CS 4210/6210 Project 4 - GTStore

Due: Tuesday April 23 at 11:59 pm

1. Goals

In this project you will implement a distributed key-value store (GTStore). You should give special attention to system properties such as: (i) scalability, (ii) availability, (iii) resilience to temporary node failures. Implementation in C/C++. This is a group project (team of 2) with extra credit potentially.

2. System Components

1. **GTStore**
    a. **Centralized manager:** One user-space process will handle all the control path decisions of the key-value store. The manager's tasks include, but are not limited to:
        i. Membership management
        ii. Load balancing of clients across replica nodes
        iii. Index management (e.g., you are allowed (but not required) to allow clients first contact centralized manager on initialization to figure out the data nodes that they should send their requests to)
    b. **Storage/Data nodes:** N user-space processes will act as the storage nodes in the key-value store. They will store in-memory and provide the key-value pairs, that they are responsible for. These processes represent the actual key-value store.
2. **Driver application:** M user-space processes (or threads) that interact with GTStore. In a typical interaction, the application (i) reads the latest version of the object, (ii) mutates it, and (iii) stores it back. For example, the application can mimic a shopping cart service. The stored object (value) can be an array of shopping cart items. E.g. {phone, phone_case, airpods} and the key can be the client-id. One application process should manipulate a single key-value pair at a time, e.g. a single shopping cart.
3. **Client library:** The driver applications should use a simple programming interface, in order to interact with the key-value store.
    The base API includes;
    a. init() - initialize the client session with the manager and other control path operations.
    b. put(key, value)
    c. get(key)
    d. finalize() - control path cleanup operations
    You can assume that Max key size = 20 bytes, Max value size = 1 KB.

3. Design Principles

1. **Data Partitioning**

   GTStore aims to be a highly scalable key value store, thus storing all the data in a single node will not work. Ideally, the system should be able to serve increasing number of client requests in proportion to hardware resources we provision (number of data nodes). Data partitioning schemes divide the responsibility of managing data elements across the data nodes. For this section you will design and implement;

   a. A data partitioning scheme to partition incoming data across data nodes.
   b. Data insert/lookup algorithm/protocol for the selected partitioning scheme in the context of your system design.
   c. Discuss pros and cons of your design.

2. **Data Replication**

   GTStore maintains K number of data replicas for any stored object. Replication increases fault tolerance of the system and under certain consistency schemes increase the system availability. In this section you will design and implement data replication mechanism for GTStore.

   a. Clearly describe the replication protocol/set of events during data insert/lookup.
   b. How does your replication mechanism works alongside with the proposed data partitioning technique? (e.g. node with the primary replica fails).

3. **Data Consistency**

   Data consistency scheme plays a major role in defining the availability of any data store. Strong consistency schemes will not benefit GTStore design as it may makes the system unavailable during replica failures. Therefore GTStore implements eventual consistency semantics. In this section you will design and implement:

   a. Versioning scheme for the stored objects.
   b. Object puts/gets are executed on majority of data nodes before deemed successful.
   c. Inserted objects get stored on K (replication parameter) nodes within a bounded time.
   d. Read operations are guaranteed to receive the latest version of the requested object.

4. **Handling Temporary Failures (Extra 10% Credit)**

   Design and implement a technique to handle temporary node failure and rejoin. The proposed system should include,

   a. Online detection of node health. (e.g. detection using heart beats)
   b. Membership management during node leave and join
   c. Preserving the original system properties related to replication and consistency of the data.

4. Code Implementation Summary (50%)

We've provided a very basic code skeleton and run script so as to give you a usage example. Feel free to make any necessary modifications to the classes and function parameters defined inside the given {.hpp, .cpp}. You can also modify the Makefile. There's only one restriction. Do not add any additional {.hpp, .cpp} files so that we have uniform submissions.

- `gtstore.hpp`
- `manager.cpp`
- `storage.cpp`
- `client.cpp`
- `test_app.cpp`
- `Makefile`
- `run.sh`
- `RPC related meta-files if any`

Implementation details

- Code implementation in C/C++.
- You will model each node in the distributed system using a separate process running on a single machine.
- Avoid using shared memory or file system to communicate between nodes. It has to be a network call, so your GTStore can work when deployed across many machines.
- You may use sockets for communication between nodes.
- You are also allowed to use any support for Remote Procedure Calls (RPC). Look at the "References" Section for choice of framework. Note that if you go this route, you will likely benefit only from the stubs for the message buffer management (i.e., sending/receiving the key value pairs), but you will still need to orchestrate the distributed interactions among the client(s) and server(s) by yourself.
- Demonstrate the interaction of clients and GTStore inside `test_app.cpp` and `run.sh`
- You are not allowed to use any other third party libs without explicit permission.

6. Writeup (50%)

Your report is as important as your code implementation and should include an elaborate description of the design choices and functionality. You can also include screenshots of the relevant log messages for every functionality. Additionally, address the questions described in Sections 2 and 3 formatted in the following way:

- **System Components**
    - GTStore Centralized Manager.
    - GTStore Storage Node.
    - Client API calls. (one-by-one)

- **Design Principles**
    - Data Partitioning
    - Data Replication
    - Data Consistency
    - Handling Temporary Failures (if implemented)
- **Client Driver Application**
  Describe anything inside `test_app.cpp` and `run.sh`.
- **Design Tradeoffs**
  We often have to tradeoff one aspect of the system, in favor of another. Describe one interesting property of your designed data store (e.g., lightning fast responses, linear scalability, etc.) and identify how that design decision impact the performance of the rest of the system/ particular aspect of the system (e.g., lowered throughput, increase in storage space/replicas, etc.).
- **Implementation Issues**
  Describe any restrictions or faults your current implementation is having, if any.

7. Collaboration

We encourage you to work in groups of 2 for this project. You can discuss solutions approaches with other students or on piazza, but absolutely no code sharing/copying is allowed. Your code will be tested for potential plagiarism with any other submission or with existing online solutions of related past semester projects.

8. Deliverables

Follow these strict guidelines for the project submission:
One team member submits `team.txt` including the names of the team members.
The other one compresses and submits the following inside
`cs6210-project04-<GTusername>.tar.gz`

- `report-project04-<GTusername>.pdf`
- `team.txt`
- `Makefile`
- `run.sh`
- `gtstore.hpp`
- `manager.cpp`
- `storage.cpp`
- `client.cpp`
- `test_app.cpp`
- `Any RPC related meta-files if any`

Do not include any object files or executables. Your code should compile and run on the provided class servers by executing the `run.sh` file. If you're not able to run it on the class servers, please inform the TAs.

9. References

Refer to the following papers, which you can find in `canvas/files/papers/`:

- **Dynamo: Amazon's Highly Available Key-value Store.** Go through the described design principles that a distributed key-value store should feature. Your implementation, however, need not be the same as the paper.
- **RPC options**
  Choose one of the following. Spend time to familiarize with their usage.
  - http://xmlrpc-c.sourceforge.net/
  - https://docs.oracle.com/cd/E19683-01/816-1435/index.html
  - https://grpc.io/docs/tutorials/basic/c.html

10. Late Policy

A penalty of 5% per day will be applied for late submissions for up to 5 days. Submissions received more than 5 days late will receive no credit.

11. Considerations

You're encouraged to fork the existing github repo that includes the skeleton code, but please do so in a private repo. Frequently commit your code under version control, do not rely on the availability of the class servers to access your codebase. Finally, kindly consider keeping your code in a private repo after finishing up with the class (e.g. available upon request) so as to help us reduce future plagiarism cases.