

15-440

# Distributed Systems

## Recitation 3

Zeinab Khalifa

# Project 1

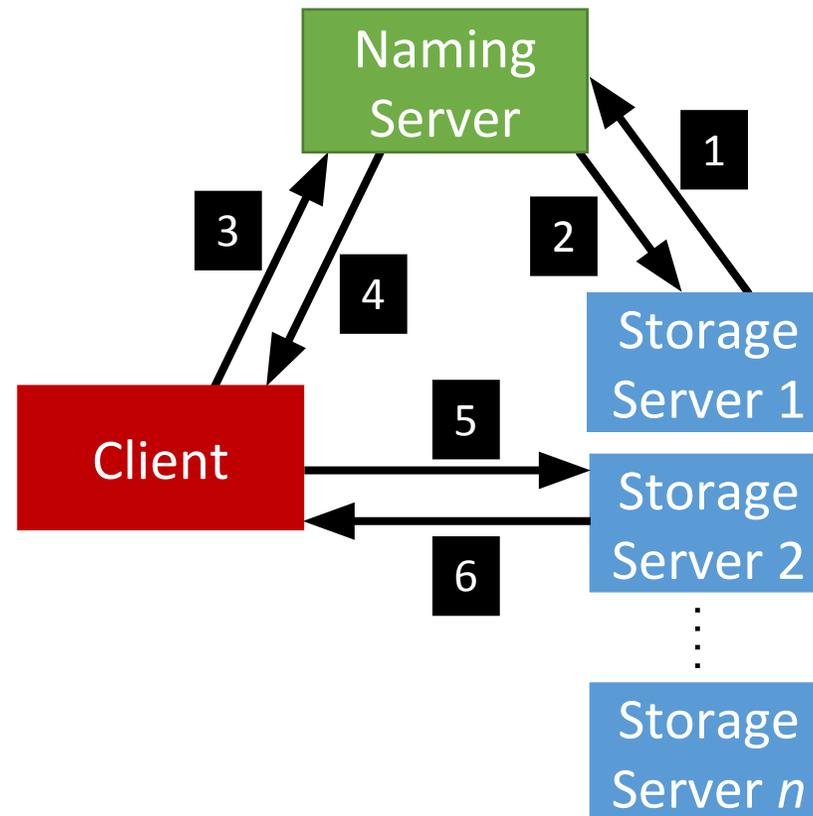
- Involves creating a *Distributed File System* (**DFS**):  
*FileStack*
- Stores data that does not fit on a single machine
- Enables clients to perform operations on files stored on **remote servers** (RMI)

# Entities

- Three main entities in FileStack:
  - **Client:**
    - Creates, reads, writes files using RMI
  - **Storage Servers:**
    - Physically hosts the files in its local file system
  - **Naming Server:**
    - Runs at a predefined address
    - Maps file names to Storage Servers
    - Therefore, it has *metadata*

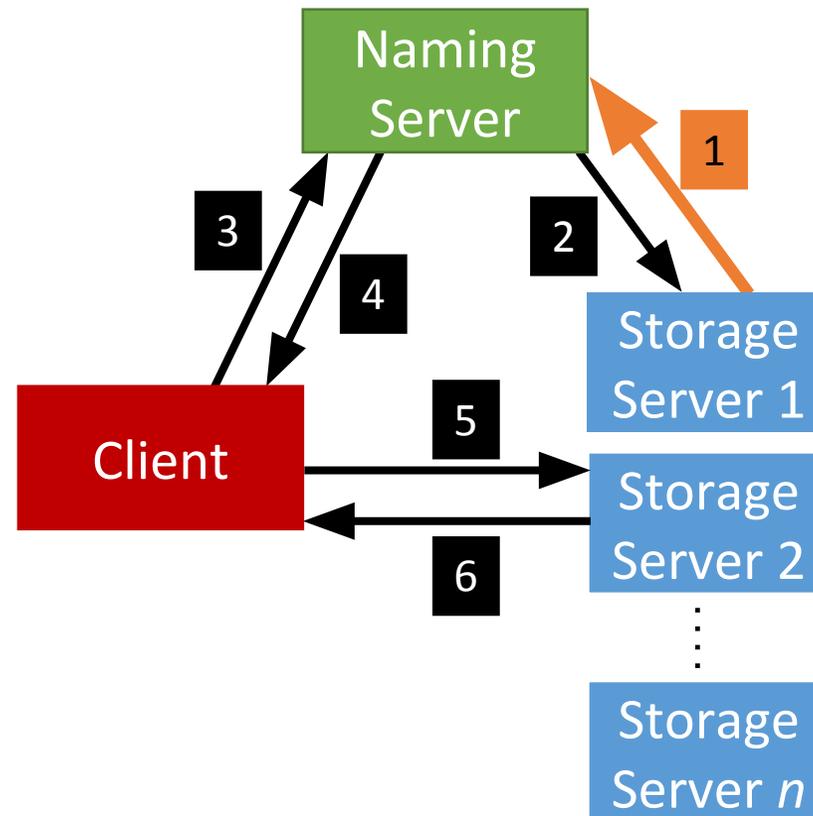
# Architecture

- FileStack will boast a Client-Server architecture:



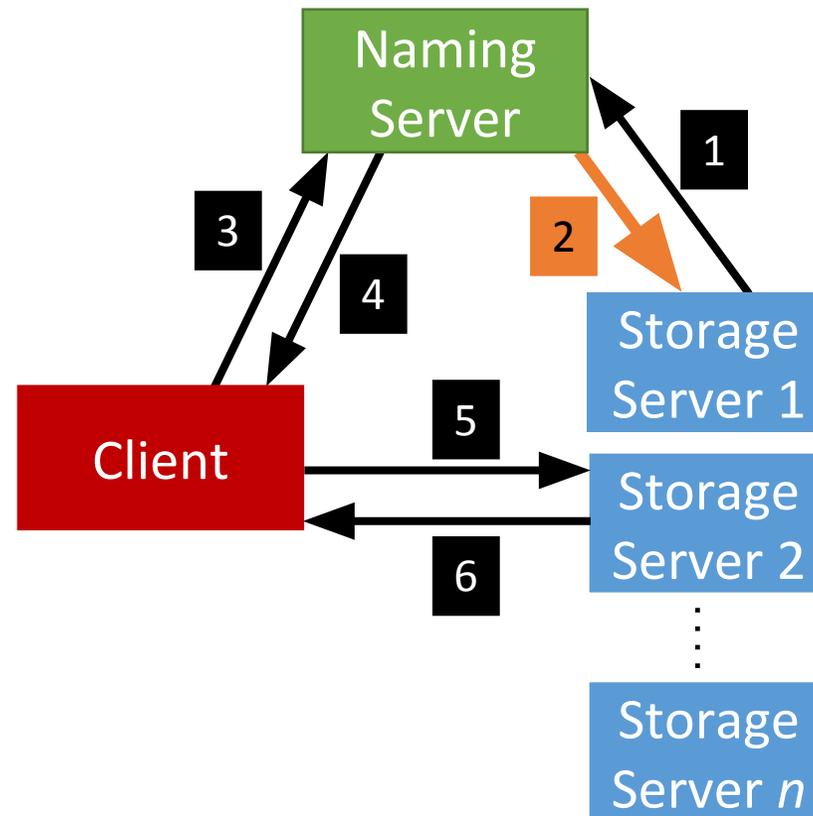
# Communication

- Registration phase



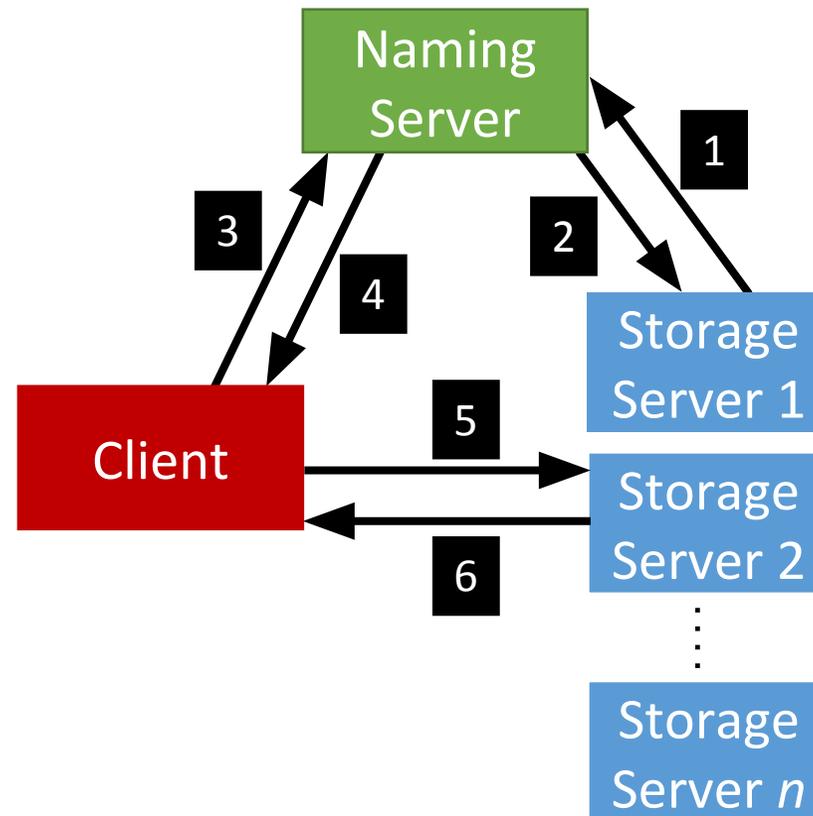
# Communication

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



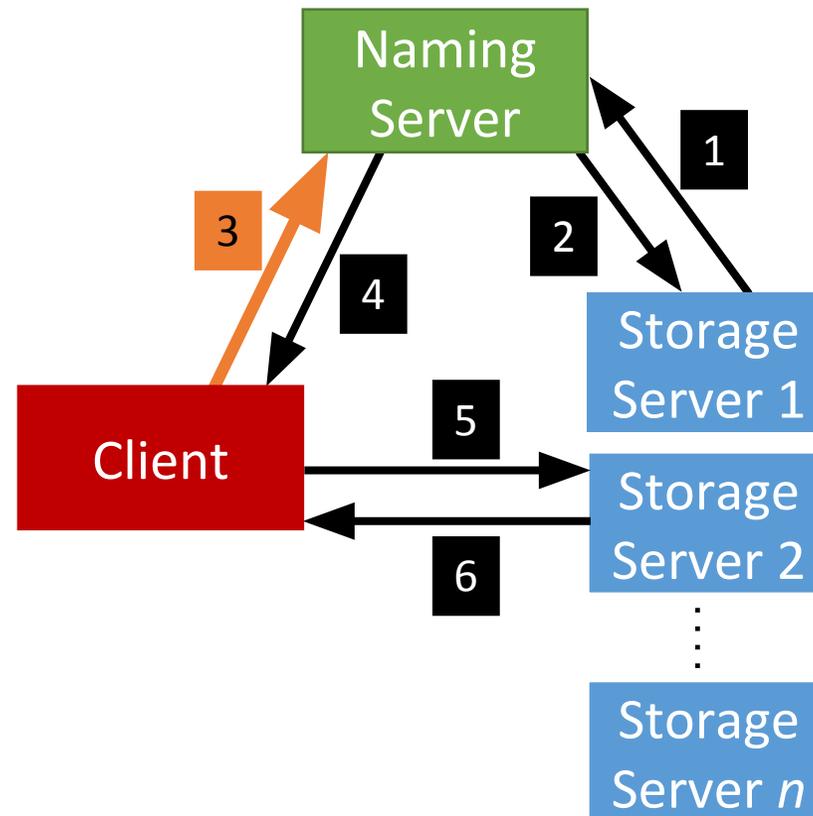
# Communication

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



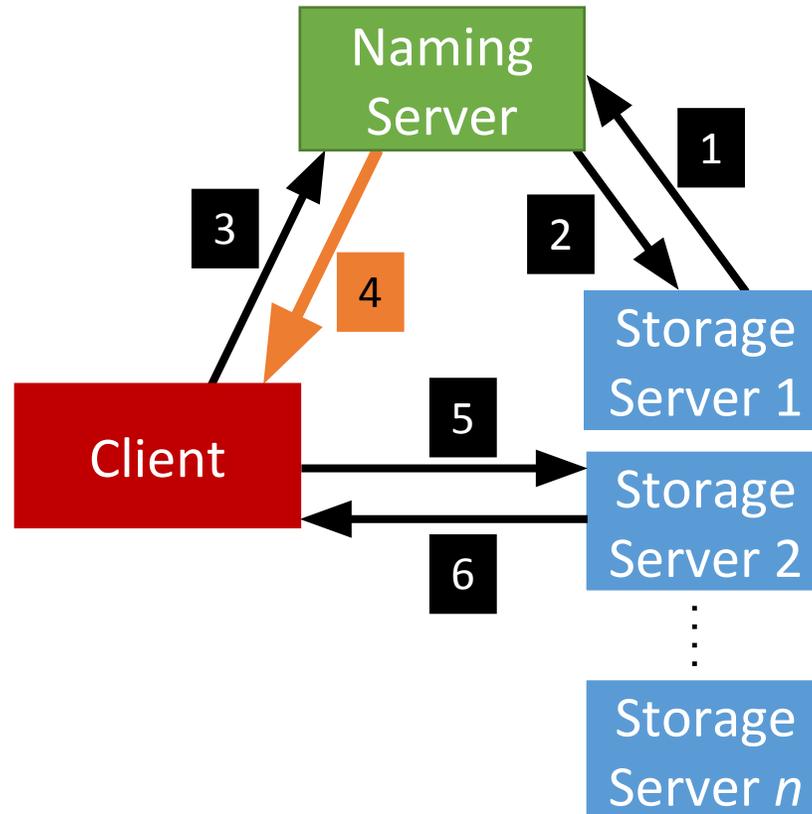
# Communication

- **Client** requests a file (to read, write etc...) from the **Naming Server**.



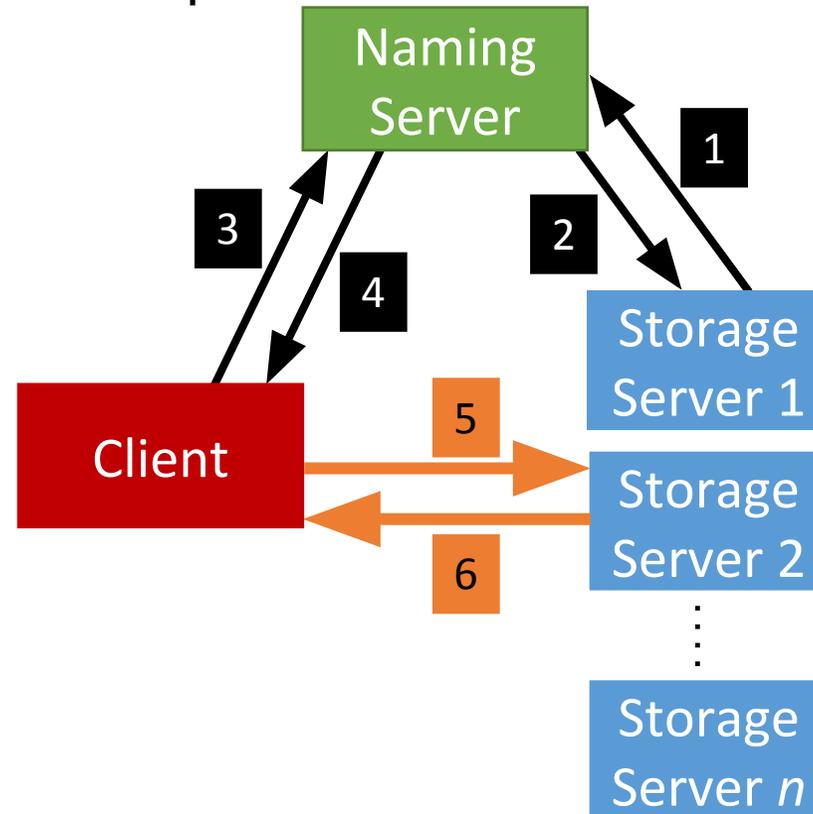
# Communication

- 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.



# Communication

- When a **Client** invokes a method, it basically invokes a **remote** method (*and hence, **Remote Method Invocation***)
  - 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 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.**

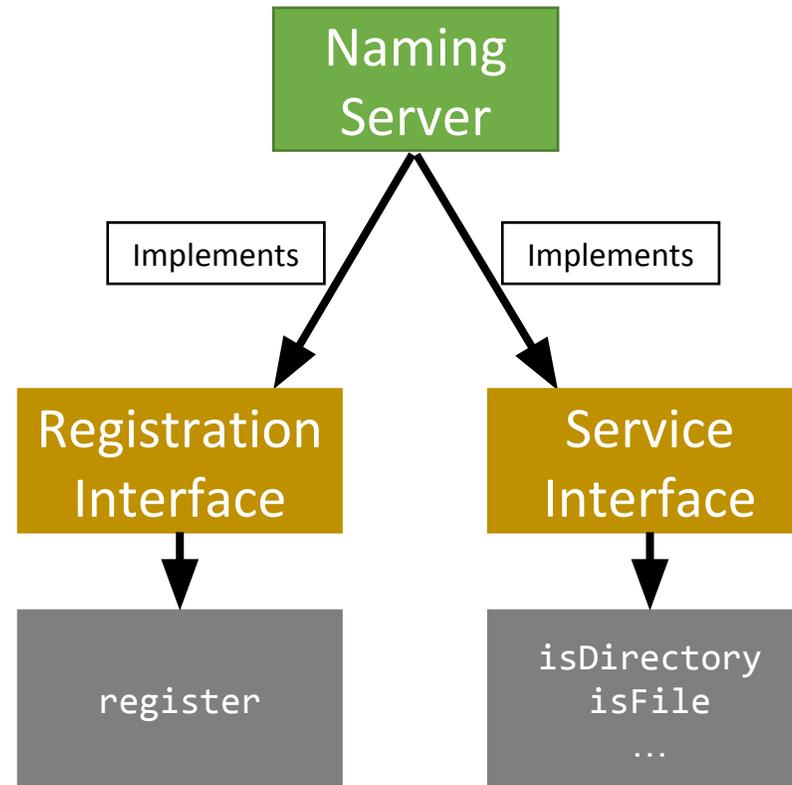
# RMI

- 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

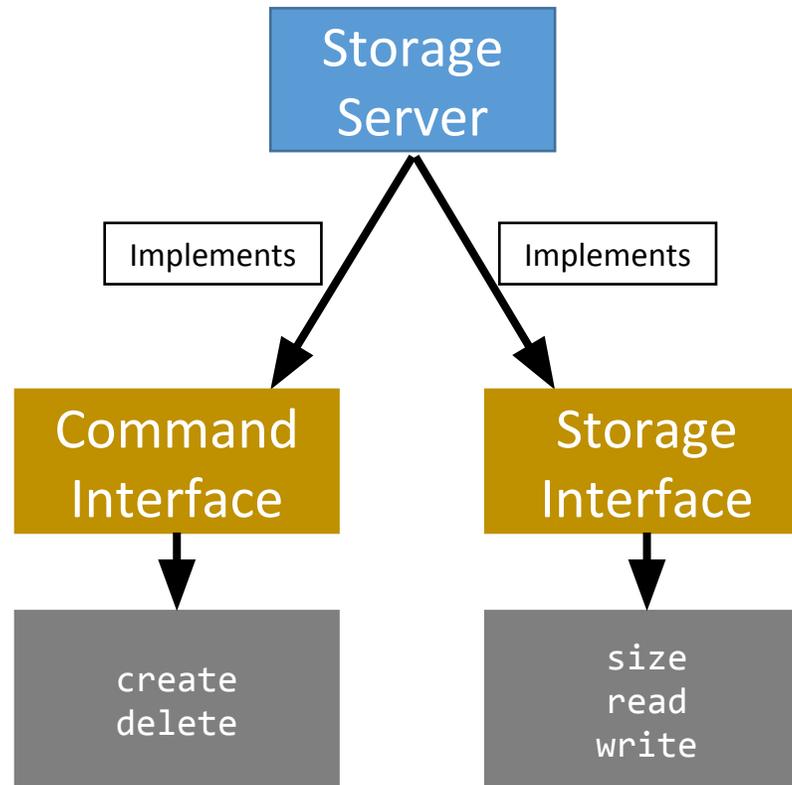
# Interfaces

- Servers declare all their methods in **interfaces**
- Such interfaces contain a subset of the methods the server can perform

# Naming Server Interfaces



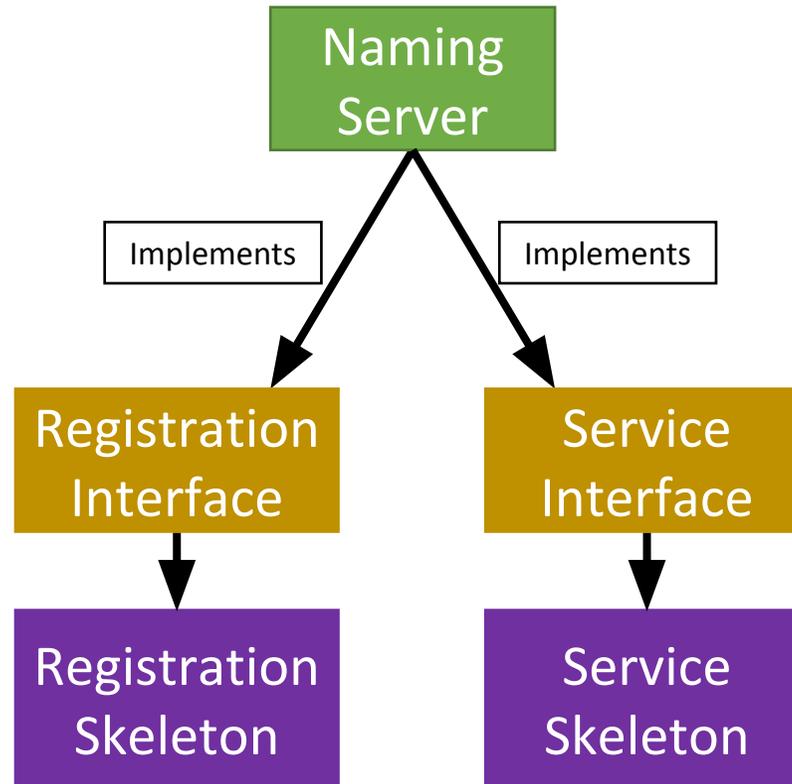
# Storage Server Interfaces



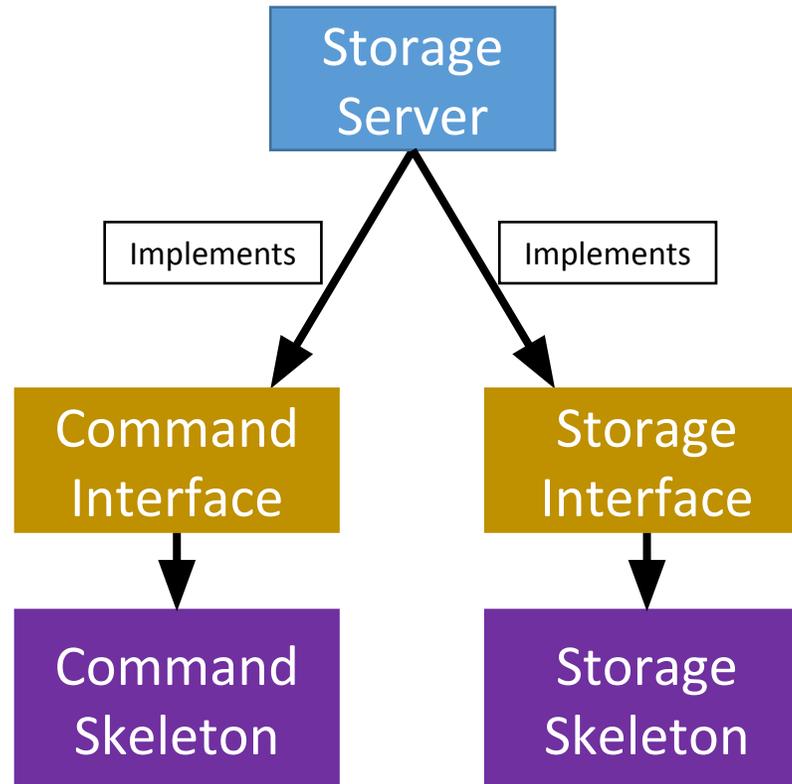
# Creating Stubs & Skeletons

- 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

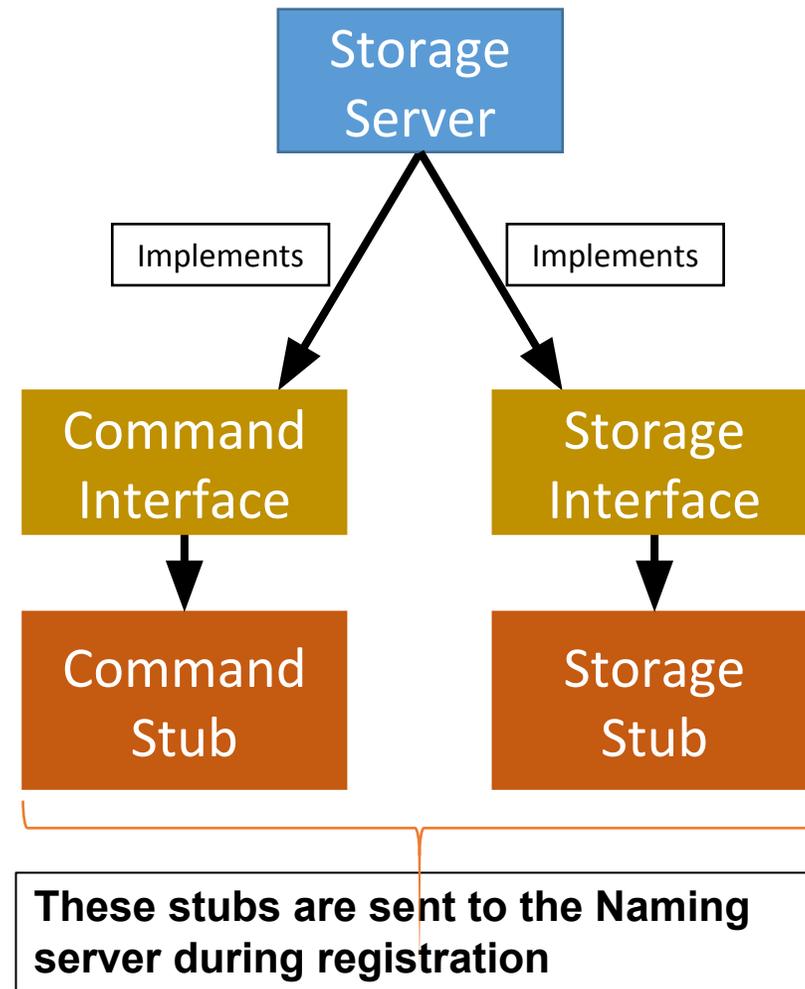
# Naming Server Skeletons & Stubs



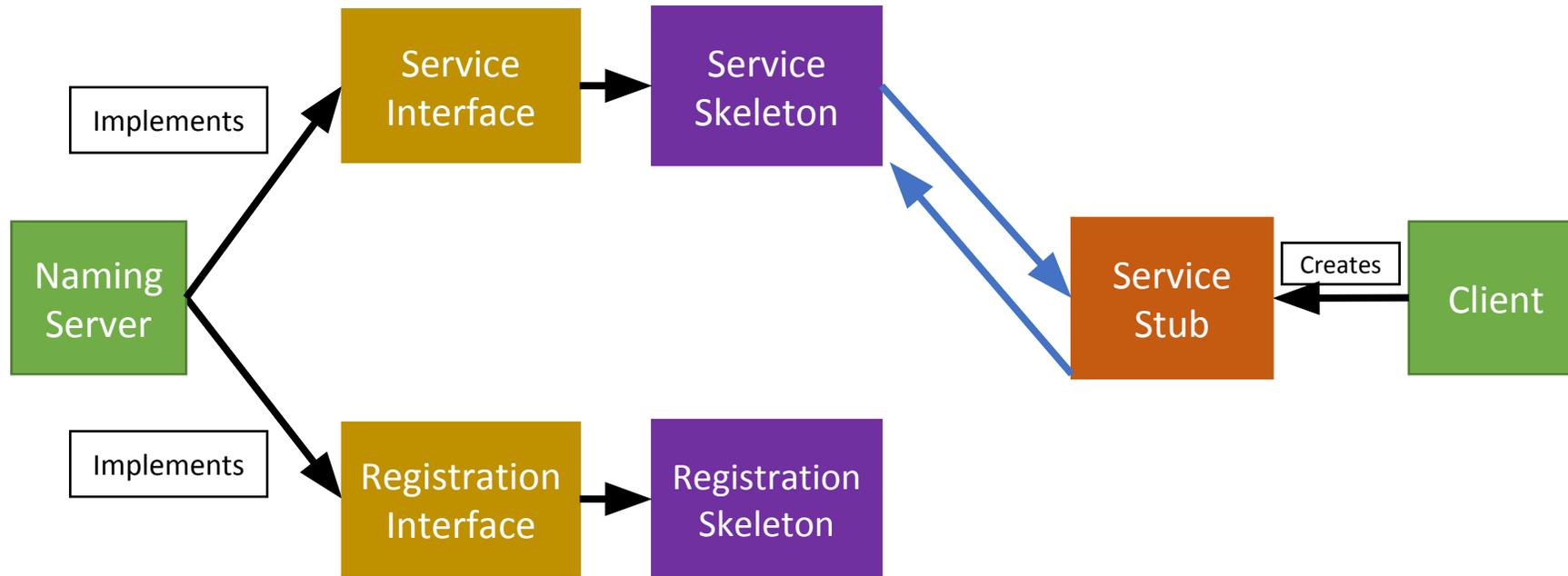
# Storage Server Skeletons & Stubs



# Storage Server Skeletons & Stubs



# Simple Stub-Skeleton Communication



# Full Example: Client Read

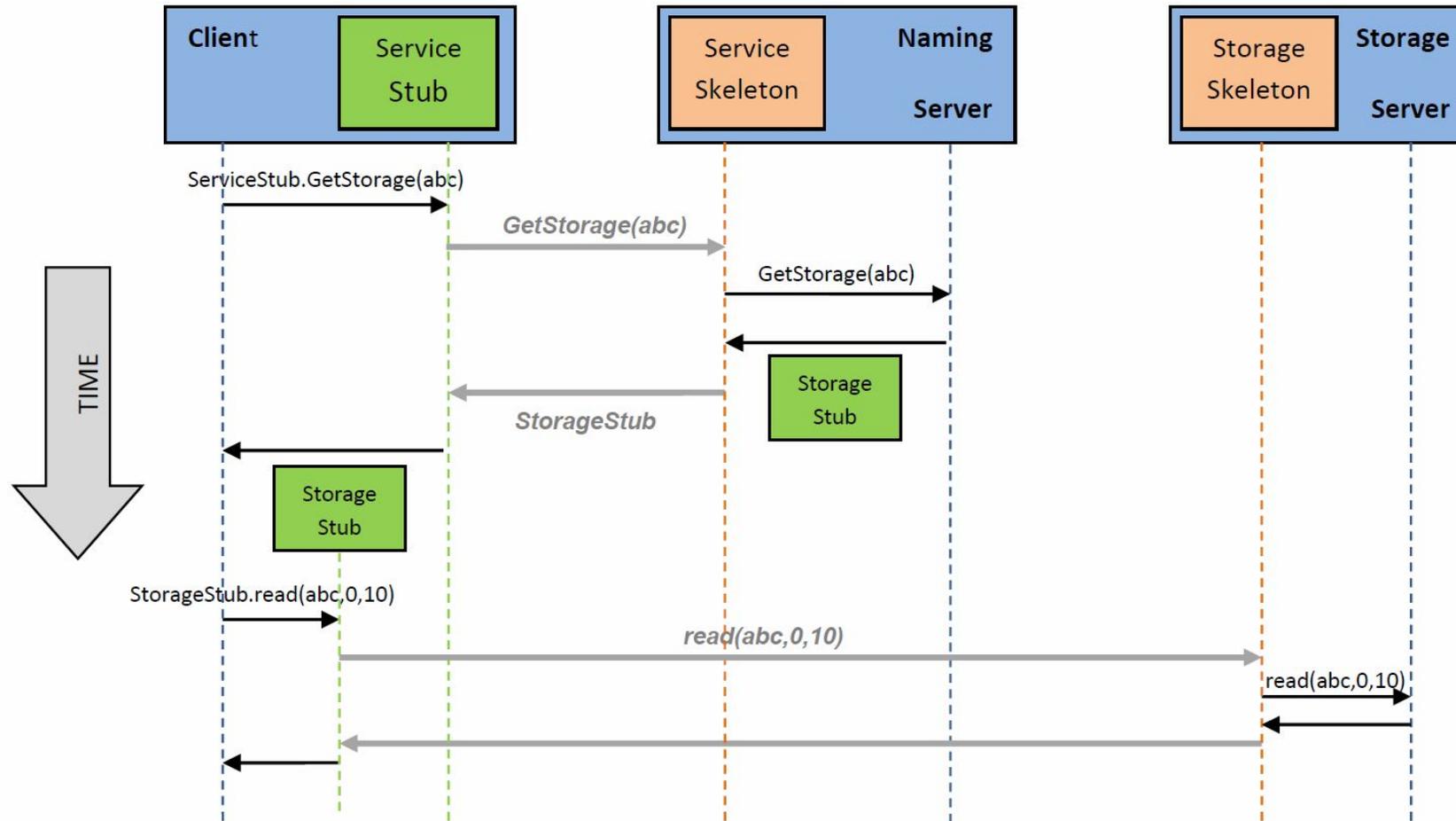
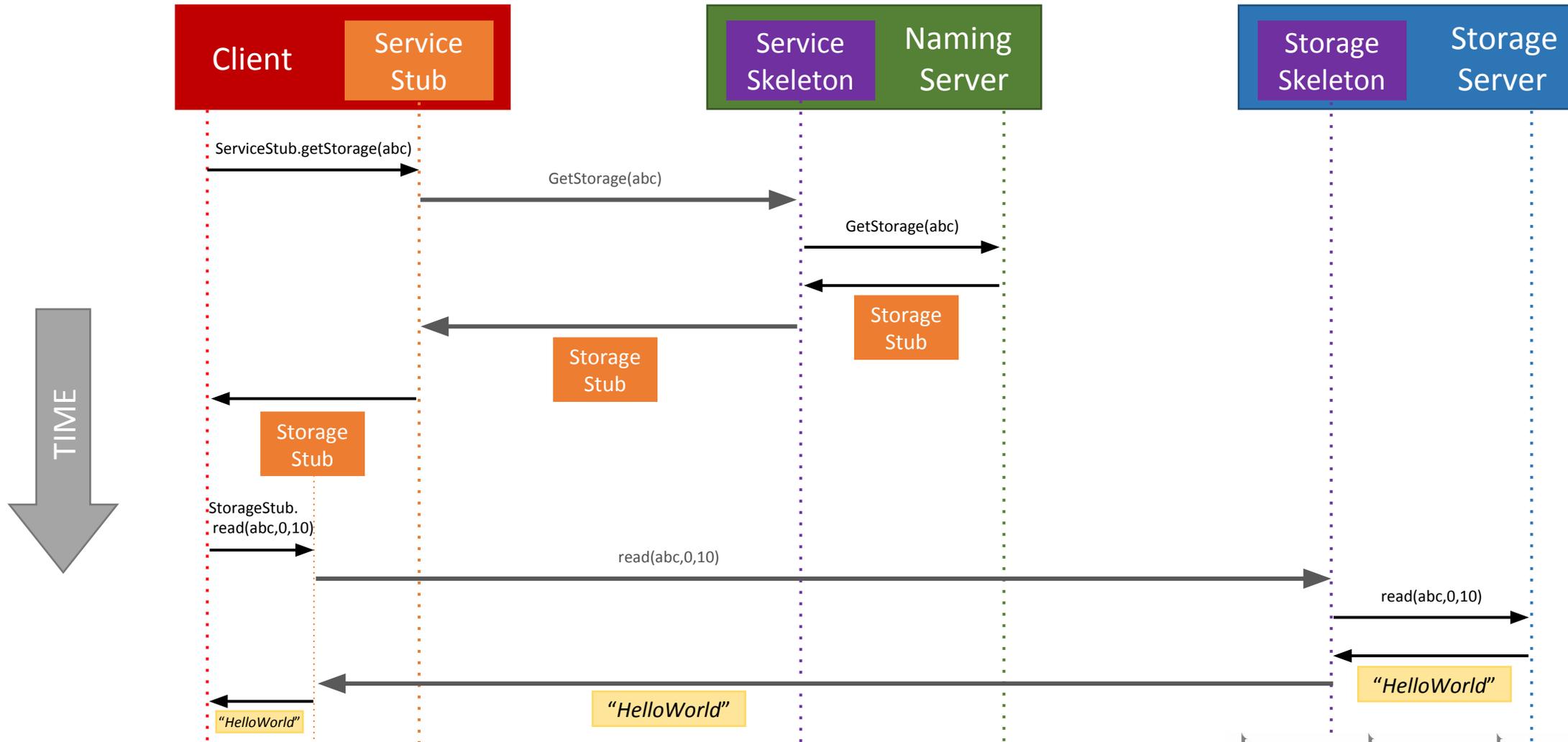


Figure 2: An example of a Client performing a read operation on file 'abc'.

# 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!**