P1 Design document

El-Hassan Wanas
ewanas@qatar.cmu.edu

September 14, 2011

1 Starter code overview

This section describes the starter code structure and the way I plan to approach this project.

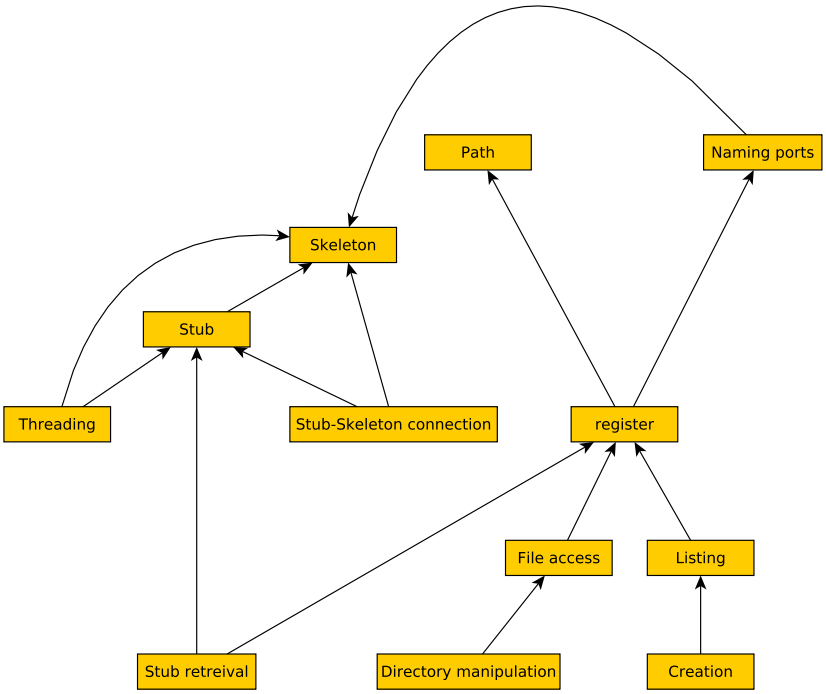

Diagram 1: Module dependancy

1.1 RMI

1.1.1 Skeleton

A `Skeleton` serves implementations of interfaces to `Stub` factories. The main point behind the `Skeleton` is to allow different implementations of the same interface to reside on multiple machines and not having to change the client code to be able to use those implementations.

In File Stack, we will be running two skeletons. One on the naming server and one on the storage server. The skeleton on the naming server will serve implementations of the File Stack interface to clients. The skeleton on the storage server will serve implementations of the `Storage` interface to the naming server.

1.1.2 Stub

A `Stub` is really an interface that isn't yet implemented, but which you request an implementation for. The convenient Stub factory should hide all the details of obtaining the implementation.

1.2 Naming

The `naming server`'s mission is twofold. For one, it should accept `storage server` requests and requests from clients. It keeps track of which files belong to which servers and it forwards `storage server` stub implementations to clients, upon invoking an operation on a file.

When a storage server connects to the naming server, the naming server should receive an implementation of the Storage interface.

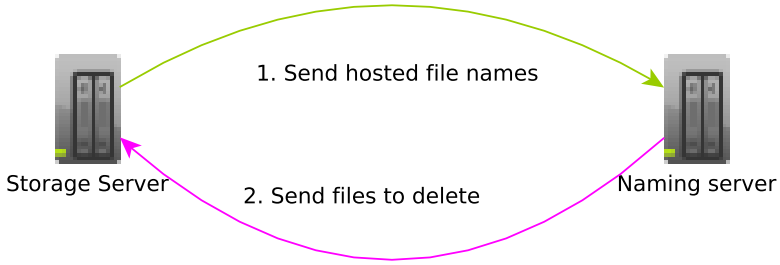

Diagram 2: Registration

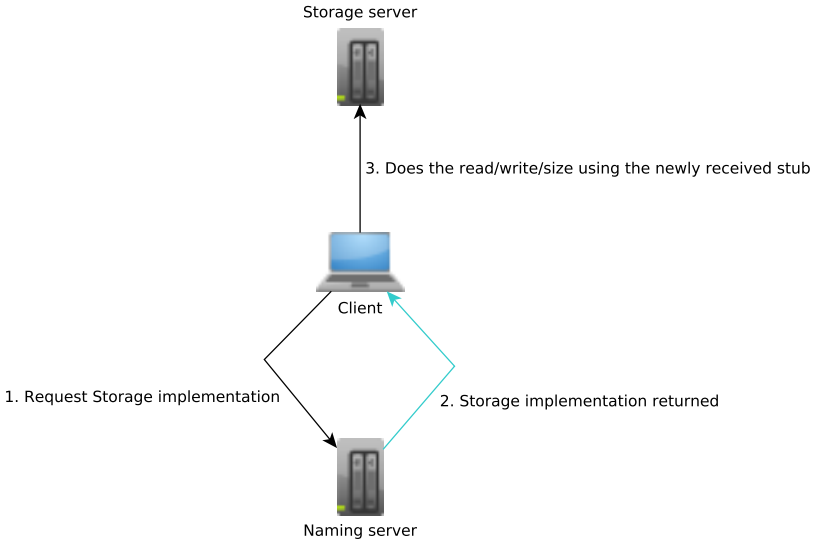

Diagram 3: Client read/write/size implementation

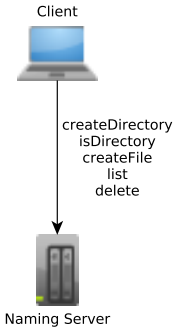

Diagram 4: Client meta file operations

1.3 Storage

Storage servers connect to naming servers, to bring forth their share of the file system to view. They will get connections from clients beyond that. When the client wants to read/write/size it should first ask the server for the actual implementation of `Storage` that will support the operation and hide the actual storage server. The communication after receiving that implementation happens between the client and the storage server only.

2 Logic of unimplemented functionalities

I would probably want to use ExecutorService to manage my threads, for this project. It has a very convenient way of managing a thread pool.

2.1 Skeleton

2.1.1 `public Skeleton(Class<T> c, T server)`

Listen on a random port for requests for implementations of the interface c.

2.1.2 `public Skeleton(Class<T> c, T server, InetSocketAddress address)`

Listen on a specific port for requests for implementations of the interface c.

2.1.3 `public synchronized void start() throws RMIException`

Starts the server on a seperate thread. For every new connection, a new thread is created.

2.1.4 `public synchronized void stop()`

Stop the main server thread.

2.2 Stub

2.2.1 `public static <T> T create(Class<T> c, Skeleton<T> skeleton) throws UnknownHostException`

Connect to the skeleton and requests an implementation of the interface designated by the type T.

2.2.2 `public static <T> T create(Class<T> c, Skeleton<T> skeleton, String hostname)`

Connect to the skeleton of type Skeleton T running at another host and request for an implementation of the interface designated by the type T.

2.2.3 `public static <T> T create(Class<T> c, InetSocketAddress address)`

Connect to the server at the given hostname and port and requests an implementation of the interface T.

2.3 NamingServer

2.3.1 `public NamingServer()`

Does nothing special, as far as I can tell at this point.

2.3.2 `public synchronized void start() throws RMIException`

Create a ServerSocket in a new thread and start listening for requests.

2.3.3 `public void stop()`

Terminate all threads created by the NamingServer.

2.3.4 `@Override public boolean isDirectory(Path path) throws FileNotFoundException`

Look up the permissions for the path in the Map¡Path,Permissions¿ that will be created within the NamingServer.

2.3.5 `@Override public String[] list(Path directory) throws FileNotFoundException`

Return all the keys in the Map¡Path,Permissions¿.

2.3.6 `@Override public boolean createFile(Path file) throws RMIException, FileNotFoundException`

Use one of the storage server implementations to create the file and add it to the global Map¡Path,Permissions¿.

2.3.7 `@Override public boolean createDirectory(Path directory) throws FileNotFoundException`

Use one of the storage server implementations to create the file and add it to the global Map¡Path,Permissions¿.

2.3.8 `@Override public boolean delete(Path path) throws FileNotFoundException`

Use one of the storage server implementations to delete the file and remove it from the global Map¡Path,Permissions¿.

2.3.9 `@Override public Storage getStorage(Path file) throws FileNotFoundException`

Return the value in the Map¡Path,Storage¿ for the key with value file.

2.3.10 `@Override public Path[] register(Storage client_stub, Command command_stub, Path[] files)`

2.4 StorageServer

2.4.1 `public StorageServer(File root)`

Initialize the StorageServer in such a way that the value of root becomes the prefix to every Path of a file on that server.

2.4.2 `public synchronized void start(String hostname, Registration naming_server) throws RMIException, UnknownHostException, FileNotFoundException`

Try to connect to the naming server running at `hostname` and register using the value for `naming_server`.

2.4.3 `public void stop()`

Terminates all the threads running on the StorageServer.

2.4.4 `@Override public synchronized long size(Path file) throws FileNotFoundException`

Return the size of the file at root + file.toString ().

2.4.5 `@Override public synchronized byte[] read(Path file, long offset, int length) throws FileNotFoundException, IOException`

Return the first `length` bytes after the `offset` of the file located at root + file.toString ().

2.4.6 `@Override public synchronized void write(Path file, long offset, byte[] data) throws FileNotFoundException, IOException`

Overwrites the file at root + file.toString () from offset with the contents of data.

2.4.7 `@Override public synchronized boolean create(Path file)`

Create a new file at root + file.toString ().

2.4.8 `@Override public synchronized boolean delete(Path path)`

This should lock access to the file at root + path.toString () and delete it.

2.5 Path

2.5.1 `public Path()`

```
{
}
```

2.5.2 `public Path(Path path, String component)`

```
    {
        if (component == null || !isValid (component)) {
            throw new IllegalArgumentException (
                    "Invalid path component"
                    );
        }

        pathPrefix      = path;
        pathComponent   = component;
    }
```

2.5.3 `public Path(String path)`

```
    {
        String [] components;

        if (path != null && path.indexOf ("/") == 0) {
            path = path.substring (1);

            pathPrefix = new Path ();

            components = path.split ("/");

            if (components.length >= 1) {
                pathComponent = components [components.length - 1];

                components = Arrays.copyOfRange (components, 0,
```

```
components.length - 1);

                for (String component : components) {
                    if (!component.equals ("")) {
                        pathPrefix = new Path (pathPrefix, component);
                    }
                }
            } else if (path.equals ("/")) {
                return;
            }

            return;
        }

        throw new IllegalArgumentException (
                "Invalid path string"
                );
    }
```

2.5.4 @Override public Iterator<String> iterator()

```
    {
        return new ComponentIterator (this);
    }

    /** The component String iterator.

        <p>
        This iterator will not support <code>remove()</code>. It will throw an
        <code>UnsupportedOperationException</code>, in such cases.
    */
    private class ComponentIterator implements Iterator <String>
    {
        Iterator <String> listIterator;

        /** Creates a new <code>ComponentIterator</code> that goes through
            a list of path components.

            @param path is a <code>Path</code> that this iterator will go
                        through the components of.
        */
        public ComponentIterator (Path path)
        {
            List <String> components = new ArrayList <String> ();

            for (Path p = path; p.pathPrefix != null; p = p.pathPrefix) {
                components.add (0, p.pathComponent);
            }

            listIterator = components.iterator ();
        }

        /** Checks if there are more components in the path.

            @return true if there are more components.
        */
```

```
        public boolean hasNext ()
        {
            return listIterator.hasNext ();
        }

        /** Returns the next component in the path.

            @throws NoSuchElementException if there are no more components
                    in the path.
            @return The next component in the path.
        */
        public String next ()
        {
            return listIterator.next ();
        }

        /** Method isn't implemented.

            @throw UnsupportedOperationException Because this method isn't
                                                 implemented.
        */
        public void remove ()
        {
            throw new UnsupportedOperationException (
                    "Can't remove component using path component iterator"
                    );
        }
    }
```

2.5.5 `public static Path[] list(File directory) throws FileNotFoundException`

```
    {
        List <Path> contents    = new ArrayList <Path> ();
        int         length      = directory.toString ().length ();

        if (!directory.exists ()) {
            throw new FileNotFoundException (
                    "File " + directory.getName () + " doesn't exist"
                    );
        } else if (!directory.isDirectory ()) {
            throw new IllegalArgumentException (
                    "Not a directory"
                    );
        } else {
            Queue <File> files = new LinkedList <File> ();
            files.add (directory);

            while (!files.isEmpty ()) {
                File currentFile = files.poll ();

                if (currentFile.isDirectory ()) {
                    for (File file : currentFile.listFiles ()) {
                        files.add (file);
                    }
                } else {
                    String relativePath =
```

```
                    currentFile.toString ().substring (length);

                contents.add (new Path (relativePath));
            }
        }
    }

    return contents.toArray (new Path [0]);
}
```

2.5.6 public boolean isRoot()

```
{
    return pathPrefix == null && pathComponent == "";
}
```

2.5.7 public Path parent()

```
{
    if (isRoot ()) {
        throw new IllegalArgumentException (
                "Root directory doesn't have a parent"
                );
    }

    return pathPrefix;
}
```

2.5.8 public String last()

```
{
    if (isRoot ()) {
        throw new IllegalArgumentException (
                "Root directory has no other components"
                );
    }

    return pathComponent;
}
```

2.5.9 public boolean isSubpath(Path other)

```
{
    return toString ().indexOf (other.toString ()) == 0;
}
```

2.5.10 public File toFile(File root)

Create a new File object and initialize it with the Path's string.

2.5.11 @Override public boolean equals(Object other)

```
{
    return toString ().equals (other.toString ());
}
```

2.5.12 @Override public int hashCode()

```
    {
        // TODO make this sensible.
        return 10;
    }
```

2.5.13 @Override public String toString()

```
    {
        String pathStr = "/";

        if (pathPrefix != null) {
            if (pathPrefix.isRoot ()) {
                pathStr = "/" + pathComponent;
            } else {
                pathStr = pathPrefix.toString () + "/" + pathComponent;
            }
        }

        return pathStr;
    }
```