Distributed Algorithms 60009

Lab 1 – Introduction to Elixir

https://www.doc.ic.ac.uk/~nd/dac/lab1.pdf

01 Welcome to the Distributed Algorithms module (60009).

02 The module uses Elixir, an elegant concurrent, distributed and functional programming language.

03 Lab 1 entails a few exercises for you to start to familiarise yourself with the Elixir programming language. Elixir will be used to code algorithms in the lectures, labs and for the coursework.

04 It is important that you have an experimental approach to the course - learning the language and tools, and trying small programming exercises of your own. Make notes, keep bookmarks and tell your classmates about things you discover - use *Piazza* to share knowledge.

05 You can work on the exercises with a classmate.

06 Although you can copy and paste text from this document, you'll find it instructive to type in commands directly. You'll make a few small mistakes, but you'll learn more doing so.

07 OPTIONAL. You can use Elixir on your own computer. Elixir is available for most platforms. Installation instructions to install Elixir on your own computer can be found `at` https://elixir-lang.org/install.html. The current version of Elixir is 1.11.3.

08 On macOS, I use **`brew`** (https://brew.sh) to install. On Raspberry Pi OS, I use **`asdf`** (https://asdf-vm.com/#/). On MacOS, you may also need to install **`Xcode`** (a very very big install) or the much smaller Xcode command line tools (using **`xcode-select --install`**). An alternative on Windows might be to use **Windows Subsystem for Linux**[1] (https://docs.microsoft.com/en-us/windows/wsl).

09 Note: In the past, the lab exercises and the coursework were carried out using Docker containers and virtual machines. However, many students found using Docker too complex. We will not be using or supporting Docker/VMs. You are welcome to do so if you wish, either, using the Department's cloud service or an external cloud service like Microsoft Azure or by installing Docker on your own computer.

EXERCISE 1 – Compiling and execution

10 The first exercise is to compile and run an Elixir program that spawns 100,000 Elixir processes and then waits for a message from each of the 100,000 processes.

11 Download the Elixir program using `curl`, for example:

```
1   mkdir -p ~/dac/lab1/ex1     # ~ is tilde character
2   cd ~/dac/lab1/ex1
3   curl -O http://www.doc.ic.ac.uk/~nd/dac_src/processes.ex
```

[1] I don't have Windows machine to test WSL compatibility however.

12 Have a quick look at the program. Don't worry if you don't fully understand the code. Check if there is there is a colour template for Elixir code for your preferred text editor – colourising code can help eliminate simple syntax errors.

13 Compile **processes.ex.** The resulting bytecode will be saved in a file named **Elixir.Processes.beam**

```
4    elixirc processes.ex
5    ls
```

14 Run the function **Processes.start** (as an Elixir process). The total time should be a few hundred milliseconds depending on the computational power of your computer.

```
6    elixir -e Processes.start
7    rm Elixir.Processes.beam      # remove the bytecode file
```

15 Note: We will not be compiling and running programs directly using the **elixirc** and **elixir** commands. Rather we will build and run **elixir** programs with the **mix** build tool.

```
8    mix new processes             # this will create a new Elixir project inside a
                                   # new sub-directory called processes

9    cd processes
10   ls -a                         # take a peek at the various files and directories
11   cat mix.exs                   # build file for our Elixir project
12   ls -a lib test
13   cat lib/processes.ex          # default program file
14   cat test/*.exs                # default unit test file

15   cp ../processes.ex lib/       # now override the default version (.ex file)

16   mix compile                   # compile source files, bytecode goes in _build/
17   mix run -e Processes.start    # run Processes.start using mix
```

EXERCISE 2 – Simple client server

16 Now download and extract the code for a simple distributed client-server system.

```
18   cd ~/dac/lab1
19   curl -O http://www.doc.ic.ac.uk/~nd/dac_src/ex2.tgz
20   tar xzvf ex2.tgz
21   cd ex2
```

17 You should find a number of files and directories including a **Makefile**, and 4 Elixir source files **client.ex, server.ex,** b, and **helper.ex** (in directory lib). Take a long peek!

18 Compile and run the system in a single Elixir node using:

```
22   make clean                    # will call mix clean
23   make compile                  # will call mix compile
24   make single                   # will run all processes in a single Elixir node
```

19 Use Ctrl-C, Ctrl-C to terminate the system. The node will exit after 10 seconds anyway (this time limit is set in the **Makefile**).

20 Now run the system using 3 Elixir nodes, 1 for the Elixir client process, 1 for the Elixir server process, and one for the top-level **clientserver** process:

```
25   make cluster                  # will run using 3 Elixir nodes on localhost
```

21 In a separate terminal window look at what Elixir nodes are running on your computer with

```
26   make ps                       # do this within the Makefile's 10 sec time limit
```

EXERCISE 3 – N-Clients, Single Server

22 The final longer exercise is to adapt the client-server system for N clients and 1 server. It is important to understand what's going on and not just follow the instructions blindly. Add notes to the code and **`Makefile`**.

23 Copy the client-server system for exercise 2 into a new directory

```
27   cp -a ~/dac/lab1/ex2 ~/dac/lab1/ex3
28   cd ~/dac/lab1/ex3
29   cd ex3
30   make clean
```

24 First edit module **`Client`**:

(i) Include the Client's process-id as the first element in the area request message sent to the server process. Use the function **`self()`** function to get the process-id of the running process.

(ii) Pass clients the server's process-id in either an initial **`:bind`** message or simpler still pass the server's process-id as a parameter to all spawned client processes in **`ClientServer`**.

(iii) When printing area results, indicate the id for clients, either the process-id (pid) or your own numbers. You can use **`inspect(self())`** to convert process-id's into strings.

25 Edit module **`Server`**:

(i) to send results to the requesting client's process-id. Recall that the process-id should now be included
as the first element in the client's area request message.

(iii No longer wait to receive a :**`bind`** message from the **`ClientServer`** process.

26 Edit module **`ClientServer`**:

(i) In *single_ start* spawn N client processes at **`clientserver_<node_suffix>.`** The number of clients will be available in **`config.clients.`** You can use Elixir **`for`** *comprehensions* to spawn N clients. Either pass the server's process-id as a parameter to each client (in the square brackets [ ]) or send a **`:bind`** message to each client. Note: the server does not need a **`:bind`** message since it will receive client process-id's in the area request messages sent to it.

(ii) In *cluster_ start* spawn N client processes at **`client1_<node_suffix>`** to **`clientN_<node_suffix>`** respectively.

(iii) After spawning the **`server`** process wait a few milliseconds to give it time to startup before spawning the **`client`** processes.

27 Edit the **`Makefile`**:

(i) Add a **`CLIENTS`** variable to the **`Makefile`** and set **`CLIENTS`** to 5.

(ii) Append **`${CLIENTS}`** to the end of the **`MIX`** variable definition.

(iii) Change the target *cluster* to 5 client nodes named
client1_${NODE_SUFFIX}** to **client5_${NODE_SUFFIX}

28 Edit the Helper module:

(i) Add **:clients** to **config** in **Helper.node_init().**
The number of clients will be in **argv[2]**, convert it to an integer.

29 Test your adapted code for 5 clients for both single node and cluster execution. Use **make ps** to see what nodes are running.

30 Now randomly vary the request that clients send between a circle and square. Also make your client randomly sleep between 1 and 3 seconds between requests. You can use **Helper.random().**

31 Add a request to calculate the area of a triangle with 3 sides using Heron's formula. You can use **Helper.sqrt().**

That's all folks! Well done for completing Lab1.

32 The are many on-line Elixir resources. Bookmark the ones that you find useful.