Computer Science 455/555
Distributed Identity Server (Phase III)
Programming Project 4 (100 points)

1 Description

In the third phase, we will allow clients to contact any server to improve performance. There are various ways to go about it and you will have choice in how you design this part. Here are the issues.

- A client (or middleware, if you are running middleware locally) can contact any server to service its requests.
- The servers may still need to elect a leader amongst themselves. However, now the clients don't really need to know who the coordinator is. The coordinator can be used for several actions. For example:
  - The coordinator can be used to help with a distributed checkpoint.
  - The coordinator keeps track of distributed ownership of objects for writes.
- If the coordinator goes down or becomes inaccessible, then a new election is held and another server takes over the job of the coordinator. There should no loss of state in this transition.
- How does a client discover a new server? You can come up with any solution. However it should be possible in your solution for the client to discover the new server dynamically. For example, by listening on a multicast group. Another method would be to have a static list of servers that start off at the beginning. Then clients can walk through this list until they get a response back.

1.1 Testing Multiple Servers for Performance/Reliability

- What happens when a server goes down? When the client connects to another server how much up to date this other server will be? In other words, are you implementing a local-write protocol or something more stringent like atomic multicasting?
- What is the smallest unit of synchronization? Is it one account or a x number of accounts?
- What happens when a previously dead server comes back to life? There are multiple scenarios here as well.
  - The server keeps running but the network wire is removed/cut. How does it get back in sync with the other servers.
  - The server crashes and then restarts after a while.

2 Project Video (20 points)

Each team is required to create a MPEG video of five-ten minutes that shows the various features of the multiple servers/clients in action. The video must have participation from all team members and must have an audio track (no silent movies and no music, please!) See below for some ideas on what to show in a demo.

- Three servers running in three windows and two clients in two windows. Then show how the two clients connect to two different servers and can modify the accounts and both see the results.
- A server crashes after a client has used it for modifications. Show that the client is automatically able to switch to another server and that state information is not lost in the transition (of course, this is subject to some constraints due to your window of inconsistency!)
- A crashed server starts up again. Show that it is able to synchronize with the latest state from other servers.

3 Documentation (20 points)

- **Please clearly describe what changes you had to make from the Phase II server to this Phase III sever.**
- Please provide a clear description of the setup and testing in your README.md. Please also provide sample test/data files that I can use in exploring the capabilities of your project.
- For multiple servers, clearly explain and provide scripts that run multiple servers on localhost. Also provide separate setup and scripts for running on multiple nodes in the onyx lab.
- Use `javadoc` comments to document your code.
- Please organize your code with packages for client and server. Provide appropriate scripts to start client and server.
- Please include a README.md with at least the following elements:
  - Project number and title, team number and members, course number and title, semester and year.
  - A file/folder manifest to guide reading through the code.
  - Document clearly how to setup and test your project.
  - 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. Describe clearly how much of the functionality is available. If you did any extra stuff, please document that clearly as well. You can do all this in the README.md file or use a separate document.

4 Required Files

- Manage your project using your team git repository. The project folder (named p4) 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 p4). 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!

5 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 p4"`
- Create a new branch for you code
  `git branch p4_branch`
- Switch to working with this new branch
  `git checkout p4_branch`

  (You can do both steps in one command with `git checkout -b p4_branch`)
- Push your files to the GitHub server
  `git push origin p4_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