DISTRIBUTED SYSTEMS

by vzbf32

Part 1

Question 1a

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

Socket programming is aimed at low-level network communication. Applications directly interact with the network interface and the packets that are sent. RMI, on the other hand, is aimed at high-level application-to-application distributed computing, providing a common programming abstraction and infrastructure for constructing distributed applications.

RMI has the innate advantage in that a programmer can remotely invoke a function on a remote object without needing to deal with the intricacies of sending/receiving commands over a socket. This makes programming much easier and more reliable, as the RMI API abstracts all those difficulties.

RMI allows us to distribute our objects on various machines and thus we can dynamically invocate new versions of remote objects.

RMI is based on normal programming ideas of library integration, method invocation and parameter passing.

Threads are handled by the RMI API, and RMI also handles the serialisation / marshalling of objects to save the programmer having to do this and manually send the data over a socket.

The RMI registry keeps track of which objects are available over the network, and this reduces complexity for the programmer. They can simply request an object by name from the registry (or the name server, in the case of `Pyro4`) and the registry will provide an interface for interacting with it.

Question 1b

How does the frontend in a distributed system facilitate access transparency to data replica?

A distributed system allows data/services to be replicated by maintaining multiple copies of them. The client never sees the details about the individual copies; the frontend handles the abstraction so that the client only sees one frontend.

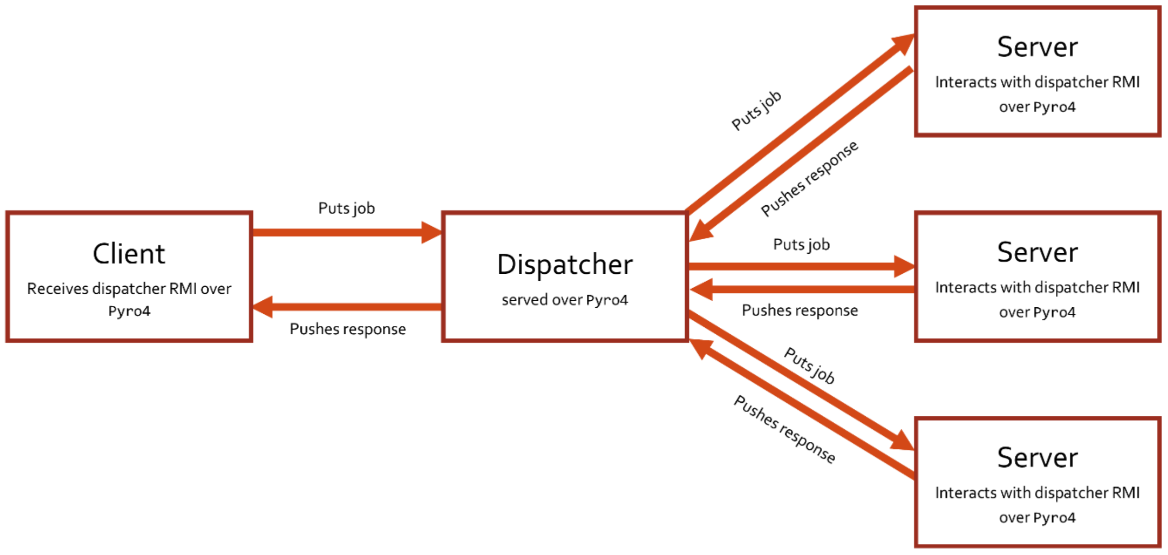
Data consistency is key here – there cannot be conflicts in the data. There are two important parts to the system, the replicas and the frontend. The replicas maintain data/service replicas on the servers, and they process requests and store results. Client requests are handled by the frontend and then pushed to a replica.

The frontend makes replication transparent. It monitors and maintains replica availability and performs request distribution, as well as collating responses.

By keeping track of which replicas have which data, the frontend maintains a level of abstraction available to the client with which the client can interact to pull data from the servers without needing to interact with them directly. This makes it easier for the client to access data replica without it having to do any load balancing, concurrency control or synchronisation itself, and allows the frontend to do any additional processing on the replica's response if needed.

Part 2

The following diagram shows how the client, frontend and the servers communicate using `Pyro4`. We use a simple queuing system, where we push a job (which consists of a command, such as `upload`, and some data, such as the file name and contents) to another component and wait for them to push a response back to us.


Client
Receives dispatcher RMI over Pyro4
Puts job
Pushes response
Dispatcher
served over Pyro4
Puts job
Pushes response
Server
Interacts with dispatcher RMI over Pyro4
Puts job
Pushes response
Server
Interacts with dispatcher RMI over Pyro4
Puts job
Pushes response
Server
Interacts with dispatcher RMI over Pyro4