CS2003 Practical Net2 – Text Messenger

Saleem Bhatti

31 October 2018

Deadline: 14 November 2018 (please see MMS)

Description

Your task is to build a simple text messaging system that will work on the Linux workstations in the School Labs. The system will be a *single program* which can:

- Detect when other Text Messenger systems are online and offline, and identify the user running the program.
- Allow the user to send text messages to any other users that are detected to be online.
- Receive text messages from any users that are detected to be online.

Your program *must* use the protocol specification given in Section 5 of this document.

Your program should be written completely in Java 8 SE (the default version of Java on the Linux workstations), using `javac` to compile, and `java` to run.

---

1 Basic Requirements

Using the files given (see Section 6, below) you are required to change and extend the code as necessary to perform the following tasks:

(B1) By default, when your program is running, it should indicate that you are online, by transmitting a UDP multicast *beacon* every 5 seconds. The beacon should also contain information about your server port (the FQDN and port number), which is used by other users to send text messages to your program. If your program stops sending the beacons (e.g. if the program is terminated), then you will be considered to be offline.

(B2) Detect when other users come online, by receiving a multicast beacon from their text messenger program. The is the complementary requirement to (B1). The GUI should be updated dynamically, as users come online or go offline, listing the currently detected users.

(B3) Allow the user to enter text messages via the GUI for transmission to other users, if and only if those other users are online. You are given a 'skeleton' GUI, to which you should add your own code. All text message transmissions must use TCP.

(B4) Allow text messages from other users to be received by your application, if and only if they have been detected as being online. Display those received messages in the GUI, including the timestamp and the sender's user name. The is the complementary requirement to (B3).

(B5) Allow the user to send a message to another user from the text entry box of the GUI by typing:

```
<username> : <text>
```

where `<username>` is a remote user who is currently online, and `<text>` is the text to send. The program should check if the `<username>` matches a user who is online, open a TCP connection to that user's program, send the `<text>` as a message formatted as below, and as explained in (Section 5):

```
[<timestamp>][<username>][text][<text>]
```

The sender then closes the TCP connection immediately.

(B6) For B1 to B5, and for any messages transmitted and received, the protocol specification in Section 5 of this document *must* be followed.

You should be able to complete the *Basic Requirements* by examining carefully the example code for week 03 and week 07.

2 Additional Requirements

Please make sure that you have a working submission fulfilling the Basic Requirements before you attempt any of the Additional Requirements listed below.

Please read the protocol specification in Section 5 before reading this section.

Below are some suggestions for additional functionality that you can choose to implement in your submission. (Difficulty is indicated in brackets.)

(A1) In your report, discuss the security and privacy issues with your application, especially with respect to detection of users, and sending and receiving of messages. *(Easy.)*

(A2) Add a simple control in the GUI so that while the application is running, the user can take themselves offline, while still monitoring who else is online. *(Easy.)*

(A3) Allow the server program to record successful text message exchanges on a per user basis, as a single, continuous and correct timeline, and allow this to be viewed via the GUI. *(Medium.)*

(A4) In its basic form, the GUI has a single Frame for text entry, and display of messages. So, if multiple conversations are in progress, all the messages are listed in a single TextArea. Change the program to allow the a separate Frame for a particular user, with its own text entry box, and display of messages. *(Medium.)*

(A5) Show interoperation of your program with at least two other class members. This means that other class members are running their own code, and you can detect each other's programs are online, as well as send each other text messages. (*Easy* for *Basic Requirements* only, ranging to *Hard* if *Additional Requirements* A6 / A7 / A8 are attempted.)

(A6) The basic operation of message transmission involves setting up a TCP connection to the remote sending a single message, and then terminating the TCP connection. This is inefficient if multiple messages need to be exchanged for a conversation. Modify your program so a handshake is used to signal the beginning and end of a conversation. To start a conversation:

1. The sender, user A, immediately after setting up a TCP connection and before sending the actual message text, transmits a `hello` message:

   ```
   [<timestamp>][<username>][hello]
   ```

2. When the receiver, user B, accepts a TCP connection, and detects a `hello` message, user B should respond with a `hello` message, to be received by user A. Then, the conversation has started and actual messages can be exchanged.
3. When either user A or user B wants to terminate the conversation, their program should send a `bye` message:

   ```
   [<timestamp>][<username>][bye]
   ```

   and wait for a `bye`, after which it should close the TCP connection.

4. If a `bye` is received, then a `bye` should be sent in response, and the TCP connection should be closed.

This could involve changes to the way a user interacts with the program, e.g. *hello* and *bye* commands need to be typed in the text entry box of the GUI to explicitly 'connect' to users. *(Medium to Hard.)*

(A7) Add a new state for your program, `unavailable`, which is in addition to `online` and `offline`. This should be set for your program via a GUI control. When set to `unavailable`, this indicates that the program is running, but that the user does not want to receive messages at this time. A beacon should be emitted every 5 seconds which has the type value `unavailable`, and should include `fqdn` and `port`, i.e. the beacon would be:

```
[<timestamp>][<username>][unavailable][<fqdn>][<port>]
```

This status should be indicated for the `<username>` in the GUI in receiving programs.

Message delivery attempts can be made for users that are seen as `unavailable`. When a program is the `unavailable` state, and a message delivery attempt is made to it (which may include a `hello` message if A6 is implemented), the program should reply with an `unavailable` message:

```
[<timestamp>][<username>][unavailable]
```

and then terminate the TCP connection (using a `bye` message if A6 is implemented). Such connection attempts should be reported to the respective users in the GUI. *(Hard.)*

(A8) If an attempt is made to send a message to a remote user who is currently offline, allow the option of storing the message locally, with the current value of `<timestamp>` (i.e. the time the message was submitted). When the remote user comes online, the stored stored message(s) should be delivered with the message type:

```
[<timestamp>][<username>][stored_text][<text>]
```

If A6 is implemented, `hello` and `bye` messages should be used. The delivery of stored messages should be indicated appropriately in GUIs at each end of the communication. *(Hard.)*

3 Submission

Your MMS submission should be a single `.zip`, or `.tar.gz` file containing:

(a) Java files. A single directory, called `textmessenger` (which may have subdirectories if required), with all the source code and configuration files needed to run your application. Please do not submit any `.class` files.

Please note the following important items:

- You must not use any external / third-party libraries or source code in your solution.
- Please do not use absolute path-names for any references to files in your submission. Please use relative path-names, and keep all files within the submitted `textmessenger` directory (sub-directories can be used).
- If the marker cannot compile and run the code you have submitted on the Linux workstations in the School Lab, then you will be penalised in marking. It must be possible to compile and run your submission using the default `javac` compiler and default `java` JVM, respectively.

(b) Report. A single PDF file which is a report with the information listed below. In your report, where appropriate, try to concentrate on why you did something (design decisions) rather than just explaining what the code does. In your report, please include:

- A simple guide to running your application, including which file has the `main()` method.
- A summary of the operation of your application indicating the level of completeness with respect to the *Basic Requirements* and *Additional Requirements* listed above.
- Suitable, simple, diagrams, as required, showing the operation of your program, especially if you have designed and implemented any of the *Additional Requirements*.
- An indication of how you tested your application, including screenshots of the web pages showing your application working. If you have designed and implemented any of the *Additional Requirements*, then testing and evidence of that functionality should also be provided.
- For *Additional Requirement* A5, please list the matriculation numbers of the people with whom you performed interoperation tests, and include some text and screenshots in your report to demonstrate your interoperation.

In order to fulfil some of the *Additional Requirements*, you may need to create additional Java files.

4 A note on collaboration

I encourage you to discuss the assignment with other members of the class. This may be especially useful in examining the code that has been given out as your starting point, but also for discussing both the *Basic Requirements* and the *Additional Requirements*. However, the code you develop for your solution should be your own, individual work, and your report should be written by you alone.

For *Additional Requirement* A5, please list in your report the matriculation numbers of the people with whom you performed interoperation tests.

5 Protocol specification

The protocol consists of text messages using ASCII, printable text characters. The various fields of the message are delimited in square bracket characters,'[' and ']'.

5.1 Message header

The *message header* is:

```
[<timestamp>][<username>][<type>]
```

and is used in *all* messages.

- `<timestamp>`: this takes the format 'yyyyMMdd-HHmmss.SSS', where yyyy is the year, MM is the month, dd is the day of the month, HH is the hour of he day (24 hour clock), mm is minutes, ss is seconds, and SSS is milliseconds.
  Hint: Look at the Java class `SimpleDateFormat` as used in *MessageCheckerCommon.java*.
- `<username>`: This must only use the characters 'a-z' and '0-9'. The `<username>` in the message is always that of the sender of the message.
  Hint: Look at the file *Main.java* which uses your username when you login, or uses a command-line argument for the user name. The latter is useful for testing your program by running it with a different user name on another machine.
- `<type>`: The value of `<type>` must only use the characters 'a-z', '0-9', and '_'. Depending on the value of `<type>`, the message will contain other fields.

An example of a header is:

```
[20181001-112113.157][saleem][text]
```

but this would not be a valid message, as there should be other fields present.

5.2 `text` message type

A `text` message has the format:

```
[<timestamp>][<username>][text][<text>]
```

The `<text>` string can have any ASCII, printable characters (see above), except '[' and ']'. An example message is:

```
[20181001-112113.157][saleem][text][Hello World!]
```

5.3 `online` and `offline` message types

The beacons transmitted on UDP multicast are to indicate if the user is online or offline. An `online` message has the format:

```
[<timestamp>][<username>][online][<fqdn>][<port>]
```

An `offline` message has the format:

```
[<timestamp>][<username>][offline][<fqdn>][<port>]
```

The `<fqdn>` can only use the characters 'a-z', 'A-Z', '0-9', '-', '_', and '.'. This is the value of the fully-qualified domain name of the host on which this program is running. This is sufficient for our needs for the School Labs, but real FQDNs can use many other characters (including Unicode). The FQDN should be determined using the Java API and should not be 'hard-coded' into your program.

The `<port>` is an unsigned integer, in decimal form, with a value between 0 and 65,535. It is the port number on which the host is waiting for incoming message delivery requests. For practical purposes, within the School Labs, the lower end of the range is unlikely to be lower than 10,000. In Section 6 are instructions on how to find the values for your port number and your multicast address.

Examples of `online` messages and `offline` messages are:

```
[20181001-112113.157][saleem][online][pc7-42-l.cs.st-andrews.ac.uk][4242]
```

```
[20181001-112120.157][saleem][offline][pc7-42-l.cs.st-andrews.ac.uk][4242]
```

Whilst an `offline` message is defined (and may be used explicitly), if a program does not emit `online` beacons at 5 second intervals, it is considered to be offline.

6 Getting started

Hint: Look at the week 03 and week 07 examples.

On studres, you will find a directory called `textmessenger` in `studres/CS2003/Practicals/Net2/`. This contains code that you should modify and extend directly for your solution:

- *Main.java*: Just a convenience for starting the GUI with a user name.
- *MessageCheckerGUI.java*: The main GUI code.
- *Messages.java*: The Frame used for sending messages and viewing received messages.
- *Notifications.java*: Updates the notifications area of the users and notifications Frame.
- *MessageCheckerCommon.java*: Mainly contains code for demonstration purposes: most of this can be deleted, but you can re-use the class if you like.
- *Users.java*: Checks for new users and updates the users list in the users and notifications Frame.

- *tx_messages.txt*: this is for demonstration purposes only – you will not need it.
- *users_list.txt*: this is for demonstration purposes only – you will not need it.

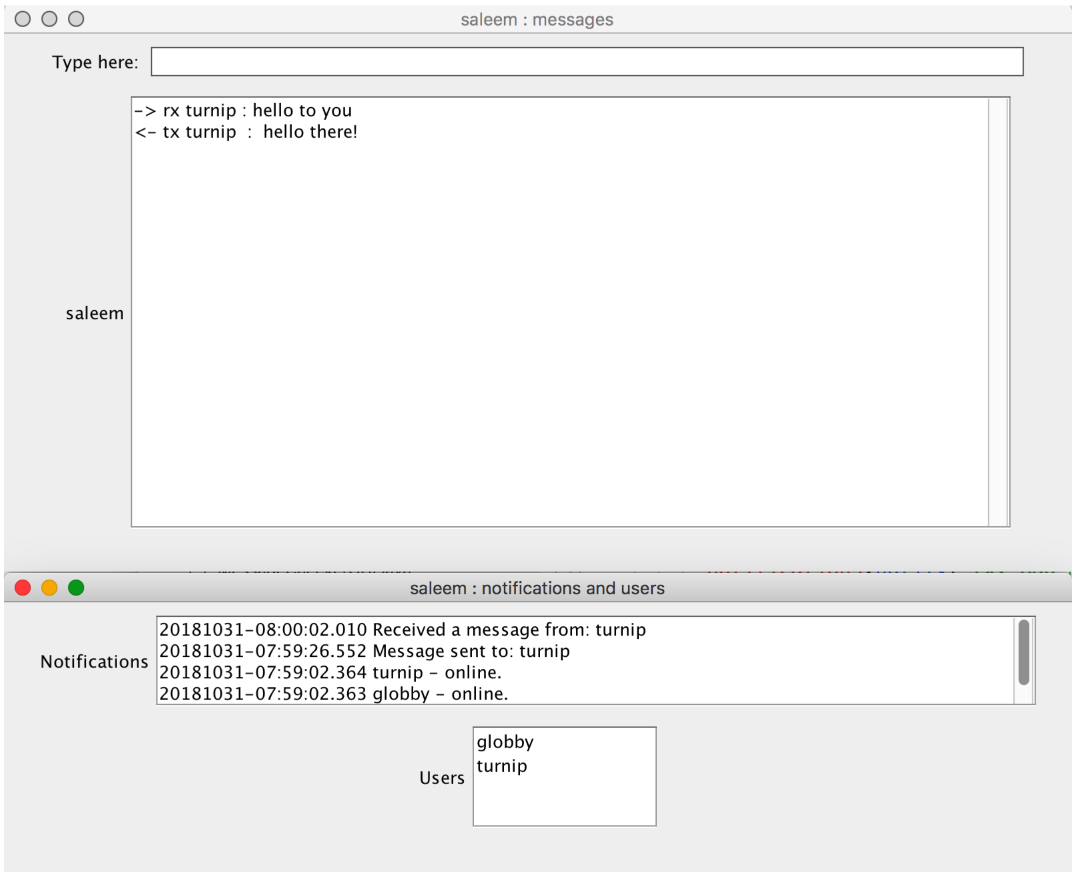
Compile with `javac`, and run:

```
$ java Main
```

to use your own unix user name, or

```
$ java Main test1
```

to use `text1` as a user name.



```
saleem : messages

Type here:

saleem
-> rx turnip : hello to you
<- tx turnip  :  hello there!

saleem : notifications and users

Notifications
20181031-08:00:02.010 Received a message from: turnip
20181031-07:59:26.552 Message sent to: turnip
20181031-07:59:02.364 turnip - online.
20181031-07:59:02.363 globby - online.

Users
globby
turnip
```

Figure 1: Demonstration of the GUI. The files it uses need to be replaced with messages sent and received across the network.

Two windows should appear – see Figure 1. If you type messages in the box marked 'Type here:', messages will appear in the window marked with your user name, and (for demonstration purposes) also be written to the file *tx_messages.txt*.

In the other window, there is a list of users, which (for demonstration purposes), is read from the file *users_list.txt*. Adding extra user names to this file will result in those names appearing in the list of users on-screen.

There is also an area where notification messages appear. If you type:

```
$ echo "turnip : hello to you" > rx_messages.txt
```

the file *rx_messages.txt* is created (for demonstration purposes only), read by the program (and then deleted by the program), and the messages in it appear in the GUI. The use of files in this way is for demonstration purposes only: in completing the *Basic Requirements*, the files will all be replaced with real messages transmitted and received over the network.

You should re-use the GUI code, extending it as required. You can use, modify, and extend any of the files as you wish. You should add other files as you see necessary.

As usual, for you port number, use your unix user id, which you can find using:

```
$ id -u
```

This number will also be used for your multicast address for the beacon messages. To form your multicast address, the number should be added to the lower 16 bits of the IPv4 address value 239.0.0.0. For example, if a unix user id value is 4242, then:

```
4242 DIV 256 = 16
4242 MOD 256 = 146
multicast address = 239.0.16.146
```

Check: $(16 \times 256) + 146 = 4242$.

For *Additional Requirement* A5, interoperation with others in the class, please use multicast address 239.42.42.42 and port 10101.

Marking

Having completed the *Basic Requirements*, to gain a mark above 16, you will need to complete some of the *Additional Requirements* listed above. In your report, highlight where you have attempted any of the *Additional Requirements*. However, please first build a working solution meeting all the *Basic Requirements*, before trying anything more complex.

Note that attempting one or more of the items listed as Additional Requirements does not guarantee you a mark above 16. Your mark will depend on the overall quality of your submission, both the code and the report.

The submission will be marked on the University's common reporting scale according to the mark descriptors in the Student Handbook at:

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors

Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof):

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good Academic Practice

Please ensure you are following the relevant guidelines on Good Academic Practice as outlined at:

https://www.st-andrews.ac.uk/students/rules/academicpractice/