SPL231: Assignment 3 instructions

Hedi Zisling, Roee Weiss Lipshitz

December 23, 2022

1 General Description

The world cup is upon us, and you want to stay updated. Thus, in this assignment, you will implement a "community-led" world cup update subscription service. Users can subscribe to a game channel and report and receive reports about the game to and from the other subscribed users.

For the above-mentioned purpose, 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 **Java** and will support both **Thread-Per-Client (TPC)** 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 STOMP 'Simple-Text-Oriented-Messaging-Protocol'.

In order to get you started, we supply a few examples for different protocols and clients, the most complete ones being **newsfeed** and **echo**, we recommend you go over them. Specifically, note how they can use both **TPC** or **Reactor** server implementations.

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 response 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` fame 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:
-----
MESSAGE
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.

If the client has no subscription with `id`,

```
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 bodey 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 VM.
- 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, Stampy, 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 template. 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 p2p (peer-to-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 **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.

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 the keyboard while the other should read from the socket.
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 **StompWCIClient** 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 World Cup 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 `game_name` input: `game_name` for a game between some Team A and some Team B should always be of the form

`<team_a_name>_<Team_b_name>`

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 Game Channel command
  - Structure: `join {game_name}`
  - For this command a `SUBSCRIBE` frame is sent to the `{game_name}` topic.
  - As a result, a `RECIEPT` will be returned to the client. A message "Joined channel `{game_name}`" will be displayed on the screen.
  - From now on, any message received by the client from `{game_name}` should be parsed and used to update the information of the game as specified in the game 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 germany_spain`
  - Frame sent:

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

^@
```

  - Frame Received:

```
RECEIPT
receipt-id:73

^@
```

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

  Example:
  - Command: `exit germany_spain`
  - 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 game name and events it contains (more on the file format in the game event section).
    2. Save each event on the client as a game 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 in the `summary` command.
    3. Send a `{SEND}` frame for each game event to the `{game_name}` topic (which, as mentioned, should be parsed from within the file), containing all the information of the game event in its body,

as well as the name of the `{user}`.
An example of a `SEND` frame containing a report:

```
SEND
destination:/spain_japan

user: meni
team a: spain
team b: japan
event name: kickoff
time: 0
general game updates:
    active: true
    before halftime: true
team a updates:
    a: b
    c: d
    e: f
team b updates:
    a: b
    c: d
    e: f
description:
And we're off!
^@
```

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 game 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 game events, the game events file, and some information on how to save them is in the **Game event** section.

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

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

```
SEND
destination:/germany_japan

user: meni
team a: germany
team b: japan
event name: kickoff
time: 0
```

```
general game updates:
    active: true
    before halftime: true
team a updates:
team b updates:
description:
The game has started! What an exciting evening!
^@
```

- Frame sent:

```
SEND
destination:/germany_japan

user: meni
event name: goal!!!!
time: 1980
general game updates:
team a updates:
    goals: 1
    possession: 90%
team b updates:
    possession: 10%
description:
"GOOOAAALLL!!! Germany lead!!! Gundogan finally has
success in the box as he steps up to take the
penalty, sends Gonda the wrong way, and slots the
ball into the left-hand corner to put Germany 1-0
up! A needless penalty to concede from Japan's point
of view, and after a bright start, the Samurai Blues
trail!"
^@
```

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

```
<team_a_name> vs <team_b_name>
Game stats:
General stats:
<stat_name1>: <stat_val1>
<stat_name2>: <stat_val2>
            ...

<team_a_name> stats:
<stat_name1>: <stat_val1>
<stat_name2>: <stat_val2>
```

```
            ...

<team_b_name> stats:
<stat_name1>: <stat_val1>
<stat_name2>: <stat_val2>
            ...

Game event reports:
<game_event_time1> - <game_event_name1>:

<game_event_description1>


<game_event_time2> - <game_event_name2>:

<game_event_description2>
        ...
```

The game event reports should be printed in the order that they happened in the game, and the stats should be printed ordered lexicographically by their name. More on game events and game stats in the Game Event section.

- 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 game 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 Game event

A **game event** is the format clients use to report about the game. Each game event has the following properties:

- `event name` - The name of the game 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 game.
- `description` - A description of the game event. Can be anything, again, you will not need to extract information from this, just to save and display it in the game summary.
- `time` - The time in the game, in **seconds**, when the event occurred. This will be used to keep the order of the events reported on the game. You can assume a game will not have 2 events reported at the same time, however, two events can indeed have the same `time` property value, as described below.
  Note: since game halves can have a time extension, the time before the half can exceed 45 minutes (when translated to minutes), thus a game event can occur after another game event with a higher time. For example, if a goal was scored 1 minute into the time extension of the first half, the game event reporting it will have a time of 2760(seconds) = 45+1(minutes). However, the game event reporting the beginning of the second half will have a time of 2700(seconds) = 45+1(minutes).
  The above-mentioned can cause a problem with saving the game events in the correct order, which can be solved by keeping a flag noting whether the halftime event had occurred yet. More on this in the `general game updates` property.
- Game updates properties:
  - `general game updates` - Any stat updates on the game that is not related to a particular team will be listed under this property.
    Special updates to be listed under this property:
    - `active` - States if that the game is active, will appear with the first game event with a value of `true`, and with the last game event for this game with a value of `false`.
    - `before halftime` - states if the events are occurring before or after halftime. The first game event will have this update with the

value `true`, and a game event reporting the start of the second half will have this update with the value `false`. This special update is related to the problem explained in the `time` property, and will allow you to keep track of whether the events are occurring before or after halftime.

These two are the only game updates that need to be parsed for information as they will give the client information about how to deal with updates about the game.
Assume all interested clients will join the game channel before the reporting on that game started, and no client will join in the middle of the reporting.

- `team a updates` - Any stat updates related to team a, such as ball possession, goals, etc.
- `team b updates` - The same as for team a, but for team b.

You can assume game updates (apart from the special ones) will have a value of string and will be accumulative. Thus, updating the stats of the game inside the client should merely be an act of saving the new string as the existing stat name. This is assuming the stat is already being tracked, that is, an event with an update on this stat was read from the file. If the stat is not tracked yet, just add it to the stats tracked for this game with the value you got for it from the game event.

Your client will receive game events to report from a JSON file. We provide a parser for game 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 the names of both teams as well as 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 game event in JSON format:

```
{
    "event name": "kickoff",
    "time": 0,
    "general game updates": {
        "active": true,
        "before halftime": true
    },
    "team a updates": {},
    "team b updates": {},
    "description": "The game has started! What an
                    exciting evening!"
}
```

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

```
{
    "team a": "Germany",
    "team b": "Japan",
    "events": [
        {
            "event name": "kickoff",
            "time": 0,
            "general game updates": {
                "active": true,
                "before halftime": true
            },
            "team a updates": {},
            "team b updates": {},
            "description": "The game has started! What
                            an exciting evening!"
        },
        {
            "event name": "goal!!!!",
            "time": 1980,
            "general game updates": {},
            "team a updates": {
                "goals": "1",
                "possession": "90%"
            },
            "team b updates": {
                "possession": "10%"
            },
            "description": "GOOOAAALLL!!! Germany lead!!!
                            Gundogan finally has success in
                            the box as he steps up to take
                            the penalty, sends Gonda the
                            wrong way, and slots the ball
                            into the left-hand corner to
                            put Germany 1-0 up! A needless
                            penalty to concede from Japan's
                            point of view, and after a
                            bright start, the Samurai Blues
                            trail!"
        }
    ]
}
```

As you can see, you can determine the `game_name` reported on in the game events file from the properties `team a` and `team b`. The game events file also contains the events to report on in a list corresponding with the property `events`.