

Relazione su “Chatline multicient”

Chen Lijun 4E

Table of Contents

1. Introduzione	1
1.1. In cosa consiste il progetto	1
1.2. Materiali usati	3
2. L'applicazione	4
2.1. Esecuzione dell'applicazione	4
2.2. Struttura dell'applicazione	4
2.3. Funzionamento generale dell'applicazione	5
2.3.1. Lato server	5
2.3.2. Lato client	8
2.4. Codice Sorgente completo	9
2.4.1. Client.java	9
2.4.2. Server.java	11
2.4.3. ServerDispatcher.java	12
2.4.4. ClientSender.java	14
2.4.5. ClientListener.java	16
2.4.6. Message.java	17
2.4.7. MessageHistory.java	19
2.4.8. ClientInfo.java	20
3. Conclusione	23

Chapter 1. Introduzione

“*Chatline multicient*” è nato come progetto supplementare all’area di progetto della classe 4E indirizzo informatico dell’istituto *IISS Galileo Galilei*.

L’area di progetto in questione riguardava principalmente la materia *Sistemi e reti*. Consisteva infatti nella creazione di una LAN (*Local Area Network*) usando dispositivi di rete (switch e router), questi ultimi forniti dalla scuola.

Ora, avendo una LAN, si è anche pensato di testarlo con qualcosa di concreto e, perché no, divertente. Così è nato, a seguito alla proposta del docente di *Informatica* Antimo Marzochella, il presente progetto, che ha come fine ultimo, quello di integrare le conoscenze e abilità maturate durante l’anno scolastico 2016-2017 inerenti alla programmazione con il quanto appreso in *Sistemi e reti*.

1.1. In cosa consiste il progetto

Il presente progetto consiste nel **completamento** di un applicazione *client-server*, di cui una prima versione funzionante era già stata fornita dal docente.

L’applicazione, che d’ora in poi denomineremo “**Chatline**”, disponeva inizialmente di un server a cui più client si potevano connettere. In questo caso il server non faceva altro che smistare i messaggi in arrivo a tutti gli altri client che sono connessi ad quest’ultimo (quello che fa un Hub).

Ai singoli alunni è stato dunque richiesto di sviluppare la funzionalità che permetta di salvare le conversazioni in un file di testo, che poi dovrà poter essere letto per presentare ad ogni utente che si collega al server una storia dei messaggi precedenti.

Applicazione client-server

Un tipo di applicazione in cui ci sono almeno due programmi in esecuzione:

- il server
- il client

Server

programma che resta in attesa delle richieste di servizio da parte di uno o più client

Client

programma che inizializza una connessione, attraverso la quale si collega al server e può richiedere a questo servizi di cui ha bisogno.

Il modello client-server è solo un modo di strutturare un applicazione e non obbliga l'uso di tecnologie specifiche: basta che ci sia un componente che offra dei servizi e uno che li usufruisce e che quindi i due comunichino in qualche modo.

Di fatto però, quando si parla di applicazione client-server, si intende un applicazione di rete in cui questi ultimi comunicano tramite la rete, e in particolare tramite **Socket**.

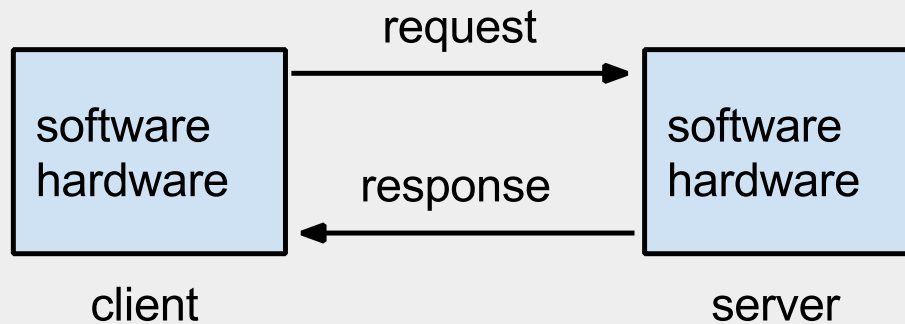


Figure 1. Modello client-server

Socket

Un socket rappresenta un punto di connessione, cioè identifica univocamente un processo (quindi un servizio) all'interno di un computer. Un socket è formato da un indirizzo IP e una porta TCP.

Ad ogni applicazione in esecuzione che usa TCP (o UDP) come protocollo di trasporto viene infatti associato dal sistema operativo un numero identificativo, chiamato porta.

Le porte virtuali sono 65.535, di cui le prime 1.023 sono dette *Well Known Port*, in quanto associate a servizi noti come HTTP, FTP, ecc.

1.2. Materiali usati

Linguaggio di programmazione utilizzato: **Java**.

Java

Linguaggio imperativo, a tipizzazione statica, orientato agli oggetti (OOP), interpretato (dalla JVM) e cross-platform.

Per la stesura del codice si è fatto ricorso a:

- **Vim** insieme a vari plugins (soprattutto *Eclim*) per la stesura del codice
- Siti come [Stack Overflow](#) e in generale [Google](#)
- **Git**: famoso programma di controllo versione

Invece per l'elaborazione della relazione sono stati usati i seguenti programmi

- **Asciidoctor**: programma che traduce in formati quali HTML, PDF, EPUB, etc. file scritti nel linguaggio a marcatori *Asciidoc*
- **Plantuml**: programma che fornisce un linguaggio dichiarativo e sulla base di questo genera immagini (diagrammi di classi, flow-chart, etc.) appropriati

Chapter 2. L'applicazione

2.1. Esecuzione dell'applicazione

Esecuzione server

```
$ java Server <porta TCP>
```

Esecuzione client

```
$ java Client <indirizzo server> <porta TCP>
```

2.2. Struttura dell'applicazione

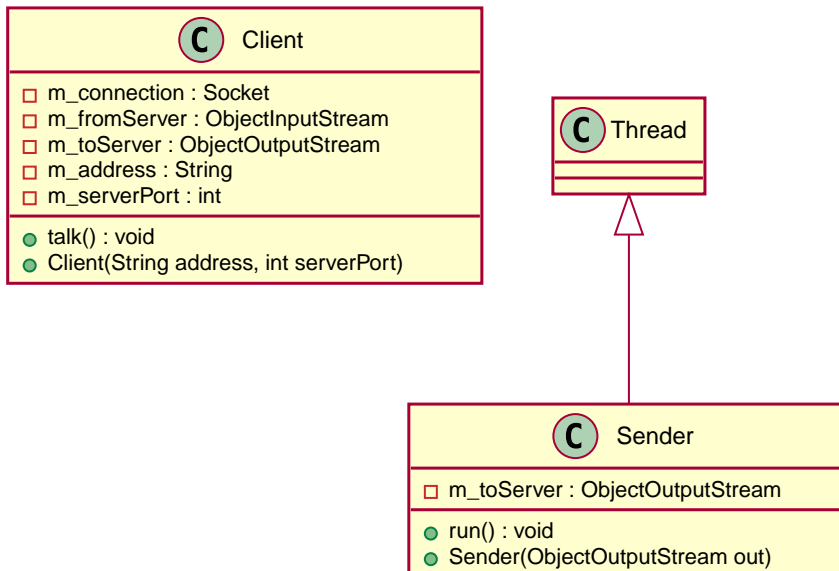


Figure 2. Server

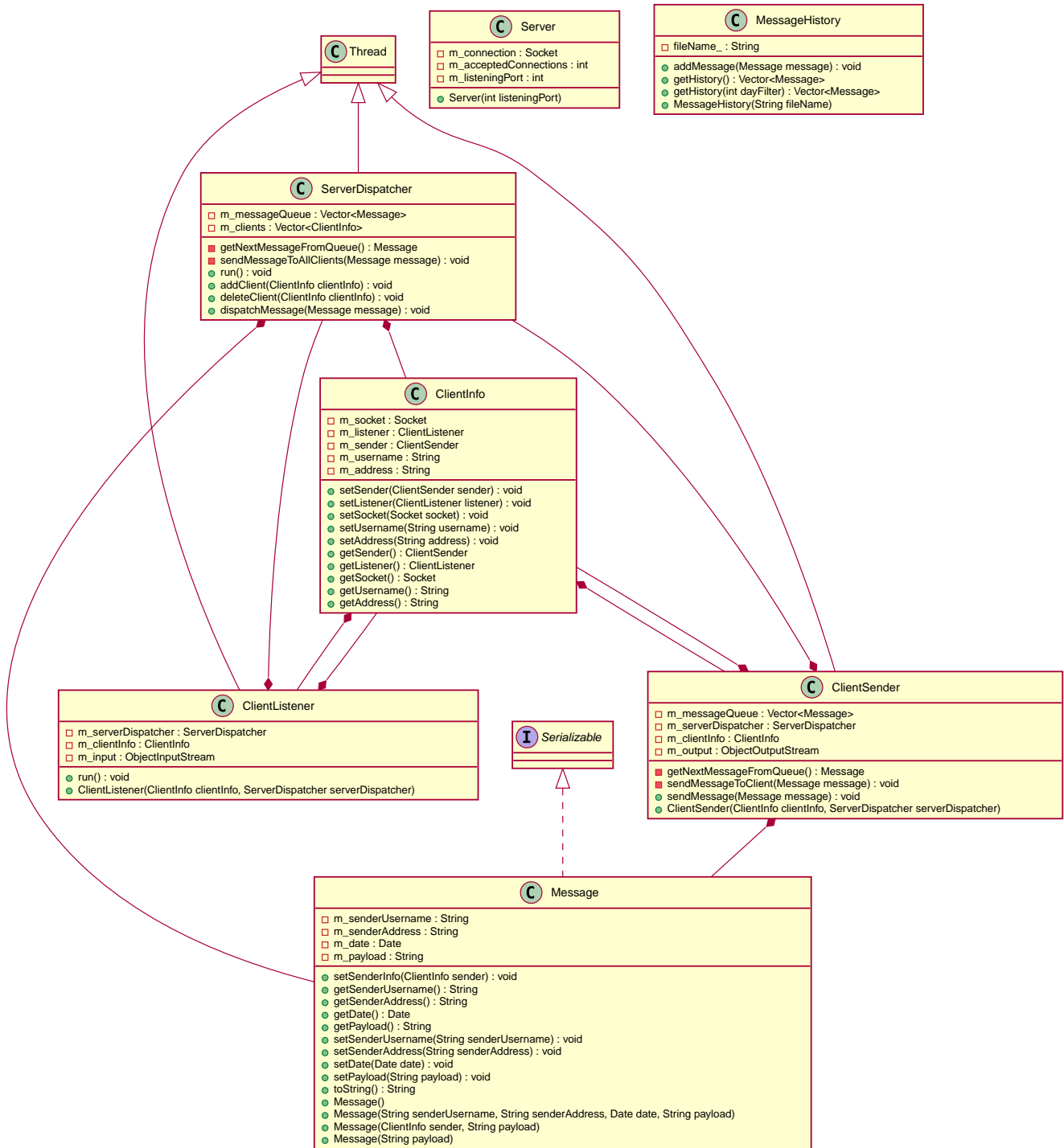


Figure 3. Client

2.3. Funzionamento generale dell'applicazione

2.3.1. Lato server

- Il server occupa una porta TCP.
- Il server crea un thread a parte, *ServerDispatcher*, che si occupa di inoltrare i messaggi a tutti i client.
- Il server resta in attesa di una richiesta di connessione.
- Al ogni richiesta a di connessione il server crea 2 thread separati, uno che aspetta di ricevere messaggi dal client (*ClientListener*) e l'altro che si occupa di inviare i messaggi al client

(*ClientSender*), associati alla connessione tramite un oggetto della classe *ClientInfo*.

Per cui ogni *ClientSender* e *ClientListener* avrà come membro la *ClientInfo* a cui è associato.

- Il server poi aggiunge al *ServerDispatcher* le informazioni (*ClientInfo*) riguardo la connessione appena creata (*ServerDispatcher* possiede un elenco di connessioni a cui poi inoltrerà i messaggi).
- Quando arriva un messaggio al server, il *ClientListener* chiamerà il metodo della classe *ServerDispatcher* che si occupa di inoltrare il messaggio a tutti i client mandandoli al *Sender* associato.

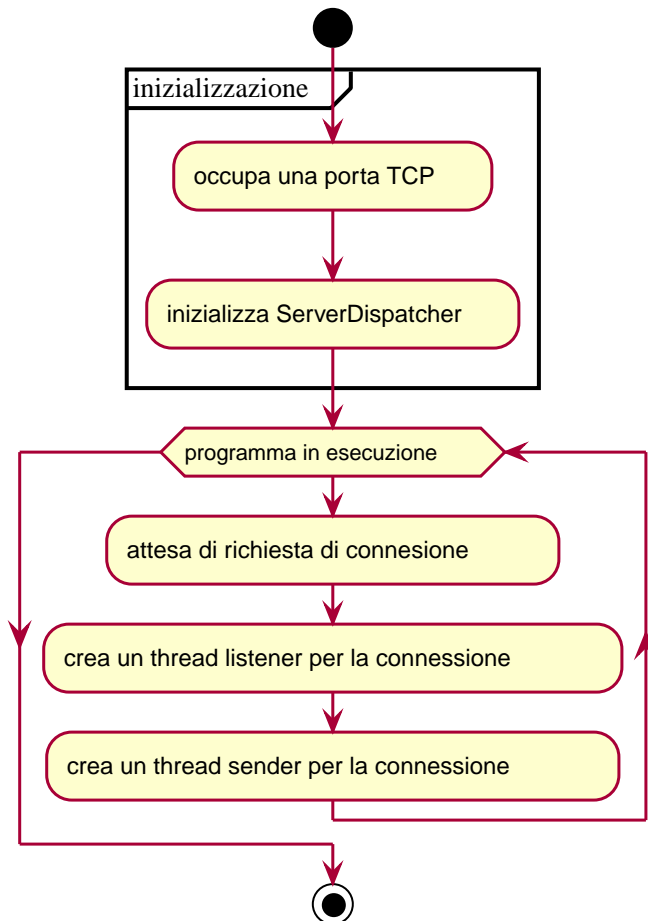


Figure 4. Server

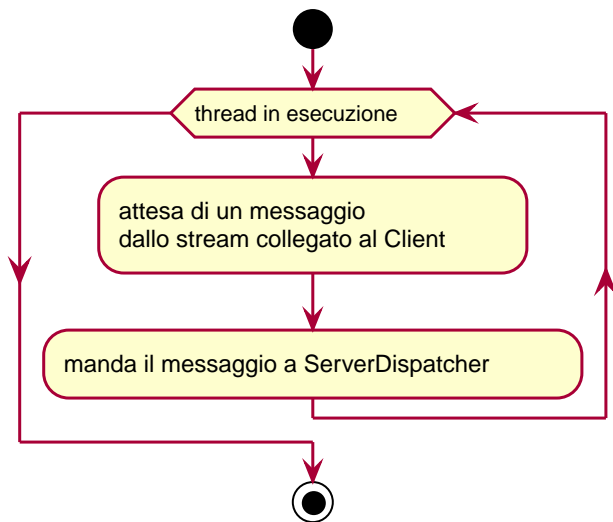


Figure 5. *ClientListener*

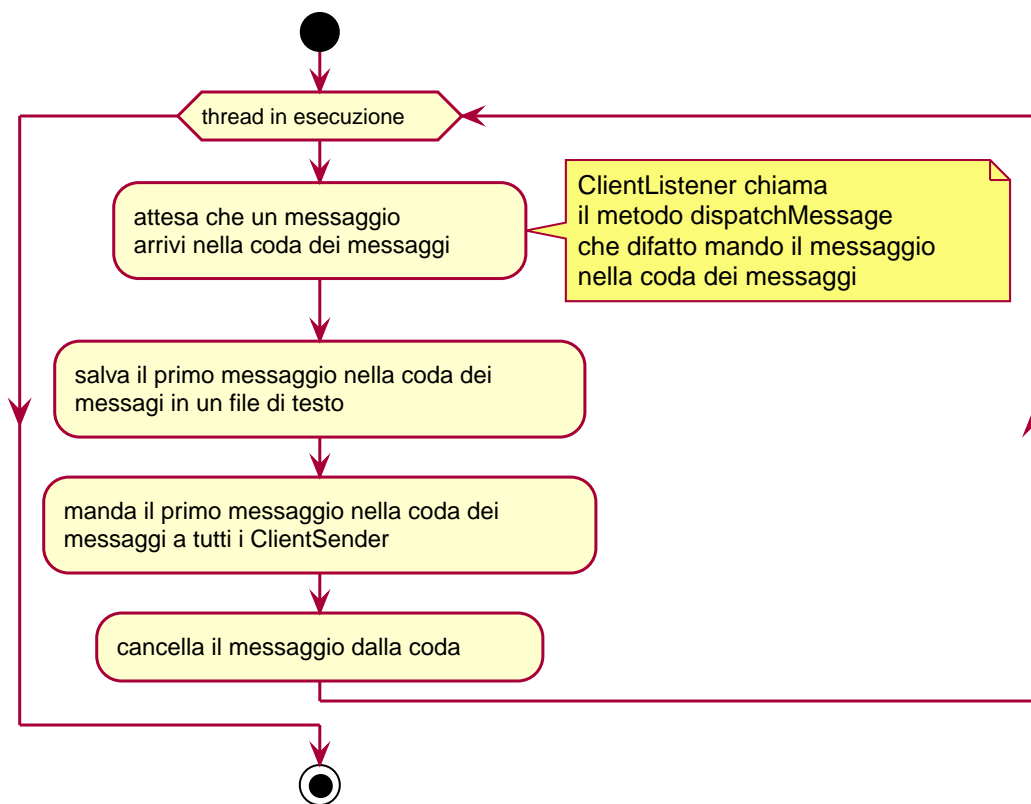


Figure 6. *ServerDispatcher*

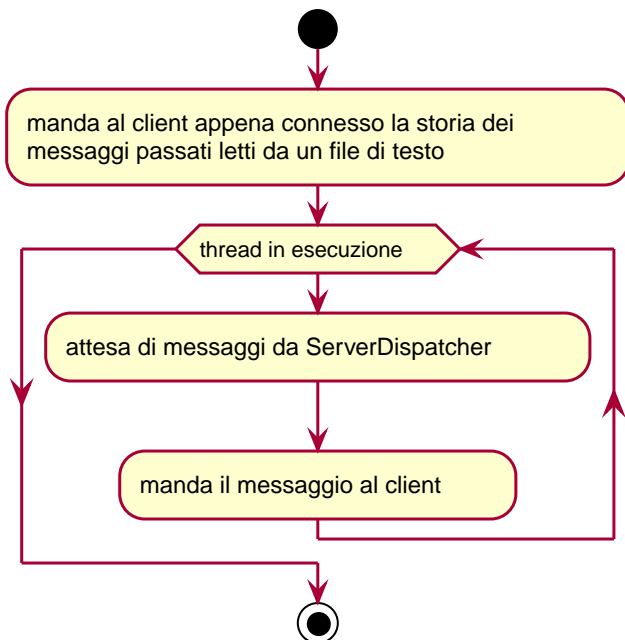


Figure 7. ClientSender

2.3.2. Lato client

- Il client fa una richiesta di connessione.
- Dopo aver ottenuto l'accettazione dal server, il client crea un thread a parte che si occupa di mandare i messaggi al server, mentre il main-thread fungerà da listener.

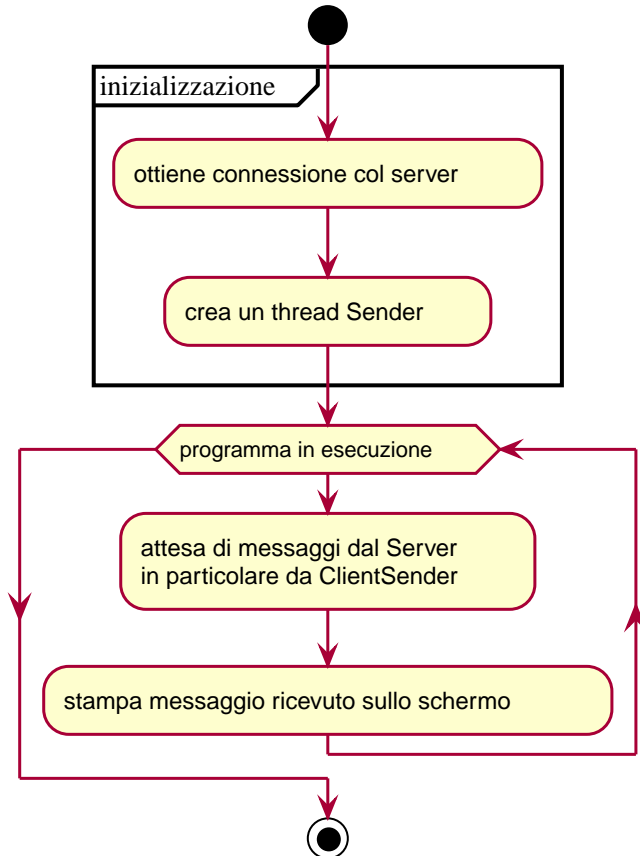


Figure 8. Client

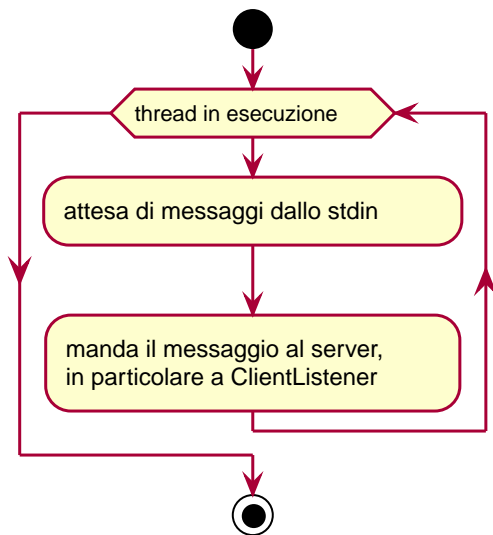


Figure 9. Sender

2.4. Codice Sorgente completo

2.4.1. Client.java

```

import java.io.*;
import java.net.*;
import java.text.SimpleDateFormat;

public class Client {

    private Socket m_connection;
    private ObjectInputStream m_fromServer;
    private ObjectOutputStream m_toServer;
    private String m_address;
    private int m_serverPort;

    public static void main(String[] args)
    {
        Client c = null;
        if (args.length == 2) {
            c = new Client(args[0], Integer.parseInt(args[1]));
        } else {
            c = new Client("127.0.0.1", 8080);
        }
        c.talk();
    }

    public Client(String address, int serverPort) {
        m_serverPort = serverPort;
        m_address = address;
        try {
            m_connection = new Socket(m_address, m_serverPort);
            m_fromServer = new ObjectInputStream(m_connection.getInputStream());

```

```

        m_toServer = new ObjectOutputStream(m_connection.getOutputStream());

        System.out.println("Connected to server " + m_connection.getInetAddress
().getHostName() +
        " at " + m_address + ":" + m_serverPort);
    }
    catch(IOException ioe) {
        System.err.println("Can not establish connection to " +
            m_address + ":" + m_serverPort);
        ioe.printStackTrace();
        System.exit(-1);
    }
}

public void talk() {
    Sender sender = new Sender(m_toServer);
    sender.setDaemon(true);
    sender.start();
    try {
        Message message;
        while ((message = (Message)m_fromServer.readObject()) != null) {
            System.out.printf("\b\b%30s%s %s %s\n%30s%s\n\n> ",
                "",
                message.getSenderUsername(),
                message.getSenderAddress(),
                new SimpleDateFormat("yyyy-MM-dd, hh:mm:ss").format
(message.getDate()),
                "",
                message.getPayload());
        }
    } catch (IOException ioe) {
        System.err.println("Connection to server broken.");
        ioe.printStackTrace();
    } catch (ClassNotFoundException e) {

    }
}
}

class Sender extends Thread
{
    private ObjectOutputStream m_toServer;

    public Sender(ObjectOutputStream out)
    {
        m_toServer = out;
    }

    public void run()
    {
        try {
            BufferedReader console = new BufferedReader(new

```

```

InputStreamReader(System.in));
    while (!isInterrupted()) {
        System.out.printf("> ");
        String s = console.readLine();
        Message message = new Message(s);
        m_toServer.writeObject(message);
        m_toServer.flush();
    }
} catch (IOException ioe) {
}
}
}

```

2.4.2. Server.java

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * Multithreaded Chat Server Implementation – Server
 */

public class Server {

    private Socket m_connection;
    private int m_acceptedConnections = 5;
    private int m_listeningPort = 4000;

    public static void main(String[] args) {
        if (args.length == 1) {
            Server s = new Server(Integer.parseInt(args[0]));
        } else {
            Server s = new Server(8080);
        }
    }

    public Server(int listeningPort)
    {
        m_listeningPort = listeningPort;

        try (ServerSocket server = new ServerSocket(m_listeningPort,
m_acceptedConnections)) {
            System.out.println("Server running on port " + m_listeningPort);

            ServerDispatcher serverDispatcher = new ServerDispatcher();
            serverDispatcher.start();

```

```

        while (true) {
            try {

                m_connection = server.accept();

                ClientInfo clientInfo = new ClientInfo();
                clientInfo.setSocket(m_connection);

                ClientSender clientSender = new ClientSender(clientInfo,
serverDispatcher);
                ClientListener clientListener = new ClientListener(clientInfo,
serverDispatcher);

                clientInfo.setListener(clientListener);
                clientInfo.setSender(clientSender);

                clientListener.start();
                clientSender.start();

                serverDispatcher.addClient(clientInfo);

            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
        System.exit(-1);
    }
}
}

```

2.4.3. ServerDispatcher.java

```

/**
    Chat Server

    * ServerDispatcher class is purposed to listen for messages received
    * from clients and to dispatch them to all the clients connected to the
    * chat server.
    */

import java.io.IOException;
import java.util.Vector;

public class ServerDispatcher extends Thread
{

```

```

private Vector<Message> m_messageQueue = new Vector<Message>();
private Vector<ClientInfo> m_clients = new Vector<ClientInfo>();

public synchronized void addClient(ClientInfo clientInfo)
{
    m_clients.add(clientInfo);
}

public synchronized void deleteClient(ClientInfo clientInfo)
{
    int index = m_clients.indexOf(clientInfo);
    if (index != -1) {
        m_clients.removeElementAt(index);
    }
}

/**
 * Adds given message to the dispatcher's message queue and notifies this
 * thread to wake up the message queue reader (getNextMessageFromQueue method).
 * dispatchMessage method is called by other threads (ClientListener) when
 * a message is arrived.
 */
public synchronized void dispatchMessage(Message message)
{
    m_messageQueue.add(message);
    notify();
}

/**
 * @return and deletes the next message from the message queue. If there is no
 * messages in the queue, falls in sleep until notified by dispatchMessage method.
 */
private synchronized Message getNextMessageFromQueue()
    throws InterruptedException
{
    {
        while (m_messageQueue.size() == 0)
            wait();
        Message message = m_messageQueue.get(0);
        m_messageQueue.removeElementAt(0);
        return message;
    }
}

/**
 * Sends given message to all clients in the client list. Actually the
 * message is added to the client sender thread's message queue and this
 * client sender thread is notified.
 */
private synchronized void sendMessageToAllClients(Message message)
{
    for (int i = 0; i < m_clients.size(); i++) {
        m_clients.get(i).getSender().sendMessage(message);
    }
}

```

```

    }
}

public void run()
{
    try {
        while (true) {
            Message message = getNextMessageFromQueue();
            MessageHistory history = new MessageHistory("history.txt");
            history.addMessage(message);
            sendMessageToAllClients(message);
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

2.4.4. ClientSender.java

```

import java.io.IOException;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.Vector;

/**
 * Chat Server
 *
 * Sends messages to the client. Messages are stored in a message queue. When
 * the queue is empty, ClientSender falls in sleep until a new message is
 * arrived in the queue. When the queue is not empty, ClientSender sends the
 * messages from the queue to the client socket.
 */

public class ClientSender extends Thread
{
    private Vector<Message> m_messageQueue = new Vector<Message>();

    private ServerDispatcher m_serverDispatcher;
    private ClientInfo m_clientInfo;
    private ObjectOutputStream m_output;

    public ClientSender(ClientInfo clientInfo, ServerDispatcher serverDispatcher)
        throws IOException
    {
        m_clientInfo = clientInfo;
        m_serverDispatcher = serverDispatcher;
    }
}

```



```

        Socket socket = clientInfo.getSocket();
        m_output = new ObjectOutputStream(socket.getOutputStream());
        m_output.flush();
    }

    /**
     * Adds given message to the message queue and notifies this thread
     * (actually getNextMessageFromQueue method) that a message is arrived.
     * sendMessage is called by other threads (ServeDispatcher).
     */
    public synchronized void sendMessage(Message message)
    {
        m_messageQueue.add(message);
        notify();
    }

    /**
     * @return and deletes the next message from the message queue. If the queue
     * is empty, falls in sleep until notified for message arrival by sendMessage
     * method.
     */
    private synchronized Message getNextMessageFromQueue() throws InterruptedException
    {
        while (m_messageQueue.size() == 0)
            wait();
        Message message = m_messageQueue.get(0);
        m_messageQueue.removeElementAt(0);
        return message;
    }

    /**
     * Sends given message to the client's socket.
     */
    private void sendMessageToClient(Message message) throws IOException
    {
        m_output.writeObject(message);
        m_output.flush();
    }

    /**
     * Until interrupted, reads messages from the message queue
     * and sends them to the client's socket.
     */
    public void run()
    {
        try {
            // first send all the history
            for (Message m : new MessageHistory("history.txt").getHistory(4)) {
                sendMessageToClient(m);
            }
        }
    }

```

```

        // wait for new messages
        while (!isInterrupted()) {
            Message message = getNextMessageFromQueue();
            sendMessageToClient(message);
        }
    } catch (IOException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    m_clientInfo.getListener().interrupt();
    m_serverDispatcher.deleteClient(m_clientInfo);
}
}

```

2.4.5. ClientListener.java

```

import java.io.IOException;
import java.io.InputStreamReader;
import java.io.ObjectInputStream;
import java.net.Socket;

/**
 * Chat Server -
 *
 * ClientListener class is purposed to listen for client messages and
 * to forward them to ServerDispatcher.
 */

public class ClientListener extends Thread
{
    private ServerDispatcher m_serverDispatcher;
    private ClientInfo m_clientInfo;
    private ObjectInputStream m_input;

    public ClientListener(ClientInfo clientInfo, ServerDispatcher serverDispatcher)
        throws IOException
    {
        m_clientInfo = clientInfo;
        m_serverDispatcher = serverDispatcher;
        Socket socket = clientInfo.getSocket();

        m_input = new ObjectInputStream(socket.getInputStream());
    }

    /**

```

```

    * Until interrupted, reads messages from the client socket, forwards them
    * to the server dispatcher's queue and notifies the server dispatcher.
    */
    public void run()
    {
        try {
            while (!isInterrupted()) {
                Message message = (Message) m_input.readObject();
                message.setSenderInfo(m_clientInfo);
                if (message == null)
                    break;
                m_serverDispatcher.dispatchMessage(message);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        // Communication is broken. Interrupt both listener and sender threads
        m_clientInfo.getSender().interrupt();
        m_serverDispatcher.deleteClient(m_clientInfo);
    }
}

```

2.4.6. Message.java

```

import java.io.Serializable;
import java.text.SimpleDateFormat;
import java.util.Date;

public class Message implements Serializable {

    private String m_senderUsername = null;
    private String m_senderAddress = null;
    private Date m_date = null;
    private String m_payload = null;

    Message() {

    }

    Message(String senderUsername, String senderAddress, Date date, String payload)
    {
        m_senderUsername = senderUsername;
        m_senderAddress = senderAddress;
        m_date = date;
        m_payload = payload;
    }
}

```

```

Message(ClientInfo sender, String payload)
{
    m_senderUsername = sender.getUsername();
    m_senderAddress = sender.getAddress();
    m_date = new Date();
    m_payload = payload;
}

Message(String payload)
{
    m_payload = payload;
    m_date = new Date();
}

public void setSenderInfo(ClientInfo sender) {
    m_senderUsername = sender.getUsername();
    m_senderAddress = sender.getAddress();
}

public String getSenderUsername() {
    return m_senderUsername;
}

public String getSenderAddress() {
    return m_senderAddress;
}

public Date getDate() {
    return m_date;
}

public String getPayload() {
    return m_payload;
}

public void setSenderUsername(String senderUsername) {
    m_senderUsername = senderUsername;
}

public void setSenderAddress(String senderAddress) {
    m_senderAddress = senderAddress;
}

public void setDate(Date date) {
    m_date = date;
}

public void setPayload(String payload) {
    m_payload = payload;
}

```

```

public String toString()
{
    return String.format("%s\\;%s\\;%s\\;%s",
        this.getSenderUsername(),
        this.getSenderAddress(),
        new SimpleDateFormat("yyyy-MM-dd, hh:mm:ss").format(this.
getDate()),
        this.getPayload());
}
}

```

2.4.7. MessageHistory.java

```

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;
import java.util.StringTokenizer;
import java.util.Vector;

public class MessageHistory {
    private String fileName_;

    public MessageHistory(String fileName)
    {
        fileName_ = fileName;
    }

    public void addMessage(Message message) throws IOException
    {
        boolean append = true;
        try (PrintWriter fileOut = new PrintWriter(new FileWriter(fileName_, append)))
        {
            fileOut.println(message);
        }
    }

    public Vector<Message> getHistory() throws IOException
    {
        return getHistory(0);
    }
}

```

```

public Vector<Message> getHistory(int dayFilter) throws IOException
{
    Vector<Message> messages = new Vector<Message>();
    File file = new File(fileName_);
    file.createNewFile();

    // calculate the filter
    Date filter = null;
    if (dayFilter != 0) {
        Calendar calendar = Calendar.getInstance();
        calendar.setTime(new Date());
        calendar.add(Calendar.DATE, -dayFilter);
        filter = calendar.getTime();
    }

    try (BufferedReader fileIn = new BufferedReader(new FileReader(fileName_))) {
        StringTokenizer tokenizer;
        for (String m; (m = fileIn.readLine()) != null;) {
            tokenizer = new StringTokenizer(m, "\\;");
            Message message = new Message();
            message.setSenderUsername(tokenizer.nextToken());
            message.setSenderAddress(tokenizer.nextToken());
            String dateString = tokenizer.nextToken();
            Date date = new SimpleDateFormat("yyyy-MM-dd, hh:mm:ss").
parse(dateString);
            if (filter != null && date.before(filter)) {
                continue;
            }
            message.setDate(date);
            message.setPayload(tokenizer.nextToken());

            messages.addElement(message);
        }
    } catch (ParseException e) {
        e.printStackTrace();
    }

    return messages;
}

```

2.4.8. ClientInfo.java

```

/**
 * Chat Server
 * ClientInfo class contains information about a client, connected to the server.
 */

import java.net.Socket;

```

```

public class ClientInfo
{
    private Socket m_socket = null;
    private ClientListener m_listener = null;
    private ClientSender m_sender = null;
    private String m_username = null;
    private String m_address = null;

    //ClientInfo(Socket socket, ClientListener listener, ClientSender sender, String
    username) {
        //m_socket = socket;
        //m_listener = listener;
        //m_sender = sender;
        //m_username = username;
    //}

    public void setSender(ClientSender sender) {
        m_sender = sender;
    }

    public void setListener(ClientListener listener) {
        m_listener = listener;
    }

    public void setSocket(Socket socket) {
        m_socket = socket;
        setUsername(m_socket.getInetAddress().getHostName());
        setAddress(m_socket.getInetAddress().getHostAddress());
    }

    public void setUsername(String username) {
        m_username = username;
    }

    public void setAddress(String address) {
        m_address = address;
    }

    public ClientSender getSender() {
        return m_sender;
    }

    public ClientListener getListener() {
        return m_listener;
    }

    public Socket getSocket() {
        return m_socket;
    }
}

```

```
public String getUsername() {  
    return m_username;  
}  
  
public String getAddress() {  
    return m_address;  
}  
}
```


Chapter 3. Conclusione

Questa esperienza è stata senza dubbio molto utile per i singoli alunni, dato che spesso nell'ambito scolastico ci si limita a scrivere piccoli programmi (dove per piccoli si intende che non superano nemmeno le cento righe di codice) avendo come unico scopo quello di imparare una determinata sintassi.

Diversamente invece, nella realizzazione di questo progetto, si ha potuto provare un'esperienza che si avvicina di più alla programmazione professionale.

Si è infatti partiti dalla lettura di un codice sorgente scritto da terze parti, finendo col aggiungere nuove funzionalità sulla base del codice fornito.

E questo è certamente significativo dal punto di vista formativo, visto che nella programmazione la manutenzione occupa molto più tempo di quello usato per l'implementazione. Imparare sin da subito a leggere programmi scritti da altri è fondamentale anche se spesso trascurato.

Nello sviluppo di "Chatline" invece abbiamo avuto la fortuna di sviluppare queste abilità, che in un futuro prossimo, per chi deciderà di continuare con la programmazione, sarà il pane quotidiano.