1 Exercise 1: A Simple Messaging System for Healthcare Professionals

1.1 General Instructions

You will have the lab session of Week 3 to develop the exercise described in this document. During the session of Week 5, you will also have the opportunity to ask questions about it (despite being given another exercise to develop), before its submission deadline on Friday 03 March 18:00 GMT. You should submit the code you develop for this exercise via Blackboard.

It is important that you test all your lab work on one of the lab machines before submitting it, making sure that your code solution runs in the lab environment, assuming that your code will be tested in one of those machines. Although the machines in the labs of the Kilburn Building are all dual-boot (with Windows and Linux), in this course unit, we will be using Linux.

1.2 Getting Ready to Start

You are asked to build a very simple messaging system for health care professionals that will allow two users (e.g., doctors, nurses, etc.) to send and receive text messages about their work. It's going to have a client/server architecture and we have provided to you a simple implementation of the server as well as a Python module that makes talking to the server fairly easy.

Your task is to design and implement a ***protocol*** that will make the messaging system work. There exists a large number and variety in types of protocols for distributed systems, with examples including the following: (the commands found in) the HTTP protocol (`https://www.rfc-editor.org/rfc/rfc9110.html`), the POP3 email protocol (`http://www.faqs.org/rfcs/rfc1939.html`), the SMTP email protocol (`https://tools.ietf.org/html/rfc5321`), etc. Note, however, that your protocol needs to be specially designed to resolve the limitations and problems associated with the simple server given to you in this exercise, which will be explained in later sections of this document. In addition, do NOT make changes to the server code provided to you. Your solution should be encoded in the protocol you are going to develop, not in the server!

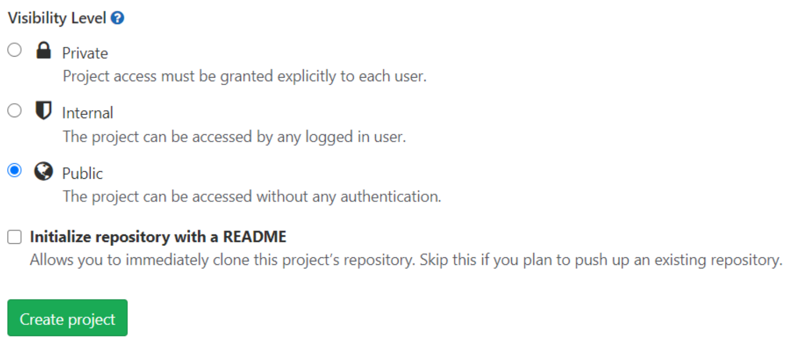
The purpose of this exercise is to get you thinking about some of the issues that arise with even the most basic distributed system, especially one that has to manage any kind of 'state'. Beware, we are not looking for very nice code or for you to demonstrate a deep understanding of the programming languages used here: the distributed computing principles are what really matters (though you will get penalised for terrible unreadable code, so do your best to keep it nicely structured and commented). It is also important that you know that the system you are going to build is going to be flawed in many ways. Some of these limitations you could work around, given enough time. Others are essentially insurmountable, and arise from the architecture and technology we have chosen for this lab. So we are not claiming that the architecture we have suggested here represents the best way of building such a system – far from it, in fact! As you are doing the exercise, try to think about what the limitations of this approach are, and how you would design a more realistic solution. We'll revisit these issues in a later exercise.

First of all, an important warning: We will give you some shell commands to type in. Please, do not make the common mistake of copying and pasting from this PDF file into the shell terminal and ending up with something that *looks like, but is not,* a tilde as the shell terminal (or, later on, the browser) understands it. If you copy and paste, do replace the character that merely looks like a tilde with a proper tilde that you enter in the shell terminal using the tilde key in your keyboard.

To start with, you will set up the "server" for messaging between two healthcare professionals. In order to do that, you will need to login to your Gitlab account:

```
https://gitlab.cs.man.ac.uk
```

Now, create a new project called “COMP28112_ex1” and make sure that your project’s visibility is set to *public* before you click on the “Create Project” button as shown in Figure 1.



Visibility Level

Private
Project access must be granted explicitly to each user.

Internal
The project can be accessed by any logged in user.

[✓ Public]
The project can be accessed without any authentication.

Initialize repository with a README
Allows you to immediately clone this project’s repository. Skip this if you plan to push up an existing repository.

Create project

Figure 1: Setting visibility of a Gitlab project

In the next step, download the `IMserver.php` and `im.py` files from the Blackboard page of this course unit.

Add `IMserver.php` to your “COMP28112_ex1” project in Gitlab.

The `IMserver.php` file, as the name suggests, is going to act as your server. It is very important to ensure that your gitlab project “COMP28112_ex1” is made *publicly* available so that it can be accessed by all.

First, open a web browser and login to your Web Dashboard account:

```
https://web.cs.manchester.ac.uk/dashboard/login
```

For help and more information on how to deploy a Gitlab project, please follow this link:

```
https://wiki.cs.manchester.ac.uk/index.php/Web_Dashboard/Website
```

1.3 Task 1.1: Testing the Code Using a Browser

Once you have deployed your Gitlab project successfully using Web Dashboard, confirm that you can access it via a web browser using the following template URL.

```
https://web.cs.manchester.ac.uk/username/COMP28112_ex1/IMserver.php
```

Here, and everywhere else in this manual, where you see `username` you must replace that string with your own username consisting of 8 alphanumeric characters (e.g. a12345zz),

You should get a page with just the heading ‘COMP28112 Server: Messaging System for Healthcare Professionals’ in it, in which case all is well (if there’s another line, after the heading, containing strange stuff, that’s okay too. . . it’ll become clear what that means shortly).

The server you have now set up acts as a simple state store: you can give it the name of a variable (called a ‘key’), and its value, and ask it to remember this for you; and you can later retrieve the value of that variable/key.

Let's experiment with it, just to confirm it's all working correctly.

Using your browser, enter the URL

```
https://web.cs.manchester.ac.uk/username/COMP28112_ex1/IMserver.php?action=set&k
ey=nurse1&value=hello
```

The browser will respond with a blank page, but should now have remembered that the value of the key 'nurse1' is 'hello'.

Look at

```
https://web.cs.manchester.ac.uk/username/COMP28112_ex1/IMserver.php
```

again. There should now be a line under the heading that looks something like:

```
a:1:{s:6:"nurse1";s:5:"hello";}
```

You should not worry about the `a:1` and `s:6` bits; they are likely to vary from case to case, and are just bits of internal implementation and really are not important. What matters is that you can see 'nurse1' and 'hello' in the list of things the server has remembered.

Try setting two more key/value pairs, and confirming that they are being remembered,
Let's make sure that we can retrieve the values sensibly. Put a key/value pair in to the server's store, then try something like

```
https://web.cs.manchester.ac.uk/username/COMP28112_ex1/IMserver.php?action=get&k
ey=nurse1
```

to retrieve the them.

Remember: Replace username with your University username wherever it appears.

Your browser should respond with just the value you have put in there previously.

Finally, try something like

```
https://web.cs.manchester.ac.uk/username/COMP28112_ex1/IMserver.php?action=unse
t&key=nurse1
```

to make sure that you can unset them too. Unsetting a key removes it from the state store. Confirm it by looking at:

```
https://web.cs.manchester.ac.uk/username/COMP28112_ex1/IMserver.php
```

If everything is going to plan so far, you should now have a basic state store of your own – you can give it any key/value pairs you like, and retrieve or unset them at a later date. The important thing for this exercise is that because this is being mediated by a web server, you can get and set these values from different programs; so you could set a value in one program, and get it back in another. This forms a basic infrastructure for the messaging system you are about to write.

1.4 Task 1.2: Testing the Code using Python

Before you start designing your protocol, there's one more bit of foundation work to do; let's make sure it's possible to talk to the server programmatically using Python.
In the first step, download the `im.py` file from the Blackboard page of this course unit.

Make yourself an `ex1` directory in `~/COMP28112` in your machine and then place `im.py` in it.

Now open a terminal and type:

```
 ~/COMP28112/ex1/
ls -a
```

You should get

```
 .  ..  im.py
```

on the terminal output. If this is the case, all is well and you can continue with the exercise.

Now, start up the Python interpreter by typing `python3` at a command prompt. Enter the following lines carefully, pressing return between each one (and making sure you replace `username` with your user name!)

```
import im
server= im.IMServerProxy('https://web.cs.manchester.ac.uk/username/COMP28112_ex1/
IMserver.php')
```

If you have done this correctly, Python shouldn't be reporting any errors. Line 1 tells Python to use the class that we have written for you in the `im.py` file; line 2 creates an instance of the IMServerProxy class, and tells it to use the PHP server that you set up in the previous task.

Now type:

```
server['pythonTest1'] = 'hello'
```

and then look at the contents of the server store using your browser; you should see that it's associated 'pythonTest1' with 'hello', exactly as though you'd done this via a URL as before (we'll look at how this Python operation is turned into a HTTP operation in a little while).

Try typing:

```
print(server['pythonTest1'])
```

and you should get back the 'hello' string you have just put in.

You can also unset an association.

Try typing:

```
del server['pythonTest1']
```

and then check the contents of the server store again using the `print` command to convince yourself that the association of 'pythonTest1' with 'hello' is no longer recorded in the store.

Just to convince yourself that this is acting as a simple distributed system, set one variable using a browser and the URL method, and once you have done that, get that variable in Python. Then try it the other way round. If you like, start up a second Python interpreter in another window (but from the same directory), and try setting a key/value pair in one, and retrieving it in the other (there is a whole Distributed System paradigm based on this simple idea called **Representational State Transfer** (or REST[1] for short).

Now we have a means of causing a 'central server' to remember a state set by one program, and retrieving it in a totally independent other program.

If you can find someone else who's got to this stage, you're welcome to spend a few minutes with them trying to talk to their server instead of your own - all you have to do is to substitute their username instead of yours in the URL or the parameter to the `im.IMServerProxy`; they can then set a key/value pair, which you can retrieve.

IMPORTANT: You MUST do this in co-operation with the other person. You MUST NOT UNDER ANY CIRCUMSTANCES mess around with someone else's server without their consent because it could seriously damage their labwork! If you are found to be disrupting someone else's work in this way, the matter will be taken very seriously (and because this is done via a webserver, all accesses are being logged!)

Before starting to write your messaging client properly, take a quick look at the `im.py` module that we have provided. This file imports the same 'urllib' module – this allows Python to 'speak http'. It then defines a class called `IMServer` that you've already used from the Python command line. The actual detail of the class is unimportant, and actually uses some fairly advanced Python concepts that we don't care about in this course. The only thing that matters here is that you can see the 'urlopen' instructions, which are simply reconstructing the same kind of `action=get&key=something` type URLs that you have already manually entered earlier. In this module, we wrap up the untidy business of creating and calling the URL as a Python 'dictionary'. Basically this means that when we do

```
server['myKey'] = 'something'
```

this module will create and GET the URL

```
https://web.cs.manchester.ac.uk/username/COMP28112_ex1/IMserver.php?action=set&k
ey=myKey&value=something
```

and similarly

```
print(server['myKey'])
```

will achieve the same as typing

```
https://webdev.cs.manchester.ac.uk/username/COMP28112_ex1/IMserver.php?action=ge
t&key=myKey
```

There are two more 'housekeeping' methods that you will find useful. The first of these is `keys()`, which will retrieve a list of existing keys from the server, e.g.

```
print(server.keys())
```

and the second is `clear()`, which deletes all the key/value pairs from the server, e.g.

```
server.clear()
```

[1]For more information about REST you can visit this link: `https://en.wikipedia.org/wiki/Representational_state_transfer`

1.5 Task 1.3: Designing a Simple Python Messaging Client

Now that the infrastructure is in place, all that remains is to create the messaging client! To keep things simple, we'll make this a messaging system that works for exactly two users at a time; not one, not three, but exactly and only two (note: this restriction does not need to be enforced by you in your code/design). We'll also assume that each user can type exactly one message, and then has to wait for the other user to reply before they can enter another message (note: your protocol needs to take this restriction into account).

At this point, move away from your keyboard and think for a while. The actual code you are going to write is not very complex, with the module we've given you doing all the messy stuff for you. What's needed is for you to come up with a sensible protocol reflecting your strategy for allowing two independent clients to exchange messages taking into account the restriction described above, and using only the functionality provided by your simple server.

Here are some hints to get you going: the basic functionality goes roughly along the lines of:

1. Wait for the user to type in a message.
2. Send a message to the server.
3. Wait for there to be a reply from the other user.
4. Fetch and display the reply.
5. Go back to step 1.

But there is at least one important thing to consider:

We are going to design a messaging system for healthcare professionals (e.g.,nurses, doctors, healthcare assistants, pharmacists, etc). In this context, let's assume that two doctors want to have a chat and discuss an important case. Obviously, if both doctors are waiting to send or receive a message at the same time, our system is instantly deadlocked and nothing will work. You will need to think of a way of starting one doctor off in the 'sending a message' state, and the other, in the 'waiting for a message' state. There are lots of ways of doing this, some more elegant than others. It is worth pointing out that there is an expectation for an elegant, well designed and well articulated solution that could be inspired from the material delivered in lectures.

1.5.1 Implementing the Messaging Client Code

Once you are convinced that you have designed your protocol it's time to write the code. Do this in a file called `imclient.py`; you'll need to submit this at the end of the exercise. Here are some handy bits of Python that you might need to use.

To get a line of input from the user, use something like:

```
myMessage = input('Type your message:  ')
```

To 'wait a while', import the 'time' module ('import time') and use

```
time.sleep(numberOfSeconds)
```

Remember you can always 'unset' variables manually using the URL method if your server gets in to a mess – if you're interested in knowing how the PHP code we have provided actually stores the key/value pairs, take a look at the source code.

1.6 Deliverables for Exercise 1 (Total marks: 5)

The deliverables for this exercise are:

- A working messaging client, written in Python, that allows a pair of users to alternate sending and receiving messages to and from one another, avoiding the system deadlocking. You should name this file `imclient_username.py`. Note that you should replace `username` with your username. (1 mark)
- A text description of your protocol, containing information about, for example, each command/term/restriction you proposed with its associated role in the protocol, as well as an explanation of how the protocol achieves synchronisation between two clients (i.e., the alternation between the sending and receiving of messages to and from one another, avoiding the system deadlocking). Note that this is marked completely independently of whether or not the submitted code runs correctly. In other words, if the description and/or explanation are/is incomplete, unclear, or senseless, marks will be lost even if the provided code runs perfectly. (3 marks)
- A text description of how to run your code and how it should be tested. In other words, describe how to install and run your code, as well as how to test it, by providing the order in which the commands of your protocol should be called to provide a complete demonstration of all the functionality provided within the protocol. (1 mark)

1.7 Submission instructions

You should submit your work via Blackboard. You are expected to submit a **zip folder** with name `comp28112_ex1_username.zip` containing your code with file name `imclient_username.py` and a *pdf* with file name `Report_username.pdf`. When naming the zip and pdf folder, don't forget to replace `username` with your own username consisting of 8 alphanumeric characters (e.g. a12345zz) before submitting your work.