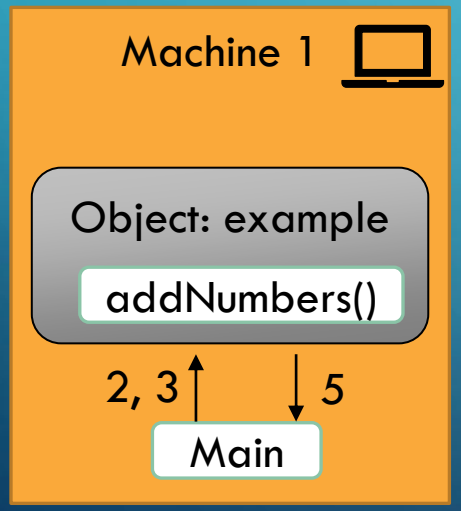
DISTRIBUTED AND CLOUD COMPUTING

LAB3 RPC AND RMI

AN EXAMPLE

- A local method invocation to addNumbers()

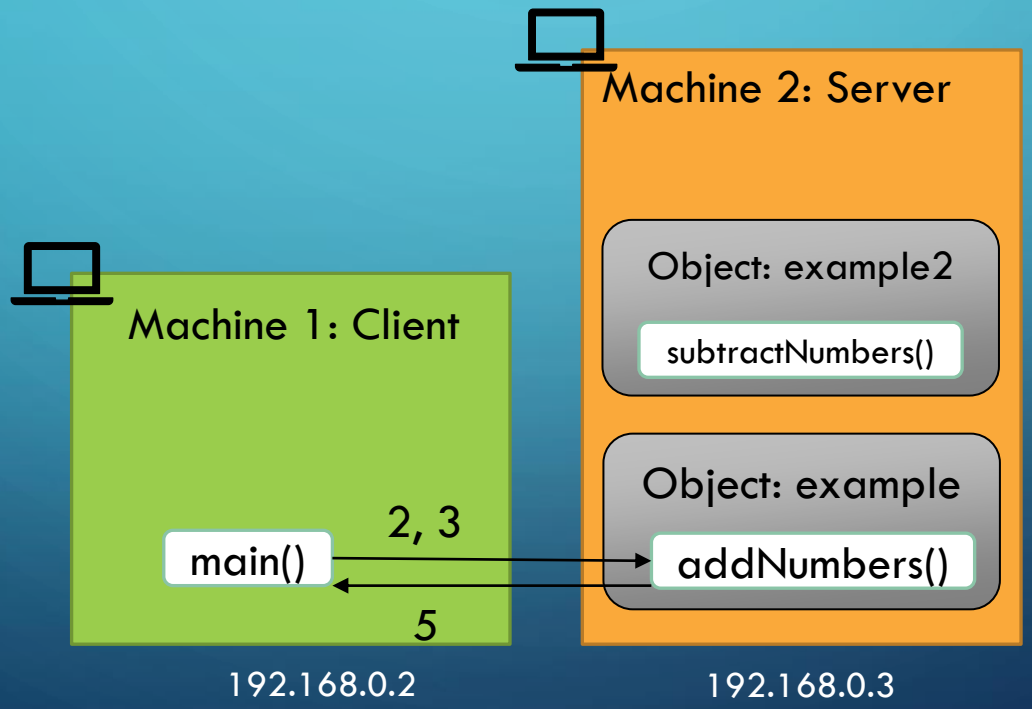



```java
public class Example {

    // Define a method to add two numbers
    public int addNumbers(int a, int b) {
        return a + b;
    }

    // Main method
    public static void main(String[] args) {
        Example example = new Example();

        // Call the addNumbers method
        int result = example.addNumbers(2, 3);

        // Print the result
        System.out.println(result);
    }
}
```

What if addNumbers() is on another machine?

AN EXAMPLE

What if addNumbers() is on another machine?
What if there are other methods in other objects we want to invoke?
How to hide the underlying network configuration when invoke the remote method?



AN EXAMPLE

Machine 1: Client

```java
public class ExampleClient {

    public static void main(String[] args) throws Exception {
        // Look up the remote object in the registry
        Registry registry = LocateRegistry.getRegistry("hostname");
        ExampleInterface example = (ExampleInterface) registry.lookup("ExampleInterface");

        // Call the remote addNumbers method
        int result = example.addNumbers(2, 3);

        // Print the result
        System.out.println("Result: " + result);
    }
}
```

As if we are invoking a local method!

Machine 2: Server

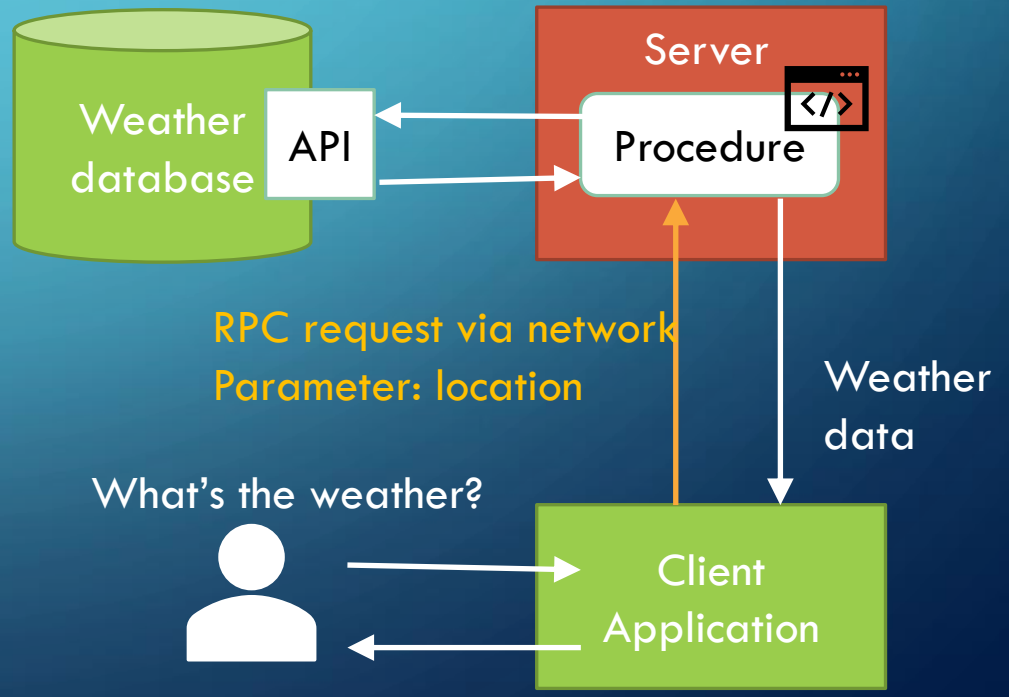
```java
public class ExampleServer implements ExampleInterface {

    // Implement the addNumbers method from ExampleInterface
    public int addNumbers(int a, int b) throws RemoteException {
        return a + b;
    }

    // Main method to start the server
    public static void main(String[] args) throws Exception {
        // Create an instance of the server object
        ExampleServer server = new ExampleServer();

        // Export the server object and bind it to a registry
        Remote stub = UnicastRemoteObject.exportObject(server, 0);
        Registry registry = LocateRegistry.getRegistry("hostname");
        registry.bind("ExampleInterface", stub);

        System.out.println("Server ready");
    }
}
```

COMMUNICATING WITH REMOTE SERVER

- RPC (Remote Procedure Call) is an **inter-process communication protocol** which allows calling a function in another process residing in local or remote machine.
  - Transparency: enabling users to work with remote procedures as if the procedures were local
  - Examples: NFS (Network File System), use RPC to route requests between C/S (client-server)
  - JSON-RPC, via HTTP (multi-language, multi-platform)
- RMI (Remote Method Invocation) is an **object-oriented** equivalent of RPC.
  - Example: Java RMI (language specific)

COMMUNICATING WITH REMOTE SERVER

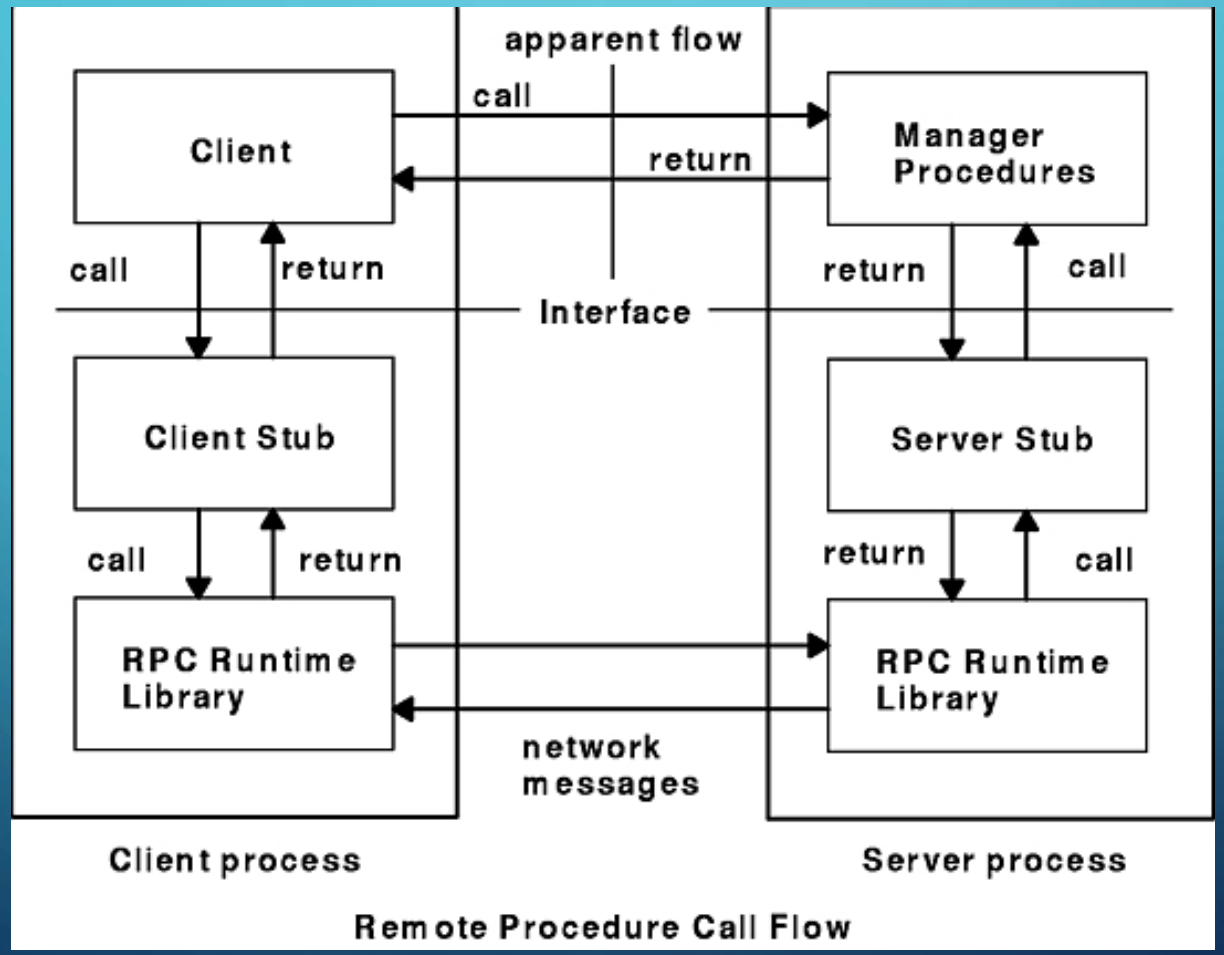
- For instance, imagine you have a client application that needs to get the current weather information for a particular location. The client sends an RPC request to the server with the location information as a parameter. The server application receives the request and executes a procedure to fetch the weather information from a weather API or database using the location information. The server then sends the result of the procedure execution, which is the weather information, back to the client application in response to the original RPC request.



WHY RPC/RMI

- Transparency: hiding underlying complexity; enabling users to work with remote procedures as if the procedures were local
  - What should we know in order to call a remote procedure?
    - IP address of the server
    - Which process on that server manages that procedure
    - Parameters
  - Imagine every time you call a remote procedure you need to all the above in the code…
- RPC/RMI can be used in 2 scenarios:
  - A user (client) want to execute some task on a server (server)
    - An implementation of the C/S architecture
  - A service (client) want to call another service (server) in a system
    - An approach to make it easier to build distributed systems

RPC WORKFLOW



https://www.ibm.com/docs/en/aix/7.1?topic=call-rpc-model

RPC VS. RMI

- RPC
  - Proxy **function**, calling procedure remotely
  - Passing ordinary data structures (string, integer etc.)
- RMI
  - Proxy **object**, can send and receive object
  - Pass objects to remote (serialized object data)

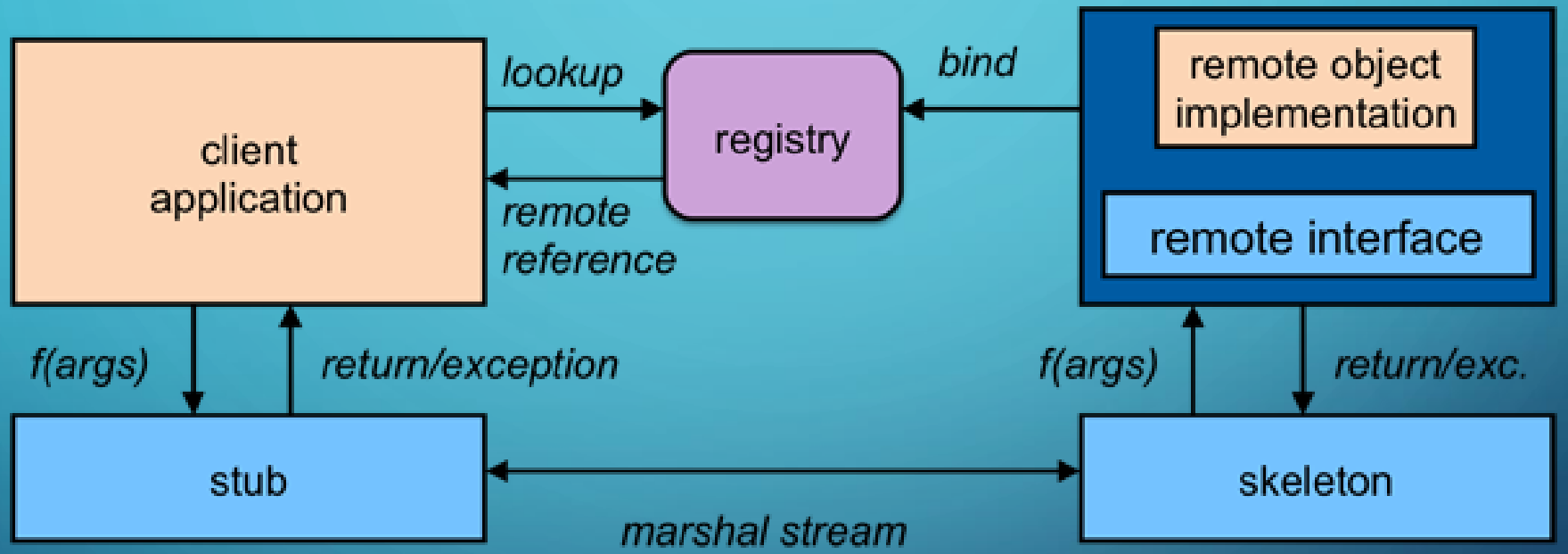
CHALLENGES IN IMPLEMENTING RPC/RMI

- Communication
  - How to communicate between the caller and callee - TCP/UDP
- Addressing
  - How to establish the linkage between function and ID - Registry/API Endpoint
- Serialization
  - How to transmit parameters and return values correctly - Serialization framework

JAVA RMI

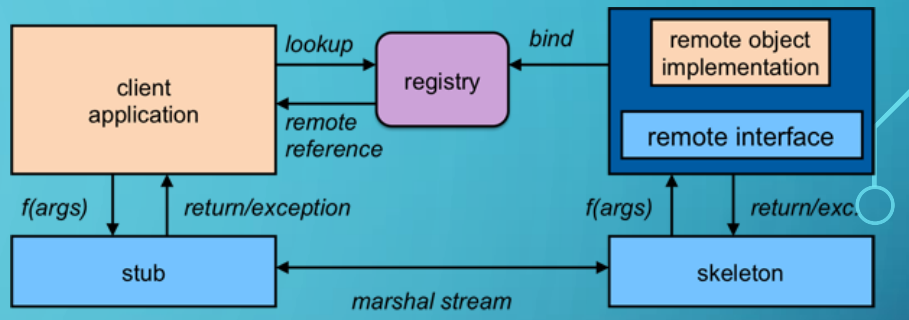
- The Java Remote Method Invocation (Java RMI) is a Java API that performs remote method invocation, with support for direct transfer of serialized Java classes.
- The original implementation depends on Java Virtual Machine (JVM) class-representation mechanisms, and it thus only supports making calls from one JVM to another. The protocol underlying this Java-only implementation is known as Java Remote Method Protocol (JRMP).

STRUCTURE OVERVIEW



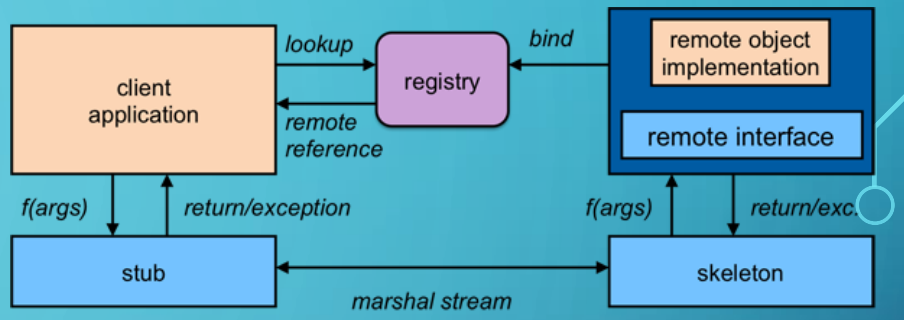
https://www.cs.rutgers.edu/~pxk/417/notes/images/rpc-rmi_flow.png

CLIENT-SERVER INTERACTION



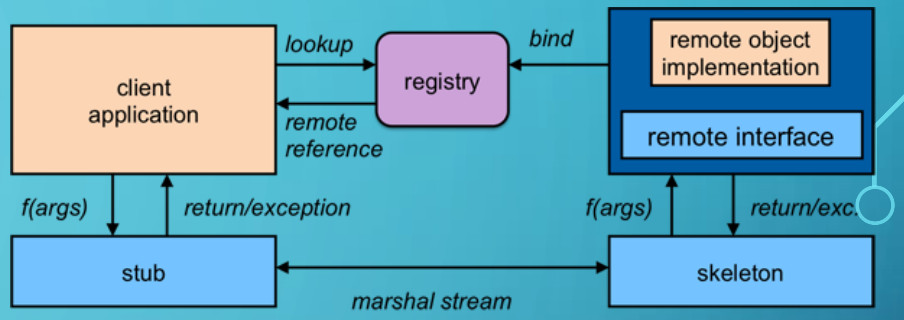
- Client and server both have the information about the *remote interface*.
- Server provides services via remote objects. Server registers its service to registry, generate skeleton to handle incoming method invocation requests.
- Client looks up a service provided by a remote object from the registry, gets the remote reference of that remote object which is implemented as a stub.
- Client invokes remote methods as it's using the local object, and data gets marshalled and transmitted to server
  - The stub packages method calls and arguments into a network message and sends them to the server over the network
- Server unmarshalls incoming arguments, invokes actual implementation of the object, and sends back returned data to client (by skeleton)
  - The skeleton receives the network message, unpackages it, and invokes the corresponding method on the server object.

STUB & SKELETON



- Stub/Skeleton serves as a placeholder at client/server side – **Proxy**
- They don't have actual implementation details of server code, just the interface definition (function name, parameters, return types, etc)
- Communicate using network – typically TCP/IP
- Java object is serialized at one side, then deserialized at another side
- Using **Java Proxy Pattern.**
  https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html

RMI REGISTRY



- RMI Registry acts as a **broker** between RMI server and client
- Stores the relationship between service name and remote reference
- Service discovery, service registration
- Provided by Java runtime. You can find it under Java's binary directory

Modifier and Type	Method and Description
void	**bind**(**String** name, **Remote** obj)Binds a remote reference to the specified name in this registry.
String[]	**list**()Returns an array of the names bound in this registry.
Remote	**lookup**(**String** name)Returns the remote reference bound to the specified name in this registry.
void	**rebind**(**String** name, **Remote** obj)Replaces the binding for the specified name in this registry with the supplied remote reference.
void	**unbind**(**String** name)Removes the binding for the specified name in this registry.

EXAMPLE: RMI HELLO WORLD

The code is available on Blackboard

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface IFoo extends Remote {
   public String getMessage() throws RemoteException;
}
```

Interface definition of the remote object

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Foo extends UnicastRemoteObject implements IFoo{
   private static final long serialVersionUID = -5655647669552931408L;

   protected Foo() throws RemoteException {
      super();
   }
   public String getMessage() throws RemoteException {
      return " Hi from remote Foo!";
   }
}
```

Implementation of the remote object

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {

   public static void main(String[] args) throws Exception {
      Registry r = LocateRegistry.getRegistry("localhost", 2000);
      IFoo foo = (IFoo) r.lookup("remoteFoo");
      System.out.println(foo.getMessage());
   }
}
```

Look up the remote object from the registry by its name and get its reference

Invoke the method call

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Main {
   public static void main(String[] args) {
      try {
         Registry registry = LocateRegistry.createRegistry(2000);
         IFoo foo = new Foo();
         registry.bind("remoteFoo", foo);
         System.out.println("RMI registry started.");
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Create a registry

Bind the remote object to the registry with a name

Name	Object reference
“remoteFoo”	foo
…	…

PRACTICE

- Step-by-step tutorial at
  - http://www.javacamp.org/moreclasses/rmi/rmi.html
  - http://www.javacamp.org/moreclasses/rmi/rmi6.html
  - http://www.javacamp.org/moreclasses/rmi/rmi22.html
- Tips
  - Server.java:16, if your Server has a connection error, try to change the function call to

```
Registry registry = LocateRegistry.getRegistry("127.0.0.1");
```

  - Running rmiregistry: You must run rmiregistry under classpath. (i.e. where your .class files are located)
  - RMI registry: In the tutorial code, we use bind() every time server starts. So there may be conflicts when killing the server and restart. Restarting rmiregistry or change it to rebind will solve the problem.
- More on RPC and RMI:
  - https://people.cs.rutgers.edu/~pxk/417/notes/rpc.html
  - https://people.cs.rutgers.edu/~pxk/417/notes/pdf/02c-rpc-studies-slides.pdf