1. General Description:

In this assignment you will implement a book club. Users will be able to signup for reading clubs and borrow books from each other.

You will implement both a server, which will provide STOMP server services, and a client, which a user can use in order to interact with the rest of the users. The server will be implemented in J**ava** and will support both **Thread-Per-Client (TPS)** and the **Reactor**, choosing which one according to arguments given on startup. The client will be implemented in **C++** , and will hold the required logic as described below.

All communication between the clients and the server will be according to the **STOMP** `Simple-Text-Oriented-Messaging-Protocol' protocol.

In order to get you started, we supply few examples for different protocols and clients, the most complete one being **newfeed**, we recommend you go over it, specifically, note how it can use both **TPS** or **Reactor** server implementations. You will need to comment out lines *19* and *21* (which was added for this assignment) in order for the supplied implementations to work.

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_1>:<HeaderValue_1>

<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 can be 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++).

2.2 *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.

2.3 *Connecting:*

A STOMP client initiates the stream or TCP connection to the server by sending the CONNECT frame:

```
CONNECT
accept-version:1.2
host:stomp.cs.bgu.ac.il
login:bob
passcode:alice


^@
```

The sever may either response with a CONNECTED frame:

```
CONNECTED
version:1.2


^@
```

Or with an ERROR frame (see below).

2.4 *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** 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
subscription:78
```

```
Message-id:00020
destination:/topic/a


Hello Topic a
^@
```

- RECEIPT

A RECEIPT frame is sent from the server to the client once a server has successfully processed a client frame that requests a receipt. A RECEIPT frame MUST include the header receipt-id, where the value is the value of the receipt header in the frame which this is a receipt for.

```
RECEIPT
receipt-id:00032


^@
```

A RECEIPT frame is an acknowledgment that the corresponding client frame has been *processed* by the server. Since STOMP is stream based, the receipt is also a cumulative acknowledgment that all the previous frames have been *received* by the server. However, these previous frames may not yet be fully *processed*. If the client disconnects, previously received frames SHOULD continue to get processed by the server.

- ERROR

The server **MAY** send ERROR frames if something goes wrong. In this case, it **MUST** then close the connection just after sending the ERROR frame. The ERROR frame SHOULD contain a message header with a short description of the error, and the body MAY contain more detailed information (or MAY be empty).

```
ERROR
receipt-id: message-12345
message: malformed frame received


The message:
-----
MESSAGE
destined:/queue/a
receipt: message-12345


Hello queue a!
-----
Did not contain a destination header, which is REQUIRED for message
propagation.
^@
```

If the error is related to a specific frame sent from the client, the server SHOULD add additional headers to help identify the original frame that caused the error. For example, if the frame included a receipt header, the ERROR frame SHOULD set the receipt-id header to match the value of the receipt header of the frame which the error is related to.

2.5 *Client Frames*

- 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 destination header. Henceforth, any messages received on the subscription are delivered as MESSAGE frames from the server to the client.The following frames shows a client subscribing to the topic, ”/topic/a”

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


^@
```

id: specify an ID to identify this subscription. 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 /topic/foo with the id 0 and 1 respectively, and someone sends a message to that topic, then 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).

- 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


^@
```

Note that the *receipt* header can be added to ANY client frame which requires a response.

3. **Implementation Details**:

3.1 *General Guidelines:*

- The server should be written in Java. The client should be written in C++ with BOOST. Both should be tested on Linux installed at CS computer labs.
- You must use maven as your build tool for the server and makefile for the c++ client.
- The same coding standards expected in the course and previous assignments are expected here.
- You can complete both parts simultaneously, you can emulate the server/client using an implementation from here, ActiveMQ, Stompy, or Gozirra should all work fine.

3.2 Server:

You will have to implement a single protocol, supporting both the **Thread-Per-Client** and **Reactor** server patterns presented in class. Code seen in class for both servers is included in the assignment wiki page. You are also provided with 3 new or changed interfaces:

- **Connections**

This interface should map a unique ID for each active client connected to the server. The implementation of Connections is part of the server pattern and not part of the protocol.It has 3 functions that you must implement (You may add more if needed):

```
boolean send(int connectionId, T msg);
```

Sends a message T to client represented by the given connectionId.

```
void send(String channel, T msg);
```

Sends a message T to clients subscribed to channel.

```
void disconnect(int connectionId);
```

Removes an active client connectionId from the map

- **ConnectionHandler<T>**

A function was added to the existing interface.

```
void send(T msg);
```

sends msg T to the client. Should be used by the send commands in the Connections implementation.

- **StompMessagingProtocol**

This interface replaces the MessagingProtocol interface. It exists to support peer 2 peer messaging via the Connections interface. It contains 3 functions:

```
void start(int connectionId, Connections<String> connections);
```

Initiate the protocol with the active connections structure of the server and saves the owner client's connection id.

```
void process(String message);
```

As in MessagingProtocol, processes a given message. Unlike MessagingProtocol, responses are sent via the connections object send functions (if needed).

```
boolean shouldTerminate();
```

true if the connection should be terminated

Left to you, are the following tasks:

1.Implement Connections<T>to hold a list of the new ConnectionHandler interface for each active client. Use it to implement the interface functions. Notice that given a connections implementation, any protocol should run. This means that you keep your implementation of Connections on T.

public class ConnectionsImpl<T> implements Connections<T>{...}.

2.Refactor the Thread-Per-Client server to support the new interfaces. The ConnectionHandler should implement the new interface. Add calls for the new Connections<T> interface.

3.Refactor the Reactor server to support the new interfaces. The ConnectionHandler should implement the new interface. Add calls for the new Connections<T> interface.

You may add classes as you wish. Note that the server implementation is agnostic to the STOMP protocol implementation, and can work with different STOMP implementations, as long as they follow the rules defined by the protocol.

Leading questions:

- Which classes and interfaces are part of the Server pattern and which are part of the Protocol implementation?
- When and how do I register a new connection handler to the Connections interface implementation?
- When do I call start to initiate the connections list? Start must end before any call to Process occurs. What are the implications on the reactor? (Note: start cannot be called by the main reactor thread and must run before the first)
- How do you collect a message? Are all message types collected the same way?

Tips:

- You can test tasks 1 –3 by fixing one of the examples in the impl folder in the supplied spl-net.zip to work with the new interfaces (easiest is the echo example)
- You can complete tasks 1 and 2 and return to the reactor code later. Thread per client implementation will be enough for testing purposes
- Note that the server only responds to frames sent by the clients, and holds no logic whatsoever! Every SEND frame from one of the clients is being distributed to the appropriate topic.

Testing run commands:

- Build using `mvn compile
- Thread per client server: ` mvn exec:java -Dexec.mainClass="bgu.spl.net.impl.stomp.StompServer" -Dexec.args="<port> tpc'
- Reactor server: ` mvn exec:java -Dexec.mainClass="bgu.spl.net.impl.stomp.StompServer" -Dexec.args="<port> reactor'

The **server** directory should contain a pom.xml file and the src directory. Compilation will be done from the server folder using: **mvn compile**

The server should be implemented in the file `StompServer`, under `stomp` subdirectory.

3.3 Client:

An echo client is provided, but it is a single threaded client. While it is blocking on stdin (read from keyboard) it does not read messages from the socket. You should improve the client so that it will run 2 threads. One should read from keyboard while the other should read from socket. ~~The client should receive the server's IP and PORT as arguments.~~ You may assume a network disconnection does not happen (like disconnecting the network cable). You may also assume legal input via keyboard. The client should receive commands using the standard input. Commands are defined below this. You will need to translate from keyboard command to network messages and the other way around to fit the specifications. Notice that the client should close itself upon reception of an RECEIPT message in response of an outgoing DISCONNECT command. The Client

directory should contain a src, include and bin subdirectories and a Makefile as shown in class. The output executable for the client is named StompBookClubClient and should reside in the bin folder after calling make. Testing run commands: StompBookClubClient

The Stomp Book Club

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. Please note that all commands can be processed only if the user is logged in (apart from login). In all 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.

The client should print all frames received from the server, in the format:

"{topic}:{content}"

Commands for all users:

- Login Command
    - Structure: login {host:port} {username} {password}
    - For this command a CONNECT frame is sent to the server.
    - You can assume that username and password contain only English and numeric The possible outputs the client can have for this command:
        - Socket error: connection error. In this case the output should be "Could not connect to server".
        - 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 successful".
        - User is already logged in: If the user is already logged in, then the server will respond with a STOMP error frame indicating the reason – the output in this case should be "User already logged in".
        - 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 - the output in this case should be "Wrong password".
        - 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 successful.".
- Join Genre Reading Club Command
    - Structure: join {genre}
    - For this command a SUBSCRIBE frame is sent to the {genre} topic.
    - As a result, a RECIEPT will be returned to the client. A message "Joined club {genre}" will be displayed to the screen.
- Exit Genre Reading Club Command
    - Structure: exit {genre}
    - For this command a UNSUBSCRIBE frame is sent to the {genre} topic.
    - As a result, a RECIEPT will be returned to the client. A message "Exited club {genre}" will be displayed to the screen.

- Add Book Command
    - Structure: add {genre} {book name}
    - For this command a SEND frame is sent to the topic {genre} with the content: “{user} has added the book {book name}
    - The book will be added to the client inventory.
    - The inventory is per genre, no book is multi genre.
- Borrow Book Command
    - Structure: borrow {genre} {book name}
    - For this command a SEND frame is sent to the topic {genre} with the “{user} wish to borrow {book name}“ in the content.
    - The server distribute the msg to all users subscribed to the {genre}, if one of them holds the book in his stock, he will send a SEND frame to the topic {genre}, with “{username} has {book name}” as content.
    - If some one has the requested book, another SEND frame will be sent to the {genre} topic, with "Taking {book name} from {book owner username}", this will result in the book adding up to the original (the borrower) user inventory, and being removed from the lender inventory.
    - Transitive borrowing is allowed (I.e. Bob borrows from John which borrowed from Alice).
    - If multiple users has a book, the borrower will take from the first one only (according to msg arrival order).
    - This command has multiple frames involved.
- Return Book Command
    - Structure: return {genre} {book name}
    - For this command a SEND frame is sent to {genre} topic with the content “Returning {book name} to {book lender}”.
    - This will result in removing the book from the borrower inventory, and adding it back to the lender.
    - If a book has been double borrowed, it need to be returned in the correct order.
- Genre Book Status Command
    - Structure: status {genre}
    - For this command a SEND frame is sent to the {genre} topic with "book status” in the body.
    - All the subscribed users will send a SEND frame, each with its current inventory, each book seperated by a comma (,), and the name (example below).
- Logout Command
    - Structure: logout
    - This command tells the client that the user wants to log out from the library. The client will send a DISCONNECT to the server.
    - The server will reply with a RECEIPT frame.
    - The logout command, removes the current user from all the topics.
    - Once the client receives the RECEIPT frame, it should close the socket and await further user commands.

4. Examples:

Command	Frames Sent	Frames Recieved

login 1.1.1.1:2000 bob alice	CONNECT accept-version:1.2 host:stomp.cs.bgu.ac.il login:bob passcode:alice ^@	CONNECTED version:1.2 ^@
join sci-fi	SUBSCRIBE destination:sci-fi id:78 receipt:77 ^@	RECEIPT receipt-id:77 ^@
add sci-fi Foundation	SEND destination:sci-fi Bob has added the book Foundation ^@	MESSAGE subscription:78 Message-id:00020 destination:sci-fi Bob has added the book Foundation ^@
borrow sci-fi Dune	SEND destination:sci-fi Bob wish to borrow Dune ^@ _____ _____ SEND	MESSAGE subscription:78 Message-id:00021 destination:sci-fi Bob wish to borrow Dune ^@ _____ MESSAGE

	destination:sci-fi Taking Dune from john ^@	subscription:78 Message-id:00022 destination:sci-fi john has Dune ^@ _____ MESSAGE subscription:78 Message-id:00023 destination:sci-fi Taking Dune from john ^@
return sci-fi Dune	SEND destination:sci-fi Returning Dune to john ^@	MESSAGE subscription:78 Message-id:00024 destination:sci-fi Returning Dune to john ^@
status sci-fi	SEND destination:sci-fi book status ^@ _____ SEND	MESSAGE subscription:78 Message-id:00025 destination:sci-fi Book status ^@

	destination:sci-fi bob:Foundation ^@	_____ MESSAGE subscription:78 Message-id:00026 destination:sci-fi bob:Foundation ^@ _____ MESSAGE subscription:78 Message-id:00027 destination:sci-fi john:Dune,1984, Harry Potter and the Methods of Rationality ^@
logout	DISCONNECT receipt:78 ^@	RECEIPT receipt-id:78 ^@