L06b. Java RMI

Java Distributed Object Model:

- Much of the hard work needed to build a client/server system using RPS (marshaling/un-marshaling, publishing the remote object on the network for the clients to access, etc.) are all managed undercover by the Java Distributed Object Model.
- Java Distributed Object Model concepts:
  - Remote Object: Accessible from different address spaces.
  - Remote Interface: Declarations for methods in a remote object.
  - Failure Semantics: Clients deal with RMI failure exceptions.
- Similarities/differences between local objects and remote objects:
  - In Local Object Model, the Object references passed as object invocation parameters, they are passed as a pure reference (i.e. if client changes the object, server will see the change).
  - In Distributed Object Model, the Object references passed as object invocation parameters, they are passed as Value/Result (i.e. if client changes the object, server will not see the change because a copy of the object is actually sent to the invoked method).

Local vs. Remote Implementation:

- Reuse of Local Implementation:
  - Extend a Local Implementation of the Account Class to implement Bank Account.
  - Use the Built-in Class called Remote Interface to make the methods in Bank Account visible remotely on the network.
  - Only the interface is visible to the client and not the actual implementation or the instantiated objects.
  - The actual location of the object is not visible to the client. Therefore, the implementer has to do the hard work of finding a way to make the location of the service visible to clients on the network (i.e. Instantiated Objects).
  - In this case, we used the Local Implementation and used only the Remote Interface to make the object instances remotely accessible. So, the all the hard work of making the object instances remotely accessible needs to be done by the implementer. This is why this approach is **not preferable**.

    However, this approach has the advantage of providing fine-grained control on selective sharing of services.

- Reuse of Remote Object Class:
  - Extend the Remote Interface so that the Account Interface now becomes visible to any client that wants to access the Object.
  - Extend the Remote Object Class and Remote Server Class in order to get the Account Implementation Object.
  - Now, when we instantiate the Account Implementation Object, it becomes visible to the network clients through the Java Runtime System.
  - The Java RMI system is responsible for all the hard work of making the Server Object Instance visible to network clients and hence this is the **more preferred** way of building network services and making them available for remote clients anywhere on the network.

How does Java RMI work?

- On Server side:
  - The server object is made visible on the network using the 3-step procedure:
    1. Instantiate the Object.
    2. Create a URL.
    3. Bind the URL to the Object Instance created.
  - This allows the clients to be able to discover the existence of the new service on the network.
- On Client side:
  Any arbitrary client can easily discover and access the server object on the network using the following procedure:
  1. Lookup the service provider URL by contacting a bootstrap name server in the Java RMI system and get a local access point for that remote object on the client-side.
  2. Use the local access point for the remote object on the client-side by simply calling the invocation methods, which look like normal procedure calls.
     - The Java Runtime System knows how to locate the server object in order to do the invocation.
     - The client does NOT know or care about the location of the server object.
  3. If there are failures in any of execution of the methods (functions), then Remote Exceptions will be issued by the server through the Java Runtime System back to the client.
     - A problem with Remote Exceptions is that the client may have no way of knowing at what point in the call invocation the failure happened.

RMI Implementation:

- The Java RMI functionality is implemented using the Remote Reference Layer (RRL).
- The Client-side stub initiates the remote method invocation call, which causes RRL to marshal the arguments in order to send them over the network. When the server responds, the RRL un-marshals the results for the client.

- Similarly, the Server-side skeleton
  - Uses RRL to un-marshal the arguments from client message.
  - Makes the call to the server implementing the Remote Object.
  - Marshals the results from the server into a message to be sent to the client.
- Marshaling and un-marshaling are also called as Serializing and De-serializing Java objects and is done by the RRL layer.
- The RRL layer is similar to the Subcontract mechanism in the Spring Network OS. Java RMI derives a lot from the Subcontract mechanism.
- In summary:
  - RRL hides the details/location of the server (replica, singleton, etc.).
  - RRL supports different transport protocols.
  - RRL marshals/serializes information to be sent across the network.

RMI Transport Layer:

The RMI Transport Layer provides the following four abstractions:

- Endpoint:
  - This is a Protection Domain or a Sandbox like Java Virtual Machine for execution of a server call or client call within the Sandbox.
  - The Endpoint has a table of Remote Objects that it can access (similar to Door Table in Spring OS).
- Transport:
  - The RRL layer decides which transport protocol to use (TCP or UDP Protocol) and it gives that command to the Transport Layer.
  - The Transport listens on channel for incoming connections.
  - The Transport is responsible for locating the dispatcher that invokes the remote method.
  - The Transport mechanism sits below the RRL layer and allows the object invocations to happen through the Transport Layer.
- Channel:
  - The type of the Transport decides the type of the Channel (TCP or UDP Channel).
  - Two Endpoints make a connection on the Channel and do I/O using the connection on the Channel.

- Connection:
  - Connection Management is a part of the Transport Layer and is concerned with:
    1. Setup and listening for client connections.
    2. Establishing the connections.
    3. Locating the dispatcher for a remote method.
    4. Teardown of the connections.
  - Liveness Monitoring is part of Connection Management and is typically done via periodic heartbeats.
  - The RRL layer decides the correct transport mechanism (TCP or UDP Transport and gives that command to the Transport layer.

Conclusion:

- The Distributed Object Model of Java is a powerful vehicle for constructing network services:
  - It dynamically decides how to establish the client-server relationship.
  - It provides flexible connection management in choosing different transports depending on network conditions, client-server locations, etc.
- Some subtle issues in the implementation of the Java RMI system are:
  - Distributed Garbage Collection.
  - Dynamic Loading of Stubs on the Client-side.
  - Sophisticated Sandboxing mechanisms on client-side and server-side to avoid security threats.