Distributed Systems
COMP90015 2016 SM1
Project 1 - Multi-server Activity Stream

Multi-server system

- We will build a simple multi-server system for broadcasting activity objects between a number of clients. The multi-server system will:
  - load balance client requests over the servers, using a redirection mechanism to ask clients to reconnect to another server.
  - allow clients to register a username and secret, that can act as an authentication mechanism. Clients can login and logout as either `anonymous` or using a username/secret pair.
  - allow clients to broadcast an activity object to all other clients connected at the time

The system was envisioned to allow clients to broadcast "activity stream objects" to each other. The activity stream format is defined at http://activitystrea.ms/

However, processing activity stream objects is beyond the scope of the project and only a simple process is required in this project (to authenticate the user who broadcast the object).

Achitecture

- The multi-server system will form a tree, as demonstrated in lectures.
  - Skeleton code will be provided that will handle all of the socket connections to do this, as well as command line argument parsing and logging for diagnostic output.
- Each client will connect to a single server, and may be redirected to another server as part of the process.

Interaction

- All communication will be via TCP sockets.
- The architecture allows a simple broadcast communication between servers, so that a message from a server can be received by all other servers without duplication. This is a network overlay.
- Each client communicates with a selected server.
- All messages will be sent as JSON objects with a newline to delimit them, using UTF8.
- A mixture of request and request/reply protocols are used.
- Activity stream objects are also JSON objects and are the "payload" of some messages in the mult-server system.

Failure model

- Servers never crash once started and they never quit. This assumption is unrealistic but it simplifies the project considerably.
- However no server should be explicitly trusted to send correct messages, it should be assumed that message corruption may occurr and that messages may not contain correct information.

- Clients may quit at any time.
- Again, clients should not be explicitly trusted, it should be assumed that messages received from them could be corrupted and that messages may not contain correct information.

The general rule is that if any received message on a given connection is corrupted or contains incorrect information (e.g. a missing field that is otherwise required to process the message) then a response will be sent to indicate this and the connection will be closed.

When coding your server and client, consider the case that your server may be connected to by somebody elses server/client. You need to program defensively to handle all the cases that could occurr.

AUTHENTICATE

Sent from one server to another always and only as the first message when connecting.

Example:

```
{
    "command" : "AUTHENTICATE",
    "secret" : "fmnmpp3ai91qb3gc2bvs14g3ue"
}
```

Receiver replies with:

- `AUTHENTICATION_FAIL` if the secret is incorrect
- `INVALID_MESSAGE` if anything is incorrect about the message, or if the server had already successfully authenticated

No reply if the authentication succeeded.

If anything other than authentication succeeded, then the connection is closed immediately after sending the response.

INVALID_MESSAGE

A general message used as a reply if there is anything incorrect about the message that was received. This can be used by both clients and servers.

Examples:

```
{
    "command" : "INVALID_MESSAGE",
    "info" : "the received message did not contain a command"
}

{
    "command" : "INVALID_MESSAGE",
    "info" : "JSON parse error while parsing message"
}
```

The exact content of the `info` field is not specified, but should be something that indicates clearly why the message was deemed invalid.

After sending an `INVALID_MESSAGE` the sender will close the connection. Receivers of such a message must therefore close the connection as well.

AUTHTENTICATION_FAIL

Sent by the server when either a server fails to authenticate or when a client fails to login (see later).

Example:

```
{
    "command" : "AUTHENTICATION_FAIL",
    "info" : "the supplied secret is incorrect: fmnmpp3ai91qb3gc2bvs14g3ue"
}
```

After sending an `AUTHENTICATION_FAIL` the server will close the connection.

A client or server that receives `AUTHENTICATION_FAIL` must similarly close the connection.

LOGIN

Sent from a client to a server.

Example:

```
{
    "command" : "LOGIN",
    "username" : "aaron",
    "secret" : "fmnmpp3ai91qb3gc2bvs14g3ue"
}
```

A `username` must be present. The value of `username` may be `anonymous` in which case no `secret` field should be given (it should be ignored). Otherwise a `secret` must be given.

The server replies with:

- `LOGIN_SUCCESS` only if the server recognizes the combination of `username` and `secret` in its local storage.
- `LOGIN_FAILED` in cases where the secret does not match that in the local storage, for the supplied username, or when the username is not found (not registered, see later).
- `INVALID_MESSAGE` in any case where the message is incorrect

LOGIN_SUCCESS

Sent from server to client to indicate that the client successfully logged in.

Example:

```
{
    "command" : "LOGIN_SUCCESS",
    "info" : "logged in as user aaron"
}
```

The server will follow up a `LOGIN_SUCCESS` message with a `REDIRECT` message if the server knows of any other server with a load at least 2 clients less than its own.

REDIRECT

Sent from a server to a client, to request the client to reconnect to a new server.

Example:

```
{
    "command" : "REDIRECT",
    "hostname" : "123.456.78.9",
    "port" : 1234
}
```

After sending a `REDIRECT` message the server will close the connection.

After the client receives a `REDIRECT` message it must close the connection and make a new connection to the system, presumably to the server as given in the message. The client starts the protocol afresh.

LOGIN_FAILED

Sent from server to client to indicate that the client did not successfully log in.

Example:

```
{
    "command" : "LOGIN_FAILED",
    "info" : "attempt to login with wrong secret"
}
```

After sending a `LOGIN_FAILED` the server will close the connection.

LOGOUT

Sent from client to server to indicate that the client is closing the connection.

Example:

```
{
    "command" : "LOGOUT"
}
```

The client will close the connection after sending. The server will close the connection after receiving.

ACTIVITY_MESSAGE

Sent from client to server when publishing an activity object.

Example:

```
{
    "command" : "ACTIVITY_MESSAGE",
    "username" : "aaron",
    "secret" : "fmnmpp3ai91qb3gc2bvs14g3ue",
    "activity" : { ... }
}
```

The actual activity object is not shown above.

The server will respond with:

- `AUTHENTICATION_FAIL` if the username is not `anonymous` or if the username and secret do not match the logged in the user, or if the user has not logged in yet. After sending such a message the connection is closed.
- `INVALID_MESSAGE` if the message is incorrect in any way, and close the connection.

If the request succeeds then the server will broadcast the activity object to all servers and they in turn will broadcast to all connected clients. The activity will be processed (see later) before being broadcast.

SERVER_ANNOUNCE

Broadcast from every server to every other server (between servers only) on a regular basis (once every 5 seconds by default).

Example:

```
{
    "command" : "SERVER_ANNOUNCE",
    "id" : "fmnmpp3ai91qb3gc2bvs14g3ue",
    "load" : 5,
    "hostname" : "128.250.13.46",
    "port" : 3570
}
```

Servers respond with an `INVALID_MESSAGE` if the message is incorrect or if the message is received from an unauthenticated server (i.e. on a connection that has not yet authenticated with the server secret). Then the server will close connection.

The `id` is unique value that each server chooses for itself and maintains constant throughout operation. The `load` is the number of clients currently connected to the server. The `hostname` and `port` and how to connect to the server.

ACTIVITY_BROADCAST

Message broadcast between servers, and from each server to its clients, that contains a processed activity object.

Example:

```
{
    "command" : "ACTIVITY_BROADCAST",
    "activity" :  { ... }
}
```

Servers respond with an `INVALID_MESSAGE` if the message is incorrect or if the message is received from an unauthenticated server (i.e. on a connection that has not yet authenticated with the server secret). Then the server will close connection.

REGISTER

Sent from client to server when the client wishes to register a new username.

Example:

```
{
    "command" : "REGISTER",
    "username" : "aaron",
    "secret" : "fmnmpp3ai91qb3gc2bvs14g3ue"
}
```

The client selects the secret that it wishes to register with.

The server replies with:

- `REGISTER_FAILED` if the username is already known (registered) by any server in the system. Connection is closed.
- `REGISTER_SUCCESS` if the username was not already known by any server in the system, and now the user can login using the username and secret.
- `INVALID_MESSAGE` if anything is incorrect about the message, or if receiving a `REGISTER` message from a client that has already logged in on this connection. Connection is closed by server.

REGISTER_FAILED

Sent by server to client to indicate that an attempt to register a username and client has failed.

Example:

```
{
    "command" : "REGISTER_FAILED",
    "info" : "aaron is already registered with the system"
}
```

The connection is closed by the server after sending this message.

REGISTER_SUCCESS

Sent by server to client to indicate that an attempt to register a username and client has succeeded.

Example:

```
{
    "command" : "REGISTER_SUCCESS",
    "info" : "register success for aaron"
}
```

LOCK_REQUEST

Broadcast from a server to all other servers (between servers only) to indicate that a client is trying to register a username with a given sercret.

Example:

```
{
    "command" : "LOCK_REQUEST",
    "username" : "aaron",
    "secret" : "fmnmpp3ai91qb3gc2bvs14g3ue"
}
```

A server that receives this message will:

- Broadcast a `LOCK_DENIED` to all other servers (between servers only) if the username is already known to the server with a different secret.
- Broadcast a `LOCK_ALLOWED` to all other servers (between servers only) if the username is not already known to the server. The server will record this username and secret pair in its local storage.
- Send an `INVALID_MESSAGE` if anything is incorrect about the message or if it receives a `LOCK_REQUEST` from an unauthenticated server (i.e. the sender has not authenticated with the server secret). The connection is closed.

Using the above mechanism, all servers currently in the system will learn about register attempts.

LOCK_DENIED

Broadcast from a server to all other servers (between servers only), if the server received a `LOCK_REQUEST` and to indicate that a username is already known, and the username and secret pair should not be registered.

Example:

```
{
    "command" : "LOCK_DENIED",
    "username" : "aaron",
    "secret" : "fmnmpp3ai91qb3gc2bvs14g3ue"
}
```

When a server receives this message, it will remove the username from its local storage only if the secret matches the associated secret in its local storage.

The server will:

- Send an `INVALID_MESSAGE` if anything is incorrect about the message or if it receives a `LOCK_DENIED` from an unauthenticated server (i.e. the sender has not authenticated with the server secret). The connection is closed.

LOCK_ALLOWED

Broadcast from a server to all other servers if the server received a LOCK_REQUEST and the username was not already known to the server in its local storage.

Example:

```
{
    "command" : "LOCK_ALLOWED",
    "username" : "aaron",
    "secret" : "fmnmpp3ai91qb3gc2bvs14g3ue"
}
```

The server will:

- Send an INVALID_MESSAGE if anything is incorrect about the message or if it receives a LOCK_ALLOWED from an unauthenticated server (i.e. the sender has not authenticated with the server secret). The connection is closed.

Successfully registering a user

The server that originally broadcast a LOCK_REQUEST, after having received a REGISTER from a client, will make a note of all server ids that it knows about at that time.

It will wait for LOCK_ALLOWED messages to be returned from all such servers. After having received a LOCK_ALLOWED message from all servers, the server will respond with a REGISTER_SUCCESS to the client.

If the server receives even a single LOCK_DENIED from anyone for the username secret pair then the server will send a REGISTER_FAILED message to the client and close the connection.

If the server itself already had a registered username for that user in its local storage, then there is no need to broadcast a LOCK_REQUEST but rather simply the server sends a REGISTER_FAILED to the client.

---

unknown commands

Respond always with an INVALID_MESSAGE and close the connection.

Processing an activity object

When a server receives an ACTIVITY_MESSAGE from a client, it will first process the activity object before broadcasting it.

To process the object it will add a single field to the object called authenticated_user.

The field will contain either anonymous if the client is logged in as anonymous or the field will contain the username of the logged in user.

The process will overwrite the contents of the field if it was already in the activity object.

Using this mechanism, a user identity is consistently represented in published objects.

Technical aspects

- Requires Java 1.7 or above.
- The client takes command line arguments when starting up, to indicate a possible username and secret, and to indicate the server to connect to.
- The server takes command line arguments when starting up, to indicate the local port number to bind to, and the remote port, hostname and secret to use if connecting to an existing server. The server will print its secret to the screen if it started alone.
- The client must be able to accept activity objects from the user and send them to the server, as well it must be able to receive activity objects from the server and display them to the user. Some simple GUI code is supplied to do this. If you complete the project early, you may work on the GUI to make a nicer application :-)
- Everyone should implement the same protocol, which means that clients and servers from different groups should interoperate fine.

Your Report

- Use 10pt font, double column, 1 inch margin all around.
- On the first page, clearly show your group's name and the names of all members in the group. Clearly show the login names with university emails as well. The members of the group MUST match the information entered into LMS.
- The report is aimed at addressing a number of questions discussed next. Have one section for in the report for each.
- Figures in the report, including examples of messages and protocol interaction, and any pseudo-code (that you may or may not use), are not counted as part of the word length guidelines.

Introduction

Write roughly 125 words to briefly describe in your own words:

- what the project was about
- what were the challenges that you faced
- what outcomes did you achieve

Server failure model

We assumed in the project that servers never fail or quit once started, though they are not trusted and so messages sent by servers may be corrupted or in error. The later part is quite prudent and usually the assumption made in practical systems. The former part is very unrealistic. Two other possible assumptions are:

- servers never crash, but they may quit gracefully, for example by broadcasting a quit message before quitting
- servers may crash at any time without warning

As well, we may assume a fail-stop module for servers, where a server that crashes or quits never starts up again, or we may assume that servers that crash or quit may eventually restart, at the same address and using the same port number.

Write roughly 375 words discussing the implications of the above revised assumptions:

- what problems would be encountered with the existing system
- what approaches could be used to overcome these problems
- what protocols/messages could be used
- give examples of messages, use interaction diagrams or show pseudo code to explain

Concurreny issues

For a small system, with only a couple of servers and a few clients, concurrency issues are unlikely to arise. However consider a system with hundreds of servers and thousands of clients. There are some aspects of the multi-server system that may require further thought to ensure that concurrency issues are properly handled. Concurrency issues may include things that go technically wrong, but may also include things that do not work as well as expected.

Maintaining that servers never crash or quit, consider all aspects of the protocol and determine where concurrency issues may arise.

In roughly 375 words describe:

- any concurrency issues that you identify, be specific with examples
- possible revisions to the protocol that may overcome these issues

Scalability

The multi-server system makes use of a broadcast to send messages throughout the system. This is used for both broadcasting activity objects and for broadcasting messages related to registering a new user.

In roughly 375 words describe:

- the scalability of this approach
    - ♦ what is the message complexity, i.e. how many messages are sent by the system in total for the reigstering of a new user
- alternative approaches that may improve the scalability

Submission

You need to submit the following via LMS:

- Your report in PDF format *only*.
- Your ActivityStreamerClient.jar and ActivityStreamerServer.jar files.
- Your source files in a .ZIP or .TAR archive *only*.

Submissions will be due at the end of Week 8. Only one member of the team needs to the submit.