# Homework 2

Mayank Mohta (mmohta@andrew)
Yuchen Tian (yuchent@andrew)

Following is an example how our RMI framework can be used for remote method invocation. Using this example we explain our design.

**RMI Registry:** Need to start the java process for RMI registry (RegistryServer.java) before anything

**RMI Server:** Creating and registering remote objects

**Step1 :** Create a new dispatcher.

```
Dispatcher d = new Dispatcher();
```

**Step 2:** Create concrete implementation of Remote object with an “unique name” (interface GreetingGiver extends Remote)

```
GreetingGiver helloGiver1 = new HelloGiver("HelloGiver1");
```

**Step 3:** Register the remote object with the Dispatcer, which will return a Remote Object Reference

```
RemoteObjectRef ror1 = d.exportRemoteObject("HelloGiver1", “GreetingGiver”, helloGiver1);
```

**Step 4:** Instantiate a registry client giving the port where RegistryServer is running (on localhost)

```
RegistryClient registry = LocateRegistry.getRegistryClient();
```

**Step 5:** Bind Remote Objects

```
registry.bind(ror1);
```

**RMI Client:** Locating remote objects and invoking remote methods on them.

**Step 1:** Locate the Registry and create client to it.

```
RegistryClient registry = LocateRegistry.getRegistryClient("localhost", 4444);
```

**Step 2:** Lookup the remote object in the registry.

```
RemoteObjectRef ror = registry.lookup("HelloGiver1");
```

**Step 3:** Create a stub object (responsible for handling communication with RMI server)

```
GreetingGiver g = (GreetingGiver) ror.localise();
```

**Step 4:** Make Remote calls (RemoteObjectReference can be returned or can be passed as parameter)

```
//Example of simple RMI call
g.giveGreeting("Mayank", false)
//Example where ROR is returned (Need to “localise” or make stub from returned ROR)
RemoteObjectRef ror2 = g.locateGreeter("HelloGiver2");
GreetingGiver g2 = (GreetingGiver) ror2.localise();
//Example where ROR is passed as parameter
g2.collectGreeting("Yuchen", ror);
```

**Components:**

**1. RegistryServer (RegistryServer.java) and RegistryClient (RegistryClient.java):**

RegistryServer maintains a hash table mapping Unique Remote Object Id to the Remote Object reference. It is a deamon process which accepts and fulfills requests from RMI Servers and RMI Clients.

RegistryClient is the api which is used by anyone to contact RegistryServer. RMI servers can bind new remote objects (Step 5 of RMI Server) and RMI clients can lookup remote objects using RegistryClient. RegistryClient is responsible for creating sockets, marshalling requests and other things required for communication with RegistryServer.

**2. Dispatcher:** The Dispatcher is the component which serves RMI calls on the RMIServer. An RMIServer, after creating remote objects, first registers them with the Dispatcher (RMIServer - Step 3). Dispatcher embeds the hostname and port in the remote object reference and returns it to the server. The server can then register it with the RMI registry. Clients can fetch this ROR from the registry and call RMI methods which would be served by the dispatcher. The Dispatcher maintains a hash table mapping remote object id to actual reference of the concrete remote object and adds the entry when the remote object is registered by the RMI server. This hash table enables it to route the request to the correct remote object. Since for any remote method, the demarshalling and method invocation logic is same, dispatcher can handle it instead of server side skeleton.

**3. Stub objects:** After getting a RemoteObjectReference, any client that wishes to use it for remote method invocation must create a stub or proxy object using “localise method” of ROR (Step 3 - RMI Client). Using the Proxy class (provided by java) and reflections, an instance of this stub object is returned to the client. This stub object implements the same interfaces which is implemented by the remote object. Hence, client can call any remote method on the stub. Behind the scenes, the Invocation Handler would be invoked, which marshalls all arguments, the method name and remote object id and sends it across to the Dispatcher listening on the other side (RMI Server). The dispatcher would call the method on the concrete object, gather the results and return it back to the Invocation Handler which deserializes the return value or the exception and returns to the client via the stub.

**Improvements:**

1. We were not able to implement automatic retrieval of .class files for stubs but we believe that would be a nice feature to have.
2. We have not implemented a retry mechanism in remote method invocation. Retrying with backoff (constant or exponential) might be a useful. However, what makes it tricky is identifying if the operation already happened in case the remote method invocations are not idempotent. Hence, we believe it should ideally be clients responsibility. But the framework could provide a configurable mechanism to retry to avoid code duplication and help clients.
3. We believe a performance and stress test should be done to gauge how the system behaves under high load