Computer Science 455/555
Identity Server (Phase I)
Programming Project 2 (150 points)

1 Description

In this assignment, we will implement a RMI based identity server. The basic idea is that client can connect to this server and submit a new login name request. The server checks its data store (or a database) and responds back either with a Universally Unique ID (UUID) if the new login id hasn't been taken by anyone before or returns back with an error. As part of the request to create a new login name, the client must submit a real user name as well. The server also permits reverse lookups, where a client submits a UUID and ask for the login name and other information associated with that UUID. The server periodically saves its state on to the disk so that it can survive crashes and shutdowns.

A data store is a repository for persistently storing collections of data. A database is a special type of data store that requires a schema. Other types of data store do not necessarily need a schema.

2 Motivation

In Linux/Unix systems, the user name is associated with an integer known as user id or `uid`. The integer `uid` is the only value that is actually stored in the file system. The mapping from user name to user id is stored in the file `/etc/passwd`. The nice thing about this is that we can change a user's name anytime without having to change the whole file system (as long as we don't change the associated user id number). This scheme breaks down across several systems since the same user may not have the same user ids on different systems. There are several solutions for maintaining this information in a centralized manner across multiple Linux/Unix machines: Kerberos, Network Information Services and OpenLDAP.

On the other hand, Microsoft Windows has the Active Directory services which plays the same role. Because LDAP-based authentication is supported on the most recent Microsoft systems and is also supported on Linux and other Unix systems, it makes an excellent choice for a cross-platform authentication system. Note that there are limitations to this. Firstly, the Microsoft clients for versions of Windows are specific to authenticating against a Microsoft Active Directory server. Although OpenLDAP uses the same LDAP protocol, there are other features of Active Directory (including a modified version of Kerberos with a Microsoft specific mechanism called a "PAC"), which means that Active Directory clients will not necessarily be able to authenticate against OpenLDAP.

The identity problem can be done for multiple platforms but it is not that simple. Our Java

based solution will take a simple approach that works across all platforms that Java runs on, which is virtually everything. Of course, we are implementing a very limited subset of the actual authentication problem as we want to use this a platform for experimenting with other concepts of distributed systems.

3 Specifications

3.1 Server

The following lists the salient features of the server.

- The server must be implemented using Remote Method Invocation (RMI) or equivalent.
- The server presents a suitable RMI interface for clients to access the id functions. The server must export remote methods that allow a client to create a new login name, to lookup a login name, to reverse lookup a UUID as well as to modify an existing login name.
- The server maintains a data store of login name to UUID mappings in memory. For each newly created login name, the server creates a UUID (use the `java.util.UUID` class for this purpose) and stores it in the mappings data store. For each login name, we also want to store the IP address from where the request was sent, the date and time when the request was received and the real user name associated with the login name. The server also stores the last change date for each id.
- Note that a separate thread dispatches every RMI call. The implication for RMI objects is that they must be thread safe, since the object may have to handle multiple requests simultaneously. The good thing about this is that your RMI server is already multi-threaded.
- Periodically, the server checkpoints the mapping data structure to the disk in a data store so that it can save all account data for next time it restarts (see the next item below on how we will accomplish storing to a data store). This makes the server more resistant to crashes and restarts. Also make sure if the server is shut down either normally or via a hangup or interrupt signal (e.g. via Ctrl-c), that it implements a shutdown hook to checkpoint before stopping.
- For simplicity, we can start with storing the accounts data structure by simply serializing it to a file. Then convert your server to use the redis data store.
- Redis (REmote DIctionar Server) is an open source, in-memory data structure store used as a data store, cache, message broker, and streaming engine. Redis provides data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, streams, and more. Redis has built-in replication, transactions, and different levels of on-disk persistence Redis is the most popular key-value NOSQL database and widely used in production distributed systems. See examples in `CS455-resources/examples/redis` to learn how to use redis.

- The server main class must be named `IdServer`. It should accept at least two optional command line options: `--numport <port#>` and `--verbose`. The `--verbose` option makes the server print detailed messages on the operations as it executes them. For the redis server, use a fixed port from the port range assigned to you on the class team page.

3.2 Client

The client program should be command-line based and only does one operation and quits (for ease in automating testing). It should provide for at least the following operations:

- The class containing the main method must be named *IdClient*. The arguments are:

  `java IdClient --server <serverhost> [--numport <port#>] <query>`

- It must support at least six types of command line queries as follows:

  `--create <loginname> [<real name>] [--password <password>]` With this option, the client contacts the server and attempts to create the new login name. The client optionally provides the real user name and password along with the request. In Java, we will pass the `user.name` property as the user's real name if the user does not specify one. Use
  `System.getProperty("user.name")` to obtain this information. If successful, the client prints an appropriate message with the generated UUID for that account. Otherwise it returns an appropriate error message.

  `--lookup <loginname>` With this option, the client connects with the server and looks up the `loginname` and displays all information found associated with the login name (except for the encrypted password).

  `--reverse-lookup <UUID>` With this option, the client connects with the server and looks up the `UUID` and displays all information found associated with the UUID (except for the encrypted password).

  `--modify <oldloginname> <newloginname> [--password <password>]` The client contacts the server and requests a loginname change. If the new login name is available, the server changes the name (note that the UUID does not ever change, once it has been assigned). If the new login name is taken, then the server returns an error.

  `--delete <loginname> [--password <password>]` The client contacts the server and requests to delete their loginname. The client must supply the correct password for this operation to succeed (if they had set a password for the account when it was created) .

  `--get users|uuids|all` The client contacts the server and obtains either a list all login names, list of all UUIDs or a list of user, UUID and string description all accounts (don't show encrypted passwords).

- The above options can be abbreviated as -s, -n, -c, -l, -r, -m, -d, -p (for password) and -g. Note that we can supply only one query at a time.
- You may use a command line parsing library to simplify your code. Some examples are the Apache Commons CLI (`https://commons.apache.org/proper/commons-cli/`) or args4j library found at `https://github.com/kohsuke/args4j`.

4 Enhanced Security

See the examples in the class repository `CS455-resources` under the folder security for help with this section.

4.1 Passwords

Add handling passwords to the server's functionality. So now, a client specifies a password when they create a new login name. To lookup information, no password is required. To modify the login name, the password is required.

The server should store the password encrypted in the account data store (both in memory and on disk). The simplest way to do this to have the client encode it using the SHA-2 algorithm. Please see `security/securehash/SHA2Test.java` in the class examples. This would be lead to more secure password handling since the server never sees the password in clear.

4.2 Encrypted Communications

This part requires that the previous part on encrypted passwords be completed first. Now we want to also encrypt all communications over the network. Use RMI over SSL (Secure Sockets Layer) to accomplish this part.

5 Hints on rmiregistry

You will need to run the `rmiregistry` in order to use RMI. By default, the rmiregistry service runs on port 1099. This isn't a problem at home. However, in the lab, if more than one student runs rmiregistry, the one started later will fail. You can run multiple rmiregistry daemons on the same system by giving them a different port number. For example:

```
rmiregistry 5113 &
```

starts it listening to port 5113. Now in your call to `Naming.bind()` or `Naming.rebind()`, you must specify the port number. Suppose the name of your server is IdServer, then you will rebind as follows.

```
IdServer server = new IdServer();
Naming.rebind("//localhost:5113/IdServer", server);
```

6 Testing

The goal of the project is to build a working prototype that allows us to explore distributed systems concepts. As such, we are not expected to develop extensive unit tests. The focus is on ensuring that basic functionality works, especially as an integrated system. We still want to develop a plan on how we will test the various features but not worry about all edge cases!

7 Documentation

- Use `javadoc` to document your code. Make sure to generate the javadocs and commit the repo to make it easier for me to read through your code.
- Please organize your code with packages for client and server. Provide appropriate scripts to start client and server.
- Create a MPEG video that demonstrates various features and scenarios for the chat server and clients. The video must have participation from all team members and must have an audio track (no silent movies, please!)
- Please include a README.md with at least the following elements:
  - Project number and title, team members, course number and title, semester and year.
  - A file/folder manifest to guide reading through the code.
  - A section on building and running the server/clients.
  - A section on how you tested it.
  - A section on observations/reflection on your development process and the roles of each team members.
  - Document any extra features that your team added that weren't required.

8 Required Files

- Manage your project using your team git repository. The project folder (named `p2`) has already been created for you.

- There must be a file named `README.md` with contents as described earlier in the Documentation section.
- The project must have a `Makefile` at the top-level (under the folder p2). You can use any build system underneath (like gradle, maven, ant etc) but the Makefile should trigger it to generate the project.
- All your source code, build files et al.
- The server and client keystores and certificates should be pre-generated. The required passwords can be hard-coded inside the source code (for ease in testing). Of course, we would not store passwords in the code in production!

9 Submitting the Project

You should be using Git throughtout the project with commits along the way. When you are eay to submit the project, open up a terminal or console and navigate to the team projects repository for the class. Navigate to the subdirectory that contains the files for the project.

- Clean up your directory and remove any auto-generated files such as .class files
  `make clean`
- Add and commit your changes to Git (this would be specific to your project)
  `git add README.md (other files and folders that needed to be added)`
- Commit your changes to Git
  `git commit -a -m "Finished project p2"`
- Create a new branch for you code
  `git branch p2_branch`
- Switch to working with this new branch
  `git checkout p2_branch`

  (You can do both steps in one command with `git checkout -b p2_branch`)
- Push your files to the GitHub server
  `git push origin p2_branch`
- Switch back to the master branch
  `git checkout master`
- Here is an example workflow for submitting your project. Replace p1 with the appropriate project name (if needed) in the example below.

```
[amit@localhost p1(master) ]$ git checkout -b p1_branch
Switched to a new branch 'p1_branch'

[amit@localhost p1(p1_branch) ]$ git push origin p1_branch
Total 0 (delta 0), reused 0 (delta 0)
To git@nullptr.boisestate.edu:amit
 * [new branch]      p1_branch -> p1_branch

[amit@localhost p1(p1_branch) ]$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

- Help! I want to submit my code again! No problem just checkout the branch and hack away!

```
[amit@localhost p1(master) ]$ git branch -r
  origin/HEAD -> origin/master
  origin/master
  origin/p1_branch

[amit@localhost p1(master) ]$ git checkout p1_branch
Branch p1_branch set up to track remote branch p1_branch from origin.
Switched to a new branch 'p1_branch'

[amit@localhost p1(p1_branch) ]$ touch newfile.txt
[amit@localhost p1(p1_branch) ]$ git add newfile.txt

[amit@localhost p1(p1_branch) ]$ git commit -am "Adding change to branch"
[p1_branch 1e32709] Adding change to branch
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 newfile.txt

[amit@localhost p1(p1_branch)↑ ]$ git push origin p1_branch
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 286 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@nullptr.boisestate.edu:amit
   36139d1..1e32709  p1_branch -> p1_branch

[amit@localhost amit(p1_branch) ]$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
[amit@localhost amit(master) ]$
```

- We highly recommend reading the section on branches from the Git book here: Git Branches in a Nutshell