TTM4100 Project

Project Description

For the second year, the KTN project is not a part of the common project. The reason for this is the challenges creating a implementation that actually will be used in the common project. However, there are many concepts in this project that can be applied in the common project as well.

In this project you will create a chat service, consisting of both a chat server and a chat client. The client will communicate with the server over a TCP connection.

The main purpose of this projects is to implement a *protocol*, so basically, we don't require any certain programming language. A protocol is by definiton a design that enables different systems, languages or endpoints communication with each other using a common, shared language.

However, since the curriculum is based on python, and the language is used throughout the course, the provided skeleton code is also written in python. It is also very likely that the course staff will be able to provide the best help and assistance if you implement the project using python.

The goal of this project is to have a practical and useful approach to network programming and modern concepts of programming with the use of application programming interfaces (API). Previously, the course have emphasised the importance of technical aspects of lower-level network programming. In the recent years, however, very good open source libraries have emerged, and we can focus on the fun parts og network programming.

As a final advice, we encourage you to make use of online documentation, google, and other online resources in addition to ask the student assistants for help. You will not find all you need in the course curriculum, and later on you will realise that you'll spend a lot of your time searching for good solution, not actually writing code. This applies both for studies and later in your professional career.

Through It's Learning you have access to a skeleton code that will be helpful as a guidance in the start phase of your implementation. This code is written in Python 2.7, so if you use another Python version or another programming language, you need to "translate" the skeleton code in order to use it.

Overall Requirements

You will implement a simple protocol for a command line interface (cli) chat client that communicates with a server backend. The client is supposed to be a "stupid" client and the logic will for the most part be placed server side. The communication between the server and clients will be using JSON[1][2][3] as the format of the communicated information. JSON has become the standard in moderns web applications through its heavy use within API's.

The client is supposed to handle input from the user, parse the input and send the payload to the server. The server will, in turn, handle the received payload and take the required actions. The server should be able to handle several clients.

---

[1] http://en.wikipedia.org/wiki/JSON

[2] http://www.json.org

[3] http://www.json.org/example

The goal is for you to implement a generic protocol. To test that, your client must communicate with other people's servers, and other students' clients must be able to communicate with your server. During the demonstration of the final program, the student assistants will test your code with their own implementation of both the server and client. If your code does not work against our implementation the solution will not be accepted.

Communication

The communication between the client and the server will be made possible through the use of sockets. A socket is one of the most fundamental technologies of computer networking, and even though sockets may seem to be a relatively new web phenomenon, socket technology has been employed for a long time.

If you use Python for your implementation of the chat, you will find that `socket` and `SocketServer` are two libraries that certainly will prove to be useful. The documentation for both `socket`[4] and `SocketServer`[5] are both available on Python's web pages.

To make the client able to both send and receive messages at the same time you need to apply threading. In the skeleton code, the MessageReceiver class inherits the Thread class. Python users will find more about threading in the documentation[6].

Client

The communication between the client and server must follow strict standards to allow for generic use.

The payload from a client to the server must be formatted as follows:

```
{
    'request': <request>,
    'content': <content>
}
```

The request describes what you are requesting of the server. The content is the requests argument (if the request requires input). If the request does not require any input, the content should contain `None`.

At least, the following requests, with the associated arguments must be supported:

`login <username>` - log in with the given username
`logout` - log out
`msg <message>` - send message
`names` - list users in chat
`help` - view help text

login <username> sends a request to login to the server

logout sends a request to log out and disconnect from the server

[4] https://docs.python.org/2/library/socket.html

[5] https://docs.python.org/2/library/socketserver.html

[6] https://docs.python.org/2/library/threading.html

msg <message> sends a message to the server that should be broadcasted to all connected clients

names should send a request to list all the usernames currently connected to the server

help sends a request to the server to receive a help text containing all requests supported by the server

Server

The server should handle all requests and send responses back to the client on the following format:

```
{
    'timestamp': <timestamp>,
    'sender': <username>,
    'response': <response>,
    'content': <content>
}
```

The different types of responses are: `error`, `info`, `history` and `message`. **Error** is an error message informing the client that something is wrong. **Info** is information from the server to the client. **History** is a list of messages that previously have been received by the server, and **message** is a message from one of the users connected to the server. When received by the client, all these responses should be formatted and presented to the user.

An important notice is that the only requests the server should accept when the client is not logged in are **login** and **help**. If the client is not logged in and sends a illegal request, the server should respond with an error message. At login, the client should receive the chat history, so that the newly logged in user can see what have been written in the chat in the time before logging in.

The server should also restrict the possible characters allowed in a username. The only accepted characters are the letters A-Z, a-z and numbers 0-9.

Notes and further work

The requirements listed above are the absolute *minimum* requirements necessary to get the project approved, and as noted earlier, both your client and server will be tested against a working implementation.

There are no requirements for persistence across sessions or server restarts, meaning that the chat and everything else can be in-memory during runtime. However, it is fairly easy to implement persistence using for instance mongodb or another non-relational database.

Other possible functions that may be implemented are:

- Moderators
  - Banning users
  - Removing messages
- Chat rooms
  - Creating chat rooms
  - Logging into and leave chat rooms

Deadlines and Deliverables

Deliverable 1

To be delivered by March 6th:

- Class diagrams
- Sequence diagrams for different use cases (e.g. login, send message and logout)
- A short textual description of your design

The goal here is for you to show that you understand the task and that you have a plan for solving it.

Deliverable 2

To be delivered March 27th:

- Your complete implementation of the chat client and server in a zip file
- An updated version of deliverable 1 so it matches the final implementation
- You should also demonstrate your chat for one of the student assistants at P15