DC ASSIGNMENT 1: RMI for supporting/building legacy services

Group Number : 7
Varad Kshemkalyani D17A, 37
Heer Kukreja D17A, 38
Gaurav Marwal D17A, 43
Pratheek Menon D17A, 44

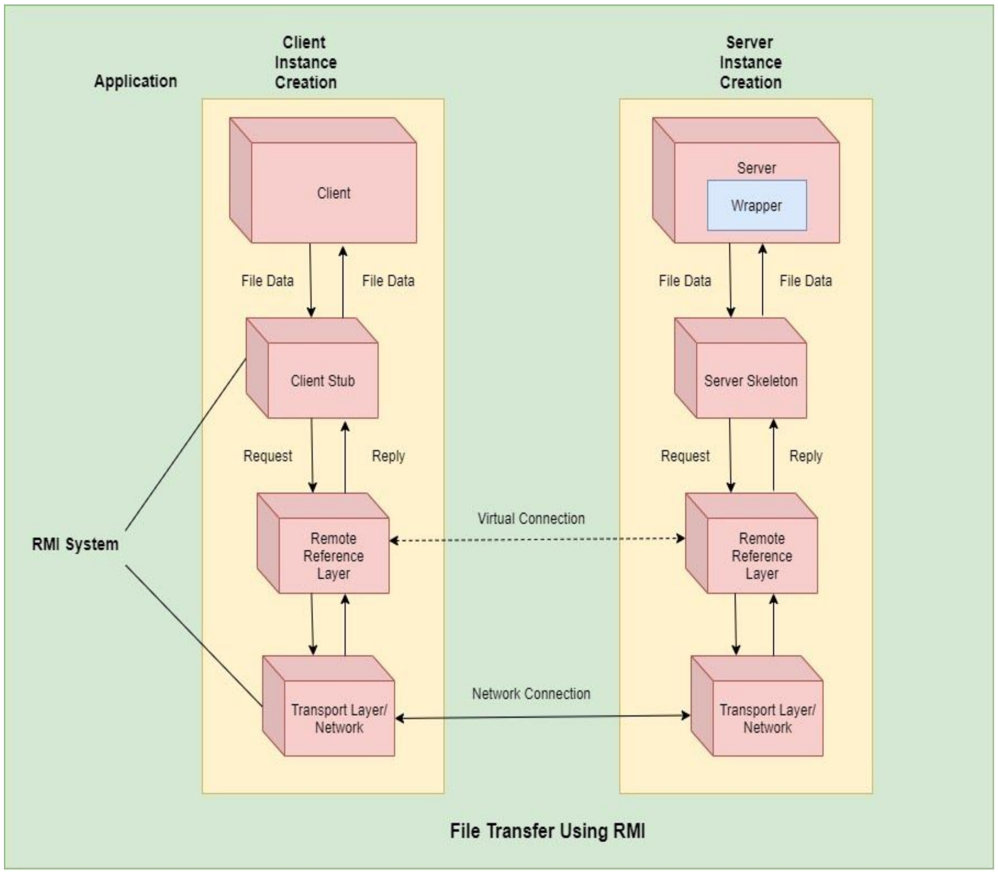
Introduction

In today's world the use of middleware in inter-system communication has become very prevalent. For distributed applications, communication between two different applications is frequent and imperative.

Remote Method Invocation (RMI) is an API provided by java which allows one object residing in one JVM (Java Virtual Machine) invokes objects running on the JVM of the same machine or a remote machine.Thus, without using any other external communication mechanism Java applications communicate with each other directly in real-time and in a secured manner.

Legacy services are services which are outdated but are still in use because they are capable of serving the purpose they were originally designed for, to a significant extent. The main reason why they usually become 'outdated' is that they don't allow for growth. Because of the older technology, the newer systems/services have trouble interacting with the legacy services. RMI can serve as a bridge in these cases and enable them to avail the features of the legacy services.

Design



Working Concept and Layered Architecture

In distributed communication systems, RMI architecture is at the heart of communication between core java based applications. The RMI provides an interface for the client program that resides on the client machine and server program residing on the server machine. The two machines are either the same or remote machines.

At the client side, the system will be able to upload files to the server or download files via the server, explore files/directories or even create or delete files/directories. For all these functions the system provides an interface containing method signatures and an implementation file for execution of these functions. The system uses the JAVA I/O stream class objects to execute these methods.

Layered Architecture:

- Application Layer:
  This is where the actual system application i.e client and server for file transfer between the machines involved in communication. The java program on the client machine invokes a method to a remote object, it converts the file to byte stream and invokes the upload method via the interface object.
- Stub/Skeleton Layer:
  Client once ready to send file/receive, calls on the stub object, the stub forwards this request to a remote object (Skeleton) via RMI infrastructure which is then executed on the server.
  The skeleton then receives the parameters(filename, directory, etc.) for file transfer and passes it on to the server for method implementation and then passes the result back to the client via the same RMI interface.
- Remote Reference Layer:
  The proxy layers - stub & skeleton are connected to the RMI mechanism via this layer.
  It is the middleware between the stub/skeleton layer and the underlying transport layer protocol. This layer receives the request for transfer, and it invokes the object of the same layer on the server side after accessing the registry for the object reference.
- Transport Layer:
  The transport layer is responsible for connection setup, connection management, and keeping track of and dispatching to remote objects (the targets of remote calls) residing in the transport's address space. This establishes the connection when the transfer is required.

So, when the client requests for a file using a method via an object which is accessed via the registry; the method call is first processed by the stub; which initiates a call to a remote object by calling the reference layer method. The reference layer then passes this request to the server side reference layer via the transport layer; which in turn passes the request to the skeleton. The skeleton then finally invokes the object at the server. The file is then transmitted to the client in reverse direction.

Explanation and Justification of communication model

a. Client/Server Design : Stateless/ Stateful

A stateful server is one which persistently maintains information about clients and their actions. For eg. in our case of a file transfer service, making the server stateful will allow it to maintain a table storing (client,file) pairs whenever any updates are made to the files. Maintaining such a table will enable the server to constantly monitor as to which client has permissions to update/modify which file which also includes the later versions of the file.
A stateful approach provides better performance in terms of read write operations from the point of view of a client. This performance improvement is very often an advantage over stateless designs.
However the stateful approach has its disadvantages in the way that if the server ends up crashing, it will have to recover the table with the (client,file) entries when it is back up and running. Otherwise the server won't be able to guarantee that the files in the system are the most recent versions. That is, the server will have to retrieve the exact state it was in before the crash. This aspect of recovery can introduce a significant amount of complexity into designing the system.

b. Server Creation Semantic

Server creation semantics refer to the manner in which the servers are instantiated and destroyed(for eg. after timeout or garbage collection). Server creation is application specific and can be of three types :

1. Instance-per-call : Server exists only for the duration of the call
2. Instance-per-session : Server exists for the entire session in which client and server interact
3. Persistent server : Server exists indefinitely

Java RMI provides a way to change server creation semantics through Remote Object Activation. The servers can either be persistent or can be instantiated on an as-needed basis.
In a file transfer system, multiple clients can upload or download files throughout the day. In such a situation, allocating and deallocating servers and their respective resources can be a time and resource consuming process. Low latency outweighs the merits of resource saving and hence a persistent server would be the optimal choice for a file transfer system.

c. Persistent and Transient Communication

Persistence means that, for an indefinite period of time before the next recipient is ready, the network is able to store messages. Transient indicates that the message is only stored as long as it is ready for the next recipient. File transfer is transient in our system because files can only be transferred while both systems are operating, i.e. the sender and the receiver. This will minimize the use of a buffer that is used to make a system persistent.
A transient system is often vulnerable to many faults that can lead to data loss in the process. A download or upload of a file can experience different types of transient failures:

- One or more data servers have failed.
- A connection to a network fails.

- The host PC will be turned off or the core client will leave.

Communication failure has to be quickly identified in the case of transient systems and the appropriate protocol has to be followed immediately.

d. Synchronous / Asynchronous Communication also (Request, Request/Reply, Request/Reply/Ack, Callback nature, Receipt based, Delivery based, etc.)

A synchronous communication model means that after each invocation, the calling process has to wait until a reply is received from the callee. Java RMI's synchronous model may cause scalability challenges when the server has to execute processes for long durations, which causes longer wait times for clients and hence reduces throughput. One way of overcoming this difficulty is adopting an asynchronous mode of operation. An asynchronous invocation allows the client to continue with its computation after dispatching a call, thus eliminating the need to wait idle while its request is being processed by a remote server. File transfer can require any amount of time from a few seconds to hours. Expecting the client to wait until the transfer is complete will be illogical. In such a scenario, asynchronous communication between the client and server with callbacks to confirm completion of file transfer will be optimal.

e. Call semantics

Request-reply protocols, such as RMI, can be implemented in different ways to provide different delivery guarantees. Call semantics pertain to the different interpretations of the reliability of the remote invocations as seen by the client. RMI implements "at most once" semantics i.e if a method returns normally, it is guaranteed to have executed once. However, if an exception occurs, the client assumes that the method will have been executed either once or not at all. In the second case, the client may wish to attempt the invocation again. In the file transfer services, faulty or incomplete transfer can not be tolerated. If the connection is lost due to some error, the transfer should start from the beginning and hence "at most once" semantics will be used.

f. Concurrent Access to Multiple Servers

Regulating concurrent access to multiple servers is a way of optimization, since most distributed applications would benefit from it. This may be achieved by using threads wherein, while designing the client process, each of the threads can be used to make independent remote procedure calls to distinct servers. However, this method would need prompt addressing mechanisms in the underlying protocol for accurate response routing.
Another way is the use of 'early reply' wherein a single call comprises two different calls, with one of them being responsible for passing the parameters and the other for requesting and retrieving the results. However, it comes with a disadvantage that the server will have to hold a call result until the client requests it and this can lead to congestion and unnecessary overhead at the server's end.
In the third alternative, there is no direct interaction between client and server but they interact via a 'call buffer server' which acts as a middleman and buffers the client call's request

parameters along with the server and client name. This allows the client to take care of other activities until it needs the results of the call. At the server end, it periodically polls the call buffer server to check if there is any call for it.

g. Serving Multiple Requests simultaneously

In a file transfer system, the server will sometimes have to cater to the request of multiple clients simultaneously. Multiple clients might request connections to upload or download data at the same time. In this situation, the server can either reply to the servers in a round-robin manner or simultaneously serve all the clients. Since Java RMI treats each client call as a different thread in the same process, as long as the server implementation is thread-safe, each client request can be served simultaneously.

h. Reducing Per-Call Workload of Servers

The per-call workload of servers can be reduced by making use of stateless servers. This will involve letting the clients keep track of the progression of requests sent by them to servers. However, since our file transfer service has the fundamental need for a stateful server this can't be achieved.
Another way is to design a multiple threaded server with the facility of dynamic thread creation according to the requirements. However, this will introduce significant complexity in the server design and implementation..
The third and more achievable alternative is to simply keep the client requests as short as possible, containing only the necessary information, which will automatically keep the work involved in each request to a bare minimum.

i. Reply Caching of Idempotent Remote Procedures

Idempotent Remote Procedures have no additional effect if called more than once. These procedures can be easily handled, hence the reply from the server can be cached to make a fixed call whenever the procedure is executed. This helps our system to be more efficient and robust. If the Idempotent Remote Procedures in the file transfer increase by a large number this will help us to save a lot of time and utilization resources. This will enable us to make the system more fast and less costly.
The procedure in Idempotent may be done more than once. The manager routine must ensure that executing the same input arguments more than once does not produce unwanted side effects. An idempotent call request may be regarded as a non-idempotent call by the implementation of the RPC protocol machines. This transformation is valid. Maybe semantics and transmission semantics are provided by RPC as special types of idempotent operations. An idempotent call, including transmission, guarantees that the information for an RPC is received and processed zero or more times with the RPC communication protocols.

j. Proper Selection of Timeout Values

The default inactivity time is about 300 seconds. For cases when the server machine is down or there is an error in transmission, in around 420 seconds which is the default transfer timeout; the client object invokes the server object again. Hence, when the server is active retransmission can be carried out after the timeout. Similarly, when there is a failure at the client machine, retransmission is carried out after the default transfer timeout.

k. Proper Design of RPC Protocol Specification

RPC is an abbreviation for Remote Procedure Call. It supports procedural programming. RPC is a library and OS dependent platform. RPC is the older version of RMI and is less efficient as it creates more overhead as compared to RMI. Also RPC doesn't provide any security along with high development costs. There are multiple codes needed for simple application in RPC.

RMI stands for Remote Method Invocation and supports object oriented programming. In RMI, objects are passed as a parameter rather than ordinary data. RMI is a java platform. It's also more efficient than RPC. Less overhead is created in RMI as compared to RPC. RMI also is more secure as it provides us with client level security. RMI doesn't hide distribution in language also RMI stubs are not needed to be compiled into the client unlike RPC. They are downloaded at runtime.

Both RPC and RMI follow the call/return style. In this the communication is started by the client and the server responds to this call. Both use local stubs which support interfaces for method calling. Message passing is used to implement calling and returning. Separate mechanism is used for dynamic binding like object registry and JINI.

RPC was designed to support communication between various languages on the other hand RMI uses similar languages at both the ends.