Computer Science 455/555
Distributed Identity Server (Phase II)
Programming Project 3 (200 points)

1 Description

In the second phase, we will add replicated servers for improving reliability.

1.1 Multiple Servers for Reliability

We will run multiple copies of the servers. The copies of servers are all potentially running on different machines. The main purpose of having multiple servers is to increase reliability. Multiple servers can also be used to increase performance but we will deal with that in the next phase. There are various ways to go about it and you will have choice in how you design this part. Here are the issues.

- At any time there is only one server that all clients are using.
- The servers need to elect a leader among themselves. The leader serves the clients. The leader also checkpoints the state to disk and with other servers.
- If a leader goes down or becomes inaccessible, then a new election is held and another server takes over the job of the leader. There should no loss of state in this transition (within the inconsistency window that we define)
- 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 leader dynamically. For example:
  - The clients have a static list of servers to start off at the beginning. Then clients can walk through this list until they get a response back.
  - The clients listen in on any election. So part of the state information exchanged between the servers would be a list of clients it has registered. So when an election is held, the servers know to copy the "I_AM_COORDINATOR" message to the clients as well. Of course, there are several issues here that would need to be resolved.

1.1.1 Testing Multiple Servers for Reliability

- How does a coordinator get elected when the system starts up? Kill the coordinator and show how a new coordinator is elected.
- What happens when the coordinator goes down? Who detects it (client or other servers). Who informs the clients about the results of the election?

- What happens when the previous coordinator 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.
- What happens if one of the backup server crashes or loses communication for a while?

2 Design White Paper

Write a 1-2 page white paper describing your design choices. It should address the following issues:

- What type of client server model you will use? One-shot client with separate servers. Or *servent* model where clients are servers as well and are always on.
- Coordinator: How to elect? How to detect if one is down or missing?
- Consistency model: Why did you choose a specific model? How will you implement it? What is your inconsistency window? Your inconsistency window should be reasonable and the instructor reserves the right to reject your consistency model and ask for an update.
- Server synchronization:
  - Lamport timestamps or Vector clocks
  - Granularity: do we copy the entire database when syncing or do we log messages and use checkpoints to reduce communication bandwidth required?

Please commit and push the white paper on your main branch for me to review. At the end of the project, when you put your whole project on the `p3_branch`, then the white paper would go along.

3 Testing setup

We can show the operation of multiple servers in one of the following three ways.

- Setup 1: Run multiple servers on separate onyx nodes on the lab.
- Setup 2: Run multiple docker instances on your machine with a server in each instance.
- Setup 3: Run multiple servers on separate instances on AWS

You must show that you can test your servers in Setup 1. Then you must demonstrate the operation of your servers in either Setup 2 or Setup 3.

If you are using multicast, then you are only required to demo it on the onyx nodes (as multicast will not work on localhost with docker and may be tricky to setup on AWS)

If you show your servers in both Setup 2 and Setup 3, you will get some extra credit (see the rubric). So your demonstration video should show the servers running on onyx nodes and then also on Docker and AWS.

4 Documentation

- You must document clearly how to setup and test your project. 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.
- Please provide a clear description of your testing. Please also provide sample test/data files that I can use in exploring the capabilities of your project.

5 Demonstration Video

- Create a demo video (five to ten minutes) with the following requirements:
  - For recording the video, you can use Zoom or Panopto (the university has site licenses for both via your Boise State account). Or you can record on your local machine and upload to Youtube or Google drive. Please do not add your video to Github as Github will reject large files. Instead, just add a link to your video to the README.md file.
  - Make sure to use larger fonts for viweability. Make sure to use light mode (instead of dark mode) to make it easier for me to see. Recall that Zoom/Panopto/Youtube will make the resolution less.
  - Makes sure to start with team# and the names of the team members.
  - The video must have participation from all team members and must have an audio track (no silent movies, please!).
  - The video shows various features and scenarios in action for the chat server and clients.
  - The video does a quick walk through the code.

6 Required Files

- Manage your project using your team git repository. The project folder has already been created for you.

- There must be a file named `README.md` with contents as described earlier in the Documentation section. It should be at the top-level of the project folder.
- The project must have a `Makefile` at the top-level. 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.

7 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 p3"`
- Create a new branch for you code
  `git branch p3_branch`
- Switch to working with this new branch
  `git checkout p3_branch`

  (You can do both steps in one command with `git checkout -b p3_branch`)
- Push your files to the GitHub server
  `git push origin p3_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