Milestone Chat System

Indice

Introduzione

La richiesta prevedeva la costruzione di un sistema di chat in C++ che presentasse le seguenti feature

- Infrastruttura p2p
- Utilizzo del protocollo UDP
- Comunicazioni asincrone tra i peers
- Gestione di packet loss e packet reordering
- Discovery automatica degli altri peer (opzionale)

Per poter realizzare il software richiesto è stata prima progettata tutta la sezione relativa al networking (mesh, protocollo, ecc), in modo da poter appoggiare la progettazione del software stesso su un protocollo in grado di fronteggiare tutte le problematiche emerse in fase di analisi.

Successivamente si è proceduto con la progettazione del codice, utilizzando un paradigma a spirale per poter implementare sequenzialmente le varie feature mantenendo, nel frattempo, il software completamente testabile (questo processo si sposa molto bene anche col paradigma Git-Flow, che è stato utilizzato per il versioning durante lo sviluppo).

Infine, al termine dello sviluppo, si è provveduto ad evidenziare i limiti noti del software consegnato, corredati di eventuali proposte di sviluppo future.

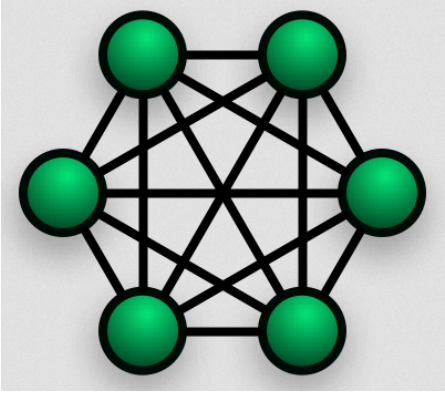
Analisi networking

Come primo passo è necessario evidenziare la mesh di comunicazione e le varie entità che caratterizzano il grafo di comunicazione su cui si basa una chat, per poi passare ad una prima ipotesi di protocollo di comunicazione

Mesh di comunicazione

Da richiesta la struttura della chat dev'essere peer to peer, quindi ogni client dev'essere in grado di comunicare con tutti gli altri client.

La mesh, pertanto, è una mesh completa, con una connessione diretta tra ogni coppia di client partecipe alla chat, come in figura sottostante.



Questo comporta già dei problemi, a seconda della tipologia di rete prescelta

- **LAN**: una LAN, per quanto grande possa essere, ha un numero relativamente basso di nodi. Per la rilevazione degli altri peer collegati, pertanto, è possibile usare il broadcast, sapendo che i pacchetti non verranno comunque inoltrati al di fuori della sottorete da eventuali router di frontiera.
- **Internet**: in Internet la situazione è più complessa, in quanto i pacchetti broadcast vengono automaticamente scartati dai router incontrati durante il path.
  E' necessario, pertanto, che la prima connessione alla rete venga fatta verso un nodo già noto che, su richiesta, possa fornire le informazioni al peer sulla topologia di rete (vedi anche la sezione “Sviluppi futuri”, più avanti).

Per realizzare il software si è scelto di limitarsi allo scenario LAN, senza prevedere un centro-stella che memorizzi l'intera topologia della rete.

Entità coinvolte ed informazioni scambiate

Nella comunicazione sono coinvolti, ovviamente, i vari peers connessi alla chat, ma vale la pena anche evidenziare alcune tra le informazioni scambiate attraverso la rete, per capire cosa vale la pena inserire nel protocollo.

- ***Peers:*** i vari utenti connessi alla rete. Ognuno di loro dispone di un nickname ed è riconosciuto univocamente dal suo indirizzo IP.
  - *NB*: mentre l'indirizzo IP è presente “naturalmente” su ogni pacchetto spedito da un dato peer è necessario inserire manualmente il nickname nel pacchetto, se si ha intenzione di renderlo noto agli altri peers.
- ***Messaggi:*** ogni peers spedisce dei messaggi di testo non formattato a tutti gli altri peers.
- ***Ack:*** da richiesta è necessario gestire eventuali packet loss e packet reordering... E' indispensabile, pertanto, istituire un sistema di acknowledgment, in modo che si possa determinare con certezza se un particolare messaggio è arrivato o meno a destinazione

Da una prima analisi emergono tre tipi di informazioni scambiate: è sufficiente per una prima ipotesi di protocollo

Ipotesi di protocollo

Il protocollo di comunicazione progettato si divide in tre step: handshake, messagging e disconnessione.

Esaminiamoli separatamente:

Handshake

Lo step di handshake avviene quando un peer si connette alla rete.
All'inizio non conosce i recapiti di nessun altro peer, quindi la prima cosa che è necessario fare è “presentarsi” e raccogliere informazioni sugli altri peer connessi.

Il procedimento si può quindi riassumere in due sotto-fasi: presentazione e ricezione delle risposte.

1. Presentazione: il peer invia in broadcast, così da essere “sentito” da tutti gli altri peer connessi, il suo nickname. In questo modo chiunque sia in ascolto saprà che dal suo indirizzo IP si è connesso un nuovo utente.
2. Risposta: ogni peer invia un acknowledgment al neo-peer, inserendovi a sua volta il proprio nickname, così da permettere la composizione di un'anagrafica dei peer connessi.

Al termine dell'handshake ogni peer connesso ha un'anagrafica completa di tutti i peer connessi attualmente alla rete.

Struttura del pacchetto: [Type, Nickname]

Dove *Type* indica se il pacchetto è di presentazione o di risposta, e *Nickname* è concluso col delimitatore di fine stringa '\0'

Questi pacchetti vengono inviati periodicamente in broadcast, in modo sia da raggiungere eventuali peer connessi di cui non è avvenuta la ricezione del pacchetto di presentazione sia per svolgere funzione di keepalive packet (vedi più avanti, nello step di disconnessione)

Messaging

La fase di messaging inizia quando l'utente desidera spedire del testo agli altri utenti connessi.

L'host, a questo punto, compone un pacchetto con dentro il testo del messaggio da spedire e lo invia, separatamente, ad ogni peer censito nella sua anagrafica

Struttura del pacchetto: [Type, Number, Messaggio]

Dove *Type* indica che il pacchetto è un messaggio,*Number* è il numero identificativo del messaggio (necessario per l'ack, vedi più avanti) e *Messaggio* è concluso col delimitatore di fine stringa '\0'

In risposta a questo pacchetto ogni peer restituirà un acknowledgement di ricezione del messaggio, indicandone il numero, in modo che possano essere diagnosticate eventuali perdite di traffico verso uno o più destinatari, che verranno risolte con eventuali rispedizioni dello stesso messaggio verso i destinatari che non hanno risposto con l'ack.

Struttura del pacchetto di Ack: [Type, Number]

Dove *Type* indica che il pacchetto è un ack relativo ad un messaggio, e *Number* consente all'host di identificare il numero di messaggio a cui si riferisce

Disconnessione

La fase di disconnessione può essere volontaria o meno.

Una disconnessione volontaria avviene quando un utente chiude il programma di chat: in questo caso viene semplicemente spedito un pacchetto che segnala la disconnessione del peer prima di chiudere il socket.

Struttura del pacchetto: [Type]

Dove *Type* indica che il pacchetto segnala una disconnessione.

Nota bene: la disconnessione volontaria non è stata implementata nel codice

Casi limite

Il protocollo descritto qui sopra può fallire in un paio di casi limite, che esaminiamo qui in integrazione a quanto già scritto

Arrivo di pacchetti terzi

Può succedere che, per caso o volontariamente, altri programmi spediscano pacchetti UDP sulla stessa porta dove il client di chat sia in ascolto.

Per evitare che questi pacchetti vengano considerati validi dal client conviene contrassegnare i pacchetti legali con una sorta di firma, in modo da poterli distinguere dal resto del traffico.

Senza sconfinare in ambito di sicurezza delle comunicazioni ci limiteremo ad inserire un preambolo fisso di alcuni bytes all'inizio del pacchetto, noto ai client.
In questo modo, ogni volta che un pacchetto viene ricevuto, è possibile vedere se i primi bytes corrispondono al preambolo; in caso contrario il pacchetto viene scartato.

Nuova struttura del pacchetto: [Preamble, Type, <altro?>]

Disconnessione involontaria

A differenza della disconnessione volontaria descritta prima quella involontaria avviene per cause di forza maggiore (ad es: power shortage dell'host o, più semplicemente, perdita di uno o più pacchetti di disconnessione volontaria lungo il canale).
In questo caso è compito degli altri peers rilevare il fatto che un loro collega è sparito dalla rete.

Questo è facilmente rilevabile con pochi accorgimenti:

1. Se un peer non genera più traffico sul canale per un certo lasso di tempo `T_Disc` si può dichiarare come disconnesso.
2. Questo traffico dovrebbe accadere naturalmente se viene pubblicato almeno un messaggio entro `T_Disc`, visto che gli acknowledgment spediti segnaleranno che il peer è ancora presente nella rete.
3. Conoscendo `T_Disc` ogni peer, però, ha interesse a spedire almeno un pacchetto sul canale (Keepalive packet) entro quel lasso di tempo per evitare la disconnessione, anche se non ha messaggi da spedire.
   Considerato che non si può fare affidamento sui soli ack per mantenere viva la connessione (gli ack possono venire persi, o non sono nemmeno mai generati nel caso non vengano spediti messaggi per un periodo di tempo superiore a `T_Disc`)
   In questo caso conviene che il peer spedisca il suo pacchetto di presentazione, a cui faranno seguito i dovuti acknowledgment da parte di tutti gli altri peer connessi.

Per risolvere questo border case, pertanto, non sono necessarie modifiche ai pacchetti esistenti

Mittente sconosciuto

Questo border case può capitare quando l'anagrafica dei peer di un singolo host (chiamiamolo **H**) si corrompa (ad es: un crash del programma) e non venga ricostruita correttamente (es: perdita di uno degli ack di risposta al pacchetto di presentazione di H).

In questo caso è possibile che uno dei peer (chiamiamolo “X”) abbia censito correttamente H, ma che H non conosca X.
Se viene spedito un messaggio da X, H lo riceverà, ma non saprà dire chi l'ha spedito, perché non conosce il nickname di X.
In questo caso conviene inviare un pacchetto di *Negative Acknowledgement (Nack)*, in modo da indicare che il messaggio è stato ricevuto correttamente, ma che non è possibile accettarlo.

X, a questo punto, capirà la situazione e potrà rispedire i suoi dati utente ad H, che sarà così in grado di censirlo correttamente.

Struttura del pacchetto [Preamble, Type]

Dove *Preamble* è il preambolo fisso di validazione del pacchetto, mentre *Type* indica che il pacchetto è un pacchetto di *Nack*.
Nota bene: non è necessario il numero del messaggio di riferimento: H attenderà che X lo rispedisca, visto che non ha inviato alcun ack in risposta al messaggio.

Analisi codice

Una volta consolidato il protocollo di comunicazione è possibile passare all'analisi del software che utilizzerà il protocollo stesso.

Per fornire una visione d'insieme delle varie classi costruite durante lo sviluppo viene prima di tutto fornito un diagramma delle classi (a pagina seguente), che illustra relazioni, gerarchie e messaggi scambiati tra le varie classi (metodi get/set nascosti per leggibilità).

Esaminiamo queste classi un po' più in dettaglio

Network Management

Le classi descritte in questa sezione hanno il solo scopo di agevolare le operazioni di gestione della comunicazione di rete.

Socket

La classe Socket nasce per nascondere, al suo interno, l'implementazione dei socket, caratteristica della piattaforma su cui sta girando il codice.

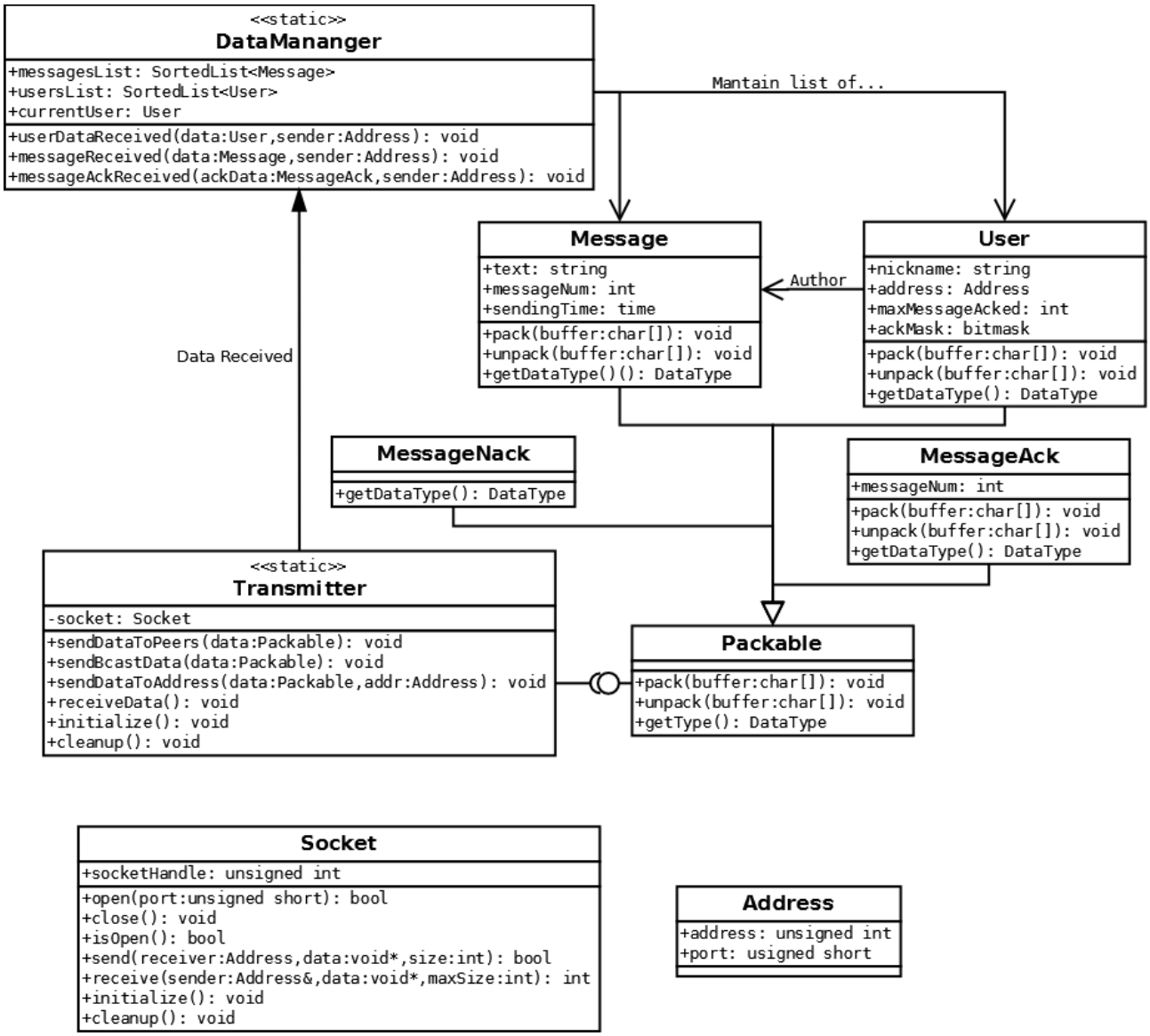
In questo modo è possibile effettuare agilmente un porting del codice verso altre piattaforme, andando ad intervenire sulla sola implementazione della classe e lasciandone intatta l'interfaccia.

Per minimizzare l'inclusion tree tutte le fuzioni che usano le librerie dei socket (comprese le funzioni di “conversione” ntoh* e hton*) sono state wrappate da questa classe e da nessun'altra.

In fase di progettazione è stato ipotizzato il supporto per windows, *nix e osX ma, per ragioni di

tempo e risorse, è stato sviluppato solo il supporto per windows.

Non vi sono particolari scelte implementative che coinvolgono questa classe, quindi si può subito procedere con le altre.



Address

La classe nasce per uno scopo simile a quello della classe Socket: opacizzare l'indirizzo IP al resto del codice, in modo da rendere il programma “agnostico” rispetto al metodo di indirizzamento utilizzato.

In fase di progettazione era stato ipotizzato il supporto per IPv4 e IPv6 ma in fase di realizzazione ci si è limitati a IPv4.

Anche in questo caso non vi sono particolari scelte implementative su cui vale la pena soffermarsi.

Data Models

Le classi in questa sezione hanno il solo scopo di raccogliere al loro interno le informazioni da memorizzare necessarie per il buon funzionamento del sistema.

Packable

Packable è una classe astratta, che nasce per l'esigenza di fornire un set di operazioni comuni necessarie per poter trasmettere/ricevere un dato oggetto lungo il canale.

In particolare queste tre operazioni sono:

1. **pure virtual getDataType**: il metodo virtuale puro getDataType prevede che la classe restituisca il tipo di dato che verrà serializzato col metodo pack (vedi punto successivo). Per evitare duplicazioni di valori e agevolare eventuali refactoring è stato dichiarato un *enum* nella classe Packable che elenca tutti i tipi di dati serializzabili.
2. ***virtual* pack**: l'operazione di packing dell'oggetto consiste nel serializzare gli attributi che si intende trasmettere dentro un *buffer,* fornito da parametro, che possa poi essere passato alla classe Socket per l'invio lungo il canale.
   La versione base di questo metodo inserisce già nel buffer i byte di preambolo (dichiarati come *static const*, sempre nella classe Packable) e il tipo di dato, appoggiandosi al metodo **getDataType**.
3. ***virtual* unpack**: il duale di **pack**, prevede che, a partire da un *buffer* passato da parametro, vengano inizializzati gli stessi attributi della classe, a partire dal contenuto del buffer stesso, facendo un vero e proprio revert delle operazioni eseguite con l'invocazione di **pack**.

Inoltre la classe mette a disposizione il metodo statico **validatePackage**, che convalida il contenuto di un pacchetto ricevuto, verificando prima la corrispondenza del preambolo e poi la coerenza del tipo di dato, ritornando il tipo di dato contenuto in caso di successo e -1 in caso di fallimento.

User : Packable

User, figlia di Packable, è la classe che mantiene le informazioni relative ad un singolo peer, come indirizzo, nickname e uno storico ridotto degli ack ricevuti dal peer in questione.

Questa classe viene spedita lungo il canale in due casi: nello step di “presentazione” e come risposta alla presentazione (vedere anche la sezione “Handshake” del protocollo).
Per distinguere quali dei due casi si sta gestendo (alla ricezione di un pacchetto di presentazione è previsto una risposta, mentre nell'altro caso no) viene sfruttato il byte riservato al tipo di dato, settato con un valore differente nel caso il pacchetto rappresenti una risposta.

Di tutta la classe vale la pena soffermarsi soprattutto sull'implementazione del mini-storico degli ack: per evitare di gestire una struttura dati complessa per memorizzare semplicemente un valore booleano (ack ricevuto o meno) per i soli messaggi più recenti si è scelto di usare una coppia di variabili organizzate in questo modo:

1. **maxMessageAcked**: questa variabile contiene il massimo numero di messaggio per il quale è stato ricevuto un ack.
2. **ackMask**: questa variabile è una maschera di bit, che contiene lo stato di ricezione degli 8*sizeof(ackMask) ack relativi ai messaggi immediatamente precedenti a **maxMessageAcked**.

In questo modo il test di reinvio di un particolare messaggio N a quest'utente è semplice:

1. se *N == maxMessageAcked* il messaggio è stato ricevuto correttamente, non serve reinvio
2. altrimenti se *N < maxMessageAcked – (8*sizeof(ackMask))* il messaggio è troppo vecchio, e non vale la pena rispedirlo (accettiamo il fallimento dell'invio)
3. altrimenti se *N > maxMessageAcked* l'ack di risposta non è mai stato ricevuto: è necessario rispedire il messaggio.
4. In caso contrario si effettua un test sul singolo bit della maschera: *ackMask & (1 << maxMessageAcked) != 0*. Se il test fallisce è necessario rispedire il pacchetto.

Questo consente di ottimizzare la gestione dello storico degli ack, senza dover per forza appoggiarsi ad una struttura esterna e negoziare memoria ad ogni ack da salvare, pur garantendo una “finestra di verificabilità” dimensionata secondo le esigenze (nell'implementazione, ad esempio, la maschera è un *unsigned short*, ma nulla vieta che possa essere più grande).

Message : Packable

La classe Message rappresenta un singolo messaggio transitato nella finestra di chat.

Per meglio gestire l'ordinamento dei messaggi nella finestra di chat nel singolo messaggio viene salvata l'ora di invio da parte dell'autore.
Questo provoca alcuni disallineamenti, perché gli orologi dei differenti peer non è detto siano sincronizzati, ma senza un super-peer in grado di gestire una simile sincronizzazione (vedi anche la sezione Sviluppi Futuri) è un compromesso che si può accettare.

E' stato definito, inoltre, un formato di tempo relativo da usare all'interno della chat, inteso come il numero di secondi trascorsi dallo 01/01/2014. Questo ha il vantaggio di consentire il salvataggio dell'informazione data/ora nel formato unsigned long, molto facile da trasmettere usando i socket.

Per comodità è stato definita una relazione di ordinamento totale sui messaggi, in modo da poterli memorizzare in una delle strutture dati ordinate presenti nella libreria standard, semplificando le operazioni di gestione.

MessageAck : Packable

La classe MessageAck rappresenta un singolo Ack, relativo ad un dato messaggio.

Al suo interno contiene il solo numero del messaggio a cui si riferisce, ed il suo lifetime è molto limitato: viene infatti costruita e inizializzata al momento di spedirla via socket e distrutta subito dopo.

MessageNack : Packable

La classe MessageNack rappresenta un singolo Nack, relativo ad un dato messaggio.

Ancora più povera della sua “sorella”, MessageAck, questa classe non contiene nessun attributo, visto che può qualificarsi come Nack modificando il valore del byte Type nei pacchetti a lei relativi (cosa che succede ridefinendo semplicemente il metodo getDataType).

Chat System

Le classi in questa sezione gestiscono il buon funzionamento della chat, organizzando le informazioni e gestendo le comunicazioni ad alto livello

Data Manager

Questa classe ha il compito di mantenere le liste di messaggi e utenti, aggiornandone i contenuti quando necessario.
Gran parte dei metodi che espone, infatti, vengono chiamati in risposta a determinati eventi (es: ricezione di un messaggio, di un ack, di una presentazione, ecc...), mentre gli altri riguardano comunque la gestione di queste informazioni: writeNewMessage(), ad esempio, che consente di creare un nuovo messaggio in base all'input, o cleanup(), che si preoccupa di ripulire tutta la memoria allocata, o, ancora, i vari metodi di stampa delle informazioni memorizzate.

Infine, è sempre suo compito gestire le informazioni relative all'utente corrente (quello che interagisce con l'host, per intendersi).
Per questo motivo tiene conto anche del tempo trascorso dall'ultimo broadcast delle informazioni dell'utente corrente, in modo da non superare i tempi di disconnessione automatica, anche se poi l'operazione di broadcast in sé viene delegata alla classe Transmitter, qui sotto.

Dalla progettazione non è emersa esplicitamente l'esigenza di avere un singleton, pertanto la classe è composta da soli metodi e attributi statici.

Transmitter

Questa classe ha il compito di gestire la comunicazione delle informazioni con gli altri peers connessi alla rete.

Espone, infatti, i metodi necessari per trasmettere (a tutti i peers, in broadcast o singolarmente) un singolo oggetto di tipo Packable lungo il canale, e gestisce internamente sia la ricezione dei dati che il reinvio dei messaggi per cui non è stato ricevuto un ack da parte di almeno un peer regolarmente censito.

Dalla progettazione non è emersa esplicitamente l'esigenza di avere un singleton, pertanto anche questa classe è composta da soli metodi e attributi statici.

Limiti noti

Purtroppo, dato il tempo a disposizione, non è stato possibile realizzare l'intero elenco delle feature che erano state pensate per quest'applicazione, lasciando alcuni limiti intrinseci al software in sé.

Supporto per altre piattaforme

Vista l'architettura adottata un porting verso *nix o osX del software sarà poco oneroso, perché coinvolge porzioni di codice molto ristrette: la sola classe Socket.

Trasmissione parziale messaggi

Al momento il sistema di chat supporta l'invio di messaggi di massimo 140 caratteri (delimitatore finale escluso).

Questo limite può essere rilassato con l'applicazione di diversi passi:

1. Interrogazione della NIC per conoscere la massima dimensione assumibile dal pacchetto da trasmettere
2. Aumento granularità del sistema di messaggi, in modo che supporti l'invio di *parti* di messaggio, e non messaggi interi.
3. Aumento granularità del sistema di ack: non è più sufficiente una sola maschera di bit per gestire gli N messaggi precedenti, ma serve una maschera per ogni messaggio spedito, in modo da poter memorizzare gli ack delle singole parti.

Funzionamento Internet

Visto l'utilizzo dell'indirizzo di broadcast per l'handshake verso i nuovi peers questo software può funzionare solo su una rete LAN.

Per estenderne il funzionamento all'intera internet è necessario che, nella mesh, esista un centro-stella (o super-peer), che si assuma delle funzioni centrali di coordinazione, tra cui

1. Essere sempre raggiungibile direttamente, magari mediante utilizzo di DNS.
2. Mantenimento di una lista di peer connessi (IP e nickname) in modo da eliminare la necessità del broadcast
    1. Lista a disposizione degli altri peer su richiesta.
    2. Aggiornamenti alla lista vanno inoltrati a tutti i peer connessi (es: quando si connette un nuovo utente il suo IP/nickname vanno inoltrati a tutti i peer già connessi), in modo da mantenere sincronizzate le copie
    3. Il meccanismo di riconoscimento della disconnessione forzata cambia: il super-peer invia un pacchetto di controllo ad ogni peer connesso, pretendendo un Ack che fungerà da keepalive packet.
3. Sincronizzazione degli orologi dei peer che si connettono. Vista la scarsa precisione richiesta

da questo dato uno scostamento di un secondo è accettabile. E' possibile unire questa funzione al meccanismo di riconoscimento della disconnessione forzata, per assicurare una sincronia prolungata tra super-peer e altri peer.

Inoltre sarebbe apprezzabile estendere il supporto a IPv6, modificando l'implementazione della classe Address, in modo da poter gestire senza problemi connessioni da diverse aree geografiche.