15-440
Distributed Systems

Recitation 4

Zeinab Khalifa
Last Time

• Entities, Architecture and Communication
• RMI
• Interfaces
• Skeleton & Stub
• Example

Today

• Packages dive-in:
  ✓ RMI
  ✓ Common
  ✓ Naming
  ✓ Storage
Quick Recap
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.
Full Example: Client Read

Client → Service Stub

Service Stub.getStorage(abc) → getStorage(abc) → Storage Stub

Storage Stub.read(abc,0,10) → read(abc,0,10) → "HelloWorld"

"HelloWorld" → "HelloWorld"

Server

Service Skeleton

Naming Server

Storage Skeleton

Storage Server

read(abc,0,10) → "HelloWorld"
RMI package
(overview)
RMI package

- It contains two parametrized (generic-type) classes:
  1. Skeleton.java
  2. Stub.java
- RMIException
- Both the Skeleton and the Stub classes take a remote interface as a parameter.
**RMI package**

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

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);
}
Stub.java

• A stub is implemented in Java as a dynamic proxy
• A proxy has an associated invocation handler
• The invoke method checks whether the invoked method is or remote
• If the method is remote, the proxy connects to the corresponding skeleton at the server side, marshalls the method name, parameter types and values, and sends the entailed byte stream.

• http://tutorials.jenkov.com/java-reflection/dynamic-proxies.html
T proxy/stub = java.lang.reflect.Proxy.newProxyInstance(c.getClassLoader(), new Class[] {c*}, new ProxyHandler());
public class ProxyHandler implements InvocationHandler {
    public Object invoke (String methodname, Class[] argTypes, Object[] args) {
        if method is local // can be toString, equals, hashCode
            call locally implemented method accordingly
        } else {
            - create socket
            - connect (address)
            - out.writeObject(methodName);
            - out.writeObject(argTypes);
            - out.writeObject(args);
            - Object result = in.readObject();
            - close socket
            - return result
    }
}
How to implement the RMI package?

Phase (1)

- Java socket APIs (PS1)
- ObjectOutputStream & ObjectInputStream: which allow writing and reading primitive data types of Java objects (or referred interchangeably to as serializing data) to output and input streams (PS1)
- Dynamix proxies
- Exceptions
- Clean-up sockets and streams
- At the end of this phase, a stub should be able to connect to a skeleton, and send and receive objects. Similarly, a skeleton must be able to accept several client connections, read objects and send back objects (including exceptions).

Phase (2)

- Stub:
  1. Check if the method is local or remote.
  2. If local, it invokes a corresponding locally implemented method
  3. If remote, the stub sends the method name and parameter types and values to the respective skeleton.
- Skeleton:
  1. Receives the stub’s request
  2. Invokes the corresponding method
  3. Send back the generated result

* Exceptions that arise due to unsupported methods during unmarshalling, and the ones that are thrown by the implementor methods should be communicated back to the client.
RMI package
(Example: File Server)
Creating a file server:

1. Defining a remote interface
2. Defining a server class
3. Creating the server object and making it remotely-accessible
4. Accessing a server object remotely
Creating a file server:

1. **Defining a remote interface**
   ```java
   public interface Server {
       public long size(String path) throws ..;
       public byte[] retrieve(String path) throws ..;
   }
   ```

2. Defining a server class
3. Creating the server object and making it remotely-accessible
4. Accessing a server object remotely
Creating a file server:

1. Defining a remote interface
2. **Defining a server class**
3. Creating the server object and making it remotely-accessible
4. Accessing a server object remotely

```java
public class ServerImplementation implements Server {
    // Fields and methods. ...
    public long size(String path) throws ..{
        //size method impl.
    }
    public byte[] retrieve(String path) throws ..{
        //retrieve method impl.
    }
    ...
}
```
Creating a file server:

1. Defining a remote interface

2. Defining a server class

3. Creating the server object and making it remotely-accessible

4. Accessing a server object remotely

// Create the server object.
ServerImplementation server = new ServerImplementation(...);

// At this point, the server object is a regular local object, and is not accessible remotely.

// Create the skeleton object.
Skeleton skeleton = new Skeleton(Server.class, server);

// Start the skeleton, making the server object remotely-accessible.
skeleton.start();
Creating a file server:

1. Defining a remote interface
2. Defining a server class
3. Creating the server object and making it remotely-accessible
4. **Accessing a server object remotely**

   // Create a stub which will forward method calls to the remote object.
   InetSocketAddress address = new InetSocketAddress(hostname, port);
   Server server = Stub.create(Server.class, address);

   // Perform some method calls using the stub.
   long file_size = server.size("/file"); ... byte[] data = server.retrieve("/file");
Common package
Path package

This 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 package

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

• The naming package contains:
  1. Registration interface
  2. Service interface
  3. NamingServer class: creates the necessary skeletons and stubs and implements the logic of all the operations handled by the Naming Server
The naming package contains:

1. Registration interface
2. Service interface
3. NamingServer class: creates the necessary skeletons and stubs and implements the logic of all the operations handled by the Naming Server
The Naming Server 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

The Naming Server builds its tree during registration.

After registration, the Naming Server uses its tree to handle operations.

It is important to design the directory tree in a way that allows the Naming Server to easily look-up, traverse and alter the tree, as well as detect invalid paths.
Naming package *(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** will represent:
      • A list of **Leafs/Branches**
Naming package (Classes)

```java
public class Node {
    String name;
}

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

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

NamingStubs.java
(public class)
  • Creates:
    • Registration Stub
    • Service Stub
Storage package
Storage package

These stubs are sent to the Naming server during registration.
Storage package

The Storage Package:

- Command.java (interface)
- Storage.java (interface)
- StorageServer.java (public class)
  - Implements:
    - Command Interface
      - methods(s): create, delete
    - Storage Interface
      - methods(s): size, read, write
  - Has functions:
    - start()
    - stop()
Storage package

• The StorageServer start() function will:
  • **Start** the Skeletons:
    • Command Skeleton
    • Storage Skeleton
  • **Create the stubs**
    • Command Stub
    • Storage Stub
Storage package

• The StorageServer start() function will:
  • 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
Storage package

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