**DISTRIBUTED SYSTEMS**

# Question 1a

Fundamentally RMI simplifies the construction of distributed systems due to the higher level of abstraction it provides, so the developer needs to consider less aspects of how the system works. In the most basic level a system based on RMI may not even need to concern itself with how the components are connected (e.g. which protocol is used). At the very most the developer only needs to instruct the RMI middleware which protocols to use but does not have to deal with the specific implementation. This is further extended to the transfer of data between components. When using sockets, we need to specify exactly what data we are transferring, as well as validating it on the receiving end. Again, a system based on RMI can abstract nearly all of this out, so that the developer only needs to handle when components disconnect or cause some sort of exception (which would normally result in disconnecting/restarting the component). While it may be possible to create a system based on socket programming fairly easily, handling errors/edge cases quickly makes the task much greater, even more so when dealing with multiple components that could fail at any time.

Systems based on RMI also simplify the construction by providing a registry. This provides a standardised way to connect/disconnect components, so that the developer does not have to concern themselves with the details. While they still must obtain the stubs to each object and handle when objects are disconnected or become available again, the registry greatly simplifies and streamlines the process.

Systems based on RMI could potentially vastly simplify cross language communication if the interface description language used supports it. This could allow one set of components that has already been written in one language to be seamlessly integrated with another language. For example, a python component that utilises OpenCL to perform GPGPU image processing could be called from a component that uses a completely different language. Using RMI would make it much easier even when primitive data types are sent across, because there may be differences between languages that an IDL can resolve in the background (eg. one language may store integers in big endian, whereas another may use little endian).

However, in practice many IDLs only support a single language. This is a worthwhile trade-off though, since system components will normally be written in the same language, and multiple IDLs/RMI systems could be used if required. More importantly IDLs that only support a single language allow for complex data representations to be transferred. For example, in Java by using the standard RMI library we can send any object across, and we do not have to concern ourselves with the complexity of object serialisation/deserialisation (which we would do if we used sockets).

# Question 1b

A front-end in a distributed system facilitates access transparency to data by hiding how each component is represented and accessed. Different replica components may use different data structures/models to store/transfer their data. The front end needs to be aware of these differences between components, but it can hide this from the client either by converting the data before giving it to the client, or by instructing the components to output data in specific formats. This goes the other way as well; the front end may need to convert the data received from the client or instruct the components to process the data in a specific way. In all scenarios the front end makes sure the client does not need to know the specifics of each replica. The way in which a component is accessed not just depends on the type/format of data, but also the underlying protocols. If components use different protocols, then the front end can facilitate transparency by handling the specific implementation for each component silently without the client’s knowledge.

If components are replicated to store data, then the front end can facilitate access to this by giving the appearance that the components are one single homogenous component. That way it doesn’t matter which components store the required data, if the front end can collate the required data set correctly and give it to the client so that it appears that they have only communicated with a single server.

For example, after receiving a request from the client the front end will then forward the request to the ‘hidden’ servers. Some operations (such as downloading a file), may only require a single server (that contains the file), whereas other operations (such as uploading a file) may require all servers to process the request. The front end is responsible for coordinating the operations on the replicated servers, and then processing the response after the servers execute to make sure an agreement has been reached. This allows the front end to facilitate access to the data replica, without the client having to know anything about the replica.

The replication model/architecture used is also hidden by the front end. For example we might assume that the front end uses a passive replication model for the data replica. However, by using a front end we could use active replication instead. The front end may use this instead if it wants more deterministic behaviour at the replicas, better performance when a replica becomes unavailable, and is prepared to deal with the reduced byzantine fault tolerance. There are more aspects to consider when choosing how data is replicated among components, and there are many ways and implementations to do this, but the important thing is that by using a front end this is hidden from the client.

# Question 2

Front End

File Servers (1 – 3)

**File Server registers object with registry**

Delete

Download

List

Upload

Client retrieves front end stub from registry

**Front End registers object with registry**

Delete

Download

List

Upload

Front end retrieves file server stubs from registry

Client

Registry