Daniel Gao, Brian Yeung
CS4500
eau2 Report

Introduction

eau2 is a distributed key-value store. It can store key-value pairs and retrieve keys from any node in the system. Each node can access other nodes very quickly to read from their respective key-value stores. Applications that run on top of each key-value store have access to all values in the system.

Architecture

- A node is a client that registers with a master server, which enables it to communicate with other clients registered to that master server.
- Each node has its own key-value store (which may or may not be a part of a larger key-value store) and can run an application.
- Each key is unique and contains information about which node it's home to.
- When retrieving a key in an application, if the key is not on the node's own key-value store, it will ask the key's home node for the value.
- Values are dataframes which are fancy tables that contain columns of data that follow some sort of schema.
- Columns consist of chunks of data, which are stored across KV stores round-robin style. Each Column simply stores a list of keys of where to find its chunks.

Implementation

Server and Client (the network)

- The *Server* class manages nodes on the network. It does not store a key-value store of its own. It supports registration from incoming clients, facilitating communication between clients, and broadcasting its list of clients every time a client is added or removed.
- The *Client* class is a node on the network. It must register with the *Server* on startup. It is able to communicate to other clients upon registration. It is also able to independently shut down by removing itself from the server registry. The *Client* is responsible for responding to requests by a user to retrieve some value from a key-value store.

Dataframe (the "values" in our key-value store)

- The *Dataframe* class is a representation of a tabular data format, which contains a list of *Column*. A user may gather data about the data in a *Dataframe* via various querying methods, which allow querying by row, column, or row-column coordinates. A user may also perform some sort of mapping operation over the contents of a certain row.

- The *Schema* class is a meta-row in a *Dataframe* that defines the types of each of its columns. Be careful updating it, especially if it describes a very complex and large *Dataframe*. Errors may result in undefined behavior.
- *Dataframe* objects cannot be modified in-place. A user may add new columns or rows to an existing *Dataframe*, but fields cannot be updated, deleted, or overwritten.

Key-Value store (the lowest level)

- A *Key* is an object that contains a unique string name, as well as a node number. The name must be unique on the node that the *Key* is stored on.
- A *Value* is any sort of serialized data, stored as a *char**, as well as its length. Most likely, this will be a serialized *Dataframe*.
- These pairs will be stored on disk in the form of a *KVStore* on whatever machines are hosting each of the nodes. They can be accessed via queries by a user, which are executed by a *Client*.

Use Cases

To create a key-value pair and insert it into the global *kv* store:

```
Key k(4, "residents");   // key stored on node 4, with name residents
Schema s("SSI");           // Schema of string, string, int (name, address, age)
Dataframe residents(s);    // Dataframe that contains a table of residents
Row joe("Joe", "13 Main St", 41);
Row ann("Ann", "14 Main St", 43);
Row mary("Mary", "15 Main St", 6);
residents.addRow(joe);
residents.addRow(ann);
residents.addRow(mary);
kv.insert(k, residents);
```

To access some value and return it:

```
Key k(4, "residents");
// Retrieves the value from the store it's on, even if it is NOT on this node!
auto val = kv.get(k);
if (val)
{
    return val;
} else
{
    // The key does not exist on node 4, proceed accordingly
    throw std::runtime_exception("Key does not exist at node " + std::to_string(k.node());
}
```

Status

Done so far:

- The Dataframe can be read in and internally constructed from any Schema-on-read (sor) file.
- The network is fairly robust and can handle new client registration and teardown, as well as support direct messages between clients. The server is able to shut down and initiate a clean network teardown as well.
- The serializer is in good shape, and can reliably serialize various forms of data. Complex objects such as Columns, ColumnChunks, and Dataframes can all be serialized reliably.
- The Key-Value store is able to store key-value pairs, and is able to handle simple queries.
- The network is able to handle requests between nodes and exchange accurate data.
- Data can be distributed evenly across the system, and data on different nodes can be accurately retrieved.
- Caching is utilized in retrieving values from other nodes, so repeat queries are no longer required. This sped up the runtime of m3 significantly (over a minute, to under a second)

TODO:

- Finish getting milestone 4 to work. We had a lot of tech debt leftover from milestone 3, and spent the majority of this past week absolving that debt.
- Continue to implement a real network interface. Right now, we are still relying on the pseudo-network that involves imitating clients with threads. Progress has been made since the last deadline, but more debugging and tests are needed.
- Continue to merge Linus code into ours.
- Continue to test the network and test that data is reliable and available after being sent to other nodes.

Time estimates:

- Key-value store: 10 hours
- Key implementation: 5 hours
- Tests: 10 hours
- General debugging: 20 hours
- Network implementation: 15
- Network testing: 15 hours
- TOTAL: 75 hours