Networking and Remote Services with RU-Store: A Remote Object Store

Introduction

Distributed systems and services are a great way to expand the amount of computation power and storage capacity used by an application beyond that of a single system. However, before we can explore more advanced distributed systems concepts and complexities, we should first refresh and understand the complexity of building a simple service that can be used in a distributed setting.

In this project you will be exploring the complexities of client-server communication and refreshing your network programming skills by implementing your own remote object store from the ground up. By the end of this project you would have implemented your own server and client library, designed your own communication protocol to transfer information between the two, as well as learned a thing or two about modern storage services.

1 Background: Object Stores

Modern social media services and applications have become more data driven, generating large amounts of data not only for media hosting but for analytics. It has been harder to find proper storage solutions to store arbitrary, unstructured data. File systems can be used to store such data within a file system tree, however, maintaining a proper file system tree becomes burdensome and the amount of file metadata (attributes that describe the contents) that can be stored for each file is limited by the file system.

To allow for better scaling of storing unstructured data, object-based stores have been developed. In this case any kind of data be stored as an object, from a string, to pictures, to videos. Within an object store, objects are stored in a flat namespace using a unique identifier or key. This removes the need to organize and maintain some data hierarchy. Along with each piece of data, a piece of metadata can also be attached for each object to allow for an arbitrary amount of metadata to be stored. This allows better flexibility in terms of storing contextual information about the file compared to other forms of storage.

Because of these benefits, object storage has become more popular with cloud services like Amazon S3, Google Cloud, and Microsoft Azure providing remote object storage to allow for clients (such as social media services) to store large amounts of data and objects in the cloud. Other popular storage solutions such Ceph, allows for clients to scale object storage on their own infrastructure and is used in private and high performance computing environments.

You can read more about object stores here: https://en.wikipedia.org/wiki/Object_storage

Figure 1: Popular Cloud-based Object Storage Solutions

2 Architecture and Design

In order to implement your own object store, there will be two things you will have to implement: (1) the RUStore client library and (2) the RUStore Server. The client library is the interface the user application will use to interact and pass information between the application and the object store server.

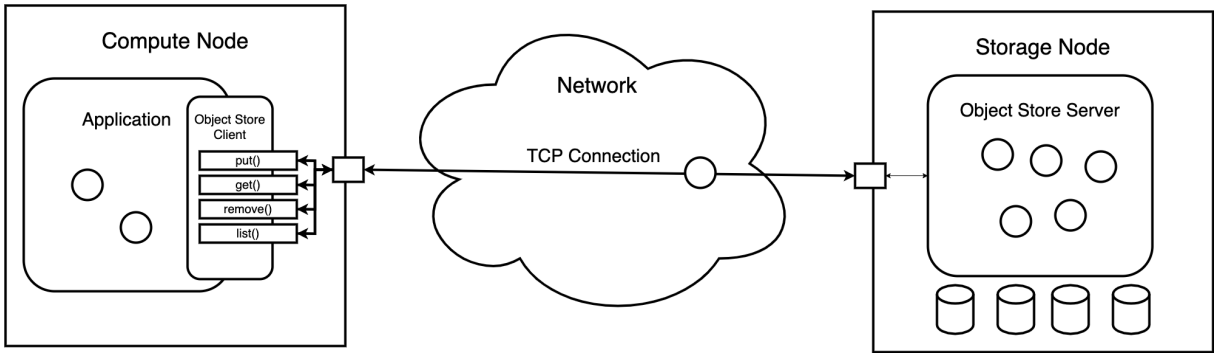

Figure 2: Object Store Architecture

RUStore Client

In order to interact with a remote RUStore Server, an application must create an RUStore Client. Using the RUStore Client, the application is then able to connect to the remote RUStore Server. The application can then use the client methods perform object store operations.

Within the client class *RUStoreClient.java* you will have to implement the following methods:

- **RUStoreClient(String host, int port)**
  Constructor that initializes client object to be used to communicate with the remote object server.
- **void connect()**
  Creates a socket and opens up a connection to the remote object server.
- **int put(String key, byte[] data)**
  Sends a data object represented by an array of bytes to the object server which will store the data with a given key. This is useful for offloading data stored within memory such as computational results or application objects that you don't want to persist locally.
- **int put(String key, String file_path)**
  Sends data from a file to the object server which will store the data with a given key. This is useful for offloading and distributing large binary files stored locally such as videos, images, etc. *Note how this is similar to put(String key, byte[]), except that the bytes are coming from a file instead.*
- **byte[] get(String key)**
  Downloads an object associated with a given key and returns the data as a byte array. This is useful for downloading data objects that you don't necessarily want to persist. For example you may want to download some results of data to perform quick processing and computation on.
- **int get(String key, String file_path)**
  Downloads object associated with a given key and writes it to a specified file. This is useful for downloading large data objects that may be too unwieldy to keep around in memory such as videos or compressed files or any data that you want to process at a later time. *Note how this is similar to get(String key), except that bytes are should be directed to a file instead of a byte array in memory.*
- **int remove(String key)**
  Removes an object from the object server. Note that this doesn't download the object, it simply tells the object store to free the object on the server side.

- **String[] list()**
  Fetches a list of all object keys from the object store and returns them as a list of String objects.
- **void disconnect()**
  Sends a message to the server to inform the connection will close so the server socket can be closed gracefully. After sending the server a message, it closes the client socket and connection.

Note: To see more details such as what kind of situations you should handle and what are the valid return values for each method, please take a look at the Java documentation within *RUStoreClient.java*.

RUStore Server

The RUStore Server a runnable java program that will need to be implemented in the server class *RUStoreServer.java*. The RUStoreServer class is empty besides the main() method. The main method accepts a single argument which is the port number the server should run on. You will have to add any necessary classes, add any static members, as well as any helper methods to help implement the server program.

3 Project Stages

To implement our Remote Object Store, we break down this project into four stages designed to help with incremental implementation:

- **Stage 1** – Building a Connection
- **Stage 2** – Implementing a Communication Protocol
- **Stage 3** – Passing Arbitrary Objects and Files
- **Stage 4** – Object Storage

4 Stage 1: Building a Connection

Goal: Build a simple connection between a client and server and pass basic information between the two.

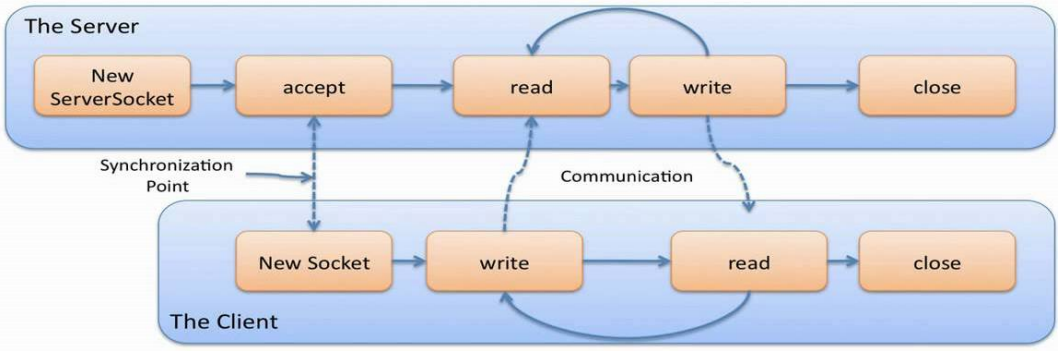

Figure 3: Basic Socket Communication Flow

Overview:

Before we can start passing things, we will need to start off by making a connection and start to learn how to pass information back and forth between the client and server. This will require to do the following things:

1. On the server, open a socket and bind it to a port to start listening for incoming connections.

2. On the client, open a socket and establish a connection to the remote server.
3. Use read() and write() to send and receive data between the client and server.
4. Close the connection and sockets on both the client and server when messages are done being passed.

Approach:

In order to implement this within you client and server you will need know how to use Java Sockets and Input/Output Streams. Before starting your implementation within the RUStoreClient and RUStoreServer classes, first build simple client and server programs outside to the project by looking at the following references and guides:

- **Network Programming Slides.** See lecture slides on Canvas
- **A Guide to Java Sockets.** https://www.baeldung.com/a-guide-to-java-sockets

One you get the hang of simple of how to use sockets and streams, you can start to implementing the client and server logic within the following places:

1. **RUStoreClient.java**
    - RUStoreClient() constructor
    - int connect()
    - int disconnect()
2. **RUStoreServer.java**
    - void main()

Testing:

Testing your implementation with the client and server requires both to be running. Running the RUStoreServer should be easy and straight forward. However, running the RUStoreClient is not as it's simply a client library that is used by applications.

To test if the RUStoreClient works with the basic client logic you implemented so far, use the Sandbox program (*TestSandbox.java*) to create a RUStoreClient object and call the connect() and disconnect() methods. To see more on how to build and run your programs including the sandbox program, take a look at Section 9 and 10.

Other Notes:

Handling Exceptions. As you start to fill in the client methods, you'll notice that certain libraries/-packages you use can potentially throw exceptions and cause an issue. For example, when using the java.net package, if you try and open a connection and the host is not valid, an *UnknownHostException* would be thrown. In this case you should throw the exception from the client method so the calling application can handle it accordingly like:

void connect() **throws UnknownHostException**{...}

Do avoid throwing the generic Exception to handle all exceptions like:

void connect() **throws Exception**{...}

Allowing a method to throw a generic exception is not good practice as any application utilizing the library will be unaware of what kind of exception it is (ex. i/o exception, unknown host exception, etc...) and will have to look into the exception details in order to figure out what to do next. It's best to list out all the specific types of exceptions a method can throw to let users of the methods know what kind of errors they should expect to possibly handle.

5 Stage 2: Implementing the Communication Protocol

Goal: Design and implement your own communication protocol to allow for our client and server to effectively communicate and understand data being sent back and forth.

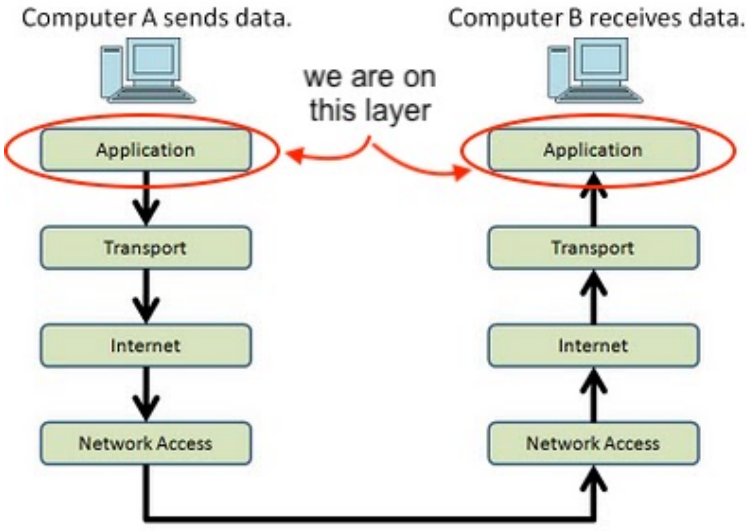

Figure 4: Data flow through the TCP/IP stack between 2 computers.
Red shows the layer our protocol will reside in.

Overview:

Need for a higher level protocol

In the previous stage, you used Java sockets to open a TCP connection. The TCP protocol exists on the transport layer and provides enough capability to reliably transfer data over the network. However at this point, how does an application know to interpret the data being received? Further, how does it know what kind of data to send back and when?

This is where the application layer comes in. Protocols existing on the application layer utilize lower level protocols (like TCP) to send data over the network while helping applications effectively exchange data by defining how the endpoints should communicate and what kind of data they should expect to send and receive. This is where we currently stand in our project.

Why need your own protocol?

We know at this point we need a higher level protocol to allow for our client and server to communicate effectively. We could perhaps use existing communication protocols such as HTTP. However, despite being widely used, there would be extra bulk when using it, since it has more functionality than we actually need. In light of low networking latencies and data generation increasing, it would be better for us to come up with our own simple and lightweight protocol to maximize performance and data throughput.

Towards designing our own protocol

When designing your communication protocol, you will need to consider a couple things:

- **Message Types.** Remember, for each of the client methods you will want the server to perform different actions, so you must format the messages being sent from the client in such as way where the server can understand what kind of data is in the message and what kind of action it should take.
- **Special Response Messages.** After sending a request message, the client should wait for a response from the server to ensure the server received and processed its message. Should a response message use the same format as the request messages coming from the server?

- **Variable Sized Messages.** You can assume keys when serialized won't be more than 1KB or 1024 bytes, however remember that data objects being sent over the network can be of any size. How can you account for this in your protocol?

Approach:

If we start trying to take bytes representing different parts of our message, jam them together and start sending them it over the network hoping it will just work, chances are that it won't. Instead you should work to incrementally build your message protocol.

1. **Start with a simple message.** For example, start on incrementally building requests (messages from the client to the server). You can start off by sending a simple message encoding what kind of request it is (e.g put, get, ..). Then, on the server side, make sure you can properly receive the request and decipher what kind of request it is, printing out "put", "get", etc. depending on the message received.

2. **Incrementally Build Add More To The Message.** After you do that, simply append more information to the end of the message. Try to append a key to the message and verify on the server side that you can properly decipher the longer message by not only printing out the request type, but also the key.

3. **Figure out how to add arbitrary amounts of data** Once you get the hang of generating the messages and deciphering it on the other end, now its just a matter or designing the message in such a way to attach an arbitrary amount of data to it.

4. **Repeat the same approach for other kinds of messages** After you finish designing and implementing client request messages, you can now use the same approach to design and implement server responses.

After figuring out the design and implementing message building, you can start populating and testing all the client methods, formatting messages with the required information for the operation (ignoring the object data), sending the message, and waiting for a response. You can also ensure that your server implementation is structured to handle every type of message and sends a response back to the client for any message it receives.

Other Hints:

Encoding Operations: You can perhaps encode an integer or string at the beginning of the message to encode what kind of operation you want to be done.

Padded Keys: You can assume keys, when serialized to bytes, won't be larger than a 1Kb or 1024 bytes. With this assumption, to make things easier, instead of having to worry about variable sized keys, simply place the key within a set range of 1024 bytes within the message. **However remember, if you take this approach, remember to properly pad the key within the 1024 bytes and zero out all the other bytes in order for the string to be properly parsed on the recieving end.**

Scalable Protocol Development: To keep the Client and Server implementation clean and keep things modular and easy to develop, it could be helpful to implement you any methods and classes that support your communication protocol in a separate java files.

6 Stage 3: Passing Arbitrary Data Objects and Files

Goal: Verify that you can properly transport data objects between the client and object server.

Overview

Now that you have some sort of protocol laying out the foundation for messages being passed between the client and server, you will need to verify that you can properly attach an object to the message in order to send the object back and forth over the network.

Approach

First you should be able to ensure you can pass any amount of arbitrary data, you should start off by passing data you can verify before moving onto things.

1. **Start off with simple Strings.** Add to the put() method that accepts an array of bytes to actually take the byte array and place it the client message to the server. Now you can start off testing whether the functionality of sending arbitrary data works by passing the bytes of an arbitrary string to the client put() method, like the following:

   ```
   put("key1", "Hello world".getBytes());
   ```

   Make note of how many bytes the String was.

2. **Verify arbitrary data on the server.** On the server side, see if you can extract the variable length string data and verify that you read the correct amount of bytes and that the data correct data by turning the bytes back into a String.

3. **Move onto passing files.** Implement the put() method that accepts a file as an argument. Once you implement it, now try to call the put method but passing in some file like:

   ```
   put("key1", "./inputfiles/chapter1.txt");
   ```

   Make note of how many bytes the file was.

4. **Verify file data on the server side.** On the server side, see if you can properly read all the file data bytes. You can verify if you received the file correctly by saving the data to a file and trying to open it.

5. **Verify the other way around.** At this point, you can properly send any arbitrary data to the server. However remember, the server needs to pass arbitrary data back to the client to support the get() operation. Use the same approach you used to verify that data can get the server properly to verify that the data can get back to client properly. At this point you should now you have enough information to finish implementing the get() methods.

6. **Moving on to the rest.** Now that you now the main put() and get() can properly send data, its a matter of starting to implement the rest of the methods such as list() and remove().

Avoid deserialization on the server: Only deserialize to test message passing such as when you verified you can successfully pass arbitrary strings back and forth. After you should remember to remove that part of your code. Remember, the object store will be used to store arbitrary objects. The server has no notion of what kind and exact format of the object it is trying to store.

7 Stage 4: Object Storage

Goal: Complete the implementation by adding data structures and classes that will allow for you to store and retrieve data objects using keys.

Overview:

Now that you have the basic essentials needed for passing objects to and from the server, now you need to figure out a way to store the objects to easily store and retrieve them. This stage is open-ended. You can use any type of data structure to store these objects. Remember you will need to store and retrieve objects using their unique identifier (key).

Once you figure out what kind of data structure you want to use, its just a matter of extending the service logic to actually store, retrieve, or remove the object within the data structures to support put(), get() and remove() messages from the client. You should also be able to easily retrieve a list of all object keys to support the list() operations.

Other Notes:

Store in Memory: Traditionally in a typical object store, since memory is limited, some objects may have to be persisted to disk to make room for incoming objects. However to keep this project simple, you can store all objects within memory.

Built-in Data Structures and Performance: You are free to use built-in Java data structures but be sure to keep in mind the performance and memory overheads of such data structures. Remember we want to store data objects fast and retrieve them fast.

Arbitrary Objects: Remember the object store should be able to store arbitrary objects, so your data structure should be able to store arbitrary byte arrays to store object data.

8 Getting Started

The following sections describe how to get started on the project.

8.1 Requirements

In order to complete this project you will need the following:

1. **Java JDK 11**
2. RU-Store project template
3. Eclipse IDE (optional)

8.2 Downloading the Project Template

If you do not already have the project template zip provided, you can download the project template two other ways:

- Download directly from Github:
  https://github.com/DaveedDomingo/Object-Store-Project/archive/main.zip
- Clone the project template using GIT:
  $ git clone https://github.com/DaveedDomingo/Object-Store-Project.git

8.3 Importing the Project into Eclipse

To import the project into Eclipse, carry out the following steps:

1. In Eclipse Go to "File" > "Import"
2. Under the Maven folder, select “Existing Maven Projects”. Click Next
3. In “Root Directory”, browse to the RUStore project template folder. If done correctly, there will be an entry under project window called /pom.xml
4. Ensure /pom.xml is selected.
5. Click Finish.

Note: Eclipse may show errors within the project after importing into Eclipse, more specifically an error in the POM.xml file. To resolve this simply refresh the project by right clicking the project and seleting "refresh". New target folders should show up and the error should go away.

Eclipse is not mandatory: Although it is recommended, **it is not necessary to use the Eclipse IDE**. We are using Eclipse because it already comes prepackaged with Apache Maven. Maven is a tool that is used to help automate the building of a Java project. If you decide you do not want to use Eclipse you will still need Maven to build and compile the project. You can look up look up instructions on how to install the Maven tool on your particular operating system. We will mention how to use Maven in both Eclipse and Command Line in the following sections.

8.4 Project Template Overview

The following is the bare structure of the project template:

```
project-template
├── src
│   └── main
│       └── java
│           └── com
│               └── RUStore
│                   ├── RUStoreClient.java
│                   ├── RUStoreServer.java
│                   ├── TestSample.java
│                   ├── TestSandbox.java
│                   └── TestStringCLI.java
├── inputfiles
│   ├── chapter1.txt
│   ├── dummysite.html
│   ├── knight.jpg
│   ├── lofi.mp3
│   └── solutions.pdf
├── outputfiles
│   └── ...
└── pom.xml
```

Notice: If you imported the project into Eclipse, additional folders may be present. This is due to Eclipse realizing that it is a Maven project, auto generating extra folders that it may use.

Main Implementation Files

Within the /src/main/java/com/RUStore directory the two main RUStore files:

- ***RUStoreLibrary.java*** - This is where you will implement the library that will be used by applications.
- ***RUStoreServer.java*** - This is where you will implement the server object store service that will run on the server

Test Files

You'll also notice within the /src/main/java/com/RUStore directory there are three Test files:

- ***TestSample.java*** - This is a sample test file that will use the RUStore client library and create and try to store and retrieve various objects.
- ***TestSandbox.java*** - This is an empty file that you can use to implement your own test application that will use the RUStore client library.
- ***TestStringCLI.java*** - This is a simple interactive program that will allow you to send test and send text to your object server as String objects.

Note: It is important to note that these sample test files are not exhaustive tests. You will have to make sure you figure out how to thoroughly test all cases on your own. The implemented *TestSample.java* and *TestStringCLI.java* are only there for convenience and to help you get going with testing and understanding how the client library should be used.

Test Input Files

There is a folder called inputfiles at the root of the project with varying types of files that you can use to test if you can properly store and retrieve files as objects:

1. ***chapter1.txt*** - plain text file holding the beginning paragraphs of a good book
2. ***dummysite.html*** - basic dummy html file showing a poor attempt at making a website
3. ***knight.jpg*** - photo of your favorite mascot
4. ***lofi.mp3*** - 1 minute of royalty-free lofi music
5. ***solutions.pdf*** - a pdf document of something you should familiarize yourself with

Test Output Directory

There is also a folder called outputfiles. This is an empty folder where you can direct data objects downloaded from the remote object store. This is simply here to act as a place to keep files separate from the original input files.

Pom Configuration File

Lastly, you will notice at the root of the project there is a file called ***pom.xml***. This file is for Maven. Like mentioned in the previous section notice, Maven is a build tool used for Java projects and uses the pom.xml file to know how to build the Java project as well as know what dependencies need to downloaded before hand. **DO NOT EDIT THIS FILE**. The pom.xml file has been preconfigured to generate Runnable JARs every time the project is compiled *for your convenience*.

9 Building the Project

To build your project, you will need to use Maven and run "goals". Goals are procedures that carry out actions related to the project lifecycle. There are two Maven goals you will need will to familiarize yourself with in order to build and compile your project:

- **package** - this goal compiles the project and packages artifacts (runnable jars) into the *target* folder
- **clean** - this goal cleans the project of any artifacts and code generated by the project

Package

The package goal will carry out the Maven build process: compiling project code, packaging artifacts, as well as carry out any additional tasks defined within the pom.xml file. You will need to run the package goal is to compile if you want to run your object service and applications as executable jars.

- To do this in Eclipse:
    1. In Eclipse, select the project folder within the Package Explorer window.
    2. Go to "Run" > "Run As" > "Maven build..."
    3. Within the Goals text box, type in "package"
    4. Click "Run"
- To do this in Command Line:
    1. Navigate to the root of the project directory.
    2. Run the following: "mvn package"

Clean

The clean goal will clean the project directory of any generated project artifacts and source code. Project artifacts and generated code are usually placed within a folder named "target". You will need to run the clean goal before you build your project with the package goal to ensure any files from your old code is gone. You can run the package goal without running the clean goal as it will just overwrite the existing files, but there may be some files that were not overwritten still lying around.

- To do this in Eclipse:
    1. In Eclipse, select the project folder within the Package Explorer window.
    2. Go to "Run" > "Run As" > "Maven clean"
- To do this in Command Line:
    1. Navigate to the root of the project directory.
    2. Run the following: "mvn clean"

10 Running Your Project

While you are implementing your service, you may want to test your server and client applications.

- To do this in Eclipse:
    1. In Eclipse, right-click the RUStoreServer.java within the Package Explorer window.
    2. Select "Run Configurations..." and under the "Arguments" tab, populate the "Program arguments:" text box with the port number you want the server to run on (ex. 12345)
    3. Run the server by selecting "Run As" > "Java Application"
    4. To run a Test Application, simply right click the Test Application you want to run and select "Run As" > "Java Application"
       (Remember to configure proper arguments if the test program requires arguments such as TestSample)
- To do this in Command Line:
    1. Navigate to the root of the project directory.
    2. Run the server by running: java -jar ./target/RUStoreServer.jar 12345
    3. Repeat the previous step but for the test jar file you want to run
       (ex. java -jar ./target/TestStringCLI.jar).

Sample execution

TestSample Program

The following is a sample of what the test application TestSample should look like when you run it. TestSample is sample program which tries to store a string and file and tries to retrieve them:

Command Line

```
$ java -jar ./target/TestSample.jar localhost 12345
Connecting to object server at localhost:12345...
Sucessfully established connection to object server.
Putting string "Hello World" with key "str_1"
Successfully put string and key!
Getting object with key "str_1"
Successfully got string: Hello World
Putting file "./inputfiles/lofi.mp3" with key "chapter1.txt"
Successfully put file!
Getting object with key "chapter1.txt"
File contents are equal! Successfully Retrieved File
...
```

TestStringCLI Program

The following is a sample of the test applications TestStringCLI should look like when you run it. TestStringCLI is an interactive program which simply calls appropriate client methods to store and retrieve arbitrary Strings you give it:

Command Line

```
$ java -jar ./target/TestStringCLI.jar
> connect localhost 12345
Connecting to server at localhost:12345...
Connection established.
> put "key1" "Hello World"
Putting string: "Hello World" with key "key1"...
Successfully put key1
> put "key2" "Foo Bar"
Putting string: "Foo Bar" with key "key2"...
Successfully put key2
> list
Going to get object keys...
Successfully retrieved keys
Object Keys: key1, key2
> get "key1"
Getting string with key "key1"...
Successfully received string.
Received string: "Hello World"
> remove "key1"
Removing object with key key1...
Successfully removed object with key key1
> list
Going to get object keys...
Successfully retrieved keys
Object Keys: key2
....
```

The following is a sample of what the server output could look like when you run the TestCLI test program:

Command Line

```
$ java -jar ./target/RUStoreServer.jar 12345
Server started. Listening on port 12345.
Waiting for client connection...
Connection established.
putting "key1"
putting "key2"
sending list of keys
getting "key1"
removing "key1"
sending list of keys
....
```

Play around and feel free to modify the available test programs. Again, remember these sample tests are not exhaustive, so understand them and try to expand them to fully test your implementations.

Server output doesn't matter. You will not have have to follow these output formats exactly. This is just an example. All that matters is if data is being passed correctly to the object server and back. You can have the input and output be as formatted as much as you would like, however it may be useful to keep it simple and intuitive if it comes to the case where we have to debug your code.

11 Submission

To submit your project, submit the following items:

1. **The following raw files:**
   - RUStoreClient.java
   - RUStoreServer.java
   - any additional java files you may you created that the client library and server depend on

   Note: Do not submit any of the test files. With these files I should be able to replace the source files within a new project template and successfully compile and run the code with no issues.
2. **A pdf document consisting of the following things:**
   - A few pages describing how you implemented your project. As you describe your implementation, be sure to answer the following questions (no particular order needed):
     - How does the client and server handle the different object store operations?
     - Describe the design and implementation of your communication protocol.
     - What kind of data structures did you use to store and retrieve your data objects.
   - A couple of paragraphs reflecting on your project experience, answering the following questions:
     - Do you think your implementation could be better? If so, explain what you think you could do to possibly make your project better. You can think about this in terms of how scalable your project is in terms of developing as well as the performance implications of your implementations of certain operations and data structures.
     - What was the hardest parts about implementing your project and how did you overcome them? If you had difficulties and couldn't finish the project, describe what you think you could've done better or what you think would've helped.

Please follow all submission guidelines. Failure to do so will result in points being taken away.

Don't slack on the doc. The write up is very important when it comes to evaluating your project. Not only does it help us understand your approach and how your implementation works, it also shows us the amount of effort you put into the project. The write up also helps us survey the amount of effort needed to complete the project as well as gauge the project's overall difficulty. Because of these things, the write up in general allows for as much partial credit as possible as well as reasonably scaling grades across all submissions.

12 Additional Tips and Tricks

When in doubt, Google (*responsibly*): If you have an issue or your program is throwing an error that you don't know how to fix, Google It. Someone, somewhere, probably faced the same issue at some point. However do so responsibly and don't cheat. If you are caught cheating, your case will be handled according to Rutgers' academic integrity policy (http://academicintegrity.rutgers.edu/).

Combining Maven Goals: You may find yourself running Maven goals frequently to build the project. You might find yourself running them so frequently it may just interfere with your precious development time. Luckily you can group goals together. For example, since it is good practice run the clean goal before the package goal, we should group them. In Eclipse this is done by running a Maven build like we are running the package goal, but instead of writing "package" in the Goals text box, we write "clean package". In Command Line, this is done by navigating the root of the project directory and running "mvn clean package". These will both run the clean goal first then the package goal.

Frequently Asked Questions

- **Can I use external libraries other than the built in java libraries in my implementations?** No.
- **Can I use another app-layer protocol such as HTTP in my project?** No. You have to build your own protocol.
- **Can I use helper methods in my implementations?** Yes, as long as the program works and the original client methods and their signatures are maintained.
- **Can I make other java files to support my implementation?** Yes you can. Just make sure you submit the supporting source files along with RUStoreClient.java and RUStoreServer.java
- **Can we implement the programs on my own computer?** Yes as long as it can compile and run on the iLab machines as that is where they will be compiled, ran, and graded.

13 Additional Questions

If you have any questions about the project or are having any issues, email me at David.Domingo@rutgers.edu