15-440: Distributed Systems

Recitation 3

Zeinab Khalifa

Sept 12, 2019

Outline

- **Project 1 (Distributed File System)**
- **Entities, Architecture and Communication**
- **RMI**
- **Interfaces**
- **Skeleton & Stub**
- **Example**

Project (1)

PS 2 is out!

Project 1

- Implement a Distributed File System (DFS)
- DFS stores a vast amount of data that does not fit on a single machine.
- Distributed files (physically) on a set of servers **(storage servers)**
- Users/Clients perform operations on the files stored on these **remote servers** using (RMI)
- Clients contact a naming server, which maps every file name to a **storage server** to identify the storage server that hosts the file they require.

Entities, Architecture and Communication

Entities

Client

Creates, reads, writes files using RMI

Storage Server

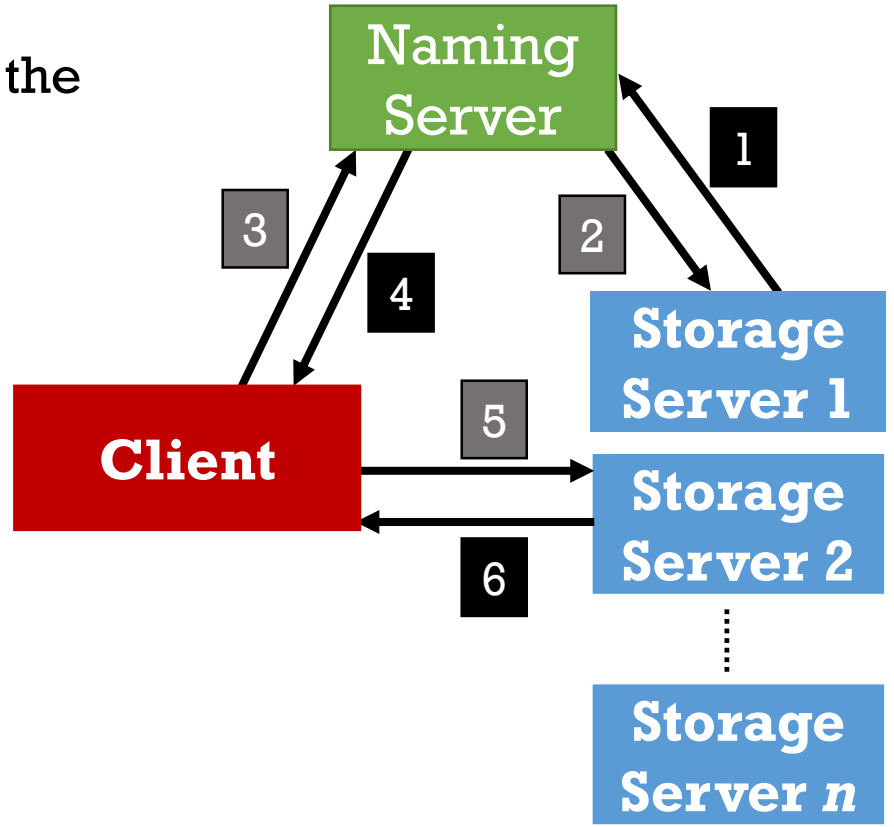
Physically hosts the files in its local file system

Naming Server

- Runs at a predefined address
- Maps file names to storage servers
- It has metadata

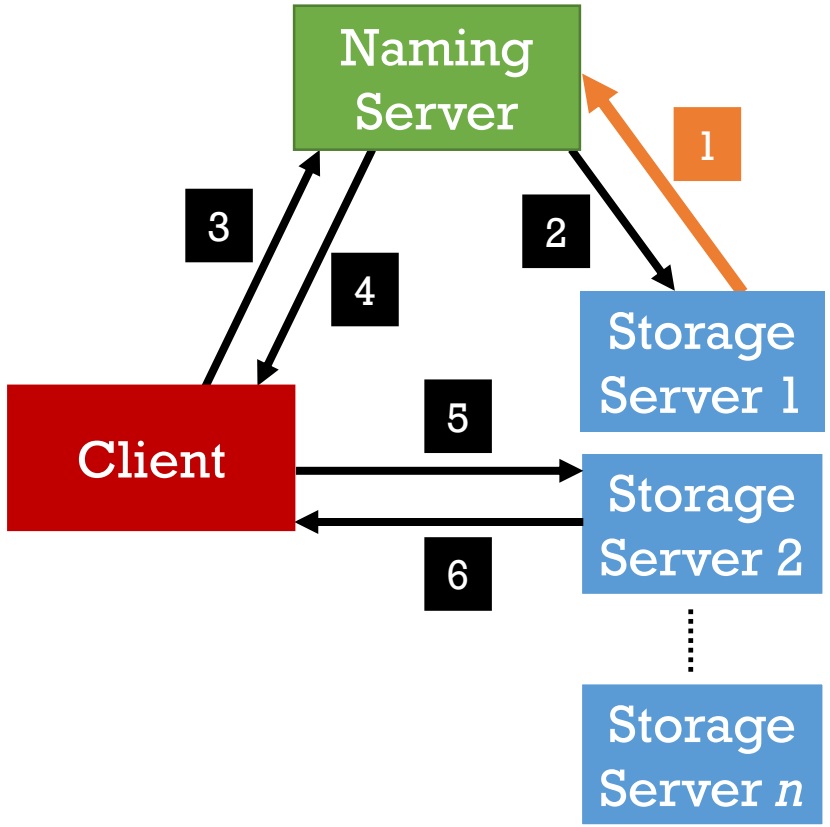
Client-Server Architecture

- Client-server architecture
- 2, 3, 5: requests originating from clients to servers
- Naming server acts as a client in 2, when requesting the services of storage servers.
- 1, 4, 6: services provided by the servers in response.



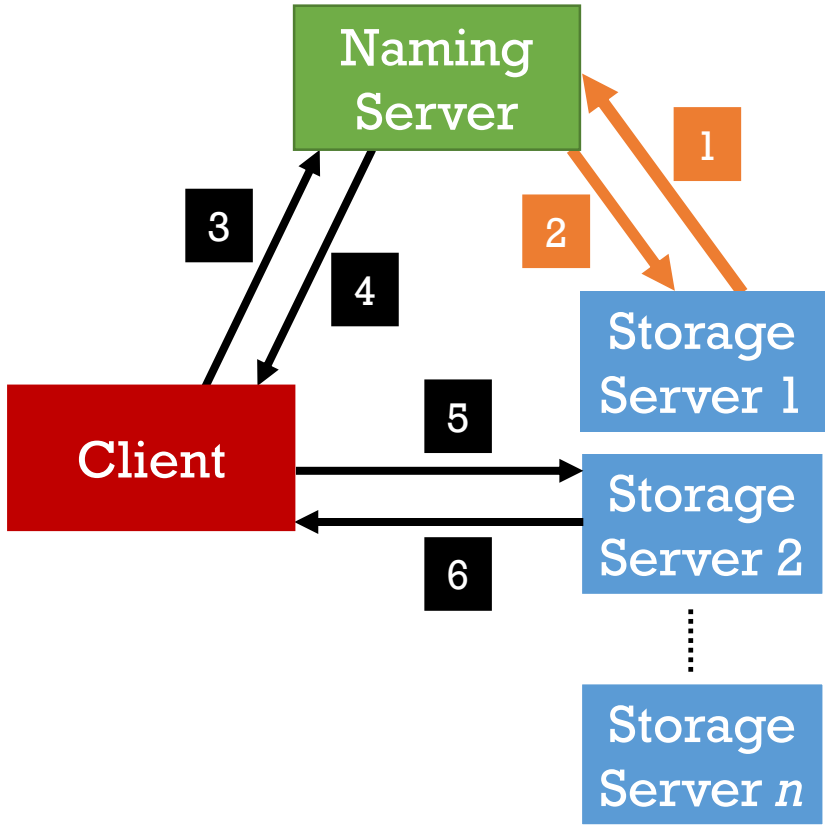
Communication

- **Naming server – Storage server**
- **Registration phase**: each Storage Server sends a list of paths (representing the les that it currently hosts) to the Naming Server
- The Naming Server traverses this list and adds the paths to its directory



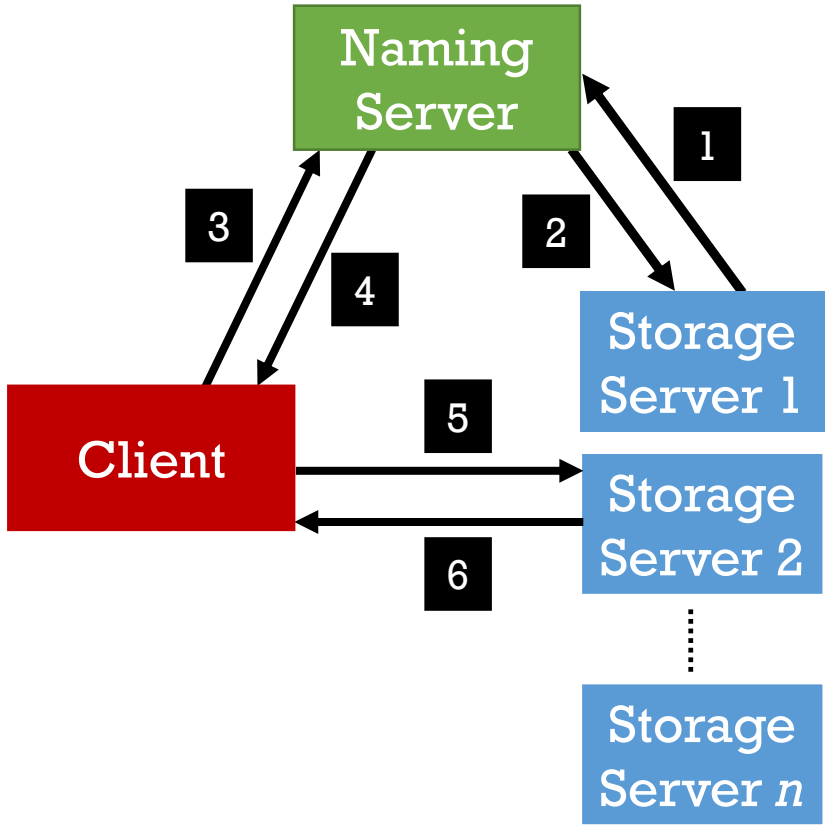
Communication

- **Naming server – Storage server**
- **Registration phase**: each Storage Server sends a list of paths (representing the les that it currently hosts) to the Naming Server
- The Naming Server traverses this list and adds the paths to its directory
- During registration of Storage Server SS1, if the Naming Server encounters f1.txt sent by SS1 in its directory tree, then f1.txt is deemed as a **duplicate** file.



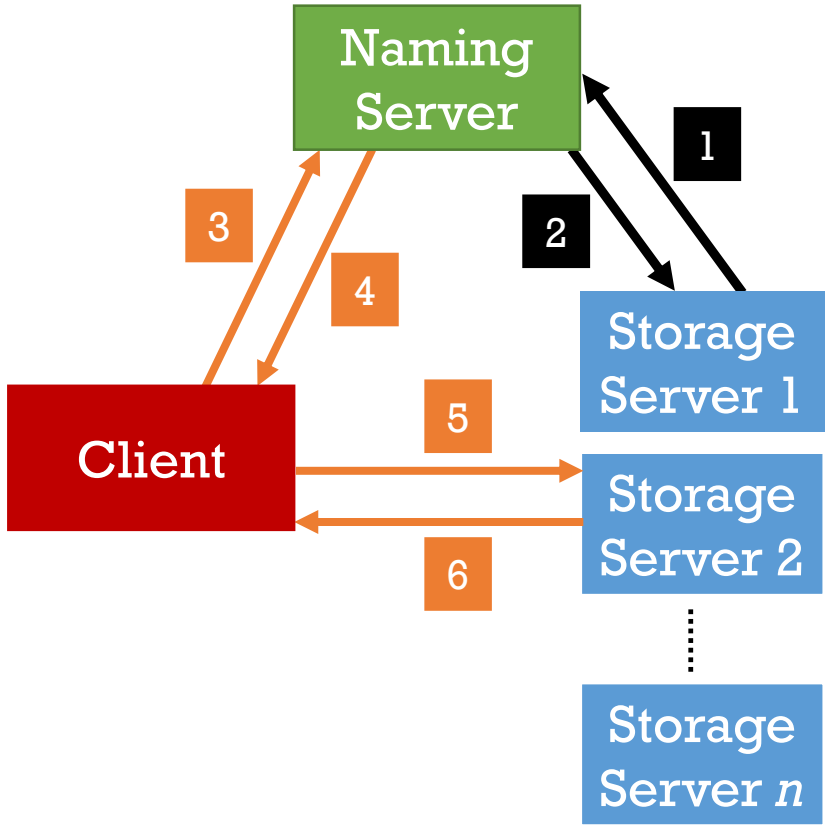
Communication

- **Naming server – Storage server**
- **Registration phase**: each Storage Server sends a list of paths (representing the les that it currently hosts) to the Naming Server
- The Naming Server traverses this list and adds the paths to its directory
- During registration of Storage Server SS1, if the Naming Server encounters f1.txt sent by SS1 in its directory tree, then f1.txt is deemed as a **duplicate** file.
- After registration, the system is now ready for the client to invoke requests.



Communication

- **Client – Naming Server**
- Client contacts the naming server whenever it needs to perform an operation on a file.
- Some requests (operations) cannot be handled directly by the naming server, then it replies back with the storage server that hosts the file (read, write, etc.)
- Other operations can be directly handled by the naming server (createFile, createDirectory, list, etc.)



RMI

RMI

- When a Client invokes a method, it basically invokes a **remote** method (*and hence, **R**emote **M**ethod **I**nvocation*)
  - This is because the logic of the method resides on the 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**.

RMI

- The RMI library is based on two important objects:
  - **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

- The RMI library is based on two important objects:
  - **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

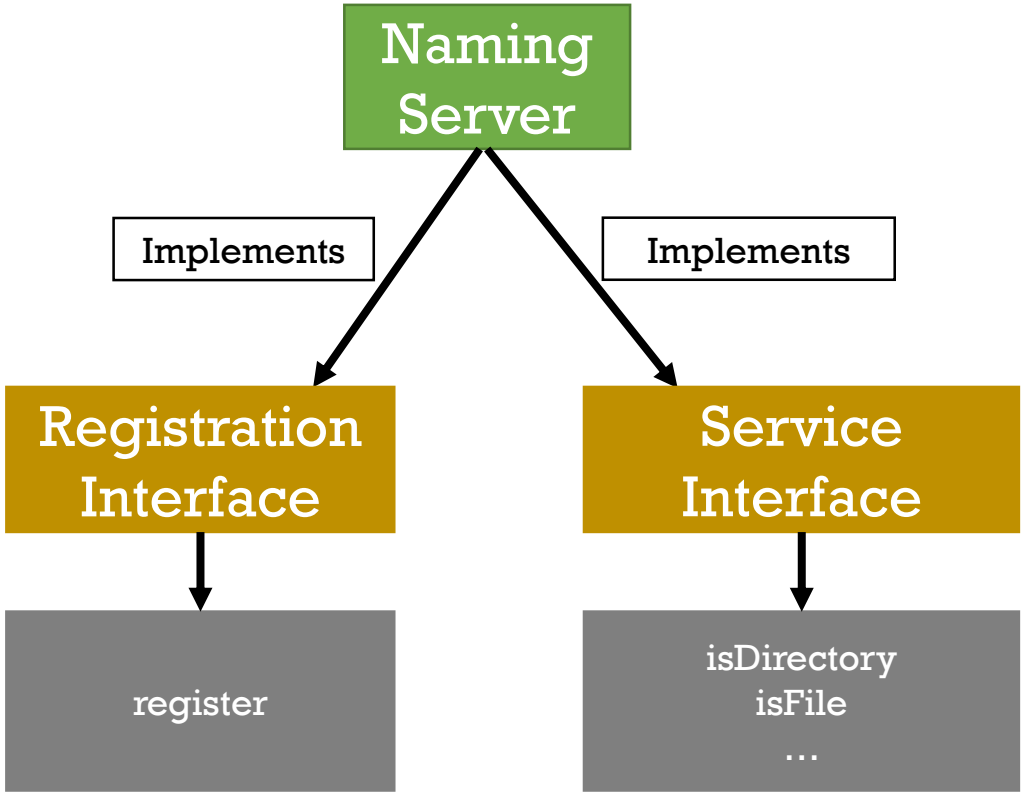
Interfaces

Interfaces

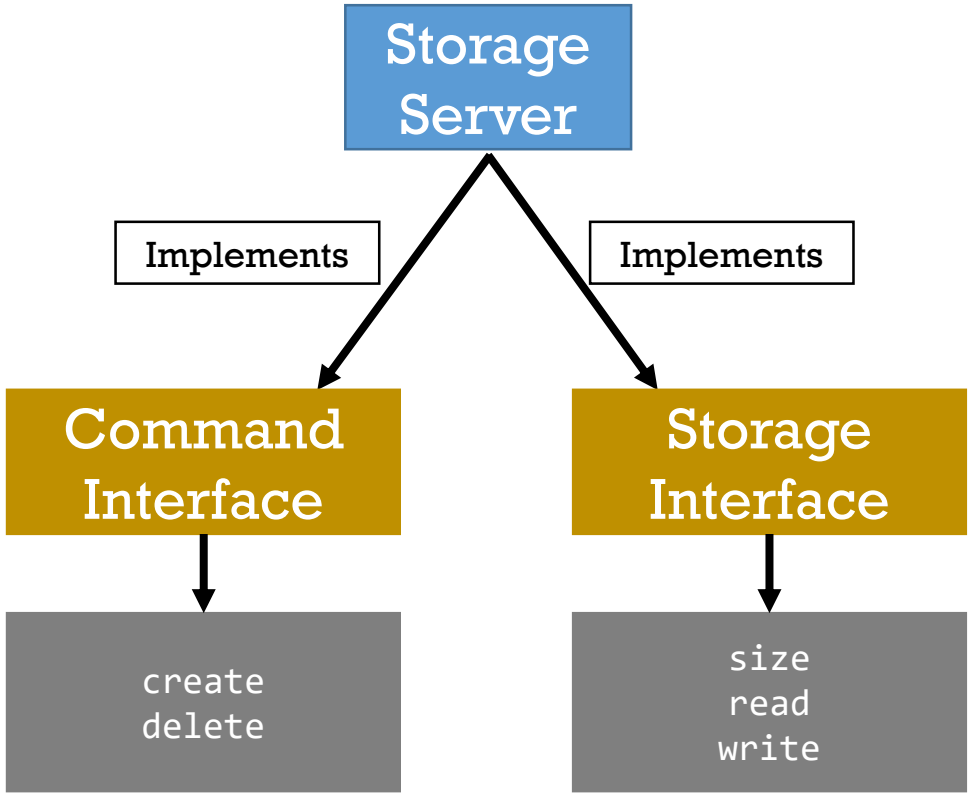
- Servers declare all their methods in **interfaces**

- Such interfaces contain a subset of the methods the server can perform

Naming Server Interfaces



Storage Server Interfaces

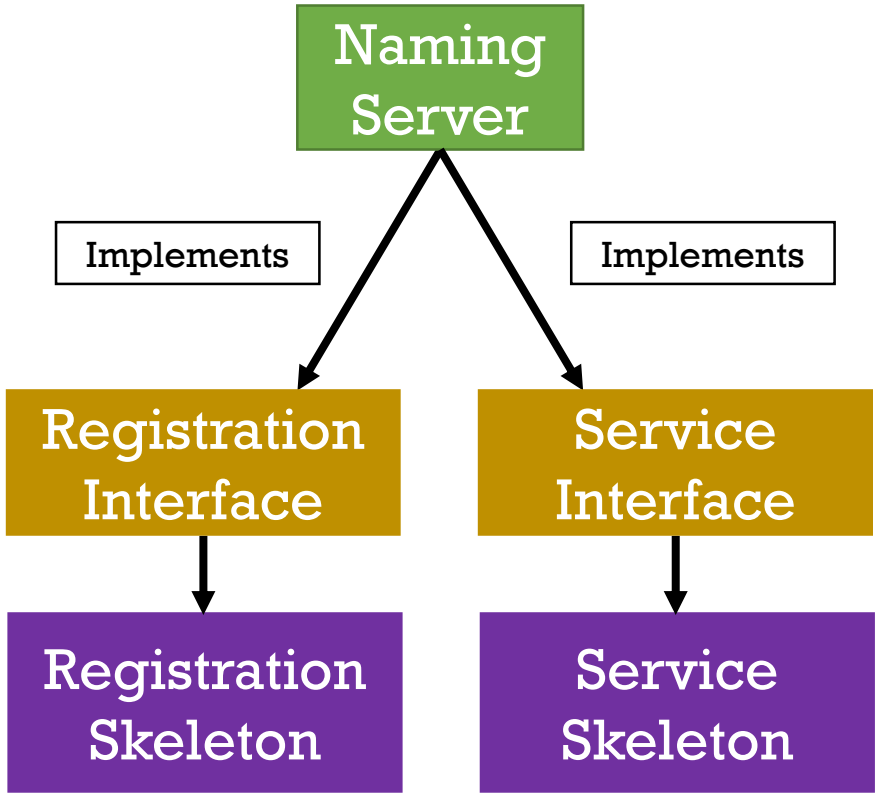


Creating Stubs & Skeletons

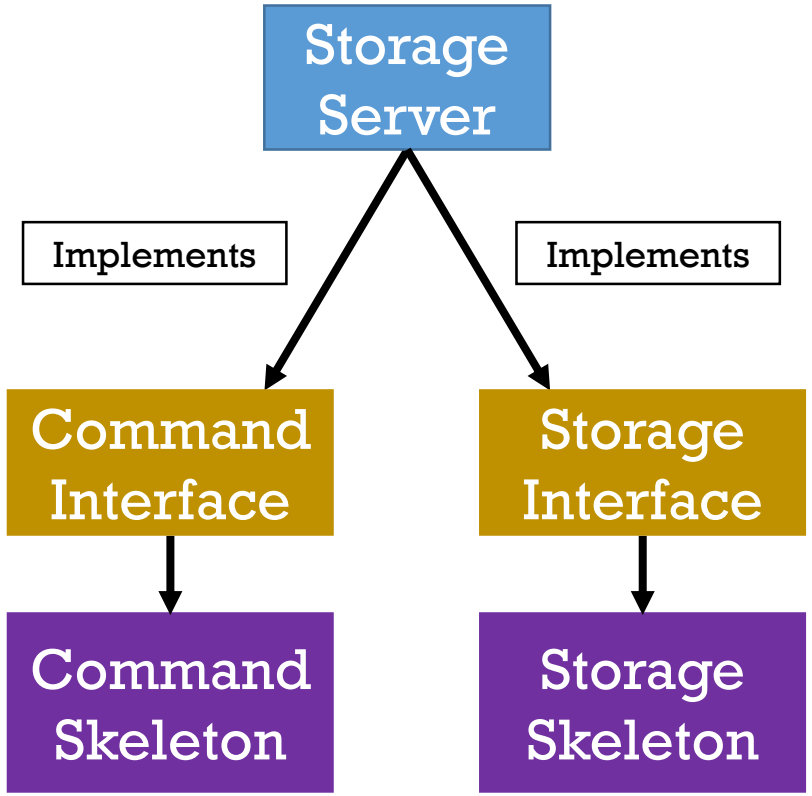
- For a client to create a **Stub**, it needs:
  - An **interface** of the corresponding **Skeleton**
  - **Network address** of the corresponding **Skeleton**

- For a server to create a **Skeleton**, it needs:
  - An **interface**
  - A **class that implements the logic of the methods** defined in the given interface
  - Network address of the server

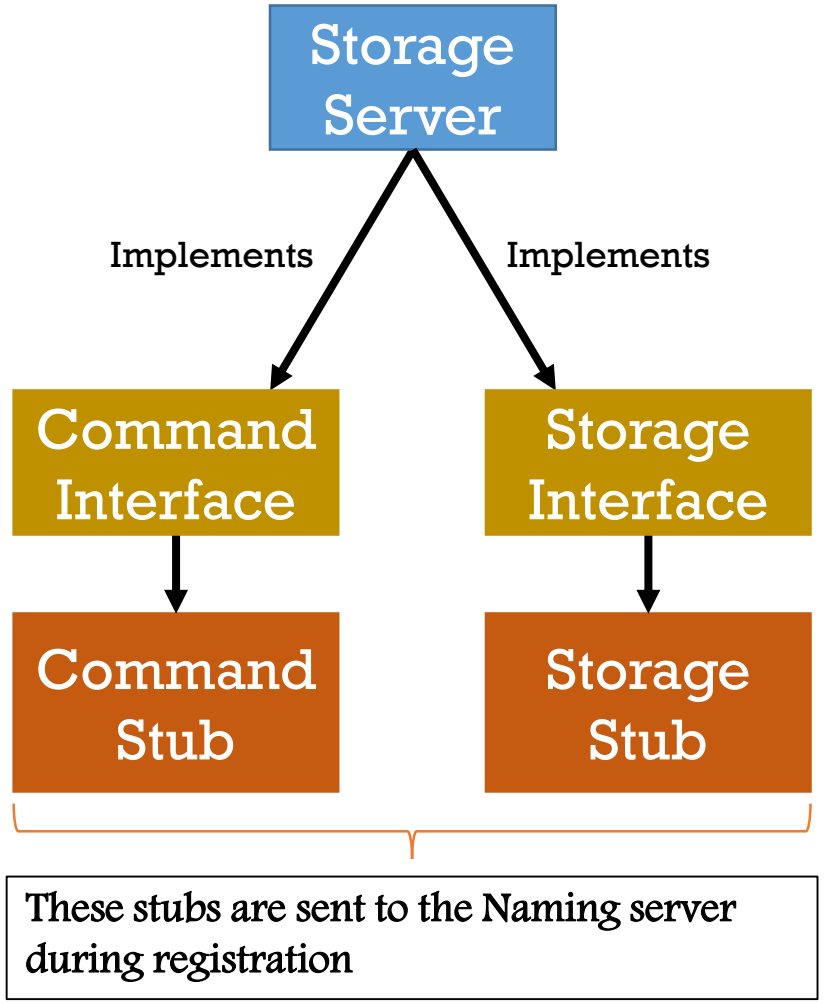
Naming Server Skeletons & Stubs



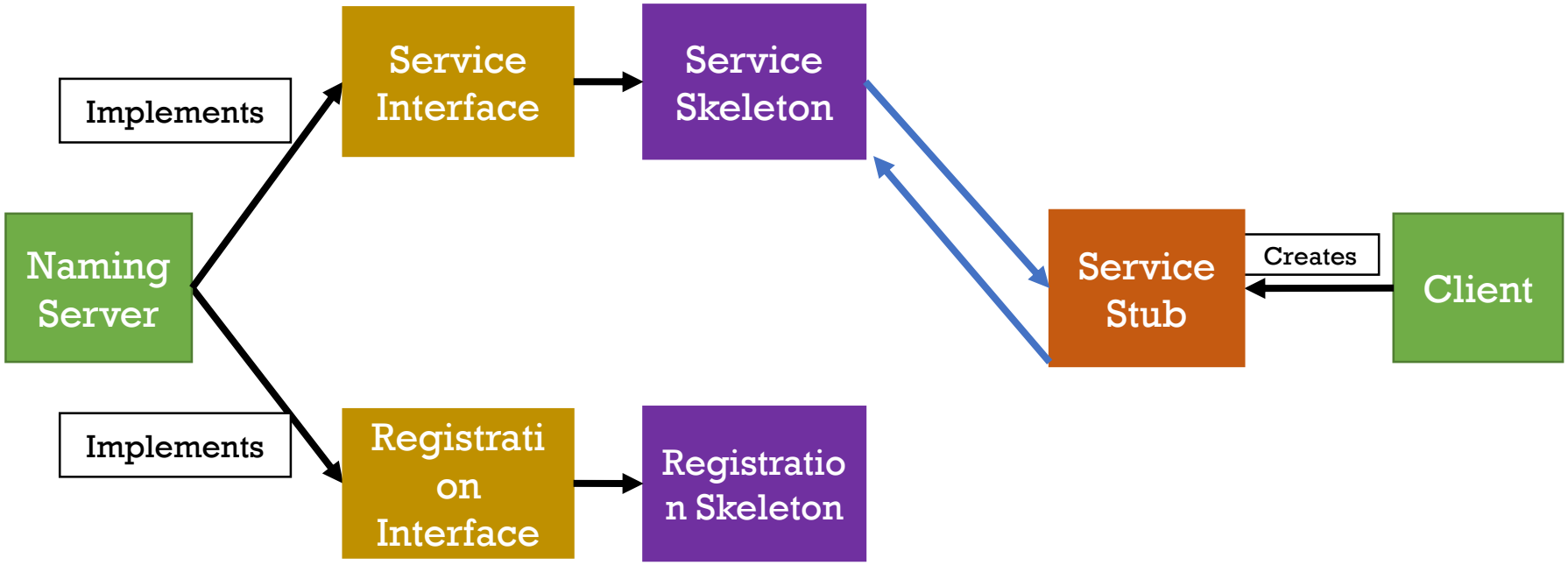
Storage Server Skeletons & Stubs



Storage Server Skeletons & Stubs

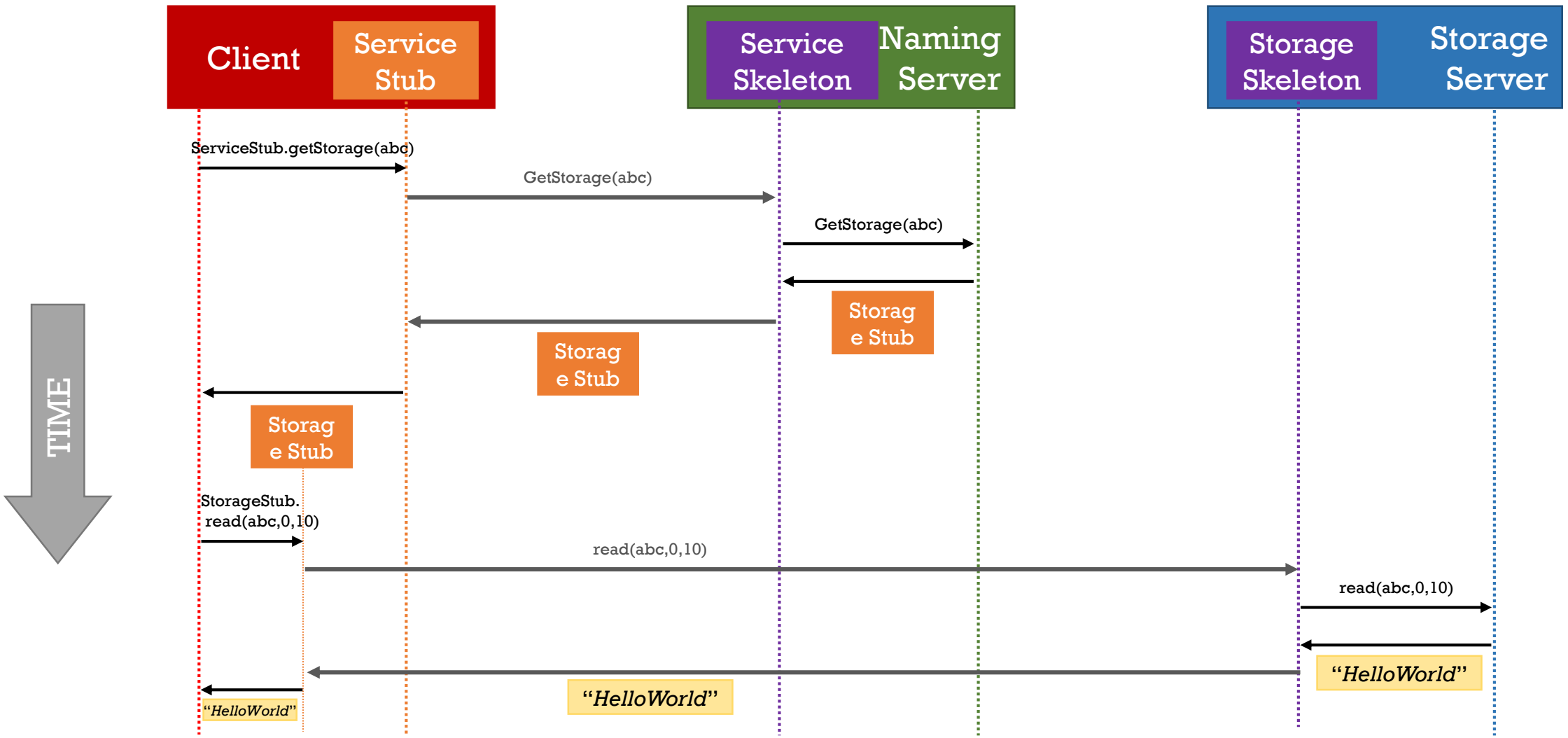


Simple Stub-Skeleton Communication



Example

Full Example: Client Read



Creating a Stub

- In Java, a stub is implemented as a ***dynamic proxy***
- A proxy has an associated ***invocation handler***
- **Example**: `getStorage` in Figure 2:
  - When `getStorage` is invoked on the Service Stub, the proxy encodes the method name (getStorage) and the argument(s) (file '*abc*')
  - The proxy sends the encoded data to the invocation handler
  - The invocation handler determines if it is a local or remote procedure, and acts accordingly (as how it was shown earlier)
- Go over `java.lang.reflect.Proxy` via the JavaDocs!