Progetto d'esame del corso di "*Reti di calcolatori*"

Realizzazione di un servizio di messaggistica istantanea su un architettura di rete client-server

Università degli Studi di Urbino "Carlo Bo"

Facoltà di Scienze e Tecnologie

Corso di Laurea in Informatica Applicata

Anno Accademico 2008/2009

LUCHETTI GIOELE – Matricola 227587
DROMEDARI MATTEO – Matricola 227637
MONACCHI ANDREA - Matricola 227377


DLM

Indice:

1. Introduzione

DLM nasce dall'idea di tre studenti del corso di laurea in informatica dell'università di urbino, per dare sfogo alla necessità di un client di messaggistica che funzioni anche senza il contributo della rete internet, un sistema di scambio per reti locali che sia quindi veloce e soprattutto portabile. Nulla vieta in qualunque caso di poter utilizzare tale programma anche al di fuori della propria LAN essendo realizzato a un livello di astrazione tale da poterne garantire il funzionamento. La portabilità è garantita grazie al linguaggio di programmazione Java che permette la creazione di applicazioni robuste e accattivanti in modo veloce e pulito.

In figura: il logo DLM Messenger

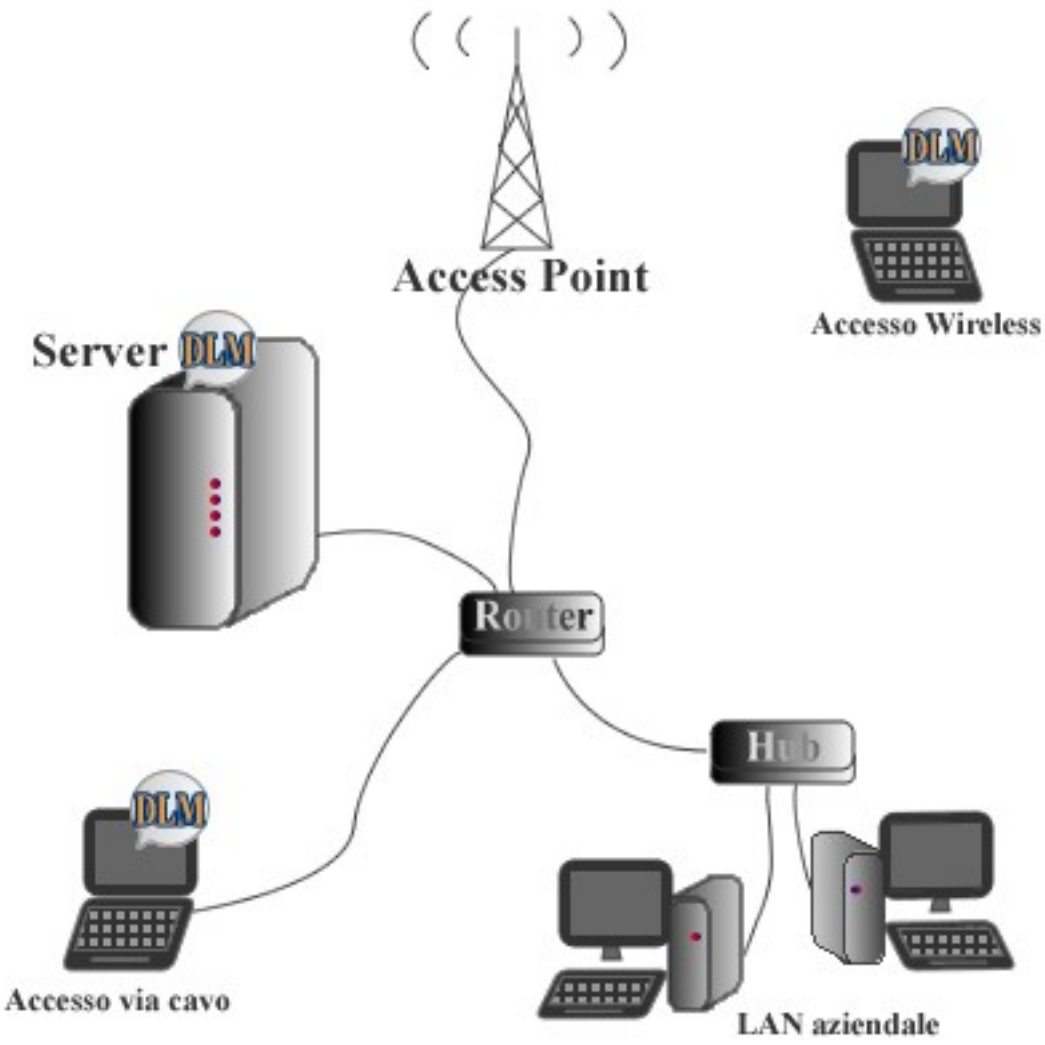
Il target potenziale di un software come DLM può quindi spaziare dalla semplice LAN domestica alla LAN aziendale dove per questioni di sicurezza si attuano restrizioni sull'accesso a internet ma sono comunque attive le comunicazioni interne.

Possiamo riassumere le fasi di realizzazione del sistema in 4 punti:

- Definizione dell'architettura di rete e del protocollo
- Definizione delle azioni compiute dal Client e dal Server
- Implementazione delle funzionalità definite
- Implementazione dell'interfaccia GUI del Client

2. La struttura client-server e il protocollo

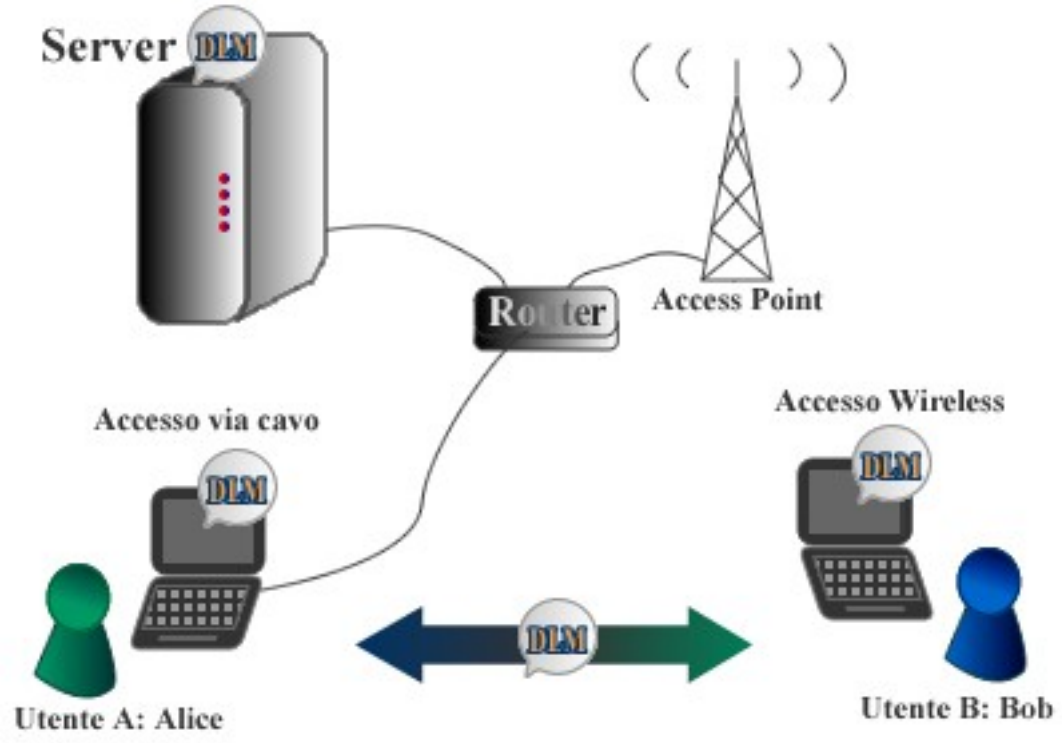
Nella struttura client-server, si distinguono clienti e servitori. Avremo quindi in un tipico esempio una rete composta da un servitore e una serie di clienti richiedenti di servizi, il server è il vero tramite senza cui questo paradigma non funzionerebbe, supervisiona le comunicazioni tra clienti che quindi avvengono in modo indiretto. Questa modalità di fruizione e condivisione dei servizi è quella che utilizzeremo in quanto avremo necessità di un computer sempre in ascolto sulla rete che si occupi di instradare le richieste e i dati provenienti dai vari computer, andando a definire un vero e proprio protocollo di comunicazione tra queste entità.



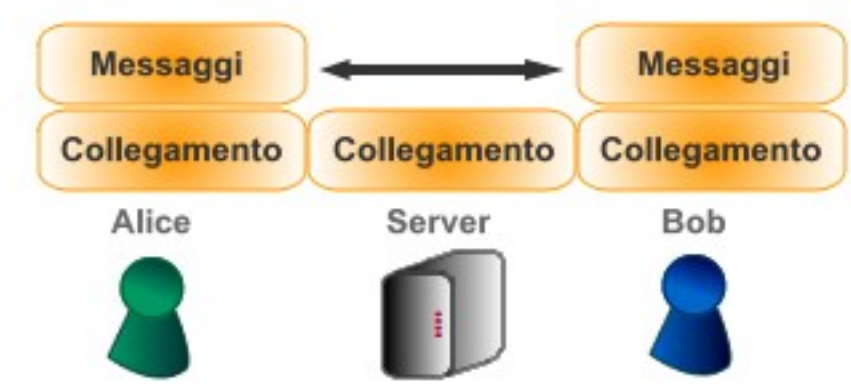
In figura: un esempio di rete locale multiaccesso (cavo e wireless), al centro il server (cuore delle comunicazioni)

2.1 Il protocollo di comunicazione

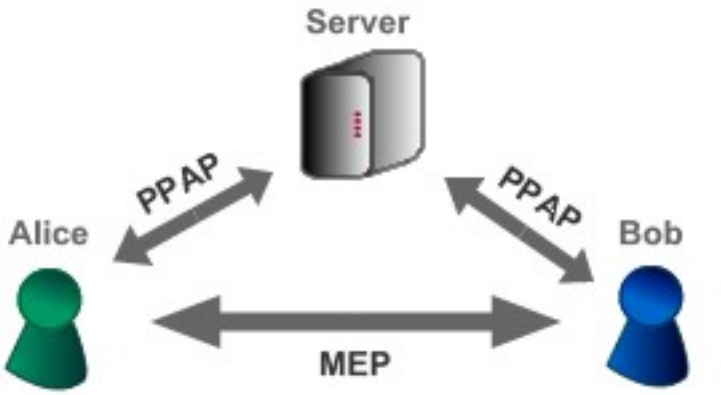
Supponiamo di avere due client che vogliono usufruire del nostro servizio di messaggistica istantanea. Il primo utente (Alice) è connesso su una infrastruttura fisica di tipo guidato mentre nel caso di Bob è sfruttata una connessione di tipo radio. Questo è possibile perchè le comunicazioni sono effettuate a livello applicazione e quindi possiamo astrarre tutti i controlli hardware e software dei livelli sottostanti, utilizzando una connessione sicura e indipendente dalla topologia di rete. Nella nostra rete "DLM" avremo sempre connessa una entità di collocamento e controllo, gestore e manutentore dei dati. Nella connessione virtuale che si instaura tra Alice e Bob avremo quindi un intermediario che garantirà sicurezza nel trasporto delle conversazioni, nascondendo le caratteristiche fisiche della rete. Nel nostro protocollo dovremo quindi gestire connessioni punto-punto client-server e il trasferimento dati host-to-host generando quindi due diversi livelli.



In questo caso Alice messaggia con Bob, non vede il suo intermediario ed è come se tra i due utenti si crei una connessione logica; d'altra parte Alice deve pur connettersi alla rete e a un nodo centrale che sia raggiungibile da tutti gli altri utenti. Riportiamo lo stack dei protocolli che definiremo in seguito.



DLM sarà quindi composto da due livelli gestiti separatamente, con due diversi protocolli, il PPAP per lo scambio di informazioni tra livelli di collegamento adiacenti e il MEP per lo scambio messaggi tra client.



2.2 Il livello collegamento

Questo livello si occupa di gestire la connessione tra il client e il server, i parametri legati all'accesso e all'autenticazione e alle varie modalità di scambio informazioni. Definiremo perciò le caratteristiche del protocollo PPAP (Point-to-Point Access Protocol), il tutto senza entrare nel merito delle primitive effettive di connessione che sono entità di livello trasporto e non ci interessa al momento trattare dato che lavoriamo a livello applicazione.

PPAP

Client-to-Server	***Server-to-Client***	
Autenticazione	Ack Autenticazione	
Richiesta lista contatti	Invio lista contatti	
Invio Messaggio	Inoltro Messaggio	
Disconnessione		
	Nuovo utente connesso	Broadcasting
	Utente disconnesso	

AUTENTICAZIONE
"Username + MacAddress + propria chiave pubblica" →

← AUTENTICAZIONE NEGATIVA -> chiusura Socket
← AUTENTICAZIONE POSITIVA -> invio Ack ("**Y**")

RICHIESTA LISTA
"**U**" →

← INVIO LISTA (lista degli username e chiavi pubbliche)
← **U** + ListaUtenti

INVIO MESSAGGIO
M + destinatario + MEP-DU →

→ INOLTRO MESSAGGIO
M + mittente + MEP-DU

DISCONNESSIONE
chiusura della Socket e del thread in lettura associato

Chiusura della connessione e aggiornamento liste utenti

Messaggi in broadcasting di aggiornamento dello stato

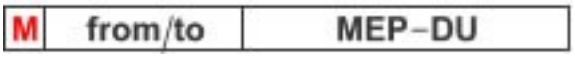
← NUOVO UTENTE CONNESSO (Username e chiave)
← **L** + Utente

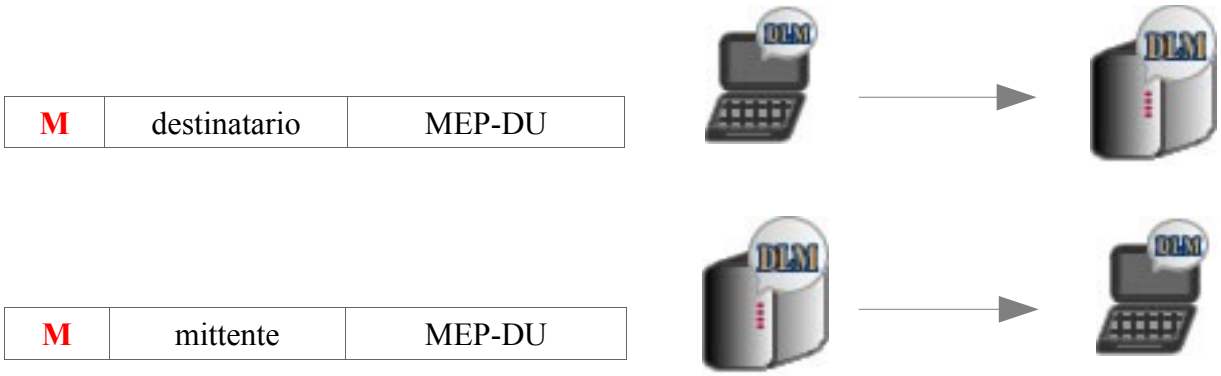
← UTENTE DISCONNESSO
← **D** + Username

Nota: La porta scelta per il funzionamento del protocollo è la 1988.

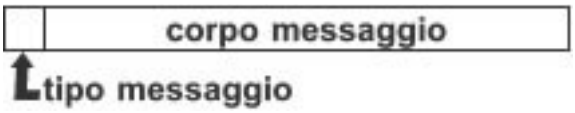
2.3 Il livello messaggi

Nello scambio messaggi a livello collegamento veniva utilizzata questa unità dati:

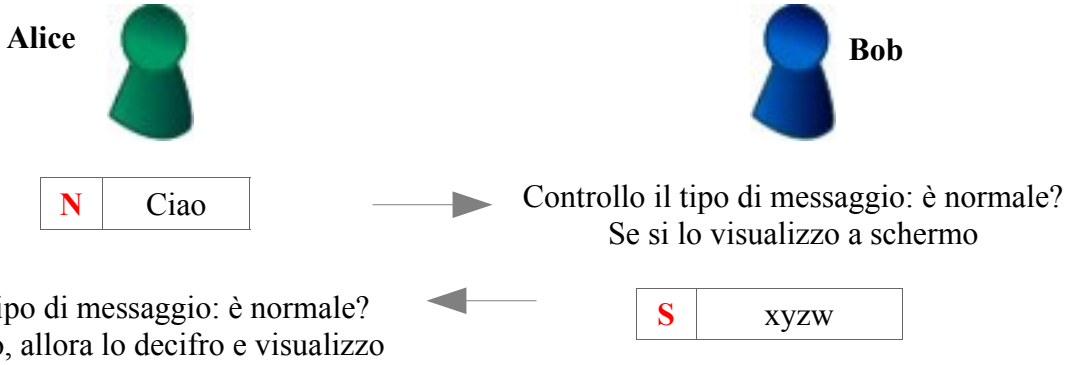




Ma cosa contiene questa MEP-DU (MEP Data Unit)?



Tipo messaggio		Corpo messaggio
N	Normal	Carico proveniente dalla finestra del client mittente, può essere in chiaro o criptato
S	Secret	



3. Il server

Il fulcro del nostro sistema, trattatore e smistatore di richieste è il Server. Vediamo di provare a capirne il funzionamento.

3.1 Le funzionalità

Il server deve essere in ascolto di eventuali client che richiedono di accedere, deve poter garantire allo stesso tempo alta reattività nel servire altri utenti connessi che possono in qualunque caso richiedere servizi o voler dialogare con qualcun'altro. Esso funziona a livello collegamento, è quindi un terminale che gestisce i parametri di connessione e quelli di routering dei messaggi, anche se non entra nel merito del loro carico, delegato a chi può usufruire del protocollo MEP quale è il client di destinazione.

3.1.2 La lista utenti connessi

Al momento della autenticazione, un client invierà sulla socket la sua username che lo identificherà univocamente, seguito dal suo MAC address e dalla sua chiave pubblica (che potremo utilizzare per il nostro algoritmo di sicurezza). Il server, ricevuti tali dati, controllerà che l'username non sia già presente nella lista degli utenti connessi e se è possibile lo inserirà.

Autenticazione: Il client invia Username + "|" + MACAddress + "|" chiave pubblica

Adesso il server controllerà nella sua lista utenti connessi i dati e in caso positivo lo aggiungerà agli altri. La lista utenti sul server è memorizzata come un vettore dove ogni elemento è un oggetto della classe userObject che contiene appunto Username, MACAddress e chiave.

```
Public class userObject
{
        public String userName;
        public String mac;
        public int n, e;          // chiave pubblica composta da due valori numerici
}
```

Controllo: Cerco nel vettore se la Username è già presente, se non lo è autentico il client

In caso di autenticazione riuscita, invio al client la lista utenti connessi (dietro sua richiesta, vedi protocollo) nella modalità

|| Utente 1 || Utente 2 || ... || Utente n ||

dove ogni Utente è rappresentato come stringa: Username | mac | n | e

A questo punto resterò in attesa di eventuali richieste dal client.

Gestione della lista utenti

Autenticazione	Se l'username richiesto per l'accesso non è già presente allora posso inserirlo nella lista utenti
Invio	Il client richiede la lista invia "U" sulla Socket, risponderemo inviando una stringa con tutti gli username e le chiavi pubbliche associate
Aggiornamento	Nel momento in cui un utente si connette o disconnette la lista utenti cambia il suo stato, sarà quindi necessario aggiornarla invocando un metodo *informaClientConnesso()* e *informaClientDisconnesso()*
Disconnessione	Al momento della disconnessione scorreremo la lista ed elimineremo l'utente non più raggiungibile (vedi metodo *termina()*)

3.2 Gestione della connessione

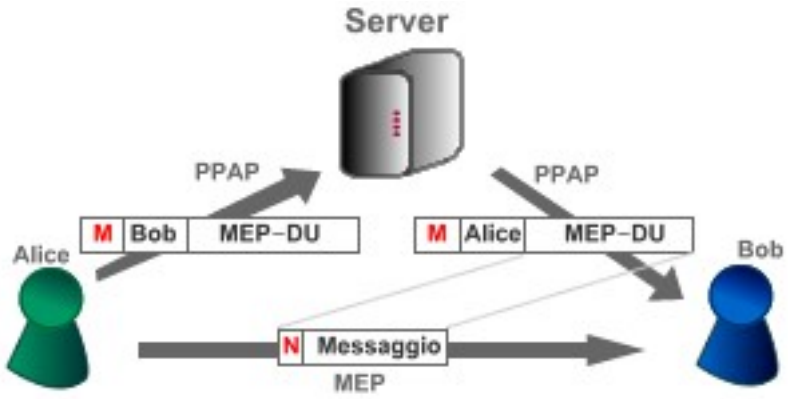
All'avvio del Server dovremo scegliere una porta su cui metterci in ascolto di eventuali richieste di connessione. Nel momento in cui un client si connette, cioè si stabilisce un canale di comunicazione, il server istanzierà un nuovo thread di gestione del singolo client, a questo punto mentre il thread proseguirà con l'autenticazione dell'utente, il processo principale potrà rimanere in ascolto (sulla porta indicata inizialmente) di altre eventuali richieste di connessione.
Una volta connesso al server, il client tenterà di autenticarsi inviando attraverso la Socket il suo username (vedi protocollo PPAP). In caso di username valido rinominerò il thread con l'username del client, risponderò inviando sul canale un "Y" e provvederò ad aggiornare la lista di tutti gli altri utenti connessi.
Bene, ora è stata impostata correttamente la connessione, il server può mettersi in ascolto delle richieste del client. Innanzitutto dovremo saper potergli inviare la lista degli utenti, inviargli un messaggio o comunicargli un aggiornamento della lista; con una serie di istruzioni di selezioni filtreremo i messaggi che l'utente ci invierà per poter far fronte correttamente alle sue richieste rispettando il protocollo di collegamento definito.

Riepilogo delle funzioni di connessione del Server:

- Creo un processo in ascolto su una porta in attesa di richieste di connessione
- Al momento della connessione sposto la comunicazione su un nuovo thread cosi che possa rimanere in ascolto di altre richieste
- Nel nuovo thread gestisco le richieste di autenticazione, di invio e inoltro messaggi, di gestione della lista: aggiornamento, invio di fronte a richiesta.
- Al momento della disconnessione, che avviene quando l'utente chiude la socket e provoca un eccezione, si invoca il metodo *termina()* che provvede ad eliminare i dati dalla lista utenti, disallocare il thread dalla lista dedicata, chiudere la socket e informare gli altri utenti della avvenuta disconnessione.

3.3 Il protocollo PPAP e il routing dei messaggi

Come anticipato nella definizione del protocollo, il server non entra nel merito del carico dei messaggi o del loro tipo, si serve solo del suo mittente e destinatario per poter effettuare un routing corretto dei dati. Nella classe UserHandler è definita l'implementazione di tale servizio, in particolare possiamo notare come all'arrivo di un dato con intestazione "M" si invochi il metodo *inviaMessaggio(messaggio)* che provvederà ad estrarre dalla PPAP-DU il destinatario e la MEP-DU, cercherà sulla lista dei thread utente se il destinatario è presente e una volta trovato gli invierà una nuova PPAP-DU con i campi "M+mittente+MEP-DU", in pratica sui dati in arrivo controllerà la loro consistenza e se il destinatario è valido effettuerà uno switch immettendo al suo posto il mittente. Effettuato questo piccolo accorgimento, potremo finalmente inoltrare il messaggio sulla socket dell'utente.



4. Il Client

L'utente, esigente e tecnologico, desidera un servizio pulito, veloce e affidabile. Dovremo essere molto attenti nella fase di progettazione dell'applicazione client-side a implementare un codice stabile e un interfaccia accessibile. Vediamo un po' di cosa stiamo parlando.

4.1 Le funzionalità

Cosa dovremmo trovare su un programma di messaggistica? Bè innanzitutto una parte di autenticazione, un altra di definizione dei parametri di connessione immagino, e poi che altro? Forse una sezione di listing degli utenti dove è possibile selezionarli e avviare una conversazione, magari potendo anche scegliere direttamente nella finestra se usare messaggi in chiaro o criptati.
Si, non sarebbe male, vediamo di provare a realizzarlo.

4.2 La connessione al Server

Per informazioni sulle modalità di connessione vedere il protocollo PPAP e la gestione della connessione sul server.
La classe connessioneClient.java si occupa di implementare i vari metodi atti alle procedure di connessione al server, cioè dell'instaurazione della connessione punto-punto che segue le caratteristiche elencate nel protocollo PPAP. I metodi implementati sono: *connetti()* che si occupa di instanziare una Socket e un nuovo thread dedicato alla sua lettura, *autentica()* che si occupa dell'invio dei dati di login e dello starting del thread, *richiediListaUtenti()* che effettua la richiesta della lista, *inviaMessaggio()* che concatena e invia sulla socket una stringa M+destinatario+MEP-DU, *macAddress()* che ritorna l'indirizzo fisico di rete e utilizza il metodo *getMACFromByte()* con cui converte la stringa di bit in notazione esadecimale. Infine la *Disconnetti()* con cui chiudiamo la socket e lo stream in output e interrompiamo il thread in lettura.

4.3 L'interfaccia GUI e i messaggi

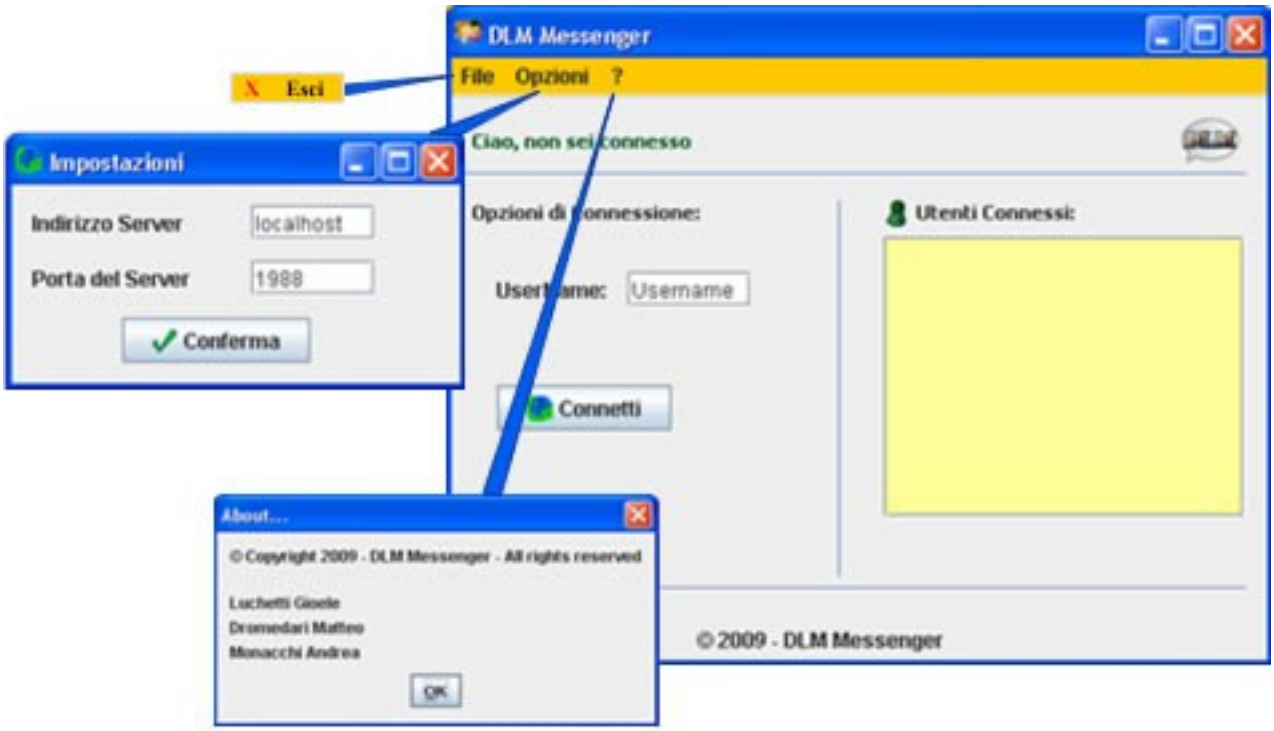
L'interfaccia GUI (Grafic User Interface) è un paradigma di sviluppo software che consente l'interazione utente/terminale tramite un ambiente grafico anziché tramite l'impartizione di comandi testuali che debbano essere memorizzati e impartiti a linea di comando (CLI Command Line Interface). All'aumentare delle funzionalità e delle alternative il tempo necessario per prendere una decisione aumenta in modo lineare (legge di Hick), è preferibile quindi utilizzare oggetti che possano rendere piu veloce l'esecuzione delle operazioni. Nell'era della multimedialità che avanza, sembra improbabile il successo di un programma a linea di comando dove l'accessibilità non è garantita a tutti gli individui e a livello psicologico sia dimostrato come il suo approccio troppo formale renda un messaggio diverso da quello per cui il programma è stato realizzato.

L'interfaccia di DLM è stata suddivisa in colori diversi per aumentare il grado di accessibilità e riconoscimento visivo delle funzionalità, secondo una coerenza estetica che possa migliorare l'apprendimento, anche tramite la diversificazione dei supporti tramite l'utilizzo di icone e immagini che possano essere meglio memorizzate e ricordate per velocizzare il riconoscimento dei comandi negli utilizzi futuri. La finestra deve essere un ambiente coinvolgente, un punto di accesso dove ogni richiesta possa essere soddisfatta in modo rapido, per questo motivo è stato preferito l'utilizzo dei menù e della visualizzazione multiform che possa evidenziare le differenze tra i contenuti e dare maggior grado di interattività a tutta l'interfaccia.

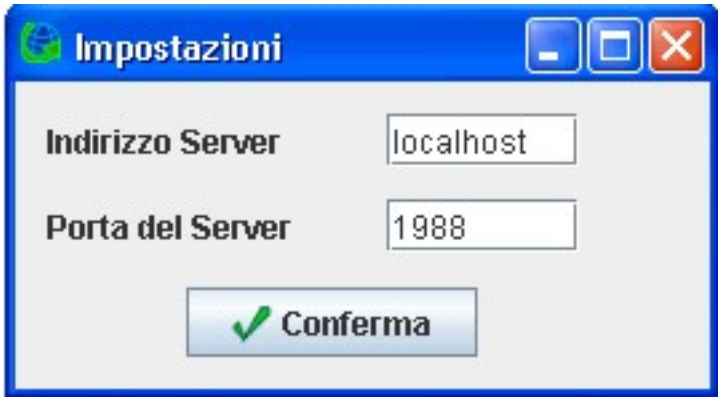
Per la realizzazione delle interfacce grafiche, java mette a disposizione del programmatore apposite librerie contenenti classi predefinite, la AWT (Abstract Window Toolkit) e la Swing. La differenza principale tra le due è che AWT è legata al sistema grafico del sistema operativo ove sono istallate mentre le Swing essendo scritte in java sono visualizzabili allo stesso modo su tutti i sistemi che ne fanno uso. Le swing offrono inoltre molta piu elasticità essendo ogni oggetto configurabile e riutilizzabile personalizzandolo.

Di seguito presentiamo le interfacce grafiche Main e Messaggi con le loro sottoaree.

Struttura della GUI (Main)

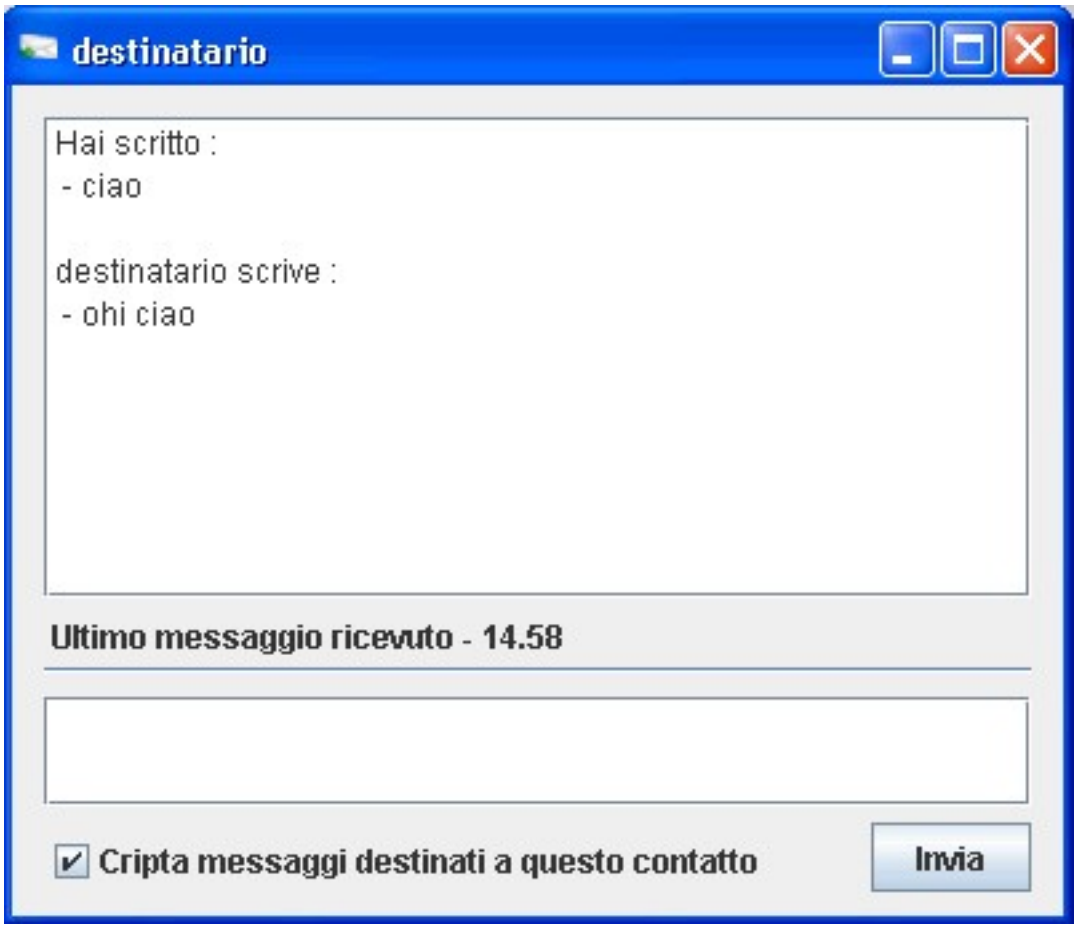


La finestra delle opzioni di connessione



nota: la finestra opzioni è settata a default con indirizzo di loopback e porta 1988

Il form messaggi

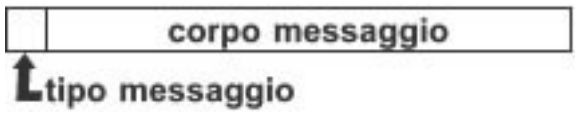


4.4 L'algoritmo di sicurezza

Nella definizione del protocollo MEP è stata anticipata l'esigenza di un algoritmo che renda sicura una comunicazione tra due utenti (Alice e Bob) e la renda incomprensibile ad un eventuale terzo che si possa interporre tra essi.

Il protocollo MEP

Riportiamo la MEP-DU (unità dati protocollo MEP):



- il tipo messaggio può assumere valore S o N
- il corpo messaggio è una stringa di caratteri alfanumerici

Ma come possiamo cifrare un semplice testo in qualcosa che gli altri non possano comprendere ma che soprattutto il nostro destinatario possa comunque capire dalle informazioni fornite?
Bè potremmo utilizzare una traslazione del nostro alfabeto sull'esempio del tanto famoso cifrario di cesare (sostituzione monoalfabetica), oppure potremmo creare un algoritmo di modifica della stringa e tenerlo nascosto in modo che nessuno ne possa venire a conoscenza (sicurezza per occultamento).
La crittografia (dal greco: “scrittura segreta”) è basata però su un teorema molto importante, quello di kerchoff, il quale enuncia come tutti gli algoritmi debbano essere pubblici, solo le chiavi possano essere segrete.
In questo modo le nostre principali idee cadono cosi come la pessima sicurezza che avrebbe avuto il nostro algoritmo se le avessimo utilizzate.
Supponiamo i nostri classici Alice e Bob in una comunicazione sicura.
Avremo un testo in chiaro che dovremmo cifrare tramite una funzione a cui passeremo come parametro una chiave di cifratura. In uscita avremo il nostro testo cifrato che trasmetteremo a Bob.
Se in mezzo a loro si interponesse un intruso non ne capirebbe il contenuto ma potrebbe adoperarsi per memorizzarlo e ritrasmettere in tempi diversi o provare a decriptare il cifrario (criptoanalisi).
Potremmo indicare il testo cifrato come $C = E_K(P)$ dove P è il testo in chiaro che è crittato dalla funzione E utilizzando la chiave K. Allo stesso modo $P = D_K(C)$ indica l'operazione di decifrazione.
Tralasciando lo studio della crittografia e l'ipotesi (per ora poco realizzabile) di creare un nuovo algoritmo di sicurezza possiamo dedicarci all'analisi di uno di quelli esistenti e che con accorte modifiche potremo utilizzare nel nostro progetto per una sicurezza di base.

La crittografia asimmetrica

Innanzitutto per completezza enunceremo la differenza tra cifrario e codice, con il primo si intende una trasformazione carattere per carattere senza considerare la struttura linguistica, con il secondo invece si rimpiazza una parola con un altra o con un simbolo. Quindi mentre un cifrario sostituisce tenendo conto della sintassi, un codice lavora sulla semantica del messaggio.
Gli algoritmi di cifratura si dividono in due diversi tipi, quelli simmetrici e quelli asimmetrici. Nel primo caso viene utilizzata la stessa chiave per cifrare e decifrare un messaggio, l'operazione è più veloce ma comporta l'utilizzo di una chiave condivisa nella comunicazione tra due diversi individui. Chiave che dovremo distribuire se vogliamo che il nostro destinatario possa capire ciò che stiamo dicendo e se qualcuno ne venisse a conoscenza non potremmo più trasmettere con sicurezza.
Gli algoritmi asimmetrici invece usano due chiavi, quella privata conosciuta solo dall'utente che la possiede, quella pubblica, di ogni utente, conosciuta da tutti gli altri.
Quando un utente vuole comunicare segretamente con un' altro, e quindi vuol'essere sicuro che i messaggi che scriverà saranno interpretabili solo dal destinatario, cifrerà il messaggio da inviare con la chiave pubblica del destinatario, così che per la particolare proprietà delle due chiavi, il messaggio sarà decifrabile solo usando la chiave privata del destinatario a cui il mittente voleva scrivere. Le chiavi saranno costruite in modo che la privata non sia derivabile facilmente dalla pubblica e con un algoritmo i cui passi saranno visibili a chiunque. (crittografia pubblica)

L'algoritmo RSA

Il protocollo MEP utilizza il metodo di cifratura a chiave pubblica RSA.
Questo particolare tipo di cifratura, inventato da R. Rivest, A. Shamir e L. Adleman nel 1978, prevede l'uso di due chiavi, una privata e una pubblica, per ogni utente che partecipa alla conversazione (crittografia asimmetrica). Al momento dell'accesso al server creeremo la nostra coppia di chiavi e provvederemo a distribuire quella pubblica al momento dell'autenticazione.
In questo modo tutti i client che mi vedranno connesso saranno capaci di utilizzare la mia chiave pubblica per potermi inviare messaggi segreti. Per comprendere meglio potremmo fare una analogia con dei lucchetti, ognuno di noi ha lucchetti (diversi tra loro) di tutti gli utenti, ma ognuno ha solo la chiave del proprio. Al momento dell'invio chiudo il messaggio con il lucchetto del destinatario, unico avente diritto all'apertura e alla lettura del contenuto.
Aumentando (il numero di bit) il grado di sicurezza del lucchetto sarà quasi impossibile poterne ricostruire la chiave. RSA richiede chiavi di almeno 1024 bit per offrire una buona sicurezza (contro i 128 bit degli algoritmi a chiave simmetrica)

La classe che implementa la cifratura nel protocollo MEP è la *RSA.java*.
Questa definisce tre attributi numerici (int e, n, d;) per identificare le chiavi:

(n,e) -> chiave pubblica
(n,d) -> chiave privata

oltre alla definizione di tre metodi pubblici che provvedono alla creazione delle stesse e al loro utilizzo nella cifratura e decifratura dei messaggi.

La sicurezza di RSA è basata sulle proprietà dei numeri primi ovvero sull'aritmetica congruenzale e la teoria dei campi, per cui è difficile scomporre in fattori i numeri molto grandi.
Per prima cosa si generano due numeri primi p e q molto grandi (tipicamente di 1024 bit) che nel nostro algoritmo (usato per una sicurezza di base) non superano le 3 cifre decimali.
Si calcolano poi ***n*** ed ***m*** come prodotto ***n = p*q*** e ***m = (n-1)*(q-1)***. A questo punto per generare l'intera chiave pubblica devo trovare un numero ***e*** che non abbia fattori primi in comune con **m**.
Adesso si dovrà calcolare la chiave privata (n, d), cioè quel numero ***d*** inverso moltiplicativo di **e** tale che ***e*d mod m = 1.*** Generate le due chiavi, attraverso i metodi *cifra(n, e, messaggio)* e *decifra(messaggio)*, potremo agire sul testo del messaggio nel seguente modo:

In fase di cifratura il mittente dovrà codificare ogni carattere del suo messaggio in un numero intero da uno a **n**, (nel nostro caso implementato con l'esadecimale del carattere) poi usando la chiave pubblica del destinatario calcolerà per ogni carattere codificate **carattere^{e} mod n**.

Il destinatario, una volta ricevuto il messaggio crittato, invocando il metodo decifra (al quale è passato il solo messaggio in quanto la chiave viene ricavata direttamente dagli attributi dell'oggetto instanziato sulla classe RSA) provvederà alla sua decifratura trasformando ogni carattere nel corrispondente valore esadecimale e calcolando **carattere^{d} mod n**.
In questo modo solo il destinatario potrà decifrare il messaggio a lui inviato.

Come potrebbe un eventuale intruso malintenzionato che si interponesse nella comunicazione a decifrare il messaggio senza conoscere la chiave privata?
Esso dovrebbe cercare i fattori primi **p** e **q** che hanno generato **n** e per cui vale **p*q=n**. Successivamente dovrebbe dedurne **m** e infine calcolare **d** come inverso moltiplicativo di **e** (vedi sopra). Se i numeri primi **p** e **q** usati sono molto piccoli sarà relativamente semplice fattorizzarli. Visto però che RSA utilizza numeri primi con 100/200 cifre decimali non esistono metodi sufficientemente veloci atti alla decomposizione di numeri così grandi.

Esempio:
un numero di 129 cifre è stato fattorizzato nell'aprile del 1994 utilizzando un cluster di circa 600 computer.

5. Testing

Un controllo del livello collegamento è stato effettuato inizialmente sul sistema e le comunicazioni risultano perfettamente funzionanti, sia in rete locale che tramite internet (adottando accorgimenti quale il redirezionamento delle porte per risolvere il problema dell'host hiding da router).
Alla versione 0.1 DLM presenta le caratteristiche di messaggistica ma non quelle di crifratura dei messaggi, messaggi che presentano inizialmente problemi quando contengono caratteri accentati e speciali, problema risolto con la versione 0.15 in attesa di una versione successiva che implementa la comunicazione del MAC Address da parte di un client al server (utilizzabile a fini statistici o di sicurezza), ma è con la versione 1.0 che si da un cambio netto al servizio e si introducono anche i messaggi cifrati (vedi capitolo dedicato) delineando il protocollo MEP (di cui abbiamo già parlato) e garantendo con finestre di helping e dialogo un funzionamento stabile di fronte a eventuali errori dell'utente o del programma.

Gli sviluppatori

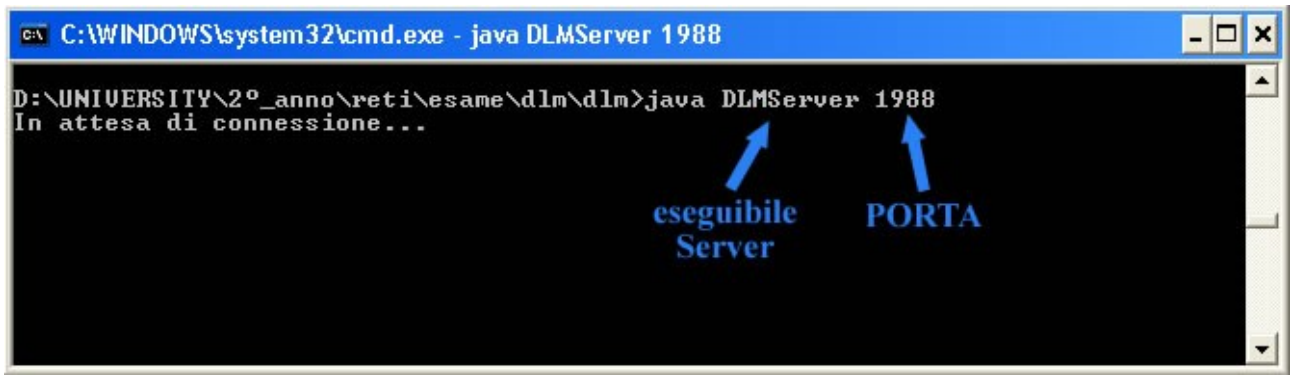
6. Manuale di utilizzo

Ogni buona applicazione che si rispetti va accompagnata da una descrizione delle operazioni da eseguire per ottenere un risultato. Le operazioni principali del nostro software sono la connessione, il dialogo con un utente, la scelta del tipo di messaggio.

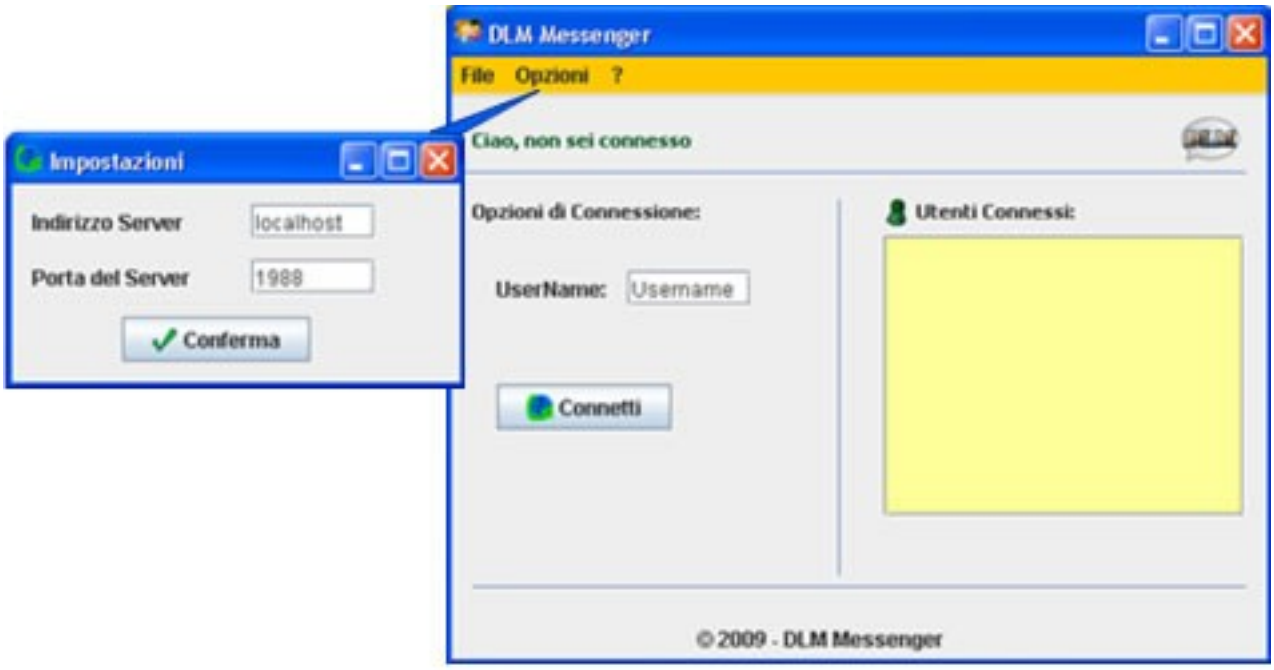
6.1 Un esempio pratico

Riportiamo un generico esempio delle operazioni principali eseguibili tramite DLM:

Server: avvio



Client: la connessione al server

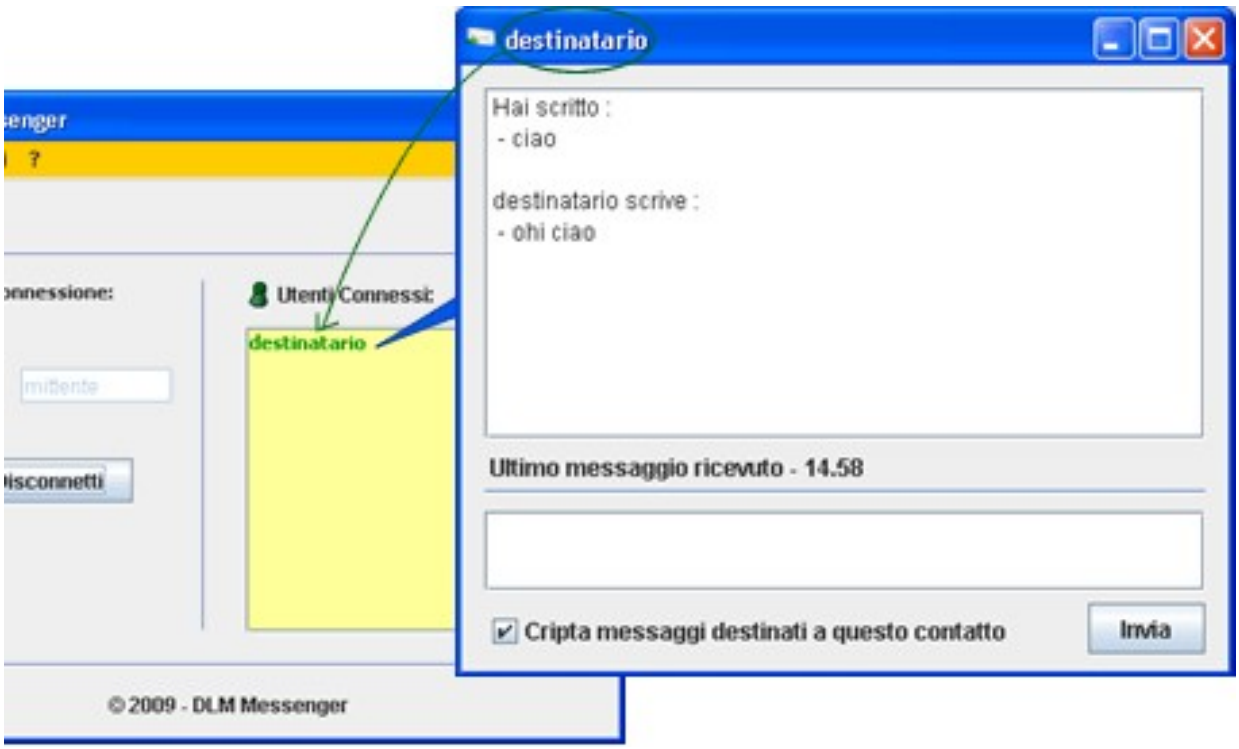


Per connettersi al Server è necessario che le opzioni di connessione inserite siano valide:

- l'username deve essere utilizzabile
- le impostazioni di connessione (menu opzioni) devono essere corrette (indirizzo IP e porta)

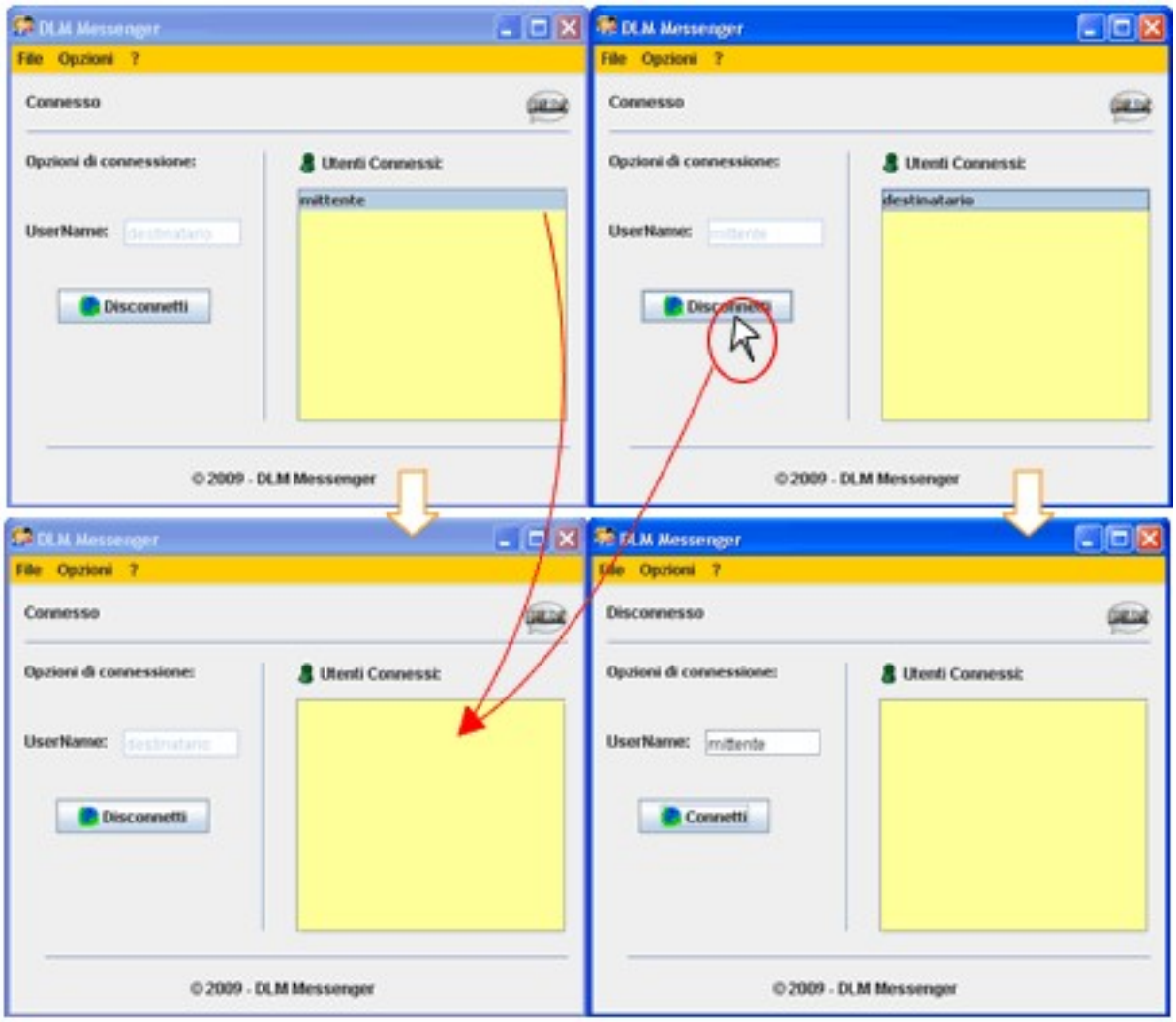
Infine premere sul bottone connetti e attendere che il collegamento con il server venga stabilito.

Client: dialogo con un utente della lista e scelta del tipo di messaggio



Supponiamo che io sia l'utente mittente.
Facendo doppio clic su un elemento della lista (destinatario) si aprirà una finestra di conversazione, ogni messaggio che scambierò con questo contatto apparirà sulla form messaggi. E' data la possibilità di scegliere se inviare messaggi in chiaro o criptati.

Client: disconnessione



Come vediamo dall'immagine:
Inizialmente gli utenti sono connessi e si vedono l'un l'altro.
La disconnessione del mittente (ottenuta cliccando sul tasto disconnetti) provoca anche l'aggiornamento della lista dell'utente destinatario. Infine l'utente mittente si trova scollegato, mentre l'utente destinatario è ancora collegato ma non può più parlare con nessuno (lista utenti vuota).

7. Appendice

In questa sezione riportiamo elementi di basilare importanza nella realizzazione del nostro progetto ma che non hanno potuto avere spazio durante la trattazione

7.1 Le Socket

L'elemento che sta alla base della nostra connessione è la Socket, un canale di comunicazione che ci permette di scambiare dati, vediamo in dettaglio di cosa stiamo parlando.

7.1.1 Porte e Socket

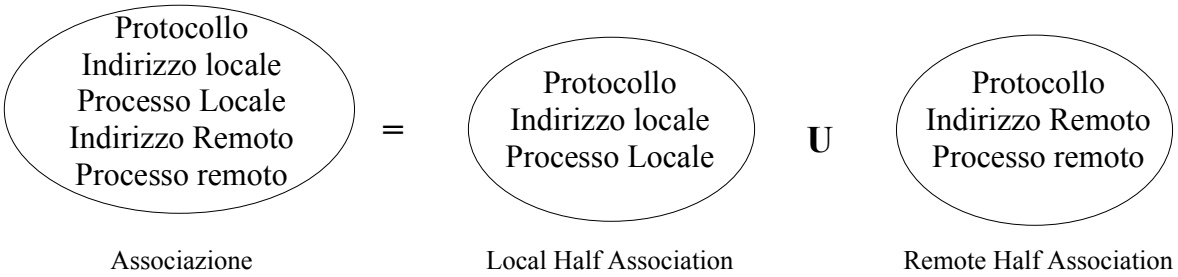
Il primo passo che un processo deve effettuare per poter comunicare con un altro è quello di rendersi visibile ai livelli sottostanti di gestione della rete, così che possa essere visto come reale entità terminale di comunicazione. Questa identificazione viene realizzata tramite dei numeri detti Porte o Port Address che sono indirizzi di livello trasporto, sono compresi tra 0 e 65535 ma è stato dedicato il range numerico tra 0 e 1024 a servizi di comune impiego (sono dette perciò porte ben note). Sulle porte ben note esistono programmi detti demoni di gestione del servizio, invece di caricarli tutti all'avvio e occupare memoria si avvia un solo demone (in Unix chiamato INETD Internet Daemon), tale applicazione nel momento in cui è richiesto l'utilizzo di una porta genera un nuovo processo associato al demone appropriato per gestire la richiesta. Per velocizzare tale operazione è data possibilità di fissare le porte più usate come all-running (sempre attive) e far gestire a INETD le altre. Tornando alle metodiche di creazione di una connessione, vediamo come il semplice identificatore della porta non possa essere in grado di descrivere in maniera univoca una connessione. Parleremo del concetto di Associazione che è una quintupla di informazioni necessarie a descrivere univocamente una connessione.

Associazione = (Protocollo, Indirizzo Locale, Processo Locale, Indirizzo remoto, Processo remoto)

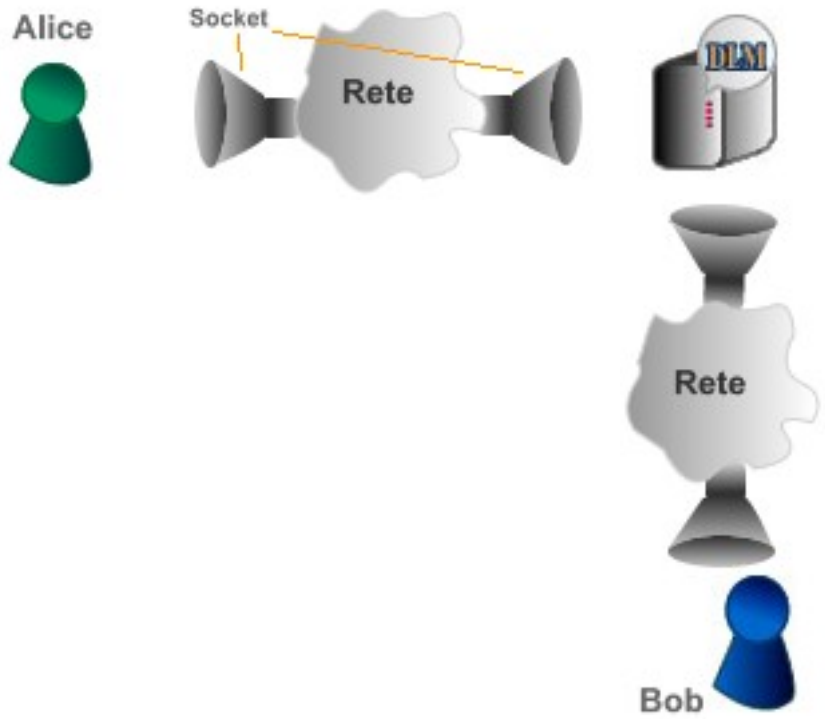
Es: (TCP, 198.240.12.4, 1880, 194.56.236.2, 21)

Come già anticipato i numeri di porta identificano i processi associati al livello applicazione, il livello trasporto effettuerà un multiplexing su tali indirizzi, consegnando i dati al processo terminale. Per indirizzo si intende invece l'identificativo della macchina sullo strato network (IP), che per standard dovrebbe essere univoco (in realtà non è così per l'esistenza dei NAT).

Ma come fanno due processi posti su sistemi diversi a dialogare e creare una associazione??
Bè possiamo vedere una associazione come l'unione di due half association, una locale e una remota. Due processi sorgente/locale e destinatario/remoto stabiliscono una connessione in modo univoco mediante la creazione di queste due half association che per convenzione sono chiamate Socket (intese come prese o punti terminali della connessione).



Ogni Socket quindi possiede un indirizzo (o numero) composto dall'IP dell'host e da un numero di 16 bit (2 byte) locale chiamato porta. Per stabilire una connessione TCP deve stabilirsi un collegamento tra una Socket su una macchina di invio e una Socket su una macchina ricevente. Una Socket supporta più connessioni contemporaneamente (Es: Webserver), due o più connessioni possono infatti terminare sullo stesso Socket. Una connessione è identificata dagli identificatori di Socket delle due estremità (Socket1, Socket2). Tutte le connessioni TCP sono full-duplex punto-punto, quindi il traffico può procedere contemporaneamente in due direzioni. Una connessione TCP è un flusso di byte, essa però non supporta il multicasting e il broadcasting. Non entreremo in merito delle problematiche che si verificano al momento della inizializzazione e del rilascio della connessione, come quello dei pacchetti duplicati ritardati o il dilemma dei due eserciti, problematiche comunque risolte a livello trasporto rispettivamente tramite handshake a tre vie e timer, ma che non ci interessa trattare in quanto le nostre applicazioni funzionano ad un livello di astrazione superiore, astrazione che ci permette di garantire la portabilità richiesta.



7.1.2 Le Socket in Java

Il package *Java.net* mette a disposizione del programmatore le API necessarie allo sviluppo di applicazioni Client/Server basate su Socket. Un indirizzo IP è rappresentato dalla classe **InetAddress** che fornisce tutti i metodi necessari a manipolare un indirizzo internet necessario all'instradamento dei dati sulla rete e consente la trasformazione di un indirizzo nelle sue varie forme (stringa, decimale separata da punto,...). Le classi **Socket** e **ServerSocket** invece implementano rispettivamente un socket client e uno server per la comunicazione orientata alla connessione, la classe **DatagramSocket** implementa i socket per i servizi senza connessione mentre per concludere la classe **MulticastSocket** fornisce supporto per i Socket di tipo multicast.

7.1.2.1 La classe ServerSocket

La classe ServerSocket rappresenta quella porzione del Server che può accettare richieste di connessione da parte del client. Questa classe deve essere istanziata passando come parametro il numero della porta su cui il server sarà in ascolto. Il metodo più importante è *accept()*. Questo metodo mette il thread in attesa di richieste di connessione (bloccante) da parte di un client sulla porta specificata nel costruttore. Quando una richiesta di connessione va a buon fine, viene creato un canale di collegamento e il metodo ritorna un oggetto Socket connesso con il client. Nel caso di connessioni multicast, affinche il server possa rimanere in ascolto sulla porta specificata in attesa di altre eventuali connessioni è necessario creare un nuovo thread a cui passare il socket connesso al client. Quindi la porta principale sarà dedicata all'ascolto delle richieste di connessioni e quando un client viene servito è spostato su un nuovo thread e una nuova Socket associata ad una nuova porta, liberando cosi il canale principale che può tornare all'ascolto di nuove richieste. Questo pone un ipotetico limite al numero di connessioni effettuabili avendo affermato precedentemente che il numero di porte massimo è 65535.

7.1.2.2 La classe Socket

La classe Socket rappresenta una connessione client/server TCP full-duplex. La differenza tra server e client sta nella modalità di creazione di un oggetto di questo tipo. A differenza del server in cui l'oggetto socket viene creato dal metodo *accept()* della classe ServerSocket, il client dovrà provvedere a creare una istanza di Socket manualmente tramite il costruttore messo a disposizione dalla classe. Esso accetta due parametri, il primo rappresenta il nome dell'host cui ci si vuole connettere, il secondo la porta su cui il server è in ascolto. Una volta invocato il costruttore sarà tentata la connessione generando un eccezione nel caso in cui il tentativo non sia andato a buon fine.
Essendo la Socket bidirezionale (full-duplex) essa avrà un metodo *getInputStream()* e un metodo *getOutputStream()*. I metodi *getPort()* e *getLocalPort()* possono essere utilizzati per ottenere informazioni sulla connessione mentre il metodo *close()* rilascia la connessione e libera la porta utilizzata.

7.2 Il codice sorgente delle applicazioni Client e Server

Riportiamo in questa sezione le varie classi java, dell'applicazione server e di quella client, il cui funzionamento è stato già anticipatamente trattato, così da rendere effettivamente completa l'esposizione della realizzazione di questo servizio.

GUIDLMClient.java
GUImessaggi.java
connessioneClient.java
clientThreadLettura.java
userObject.java
frameOpzioniConnessione.java
RSA.java

DLMServer.java
userHandler.java
userObject.java