SPL251: Assignment 3 instructions

Omri Hirsch, Dan Zlotnikov

January 5, 2025

1 General Description

This assignment involves implementing an "Emergency Service" platform subscription service, where users can subscribe to specific emergency channels based on their interests or needs, such as fire, medical, police, and natural disasters. Subscribers can report emergencies and receive updates relevant to the channel's topic, enabling effective community collaboration during crises.

For this purpose, you will implement both a server and a client:

1. The server will provide centralized STOMP (Simple-Text-Oriented-Messaging-Protocol) server services, ensuring efficient communication between users. It will be implemented in Java and support two modes of operation:

   - Thread-Per-Client (TPC): Handles each client with a dedicated thread.
   - Reactor: Uses an event-driven model for handling multiple clients efficiently.

   The mode will be chosen based on arguments given at startup.

2. The client, implemented in C++, will allow users to interact with the Emergency Service system. Each client will handle logic for subscribing to channels, sending emergency reports, and receiving updates from the server.

All communication between the clients and the server will adhere to the STOMP protocol, ensuring standardized messaging.

To assist you in getting started, we provide examples of different protocols and clients, including the newsfeed and echo examples. These examples demonstrate how to utilize both TPC and Reactor server implementations. We recommend reviewing these to better understand the architecture and implementation details.

2 Simple-Text-Oriented-Messaging-Protocol (STOMP)

2.1 Overview

STOMP is a simple inter-operable 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 STOMP, 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>
<HeaderName1>:<HeaderValue1>
<HeaderName2>:<HeaderValue2>

<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. The order of the headers shouldn't matter, that is, the frame:

```
SUBSCRIBE
destination: /dest
id: 1

^@
```

should be handled the same as:

```
SUBSCRIBE
id: 1
destination: /dest

^@
```

and not cause an error due to permutation of the headers.
A blank line indicates the end of the headers and the beginning of the body `<FrameBody>` (which can be empty for some commands, as in the example of `SUBSCRIBE` above). The frame is terminated by the **null character**, whichis

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 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:meni
passcode:films

^@
```

Your STOMP clients will set the following headers for a `CONNECT` frame:

- `accept-version`: The versions of the STOMP protocol the client supports. In your case it will be version 1.2.
- `host`: The name of a virtual host that the client wishes to connect to. Since your server will hold only a single host, you can set this one to be `stomp.cs.bgu.ac.il` by default.
  Note: Generally servers can simulate many different virtual hosts for different purposes, and this header is used to determine to which of them the clients wishes to connect.
- `login`: The user identifier used to authenticate against a secured STOMP server. Should be unique for every user.
- `passcode`: The password used to authenticate against a secured STOMP server.

The `CONNECT` sets `<FrameBody>` as empty.

The sever may either respond with a `CONNECTED` frame:

```
CONNECTED
version:1.2

^@
```

Or with an `ERROR` frame, as will be shown below.
Your `CONNECTED` frame should have a single `version` header defined (and no other header), which has the value of the STOMP version used, which is 1.2 in your case. The `<FrameBody>` is again defined as empty.

2.5 Stomp frames

In addition to the above defined `CONNECT` and `CONNECTED` frames, we define a few more STOMP frames to be used in your implementation.

The following is a summary of the frames we will define:
Server frames:

- CONNECTED (as defined above)
- MESSAGE
- RECEIPT
- ERROR

Client frames:

- CONNECT (as defined above)
- SEND
- SUBSCRIBE
- UNSUBSCRIBE
- DISCONNECT

2.5.1 Server frames

- MESSAGE:
  The `MESSAGE` command conveys messages from a subscription to the client.

```
MESSAGE
subscription:78
message-id:20
destination:/topic/a

Hello Topic a
^@
```

  The `MESSAGE` frame should contain the following headers:

  - `destination`: the subscription to which the message is sent.
  - `subscription`: a **client-unique** id that specifies the subscription from which the message was received. This id will be supplied by the client, more on that in the `SUBSCRIBE` client frame.

  - `message-id`: a **server-unique** id that for the message. To be picked by the server.

  The frame body contains the message contents.

- 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.

```
RECEIPT
receipt-id:32

^@
```

  The `RECEIPT` frame should contain the single header `receipt-id`, and it's value should be the value specified by the frame that requested the receipt. The frame body should be empty.

  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.

  NOTE: the `receipt` header can be added to ANY client frame which requires a response. Thus, ANY frame received from the client that specified such header should be sent back a receipt with the corresponding `receipt-id`.

- 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.

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

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

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

The `ERROR` frame SHOULD contain a `message` header with a short description of the error, and the body MAY contain more detailed information (as in the example above) or MAY be empty.
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 to which the error is related (as in the above frame example).

2.5.2 Client frames

- SEND:
  The `SEND` command sends a message to a destination - a topic in the messaging system.

  ```
  SEND
  destination:/topic/a

  Hello topic a
  ^@
  ```

  The `SEND` frame should contain a single header, `destination`, which indicates which topic to send the message to.
  The body of the frame should contain the message to be sent to the topic. Every subscriber of this topic should receive the content of the body as the content of a `MESSAGE` frame's body, sent by the server.

  If the server cannot successfully process the `SEND` frame for any reason, the server MUST send the client an `ERROR` frame and then close the connection.

  In your implementation, if a client is not subscribed to a topic it is not allowed to send messages to it, and the server should send back an `ERROR` frame.

- SUBSCRIBE:
  The `SUBSCRIBE` command registers a client to a specific topic.

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

  ^@
  ```

  The `SUBSCRIBE` frame should contain the following headers:

  - `destination`: Similar to the destination header of `SEND`. This header will indicate to the server to which topic the client wants to subscribe.

    ◦ `id`: specify an ID to identify this subscription. Later, you will use the ID if you `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. For example, 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 equal to 0 and client b will receive the message with the `id` header equals to 1.
      Thus, you must generate this ID uniquely in the client before subscribing to a topic.

  The body of the frame should be empty.

  After this frame was processed by the server, any messages received on the `destination` subscription are delivered as `MESSAGE` frames from the server to the client.

  If the server cannot successfully create the subscription, the server MUST send the client an `ERROR` frame and then close the connection.

- UNSUBSCRIBE:
  The `UNSUBSCRIBE` command removes an existing subscription, so that the client no longer receives messages from that destination.

```
UNSUBSCRIBE
id:78

^@
```

  The `UNSUBSCRIBE` should contain a single header, `id`, which is the subscription ID supplied to the server with the `SUBSCRIBE` frame in the header with the same name.
  The body of the frame should be empty.

- DISCONNECT: The `DISCONNECT` command declares to the server that the client wants to disconnect from it.

```
DISCONNECT
receipt:77

^@
```

  The `DISCONNECT` should contain a single header, `receipt`, which contains a the `recipt-id` the client expects on the receipt returned by the server. This number should be generated uniquely by the client.
  The body of the frame should be empty.

  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. For example, the one shown above.
2. Wait for the `RECEIPT` frame response to the `DISCONNECT`. For example:

```
RECEIPT
receipt-id:77

^@
```

3. close the socket.

   This is graceful since after receiving the response, the client can be sure that every message he sent (barring packet losses, which is a subject not covered in this course) was received and processed by the server, and thus it can close its socket and no messages will be lost.

The `receipt` header:

As mentioned before, the `receipt` header can be added to ANY client frame which requires a response. The `DISCONNECT` frame MUST contain it, but it is not unique in that regard. We specify, in the client implementation section, some cases in which you will be required to get a receipt on frames, so you will have to use this header.
In addition, you can decide to use the `receipt` header discriminately, for ANY frame instance you send, if you wish to receive a `RECEIPT` frame for it from the server (This could be useful for debugging your server-client communication).

3 Implementation Details

3.1 General Guidelines

- The server should be written in Java. The client should be written in C++. Both should be tested on Linux installed at CS computer labs or the Docker.
- 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.

s

3.2 Server

A '.jar' for a solved server-side: `StompServer.jar` is provided in the `/server /target` folder so you can test the client before completely implementing the server side.
In order to run it please make sure you change the execute flag on with `chmod +x server-1.0.jar`.
Then run the following line in the `/server/target` folder: `java -jar StompServer .jar <port number: 7777> <reactor / tpc>`,
for example: `java -jar StompServer.jar 7777 tpc`.
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 template. You are also provided with 3 new or changed interfaces:

- `Connections<T>`

  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<T>`

  This interface replaces the `MessagingProtocol` interface. It exists to support p2p (peer-to-peer) messaging via the Connections interface. It contains 3 functions:

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

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

- `void process(T 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 **TPC** 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
4. Create an implementation of the `StompMessagingProtocol` interface according to the specification in the previous subsection.

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 for 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 distributed to the appropriate topic.
- In the server side, you should start by implementing the protocol and the encoder and decoder. In general, what you need to do in the server side is implement the protocol and encoder decoder for STOMP so that they can serialize and process the messages from the client, and use the Connections object as mentioned above to send the frames you need to send to the clients you need to send them to.
- The Connections object should be the used for sending messages to different clients from the protocol. It has 2 main sending methods you need to implement, one for sending a message to a single client and one for sending a message to a topic, that is, to a list of clients that's subscribed to that topic. In addition, we tasked you with implementing the send() method in the connection handlers. This method should be called from the send() methods of the Connection objects in order to use the connection handler to send messages to the clients they represent.
- The encoder-decoder implementations do not need to be generic on T, they should be implementations on some type (of your choosing) of the generic interfaces. In each of the examples we gave you, the classes that hold and use the protocol and encoder-decoder are referring to the generic interfaces, but the implementations of those interfaces are on a specific type.
- When a user disconnects from the system, the server should keep the username and the password so that a client can log into it at a later time. **The subscriptions, however, should be deleted for this user.**
- **You may change the interfaces you are required to implement as you see fit. That is, you can add methods or change existing method signatures. That being said, don't move any of the protocol logic out of the protocol. (Thus, you are not required to use the send() method with channel if it does not work with your implementation).**
- You can assume the events in the json will appear in the order of their time.

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

A '.exe' for a solved client-side: `StompESClient` is provided in the `bin` folder so you can test the server before completely implementing the client side.
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 the keyboard while the other should read from the socket, please read how to create a lock in C++ with mutex (lock) and how to initialize a thread in C++. (tip: use *#include < thread >*, *#include < mutex >*). We advise you to visit this page and see the full C++ multithreading example - just search for the section **"A Complete C++ Program For Multithreading"** inside this page.
You can see the EchoClient for socket usage in C++.
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 (terminal). The required commands are defined below, and you need to implement all of them as the requirements specify. Your client will need to translate the keyboard commands it receives to local behavior and network messages (frames) to implement the desired behavior.
The Client directory should contain a `src/`, `include/` and `bin/` subdirectories and a `makefile` as shown in class. The output executable for the client should be named **StompEMIClient** and should reside in the bin folder after calling `make`.

Note: the C++ EchoClient as it is in the assignment template will work with the EchoServer provided in the Java code. You can test them together to understand how to implement the STOMP server and client.

3.3.1 The STOMP Emergency Messaging Informer

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.

3.3.2 Client commands for all users

For any command below requiring a `channel_name` input: `channel_name` for an emergency channel of a topic, for example, Fire Department

```
<channel_name> (e.g. fire_dept)
```

A channel (topic) is created in the server the first time a client tries to subscribe to it.
Client commands:

- 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".
    - Client already logged in: If the client has already logged into a server you should not attempt to log in again. The client should simply print "The client is already logged in, log out before trying again".
    - 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".

Example:

  - Command: `login 1.1.1.1:2000 meni films`
  - Frame sent:

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

^@
```

  - Frame received:

```
CONNECTED
version:1.2

^@
```

- Join Emergency Channel command
  - Structure: `join {channel_name}`
  - For this command a `SUBSCRIBE` frame is sent to the `{channel_name}` topic.
  - As a result, a `RECIEPT` will be returned to the client. A message "Joined channel `{channel_name}`" will be displayed on the screen.
  - From now on, any message received by the client from `{channel_name}` should be parsed and used to update the information of the emergency channel as specified in the emergency events section. As stated in the `report` command specification, each report will contain the name of the reporter, and you should save reports from different users separately.

Example:

    - Command: `join fire_dept`
    - Frame sent:

```
SUBSCRIBE
destination:/fire_dept
id:17
receipt:73

^@
```

    - Frame Received:

```
RECEIPT
receipt-id:73

^@
```

- Exit Emergency Channel command
    - Structure: `exit {channel_name}`
    - For this command an `UNSUBSCRIBE` frame is sent to the `{channel_name}` topic.
    - As a result, a `RECIEPT` will be returned to the client. A message ”Exited channel `{channel_name}`” will be displayed on the screen.

  Example:

    - Command: `exit fire_dept`
    - Frame sent:

```
UNSUBSCRIBE
id:17
receipt:82

^@
```

    - Frame received:

```
RECEIPT
receipt-id:82

^@
```

- Report to channel command
    - Structure: `report {file}`
    - For this command, the client will do the following:

1. Read the provided `{file}` and parse the channel name and events it contains (more on the file format in the emergency event section).

2. Save each event on the client as an emergency update reported by the current logged-in `user`. You should save the events ordered by the time specified in them, as you will need to summarize them in that order.

3. Send a `{SEND}` frame for each emergency event to the `{channel_name }` topic (which, as mentioned, should be parsed from within the file), containing all the information of the emergency event in its body, as well as the name of the `{user}`.
   An example of a `SEND` frame containing a report:

```
SEND
destination:/police

user: meni
city: Liberty City
event name: Grand Theft Auto
date time: 1762966800
general information:
    active: true
    forces_arrival_at_scene: false
description:
Pink Lampadati Felon with license plate "STOL3N1
   ". White male 6'2 with black baseball hat.
^@
```

```
SEND
destination:/fire_dept

user: joe
city: Liberty City
event name: Fire
date time: 1773279900
general information:
    active: true
    forces_arrival_at_scene: true
description:
A gas pipe leak caused a fire at the fabric
   factory on 23rd Street and 2nd Avenue.
^@
```

   You should format the body of your reports as in the example above. A client receiving such a message will have to parse the information of the emergency event from the body.

- You can decide to save all the events and then send them one by one or send each event right after saving it.

  - The specification on the format of the events, the events file, and some information on how to save them is in the **Emergency Event** section.

Example: (with the `events1_partial.json` file that you received in your assignment)

  - Command: `report events1_partial.json`
  - Frame sent:

```
SEND
destination:/police

user: meni
city: Liberty City
event name: Grand Theft Auto
date time: 1762966800
general information:
    active: true
    forces arrival at scene: false
description:
Pink Lampadati Felon with license plate \"STOL3N1\".
     White male 6 2 \" with black baseball hat.
^@
```

  - Frame sent:

```
SEND
destination:/police

user: meni
city: Liberty City
event name: Burglary
date time: 1773279900
general information:
    active: true
    forces arrival at scene: true
description:
Suspect broke into a residence through a back window
    . Described as wearing a gray hoodie and blue
    jeans. Seen fleeing on foot towards Oak Street.
^@
```

- Summarize emergency channel command
  - Structure: `summary {channel_name} {user} {file}`
  - For this command the client will print the emergency updates it got from `{user}` for `{channel_name}` into the provided `{file}`.
    The print format is as follows:

```
Channel <channel_name>
Stats:
Total: <sum of total reports>
active: <# of 'true' active>
forces arrival at scene: <# of 'true'
   forces_arrival_at_scene>

Event Reports:

Report_1:
    city: <city>
    date time: <date_time_string>
    event name: <event_name>
    summary: <summary of description>

    ...

Report_N:
    city: <city>
    date time: <date_time_string>
    event name: <event_name>
    summary: <summary of description>
```

`Stats` should be a brief summary, you can assume that the event won't change between false and true (which is not as in real life system). `Reports` should be sorted by `date_time` by time and after that by `event_name` lexicographically. `Summary` should include the first 27 chars of the original description. If there are more, the summary should end with "..." (30 chars with the "...").

  - date_time MUST be converted to a **date time string**, e.g "29/12/24 22:15", and for that you can create a method epoch_to_date().
  - If `{file}` doesn't exist, create it. Otherwise, write over its content.
  - Note that `{user}` can be the clients current active user. This should not cause a problem for this command since the client is saving every emergency event it sends.

- Logout Command
  - Structure: `logout`
  - This command tells the client that the user wants to log out from the server. 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.

Example:

- Command: `logout`
- Frame sent:

```
DISCONNECT
receipt:113

^@
```

- Frame received:

```
RECEIPT
receipt-id:113

^@
```

3.4 Emergency Event

A **Emergency Event** is the format clients use to report about the events. Each emergency event has the following properties:

- `event name` - The name of the emergency event. Does not have to be unique to the event. You will not need to extract any information from this property, just to show it when reporting on the emergency.
- `description` - A description of the event. Can be anything, again, you will not need to extract information from this, just to save and display it in the summary, only need to be truncated.
- `date time` - The time of the report, in `DD/MM/YYYY_HH:MM`, when the event occurred. This will be used to keep the order of the events reported on the event. Two events can indeed have the same `date time` property value, as described below.
- `forces_arrival_update` a boolean field, see `summary` for more details.
- `active` a boolean field, see `summary` for more details.

Assume all interested clients will join the Emergency channel before reporting on those events begins, and no client will join midway through the reporting.

Your client will receive emergency events to report from a JSON file. We provide a parser for emergency event files to C++ HashMap. The parser is given in the files `event.h` and `event.cpp` along with the class `Event`. To use the parser, simply call the `parseEventsFile(std::string json_path)` function with a path to an events file JSON as the argument. The parser returns a struct containing a vector containing the parsed events.
An example of the usage of the parser:

`names_and_events nne = parseEventsFile("data/events1_partial.json")` An example of a emergency event in JSON format:

```
{
    "event_name": "Grand Theft Auto",
    "date_time": "1762966800",
    "description": "Pink Lampadati Felon with license plate
   \"STOL3N1\". White male 6 2 \" with black baseball hat
   .",
    "general_information": {
        "active": true,
        "forces_arrival_at_scene": false
    }
},
```

An example of a emergancy events JSON file: (the `events1_partial.json` file provided in the template)

```
{
    "channel_name": "police",
    "events": [
        {
            "event_name": "Grand Theft Auto",
            "city": "Liberty City",
            "date_time": "1762966800",
            "description": "Pink Lampadati Felon with
   license plate \"STOL3N1\". White male 6 2 \" with black
   baseball hat.",
            "general_information": {
                "active": true,
                "forces_arrival_at_scene": false
            }
        },
        {
            "event_name": "Burglary",
            "city": "Raccoon City",
            "date_time": "1773279900",
            "description": "Suspect broke into a residence
   through a back window. Described as wearing a gray hoodie
    and blue jeans. Seen fleeing on foot towards Oak Street
   .",
            "general_information": {
                "active": true,
                "forces_arrival_at_scene": true
            }
        },
    ]
}
```

The emergency events file contains the events to report on in a list corresponding with the property `events`.

4 Bonus Points: Working With Git

Working with Git is an essential skill in many industry and academic projects. Git is a version control software that helps you manage and track the progress of files in your project, especially when collaborating with a development team. In Git, a project is referred to as a repository (or *repo*), and saving new changes to the *repo* is done using *commit* and *push* commands.

Git is installed locally on your computer (in this case, it should be installed within the Docker container, which we already included in the). Additionally, you can use cloud-based platforms to host your *repo* remotely. One popular free option is GitHub. If you decide to create a Git *repo* for this bonus, please ensure that you set the *repo* to **Private** on GitHub so others cannot copy your work.

You can earn up to **5 bonus points** on this assignment by using Git effectively. Since Git is not part of the course material and has not been covered in class, **this bonus task is entirely optional, and no questions regarding it will be answered.**

For those interested in learning Git, here are some helpful resources:

- Learning Git
- Git Tutorial
- GitHub

How to earn the bonus points?

- **3-point bonus**: Create a Git repository and use it to work on this assignment. Show consistent progress with multiple meaningful commits (not just a single final commit).
- **2-point bonus**: To qualify for this additional bonus, your commit messages must be clear and descriptive. For example, a good commit message could be: *"Fixed a bug in function 'f' in 'txt1.cpp'"*.
  Avoid vague messages like ***"did some work"***, ***"bug"***, ***"fix"***, ***"work"***, ***"progress"***. Two examples of bad and good commits links:
  link1
  link2