# CSEE W4119 Programming Assignment I

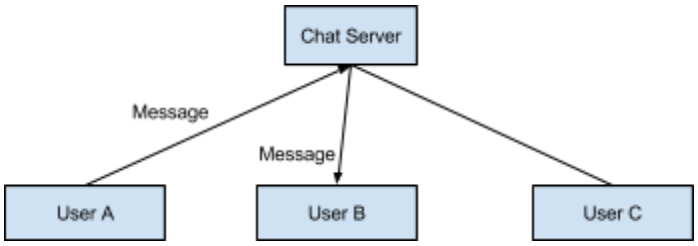
**Due:** Wednesday, 11th March, 2015

Prof. Augustin Chaintreau, Pooja Shah, Olalekan Afuye, Yelin Hong, Jen-Chieh Huang, Nivvedan Senthamil Selvan, Chenze Zhao

---

## Description

The brilliant W4119 class last year implemented a chat room service using a client-server architecture. This legacy architecture maintains a permanent link between the server and the client until the user decides to leave the server. Everything seems to work pretty well. The following diagram shows the concept of the legacy architecture when user A sends a message to user B. The line connecting the users and servers denotes persistent TCP links, and the arrows indicates the directions which the message is sent.




However, many users started to complain about the service, and demanded an upgrade of the application because of the following reasons.

- A permanent connection makes the server unscalable and limits the number of users serviced by a single server. Users can not login to the server once the maximum connection capacity is reached.
- Besides, more and more users are running the chat application using wireless network. The unstable wireless link results in a disaster of the legacy chat room architecture as the user is disconnected each time the wireless signal is temporarily unavailable. (Note: it would be challenging if you'd like to implement fully automatic reconnection. The point here is to work out a method that the client does not have a consistent link to the server.)
- Some users would also like to have some privacy. They want to chat with their friends directly without all their conversation being routed via the chat server. A peer-to-peer chat client seems to be a good solution for this scenario.

Thus, the TAs turn to the even more brilliant 2015 class for help. The idea is to develop a chat application with a message center server, and the key criteria of the new design is that ***no permanent connection*** is maintained between the server and client. Bear in mind that the states of each party will need to be maintained locally, and synchronized using the network connection when needed.

---

# Technical Specifications

The assignment includes 2 major modules, the message center and the chat client. It is important to note that at no point in time should there be a permanent TCP connection from the client to the message centre or vice versa.

## 1. Message Center

The message center, also known as the server, has the following responsibilities -

a. **User Authentication** - Each user logs into the message centre with a username and a password. On entering invalid credentials, the user is prompted to retry. After 3 consecutive failed attempts, the user is blocked for a duration of 60 seconds (this should be configurable), and cannot login during this duration (even from another IP). While a user is online, if someone uses the same username/password to log in from another IP, current user will be notified and automatically logged out.

b. **User Message Forwarding** - Forward each chat message to the correct recipient.

c. **Timeout** - The message centre should have a configurable timeout value. If it does not receive a 'LIVE' signal from a user within the timeout, it considers the user to be disconnected.

d. **Blacklisting** - Allow a user A to block / unblock any other user B. If user A has blocked user B, B can no longer send chat messages to A i.e. the message centre should intercept such messages and inform B that the message cannot be forwarded. Blocked users also do not get presence notifications i.e. B will not be informed each time A logs in or logs out.

e. **Presence Broadcasts** - Notify the presence of other users logged into the chat room i.e. send a broadcast notification to all online users when a user logs in / logs out.

f. **Offline Messaging** - When the recipient of a message is not logged in, the message will be saved by the message centre. When the recipient logs in next, the message centre will display all the unread messages stored.

## 2. Chat Client

The chat client has the following responsibilities -

a. **Authentication** - Provide a login prompt to enable the user to authenticate with the message center.
b. **Chat** -
    i. Allow the user to send a message to any other user.
    ii. Display messages sent by other users.
c. **Notifications** - Display presence notifications sent by the message center.
d. **Find users online** - Obtain a list of all the users currently online.
e. **Heartbeat** - Every 30 seconds (this duration should be configurable), the client should send a 'LIVE' signal to the message centre to indicate that the user is still logged in.
f. **User Interface (BONUS)** - A simple graphical user interface can be implemented for extra credit.

## Peer-to-peer chat

The aim of this feature is to implement a chat service in which the message centre has minimal intervention once a conversation has been established. It can be developed on top of the basic chat room architecture.
The basic requirements of this feature are -

g. Any user A wanting to private chat with user B will first request the message centre for user B's IP address.
    i. If B is online, the message centre will provide B's IP address and PORT, else it notifies A that B is offline.
    ii. If B is offline, the chat client should report that the request failed.
h. On obtaining B's IP address, user A can establish a connection directly with user B using said address.
i. From this point onwards, A can send private messages directly to B using the "private" command. Such private messages would be sent directly to the IP address without going through the server.

**Bonus Part:**
The following are suggested for bonus points. Feel free to discuss other ideas with the TAs. Ideas not in this list need to be discussed with the TAs to confirm them to be worthy of bonus points.

1. **P2P Privacy and Consent**
    1.1 When A requests for B's IP address, the message centre should notify B that A wants to talk it. If B agrees to the conversation, the server should provide A with B's IP address. Else, A cannot initiate a conversation with B.

1.2 When A requests for B's IP address, the message centre should check B's blacklist preferences. If B's blacklist includes A, the message centre should not provide B's IP address to A.

2. **Guaranteed Message Delivery**
   By the problem definition, the server maintains a database of the clients and their IP addresses. Sometimes, this database might be outdated and the client might actually be offline. For e.g. if the client has been disconnected abruptly, and the message centre is still waiting for timeout. In this case, the sending client will attempt to connect to the last known IP of the receiving client and the connection will fail. Such failure should be handled and the sender can re-contact the server to leave an offline message. Also, if the receiving client logs in with a new IP, the sending client should also be aware of this and not sending message to the old IP any more.

---

## Authentication requirements

- You should use the file "credentials.txt" to import all the username/password combinations. The first word in each line is the username and the second is the corresponding password. Note that the content of this file could be changed during the testing of your assignment.

## Commands

The following commands need to be implemented in accordance with the technical specifications above. For the following, assume the commands were run by user *A*.

| **Command** | **Description** |
|---|---|
| message <user> <message> | Sends <message> to <user> **through** the server |
| broadcast <message> | Sends <message> to all the online users (except those users who have blocked A) |
| online | This should print the list of users currently online. A user is defined to be online if:<br>❖ It has successfully logged in at least once since the server was invoked AND<br>❖ The last command from it wasn't <*logout*> AND<br>❖ The hearbeat timeout is yet to elapse |

| | |
|---|---|
| block <user> | This blocks <user> from being able to send a message to A through the server. <user> should get a notification about being blocked when it attempts to message A. The server should not provide *A*'s ip address if <user> enters the <*getaddress A*> command. Note that if <user> already has the IP address of A, then the command <*private A"* will still be successful. |
| unblock <user> | This reverses the block command. Messages and getadress requests from <user>can now be serviced. |
| logout | This notifies the server that user *A* is no longer online. Subsequent messages to *A* should be saved and delivered whenever *A* logs in again. |
| getaddress <user> | This returns the IP address and PORT of <user>. The address should be retrieved from the server. This command is useful when user *A* wants to begin a private conversation with <user>. Subsequent *"private <user>"* commands would be internally sent to this ip address and PORT. If <user> had previously blocked *A*, then the server should not provide this information. |
| private <user> <message> | Sends <message> to <user> directly i.e. **without routing through** the server. If <user> is no longer available at the original address supplied by <*getaddress user*>, the failure should be indicated to *A* so *A* can choose to send an offline message through the server. |

## Sample Run

**Case 1 : Successful login**
**Terminal 1**

```
>make
>java Server 4009
```

**Terminal 2**

```
>make
>java Client 10.11.12.13 4009          //Server's IP address / port
>Username: foo
>Password: bar
>Welcome to simple chat server!
>                                      //Prompt to enter commands
```

**Case 2 : Unsuccessful login**
**Terminal 1**

```
>make
>java Server 4009
```

**Terminal 2**

```
>make
>java Client 10.11.12.13 4009          //Server's IP address / port
>Username: foo
>Password: bar1
>Invalid Password. Please try again
>Password: bar2
>Invalid Password. Please try again
>Password: bar3
>Invalid Password. Your account has been blocked. Please try again
after sometime.
```

The user should now be blocked for a configurable BLOCK_TIME (e.g. 60 seconds)
The terminal should shut down at this point.

**Terminal 2 (reopened before BLOCK_TIME seconds are over)**

```
>java Client 10.11.12.13 4009
>Username: foo
>Password: bar
>Due to multiple login failures, your account has been blocked.
Please try again after sometime.
```

**Terminal 2 (reopened after BLOCK_TIME seconds are over)**

```
>java Client 10.11.12.13 4009
>Username: foo
>Password: bar
>Welcome to simple chat server!
>
```

(NOTE : See the appendix for more examples of command behavior).

---

# Development Environment Specifications

## Development

You can choose between Java, Python and C/C++ to write your program. Whichever you choose, your program **must compile and run on the CLIC machines**. For your reference, the versions of the compilers/interpreters on the CLIC machines include:

- Java - 1.6.
- Python 2.7.3
- C/C++: gcc/g++ 4.6.3

**NOTE - If your code doesn't compile, we will call you to have a look at it and fix it. However, it will result in a deduction of 20% of your total points.**

### Deliverables

Your program will be submitted via courseworks. Please submit a zip file using the format **<UNI>_<Programming Language>.zip** (e.g. cn1111_python.zip) to the Programming Assignment 1 folder. Make sure you include all of the following files in your zip file:

- README.txt: Your readme file should include but not limited to the following parts
    - A general description of your programming design and data structure
    - Explanation of your source code (make sure your code is well commented and readable or you will be taken 5-10 points. To avoid such an undesirable scenario, tools like astyle[1] may be helpful.)
    - Detailed instructions on how to run/compile your source code
    - Sample commands to run your program
    - A short introduction of your additional features and sample test cases (**Notice: discuss your design of additional features to one of TAs during office hour or post a private message on Piazza before you actually implement them, some features may not be considered worthy of the bonus points**)
- Makefile: if you are not familiar with makefile, you can read through the tutorial: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/ This article is also helpful: http://www.devin.com/cruft/javamakefile.html
- Source code and other files you think are important

## Grading Scheme:

| **Functionality** | **Points** |
|---|---|
| Basic Client/Server Model | 80 |
| - User authentication | 10 |
| - Message exchange | 10 |
| - Multiple clients support | 10 |
| - Heartbeat | 10 |
| - Blacklist | 10 |
| - Offline messaging | 10 |

[1] http://astyle.sourceforge.net/

| | |
|---|---|
| - Broadcast | 5 |
| - Display current users | 5 |
| - Logout | 5 |
| - Graceful exit using control + c | 5 |
| | |
| Basic P2P Model | 20 |
| - Obtain online user's IP address | 5 |
| - Offline report | 2 |
| - P2P message exchange | 13 |
| | |
| Advanced P2P Features OR other personalized features (Bonus) | 20 |
| - P2P privacy and consent | 10 |
| - Guaranteed message delivery | 10 |

**Note:**

1. 50% of the total points will be deducted if you use a permanent connection.
2. If we find detect any plagiarism in your assignment, you will get 0 points in this assignment. In addition, you will be referred to the Dean of Students without any exception.

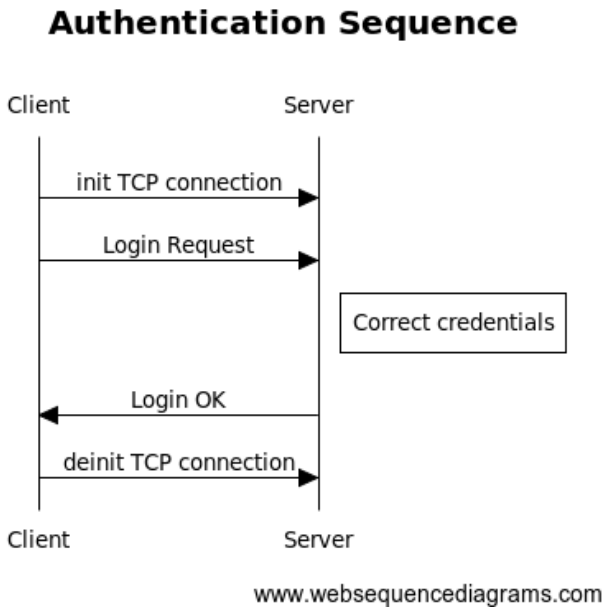
**Bonus:**

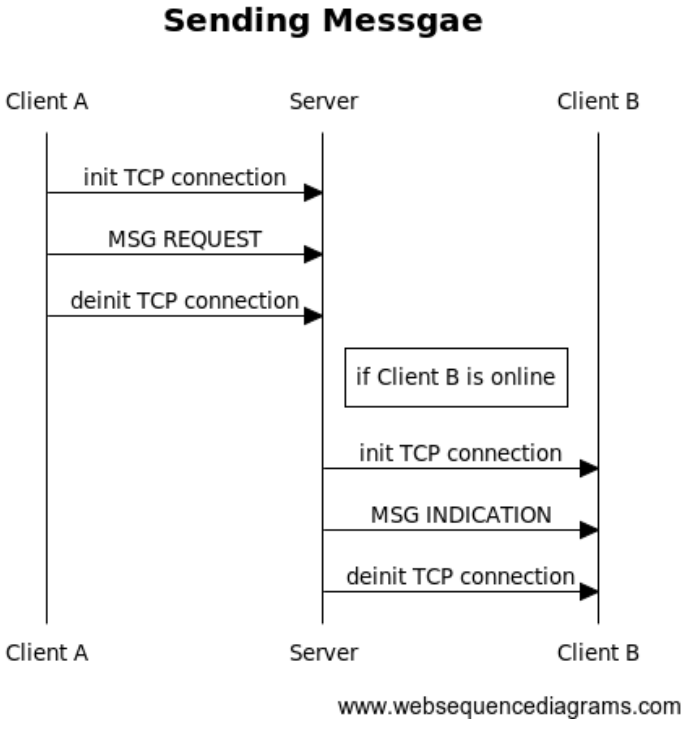
1. Advanced P2P features - 20 points OR Any 2 other features for a maximum of 20 points

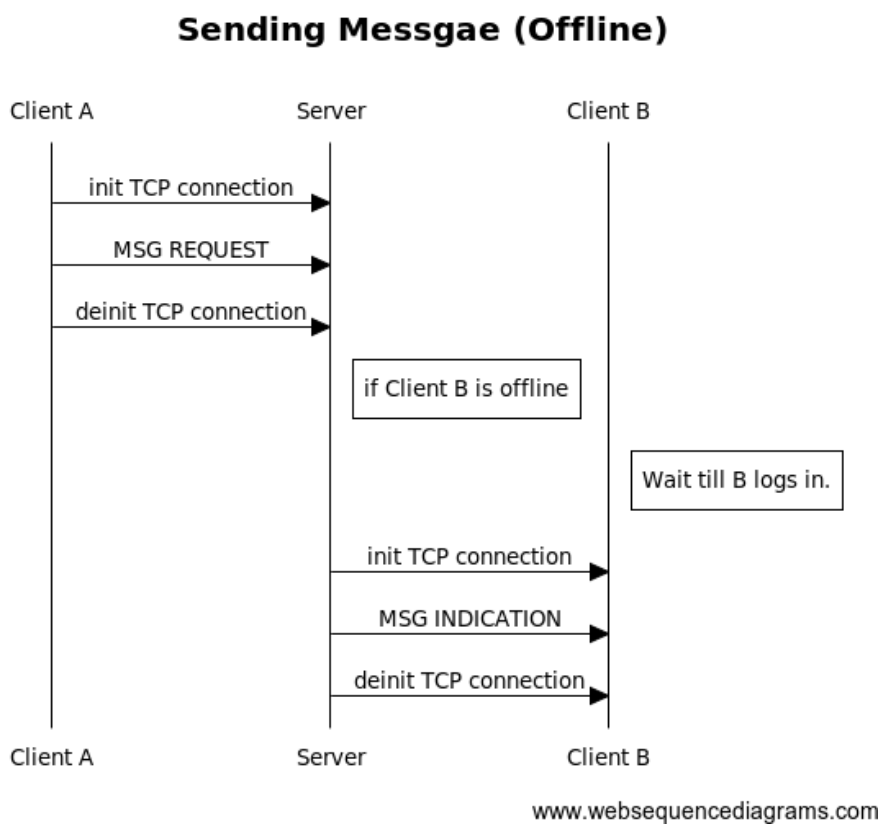
Maximum points for this assignment - 120 points.
There are a lot of ways to get extra credit, so make sure to use this opportunity!

---

# Reference Message Sequence Charts (MSC)






---

# Appendix A: Command Usage

Consider a scenario where 3 users, **A, B and C** are currently logged in.

The below command should print the following results on A's terminal.

1. **online**

   **online** lists all the users who are currently online (except for the logged in user). So the output should be -

| A's Terminal | B's Terminal | C's Terminal |
|---|---|---|
| **>online**<br>**>B**<br>**>C** | | |

2. **message**

| A's Terminal | B's Terminal | C's Terminal |
|---|---|---|
| **>message B Hi** | | |
| | **>A: Hi** | |

3. **broadcast - broadcast sends the message to all online users (who haven't blocked A)**

| A's Terminal | B's Terminal | C's Terminal |
|---|---|---|
| **>broadcast Hi there** | | |
| | **>A: Hi there** | **>A: Hi there** |

4. **block**

| A's Terminal | B's Terminal | C's Terminal |
|---|---|---|
| **>block B**<br>**>User B has been blocked** | | |
| | **>message A Hi**<br>**>Your message could not be delivered as the recipient has blocked you** | |
| | **>broadcast Hello**<br>**>Your message could not be delivered to some recipients** | |
| | | **>B: Hello** |

5. **unblock**
   Consider that A has blocked B

| A's Terminal | B's Terminal | C's terminal |
|---|---|---|
| **>unblock B**<br>**>User B is unblocked** | | |

| | | |
|---|---|---|
| | **>message A Hi** | |
| **>B: Hi** | | |
| | **>broadcast Hello** | |
| **>B: Hello** | | **B: Hello** |

## Appendix B: In Case You Need a Clue

There is some information which may help you to know more about the assignment. In the world of networks, protocol is defined as "a system of digital rules for data exchange within or between computers."[2] In other words, a protocol is a method which defines how the participants in the network communicate with one another. It is easy to find some analogy in our daily lives. Let's say, when you meet a new friend, you will need to do handshake first, introduce yourself, and start to talk (most of the time) . After the conversation is over, you may need to say "see you next time" or "bye" to end the session. The whole process and the messages exchanged (handshakes, introduction, saying goodbye) is an example what a protocol looks like.

Another good example of protocol is HTTP[3]. When a user enters the URL in the browser, the browser will send a HTTP request to the server like this:

```
GET /index.html HTTP/1.1
Host: www.example.com
```

This message indicates that this is a GET message using HTTP v1.1 protocol, and the client wants the contents of index.html from www.example.com. Upon receiving this request, the server may return the following information.

[2] http://en.wikipedia.org/wiki/Communications_protocol
[3] http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

**HTTP/1.1 200 OK**
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Content-Type: text/html; charset=UTF-8
**Content-Length: 131**
Accept-Ranges: bytes
Connection: close

<html>
<head>
<title>An Example Page</title>
</head>
<body>
Hello World, this is a very simple HTML document.
</body>
</html>

The returned status code, 200, means that the request is successfully handled, and the contents of the HTML page is attached in the end of the message.

In this assignment, you may need to design your own protocol and process these messages. The intuitive way to specify the message exchanged is to set up the **fixed fields** in the message. For example, your messages may look like the following. It is an example from RTP.

**RTP packet header**

| Bit offset[b] | 0-1 | 2 | 3 | 4-7 | 8 | 9-15 | 16-31 |
|---|---|---|---|---|---|---|---|
| 0 | Version | P | X | CC | M | PT | Sequence Number |
| 32 | Timestamp | | | | | | |
| 64 | SSRC identifier | | | | | | |
| 96 | CSRC identifiers ... | | | | | | |
| 96+32×CC | Profile-specific extension header ID | | | | | | Extension header length |
| 128+32×CC | Extension header ... | | | | | | |

One common technique to design a protocol is called **TLV (Tag-Length-Value)**[4]. You may find it very useful when you need to expand or upgrade your protocol. In this design paradigm, the messages can be specified as the following:

| Header Length | Number of Options | Tag 0 | Value Length of Tag 0 | Value (Data) of Tag 0 | ... |
|---|---|---|---|---|---|

[4] http://en.wikipedia.org/wiki/Type-length-value

With this technique, you can extend your protocol if needed without renovating your codes completely.

---

## Academic Honesty Policy

You are permitted and encouraged to help each other through Piazza's web board. This only means that you can discuss and understand concepts learnt in class. However, you may NOT share source code or hard copies of source code. Refrain from sharing any material that could cause your source code to APPEAR TO BE similar to another student's source code enrolled in this or previous years. Refrain from getting any code off the Internet. Cheating will be dealt with severely. Cheaters will be penalized. Source code should be yours and yours only. Do not cheat.

---