15-440 Distributed Systems Recitation 3

Slides By: Hend Gedawy &

Tamim Jabban

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Announcements

Grades for Pop Quiz 1 are out

Average: 8, Highest: 10

Grades for Problem Set 1 – Sunday

Problem Set 2 is Out

Due: Sep. 26th

Big Picture

PROJECT 1

Recitation 3:
Project 1

Problem Set 1:
Java Concepts,
Thread, Socket
Programming

Recitation 2:
Java Threads
and Socket
Programming

Recitation 1:
Java Concepts

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Outline

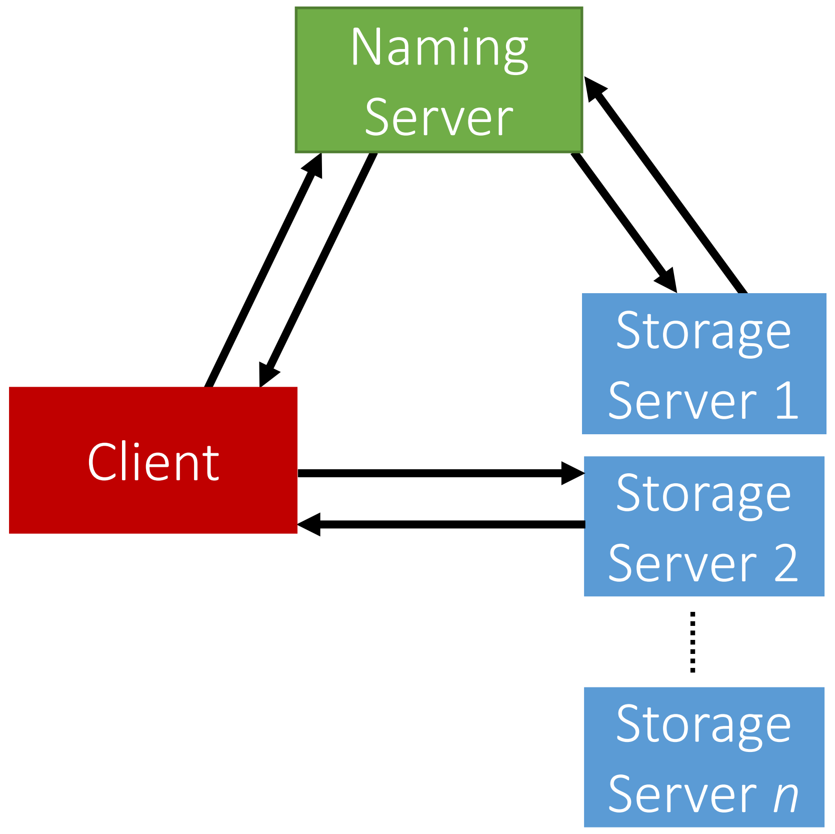
- **Project Overview**
- Architecture & Process Flow
- RMI Concepts & Example
- RMI in the Project
- Code Overview

Project 1

- Involves creating a *Distributed File System* (**DFS**): ***FileStack***
- Stores data that does not fit on a single machine
- Enables clients to perform operations on files stored on **remote servers**
  - Using **Remote Method Invocation (RMI)**

Entities

- Three main entities in FileStack:
  - Storage Servers:
    - Physically hosts the files in its local file system
  - Client:
    - Creates, reads, writes files *using RMI*
  - Naming Server (Mediator):
    - Runs at a predefined address
    - Uses a Directory Tree to maintain knowledge about the files in the system
      - Maps file names to Storage Servers
      - Repository of *metadata*



Implementation Notes

Provided ☺
You need to implement

Main Entities

Client entity is already implemented ☺

Naming Server

- naming package- NamingServer.java

Storage Server

- storage Package- StorageServer.java

Modules Common to all Entities

- **Communication (RMI)**
  - RMI package
    - Skeleton.java generic class
      - (used at the service hosting entity)
    - Stub.java generic class
      - (used at the invoking entity)
- **File/Directory Path Helper Methods** used by naming & storage server
  - common package - Path.java

Testing Code:

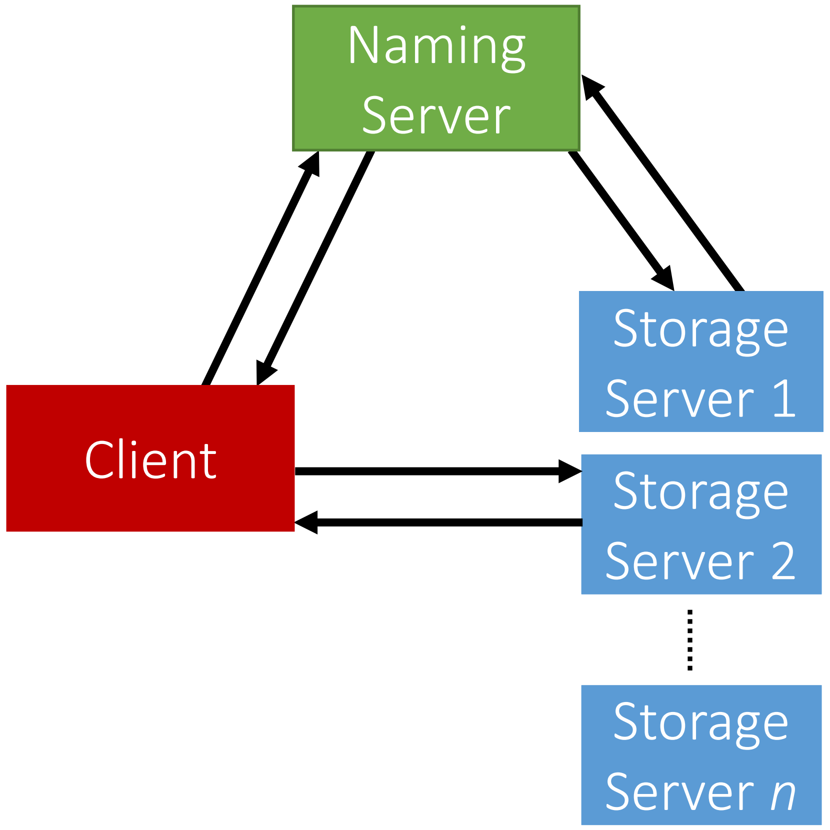
- Conformance package
- Main file: conformanceTests.java

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Outline

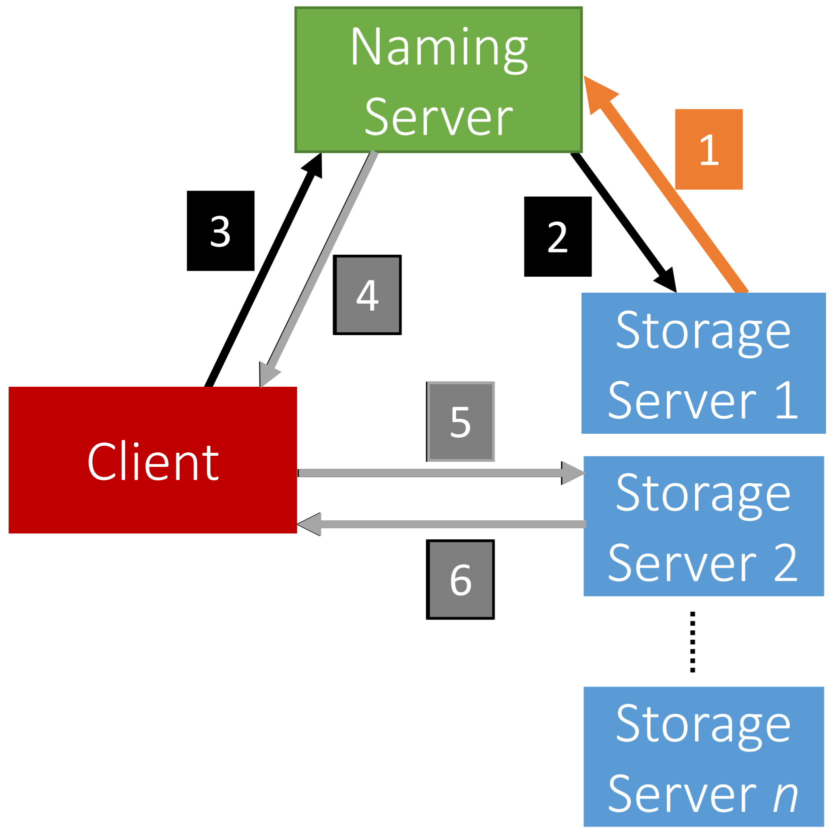
- Project Overview
- **Architecture & Process Flow**
- RMI Concepts & Example
- RMI in the Project
- Code Overview

Architecture



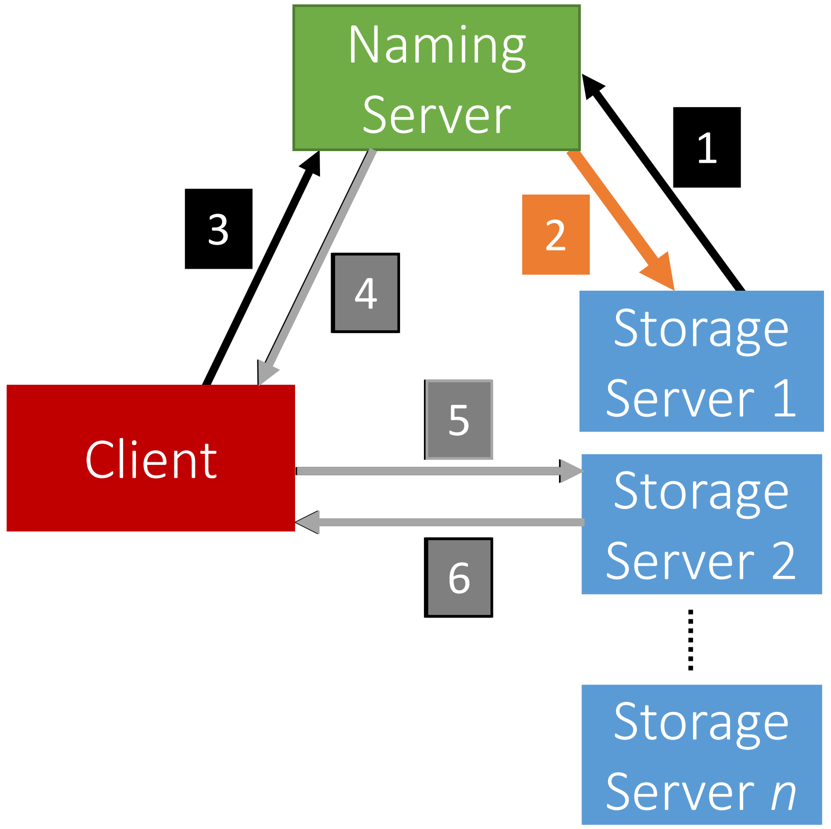
Process Flow

- Registration phase: storage sends its list of file paths that it hosts



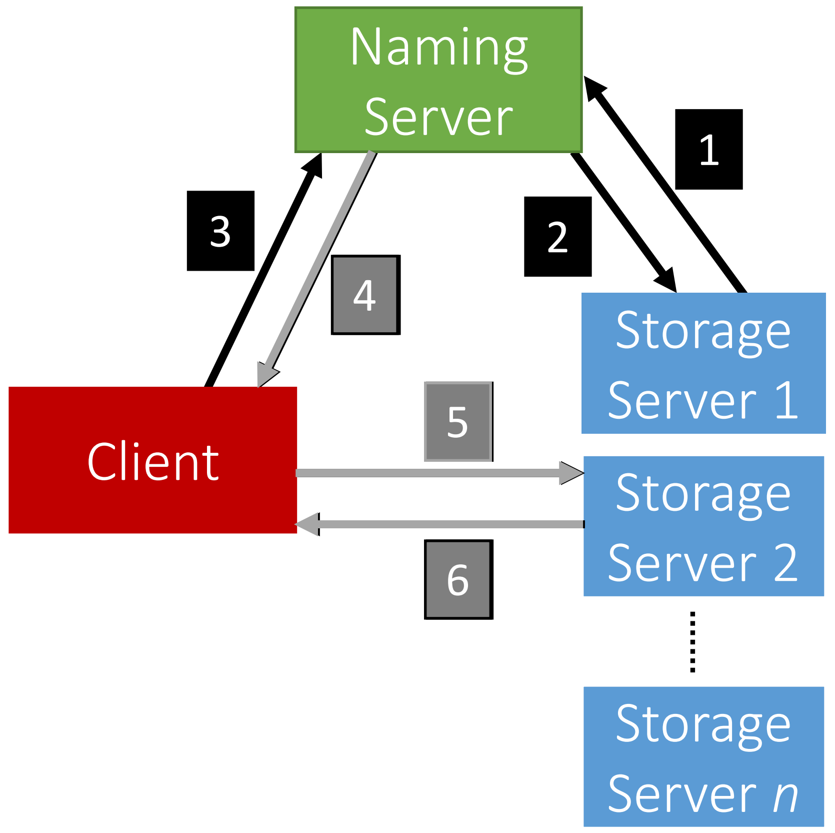
Process Flow

- Post registration, the Naming Server responds with a list of *duplicates* (if any).



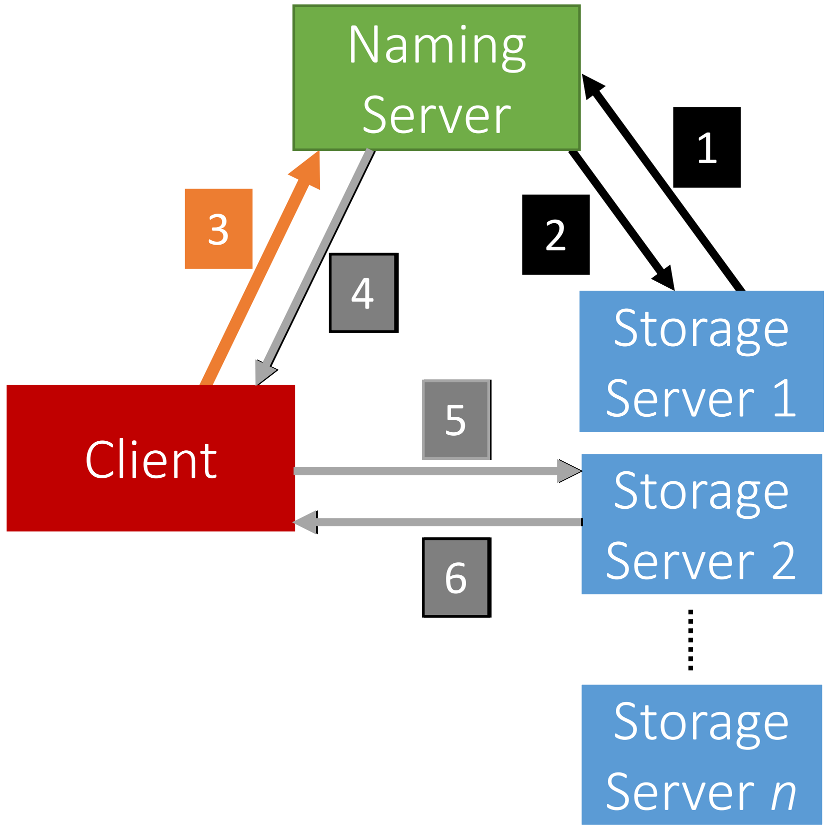
Process Flow

- System is now ready, the Client can invoke requests.



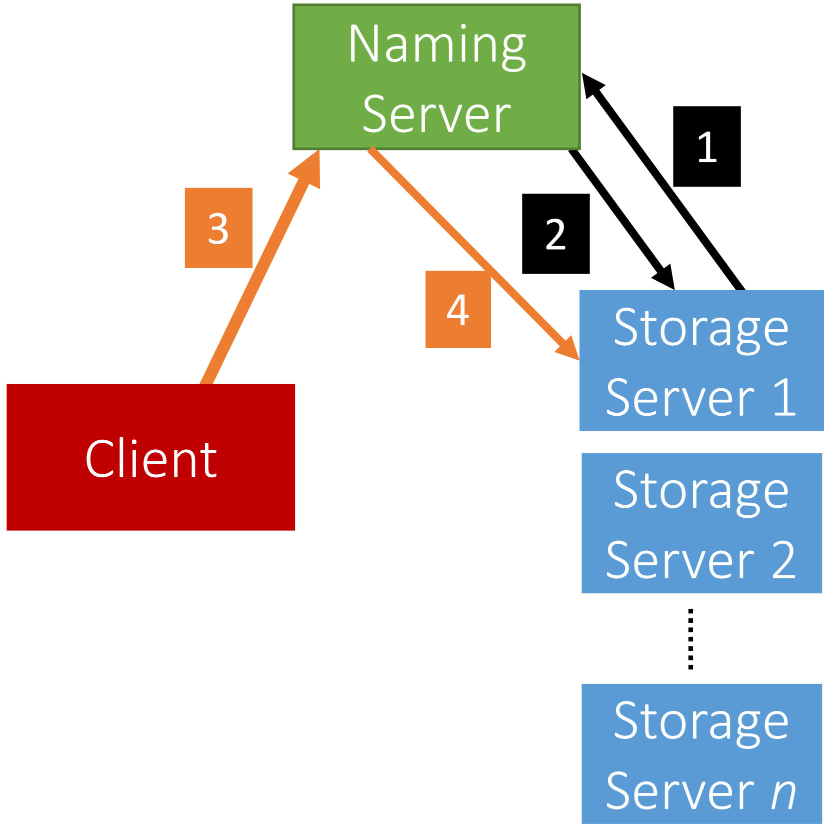
Process Flow

- Client requests a file <u>operation</u> from the Naming Server.



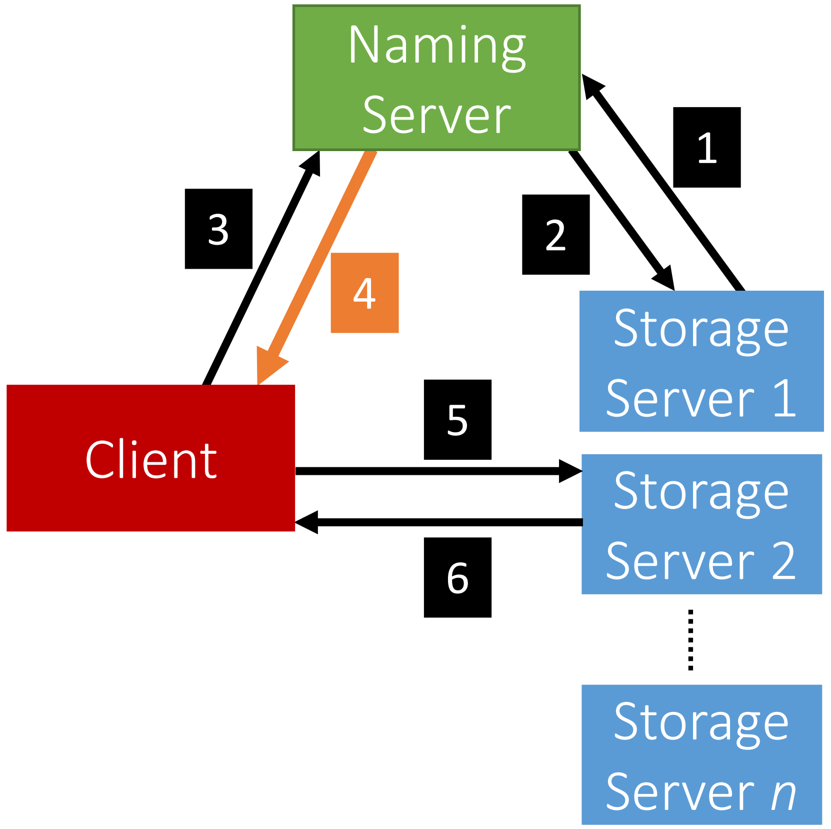
Process Flow

- If the client requests to create/delete a file or create/delete a directory, then the Naming Server takes care of handling the request with the Storage Servers



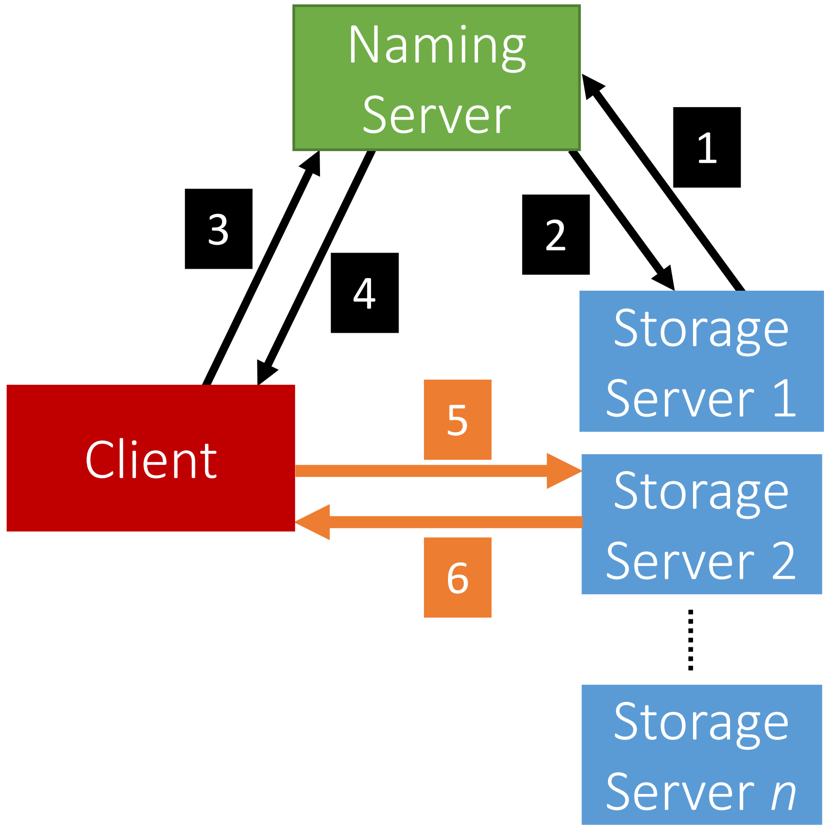
Process Flow

- **Otherwise,** the Naming Server responds back to the Client with the Storage Server that hosts the file.



Process Flow

- After the Client receives which Storage Server hosts the file, it contacts that Server to perform the file operation.



Outline

- Project Overview
- Architecture & Process Flow
- **RMI Concepts & Example**
- RMI in the Project
- Code Overview

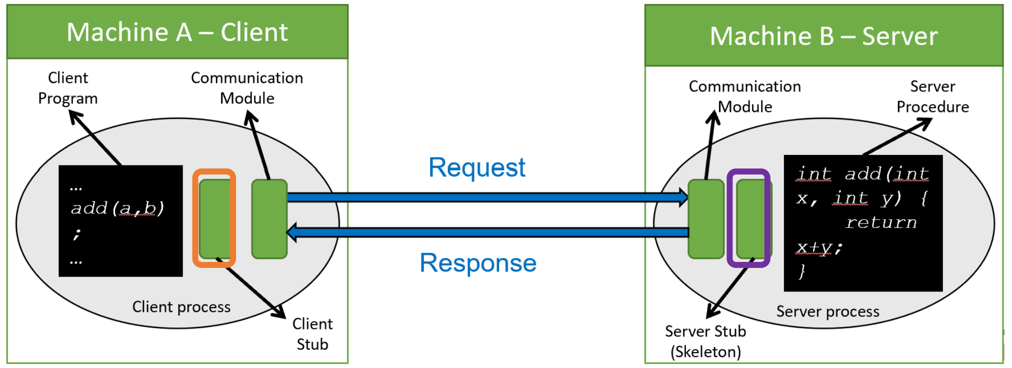
RMI

- When a Client invokes a method that is not local (**remote**), it does a (***R**emote **M**ethod **I**nvocation*)
  - This is because the *logic of the method resides on a remote server*

- To perform this remote invocation, we need **a library**: **Java RMI**

- **RMI allows the following:**
  - When the **client** invokes a request, it is **not a aware of where it resides** (local or remote). It only knows the **method's** name.
  - When a **server** executes a method, it is **oblivious to the fact that the method was initiated by a remote client**.

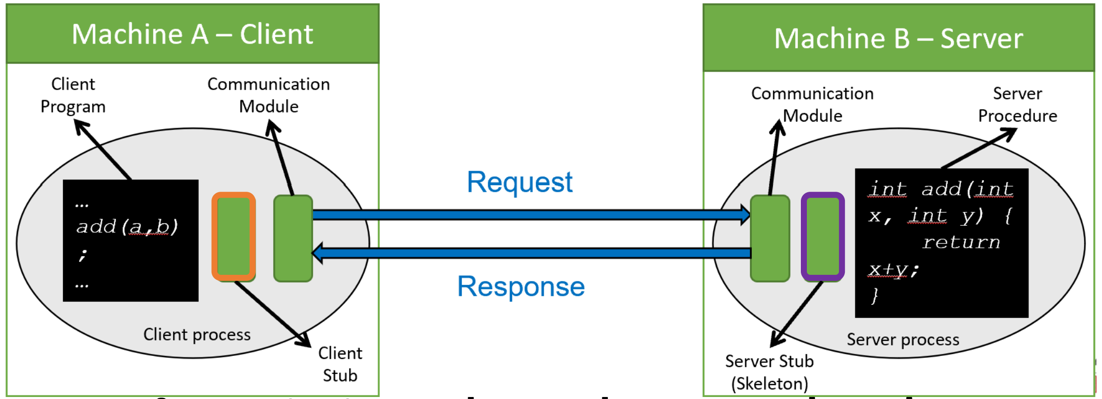
The **RMI library** is based on two important objects: **Stub** & **Skeleton**



```
…
add(a,b)
;
…
```

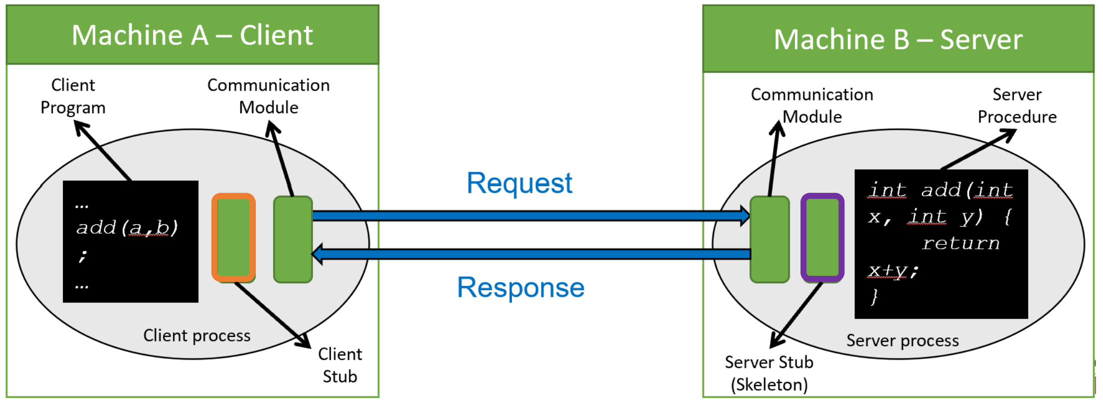
```
int add(int
x, int y) {
   return
x+y;
}
```

RMI Objects - Stub



- **Stubs**:
  - When a client needs to **perform an operation**, it invokes the method via an object called the "**stub**"
    - If the operation is **local**, the stub just calls the *helper function that implements this operation's logic*
    - If the operation is **remote**, the stub does the following:
      - **Sends (*marshals*) the method name and arguments** to the appropriate server (*or skeleton*),
      - **Receives the results (and *unmarshals*),**
      - **Reports them back to the client.**

RMI Objects- Skeleton



- **Skeletons:**
  - These are **counterparts** of stubs and reside reversely at the **servers**
    - Therefore, each **stub** communicates with a corresponding **skeleton**
    - **It's responsible for:**
      - **Listening** to multiple clients
      - **Unmarshalling** requests (**method name** & **method arguments**)
      - **Processing** the requests
      - **Marshalling & sending results** to the corresponding stub

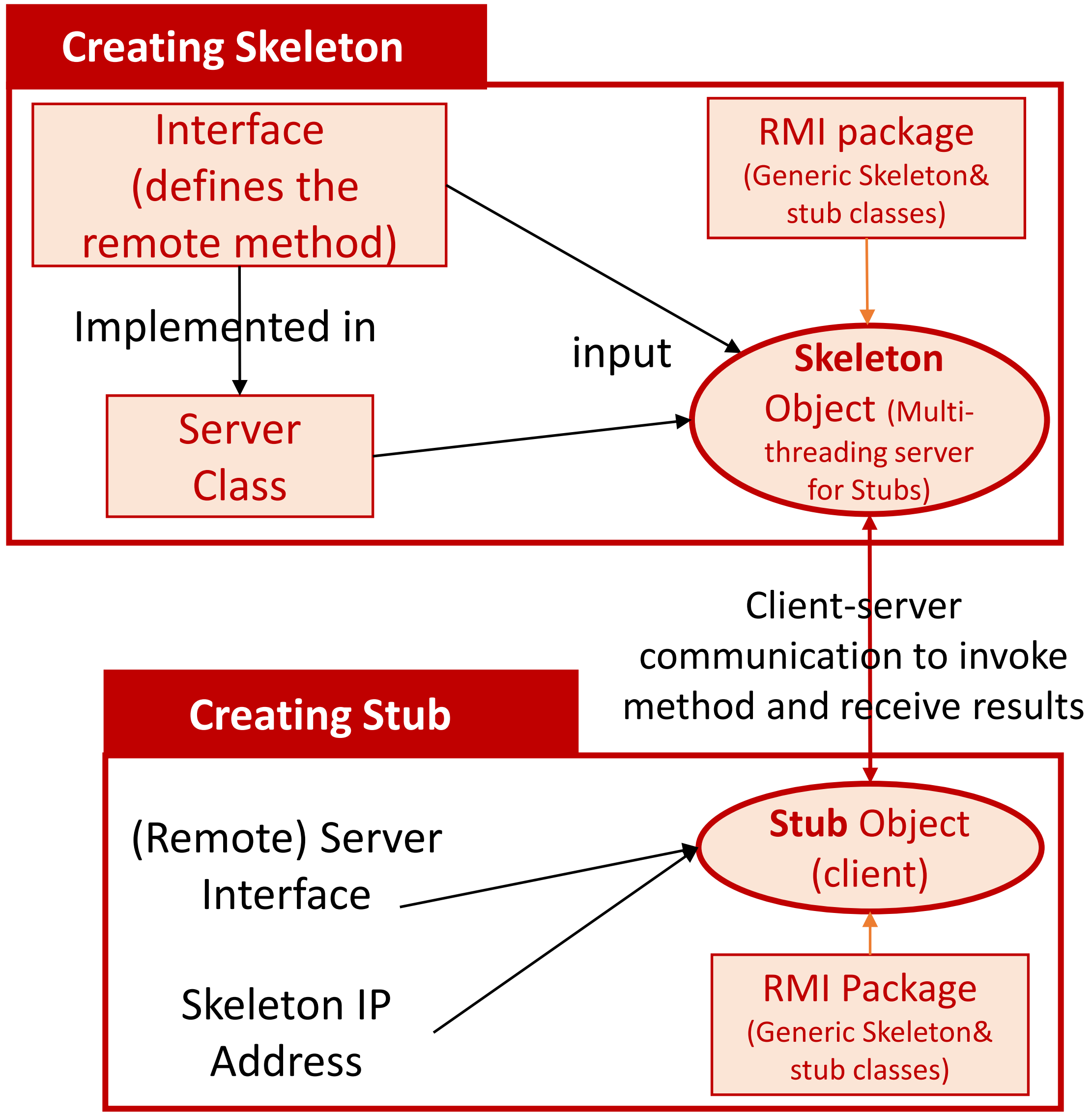
RMI – Implementation Logic

1. Creating **remote interface** that the server implements

2. Defining a **server class**

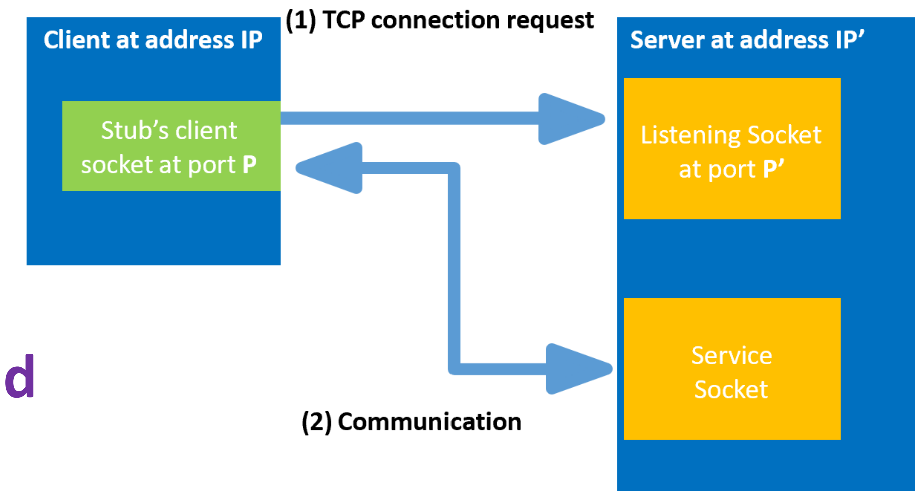
3. Making **it remotely accessible** (using a Skeleton)

4. **Accessing** a server object remotely (Using a Stub)



Creating Skeleton
Interface (defines the remote method)
RMI package (Generic Skeleton& stub classes)
Implemented in
input
Skeleton Object (Multi-threading server for Stubs)
Server Class
Client-server communication to invoke method and receive results
Creating Stub
(Remote) Server Interface
Stub Object (client)
Skeleton IP Address
RMI Package (Generic Skeleton& stub classes)

RMI – Skeleton Class


Client at address IP
Stub's client socket at port P
(1) TCP connection request
Server at address IP'
Listening Socket at port P'
Service Socket
(2) Communication

We implement multi-threaded socket programming

- The **skeleton** is **multi-threaded**
- When it is started, the main thread creates a listening socket and waits for client requests.

```
public void start() {
    create serverSocket();
    bind(address);
    while (!stopped) {
        clientSocket = accept();
     Thread a = new Thread
         (new serviceThread(clientSocket));
     a.start() ;
     }
}
```

- Once a client's request is received, the skeleton accepts the request, creates a new thread, and instantiates a new service socket to handle the communication

```
serviceThread {
    String methodName = (String) in.readObject();
    Class[] argTypes = (Class[]) in.readObject();
    Object[] args = (Object[]) in.readObject();
    Method m = c*.getMethod(methodName,argTypes);
    Object result = m.invokeMethod(implementation*, args);
    out.writeObject(result);
}
```

*c is the interface,
*implementation is the implementation of the interface

RMI Code Example – Server Side

1)

```
public interface IFile

 public String writeToFile (String filename, String txt) throws IOException;
```

2)

```
public class IFileServer implements IFile

 @Override
 public String writeToFile(String filename, String txt)
      { ....}

public static void main(String[] args) {

     // create InetSocketAddress given a port #

     // create and start skeleton
     Skeleton skltn = new Skeleton(address,
                          IFile.class, IFileServer.class);
     skltn.start(); }
```

3)

This is the method that needs to be **remote**!

```
public class Skeleton

   // server socket programming logic
   //create a serviceThread for each stub

   class ServiceThread implements Runnable

       // logic to read the request
       // invoke the method
       // return results
```

RMI – Stub Class

Goal: Stub *pretends* that it is *implementing* the corresponding skeleton's *interface locally at the client , while it is actually implemented at the remote server*

In java, a stub is implemented as a ***dynamic proxy,*** *using:*

1) Class loader: *for the interface*

- *Class loaders are responsible for* loading Java classes dynamically to the JVM (Java Virtual Machine) during runtime.

2) Interface: *the interface of the corresponding skeleton*

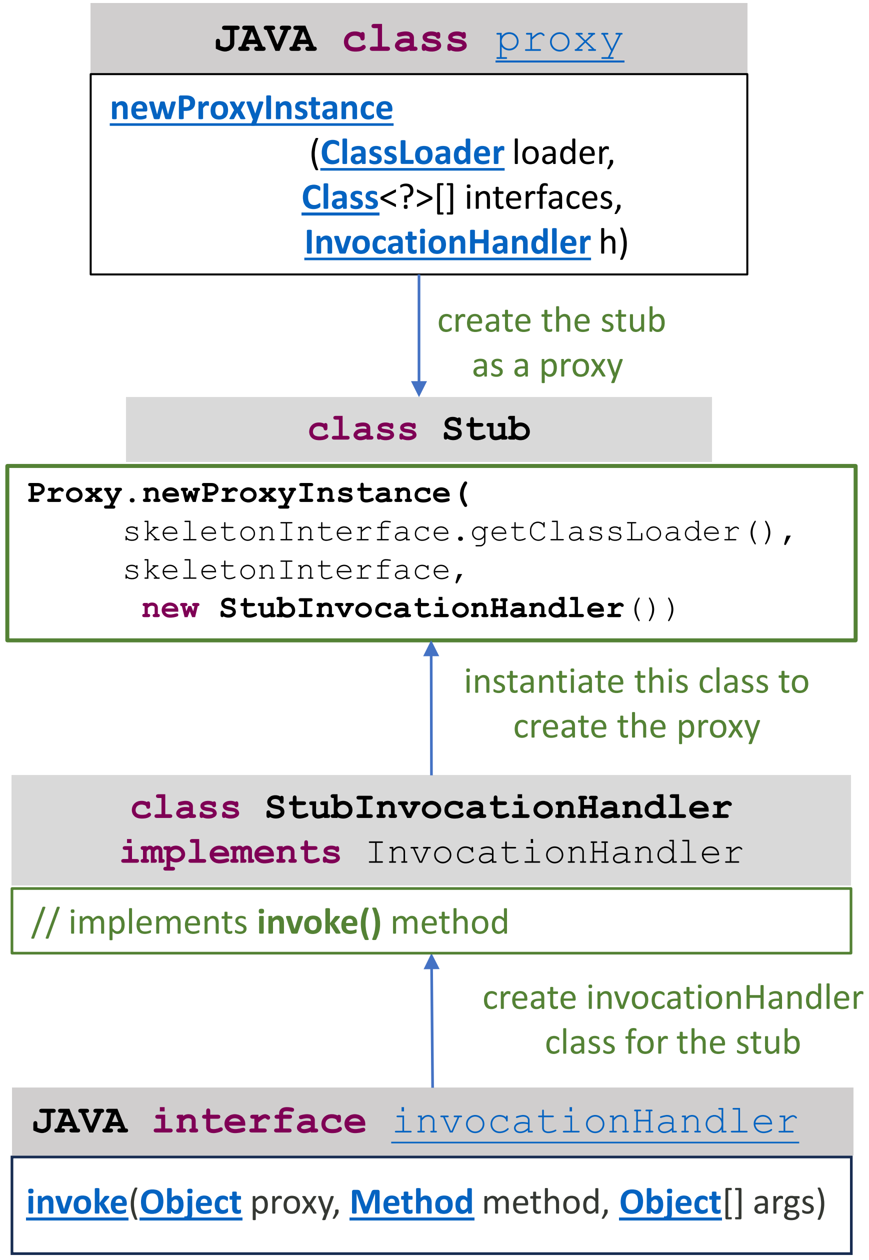
3) Invocation Handler: the *proxy* instance *dispatches method calls to* an associated *invocation handler object which implements the interface* InvocationHandler

- **Invoke():** logic to handle method invocation
  - Determines if method is local or remote
  - If remote (i.e. if it is one of the methods in the interface)
    - Connects to the corresponding skeleton
    - Marshals method name, argument types and values
    - Sends entailed byte stream
    - Waits for results
    - Unmarshals the result and send it back to client

Dynamic proxies allow one **single class** with one **single method** to service multiple method calls to arbitrary classes with an arbitrary number of methods.

http://tutorials.jenkov.com/java-reflection/dynamic-proxies. html

Go over `java.lang.reflect.Proxy` **via the JavaDocs!**



```
JAVA class proxy
newProxyInstance
        (ClassLoader loader,
         Class<?>[] interfaces,
         InvocationHandler h)
```

```
class Stub
Proxy.newProxyInstance(
      skeletonInterface.getClassLoader(),
      skeletonInterface,
       new StubInvocationHandler())
```

```
class StubInvocationHandler
implements InvocationHandler
// implements invoke() method
```

```
JAVA interface invocationHandler
invoke(Object proxy, Method method, Object[] args)
```

create the stub as a proxy
instantiate this class to create the proxy
create invocationHandler class for the stub

RMI Code Example – Client Side

```
public interface IFile

public String writeToFile (String filename,
        String txt) throws IOException;
```

This is the **remote** method that the client invokes

4)

```
public class Client

public static void main(String[] args) {

// create InetSocketAddress

// create stub and invoke method
Stub stub = new Stub(address, IFile.class);
IFile myStub = (IFile) stub.getStub();

myStub.writeToFile("File2.txt", "Whats up!");
```

```
public class Stub

public Stub(InetSocketAddress address, Class<IFile> intf)
{
      Object stub = Proxy.newProxyInstance(
            // The ClassLoader that is to "load" the
            dynamic proxy class.
            intf.getClassLoader(),
            // An array of interfaces to implement.
            new Class[] {intf},
            // An InvocationHandler to forward all methods
            calls on the proxy to
            new StubInvocationHandler());          }

public Object getStub() {
      return this.Stub; }

  class StubInvocationHandler implements
  InvocationHandler

  @Override
  public Object invoke(Object stub, Method
  method, Object[] args){

        // connect to corresponding skeleton
         // encode & send the request
        //receive and decode results    }
```

RMI Code Example

Let's Try It Out ☺

Outline

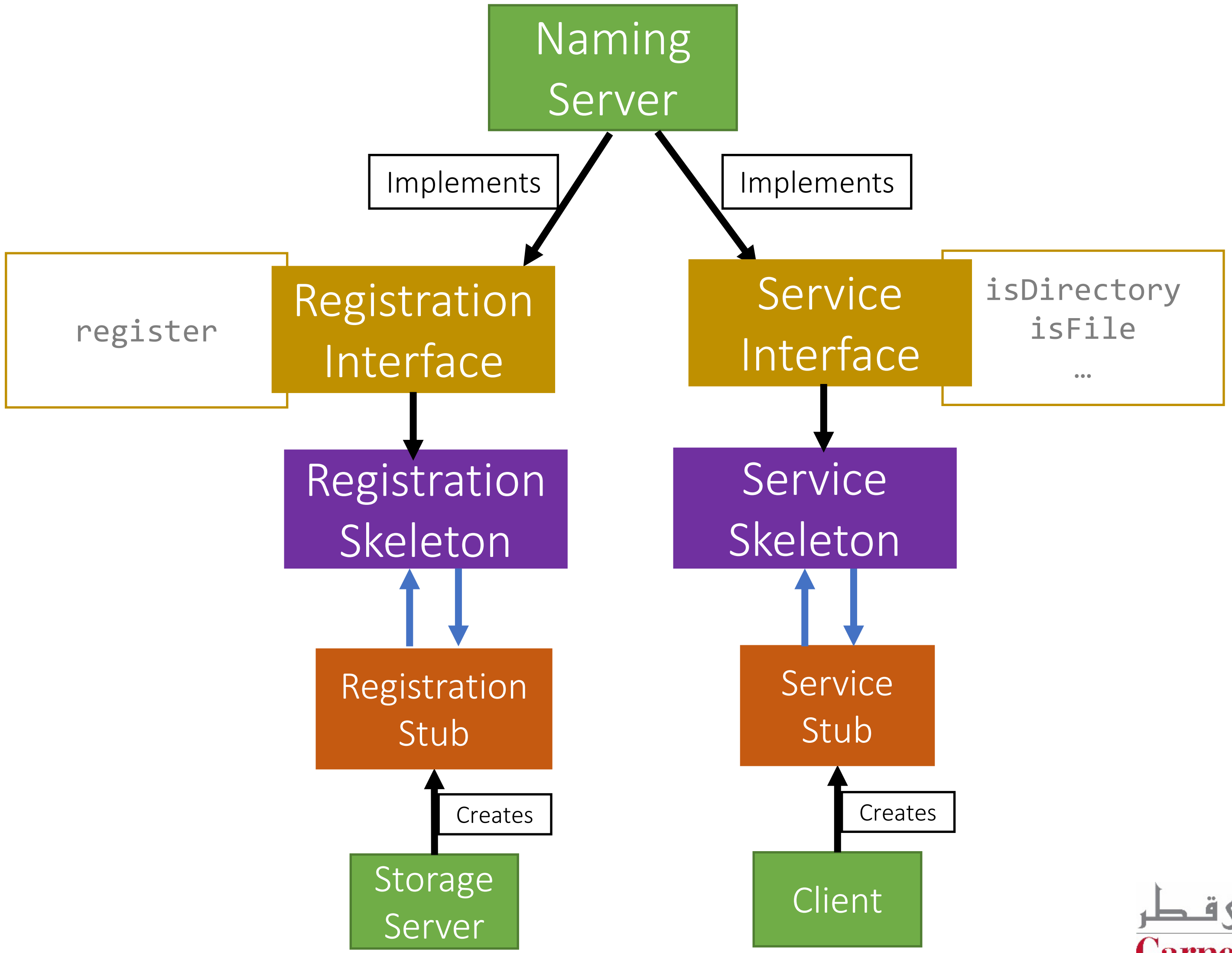
- Project Overview
- Architecture & Process Flow
- RMI Concepts & Example
- **RMI in the Project**
- Code Overview

RMI in Project 1

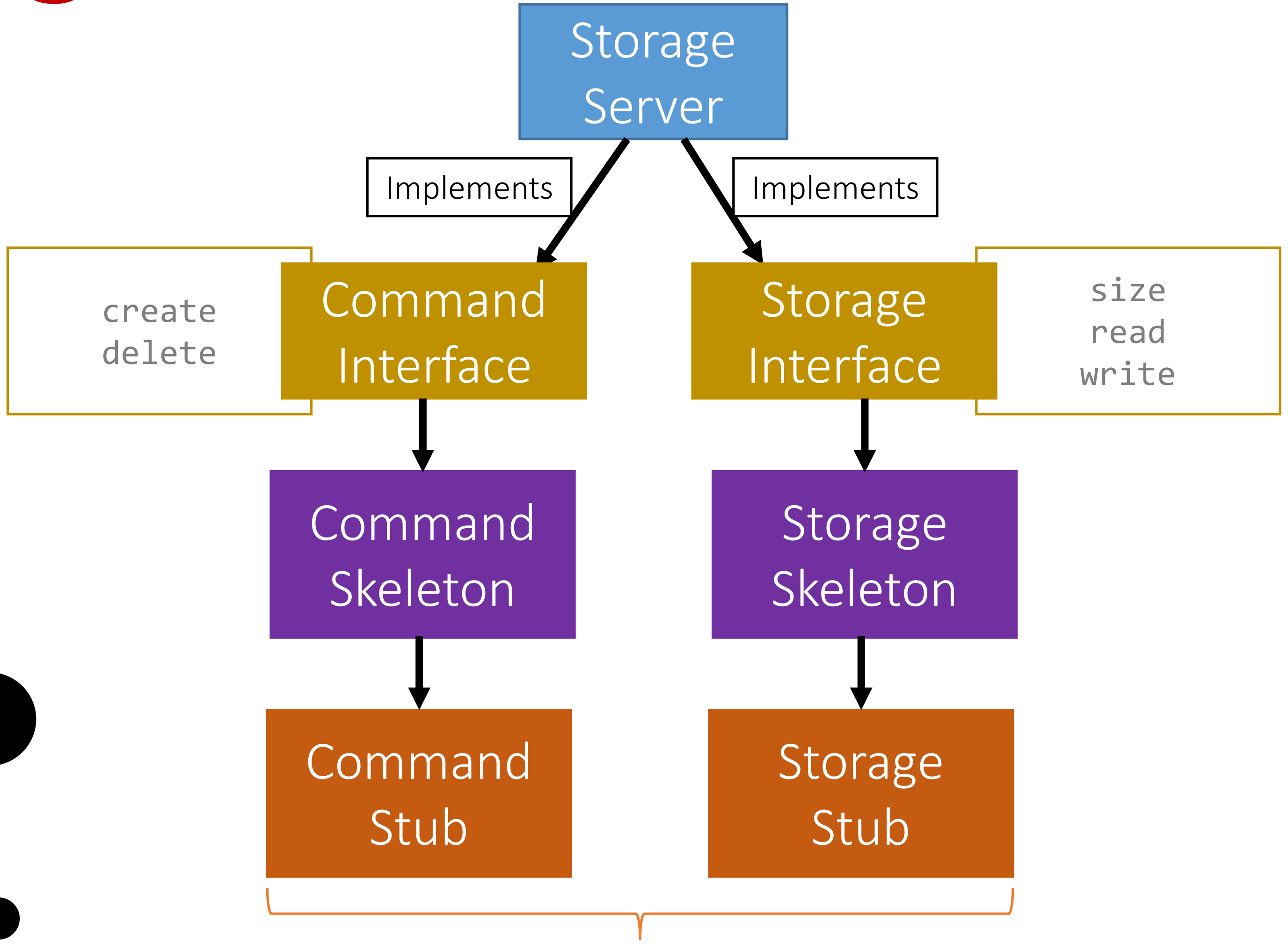
You will Implement the Skeleton & Stub classes (RMI Library)

Where are skeletons and stubs used in the Project?

Naming Server Interfaces& Skeletons



Storage Server Interfaces, Skeletons



These stubs are created at the storage server and sent to the Naming server during registration

Who invokes methods in each interface?

Why ???

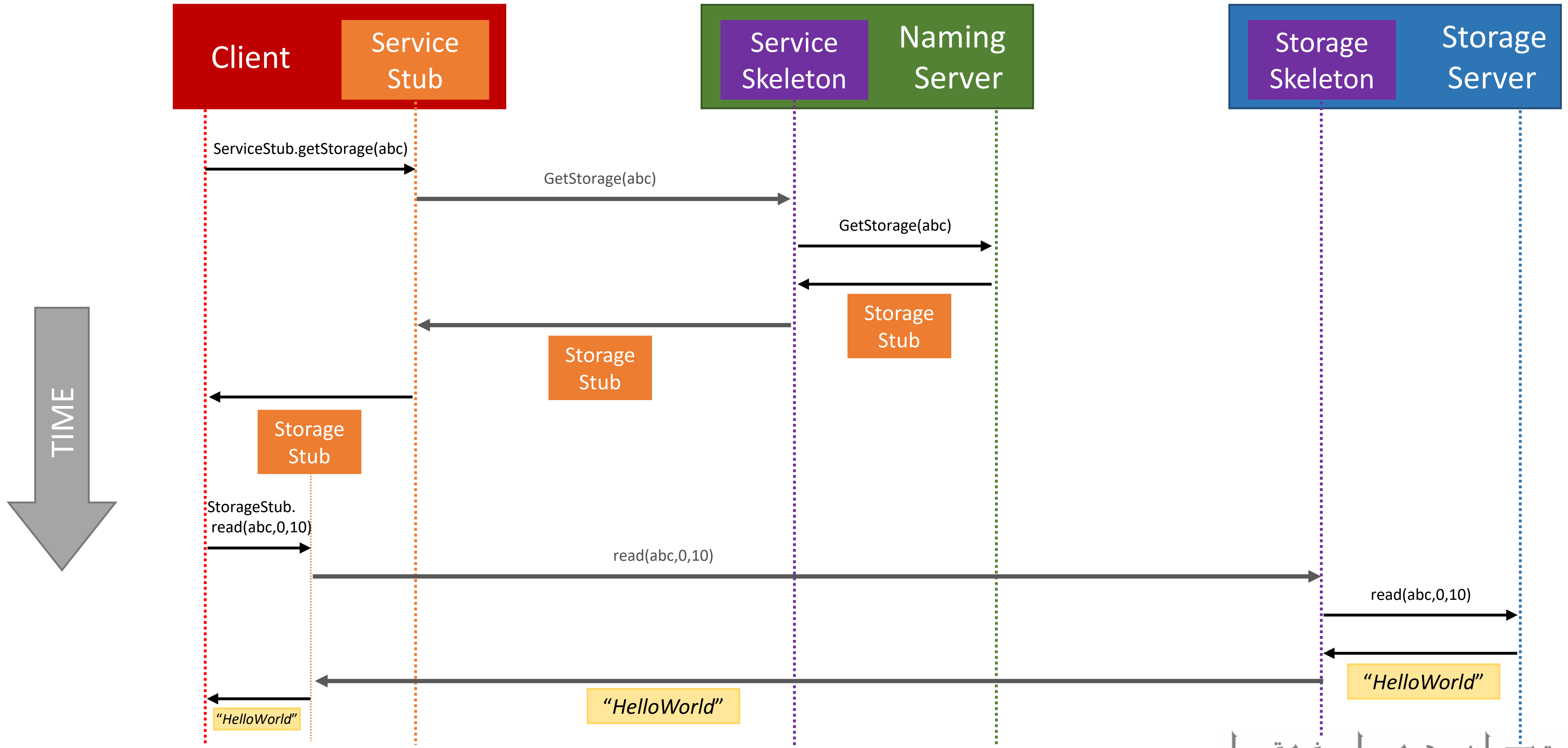
RMI in Project 1: Skeletons & Stubs Summary

- **At Client**
  - Service Stub: connects to the Naming server - sends file operation requests

- **At Naming Server**
  - **Implements Service & Registration Interfaces**
  - Service Skeleton: serves Clients for file operations
  - Registration Skeleton: serves Storage server for registration

- **At Storage Server:**
  - **Implements Storage & Command Interfaces**
  - Registration Stub: registers with naming server
  - Storage Skeleton: serves Clients
    - Storage Stub (sent to Naming Server to send it to Clients to use to write/read file and get file size)
  - Command Skeleton: serves the Naming Server
    - Command Stub (sent to Naming Server to use it to act as client to the Storage Server when it needs it to create/delete files/directory)

RMI Full Example: Client Read



Outline

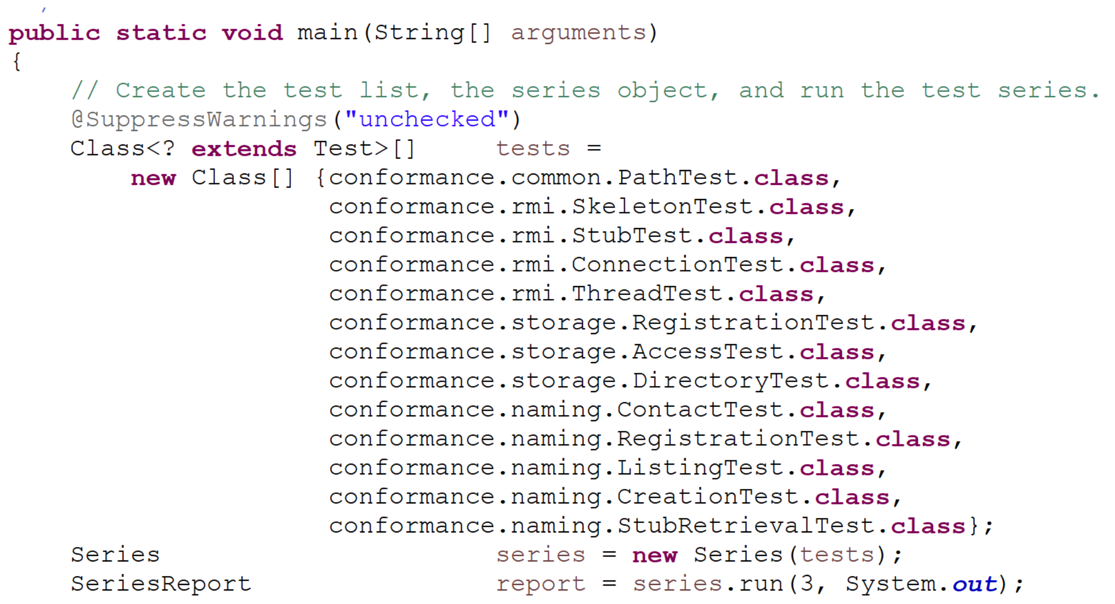
- Project Overview
- Architecture & Process Flow
- RMI Concepts & Example
- RMI in the Project
- **Code Overview**

Outline: Code Overview

- The main entities
  - Look at the files that need implementation
- The Conformance testing code
  - The main file where tests are called
  - How test classes are structures
  - Knowing dependencies among test classes
  - Example: Look at the testing code of
    - Path
    - RMI

Running Code Notes

- Edit …./Project1/conformance/ConformanceTests.java
  - Comment out the test lines that you don't want to run



```
public static void main(String[] arguments)
{
    // Create the test list, the series object, and run the test series.
    @SuppressWarnings("unchecked")
    Class<? extends Test>[]     tests =
        new Class[] {conformance.common.PathTest.class,
                     conformance.rmi.SkeletonTest.class,
                     conformance.rmi.StubTest.class,
                     conformance.rmi.ConnectionTest.class,
                     conformance.rmi.ThreadTest.class,
                     conformance.storage.RegistrationTest.class,
                     conformance.storage.AccessTest.class,
                     conformance.storage.DirectoryTest.class,
                     conformance.naming.ContactTest.class,
                     conformance.naming.RegistrationTest.class,
                     conformance.naming.ListingTest.class,
                     conformance.naming.CreationTest.class,
                     conformance.naming.StubRetrievalTest.class};
    Series                      series = new Series(tests);
    SeriesReport                report = series.run(3, System.out);
```

- **…./Project1$** make
- Run ConformanceTests file

Recap …

- **Project1 Overview**
  - Main Entities
    - Naming Server
    - Storage Servers
    - Client
  - Path and RMI Library
- **Project 1 Process Flow & Communication**
- **RMI Concepts & Example**
  - Skeletons & Stubs
  - Implementation w/ Example
- **RMI in the Project**
  - Skeletons and Stubs in Project 1
  - Example: Client Read
- **Starter/Testing Code Overview**