NTNU
Norwegian University of
Science and Technology
Department of Telematics

TTM4100

Communication – Services and Networks

Project Description

The TTM4100 project consists of two parts: KTN1 – Design and KTN2 – Implementation. Both parts have a deliverable, and both deliverables must be approved in order to be qualified for taking the exam. In addition to the delivery, KTN2 also requires the project implementation to be demonstrated to the student/teaching assistants.

Deadline of submission KTN1: **10.03.2017**

Deadline of submission KTN2: **24.03.2017**

Deadline of KTN2 Demonstration: **24.03.2017**

Introduction

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

However, since the curriculum (and textbook) is based on Python (Python 2.7), and the language is used throughout the course, the provided skeleton code is also written in Python 2.7 and Python 3. 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 2.7.

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 has emphasized 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 of network programming.

As a final piece of 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 realize that you will spend a lot of your time searching for information, 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 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.

Groups

The project must be performed individually, or in groups of 2 students.

Requirements

You will implement a simple protocol for a command line interface 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 as the format of the communicated information. JSON has become the standard in modern web applications through its heavy use within API's.

The client is supposed to handle input from the user (through the keyboard), parse the input and send the payload to the server. The format of the payload is fixed (see next section). The server will, in turn, handle the received payload and take the required actions. The server must be able to handle several clients. After the server has performed the corresponding actions, it must send a response back to the client. Again, the format of these responses is fixed and detailed in the next section.

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

This section describes the communication between the client and the server. This is the main point that will enable use between your project and other people's projects. Hence, this section is a **strict standard**, and the following formats for client-server interaction must be implemented **as it is explained here**.

Remember, messages exchanged between the client and the server must be implemented with JSON.

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.

- Client

The payload from the client to the server (i.e. **all** messages from the client to the server) must be in the following format:

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

The request describes what you are requesting of the server. The content is the request's 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. The content is a string with the username.
logout sends a request to log out and disconnect from the server. The content is None.
msg <message> sends a message to the server that should be broadcasted to all connected clients. The content is a string with the message.
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.

Additionally, the client must receive the server responses (the format is defined in the next subsection) and present them to the client. 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.

Actually, the skeleton already contains all the necessary code to implement the threading, but understanding how it works will help you in both parts of the project (both KTN1 and KTN2). Python users will find more about threading in the documentation (see useful links section below).

- Server

The server should handle all requests and send responses back to the client always (i.e. **all** messages from the server to the client) on the following format:

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

The timestamp is the moment in which the server generates the response, it must be a **string**. The sender can be the server itself, or any of the logged in users. The different types of responses are: **error**, **info**, **message** and **history**. The content is the argument associated to each type of response (in this case, all responses have an associated content).

The following responses must be supported:
Error is an error response informing the client that something is wrong. The sender will always be the server. The content is a string detailing the error (e.g. 'Username taken').

Info is an informational response from the server to the client. Again, the sender will always be the server. The content is a string detailing the information (e.g. 'Login successful').

Message is a response which contains a message from one of the users connected to the server. The sender is the username of the client who originated the message. The content is a string with the message itself (e.g. 'Hello'). This response is broadcasted to all the connected clients, including the one who originated the message.
!!N.B. do not confuse **msg** (a request sent from the client to the server) with **message** (the response to that request from the server, which is sent to all connected clients). Both will have the same content though.

History is a list of all the **message** responses that the server has previously issued. The sender is the server. The content is a list that contains all the **message** responses (as JSON objects) that the sender has previously sent.

An important notice is that the only requests the server will accept when the client is not logged in are **login** and **help**. If the client is not logged in and sends an illegal request, the server must send back an **error** response.

Additionally, the server should also restrict the possible characters allowed in a username. When the server receives a **login** request, the server must check that the username only contains **characters that are the letters A-Z, a-z and numbers 0-9.** Also, if the login request is successful, 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.

Further Work (optional)

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 (KTN1)

To be delivered by **10.03.2017**:

- 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. Thus, we do not expect perfect UML-compliant class and sequence diagrams (you do not have to worry about it) or very detailed at this step. Hence, things like the difference between solid arrow heads and open arrow heads in a sequence diagram are not important to us.

The class diagrams should show the classes you plan to program, their main methods and attributes and how they are related. The sequence diagrams should show how the different objects will interact in a use case: which methods are called, which messages are send and in which order.

- Deliverable 2 (KTN2)

To be delivered March **24.03.2017**:

- (**You must also demonstrate your chat to one of the student assistants at P15**.)

- 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.

Useful Links

JSON:

- http://en.wikipedia.org/wiki/JSON
- http://www.json.org
- http://www.json.org/example
- https://docs.python.org/2/library/json.html

SOCKETS:

- https://docs.python.org/2/library/socket.html
- https://docs.python.org/2/library/socketserver.html (class ThreadedTCPServer)

THREADING

- https://docs.python.org/2/library/threading.html
- https://docs.python.org/2/library/thread.html#module-thread

PYTHON DATA STRUCTURES (Dictionaries and lists)

- https://docs.python.org/2/tutorial/datastructures.html

PYTHON REGULAR EXPRESSIONS

- https://docs.python.org/2/library/re.html

CLASS & SEQUENCE DIAGRAMS

- https://en.wikipedia.org/wiki/Class_diagram
- http://www.tutorialspoint.com/uml/uml_class_diagram.htm
- https://en.wikipedia.org/wiki/Sequence_diagram
- http://www.tutorialspoint.com/uml/uml_interaction_diagram.htm

Notes and additional hints

This section is mainly thought for first year students with little programming experience and it should give them some more ideas about how to handle the project. Students with more programming experience may find these hints not very useful, or they may make the project very easy. If you have considerable programming experience and are looking for a challenge, you may consider skipping this section.

- A short (20 min.) talk with additional help for the project will be given on Wednesday 22.02.2017, It will cover threading (conceptually and classes used by the skeleton), diagrams and JSON. The corresponding slides are on its learning, so it is not compulsory to attend this talk. It will **NOT** be video-recorded.

- Examples for creating payloads:

EXAMPLE 1:

Imagine the client has to send the request **login** to the server. The user input is:

>> login alvaro

Your client must parse this input, let's say to variables `req = 'login'` and `cont = 'alvaro'`. Then, the payload must be created in Python 2.7 as:

```
payload = {'request': req, 'content': cont}
```

This payload must now be encoded with JSON and sent to the server through the socket.

EXAMPLE 2:
Imagine the client has to send the request **logout** to the server. The user input is:
>> logout

Your client must parse this input (i.e. here, it should realize that the **content** is **None**), again to variables `req = 'logout'` and `cont = None`.

The payload is created as before, then encoded with JSON and finally sent through the socket.

- Payloads, as defined above, are dictionaries in Python. Also, when you decode a JSON object, you will get a dictionary in Python. Dictionaries are like Hashtables in Java: an unordered set of key: value pairs. Hence, if following EXAMPLE 1 above, printing `payload['request']` will give you `'login'`.

- Following the previous point: when the client (the same logic can be applied to the server) receives a response from the server, it has to check which type of response it is, in order to perform the corresponding actions. This can be implemented quite neatly with dictionaries. The client can have a dictionary of possible responses: the keys will be the possible responses (e.g. `'login', 'msg'),` the values will be the corresponding methods to be called when that response is received:

EXAMPLE 3 (included in the skeleton)
If `self.possible_responses` is a dictionary as described below (in this example, only error and info are given), and `payload` is the decoded JSON object received through the socket (i.e. a dictionary), the following code will check if `payload` is a possible response and will call the corresponding method passing the content of the payload as a parameter:

```
self.possible_responses = {
             'error': self.parse_error,
             'info': self.parse_info,
}
if payload['response'] in self.possible_responses:
            return self.possible_responses[payload['response']](payload)
```

Keep in mind that this is just a possible implementation, but is not a requirement in itself. You can disregard this part of the skeleton and implement it the way you like. However, dictionaries are quite powerful in Python, so it is a good example of what can be done with them and may be useful in other programming courses.

- A dictionary is also a good way to keep track of connected clients, with key the username and value the object that represent connected clients.

- When a client handles a **history** response, several JSON decodings are required: the first one is for the **history** response itself. Then each of the elements of the list in **'content'** are also a JSON object, so they must be decoded before presenting them to the user.

- Consider using time and datetime modules for the timestamps. Useful methods are fromtimestamp() and strftime().

- The re module in Python can be helpful in order to parse the user input, and also to check if the username uses only allowed characters. You will have to learn how to use match() and/or compile() and search().