MANCHESTER 1824
The University of Manchester

School of Computer Science

COMP18112: Fundamentals of Distributed Systems

Fundamentals of Distributed Systems

Steve Pettifer

steve.pettifer@manchester.ac.uk

@srp

January 2014

Contents

A Note on Plagiarism and Working Together

You must always work individually. You must only consult and discuss things with your friends once you have gained a thorough understanding of what constitutes academic malpractice. You are not allowed to copy solutions from anyone else, or to pass solutions to anyone else, in any form (conversation, paper, electronic media, etc.), unless otherwise authorised explicitly.

If you have not done so yet, make absolutely sure that you understand and abide by the guidance on plagiarism and cheating: `http://studentnet.cs.manchester.ac.uk/assessment/plagiarism.php` . You should remind yourself of, and be wise to follow, John Latham's thoughts on the issue of students working together. Here is a reminder (quoted, but very lightly adapted) for your convenience:

> *Many of you will be more than tempted to work together with your friends on the solutions to the problems. We have no objection to this in principle, but be aware of the fine line between working together and copying. We will not tolerate copying. If you do not actually do, on your own, with full understanding, every part you are supposed to do of each exercise that you get marked, then you have copied. Remember the requirement that you fully understand your work, and our reserved right to viva you on it without notice!*
>
> *When working together in an informal group of friends, pay special attention to the relative abilities of people in that group. The real point of laboratory work is not to get the answer, but to learn by doing it yourself, even if you actually get the wrong answer! If you find yourself always telling your friends the answers, then you are not much of a friend to them! You are holding them back and patronising them, but more to the point, you are undermining their learning. Equally, if you find yourself with a friend who keeps telling you the answers, don't be grateful! There is a good chance that he, or she, is simply trying to impress you. Don't be impressed!*
>
> *Everybody who works in an informal group should actually do the work themselves, individually. Such working together should be restricted to discussing ideas and getting the work off the ground. Anybody who cannot actually do the work, should get help from one of the demonstrators or supervisors. These people are experienced at helping you in such a way that you can do the work yourself, rather than just giving you the answer.*
>
> *In determining the line between help and plagiarism, a good rule of thumb to stick to is this: if you get help from another student, or give help to another student, make sure that the passage of information between you is at a much higher level of abstraction than the final answer, no matter what the medium is (soft copy, typed, written, spoken, etc.). So, for example, for exercises which involve you writing some program code (and which is not supposed to be team work!), never show your code, or your detailed pseudocode, to another student: that amount of "help" is cheating. Thus, for example, if you and a few friends develop a solution to a laboratory exercise together, you are, by definition, cheating.*
>
> *On the other hand, you can help each other as much as you like about things that you have not been asked to create, such as explaining the exercise question to each other, or explaining material that has been covered in lectures.*

Read and heed the above. To reiterate once more: we do not tolerate copying.

Exercise 1: A Simple Web Browser (1 lab session)

This exercise lasts for one lab session, and has a deadline at the end of it. When you've finished your work, you should `submit` it, and get a `labprint` copy on paper ready to show your demonstrator. You *must* get the work marked at the next available opportunity; please do not wait until later in the course, otherwise we can't promise that everything can get marked. You can get an extension to the deadline for this exercise, but only by asking for it explicitly—and you'll need to convince the lab supervisor that you've made decent progress already, and will be able to complete the work by the extended deadline without messing up other exercises in this course unit, or exercises in other labs.

The purpose of this lab is to help demystify some of the fundamental technology behind the Web; you're going to be communicating directly 'by hand' with a Web Server, fetching some simple raw HTML from it, and parsing that so that it can be displayed to a user. It'll will expose you to the ideas of IP addresses and ports, and is also a gentle practical introduction to programming in Python.

The early part of this lab is quite prescriptive, and you can get most of the marks just by following the instructions in the script. Towards the end there are a few more challenging open-ended tasks, but these are really only there in case you've been able to get through the early stuff very quickly. As you work through the exercise, write notes in your lab book (this is very important—some of the marks are for showing your notes to the demonstrator when you come to get marked, and other marks are awarded for answering questions that the demonstrator will ask you, and you'll need your notes to refer to for these).

As a guideline, you should definitely be able to complete all but the last deliverable for this lab in the lab session itself. Although you can get an extension for this exercise, you really shouldn't be doing that unless you are trying the very last task. If you find yourself needing an extension for anything but the last deliverable, then you probably haven't done enough preparation in advance of the lab, and should get into the habit of doing that in future. If you find that you're struggling with the programming part of the exercise and can't complete it in time, make sure you submit your results from the first few tasks anyway, and take this as an indication that you might need to brush up on your Python programming skills before the next lab starts.

In this lab exercise, you're going to be building a very simple web browser. Unlike Firefox and Internet Explorer, which are huge sprawling applications with an ever growing set of features, the browser you're going to write is going to be tiny, lean and very very simple. In fact, the core of it can be implemented in half a dozen lines of code.

There are no files to copy for this exercise, but you should make yourself a directory called `COMP18112` in your home directory, and save anything you create in this exercise in a subdirectory called `ex1`.

Task 1.1: Communicating with a web server (3 marks)

Before we get going on building anything, let's remind ourselves of how a web server works by talking to one directly in its 'native language'. For this, we'll use a program called **telnet**[W], which was originally designed as a means of logging in to remote machines[1], but as far as

[1]The term 'telnet' refers both to the program we're going to be using, and to the protocol it uses, but in any case these days it has been superseded by **secure shell**[W] (ssh) as a means of remote login because, amongst other

we're concerned is just a program that will take input from the keyboard, send it over the network to a remote server, and send us back whatever the server says (this only works for us because the HTTP protocol used by web servers is based on plain text, i.e. characters that you can type on a keyboard and display on a screen—this isn't always going to be the case for other protocols). Before we have a play with `telnet` though, let's remind ourselves of the basics of a web server:

Start by firing up a terminal window, and change directory into one of the COMP101 examples that contains some html files (it really doesn't matter what the HTML is at this stage; if you've removed the exercises from your filestore you can always get them back from gitlab). From within this directory, run

```
python -m SimpleHTTPServer
```

which will start up the same simple HTTP server that we used in the introductory labs. Remember, this program will serve up pages from whatever directory you started it in, so if there's a file called `index.html` you should now be able to point a web browser at

```
http://localhost:8000/index.html
```

and see its contents. Once you've confirmed that you can see the rendered HTML pages using a browser, start up another terminal window, and convince yourself that you can also fetch the raw HTML by using the `curl` command.

Now, both `curl` and your web browser are clients that know how to 'speak' HTTP. In the case of the `curl` command, it uses HTTP to communicate with the SimpleHTTPServer, and fetch the file, which it can then display on the screen or save to disk; it doesn't really know how to interpret the file it received any further. In the case of the browser, it knows how to speak HTTP to fetch the file, and it also knows how to then interpret the HTML that it received to draw the web page.

Next we're going to use `telnet` to talk to the server. Now, `telnet` knows about the TCP/IP network protocol used by the Internet, but it doesn't really know anything about HTTP or URLs or any of the ideas common on 'the web'; so we're going to be operating at a much lower level here. In a new terminal window type:

```
telnet localhost 8000
```

The first and second parameters here are the host name the port number of the server we want to talk to (remember, telnet doesn't know about the URL notation, so we can't use 'localhost:8000' here like we did with curl).

You should get a response something along the lines of

```
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```

problems, it used to send your password as plain text over the network.

and if so, all is well (you may see different IP addresses and different names…don't worry about that if it's the case, and don't worry about the first connection refused event above). In particular, note the line that tells you what the 'Escape character' is, we'll need that later.

Now try typing:

```
HELLO
```

and press return.

How rude! The server has sent back an error, and closed our connection! If we look closely at what's come back though, we can see that actually what the server did was send us back perfectly formed HTML containing a description of an error that would look sensible in a web browser (try cutting and pasting the response in to a file called `error.html` and then opening that file using your favourite web browser if you're not convinced).

Before going any further, you should understand what the 'Escape character' message you got from telnet means. When you start up telnet, you are talking directly to whatever server is at the other end of the connection (in this case, a web-server): every character you type is sent by the telnet program to the server that's listening at the other end of the connection. In our previous example, sending 'HELLO' caused the web server to respond with an error and then terminate it's connection to your telnet client. But in some cases you might find that the server is still 'holding on' to the connection at a point where you want to quit, or isn't responding. You could kill the terminal or the telnet process, but that's a bit extreme. If you type whatever escape character telnet told you it was using, however, you'll be dropped into telnet's own command prompt. Try it now: connect again to the web server, but instead of typing 'HELLO', type the 'ctrl' and ']' keys simultaneously, and you should find that you get a

```
telnet>
```

prompt. Now anything that you type is an instruction to the telnet program itself, and not to the server. Typing 'help' at this point will give you a list of all the commands that telnet understands. But for now, just use the 'quit' command to get you back to your terminal's command prompt.

So, what actually happened before when we sent the 'HELLO' command to a web server? The problem is that 'HELLO' isn't a valid way to start a HTTP session[2] , so the server has quite reasonably rejected our attempt at communication. In the real world this kind of error would only occur if our browser was buggy at a very basic level—but it's nice to know that it would be dealt with gracefully. We'll try again, this time sending something more meaningful to the server. Start up the telnet client again using the same command as before, but this time instead of 'HELLO' type

```
GET /
```

(In such interactions, you may need to hit the RET/ENTER key twice.) This time you should get a longer response. Type the same host name that we used for telnet into a web browser,

[2]Saying 'hello' to a server isn't as daft as it perhaps seems though; although its not the right way to begin a conversation with a web server, it is the right thing to say to a mail server (actually it's 'HELO' because for historical reasons the commands have to be four characters, but you get the idea).

and look at the source for the page that's displayed to convince yourself of what's happened here.

Next, connect to the server again and try something like

```
GET /[NAME-OF-A-FILE]
```

replacing [NAME-OF-A-FILE] with the name of one of the HTML files in your directory (note there is no space between the / and the filename). Confirm too what happens if you replace [NAME-OF-A-FILE] with something that *doesn't* exist in that directory.

Copy and paste all the responses you've got from the server so far into a text file called `ex1.txt`, and add some notes to explain what each response means; you'll need to show this to a demonstrator later to collect your first two marks.

From this little experiment, it should be fairly easy to see that a web browser is in fact a comparatively straightforward piece of software: to request a page, the client (normally a web browser, but in this exercise you've simulated being a web browser by typing in commands manually via telnet), which responds with more text formatted in HTML that can be interpreted to generate the kind of web pages we're all familiar with. The reality is a little more complicated than this, but not hugely so, and a lot of the hard work that goes in to building a 'real' browser comes from all the visual effects that need to be implemented in order to make web pages look attractive to humans.

Deliverables for Task 1.1 (3 marks)

1. Show the contents of the `ex1.txt` file to your demonstrator. **[1 mark]**
2. Explain the contents of `ex1.txt`. **[1 mark]**
3. Be prepared to answer one of the following questions, which will be chosen at random by your demonstrator **[1 mark]**:
   (a) So far you've seen two error codes returned from the sever; Error 400 that means that the server didn't understand the command you sent it, Error 404 means that the page you've asked for doesn't exist. What would error code 301 mean?
   (b) When you sent the `GET /[NAME-OF-A-FILE]` commands, you specified a file to retrieve (in this case, one that did and one that didn't exist). But in `GET /` there is no explicit file mentioned. What has the server done here?

Task 1.2: Displaying a HTML-like page (3 marks)

We're now going to build a very simple text-based browser that implements an extremely small subset of the possible formatting and layout instructions that can be specified using HTML. We're going to cut all kinds of corners, and make a whole bunch of assumptions that wouldn't hold for an industrial strength browser—but it should be plenty to give you an idea of what's really going on behind the scenes. You'll notice that this part of the exercise has

nothing much to do with Distributed Computing: it's just a bit of simple text parsing. Treat it as some Python practice, and we'll get back to the 'distributed' stuff soon enough.

Start by using any normal web browser to look at the web page we're going to render. The address is

`http://studentnet.cs.manchester.ac.uk/ugt/COMP18112/page1.html`

Next, use your browser to look at the source of that page (it'll be a menu option in your browser). You'll see that it's very simple, but valid, HTML. Next, fire up your favourite text editor, and into a blank file type the following:

```
1 #!/usr/bin/env python
2
3 import urllib
4
5 url = "http://studentnet.cs.manchester.ac.uk/ugt/COMP18112/page1.html"
6 data = urllib.urlopen(url)
7 tokens = data.read().split()
8 print tokens
```

Save the file as `browser.py` in `~/COMP18112/ex1`, and try executing it by typing in that directory:

```
python ./browser.py
```

The output should look something like...

```
['<html>', '<head>', '<title>', 'This', 'is', 'the',
'web', 'page', 'for', 'exercise', '1', '</title>',
'</head>', '<body>', '<h1>', 'Hello', 'world', '</h1>',
'<p>', 'Oh', 'I', 'do', 'like', 'to', 'be', 'beside',
'the', 'seaside', '</p>', '<p>', 'Oh', 'I', 'do',
'like', 'to', 'be', 'beside', 'the', 'sea', '</p>',
'<p>', 'Here', 'is', 'some', '<em>', 'emphasised',
'text', '</em>', '</p>', '</body>', '</html>']
```

and if it doesn't, you've probably typed something wrong!

Let's examine `browser.py` in a bit more detail.

Line 3 tells Python to include all the 'web' type functions for talking to web servers etc. We could at this stage ask you to use a slightly lower level library for this purpose, which would mean managing the connections and sending and receiving of packets of data and so on yourself; but that's rather more complicated than we want to care about right now, so we'll take this easier route instead. We'll explore so-called 'socket' communication later in the course.

Line 5 just assigns a string representing the address of the page we want to retrieve to a variable. Nothing complicated here (and feel free to try other addresses if you like; just be sure to change it back for the next bit of the exercise).

Line 6 does most of the hard work for us. It connects to the web server at the address in the 'url' variable, and constructs a `GET` line like we did manually a while back. It then sends this to the server, and retrieves the resulting stream of characters.

Line 7 reads the stream of characters, and splits them up into a list of individual words based on any whitespace in the page (in this case, just newlines and spaces); we'll call these words 'tokens' from now on.

So when we finally come to `print tokens` on the last line (Line 8) of the code, what we get is a list (we know this because it is a comma delimited thing surrounded by square brackets) of strings, each representing a word on the web page.

Most of the remainder of this exercise revolves around interpreting those tokens and generating a readable representation of the web page.

As to exactly how to do this, for the most part you're on your own, but you shouldn't find it too difficult since you now have a list of words, some of which you need to print out on the terminal, others of which you need to interpret as having particular meaning in terms of the way that the text is formatted. There are many many ways of approaching the problem, some of which are more elegant than others, but for the purposes of this course we don't really care too much exactly how you achieve this.

Here are some hints to get you going:

1. You've been given a list of strings, so you're almost certainly going to have to iterate through that list, performing some action (either printing the word or altering the formatting) as you go along. The following Python loop will print out each individual word in the list on its own line, and should give you a good starting point...

```
1 for token in tokens:
2     print token
```

2. In Python, comparing strings is easy, just do something like:

```
1 if token == '<body>':
2     startPrinting = True
```

For this part of the exercise, you'll get more marks the more sophisticated your interpretation of the page is.

For up to 2 marks, simply strip out the formatting tokens, and print the page with some obvious description, e.g.

```
Page Title : The title of the ex1 page

HEADING: Hello world
PARAGRAPH: Oh I do like to be beside the seaside
PARAGRAPH: Oh I do like to be beside the sea
PARAGRAPH: Here is some *emphasised text*
```

By way of an aside, if you want to generate **bold** text on a terminal (so that your output looks a bit nicer than just putting asterisks around the emphasised text, try printing the sequence `\033[1m` and then using `\033[0;0m` to get you back to normal text. If you've no idea why this worked, you should probably look up **Escape Characters**[w] and how they interact with terminals[3].

[3]This is the second time (at least) that you've come across the term 'escape' in a programming context (recall its use in telnet). It's usually interpreted as meaning 'stop doing whatever you normally do, and *escape* into some other mode or context'. For telnet, this means 'interpret what I type from now on as being instructions to telnet', on the terminal it means 'interpret the following characters as having a special meaning'

Deliverables for Task 1.2 (3 marks)

1. Strip out the `<h1>` and `<p>` formatting tokens and display the page in some human-readable way (i.e. with 'HEADING: Hello world') etc. **[1 mark]**.
2. Identify emphasised text, and display it in a way that distinguishes it from normal body text. To get the mark, it's enough to do this by just printing EMPHASISED in front of it as in the example above, but you'll feel like you've done a better job if you work out how to make it appear in **bold**! Make sure this works for `page1.html` and `page2.html`. **[2 marks]**.

Task 1.3: Hyperlinks (2 marks)

In this task you extend your web client to be able to identify hyperlinks, and to allow the user to move from page to page by selecting the links in some way.

Web pages aren't really web pages without hyperlinks. So have a look at `page3.html` on our server: it has three very trivial interlinked pages. Now, to make a decent browser, we'd have to be able to interact with the position of those links on your page (and there are text-only browsers that do this quite nicely, e.g. `http://www.delorie.com/web/lynxview.html`). The Python package that allows you to interact with the terminal like this is called 'curses' (`http://docs.python.org/2/howto/curses.html`) and you're welcome to have a play with that, but it's not necessary for this bit of the exercise. Instead, simply generate a numbered list of the links that are available on a page, and print the list at the bottom of the page, ask the user to type in a number, and, once the user has done that, fetch and display the corresponding page. And voila, a simple web browser!

We'd expect the output for `page3.html` to look something like this:

```
TITLE : The interlinked bit (the index)
HEADING: Links!
PARAGRAPH: Some interesting stuff can be found here
PARAGRAPH: And some other bits are also available
1 : ./page4.html
2 : ./page5.html
Select a link :
```

Reminder: as you've seen in the Python tutorial, to get a line of input from the user, use something like:

```
1 myMessage = raw_input('Type your message: ')
```

Deliverables for Task 1.3 (2 marks)

For one mark, make it possible to navigate from one page to a second page by selecting a hyper link (e.g. from a 'menu' as suggested... but any other mechanism you can

think of is okay too). For both marks, make it possible to navigate repeatedly from page to page. **[2 marks]**.

Task 1.4: Oh Curses! (2 marks)

If you're feeling brave and have been able to breeze through the majority of this lab, do something impressive that makes your browser more like a real browser (please don't go beyond the extended deadline for this; its more important that you keep on schedule with the labs than that you spend loads of time to get these final two marks). For example, make your it work with Python's curses library, or add some other Python-friendly graphical user interface framework. Or use an XML parser, or Beautiful Soup to interpret the HTML rather than your own home-made one. Look these things up on Google. No hints; you're on your own here! We also cannot offer help to install Python libraries that arenál̇ready available, but you may wish to explore how to do this on a machine over which you have full administrative control.

Deliverables for Task 1.4 (2 marks)

Something impressive. Ask your demonstrator for advice on how impressive it has to be to get both marks. **[2 marks]**.

Task 1.5: Submitting your work and getting it marked

Whether or not you've tried Task 4, you should now `submit` and `labprint` your work (the system is expecting to find a file called `browser.py` in your ~COMP18112/ex1 directory). Put your name on the marking queue, and make sure you submit your final mark. Please do not just submit the work and then leave the marking for another day—it causes problems later in the semester, and we can't guarantee to get work that is long overdue for marking done, in which case all your efforts today will have been wasted!

Exercise 2: A basic instant messaging system (2 lab sessions)

This exercise runs for two lab sessions. It has a regular deadline at the end of the second lab session, which you'll have to submit some code to meet, and an intermediate 'checkpoint' deadline at the end of the first session (to meet this you just need to convince your demonstrator that you've completed up to Task 2; this checkpoint deadline cannot be extended). When you've finished the whole exercise (or as much of it as you're able to complete) you should `submit` it, and get a `labprint` copy on paper ready to show your demonstrator. You *must* get the work marked at the next available opportunity.

In this second lab exercise, you're going to build a very simple instant messaging system that will allow two users to send and receive text messages. It's going to be a client/server architecture, and we've provided you with a simple server implementation, and a Python module that makes talking to the server very straightforward. Your task is to design and implement a protocol that will make the chat system work. The purpose of this lab 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'. Once again, we're not looking for beautiful code or for you to demonstrate a profound understanding of the programming languages used here: the distributed computing principles are what really matter (though you will get penalised for terrible unreadable code, so do your best to keep it nicely structured and commented).

It's also important to say at the outset that the system you're going to build is going to be inherently flawed in many ways. Some of these limitations you could, given enough time, work around. Others are essentially insurmountable, and arise from the architecture and technology we've chosen for this lab. So we're not claiming that the architecture we've suggested here is the ideal way of building such a system—far from it, in fact! As you're doing the exercise, try to think about what the limitations of this approach are, and how you would design a more realistic solution (when you come to get your work marked, you'll be asked questions about this, which are worth a few marks!). If you find yourself thinking "this is a bizarre way to do X, surely it would be better if it was done like Y", pat yourself on the back, you've probably learned something valuable about distributed computing! We'll revisit these issues in the next lab exercise.

To start with, you'll need to download some files that we've prepared for you. Make yourself an `ex2` directory in `~/COMP18112` and then copy the contents of

```
/opt/info/courses/COMP18112/Lab/ex2/*
```

into that directory.

You should end up with two files called `IMServer.php` and `im.py`. If this is the case, all is well and you can continue with the exercise. Otherwise, if you've followed the instructions properly but don't have these files, something odd has happened, and you should probably get some help from a demonstrator at this stage.

Task 2.1: Setting up and interacting with the 'state store' via a browser (2 marks)

ls

The `IMServer.php` file, as the name suggests, is going to act as your server. You need to have a directory called `cs_html` in your home directory that is readable and executable by all (you may have already created this directory during COMP10120 if you forgot your Raspberry Pi at

some point!). Copy the `IMServer.php` file into `cs_html` in your home directory, make sure that it is readble by all, and then confirm that you can access it via a web browser using the URL

```
http://webdev.cs.manchester.ac.uk/~username/IMServer.php
```

You should get a page with just the heading 'COMP18112 IM Server' in it, in which case all is well (if there's another line, after the heading, containing strange stuff, that's ok too... it'll become clear what that means shortly).

If you're having problems, make sure again that the `cs_html` in your home directory is readable and executable by all and that the `IMServer.php` file is readable by all,—so check permissions carefully (`chmod a+r,a+x nameofdirectory` and `chmod a+r nameoffile` may help you out here).

If you can't see anything at all, and have followed these instructions carefully, check with a demonstrator.

The server you've 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

```
http://webdev.cs.manchester.ac.uk/~username/IMServer.php?action=set&key=test1&value=hello
```

The browser will respond with a blank page, but should now have remembered that the value of the key 'test1' is 'hello'.

Look at

```
http://webdev.cs.manchester.ac.uk/~username/IMServer.php
```

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

```
a:2:s:5:"test1";s:4:"hello";
```

Don't worry about the a:2 and s:5 bits; they are likely to vary from case to case, and are just bits of internal implementation and really aren't important. What matters is that you can see 'test1' and 'hello' in the list of things the server has remembered.

Try setting a few more key/value pairs, and confirming that they are being remembered, then try something like

```
http://webdev.cs.manchester.ac.uk/~username/IMServer.php?action=unset&key=test1
```

to make sure that you can unset them too. Unsetting a key removes it from the state store.

Finally, 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

```
http://webdev.cs.manchester.ac.uk/~username/IMServer.php?action=get&key=test1
```

to retrieve it. **Replace ~username with your University username (e.g. bloggf1) wherever it appears**.

Your browser should respond with just the value you've put in there previously (assuming you've not unset it in the meantime!)

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 chat system you're about to write.

Task 2.2: Interacting with the state store from a Python program (1 mark)

Before you start designing you program, 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.

Make sure you are in the same directory as the `im.py` file you downloaded earlier, and start up the Python interpreter by typing `python` 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('http://webdev.cs.manchester.ac.uk/~username/IMServer.php')
```

If you've done this correctly, Python shouldn't be reporting any errors. Line 1 tells Python to use the class that we've 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've just put in.

You can also unset an association.

Try typing:

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

and then check the contents of the server store again 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 some variable using a browser and the URL method, and once you've 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**[w] (or REST for short); we'll be looking at this in more detail later in the course).

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.IMServer; 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 chat client properly, take a quick look at the `im.py` module that we've provided. This file imports the same 'urllib' module that we used in the first lab exercise—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' (see the notes from Examples Class 1 if you want to know more about this). Basically this means that when we do

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

this module will create and GET the URL

```
http://webdev.cs.manchester.ac.uk/~username/IMServer.php?action=set&key=myKey&value=something
```

and similarly

```
print server['myKey']
```

will do the same as typing

```
http://webdev.cs.manchester.ac.uk/~username/IMServer.php?action=get&key=myKey
```

Deliverables for Task 2.2 (1 mark)

This is the 'checkpoint deliverable' for this lab. You must reach this point within the scheduled lab session to get this mark; no extension is available. Once you're happy you can do what's required, put your name on the marking list. You do not need to `submit` anything for this checkpoint.

Show that you've been able to follow the instructions in task 1 and that you're able to manipulate the contents of the state store both using a web browser and from Python. Your demonstrator will ask you to insert a key/value pair using Python (you can do this from the interactive Python prompt, or by saving it in a file if you prefer), and then to show the contents of the store via a web browser.

Task 2.3: Designing a simple Python chat client

Now that the infrastructure is in place, all that remains is to create the chat client! To keep things simple, we'll make this a chat system that works for exactly two users at a time; not one, not three, but exactly and only two. 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.

At this point, move away from your keyboard and think for a while. The actual code you are going to write is very simple, 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 strategy (a 'protocol') that will allow two independent programs to exchange messages using only the functionality provided by your simple server.

Here are some hints to get you going.

The basic functionality really is very simple. Almost trivial. It goes roughly along the lines of:

1. Wait for the user to type in a message.
2. Send that 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:

Obviously, if both users are waiting to send or receive a message at the same time, our system is instantly deadlocked and nothing will work. You'll need to think of some way of starting one user off in the 'sending a message' state, and the other user off in the 'waiting for a message' state. There are lots of ways of doing this, some more elegant than others.

Task 2.4: Writing the chat client code (7 marks)

Once you're convinced that you've worked out the structure and logic of the chat client, 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 = raw_input('Type your message: ')
```

(we've used this in the examples class notes, and in the previous lab)

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

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

You might want to think about what happens if your program crashes mid chat. Does it leave the server in a state from which it can recover? (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've provided actually stores the key/value pairs, take a look at the source code).

If you've started writing any of your chat client code before you read this line, give yourself a slap on the wrist, and stop and think about the design of your program for a moment: if you plan it in advance and think through what your algorithm/protocol is going to do, the code exercise is VERY easy—almost certainly shorter than the code for Exercise 1; but if you rush to put finger to keyboard, you'll be in a muddle before you know it.

Deliverables for Task 2.4 (7 marks)

- A working chat client, written in Python, that allows a pair of users to alternate sending and receiving messages to and from one another. You should submit this file in the usual way as `imclient.py`.
  - Allows two clients to exchange alternate messages as described in the task. **[3 marks]**
  - Includes some mechanism to avoid the system deadlocking on startup. **[1 mark]**
  - System is robust to a client failing mid conversation (i.e. the server doesn't need restarting, etc). **[1 mark]**
- Explain to your demonstrator what the problems with architecture are, and what kinds of things would need to change to make this a 'realistic' system. **[2 marks]**

Task 2.5: Something extra (2 marks)

As with Exercise 1, do something impressive! Examples might include:

- Using only PHP server provided, allow more than two users to chat.
- Find a way of allowing messages to be sent and received asynchronously (i.e. so you don't have to wait for the other user(s) to send a message before you can send another)
- Build a 'graphical' version of the system, either using ncurses or any Python GUI toolkit of your choice.

Deliverables for Task 2.5 (2 marks)

Something extra, along the lines of what's suggested above (but we're open to other ideas too; if you're unsure whether your idea is sufficiently impressive, check with a demonstrator beforehand)

Task 2.6: Submitting your work and getting it marked

Regardless of whether you've tried some of the harder bits in the previous task, you should `submit` and `labprint` your work, and get your name on the marking list at the earliest possible opportunity. The system is expecting a file called ~/COMP18112/ex2/imclient.py.

Exercise 3: A more realistic instant messaging system (2 lab sessions)

This is a two-session exercise, with a normal deadline at the end of the second session (which you can ask to be extended to the marking session if you really need it), and a checkpoint deadline at the end of the first session (which can't be extended under normal conditions). To meet the checkpoint deadline you'll need to get at least as far as T in the first lab session. If find it at all difficult, please make the most of the examples class clinics that will be held in the week between your two lab sessions!

In the previous lab exercise, we built an instant messaging system (of sorts!) using a page of PHP hosted by a web server to act as a central 'state store'. Building this far-from-optimal architecture should have highlighted several fundamental issues in distributed computing—in this exercise we'll revisit the problem and develop a more realistic and robust solution. This time we're going to build an instant messaging setup using a proper message passing client and server.

It should be able to:

- Allow any number of clients to connect, and identify themselves with a screen name.
- Allow any user to send messages to all other users.
- Allow any user to send a private message to a particular user.

Get started by downloading the files you need from

`/opt/info/courses/COMP18112/Lab/ex3`

and put them in a suitably named subdirectory of your `~/COMP18112` directory.

You should have the following files:

- `ex3utils.py` : this is a Python module containing a couple of classes that you'll need to extend; more on this shortly.
- `client.py` : a very simple client example.
- `server.py` : some simple server examples for you to experiment with.

Let's start by testing out the server example we've provided. Run the `server.py` program, giving it two command-line parameters, the first of which is the host on which the server is to run (which is going to be whichever machine you're on right now), and the second of which is a port number. For example

```
python ./server.py localhost 8090
```

The server should respond by saying 'Echo server has started'. Now, typically using another shell, connect to the server with telnet, using a command such as

```
telnet localhost 8090
```

and after the usual preamble about addresses and escape characters has appeared, try typing a line of text in to telnet and pressing return. You should see the same line of text reflected back to you, but in capital letters. If you do this in several attempts, you may eventually note that if the server did not close cleanly in the previous interaction, the port may still be held. If this help, try closing the server (e.g., type the 'ctrl' and 'c' keys simultaneously) or use another port altogether.

Now have a look at the `server.py` code using your favourite text editor, and we'll walk through what's actually happened here.

At the top of the file are some reasonably detailed comments on how the server implementation works; skip past these for now. Next you'll see the usual 'import' section, which in this case uses 'sys' (Python's way of getting access to boring operating system functions such as 'exit' to cleanly quit a program or access the command line parameters), and 'ex3' (which is the module we've provided, and from which in this case we import the 'Server' functions).

The next bit of the file does something we've not looked at in much detail in Python yet, and creates a Python class called `EchoServer`, which inherits its functionality from the Server class we've provided in `ex3utils.py` (this is pretty much the same as Java's use of 'extends', it's just that the syntax is different). As with our approach to Python in the rest of this course, the subtleties of the language's object orientated model and class system are totally irrelevant here; and all that you need to know for this lab exercise should be quite clear from this simple example. If you look at the definition of the `EchoServer` class, you'll see it contains two methods, `onStart` (which does nothing apart from print the welcome message), and `onMessage` (which you can see does very little apart from convert the incoming message in to upper case, and then 'send it on a socket'—we'll look at what this means in a little more detail shortly).

The file then contains an alternative server implementation called `EgoServer`; skip this for now and look at the last few lines starting from the comment

```
# Parse the IP address and port you wish to listen on
```

These last few lines of code simply take the command line arguments for IP address and port, create an instance of our server, and start it going. The Server superclass we've provided then attaches itself to that port, and waits for incoming connections, calling the methods provided in `EchoServer` (`onStart`, `onMessage` etc.) whenever something interesting happens, such as a client connecting, or a message being received. If you like, swap the instance of `EchoServer` for an instance of `EgoServer`, and try talking to the server once more with telnet; it should be fairly obvious what's going on here, but ask for help if it's not!

Now let's return to the comments at the top of the file; these document all the different methods that can be overridden to make your own server implementation. Make sure you've read the comments thoroughly, and ask for help if they don't make sense.

Task 3.1: Your own basic server (1 mark)

Now it's your turn to write some code. Your first task is to create a server (call it `myserver.py`) that simply acknowledges that it has started, that a client has connected, that it has received a message, that a client has disconnected, or that it has been instructed to stop. All you need to do here is print out a suitable message on screen for each. You should be able to test all of these eventualities using telnet, as we did for `EchoServer` and `EgoServer`.

Hint: your server is going to be very similar to our `EchoServer` implementation, but with a few more methods implemented (and these methods are explained in a lot of detail at the top of the file!)

Deliverables for Task 3.1 (1 mark)

Your server code must be able to respond to start, connect, message and disconnect events by displaying a suitable message on screen. **[1 mark]**

Task 3.2: Counting clients (1 mark)

Still using telnet as a client, arrange for your server to remember the number of currently active clients, printing out the new total every time a client connects or disconnects.

Deliverables for Task 3.2 (1 mark)

Your server code must be able to remember how many clients are connected at any one time, displaying a message with this value every time a client connects or disconnects. **[1 mark]**

Task 3.3: Accepting and parsing commands (1 mark)

So now you have a basic 'generic' server; it will sit and listen for incoming connections, and handle messages sent to it in text by clients. The next task is to do something more meaningful with the messages. Later, we'll arrange for them to be sent back to clients to make a chat system, but first we need to establish a protocol for how we're going to interpret messages. It should be clear by now that anything you send (e.g. via telnet) to this server ends up appearing in the `onMessage` method as a string of text. We could at this stage build a simple server that simply 'reflects' all the messages that are sent to it from clients back to any clients that have connected (i.e. so a message sent from one person gets sent to all the others). But it would be nice to do something more sophisticated. For example, we should really allow clients to register a name, so you can tell who sent which message. It would also be nice to be able to send messages either to all members of the chat session, or privately to specific users (in which case, of course, we need to know their name!).

Achieving this means we'll need to encode some more meaning in the text that's being exchanged; for example, what 'screen name' is being used by which user, or the name of another user to which you'd like to send a private message. Following on from the style used by http and smtp servers, we'll simply establish a protocol based on text commands that look like this:

```
<COMMAND> <some parameters>
```

so we might have something like

```
MESSAGE Hello world
```

Where MESSAGE becomes the command, and 'hello world' becomes the parameters.

In Python a line such as

```
(command, sep, parameter) = message.strip().partition(' ')
```

will take the string, and separate it out on the first space we encounter . For example if we did...

```
myMessage = 'MESSAGE Hello World'
(command, sep, parameter) = message.strip().partition(' ')
print 'Command is ', command
print 'Message is ',parameter
```

we should get an output of:

```
Command is MESSAGE
Message is Hello World
```

You can use this (or any variation you like) in your `onMessage` method to start interpreting commands as they are sent to the server.

Write some code in the `onMessage` method to do exactly this, and test using telnet that you can indeed parse out a command and its parameters.

Deliverables for Task 3.3 (1 mark)

Your server code should be able to accept commands sent from the client in some sensible format, and display the commands and their parameters on screen. **[1 mark]**

Task 3.4: Designing the protocol (1 mark)

Step away from the keyboard; time to think, not write code.

Now we come to the crux of this exercise, the design of a chat system. You should by now understand the capabilities of the server, which are a lot more useful for a chat system than our previous Apache/PHP based server. We can accept connections from an arbitrary number of clients, and unlike the server from Exercise 2, remember that those connections exist. We can accept messages sent from those clients, and are free to decide what messages are meaningful or useful.

This next task requires you to design a suitable set of messages that together form a protocol for your chat system. You can do this with pen and paper, or if you must, with a text editor - but don't be tempted to try to implement these until you're confident you have a sensible

protocol (you might want to get this sanity-checked by a demonstrator before you continue, it will only take a moment and could save you a lot of pain later on!)

You'll need to account for at least the following

1. A new user registering themselves with the server, and giving the server a 'screen name' that they want to use.
2. Some way of sending a message to everyone that's connected to the server.
3. Some way of sending a message to a specific user (this bit is 'optional' in the sense that you can get most of the marks without it…but even if you don't implement the private message options, you should take them into account in your protocol).

Hint: remember to design your protocol so that it can be tested from telnet; i.e. don't make it difficult to actually type your commands!

You will need to show your design when you get your work marked. It should include a list of the commands, and what they do. You should also write pseudo code that describes how your server will handle the various commands, and in what order etc (you can draw an informal flowchart for this if you prefer).

Deliverables for Task 3.4 (1 mark)

You'll get one mark for showing your demonstrator a sensible attempt at protocol design (which can be on paper, on on screen, whichever works best for you). This mark is the 'checkpoint' deadline for this exercise, and you can only get it if you have the protocol to show before the end of the first lab session (as with the previous exercise, there is nothing to `submit` for this checkpoint; just show your demonstrator your work. You may change your protocol later as you implement the client and server code; all you need to do to get this mark is to have made a sensible stab at the design by the right time. **[1 mark]**

Task 3.5: Implement your protocol in the server (2 marks)

Now you should implement your protocol; most of the work will take place in your `onMessage` method, though you may find that you want to use the `onConnect` method too for initialising user-specific data (and perhaps even the `onStart` method for initialising server-wide variables). You have already seen (or written) enough code in this lab exercise to have covered all the important bits of functionality you need, e.g. decoding a message, sending a message back over a socket to a client etc.

One important point to note at this stage. Remember that unlike in Java, Python allows you to create variables on the fly (and is happy as long as you've assigned some value to a variable before you try to read from it). This also applies to member variables—which may seem a little odd—but is quite useful for short programs like this. What this means is that if you want to add new attributes to a 'user', you can hang them off the socket information that gets passed to you via `onConnect` and `onMessage`. So for example, in `onConnect` you might do something like:

```
def onConnect(self, socket):
        # Initialise connection-specific variables
        socket.screenName = None
```

and then later in `onMessage`, something like

```
def onMessage(self, socket, message):
        # Split incoming message
        (command, sep, parameter) = message.strip().partition(' ')

        # Act upon REGISTER message
        if command == 'REGISTER':
                socket.screenName = parameter
```

Hint: you should familiarise yourself with Python maps, you'll find them very useful for remembering things in the server; we used them in the previous lab for our key/value pairs.

Deliverables for Task 3.5 (2 marks)

A server implementation (in a file called `myserver.py`) that responds to the protocol you designed in Task 4. **[2 marks]**

Task 3.6: Writing the client (2 marks)

By now you should have established a protocol, and tested that your server can respond sensibly to your protocol using telnet. This next task involves writing a Python client that makes using your server a little more friendly.

We've provided you with an empty skeleton for your client code, in `client.py`, which should be enough to get you going. Copy this file to make your own version called `myclient.py`. As with the server implementation, it's done by subclassing (extending) a Python class that we've provided. An important thing to note here is that the client skeleton deals with the problem of messages arriving at any time—even in the middle of 'raw input'—so the problem we saw in the previous lab of having to worry about who connected first etc. is avoided. If you'd like to see how much effort and code is actually required to achieve what's apparently a very simple thing, then have a look through the implementation of the Client class in `ex3utils.py`, but don't worry about understanding the detail of it.

The basic task here is simply to implement your own protocol from the client's point of view.

i.e. your client needs to

- Connect to the server.
- Register your screen name.
- Repeatedly request input from the user (using `raw_input`), and then send the message via the server, either to a single user or to the group.

Deliverables for Task 3.6 (2 marks)

A client program (`myclient.py`) capable of interacting with your server. **[2 marks]**

Task 3.7: Finishing touches (2 marks)

For the really adventurous…

- Build a GUI for the client that separates out the input and output of messages, or
- Implement some 'administrative' functions, such as finding from the client the names of other users that are connected, or
- Implement a 'group message' function, that allows you to send a 'private' message to a specified subgroup of users, or
- Do something else impressive!

Deliverables for Task 3.7 (2 marks)

Something impressive. **[2 marks]**

Task 3.8: Submitting your work and getting it marked

You'll need to `labprint` and `submit` both `myserver.py` and `myclient.py` to finish. If you didn't get as far as creating your own `myclient.py`, just submit a blank file for this to allow the `submit` command to work.