EAU2 System Design Document

Introduction

The EAU2 system is an application that can be booted up and create a cluster to hold data. Data will be read using a Sorer (schema on read) based data reading system from text files. The data for this system is read only so once data is inputted it cannot be overwritten. Queries can be made to access the data across multiple nodes in the cluster. Being able to have fast/efficient queries is especially important for this system.

Architecture

At a high level, there are three main parts of the this system. First, there is a key value store at the bottom that handles management/retrieval of data. This layer also includes networking code so that this system can be distributed across multiple nodes that each have their own memory. Concurrency of the system will also be handled in the bottom layer. Key's will have information on not only the key to the data that is needed, but also which node the value is stored on so that when a query is made, the node on which it is made knows which node it needs to communicate with to read the data. Having this home node be known for keys speeds up the system in the long run because then if a node does not have the key in it's own KV store then it would have to wait to hear back from every other node to see if they have the data.

Above that is abstraction of data in the form of DataFrames and distributed arrays. These data structures build up their data by calling for data from the bottom layer which then accesses it's KV store.

The top layer above that is where the application code lies so that their can be interactions with the users. What this layer will have to do more specifically remains to be seen.

To go further on how the system will be distributed, there will be one main node that handles the registration of the system. Other nodes will be made and they will register with the main node and be given information on how to access the other nodes in the system. This allows each node to be able to have direct communication with the other nodes. The Rendezvous Server and Node classes will have to be reworked to work with the application and actually allow the application to be a distributed system rather than just having a naive message sending system that is the current network code implementation created.

Implementation

Application:

The main application class. Running this class will handle launching the entire application and setting up all other classes necessary to set up and run the application. The Trivial Application subclass was also added to the code base as an example usage of how an application may run using the infrastructure. This Trivial subclass was provided in Project Milestone 2 as an example usage of the infrastructure that is in the process of being built. The Demo code from Project Milestone 1 to be implemented in Milestone 3 was also added and is fully implemented. Applications take in a KDStore* to be used for the store functionality, and this is a pointer because the main file will handle starting the server functionality of the store. ##TODO0 Add Word Count Stuff

RendezvousServer and Node:

These two classes have the network code needed for the application. The Rendezvous Server is essentially a server meant to sync up all Nodes in the system. Another way of describing it is by calling it a Registrar. When a Node is created, it needs to know the IP of the Registrar to connect to so that the Node can learn the IP's of other Nodes in the system. When the Registrar has a new Node connect to it, it sends a list of the IP addresses of the other Nodes to the new one along with sending the received new IP address to all of the other Nodes in the system. This allows all of the Nodes to be fully aware of each other. Once a Node is integrated in the system, it will have the IP addresses of the other Nodes so it can then send and receive messages to/from them. There are also two lower level classes for abstracting the actual network behavior even more, these are the Server and Client classes. A Server object is used by the Registrar so that the Nodes can connect to it. The Nodes use both Clients to send a message to the Registrar along with sending messages to other Nodes. For Nodes to be able to receive their messages from other Nodes they also need their own Server object also listening. Since the Nodes need to be simultaneously connected to the Registrar along with being able to receive/send messages, threading is needed to run both of these functions at the same time.

A lot changed to the Node and Rendezvous Server this week to make the Demo work and a lot more will have to happen moving forward. For what is is new, first the Node and Rendezvous Server now interact with Register and Directory objects being serialized. The other big change starts with the Node taking in a node index and the Map in its constructor so that it is tightly coupled with the KDStore. This way, there can be put and waitAndGet functions in the Node. The Node class also has functions for being on the receiving end of these requests that handle the other end of the back in forth that goes in between Nodes. The Node to Node interactions involve normal Messages, Status's for passing on a string, Acks for when no information needs to be sent but want to let the other know that it is ready to receive, and then the Column's are serialized and then broken up into 1024 byte chunks to be sent. The receiver of a DataFrame is told how many bytes the serialized column it is going to receive by that being the ID of a message. It can then have a buffer large enough to handle that full message. Once a Column is fully sent, another normal message is sent to show that the column is finished and can then be built.

DataFrame:

The top level class for data storage in the application. The DataFrame will store data in row/column format. There are column classes specific to being used for the DataFrame, one for each of the main four data types (Boolean, Float, Integer, and String). A DataFrame also has a Schema object inside of it that is meant to know the format of the data along with the name of any of the rows/columns if they exist. Data can be added to a DataFrame by inserting entire Rows or entire Columns but the data being added must stay consistent with the format of the current data.

Parser and Sorer Columns:

These classes make up the Sorer functionality used to read in data from .sor files. A parser object is created with the file to be read from, starting location, byte length, and the total file size. This object then has functions for guessing the schema of the given file and for handling the full parsing. Once these two are used, the Parser object will be holding Sorer Columns that contain all of the data. These columns are aware of missing data and are different from the columns used in the DataFrame. The data can be accessed by reading through the columns. For this application, the data is read by first seeing how long the columns are by checking the length of the first one (the length of all columns should be the same). Once this is found, it can be used along with the length of the set of columns to read the data row by row. Each column can be casted to figure out whether the data inside is a Boolean, Float, Integer, or String. The data that is read can then be added to a DataFrame.

KVStore and KDStore

The store layer of the system is currently comprised of four classes: Key, Value, KVStore, and KDStore. The KVStore uses the Map from the utility classes to build a key-value store, using the Key class for key objects and the Value class for value objects. The KVStore has no knowledge of DataFrames or other classes that may be used in the running of an application. The KDStore is the same concept as the KVStore, serving as a key-value store, but this class is built specifically to handle DataFrames. The KDStore acts as an API for the application tier of the system, allowing for storage of DataFrames. The Key class is used as keys in the KDStore, as they were in the KVStore, but instead of the Value class being used for values, DataFrames are specifically used as the store's values. This allows the application tier to access the store tier, with a convenient API that does not require casting of Values to DataFrames. The Application class includes a KDStore instance as a field. The fromArray method on DataFrame can be used to create a new DataFrame and add it to the calling application's KDStore.

Currently, the implementation for the network behavior is having the KDStore hold a Node* and have that object hold a pointer to the Map created for the store. That way when the Node is running it's network behavior in the separate thread, it has access to the map in case another Node asks it for a value for a Key.

Serialize and Message Classes:

These classes were pulled over with the intent of being used by the network layer to communicate. There is a Serial class that is used to both deserialize and serialize eligible objects. The deserialize method determines which kind of Object the input is and calls on the deserialize constructor for that object. To serialize an object, there is a function that takes in the object and appends different characters and/or a char* that is the serialized message which is then returned.

Use Cases

```
// Creation of Dataframe:
Schema builder("IFFSB");
builder.add_row(nullptr);
builder.add_row(nullptr);

Dataframe dframe(builder);

Row r(builder);
r.set(0, 1);
r.set(1, (float)1.0);
r.set(2, (float)2.0);
r.set(3, new String("Hi"));
r.set(4, true);
dframe.add_row(r);

r.set(0, 2);
r.set(1, (float)2.0);
r.set(2, (float)4.0);
r.set(3, new String("Hey"));
r.set(4, false);
dframe.add_row(r);

//================================================================

// Usage of KDStore and fromArray in Application layer:
size_t SZ = 10;
double* vals = new double[SZ];
for (size_t i = 0; i < SZ; ++i) vals[i] = i;
Key key("triv",0);
DataFrame* df = DataFrame::fromArray(&key, &kv, SZ, vals);
DataFrame* df2 = kv.get(key);
double second = df2->get_double(0,1);
delete df;
delete[] vals;

//================================================================

// Demo Usage of basic Distributed System Functionality

class Demo : public Application {
public:
    Key main("main", 0);
    Key verify("verif", 0);
    Key check("ck", 0);

    Demo(size_t idx, KDStore *kd) : Application(idx, kd) {}

    void run_() override {
        switch (this_node()) {
            case 0:
                producer();
                break;
            case 1:
                counter();
                break;
            case 2:
                summarizer();
            case 3:
                break;
        }
    }


    void producer() {
        size_t SZ = 100 * 1000;
        double *vals = new double[SZ];
        double sum = 0;
        for (size_t i = 0; i < SZ; ++i) sum += vals[i] = i;
        DataFrame::fromArray(&main, kv, SZ, vals);
        DataFrame::fromScalar(&check, kv, sum);
    }


    void counter() {
        DataFrame *v = kv->waitAndGet(main);
        size_t sum = 0;
        for (size_t i = 0; i < 100 *1000; ++i) sum += v->get_double(0, i);
        pl("The sum is  ").pln(sum);
        DataFrame* s = DataFrame::fromScalar(&verify, kv, sum);
        delete v;
        delete s;
    }

    void summarizer() {
        DataFrame *result = kv->waitAndGet(verify);
        DataFrame *expected = kv->waitAndGet(check);
        pln(expected->get_double(0, 0) == result->get_double(0, 0) ? "SUCCESS" : "FAILURE");
        delete result;
        delete expected;
    }
};

//================================================================

// Run Distributed WordCount Application Usage (not currently working)
int node = atoi(argv[2]);
char* filename = argv[4];

KDStore* store = new KDStore(node);

WordCount w (node, store, filename, 0);

std::thread network_thread(&KDStore::run_network, store, kill_switch);

std::thread wordcount_thread(&WordCount::run_, w);

network_thread.join();
wordcount_thread.join();

delete store;
```

Open Questions

Are they any huge problems or design decisions that you would recommend staying away from in our network code?

Status

All of the code to be reused has been brought over into the directory along with the tests for all of them. The tests are all together in one folder and can be run together with the main Makefile with "make". The code to be reused can be split up into a few different types of classes. The first is utility classes; these include Object, String, Array, and the other common data structure classes. The next group of classes are the classes that are used by the DataFrame; this folder has all of the classes that were new for the DataFrame assignment. For network specific code, this is in the network folder, and related to that, the serialization code along with classes to be used as serialized messages are in their own folder. There is also a folder called store to hold objects pertaining to the KVStore / KDStore. An application folder exists to hold the Application class as well as subclasses of Application. Lastly there is a folder for sorer code that came from the group 45000NE. There is one main.cpp file in the top level directory that is currently used to run the Demo application that was meant to be functional for the last assignment. The tests have been run with valgrind to ensure that any memory issues that were in the code for the previous assignments have been fixed.

There was a large list of problems that needed to be cleaned up before moving on to the WordCount application that were mentioned in the report from last week. All of those issues were fixed and they took up a large amount of time. We were able to finish making sure the Demo works smoothly now with only 3 nodes and the code was trimmed down to remove repetition. Currently, the Demo can be run using runDemo in the Makefile and after running that over 20 times in a row there were no problems. There was an occasional issue where Node 1 was stuck and then after ending the Rendezvous Server executable (node 0) it appeared that a DataFrame was not being sent correctly. This issue was happening around every 10 uses on average so it was not easy to isolate. After a small change to the network code related to checking the type of acknowledgement received during a put request this issue has not occurred but we are not 100% certain that it was the issue. The Demo can be run with valgrind and has no memory leakage, but to do that we would lower the size of each DataFrame to 100 instead of 100 * 1000 so that the process would not take a long time. Running valgrind without lowering the size was taking longer than 5 minutes so that seemed like not worth actually checking.

The provided WordCount application from Milestone 4 has been implemented under application/wordcount and is run by compiling wordcount.cpp and running it with `./wordcount --node [node-number] -f [filename]`. Example files to run WordCount on are found at `data/10.txt` or `data/100k.txt`. `10.txt` is a small sample of 10 rows. Currently wordcount compiles correctly, but segmentation faults on attempting to run it. Due to the fact that we spent much of this milestone covering technical debt as described above, we were not able to implement a fully-functional wordcount application. We will continue to debug wordcount as we approach the date of our final code walk so that we have an example application that utilizes a distributed system to demonstrate. So far, we have been able to add the necessary WordCount code we were provided without having to modify our existing code very much, but rather by modifying wordcount.h to adjust to the APIs we created.

UPDATE: We both underestimated the work we would have to do to prepare for our other finals so we never got to actually fixing the problems. Therefore we only have the Demo working.