SPL Assignment 4

Adiel Ashrov ; Ramzi Kahi

Published on: 30/12/2013
Due date: 16/1/2014 23:59

1 General Description

Please read the **whole** assignment before you start your work.
This assignment is all about networking, and to get your hands on the network, you will implement a simple version of Twitter. This is a social network where everyone can open a user and post tweets (short messages), and every user can follow other users, and get updates about the other users tweets. You will implement your twitter in two steps: 1) write a Twitter client, 2) write your very own Twitter server.

Your Twitter client will receive commands from the user and send them to the server. For example your user can use her Twitter client to post a Tweet. The server's job in this case it to send the tweet to all the clients following her. This is a rough description and we will extend on these matters later on. The client and server connect to each other using the TCP/IP framework, much like you have seen in class. TCP/IP is how you set a *phone line* between two machines. Once there is a connection between the two we have to decide what is the *communication protocol* used for communicating. A *protocol* is a set of rules for message exchange between computers. In a real life phone call the *protocol* can be an exchange of *Hello* messages and then each side speaks while the other side listens. The *communication protocol* you will implement in this assignment is the STOMP protocol[1], which resides above TCP/IP. STOMP supports spreading a message destined to a specific topic to a group of users subscribed to it. This is why we have chosen STOMP as the *communication protocol* between the server and the clients.

2 STOMP protocol

2.1 Overview

STOMP is a simple interoperable protocol designed for asynchronous message passing between clients via mediating servers. It defines a text based wire-format for messages passed between these clients and servers.
We will use the STOMP protocol in our assignment for passing messages between the client and the server. This section describes the format of STOMP messages/data packets, as well as the semantics of the data packet exchanges. For a complete specification of the STOMP protocol, read: STOMP 1.2.

2.2 STOMP Frame Format

The STOMP specification defines the term *frame* to refer to the data packets transmitted over a STOMP connection. A STOMP frame has the following general format:

```
<StompCommand>
<HeaderName_1>:<HeaderValue_1>
<HeaderName_2>:<HeaderValue_2>
```

[1]only the *frames* defined in the assignment should be supported

```
<FrameBody>
^@
```

A STOMP frame always starts with a STOMP command (for example, `SEND`) on a line by itself.
The STOMP command may then be followed by zero or more header lines. Each header is in a $< key >:< value >$ format and terminated by a newline.
A blank line indicates the end of the headers and the beginning of the body, $< FrameBody >$, (which is empty for some commands).
The frame is terminated by the null character, which is represented as `^@` above ($Ctrl + @$ in ASCII , '\u0000' in Java, and '\0' in C++).

2.3 STOMP Server

A STOMP server is modeled as a set of topics(queues) to which messages can be sent.
Each client can subscribe to one topic or more and it can send messages to any of the topics.
Every message sent to a topic, is being forwarded by the server to all clients registered to that topic.

2.4 Oneway Commands

Most STOMP commands have oneway semantics (that is, after sending a *frame* , the sender does not expect any reply). The only exceptions are:

- `CONNECT` command: after a client sends a `CONNECT` frame, it expects the server to reply with a `CONNECTED` frame.
- `ERROR` commands: if a client sends a frame that is malformed, or otherwise in error, the server may reply with an `ERROR` frame. Note, however, that the `ERROR` frame is not formally correlated with the original frame that caused the error.

There is a possibility in STOMP to request an acknowledging for all other commands/frames but in our assignment we do not require that.

2.5 Client Frames

We will now go over the client *frames*. For every *frame* we only mention the mandatory headers and you can add additional headers. Additional headers can be STOMP headers which are defined as optional for a certain *frame* or completely new headers defined by you.

An important requirement you must follow is if the *frame* sent to the server is in malformed format or causes a logical error, then the server must reply with an `ERROR` *frame*.

- CONNECT

  After opening a socket to connect to the remote server, the client sends a `CONNECT` command to initiate a STOMP session.
  For example, the following *frame* shows a typical `CONNECT` command, including the following headers[2]:

  – accept-version:STOMP version client is supporting, you should use 1.2.
  – host:IP address of the server running STOMP
  – login:login name
  – passcode:user password

[2]Take notice that are no spaces before or after the ':'. For example: accept-version:1.2

The *accept-version* and *host* headers are mandatory in STOMP 1.2. The *login* and *passcode* are optional headers, however in this assignment it is mandatory. We will now see an example of a `CONNECT` *frame*.

```
CONNECT
accept-version:1.2
host:[STOMP server IP]
login:[username]
passcode:[password]


^@
```

After the client sends the `CONNECT` frame, the server can acknowledge the connection by sending a `CONNECTED` frame, as follows:

```
CONNECTED
version:1.2


^@
```

- SEND

  The `SEND` command sends a message to a destination - a topic in the messaging system. It has one required header, destination, which indicates where to send the message. The body of the `SEND` command is the message to be sent. For example, the following frame sends a message to the "/topic/a" destination:

  ```
  SEND
  destination:/topic/a

  Hello topic a
  ^@
  ```

- SUBSCRIBE

  The `SUBSCRIBE` command registers a client to a specific topic. Like the `SEND` command, the `SUBSCRIBE` command **requires** a destination header. Henceforth, any messages received on the subscription are delivered as `MESSAGE` frames from the server to the client, for more detail see section2.6.
  The following *frame* shows a client subscribing to the topic, "/topic/a":

  ```
  SUBSCRIBE
  destination:/topic/a
  id:78


  ^@
  ```

  – id: specify an ID to identify this subscription[3]. Later, you will use the ID to `UNSUBSCRIBE`. When an id header is supplied in the SUBSCRIBE frame, the server must append the subscription header to any MESSAGE *frame* sent to the client (i.e. If clients a and b are subscribed to to /topic/foo with the id 0 and 1 respectively, and someone sends a message to that topic, than client a will receive the message with the id header equals to 0 and client b will receive the message with the id header equals to 1).

[3]You should generate the ID uniquely on the client side. There is no problem that the server will receive two subscriptions from different clients with the same ID.

- UNSUBSCRIBE

  The `UNSUBSCRIBE` command removes an existing subscription, so that the client no longer receives messages from that destination. It **requires** an id header.
  For example, the following frame cancels the previous subscription to "/topic/a" identified by the id 78:

  ```
  UNSUBSCRIBE
  id:78


  ^@
  ```

- DISCONNECT

  A client can disconnect from the server at any time by closing the socket but there is no guarantee that the previously sent frames have been received by the server. To do a graceful shutdown, where the client is assured that all previous frames have been received by the server, the client should:

  1. Send a `DISCONNECT` frame with a receipt header set (which again, need to be unique from the client side). For example:
     ```
     DISCONNECT
     receipt:77


     ^@
     ```
  2. Wait for the `RECEIPT` *frame* response to the `DISCONNECT`. For example:
     ```
     RECEIPT
     receipt-id:77


     ^@
     ```
  3. Close the socket.

  Note that, if the server closes its end of the socket too quickly, the client might never receive the expected `RECEIPT` frame. Therefore the client should close the TCP connection once it receives the RECEIPT frame. Clients MUST NOT send any more frames after the `DISCONNECT` frame is sent.

2.6 Server Frames

- MESSAGE

  The `MESSAGE` command conveys messages from a subscription to the client. The `MESSAGE` *frame* must include a destination header, which identifies the destination subscription. Another header which should be added is the id header which contains the id the client used to subscribe to this topic. The `MESSAGE` *frame* must also contain a message-id header with a **unique**[4] message identifier.
  The *frame* body contains the message contents.
  For example, the following frame shows a typical `MESSAGE` *frame* with destination and message-id headers:

  ```
  MESSAGE
  destination:/topic/a
  subscription:78
  message-id:00020

  hello topic a
  ```

[4]Every message the server delivers to subscribers has a **system-wide unique** message ID.

^@

- RECEIPT

  A `RECEIPT` frame is issued from the server whenever the client requests a receipt for a given command (for example `DISCONNECT`. The `RECEIPT` frame includes a receipt-id, containing the value of the receipt-id from the original client request. For example, the following frame shows a typical `RECEIPT` command:

  RECEIPT
  receipt-id:77

  ^@

  A `RECEIPT` frame is an acknowledgment that the corresponding client frame has been processed by the server. The `RECEIPT` body is always empty.

- ERROR

  The server may send `ERROR` frames if something goes wrong.
  According to the STOMP protocol, the `ERROR` *frame* should contain a `message` header with a short description of the error. In our simulation, the message header is obligatory.
  The body may contain more detailed information (or may be empty).
  For example, the following *frame* shows an `ERROR` command with a non-empty body:

  ERROR
  message:malformed STOMP *frame* received

  The message:
  ———————
  SEND
  destined:/topic/a

  Hello topic a!
  ———————
  Did not contain a destination header, which is required for message propagation.

  ^@

3 The Client

The client is a program that each Twitter user will run in order to connect to the service. The client will be written in C++ and will process commands from the user, that will require sending messages to the server and sometimes receive responses from the server. The client should handle commands from the user (from stdin) and responses from the server independently. Meaning a command from the user should never block the client from handling a message that arrived from the server and vice versa. Please have a look here for code examples in C++.

4 The Server

The server is a program that accepts users requests to connect to it, and handles their messages. The server will be written in Java in a *Thread Per Client* fashion. Meaning that each client that connects to the server will have it's own thread that handles its messages. The server will only react to messages it receives from clients.
For each client the server will maintain a list of followers. This list should stay forever, even if a all users logout

and then re-login later. In this case a user that re-login should receive all of the tweets that were sent while she was not connected (i.e., the server should remember for each user the last message it received).
In addition the server will create a "fake" user called "server".This user has its own topic, `server`, that all users may follow, however it will not have its own thread/socket. We will use this topic in order to retrieve information regarding the server.
Any message sent to this topic should be send to the server stdout, as well as to the followers of the topic.

5 The Client's Commands

This section describes the commands the client will receive from the console, and what it will do with them - namely, what frames it will send to the server and what possible responses the client may receive. This will be accompanied with an example of two users *KarlMarx* and *LuisKelso*. Please note that all commands can be processed only if the user is logged in (apart from login). In all of these commands, any error (whether an error frame or an error in the client side) should produce an appropriate message to the client stdout. In case of an error frame you can print the message header if it is informative enough, or the entire frame.

- *login* {*host IP*} {*host port*} {*username*} {*password*}
  This command will open a socket to the server at {*host IP*} {*host port*}. Then it will send a login request to the server. You can assume that *username* and *password* contain only English and numeric characters. The possible outputs the client can have for this command:

  - **Socket error:** If the {*host IP*} {*host port*} is incorrect the socket opening operation will fail - the output then should be "Could not connect to server. Check your Internet connection, IP and port.".
  - **New user:** If the server connection was successful and the server doesn't find the *username*, then a new user is created and the password is saved for that user. Then the server sends a CONNECTED frame to the client and the client will print "Login successfully.". The server should also open a topic for the client, and register the client to her own topic. The client cannot un-follow her own topic.
  - **User is already logged in:** If the user is already logged in, then the server will respond with a STOMP error frame that has in the header the reason.
  - **Wrong password:** If the user exists and the password doesn't match the saved password, the server will send back an appropriate ERROR frame indicating the reason.
  - **User exists:** If the server connection was successful, the server will check if the user exists in the users list, and if the password matches, also the server will check that the user does not have an active connection already. In case these tests are OK, the server sends back a CONNECTED frame and the client will print to the screen "Login successfully.". The user should again be added to her\his own queue, and receive all the tweets that were send while she was logged off.

  Example: The user KarlMarx opens the client for the first time and types in *login* 127.0.0.1 4444 *KarlMarx* 1234. Then the following scenario will happen:

  1. The client opens a socket to 127.0.0.1 on port 4444.
  2. The client sends a CONNECT frame to the server with the username KarlMarx and password 1234.
  3. The server checks its data and finds no KarlMarx, so it creates this user with the password 1234.
  4. The server sends to the clients CONNECTED frame.
  5. The client prints to the screen "Login successfully.".

  Note: The socket stays open until the clients sends a "logout" request, or until the server closes it (see *stop* command).

- *follow* {*username*}
  This command tells the client that the user wants to follow another user. The client will send a message to the server telling that the current user wants to follow another user called {*username*}.

  - **Wrong username:** The server checks if *username* exists. If not, the server will return to the client an appropriate error frame.
  - **Already following username:** If the user is already following the *username*, the server will return the client an appropriate error frame.
  - **Success:** The server sends back a MESSAGE frame saying "following {username}" and the client will print to the screen "Now following {username}.".

  Example: The user KarlMarx wants to follow the user LouisKelso. So he writes the command $followLouisKelso$. Then the following scenario will happen (assuming LuisKelso exists):

  1. The client sends a $follow\ LouisKelso$ message to the server.
  2. The server finds the user LouisKelso, and saves KarlMarx as one of the users following LouisKelso.
  3. The server responses with $following\ LouisKelso$.
  4. The client receives this message and prints "Now following LouisKelso.".

  Another example: The user KarlMarx is again typing the same command "follow LouisKelso" while he already is following him. Then the following scenario will happen:

  1. The client sends a $follow\ LouisKelso$ message to the server.
  2. The server finds the user LouisKelso, and finds that the user KarlMarx is already following LouisKelso.
  3. The server responses with an ERROR frame.
  4. The client receives this message and prints "Error: Already following LouisKelso.".

- $unfollow$ *{username}*
  This command tells the client that the user wants to unfollow another user. The client will send a message to the server telling that the current user wants to unfollow *{username}*

  - **Wrong username:** The server checks if the *username* exists. If not, the server will return an appropriate error frame.
  - **Not following this user:** If a *username* exists but the client does not follow *username*, the server should return an appropriate error frame.
  - **Trying to unfollow itself:** As stated, every user should be following herself. An unfollow command on the current user should be followed with an appropriate error frame.
  - **Success:** The server sends back a message saying "unfollowing {username}" and the client will print to the screen "No longer following {username}.".

  Example: The user KarlMarx wants no longer to follow the user LouisKelso due to their different economic views so he writes the command $unfollowLouisKelso$. Then the following scenario will happen:

  1. The client sends a $unfollow\ LouisKelso$ message to the server.
  2. The server finds the user LouisKelso, and deletes KarlMarx from the followers list.
  3. The server sends a messge back to the user saying "unfollowing LouisKelso".
  4. The client the prints "No longer following LouisKelso.".

- *tweet {message}*
  This command tells the client that the user wants to send a new tweet. The client will send a `send` *frame* to the server.

  - **Success:** This kind of message should always be accepted by the server (as long the user is logged-in and that the frame is not malformed). The server will send the tweet to all users that are following the current user, using the MESSAGE frame. The MESSAGE frame must include in the header the information about who sent this message, and when it was received at the server. The body of the MESSAGE will include the tweet itself. These other users should then print this message to the console and also to an HTML file (described in section 6).

Example: The user LouisKelso wants to publish a tweet saying "Hello, World!", and KarlMarx is following him. The following scenario will happen:

1. The client sends a SEND frame, with "Hello, World!" in the body (and appropriate headers).
2. The server finds the user LouisKelso's topic, and sends to the users LouisKelso and KarlMarx the message "Hello, World!" from LouisKelso.
3. Both clients (LouisKelso and KarlMarx) print the tweet "Hello, World!" to the screen.

  - **Mention a user**: A nice feature of twitter is the ability to mention another user in your tweets using the @ character. For example: "Hello, World! and Hello @KarlMarx" tweeted by LouisKelso.
    In this case the server should accept the the tweet and send it to all of LouisKelso followers **and** to all of KarlMarx followers.

- *clients* [*online*]

  When this command is triggered, the client sends a SEND frame to the server fake user , `server`, requesting a list of all users. The server sends to it's stdout a list of all users.
  In addition, the server will send a MESSAGE frame to the "server" fake user topic, `server`, with a list of all users in its body.
  If the online parameter is given, only the online users are sent (not all users).

  Example: Lets assume we have 7 clients that have logged into the server: *Achiya Majeed Jumana Rafi Matan Adiel Ramzi*
  Then: *Adiel* and *Ramzi* logout.
  Jumana and *Majeed* type the command *follow server*.
  Achiya types the command *clients*.
  The server will receive this message and print to it's stdout a list of all users. It will also send a message to *Jumana* and *Majeed* containing a list of all the users: *Adiel Ramzi Achiya Majeed Jumana Rafi Matan*

  Achiya types the command *clients online*.
  The server will receive this message and print to it's stdout a list of all **online** users. It will also send *Jumana* and *Majeed* a list of all the **online** users: *Achiya Majeed Jumana Rafi Matan*

- *stats*
  The client will SEND a frame to the server fake user, `server`, and the server will send to all the clients subscribed to the `server` topic the following server statistics.

  - Max number of tweets per 5 seconds. // a minute is too long
  - Avg. number of tweets per 5 seconds.
  - Avg. time (in seconds) to pass a tweet to all users following an account.
  - Name of the user with the maximum number of followers and the number of the followers.
  - Name of the user with the maximum number of tweets and the number of tweets.
  - Name of the user with the maximum mentions in other followers tweets.
  - Name of the user with the maximum number of mentions in her own tweets (used more mentions than anyone else)

- *logout*
  This command tells the client that the user wants to log out from twitter. The client will send a message to the server requesting a logout.

  - **Success:** This kind of request will always be accepted by the server (as long the user is logged in). The client should send a DISCONNECT stomp frame. The server will reply with a RECEIPT frame. Once the client receives the RECEIPT frame, it should close the socket and await further user commands.

The client should not send any message to the server if the user has not logged in yet. In case any command is typed by the user before a login (or after a logout) the client should print an appropriate message. The only exception is the *exit_client* command.

- *exit_client*
  This command tells the client to shutdown and **exit gracefully**. If the client is logged in the behavior is similar to the *logout* command with the addition of exiting the application. If the client is not *logged in*, the client simply needs to exit gracefully.

- *stop*
  This command tells the client that the user wants to stop the server. The client will send a SEND frame to the "server" topic (fake user), and the server will close all its connections and exit the process. This will be always accepted from all users (after a login) and will not respond to the client.
  Example: The user $LouisKelso$ types the command "stop", then client sends a a SEND frame to the server (with a header saying that it's a stop command). The server then closes all its connections and exits.
  If the server closes all of it's connections, all of the clients which were connected to it are still required to **exit gracefully**.

6 Logging

The client should log all tweets that were sent to the client (i.e., all tweets that were sent by the clients that are being followed). This log should be written to a file named {*username*}*.html*, in the HTML format. You may use a logger (see assignment 2 and 3) for any other logging messages, though it is not obligatory.

6.1 What is HTML

HTML stands for Hyper Text Transfer Protocol. HTML is a subset of SGML with predefined tags, which web browsers can parse into a pretty web site. On the assignment site you can find an example HTML file for the user KarlMarx. You may open this file with the browser, and then with an application like kate to see the source. (If you open the file in the browser and do a right-click then select "View source" you will get the same text). On the course site there is some information to get started, look here.

6.2 What should be in the file

The {*username*}*.html* file should contain the following information:

- The user's username in a header.
- All tweets the user recieved because she is following (or followed) other users.

The tweets should be orderd chronologically, and should be formated as follows:
{username} {Tweet body} posted at: {Date and time recieved at the server}
You may format the HTML as you like, but keep it clear where one tweet starts and ends, and who posted this tweet and when. There are many ways to make it look good in the browser. You may wish to look up the following tags: *table div span*. Two simple examples:
Example 1:
$< table >$
$< tr >< td > user1 < /td >< td > tweet1 < /td >< td > 12.12.199912 : 20 : 01 < /td >< /tr >$
$< tr >< td > user2 < /td >< td > tweet2 < /td >< td > 12.12.199912 : 20 : 20 < /td >< /tr >$
$< tr >< td > user3 < /td >< td > tweet3 < /td >< td > 12.12.199912 : 27 : 03 < /td >< /tr >$
$< tr >< td > user1 < /td >< td > tweet4 < /td >< td > 12.12.199913 : 12 : 25 < /td >< /tr >$
$< /table >$
Example 2:
$< div >< span >< b > user1 < /b >< /span >< span > tweet1 < /span >< span > 12.12.199912 : 20 : 01 <$
$/span >< /div >$

< div >< span >< b > user2 < /b >< /span >< span > tweet2 < /span >< span > 12.12.199912 : 20 : 20 < /span >< /div >
< div >< span >< b > user3 < /b >< /span >< span > tweet3 < /span >< span > 12.12.199912 : 27 : 03 < /span >< /div >
< div >< span >< b > user2 < /b >< /span >< span > tweet4 < /span >< span > 12.12.199913 : 12 : 25 < /span >< /div >

* Note: these are just HTML snippets, and cannot be rendered as is, but this is just a part of how you may do it. Also you may wish to add a little more formatting attributes to make it look better.
Your file is expected to be rendered by Firefox, please check that this browser really opens your file, because there are some minor differences between the browsers.

7 Design

There are some things you are required to do in the assignment, and some we leave up to you.
Things you MUST do:

- Both, client and server, have to implement an abstract class called StompFrame, which will have a child class for each type of STOMP frame (e.g., SendFrame, ConnectFrame, DisconnectFrame, etc.). If a client wants to tweet "hello world" for example, it should construct an instance of SendFrame and send it to the connection handler write method.
- The server should follow the Thread per client design patter as taught in class.
  You should notice that the ConnectionHandler in our code receives from the client one line each time while a STOMP frame has more than one line. Therefore you will need to improve our Thread Per Client server and make it use of the StompTokenizer interface given in the assignment page. You can get an idea of how to implement it and how to use it by looking at the NIO echo client.
  Make sure that at the end, your server is not STOMP specific and only the tokenizer and the protocol are STOMP specifically. As you will see, without using generics, there is no way to create a genreic thread per client server , as you will need to assign the return value of the nextMessage method of the StompTokenizer from within the ConnectionHandler. However this is the only exemption we allow.
- On the server every socket should be inside a ConnectionHandler object, as taught in class. The thread should hold a ConnectionHandler and read/write through this object's methods and not directly to the socket or one of its streams.
- The client should have one thread that listens to the keyboard (stdin), which will execute commands from the user. Another thread should listen to the network and get responses/notifications from the server and handle them.
- You can add methods and members to the client's ConnectionHandler as seen in the practical session, but don't change exiting methods.
- The server and the client should have separate packages/namespaces (respectively) for the STOMP related classes and the server/client related classes. You may not use the default package/namespace (i.e., without entering any package/namespace).

8 How should I start?

1. Download ActiveMQ and run it.
2. Connect to the ActiveMQ server using telnet (see the ActiveMQ page in the Assignment page).
3. Write a few STOMP messages according to the STOMP Protocol.
4. Implement a STOMP client in C++ (not a twitter client!), make sure it talks with the ActiveMQ server.

5. Implement a STOMP server to replace the ActiveMQ server.
6. Implement the Twitter client and server.

9 Reactor

9.1 Overview

In this part of the assignment you will change the server implementation from the thread per client design pattern to the reactor design pattern. Except for the Message, ServerProtocol and the MessageTokenizer, the reactor classes are general and may serve other server types as well. In your implementation - keep it general as well.

9.2 A Reactor-based STOMP Server

Your objective is to build a Twiter server using the Reactor design pattern instead of the thread-per-client model. You should reuse as much as possible the existing code. Assuming you have designed your thread-per-client server correctly, the changes should be minimal.

9.3 A Stress Test Client

We will provide you with a stress test client implementation which will allow you to push your Reactor to it's limits and view how different configurations affects on the Reactor performance statistics.

9.4 Measuring Metrics

When calling the stats command, the reactor server should return the same statistics described in section 5. In addition, the reactor will keep track of counters that reflect its capability to handle the stress imposed by the stress client. These counters are called *metrics*, and they should be sent to the server followers along with the regular statistics.
The metrics we will measure on the server side are the following:

- **Incoming bytes**: how many bytes altogether have been received by the server since it started running.
- **Outgoing bytes**: how many bytes altogether have been sent by the server to its clients since it started running.
- **Connections number**: how many connections have been received by the server since it started running.
- **Incoming messages**: how many messages did the server receive and processed since it started running. A message is counted only after the corresponding $processMessage$ call has terminated.
- **Connection latency**: how long did it take(on average) for the server to react from the time its selector indicated an acceptable connection to the time the connection was established (this is not end-to-end latency as seen from the client - it is an internal metrics of how long it takes for the server to react to an event of type "acceptable" until the connection handler is ready to receive bytes from the client. This metric should be measured in milliseconds and the counter should return the sum of all the delays for all the connections since the server started running.
- **Response latency**: how long did it take(on average) for the server from the time it receives a message from a client (a complete message) until the message was completely handled. Again this is not an end to end measure as seen from the clients, it is an internal measure of how long the server works inside the $processMessage$ calls of the protocol. This metric should be measured in milliseconds and the counter should return the sum of all the delays for all the connections since the server started running.

- **Read Fragmentation Level**: this metric counts how often a sequence of bytes received from a client does not contain a full protocol message. That is, each time a socket channel becomes readable, a protocol Task is invoked but the tokenizer does not find a complete message, the "Read Fragmentation Counter" is increased by one. In parallel, each time a socket channel becomes readable, the "Read Counter" is increased by one.The "Read Fragmentation Level" is computed as $ReadFragmentationCounter/ReadCounter$. It is reported as a float number between 0 and 1. (Make sure to initialize ReadCounter to 1).
- **Write Fragmentation Level**: this metric counts how often the server tries to send data to the client but has to wait because the SocketChannel is not writable. It is computed as $WriteFragmentationCounter/WriteCounter$ where $WriteFragmentationCounter$ is increased each time the server attempts to write bytes to the socket but does not complete and send all the bytes and $WriteCounter$ is increased each time the server sends bytes.

9.5 How to Run the Stress Simulation

To execute your program, you must:

- First run the server.
- Run the stress client (see stress test client).
- The client will automatically terminate when all messages have been sent and all subscribed messages have been received (each thread should know how many replies it should wait for depending on the subscription mode).
- Before terminating, the client will automatically call the stat command, and then the stop command, and the server will send to stdout and to the server followers the regular statistics as well as:
  Incoming bytes: N1
  Outgoing bytes: N2
  Connections number: N3
  Incoming messages: N4
  Connection latency: N5 msec – N5' msec per connection average
  Response latency: N6 msec – N6' msec per response average
  Read Fragmentation Level: N7
  Write Fragmentation Level: N8
- After sending the statistics, the server will terminate gracefully.

10 Documentation and Coding Style

There are many programming practices that must be followed when writing any piece of code.

- Follow Java Programming Style Guidelines. A part of your grade will be checking if you have followed these mandatory guidelines or not. It is important to understand that programming is a team oriented job. If your code is hard to read, then you are failing to be a productive part of any team, in academics research groups, or at work. For this, you must follow these guidelines which make your code easier to read and understand by your fellow programmers as well as your graders.
- Do not be afraid to use long variable and method names as long as they increase clarity.
- Avoid code repetition - In case where you see yourself writing same code block in two different places, it means this code must be put in a private function, so both these places may use it.
- Full documentation of the different classes and their public and protected methods, by following Javadoc style of documentation. You may, if you wish, also document your local methods but they are not mandatory.

- Add comments for code blocks where understanding them without, may be dificult.
- Your files must not include commented out code. This is garbage. So once you finish coding, make sure to clean your code.
- Long functions are frowned upon (30+ lines). Long functions mean that your function is doing too many tasks, which means you can move these tasks to their own place in private functions, and use them as appropriate. This is an important step toward increased readability of your code. This is to be done even in cases where the code is used once.
- Magic numbers are bad! Why? Your application must not have numbers throughout the code that need deciphering.
- Do not reveal the internal implementation of your objects.
  - For instance: The thread of user X should never access the socket of user Y, only the ConnectionHandler of user Y. In other words, if one client has posted a tweet, that client's thread at the server side should write to the other clients' ConnectionHandlers the frame, by calling the write method.

Your implementation must follow these steps in order to keep the code ordered, clean, and easy to read

11 Submission Instructions

Submission is done only in pairs. If you do not have a pair, find one. You need explicit authorization from the course staff to submit without a pair. You cannot submit in a group larger than two.

- You must submit one .tar.gz file with all your code. The file should be named "assignment4.tar.gz". Note: We require you to use a .tar.gz file. Files such as .rar, .zip, .bz, or anything else which is not a .tar.gz file will not be accepted and your grade will suffer.
- Your tar.gz file must include three folders, $Twitter-thread-per-client-server$, $Twitter-reactor-server$, and $Twitter-client$. Both server directories should include the source files as well as an ant build file. The client folder should include the source files as well as a makefile. The grader will extract your file, build your application using ant build file and the makefile, and execute it. You need to submit only source/header files and not binary files, take notice.

11.1 Extensions

- Extension requests are to be sent to spl141@cs.bgu.ac.il. Your request email must include the following information:
  - Your name and your partner's name.
  - Your id and your partner's id.
  - Explanation regarding the reason of the extension request.
  - Your submission system group number.
  - Official certification for your illness or milluim.

Requests without a compelling reason will not be accepted.

12 Grading

Although you are free to work wherever you please, assignments will be checked and graded on Computer Science Department Lab Computers - so be sure to test your program thoroughly on them before your final submission. "But it worked fine on my windows based home computer" will not earn you any mercy points from the grading staff, if your code fails to execute at the lab.
Grading will tackle many points, including but not limited to:

- Your application design and implementation.
- Automatic tests will be run on your application. Your application must complete successfully and in reasonable time.
- Synchronization in the server and in the client - How do you know there are no deadlocks\livelocks?
- Explain how the server maintains consistency. E.g. what if a user X wishes to unfollow user Y, and user Y just posted a new tweet?
- Networking principals - You may need to explain what a socket is, what the IP layer etc.
- Memory leaks.
- Checking if your implementation follows the guidelines detailed in 6.

Good luck and have fun!