca della chat prova . . . . .	66
6.10	Join di ibra . . . . .	66
6.11	Conversazione tra gli utenti . . . . .	67
6.12	Invio di un messaggio ad un sottoinsieme degli utenti . . . . .	68
6.13	Invio di un messaggio privato . . . . .	68
6.14	Ricezione di un messaggio privato . . . . .	69
6.15	Leave dalla chat prova . . . . .	69
6.16	Ricezione di un bye . . . . .	70

# Indice analitico

- Anjuta 2.4, 53
- architettura decentralizzata, 10
- Architettura P2P, 5
- backward routing, 11, 20, 21
- bootstrap, 72
- client thread, 71
- client/server, 6
- Debian Lenny, 53
- failure detection, 4
- file di configurazione, 5
- file-sharing, 6
- file-storage, 7
- flooding, 20
- gen\_start, 20, 62
- glib, 56
- join, 4
- killer-application, 7
- leave, 4
- libglib 2.16.3-1, 53
- libgtk 2.12.9-3, 53
- list, 4
- master/slave, 6
- MAX, 26
- multiprocess, 53
- Napster, 8
- new\_connection\_counter, 62
- nickname, 5, 12
- path, 62
- point-of-failure, 8
- POSIX, 54
- process address space, 53
- processori multicore, 54
- programmazione multithreading, 53
- protocollo HTTP, 5
- pthread\_attr\_t, 55
- pthread\_create(), 55
- pthread\_mutex\_init, 55
- pthread\_mutex\_t, 55
- pthread\_rwlock\_rdlock, 56
- pthread\_t, 55
- ptrhead, 54
- qlen, 61
- query, 10
- random(), 52
- routing, 21

rwlock, 56

servent, 8

supernodi, 8

TIME TO LIVE, 20

timeout, 4

timer\_interval, 62

timestamp, 26

TorTella, 12

TTL, 20

Ubuntu 8.04, 53

Unix, 53

verbose, 62

web chat, 20

WebCache, 72