15-440

Distributed Systems

Recitation 4

By: Hend Gedawy
& Previous TAs

Announcements

Grades for Problem Set 1 are out

Design Report for Project 1

Due: Sep. 17th (Sunday)

PS1 misconception

2pts

d. What is wrong with the following code?

```
class MyException extends Exception { }

public class Q1d {

  public void foo() {
    try {
      bar();
    } finally {
      baz();
    } catch (MyException e) {}
  }

  public void bar() throws MyException {
    throw new MyException();
  }

  public void baz() throws RuntimeException {
    throw new RuntimeException();
  }
}
```

- Since the method foo() does not catch the exception generated by the method baz(), it must declare the RuntimeException in a throws clause
- A try block cannot be followed by both a catch and a finally block
- An empty catch block is not allowed
- A catch block cannot follow a finally block
- A finally block must always follow one or more catch blocks

When a method declared with **throws** (e.g. baz()) is called by another method (e.g. foo()), …

the thrown exception has to be handled at the caller in one of two ways to prevent compile time error:

1. By using try catch
2. By using the **throws** keyword

You will encounter this in Project 1

Last Time

- Entities, Architecture and Communication
- RMI Concepts
- RMI Demo
- RMI in Project 1
- Starter Code Overview

Today

- Packages dive-in:
  - ✓ RMI
  - ✓ Common
  - ✓ Naming
  - ✓ Storage

RECAP
$

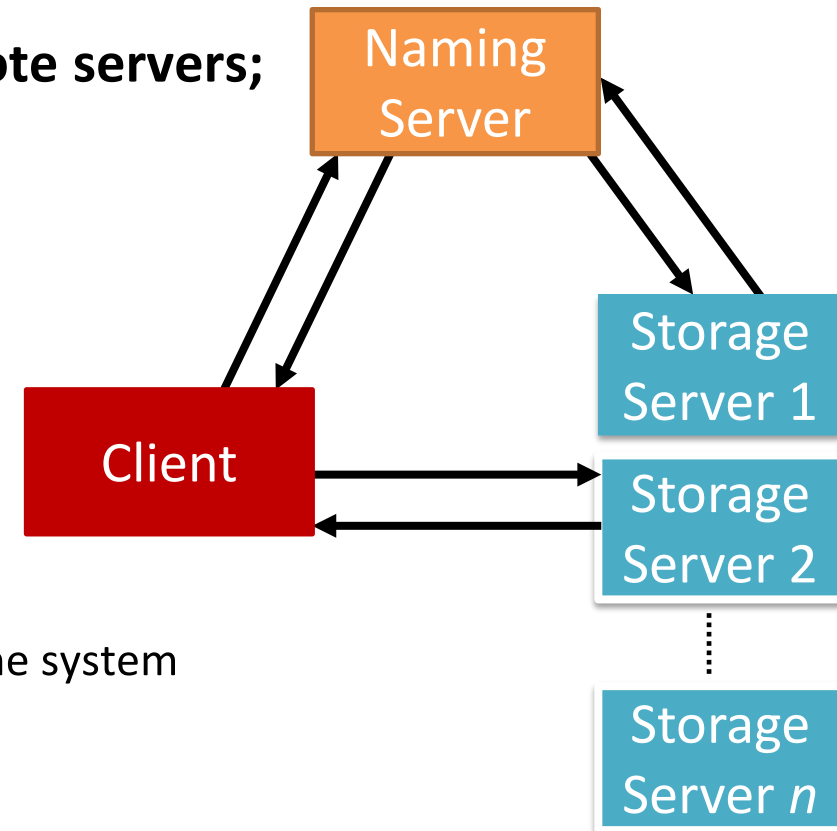
Project 1 Overview

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)**

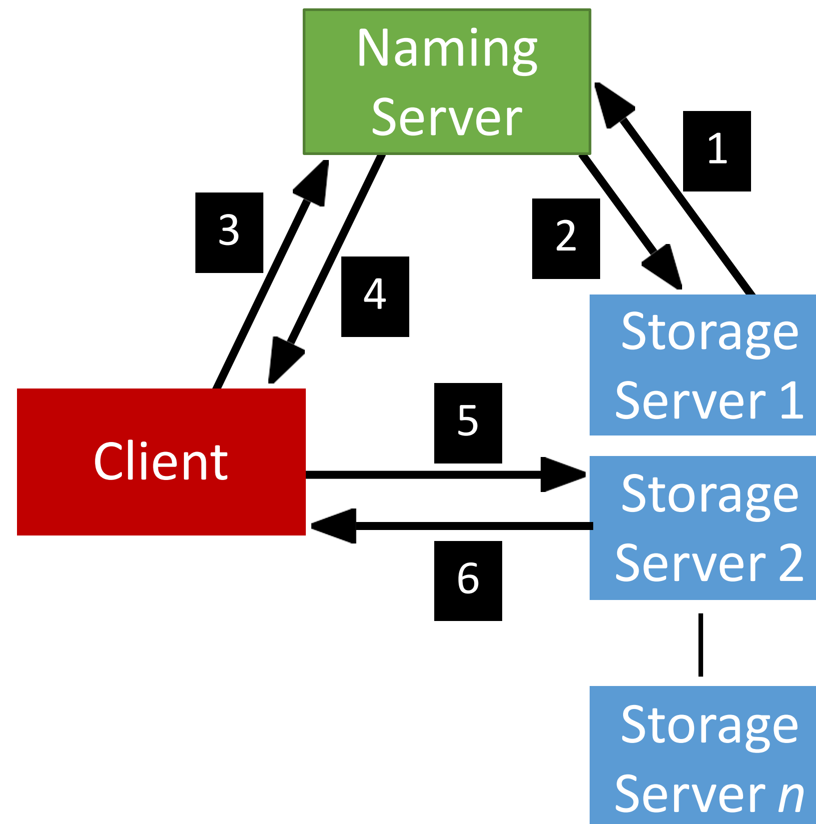
Three main entities in FileStack:

- 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*



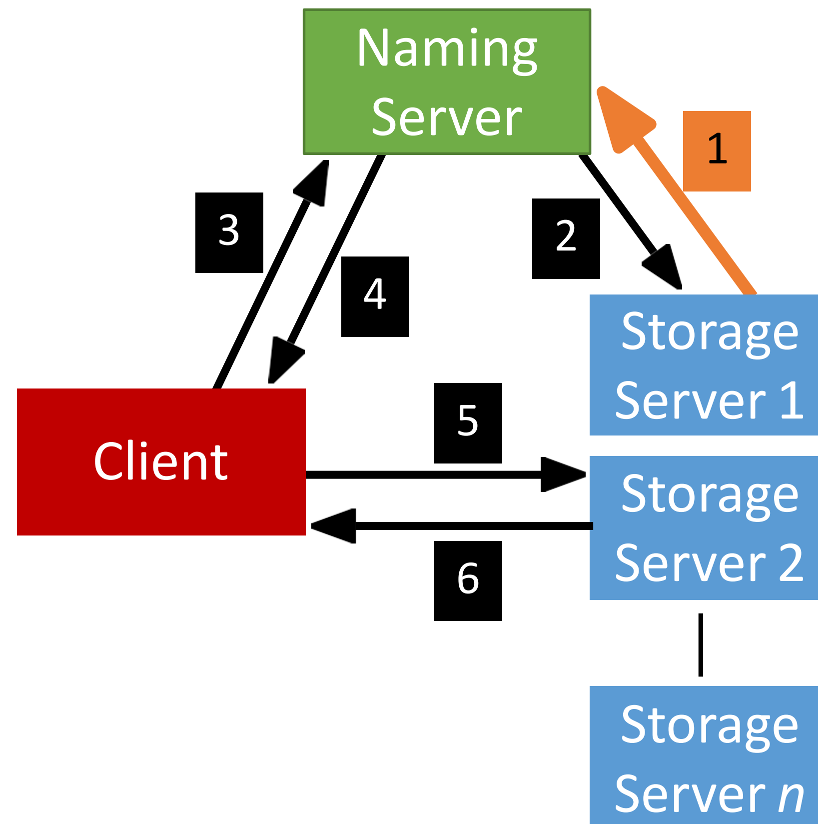
Architecture

- FileStack will boast a Client-Server architecture:



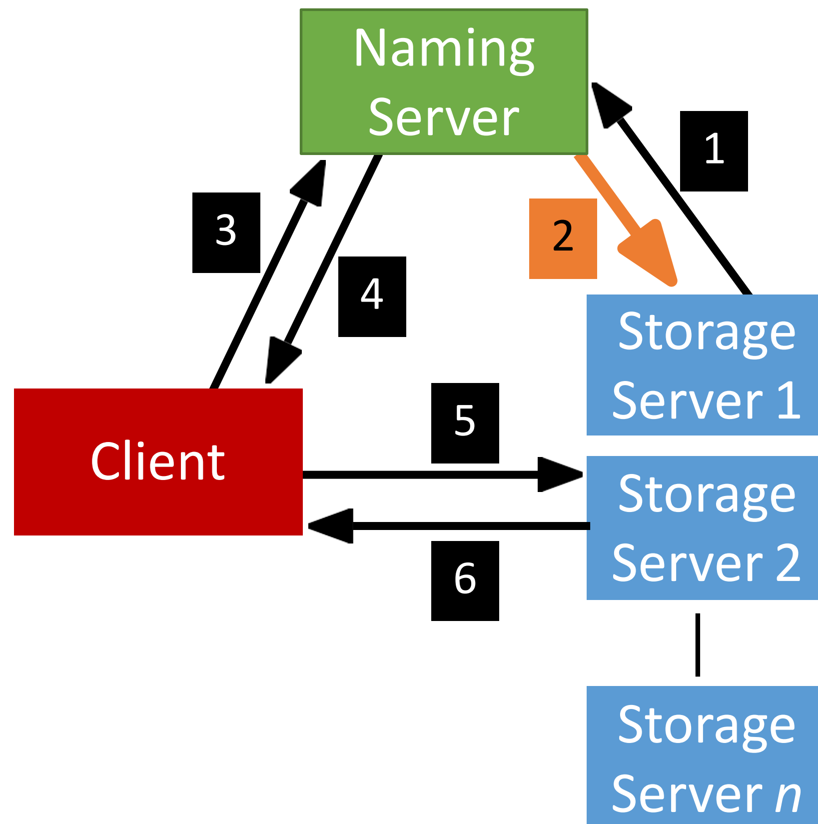
Communication

- Registration phase



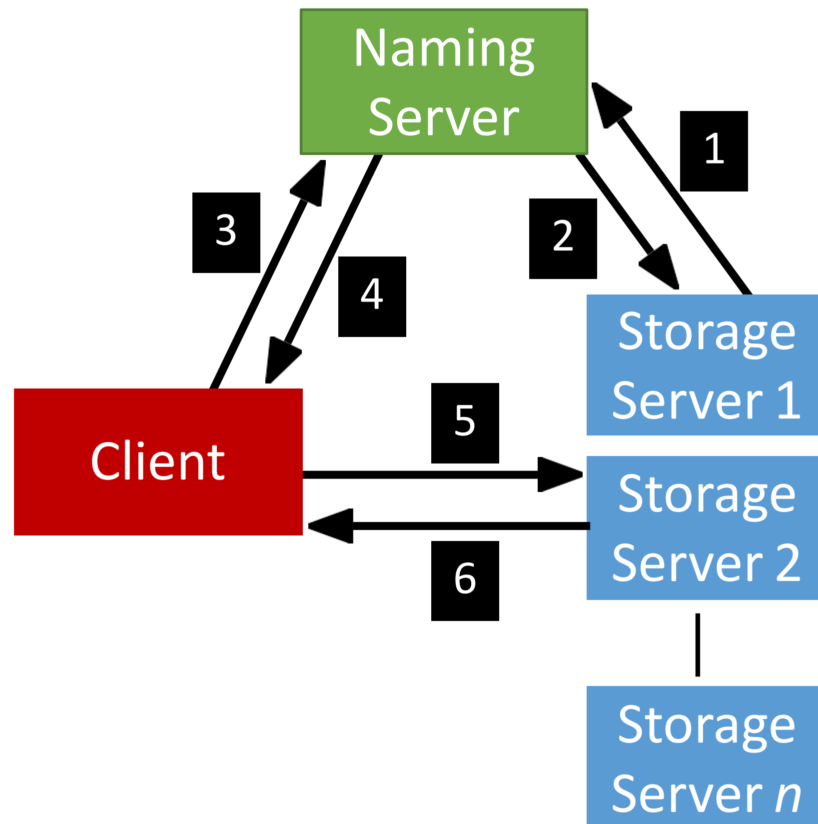
Communication

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



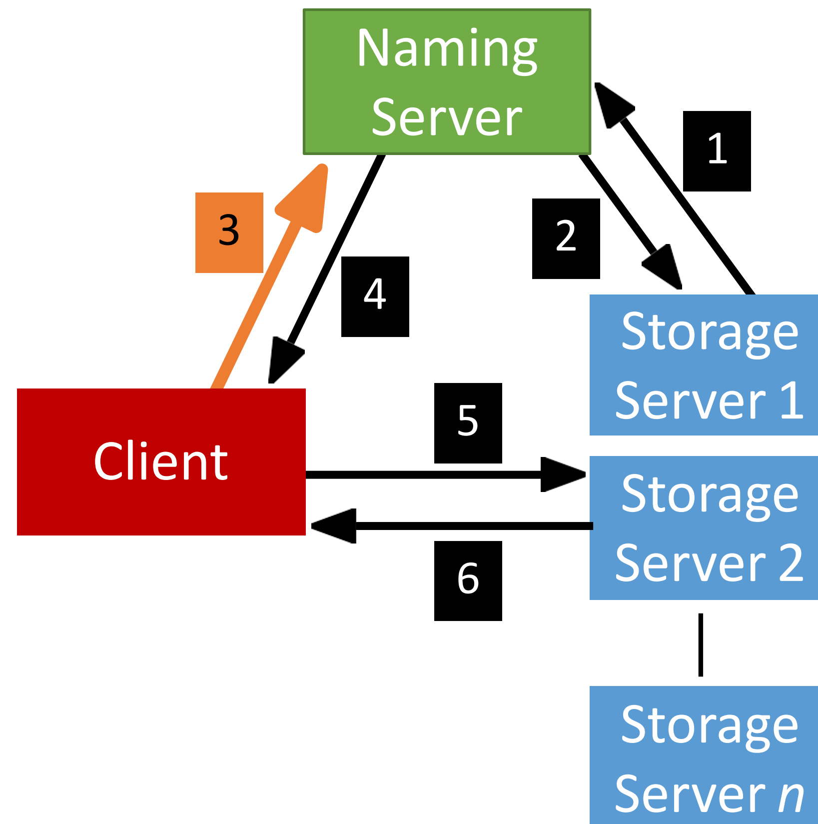
Communication

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



Communication

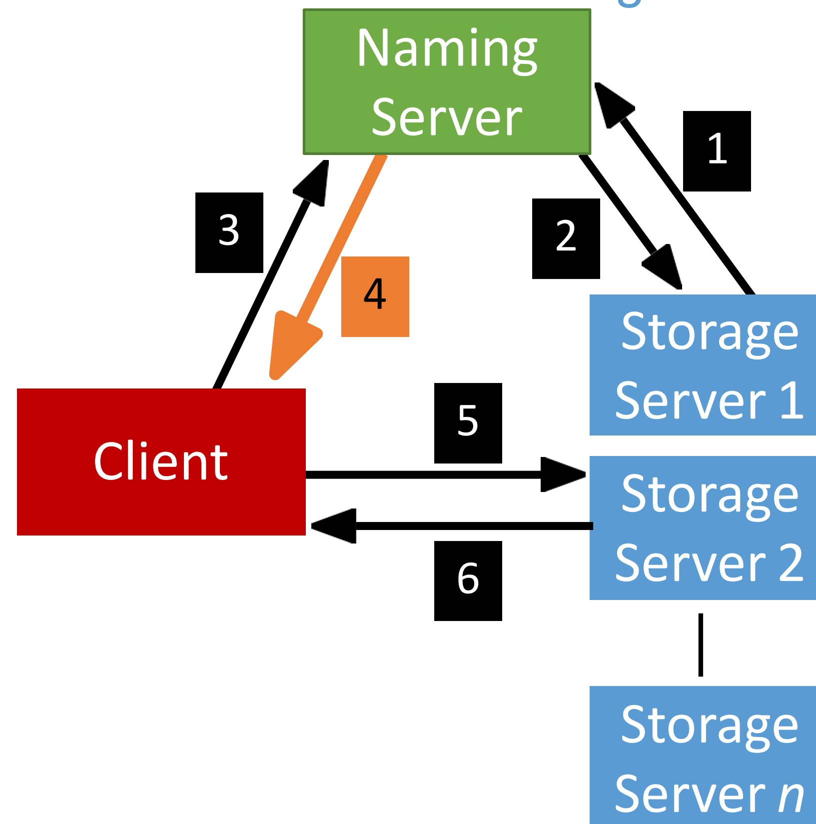
- Client can send file operation requests to the Naming Server.



جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

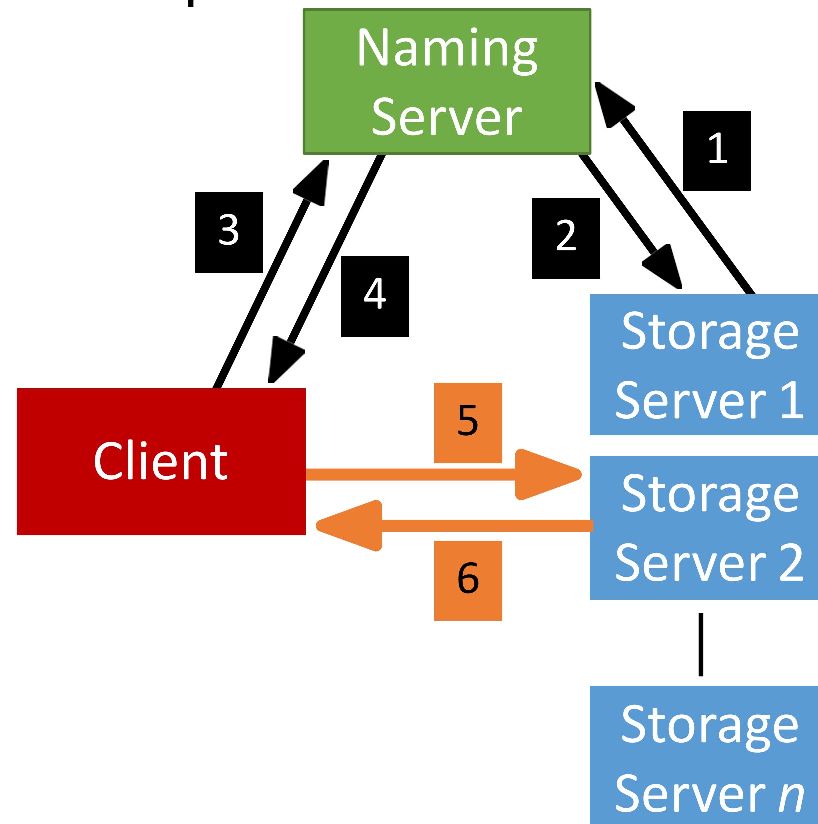
Communication

- Depending on the operation, the Naming Server could either perform it,
  - or, respond back to the Client with the Storage Server that hosts the file.



Communication

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



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

Today's Outline

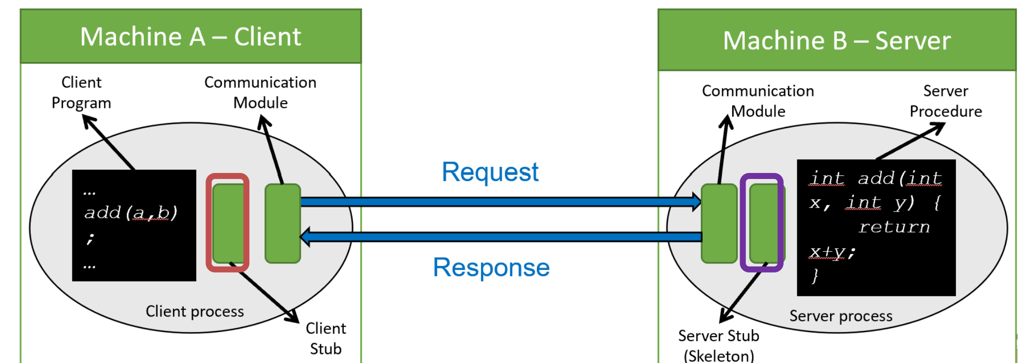
Packages dive-in:

- ✓ **RMI**
- ✓ Common
- ✓ Naming
- ✓ Storage

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**



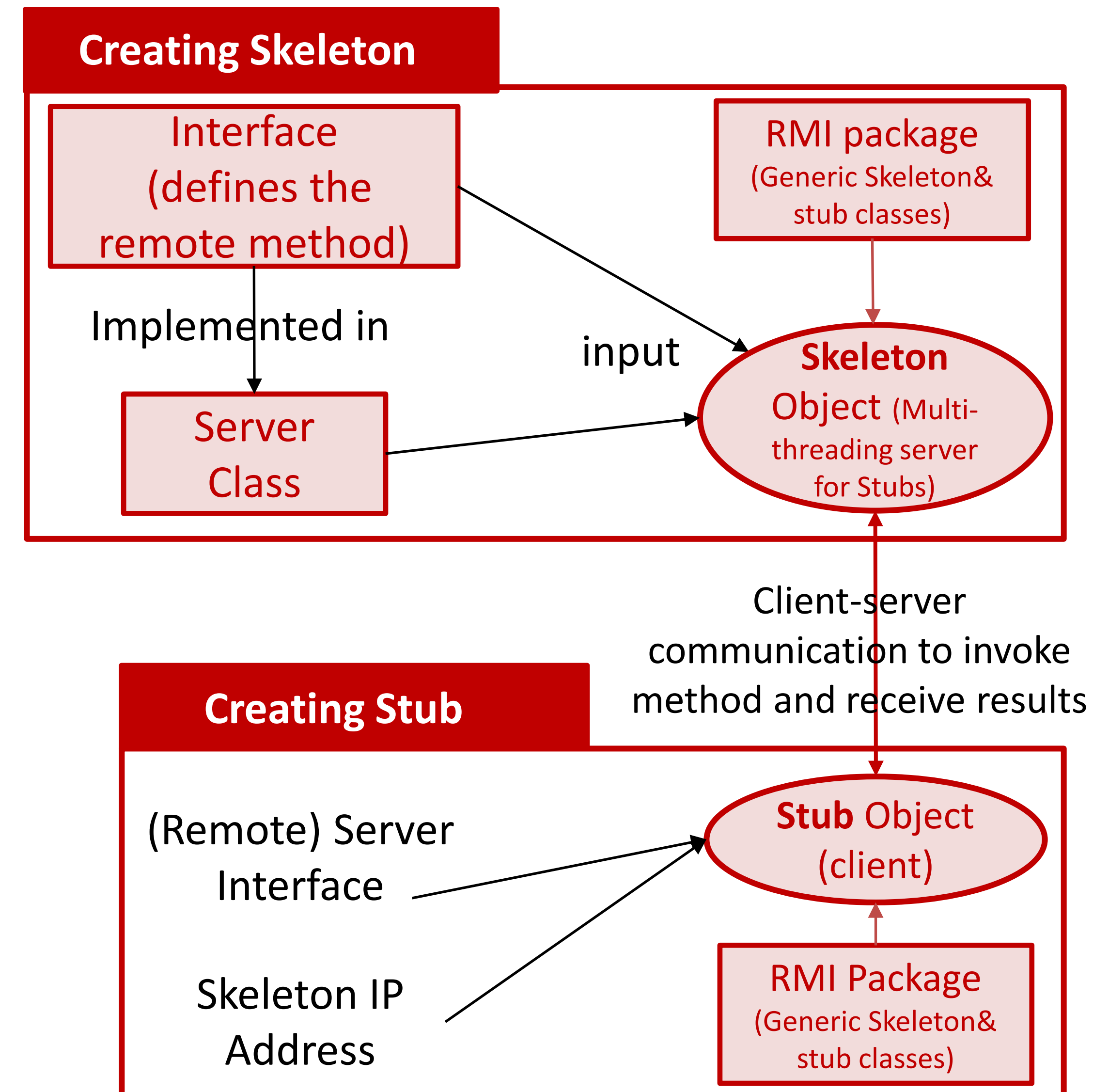
RMI – Implementation Logic

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

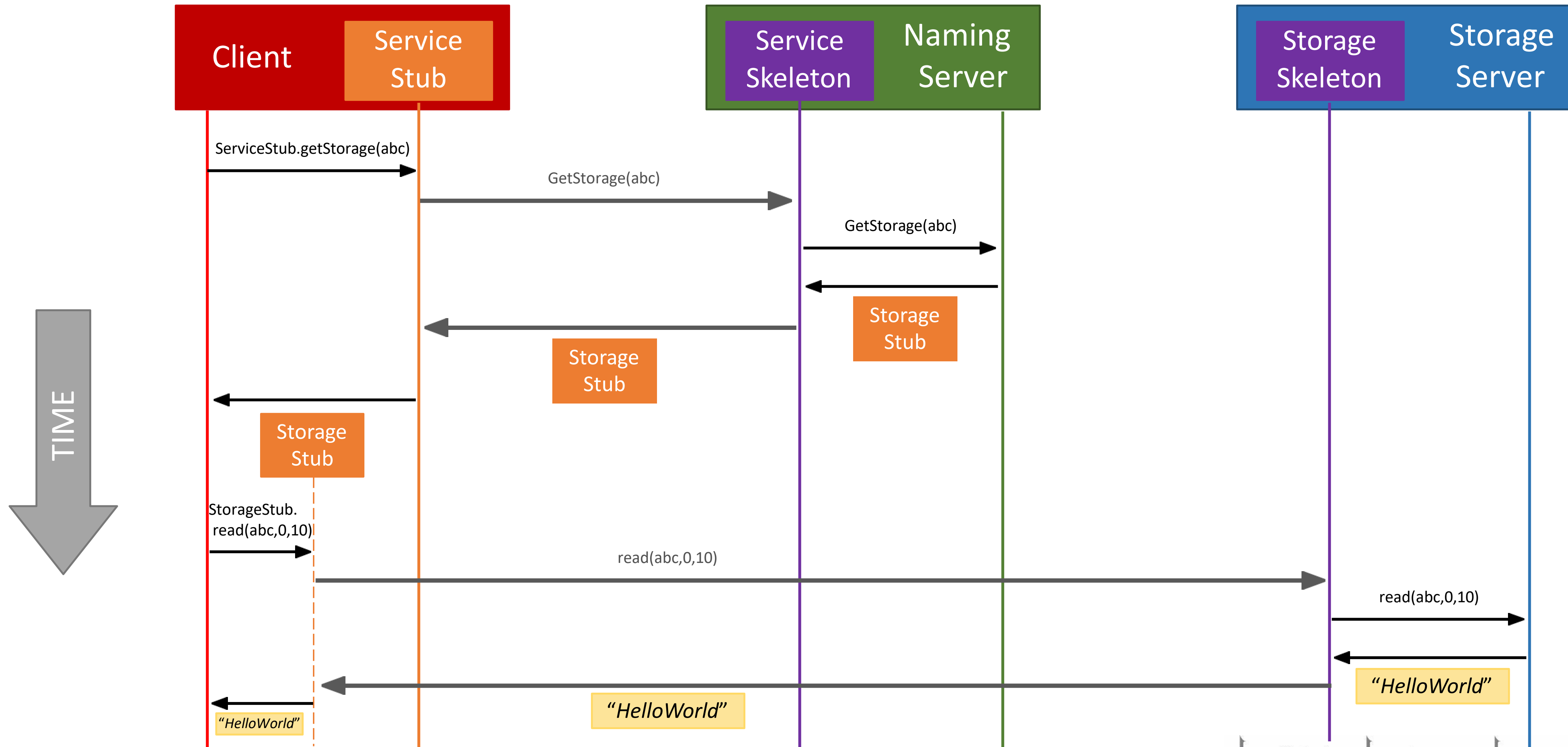
2. Defining a **server class**

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

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

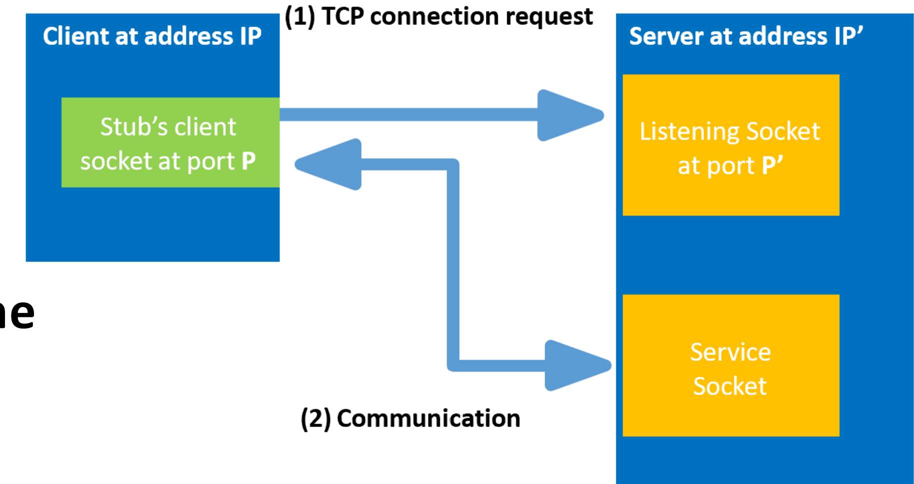


RMI in Project 1- Full Example: Client Read



Skeleton: Expected Performance

- Skeleton is a multi-threaded TCP server
- When it is started, the main thread **creates a listening socket** and **waits for client requests**.
- Once a client's request is received, the skeleton **accepts the request**, creates a new **service thread**, and instantiates a new service socket to handle the communication



- *The service thread lives till an exception is thrown OR the request is processed, and result is returned*
  - *The* ***result*** *returned to client could be …*
    - *a value: returned by the invoked method*
    - *Or an* InvocationTargetException cause
  - *For other socket I/O Exceptions, an* **RMIException** should be thrown

Stub: Expected Performance

- A stub is created as a **dynamic proxy** instance
- It is associated with an instance of a class that implements **InvocationHandler** Java interface (e.g. StubInvocationHandler)

The class implements **invoke method** to do the following:

- If **method is remote**:
  - Connect to skeleton
  - Marshall and send request
  - Unmarshall result
    - Value: return it to client
    - InvocationTargetException, throw
  - Throw RMIException for I/O Exceptions
- If **method is local**, call it. Local methods are:
  - equals
  - hashCode
  - toString
- If method is **neither**: throw NoSuchMethodError

```
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 Questions

When creating a Skeleton or a Stub, you are asked to throw an Error if the passed class c doesn’t represent a remote interface.

How to know an interface is remote??

An interface is remote if all of its methods throw an exception of type RMIException

Today's Outline

Packages dive-in:

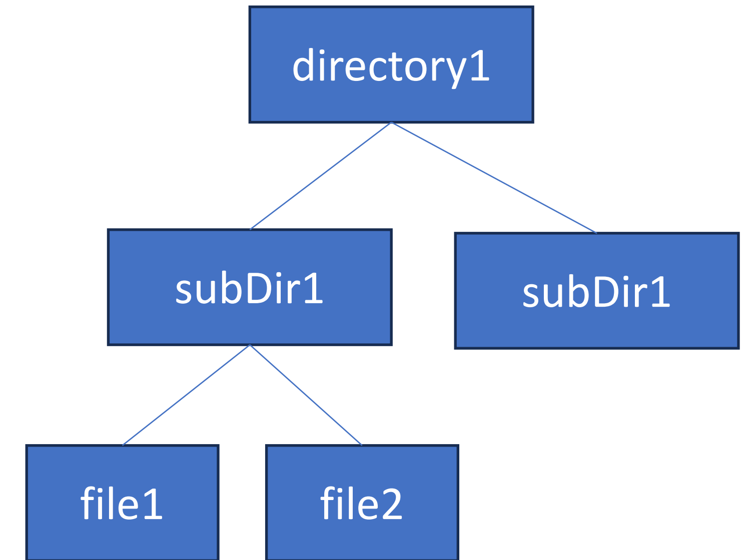
- ✓ RMI
- ✓ **Common**
- ✓ Naming
- ✓ Storage

Path class Overview

- Common package contains the class Path which contains helper methods that are used by Naming Server and the Storage Servers.

- Path creation
- Listing
- toString
- Equals
- Hashcode
- isRoot
- …

Path class – Highlights



Path: a sequence of components (names of files/directories)

Root: empty list or array of components,
string representation: "/"

Not Root:

directory1	subDir1	file1

string representation: `"/directory1/subDir1/file1"`

File toFile(File root) Expected Performance

Starting at the given directory, convert the given path to a File

i.e. create/add all path components starting at the given directory

Path[] List(File directory) Expected Performance

Create and Return a list of the Paths of all files under the given directory

Today's Outline

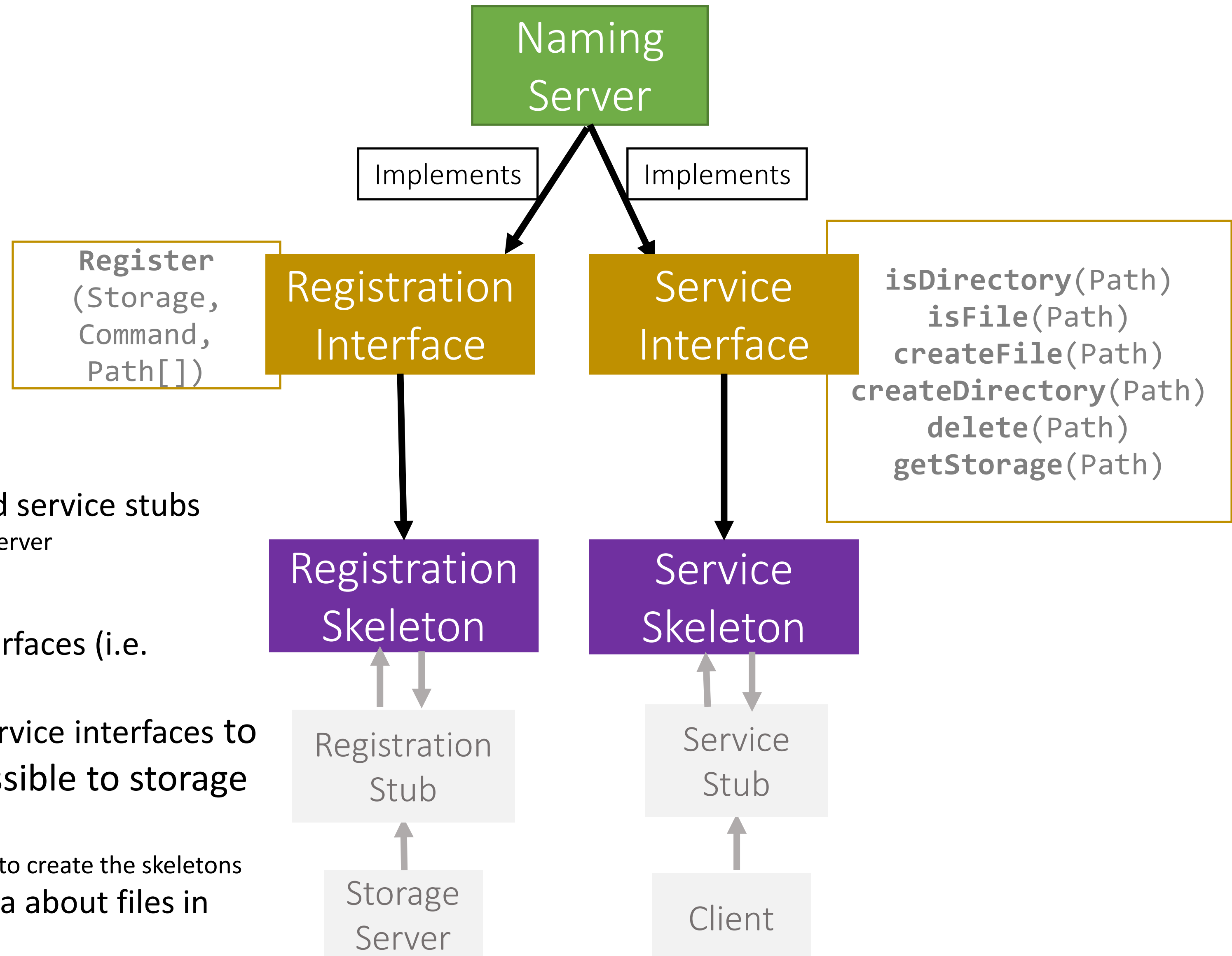
Packages dive-in:

- ✓ RMI
- ✓ Common
- ✓ Naming
- ✓ Storage

Naming package

The naming package contains:

1. **Registration interface**
2. **Service interface**
3. **NamingStubs.java** public class:
   Defines service and registration ports
   Has static methods to create registration and service stubs
      You could use it to create the stubs at the Storage Server
4. **NamingServer.java** class:
   - *Implements Registration and Service* interfaces (i.e. implements all their methods)
   - *Creates skeletons* for Registration and Service interfaces to make methods them remotely accessible to storage servers and clients.
     - Uses Port numbers defined in NamingStubs.java to create the skeletons
   - Creates and maintains a repo of metadata about files in the system
     - Maps files to hosting servers
     - Uses a directory tree to track files



Naming server – Directory Tree

- Creates and maintains the **FileStack directory tree**:
  - ✓ *Top-level directory* being the *root* represented by the path "/".
  - ✓ *Inner tree* nodes represent *directories*,
  - ✓ the *leaves* represent *files*
- Builds its tree during registration.
- After registration, uses its tree to handle operations (e.g. getStorage()).
- It is important to design the directory tree in a way that allows the NamingServer to easily *look-up, traverse and alter* the tree, as well as *detect invalid paths*.

Building the Directory Tree During Registration

Example from Testing Code

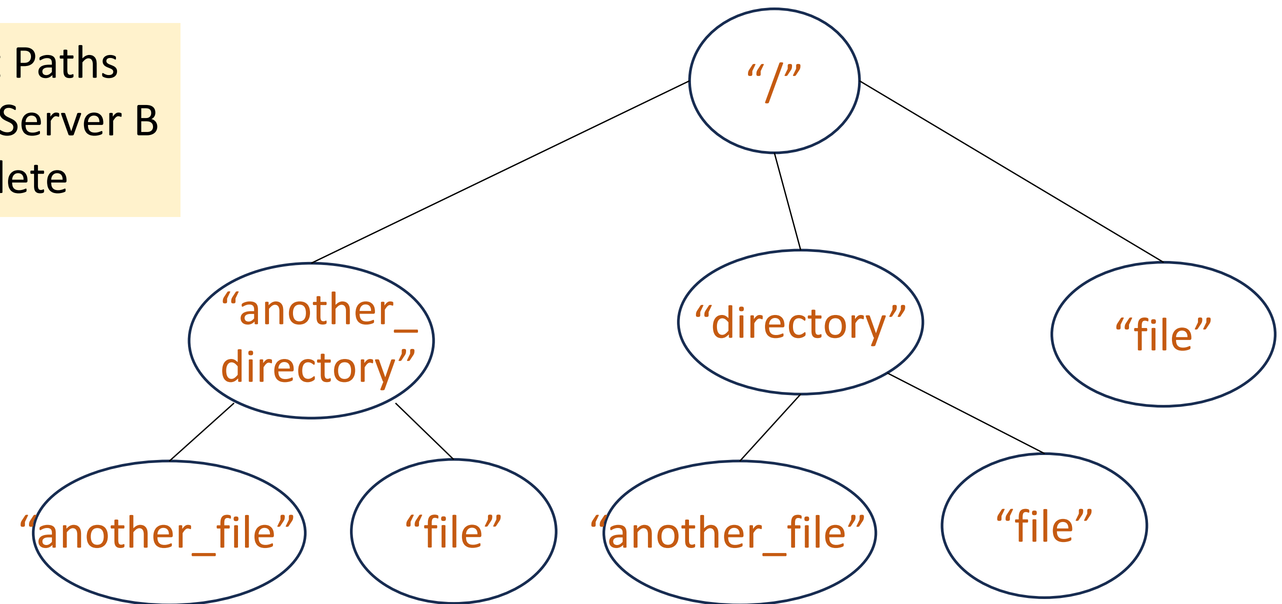
Storage Server A registers:

```
Path[] serverA_files = {
    • Path("/file"),
    • Path("/directory/file"),
    • Path("/directory/another_file"),
    • Path("/another_directory/file")
};
```

Storage Server B registers:

```
Path[] serverB_files = {
    ✗ Path("/file"),
      Path("/directory/file"),
    • Path("/another_directory/another_file")
};
```

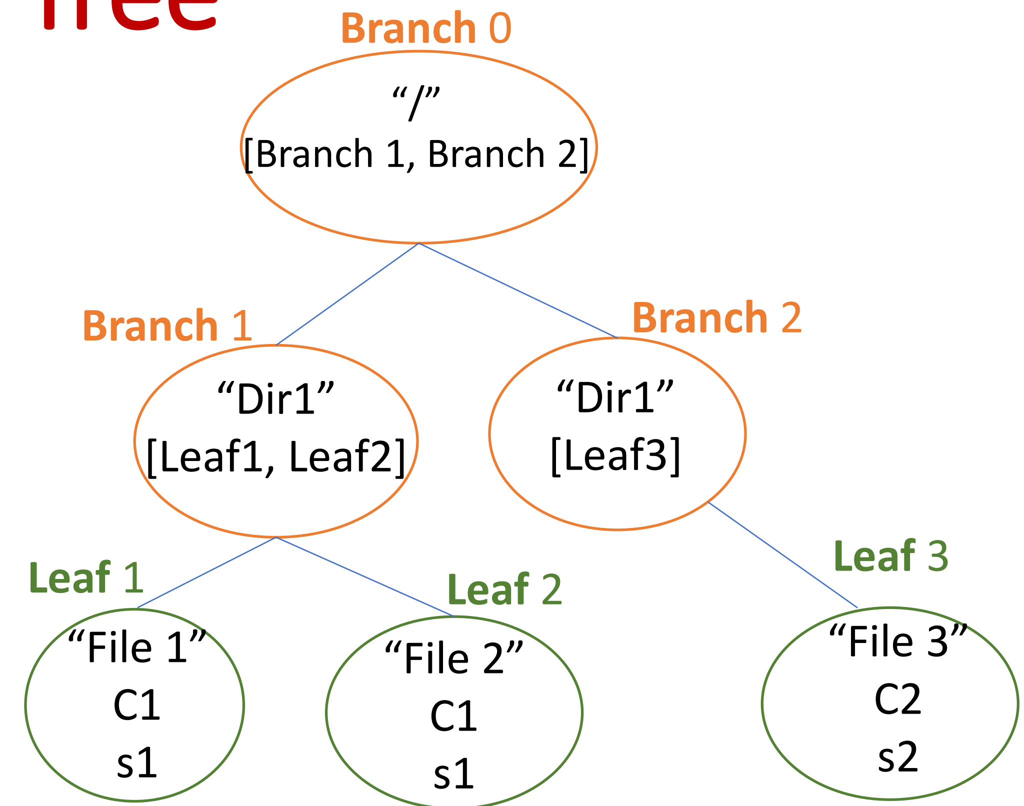
What Paths should Server B delete



Rule: Files with same Paths shouldn’t be duplicated across Storage Servers

Implementing the Directory Tree

- How can we build the Directory Tree?
  - One way is to use Leaf/Branch approach:
    - Leaf will represent:
      - A file (name) and stub
    - Branch (inner node) will represent:
      - A list of Leafs/Branches

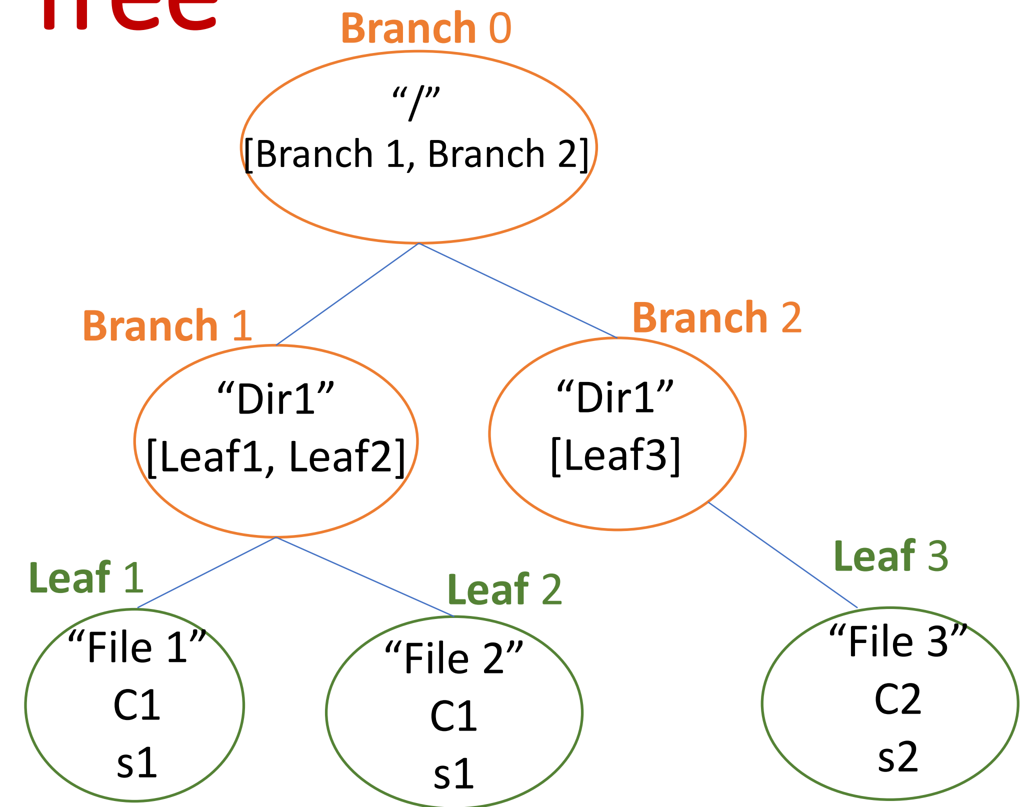


Implementing the Directory Tree

```
public class Node {
    String name;
}

public class Branch extends Node {
    ArrayList<Node> list;
}

public class Leaf extends Node {
    Command c;
    Storage s;
}
```



جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Implementing the Directory Tree

- What data should it Capture??
  - Go back to all the methods that the naming server needs to implement
  - For each method, think of what information do you need to capture in the nodes to be able to complete the method/operation?
    - You will leverage Path helper methods also to complete these operations

```
Register (Storage, Command, Path[])
```

```
isDirectory(Path)
isFile(Path)
createFile(Path)
createDirectory(Path)
delete(Path)
getStorage(Path)
```

Naming server – Methods Highlights

- Start()
  - Start skeletons
- Stop()
  - Stop skeletons

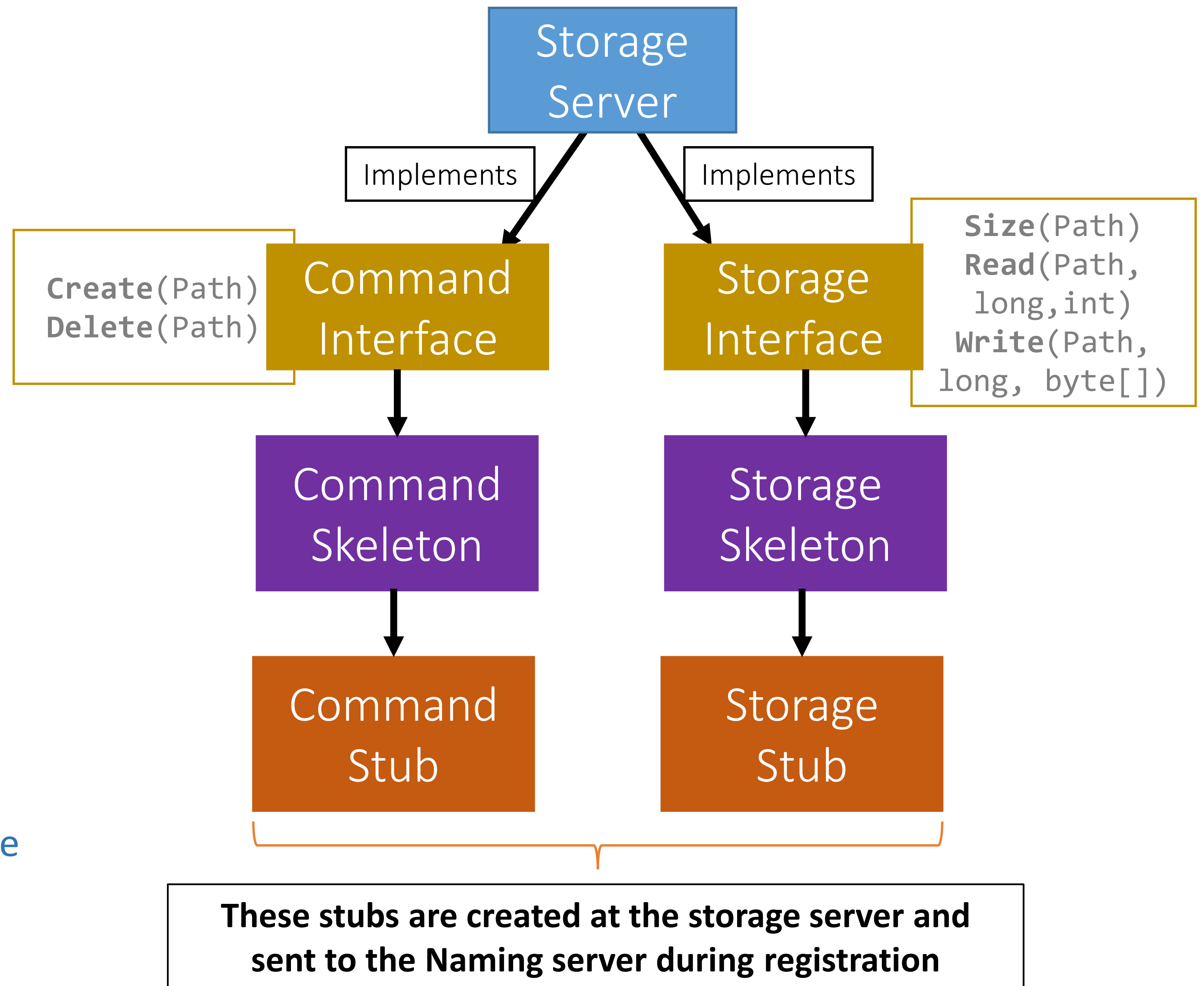
Today's Outline

Packages dive-in:

- ✓ RMI
- ✓ Common
- ✓ Naming
- ✓ Storage

Storage Package

- The **Storage** Package contains:
  - **Command**.java (interface)
  - **Storage**.java (interface)
  - **StorageServer.java** (public class)
    - Implements:
      - Command ***Interface***
        - **methods(s):** create, delete
      - Storage ***Interface***
        - **methods(s):** size, read, write



Storage Server – start()

- The StorageServer **start()** function will:
  - **Start** the Skeletons:
    - *Command* Skeleton
    - *Storage* Skeleton
  - **Create the stubs**
    - *Command* Stub
    - *Storage* Stub
  - **Registers** itself with the Naming Server **using**:
    - Its **files**
    - The created **stubs**
  - Post registration, we receive a list of **duplicates** (*if any*):
    - **Delete** the duplicates
    - ***Prune* directories** if needed

The Directory Tree should not have duplicate File Paths across storage servers.

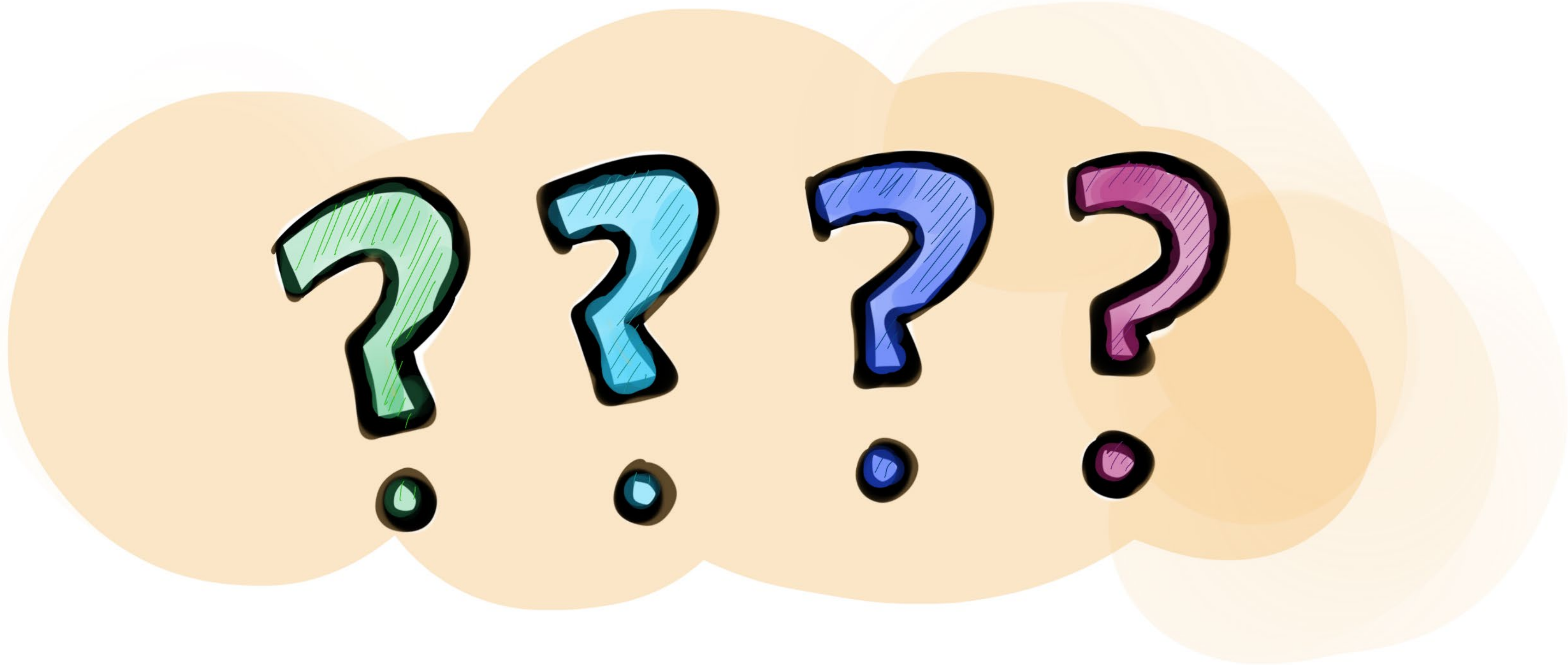
Whenever a storage server registers, if the tree already tracked the file with the same path at another server that registered earlier,

then the new registering server should delete it

Storage Server – stop()

- The StorageServer **stop()** function will:
  - **Stop** the skeletons:
    - *Command* Skeleton
    - *Storage* Skeleton

Other File Methods are straight forward ☺



Early Feedback