2021

Distributed Systems

COMP90015

ADNAN MUNSHI
STUDENT ID : 1139173
PROJECT 1
SEMESTER 1 2021

Table of Contents

INTRODUCTION

This project has been designed to demonstrate the use of two fundamental technologies that have learned through our lectures. The basic idea behind it is that the program will follow a client-server architecture in which multiple clients can connect to a (single) multi-threaded server and perform operations concurrently. This report will discuss and analyse the problem context of the multithreaded dictionary server as well as the components and user interface.

PROBLEM CONTEXT

The project requires us to build a multithreaded dictionary server which would be in charge of handling multiple clients to be able to carry out the basic functionalities. The basic functionality of this program includes adding, removing or querying words in the dictionary.

COMPONENTS OF THE SYSTEM

System Architecture

We have learned several types of System Architectures such as Server-Client, Peer-to-Peer , Mutiple-Servers etc. However, for the purpose of this project, I chose the Server-Client architecture. This client-server architecture which uses the TCP socket for communications. Our system has two separate files ; “Server.java” and “Client.java” which are for the server and the client respectively.

Thread

For the multithreading aspect, we were given a couple of options to choose from; worker pool, thread per request or thread per connection. I have implemented thread per request to allow multiple clients to the connect to the server simultaneously and use it’s functionalities. This allows multiple clients to access the dictionary at the same time.

Thread per connection uses one thread per connection regardless of whether the client is active or not which may lead to threads being occupied even when it is not in use. Based on this scenario, thread per connection is a better choice as the threads will be used based on the needs of the clients.

Dictionary

The system also uses the JSON format to transfer messages between the server and the client as well as for storing the definitions in the dictionary.

GUI

To improve the user experience for both the server and the clients, I have implemented a simple but effective GUI. The clients can easily add, remove or query words with their respective buttons. The server can also add and remove words. However, the server can also see the full list of words in the dictionary. The GUI is covered in detail in the sections below.

OVERALL DESIGN CLASS DESIGN AND INTERACTION DIAGRAM

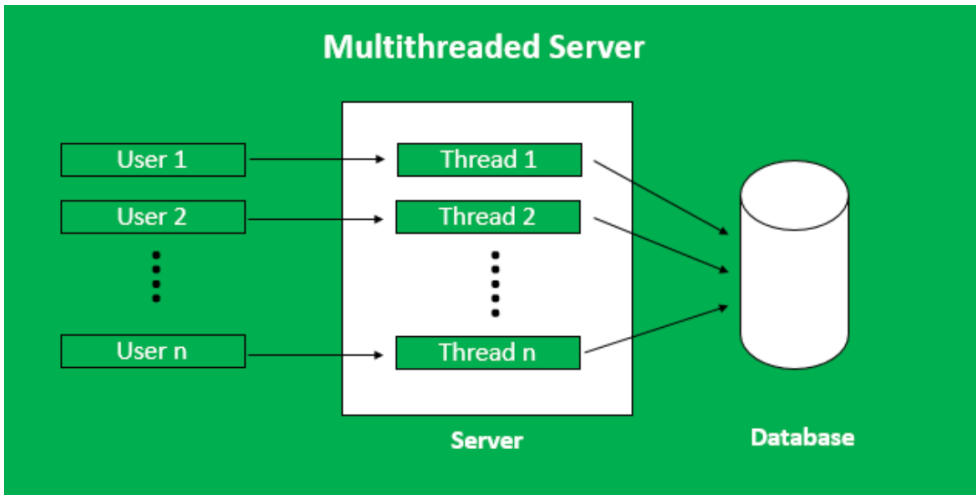

Figure 1 Mutilthreaded client server retrieved from geeksforgeeks.org

Our overall design follows the architecture shown in Figure 1 where database is our dictionary. Our program consists of 2 main classes; server.java and client.java where we have implemented the server socket connection as well as multithreading. We also have 2 additional class for transmitting data in JSON format ; CommandWrapper.java which takes a command and the word in JSON format and Word.java which holds the word and its meaning in the JSON format.

The dictionary file doesn't contain any words but is created upon initiating the program. The uses GSON-2.6.2.jar to build a JSON object.

To run the program, follow the instructions below:
- By terminal : run the DictionaryServer.jar by ***> java –jar DictionaryServer.jar <port> <dictionary-file>*** and then run the client by ***> java –jar DictionaryClient.jar <server-address> <server-port>***
- By IntelliJ : run the server class and then the client class.

After the server and clients are running, you can add, remove or query words either through the server or through the client. You can run multiple clients simultaneously.

Client and server windows

When we first run the client and server programs, this is what is looks like initially. The server logs show the number of clients currently connected as well as the Word List currently present in the dictionary as shown in Figure 2.

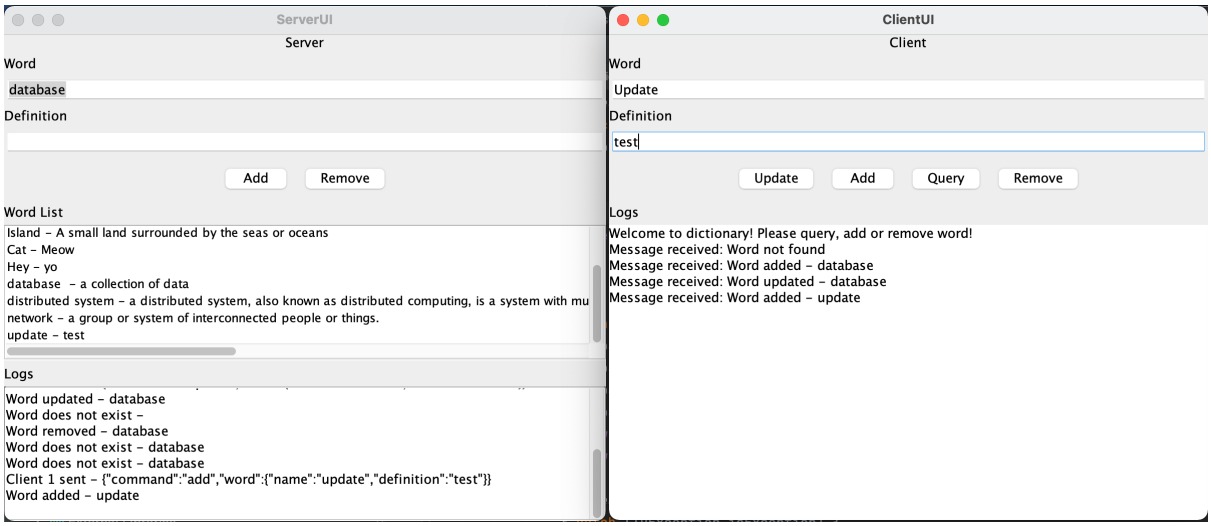

Figure 2 Opening windows of server and client

Figure 2 shows the windows of both the server and the client after adding a word, “Update” and it’s definition, “ Test”. It can be seen the server has a real time view of the words currently present in the dictionary. Figure 3 shows that the word, “Update” successfully got updated to “update success”.

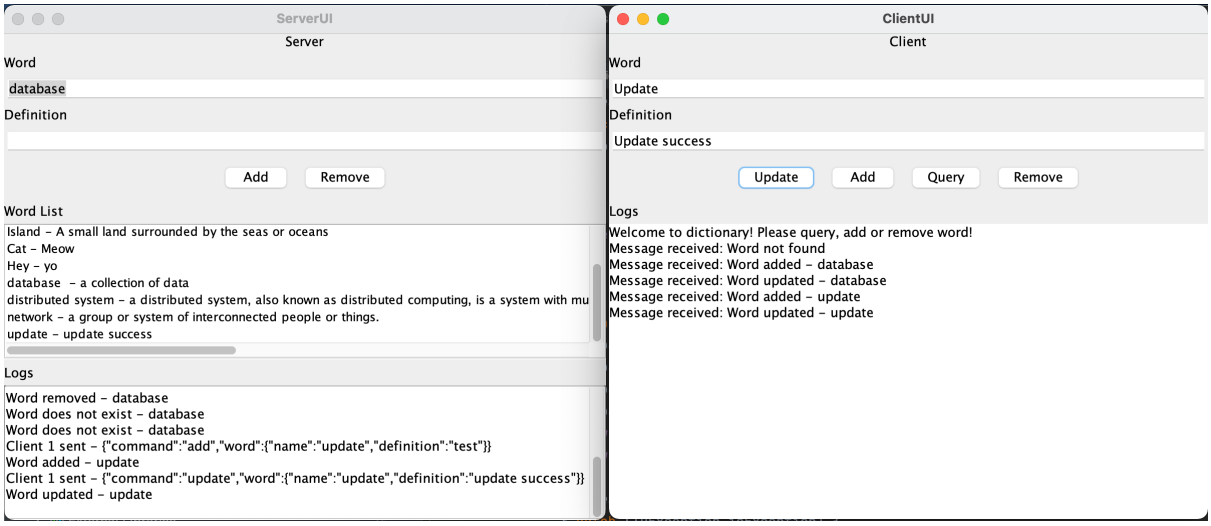

Figure 3 Adding a word from the client

Adding and quering words from multiple clients simultaneously

This is where the multithreading aspect of the system comes into play. It can be seen that Client 1 connects and adds the word, “spectacles” and then Client 2 connects and adds the words, “food”. While Client 2 is connected, Client 1 is still able to retrieve the meaning of “spectacles”. This is one of the reasons of implementing multithreading. It gives each user the ability to simultaneously do their work without waiting for other clients to finish their tasks.

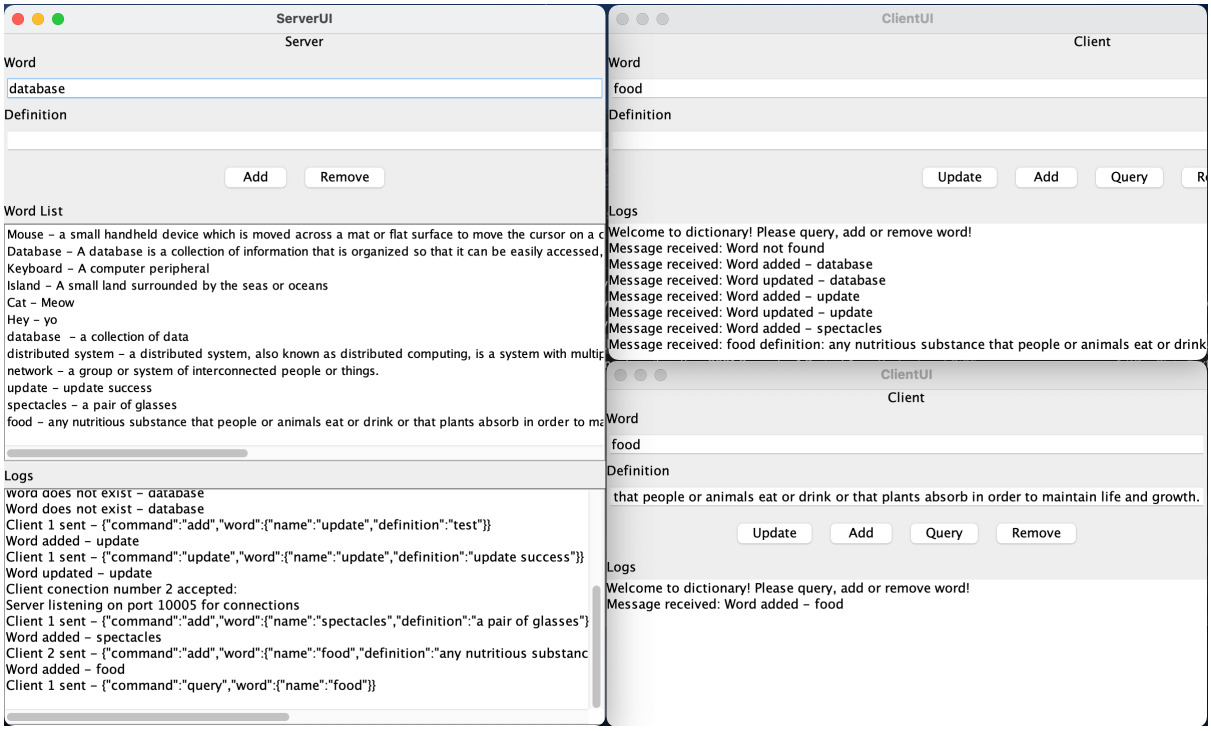

Figure 4 Adding words simultaneously from different clients.

Querying words

Figure 5 shows how a client can query a word from the available words in the server.

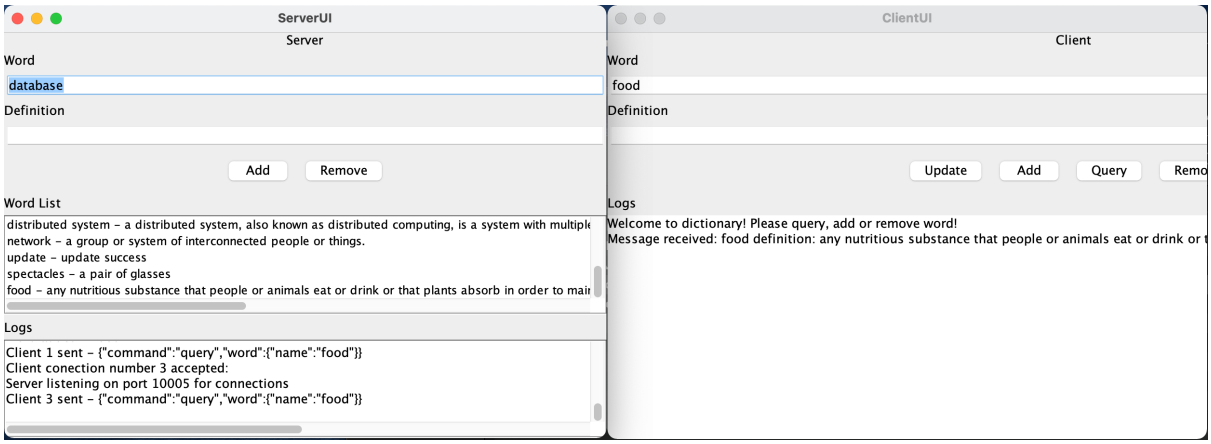

Figure 5 Query of a word

Removings words

Figure 6 shows how a client can remove a word from the dictionary.

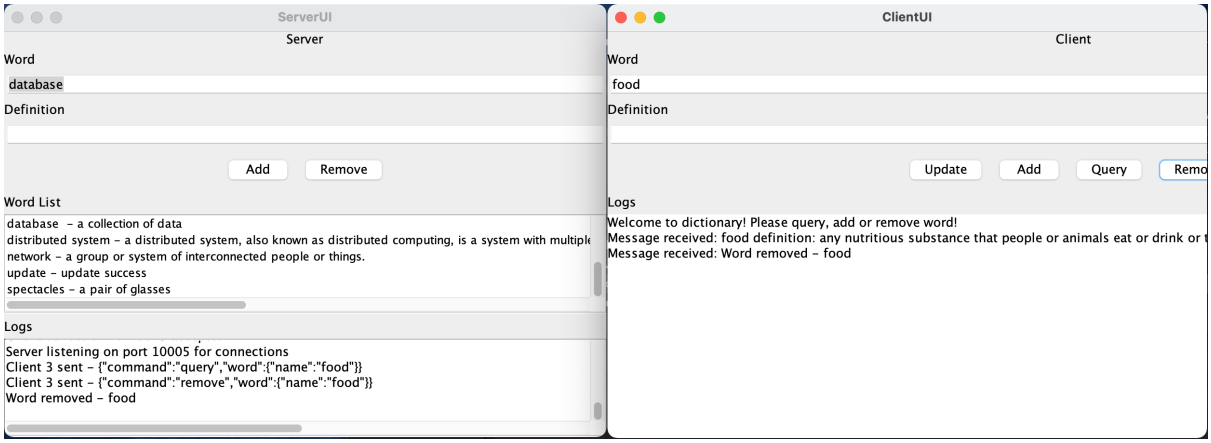

Figure 6 Removing a word

Server Controls

We have also implemented server controls where the server can also add or remove words. This is to give the server an easier access to the words so it can remove or add words when necessary. Figure 7 shows an example of how a server can also add or remove words.

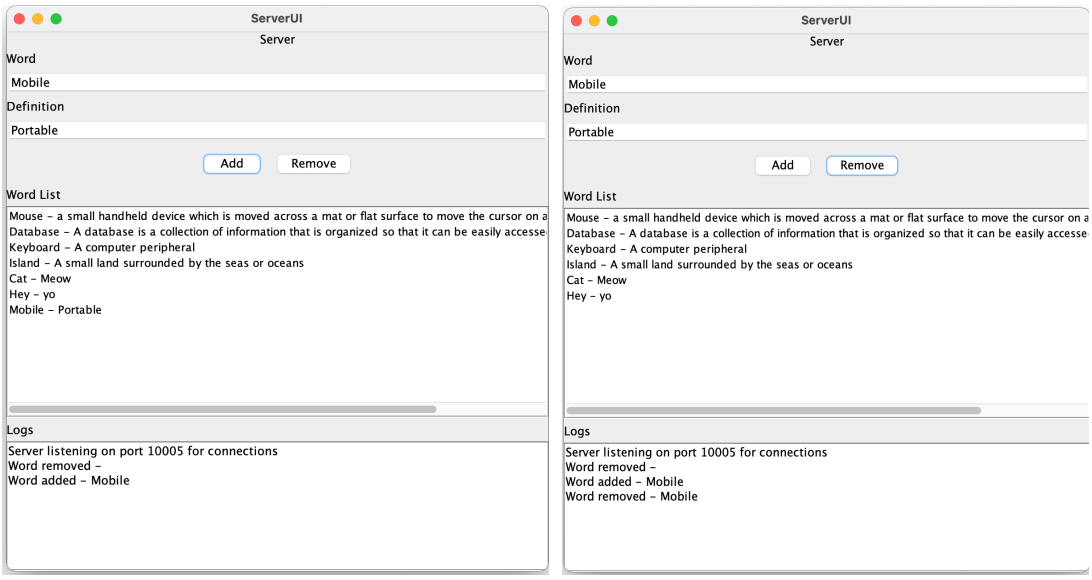

Figure 7 shows adding and removing words from the server.

Exception handling

There are many exceptions which we try to catch within a try catch loop. However, it is important to display the right error messages so the users can diagnose the errors. Below are some examples of exception handling.
Figure 8 shows a code fragment of how exception handling is done. This one is for catching errors related to the port being in use. Figure 9 shows 2 windows of the server, the first one is connected to port 10005 and second one also tried to connect to the same port but received an error message, "Port is in use. Please exit and try again."

```
    }
    catch (BindException e) {
    System.out.println("The port is in use please exit it and try again."+"\n")
    return;}
    catch (IOException e) {
        e.printStackTrace();
    }
```

Figure 8 code fragement for BindException e from server.java

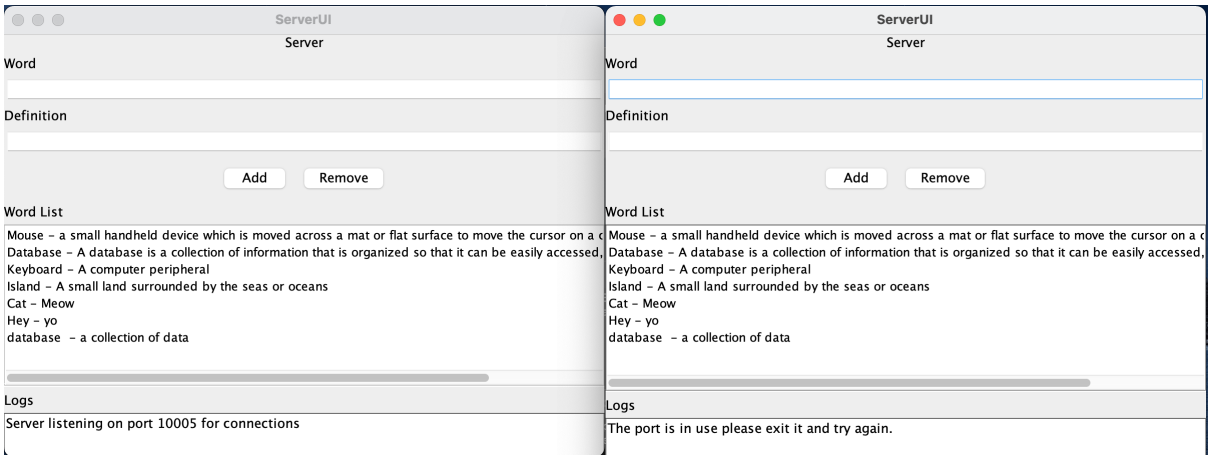

Figure 9 Exception handling of port in use

Figure 10 shows handling of I/O and Host errors related to socket etc

```
catch (UnknownHostException e)
{
    //e.printStackTrace();
    System.out.print("A host error occured");
}
catch (IOException e)
{
    //e.printStackTrace();
    System.out.print("An I/O error occured");
}
finally
{
    // Close the socket
    if (socket != null)
    {
        try
        {
            socket.close();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
```

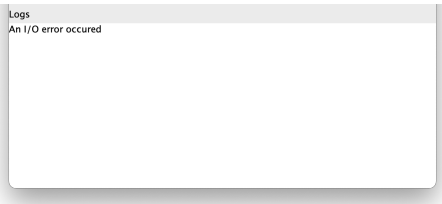

Figure 10 I/O and Host error handling

Other errors related to the functions of the system

We have also designed the system to handle invalid inputs such as querying or removing words which don't exist as well as adding words which are already present in the dictionary.

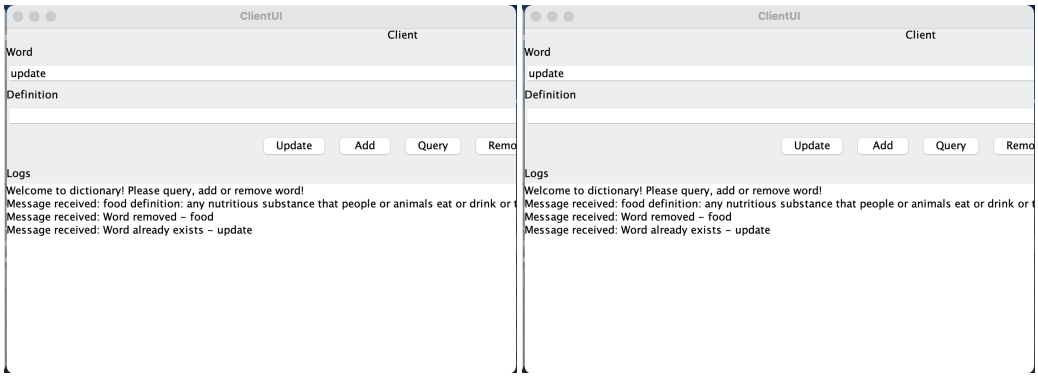

Figure 11 Adding existing word (Left) and removing non-existent word (Right)

CRITICAL ANALYSIS

In this section, we will be discussing more on why we made our design choices:

1. **Transport Protocol : TCP**
   Firstly, we will talk about the transport protocol we used. We have used the TCP as it is used on top of IP to ensure reliable transmission of packets. TCP establishes a connection before communication. This ensures more reliable connection compared to UDP due to it's acknowledgement scheme. Furthermore, it's much easier to implement TCP than UDP.

2. **Thread model : Thread per request**
   We chose this thread model because it's independent of the other threads. If any error occurs in any of the threads then no thread is disturbed. However, this threading method is not scalable. If the number of clients increases more than the available threads, then the system might crash. A better implementation would be the worker pool architecture. For the simplicity of this project, I chose a thread per request.
3. **Dictionary : JSON**
   We chose the JSON format for sending commands between the server and the client as well as storing the files in the dictionary.txt file. The reason for using JSON is because it uses less data overall, so the cost is minimised and the parsing speed is maximised. It's also very straightforward and readable and its easier to map no matter what the language used is.
4. **GUI**
   The GUI is pretty straightforward and easy to operate; its self-explanatory. I implemented an extra space for the server to show the list of words available in the dictionary as I felt that as a server we should be able to remove or add words easily without having to enter as a client or go through the raw data.

CONCLUSIONS

To conclude, we have successfully built a working dictionary server using TCP client-server and multithreading architecture. We all the basic functionalities required by the project as well as some extra features like the word lists and log for the server. We have also analysed our design choices and how it could be improved in the next prototypes.

REFERENCES

[1] R. Buyya, COMP90015. Class Lecture, Topic: “Inter-Process Communication: Network Programming using TCP Java Sockets” Department of Computing and Information System, The University of Melbourne, Melbourne, 2021.

[2] R. Buyya, COMP90015. Class Lecture, Topic: “Multithreaded Programming using Java Threads” Department of Computing and Information System, The University of Melbourne, Melbourne, VIC, 2021.

[3] R. Buyya, COMP90015. Class Lecture, Topic: “Distributed System Models” Department of Computing and Information System, The University of Melbourne, Melbourne, VIC, 2021.