Università di Cagliari
Facoltà di Scienze MM.FF.NN
Corso di Laurea in Informatica

RELAZIONE PROGETTO
LAN MESSENGER

Corso T.A.R.I.
A.A. 2010 -2011

VALERIO PILO NM 38166

1 Introduzione

Il progetto denominato “LAN Messenger” è stato realizzato per essere presentato all'esame del corso di Tecnologia e Applicazioni della Rete Internet. Il percorso scelto è quello della applicazione basata su socket.
LAN Messenger è una applicazione C++ basata sulle librerie standard, ncurses e pthread, che implementa un servizio di messaggistica istantanea con la possibilità di trasferire file.

Il servizio offerto è simile, ma molto semplificato, a quello di un canale IRC: gli utenti possono connettersi al canale e parlare a vicenda, e cambiare il proprio nome. In aggiunta, gli utenti possono scegliere di inviare un file alla volta a tutti gli altri partecipanti.

Il progetto compila un server e un client, basati sullo stesso codice di rete e protocollo; questi verranno trattati in dettaglio nei seguenti capitoli.

L'applicazione è stata compilata e testata sulle seguenti macchine:

- OpenSUSE Linux 11.3 64-bit
- Ubuntu Linux 10.04 32-bit

2 Il protocollo

Entrambe le applicazioni, il server e il client, sono basati sullo stesso codice di rete, e possono essere residenti su macchine diverse in una stessa LAN.

Il protocollo è basato su messaggi in formato binario instradati utilizzando il protocollo TCP. Confidando nel controllo e risoluzione d'errore fornito dal protocollo si è alleggeriti i requisiti dalla necessità di verificare i dati in transito sulla rete e controllare i timeout. Ulteriore motivazione per questa scelta è il fatto che nell'ambiente a cui il progetto è destinato, ovvero una rete LAN Ethernet, errori e problemi di latenza sono spesso trascurabili.

Il protocollo è basato su messaggi binari. Ogni messaggio è formato da una intestazione (header), seguita da un'opzionale carico dati (payload). L'header è formato da un comando, il quale definisce il tipo di messaggio ed è una stringa di massimo 3 lettere (4 con il carattere NULL di terminazione), e da 4 byte che indicano la dimensione dei dati del payload associato. Per alcuni messaggi, ad esempio quelli di connessione e disconnessione, questi dati sono sufficienti e la dimensione del payload è 0; altri invece necessitano di dati aggiuntivi, che vengono messi in coda all'header.

Questi sono i comandi attualmente riconosciuti dai componenti dell'applicazione:

Enumerazione	Comando	Descrizione
Message::MSG_HELLO	"HI"	Accesso al server
Message::MSG_NICKNAME	"NAM"	Cambio di nome del client
Message::MSG_BYE	"BYE"	Uscita dal server
Message::MSG_STATUS	"ST"	Informazione di stato
Message::MSG_CHAT	"MSG"	Messaggio di chat
Message::MSG_FILE_REQUEST	"REQ"	Richiesta di trasferimento file
Message::MSG_FILE_DATA	"DTA"	Dati di un trasferimento file
Message::MSG_INVALID	N/A	Tipo di messaggio non valido

La stringa è stata scelta invece di (ad esempio) un codice numerico per migliorare la visibilità dell'inizio di ogni messaggio all'interno dello stream TCP.

L'header è così rappresentato in codice:

```
struct MessageHeader
{
  char command[ COMMAND_SIZE ];
  int size;
};
```

[common/protocol.h]

Un esempio di payload può essere questo, tratto dalla classe ChatMessage:

```
struct Payload
{
  char sender[ MAX_NICKNAME_SIZE ];
  int messageSize;
  char* message;
};
```

[common/messages/chatmessage.h]

2.1 Il codice di rete

Al livello più basso, le applicazioni server e client condividono lo stesso codice, che si occupa di ricevere i dati grezzi in ingresso dalla rete e inserirli in un buffer temporaneo. Questo per ovviare al problema che lo strato rete del sistema operativo non consegna i messaggi separati tra loro, così come sono stati inviati, ma li rende disponibili in modo imprevedibile.
Una volta che il buffer contiene abbastanza dati per poter contenere almeno un messaggio, viene avviata la procedura di analisi. Il buffer viene analizzato per identificare un comando, e a seconda di quale viene identificato, viene creato un messaggio; un certo blocco di dati successivi è altresì analizzato per estrarne il payload del messaggio (se presente). Nel caso il buffer non contenga ancora dati a sufficienza anche per il payload, l'analisi viene rinviata alla ricezione del successivo blocco di dati.
Una volta individuato un messaggio completo, esso è inserito in una coda in ingresso, dal quale l'applicazione sovrastante è libera di estrarlo in qualunque momento.
Il codice di rete condiviso si occupa anche di serializzare e inviare i messaggi, rimuovendo i messaggi da una coda che viene riempita dall'applicazione.
Quando non sono disponibili dati da inviare o da ricevere, il codice di rete rimane in attesa in un proprio thread.

2.2 Meccaniche

Vengono di seguito esplicati i vari scambi di messaggi tra server e client.

2.2.1 Connessione

Dopo aver effettuato con successo una connessione al server, il client invia un messaggio di saluto HI. Il server risponde inviando al client un nickname univoco (messaggio NAM). Dopo di che, il client è connesso e può inviare messaggi di testo (messaggi MSG) e trasferimenti file (REQ/DTA).

2.2.2 Messaggistica testuale

Il client, una volta connesso, può inviare messaggi MSG: ogni messaggio contiene un nome di mittente (vuoto) e il messaggio inviato. Il server riconosce il client mittente e riempie il campo sender, per poi inoltrare il messaggio agli altri client. Non sono state previste conferme di invio ne' sequenziazione dei messaggi, per semplicità implementativa e per oggettiva poca necessità (vedere 5.2, Messaggi).

2.2.3 Cambi di nickname

Tutti gli utenti possono selezionare il proprio nome autonomamente, a patto che nessun altro client possegga un nome identico. Questa verifica è effettuata dal server. Il client invia un messaggio NAM con il nickname selezionato dall'utente; e il server verifica l'univocità del nome. In caso il nome sia un duplicato, invia un messaggio ST 202 (*Status_NickNameAlreadyRegistered*) al client per notificare il problema. In caso sia univoco, invia un altro messaggio NAM con il nuovo nickname al client. Il messaggio sarebbe potuto essere anche inviato agli altri client, ma si è preferito evitare questo comportamento per brevità (vedere 9, Messaggi di stato).

2.2.4 Richieste di trasferimento di file

Sul sistema può essere in transito al massimo un file. Ogni client ha la facoltà di inviare file, e per farlo invia al server un messaggio REQ, contenente il nome del file che l'utente vuole inviare agli altri partecipanti.
Il server controlla che non siano in corso altri trasferimenti (il client potrebbe infatti essersi connesso dopo l'avvio di un'altro trasferimento di file, e non saperne quindi nulla), e invia una richiesta REQ identica all'originale agli altri client, perchè gli utenti confermino o annullino la richiesta.

2.2.5 Conferme di trasferimento di file

Ogni altro utente deve quindi obbligatoriamente rispondere alla richiesta postagli dal mittente del file: l'interfaccia utente chiede se si vuole accettare il file o meno, e in caso affermativo, chiede dove si voglia salvare il file in arrivo.
Ottenuta una risposta dall'utente, il client risponde al server con un messaggio ST 203 o ST 204, a seconda che la richiesta sia stata rispettivamente accettata o rifiutata.
Non appena tutti i client hanno risposto alla richiesta di trasferimento, il server verifica se almeno una delle risposte è affermativa: in tal caso invia al client mittente della richiesta di trasferimento file un messaggio ST 203.
Il client riconosce questo come il segnale di inizio, e avvia l'invio dei dati.

2.2.6 Trasmissione dati in un trasferimento di file

Il file da inviare viene scomposto in blocchi di dimensione appropriata: la dimensione massima del payload di un segmento TCP viene infatti utilizzata per limitare le dimensioni totali dei pacchetti ed evitarne quindi la frammentazione.
Ogni messaggio (di tipo DTA) contiene un'indicazione della posizione nel file a cui iniziano i dati contenuti nel payload del messaggio. In questo modo, i client che abbiano accettato il file possono sapere dove andare a inserire i dati ricevuti. Nel caso in cui si dovesse andare a utilizzare un protocollo diverso per l'invio dei dati, ciò consentirebbe che i dati arrivati siano comunque riordinabili dal client in modo trasparente.
In aggiunta, i messaggi DTA contengono una flag che viene impostata a TRUE quando i dati sono terminati: ciò per semplificare il protocollo ed evitare di mandare a tutti un messaggio di stato aggiuntivo.
Non appena l'ultimo blocco di dati è stato inviato, il trasferimento è dichiarato come concluso.
Il server riceve dal mittente i pacchetti dati e li inoltra a tutti e soli i client che abbiano accettato la richiesta, per evitare traffico dati inutile.

3 Struttura del progetto

3.1 Il server

Il server è una applicazione console, che non necessita di interfaccia in quanto il suo ruolo è meramente di fungere da tramite e arbitro ai messaggi dei vari client connessi. Pertanto rimane in attesa di messaggi senza richiedere input dall'utente né emettere output, eccezion fatta per gli eventuali messaggi di debug (vedere 4, Debug).

Il programma all'avvio istanzia un thread, il quale si mette in attesa di connessioni TCP ad un socket IPv4, sulla porta 12345 (al fine di non richiedere permessi di amministrazione per il possesso di una "well-known port" sotto la 1024).

Ogni connessione da parte di un client causa la istanziazione di un nuovo ulteriore thread che viene incaricato di gestire il flusso dati da e per quel client. Su quel thread infatti una classe dedicata, **SessionClient**, viene messa in attesa di dati in ingresso o in uscita.

Ad esempio, con 1 client connesso, sono attivi 3 thread: 1 thread principale, 1 che funge da server, 1 per il flusso dati da/per il client. Ciascuno è descritto nei paragrafi successivi.

3.1.1 Il thread principale

Il thread principale è sostanzialmente dedicato a inizializzare il server all'avvio e distruggerlo all'uscita.

Dopo l'inizializzazione, resta in attesa di un segnale di uscita: i segnali Unix SIGQUIT, SIGINT, SIGHUP e SIGTERM causano tutti l'uscita. Altri (come SIGTSTP) sono ignorati.

3.1.2 Il thread di connessione

Il server istanzia un thread di questo tipo per ogni client connesso. Questo thread, associato direttamente ad una istanza della classe SessionClient, resta in attesa di dati provenienti dal client oppure di dati da inviare allo stesso, e si occupa di smistare i tipi di messaggio in arrivo verso il thread server.

3.1.3 Il thread server

Questo thread, implementato dalla classe **Server** (*server/server.h*) è il cuore del server. Rimane costantemente in attesa di connessioni da parte di nuovi client, e non appena stabilita, istanzia un thread di connessione per la sua gestione.

Non appena almeno un client è connesso, il suo ruolo funge anche da sistema di routing dei messaggi e di verifica del rispetto del protocollo da parte dei client. I client devono infatti rispettare una certa sequenza di comandi, secondo una piccola macchina a stati che, in caso di riscontro di violazione, causa la disconnessione del client. Ad esempio un client non può mandare un messaggio HI dopo l'identificazione col comando NAM, dato che il "saluto" è già avvenuto.

Tutti i messaggi, sia di testo che di dati di file, vengono solitamente inoltrati da un client a tutti gli altri: alla ricezione del messaggio, il server lo re-invia a tutti gli altri client connessi, senza quindi rimandarlo indietro al client mittente.

Messaggi particolari (**StatusMessage**, *common/messages/statusmessage.h*) sono utilizzati dal server per segnalare problemi o dare notifiche: ad esempio, ogni client deve avere un nome univoco; se un utente tentasse di assumere lo stesso nome di un altro utente connesso, il server invierebbe al client uno StatusMessage con codice 202, *Status_NickNameAlreadyRegistered* (enumerazione **Errors::StatusCode**, *common/errors.h*).

3.2 Il client

Ogni client è una applicazione console dotata di una semplicissima interfaccia utente, scritta mediante le librerie ncurses.

All'avvio il client tenta automaticamente di connettersi ad un server sulla medesima macchina, ovvero all'indirizzo 127.0.0.1:12345 . È ovviamente possibile far connettere il client ad un'altra macchina remota semplicemente specificando l'IP della stessa come parametro all'avvio del programma:

```
lanmessenger 192.168.1.56
```

Una volta avviato, l'interfaccia mostra in alto il nome del programma seguito da un breve messaggio di stato, ad esempio *"Connecting.."*. Non appena la connessione viene stabilita, il messaggio di stato diviene il nome dell'utente: *"In chat as User 1"*.

In basso, una riga è dedicata all'inserimento dei messaggi di testo, ma anche all'inserimento dei dati richiesti dall'applicazione, come la richiesta di inserimento di un nuovo nickname o di accettazione o rifiuto di un file.

Al centro, l'area in cui vengono mostrati i messaggi degli utenti, assieme ai messaggi di stato provenienti dal server. La visualizzazione è ispirata ai canali IRC, eccone un esempio:

```
17.36 <SERVER> Your nickname is now "User 2"
17.36 <SERVER> Your nickname is now "Valerio"
17.36 <Valerio> Ciao
17.36 <SERVER> There are no other participants to the chat!
```

Premendo il tasto F2 il client richiede l'inserimento di un nuovo nickname, che viene alla conferma inviato al server, validato, e rimandato indietro. Premendo F4, il client chiede di selezionare un file da inviare (per semplicità, è necessario inserire il percorso assoluto del file, o relativo alla directory corrente da cui si è lanciato il client).

Una volta confermato il nome del file, viene spedita una richiesta agli altri partecipanti (viene interrotta temporaneamente la loro capacità di inviare messaggi). Quando tutti hanno risposto, se almeno uno ha accettato il file viene inviato.

Per uscire dal client, è sufficiente premere il tasto ESC.

3.2.1 Struttura delle classi nel client

- Il file *main.cpp* analizza gli input da riga di comando (l'indirizzo del server alternativo) e inizializza sia il codice di rete (classe **SessionServer**) che l'interfaccia utente (classe **Client**).
- La classe **SessionServer** (*client/sessionserver.h*) funge da interfaccia tra il codice basilare di rete, la classe **SessionBase** (*common/sessionbase.h*) che implementa lo strato rete dell'applicazione, e il codice dell'applicazione, la classe **Client**. Qui sono concentrate le funzionalità del protocollo, ovvero sono inviati e ricevuti i messaggi e sono gestite le mecccaniche del protocollo, come ad esempio l'invio dei dati di un trasferimento file.

  Tutti i messaggi in ingresso sono analizzati da questa classe, che aggiorna il proprio stato e si occupa di inviare notifiche all'interfaccia utente.

  Ogni istanza di **SessionServer** tiene inoltre traccia di diversi dati richiesti dal protocollo: il nickname del client, il nome del file che si sta trasferendo, la posizione nel file, etc.

- La classe **Client** (*client/client.h*), d'altro canto, si occupa esclusivamente dell'interfaccia utente: è effettivamente l'unica che includa l'header <*ncurses.h*> e che si occupa di prendere l'input dell'utente e mostrare i messaggi degli altri utenti connessi.

  La funzione *run()* contiene il ciclo principale di analisi dei caratteri immessi dall'utente; cambio nickname, invio file, uscita e soprattutto, invio di messaggi di testo sono le funzionalità offerte.

  La ricezione dei messaggi è invece gestita in maniera asincrona: l'istanza di **SessionServer** vive in un thread separato da quello principale con l'interfaccia utente, e ciò permette al programma di gestire contemporaneamente le due attività.

4 Debug

Al momento della compilazione, è possibile impostare alcune variabili che determinano la quantità di messaggi di debugging mostrati da client e server.
È possibile infatti modificare il file *Makefile* con queste variabili:
per abilitare i messaggi di debug nel server, commentare la riga:

```
SERVER_DEFINES += -DNODEBUG
```

Rispettivamente per abilitarli nel client, commentare:

```
CLIENT_DEFINES += -DNODEBUG
```

Il server stampa i messaggi di log direttamente su standard output, invece i client su un file (dato che utilizzano lo standard output per l'interfaccia utente). I file sono chiamati ad esempio "*lanmessenger-client.11898.log*".
È anche possibile avere una stampa dei byte grezzi dei vari messaggi inviati e ricevuti.
Per il server, decommentare la riga:

```
# SERVER_DEFINES = -DNETWORK_DEBUG
```

Per il client, decommentare invece la riga:

```
# CLIENT_DEFINES += -DNETWORK_DEBUG
```

5 Limitazioni e espansioni future

A causa del limitato tempo a disposizione, il programma difetta in alcune utili funzionalità.

5.1 Messaggi di stato

Sarebbe utile avere un messaggio di stato che segnala a tutti i client il cambio di nome di un certo client.

Inoltre sarebbe una buona cosa avere un messaggio di stato che comunichi a tutti gli utenti connessi che un nuovo utente si è collegato alla chat, o che è uscito; magari anche con indicazione del numero totale di partecipanti alla chat.

5.2 Messaggi

I messaggi non dispongono attualmente di un numero di sequenza. Potrebbe essere utile sequenziarli per prevedere un uso con protocolli di rete che non garantiscano l'ordinamento dei messaggi all'arrivo (ad es., UDP).

Potrebbe altresì essere interessante implementare un sistema di acknowledgement dei messaggi ricevuti, ad esempio nel caso messaggi a client disconnessi andassero persi; tuttavia ciò andrebbe lievemente a scapito dell'utilizzo di banda.

5.3 Invio file

Sarebbe appropriato avere la dimensione del file offerto quando l'utente deve accettare o rifiutare una richiesta di trasferimento file.

Al termine dell'invio di un file, anche i client che hanno rifiutato il trasferimento file dovrebbero essere notificati del termine del trasferimento, di modo che possano a colpo sicuro inviare un file

(attualmente se provassero, e un utente stesse già inviando un file, il server gli risponderebbe che c'è già un trasferimento in corso).

5.4 Aiuto

L'interfaccia utente potrebbe mostrare un messaggio che spieghi i comandi da tastiera.