Distributed Systems: Distributed File Server System

1

1.1 Why can RMI simplify the construction of distributed systems, comparing to using socket programming?

One of the main differences between RMI and Socket Programming is that Socket Programming requires you to handle all of the formatting of messages travelling between client and server. In RMI I can pass complex types (such as full objects) as arguments, this simplifies much of the data transfer from the client to the server. In Socket Programming we would have to decompose such an object into primitive data types, and format our argument in such a way that after transferring it to the server we could reconstruct the object. Similarly to this we have that when receiving data using Socket Programming we must specify a buffer size, if this buffer size is too vague then it can often lead to variables being assigned incorrectly and other complications. Thus extra computation is required in order to determine the appropriate buffer size, RMI however not only accepts complex types as parameters but also allows us to return complex types and these returned values can easily be assigned to values (removing the extra computation).

In socket programming we need to bind the server to a specific port which we can then initialise a connection with on the client-side, this network specific code can often be confusing or difficult to make sense of. RMI hides a lot of this network specific code, and allows us to focus on other aspects of the system. In RMI there is around 3-4 lines of code needed in order to declare a server and assign it a logical name, and apart from this we can use it as we would any other object when programming. This makes it easy to write servers for full-scaled distributesd systems quickly - thus it is also easy to maintain.

RMI makes use of multi-threading, allowing the server to exploit threads for better concurrent processing of client requests, which obviously is more efficient and faster. In socket programming we need to initialise each individual thread and it adds an element of complexity and something to confuse the program, developing is made easier by it being 'behind the scenes'. It also means that we need not add any extra code in order to perform efficiently compared to in socket programming when we need to create threads.

RMI uses a Garbage Collector which Socket Programming does not, this means that any system developed in RMI provides memory management facilities as default where as in Socket Programming we would need to implement them ourselved. The Garbage Collector means that when the client creates a remote reference after the client has finished with the remote reference it is removed from memory, it only 'hold' the reference for a short time.

RMI can make use of Dynamic Class Loading which socket programming cannot, this allows the laoding of java code that was not known about prior to the running of the program. From this when instances classes are loaded we are able to make use of methods for that class. As you will see in my assignment and I'm sure you can imagine this makes it easy to make use of methods for the frontend, and interact with the server through the frontend.

RMI naturally uses method-call semantics which are more familiar to many developers such as myself, and a much better fit when we have objects such as servers and clients compared to sockets. Socket programming however is rather unintuitive and difficult to pick up as it's not really similar to the majority of other programming.

RMI uses a stateless protocol and and RMI connections do not implement the notion of a session, this makes it easy for clients to quit and reconnect without affecting how the server functions.

RMI allows for convenient and easy object-to-object communication making it very easy to transfer data between client, frontend and server this way functions are easily excecuted and

data is easily transferred. Socket programming only communicates via transmitting bytes which as we have talked about can be a hastle to decode if their encoding is not given to us.

The fact that we do not have to format data in RMI makes it easy for us to perform functions and to implement validation as we do not have to constantly worry about whether the data we are validating is in the format we would like it to be. This is the opposite of socket programming where the format of data is of critical importance to both receiving and sending data.

In socket programming when transferring a file from client to server there needs to be two 'loops' one for the client side and one for the server side, in order to make sure in the case of uploading that the file is uploaded correctly we need to make sure the loops are at the same point at all times. However, in RMI I was able to call one loop on either side and call a method on the other side iteratively, this ensured proper upload of the file without having to write two loops maticulously.

As with the above, I believe that in RMI there is likely to be less code redundance as if we continue with that theme many functions such as transferring data just repeatedly use simple methods rather than having to repeatedly send and receive data.

1.2 How does the front-end in a distributed system facilitate access transparency to data replica?

Data Replication is simply the process of storing data in more than one node, in this case we have that data can be stored in multiple (3) different servers. This is necessary for improving the availability of data, if each time we were to upload a file with high relaibility then this would mean that we would have 3 copies of our data stored in 3 different servers. For obvious reasons this is much more secure than storing one copy of the data on a single server that could easily become unavailable. This to ensure that we are able to access our data most of the time we want to store many replicas of the data.

However, this can be somewhat inconvenient as when we uploaded data a server may have been unavailable in which case the data was only uploaded to the running servers and it is often difficult to keep track of which servers have which data.

A transparency is an aspect of the distributed system that is hidden from the user. A transparency is often provided below the interface in order to make data easier to interpret or hide some technicalities from the user.

In our case we implement a multitude of transparencies throughout the system. In order to explain this, I will describe these transparencies in the scenario of a user interacting with the system.

Initially, the user is likely to 'CONN' (CONNECT) to the distributed system, however at this point the user has no idea whether the frontend is interacting with one server, two servers or even three - only that it is connected in some way.

The user may now proceed to input other operations, I will discuss these operations in the order they were given in the assignment.

When the user inputs 'UPLD' to upload they are asked for the file name of the file they wish to upload. After this they are asked if they wish to upload with high reliability. In the case that they upload without high reliability the file is uploaded to the server with the least number of files, however the user again has no idea which server this is. In the case they opted for high reliability the file is uploaded to all currently running servers but again the user is oblivious to which servers are currently running. This avoids the user having the inconvenience of choosing which server to upload to themselves and the user need not be aware of the best server to upload to.

When the user inputs 'LIST' the files in every server directory are listed however the only information presented is that these files are present on at least one of the servers and we have no idea where it is they are stored. This prevents an inconvenience the user may experience from being bombarded with repetitions of the same file being presented them. However, it does not

take away from the fact that having multiple replicas of the file ensures the file is less vulnerable.

When the user inputs 'DWLD' to download as this downloads the file from a random server and the user is oblivious to where the file has came from. In turn this means that the user did not have to locate the file in order to download it he just needs to input the file name of some fle he wants to download that is present somewhere on the system. This combined with the 'LIST' function means it is very easy for the user to download a file without having to actually locate it etc.

Similarly to the above when inputting 'DELF' to delete a file the user need not navigate through every directory and delete the instance of it as the 'DELF' function ensures every instance of the file is removed from the system - this is critical when malicious files such as those containing viruses are present on the system.

In summary the system is able to hide away a lot of the technicalities that are of no user to the majority of users and simply perform the functions which are of use to them. It also ensures the user is about to maintain many copies of files and thus make sure that they are secure without having to go through tedious tasks of finding them in order to download them etc. The access transparency in the system makes the system both far more reliable and far more convenient.

2

2.1 Draw a simple diagram to depict the distributed file server system that you have implemented. The diagram should clearly show how you design the communications among servers, a client and the front-end.

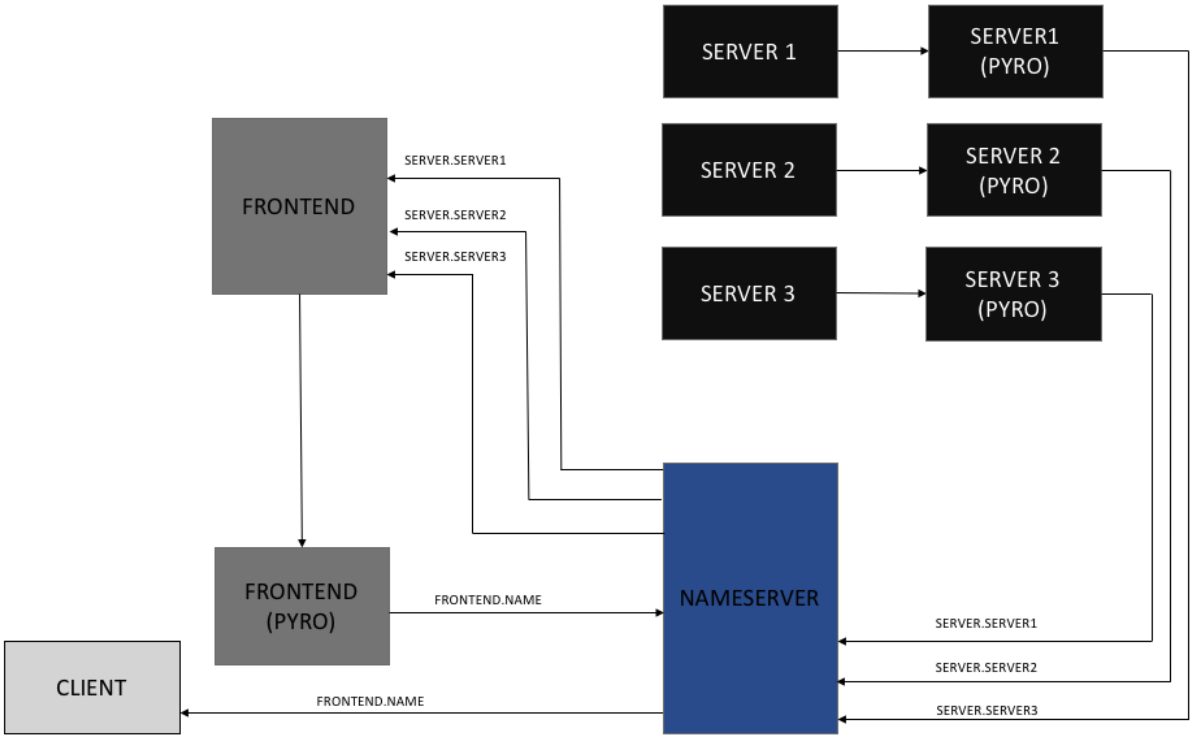

Figure 1: Diagram depicting the distributed file server system implemented.

Brief description:

The diagram shows that we have 3 Server files/programs and each of them have their own Pyro object. Each of these then register a name with the nameserver (as shown).

Within my frontend program (outside of the class), I access each of the servers individually (and in my implementation you will notice I store them in a list). Similarly, the Frontend also has a Pyro object which registers with the nameserver and the Client accesses the Frontend Pyro object via the nameserver.