Jessica Cheng
Sharon He

Introduction:

The eau2 system is a distributed system that consists of three layers. At the bottom layer, there is a distributed KV store running on each node. The KV store is essentially a map that has get and put methods. The eau2 system is a distributed system so there will be multiple nodes. On top of each node, it will run a KV store. Each KV store has part of the data. There exists a network communication layer that connects KV stores together so KV stores talk to exchange data when needed.

The level above provides abstractions like distributed arrays and data frames. Calls to the KV store will implement these abstractions.

The application layer is the top layer. An application can access any data that is hosted through the system. It is running on some number of nodes somewhere in the distributed system. These applications can call the KV stores to get and put keys

Architecture:

At its highest level, eau2 is a distributed structure, such as an array or dataframe. Below the distributed layer, there is a networking layer, for the KV stores to communicate, and even below that, there is a KV store node.

In the networking layer, we have a "server" node that registers each of the other client nodes. All the nodes are KV stores. First, they register with the "server" KV store to let the server know they are part of the network. Then, when a node needs to get something at a given key, it will check whether the key is in the current node. If not, the key has a node index, and the client current node can send a request to the node index of the key, and that node will, in turn, return the key and value necessary for the current node to reconstruct the data it needs.

For the distributed structure, we have distributed KV stores across several client nodes. In prior implementations, we believed each node, either server or local, would store a KVStore, but it is actually the opposite, where each KV store contains an instance of a node. In our main functions, we run the networking layer by starting creating a new KV store and then calling init_server, which initializes the server node and allows for us to call new KV stores and provide client nodes. The way these KV stores work is related directly to the nodes, where when we want information from another KV store, we must communicate between KV stores using the nodes. The idea is the node stored in the requesting KV store will be able to send a message to the providing KV store, and the providing KV store, upon receiving the message, will return a message to the requesting KV store with the serialized results of the requesting KV store's query.

Implementation:
The Key class is a subclass of class Object and represents a key in the KV store. It contains fields String name and size_t home. It contains Object methods such as equals and hash. The Key class's home node field indicates which KV store the key is in, to be used when networking, so clients can directly request and send to the node that contains the data they need.

The Value class represents the value in a KV store and contains a field which is the serialized char* data as well as a length field for the total length of the serialized string.

The KV store is represented in a class KVstoreand is similar to Map. The class contains Key and Value fields and contains methods:
get(k) => DataFrame*   returns a deserialized DataFrame based on the given key k if k exists in the current KV store, or nullptr if k is not in the current node
put(k,v) => void   puts a new key k and value v pair into the KV store
getAndWait(k) => DataFrame*   returns a deserialized DataFrame. First checks if the key exists on the current node, but then blocks and waits for a DataFrame to be put in the node given by the key k before sending the DataFrame key over to the current node, which allows for the current node to recreate the DataFrame from the values stored at the keys in the distributed kv stores. After further discussion with instructors and classmates, we have decided on a design that can store a DataFrame in the KV store, but that there will also be ways to access the parts of a DataFrame, such as through keys to columns.

Our implementation no longer has separate server and client classes. Instead, we have one class, a NetworkIP class, that stores the current node and ip address, and has five main methods: init_sock_, server_init, client_init, send_m, and recv_m. The init_sock_ method is called by both server_init and client_init to initialize a new client. Then, when the network is first starting up, we call server_init, which initializes a server node at node 0 (the server is always at node 0). We can then call client_init as many as nodes-1 times to create the client nodes and complete the network. Once all of the client nodes have been created, server_init will send the complete directory to all of the clients (and will have its own directory) and the server node will become just another client node. With the directory and the "target_" field of messages, we can directly communicate between clients and send data back and forth.

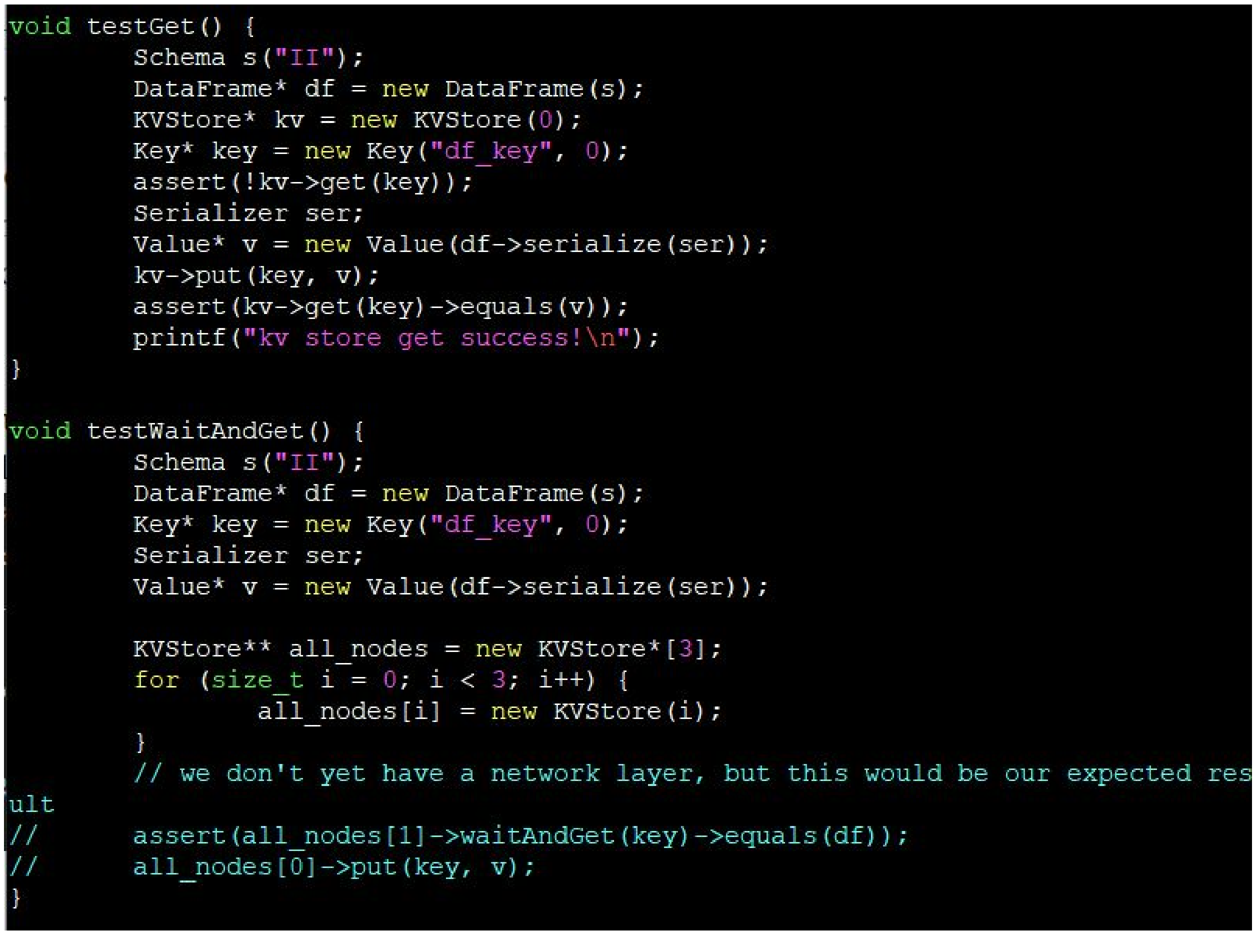
The DataFrame class is built off of a Schema, which holds the row and column names in the DataFrame. The DataFrame itself stores data in a columnar format, keeping track of the columns according to their types, the order of which are determined by the initial schema. In our updated code (as of milestones 2 and 3), the DataFrame can be built off of arrays of data and scalar data. This is done by first constructing a new DataFrame, given the type of the element being added. We then create a new column, setting the values from the array inside the column, and add the column to the DataFrame. In this way, we create a DataFrame with a single column, which is built up from an array or scalar value.

The Column class has been changed and implemented using chunks, which are essentially arrays of arrays, using our hand rolled Array classes. We have arbitrarily determined the size of each chunk (currently 1000, but can be changed), meaning each "element" in the column is an array of length <size of chunk>. This allows us to build a DataFrame from a column without becoming too large, and allows for more abstraction between the application and the payload.

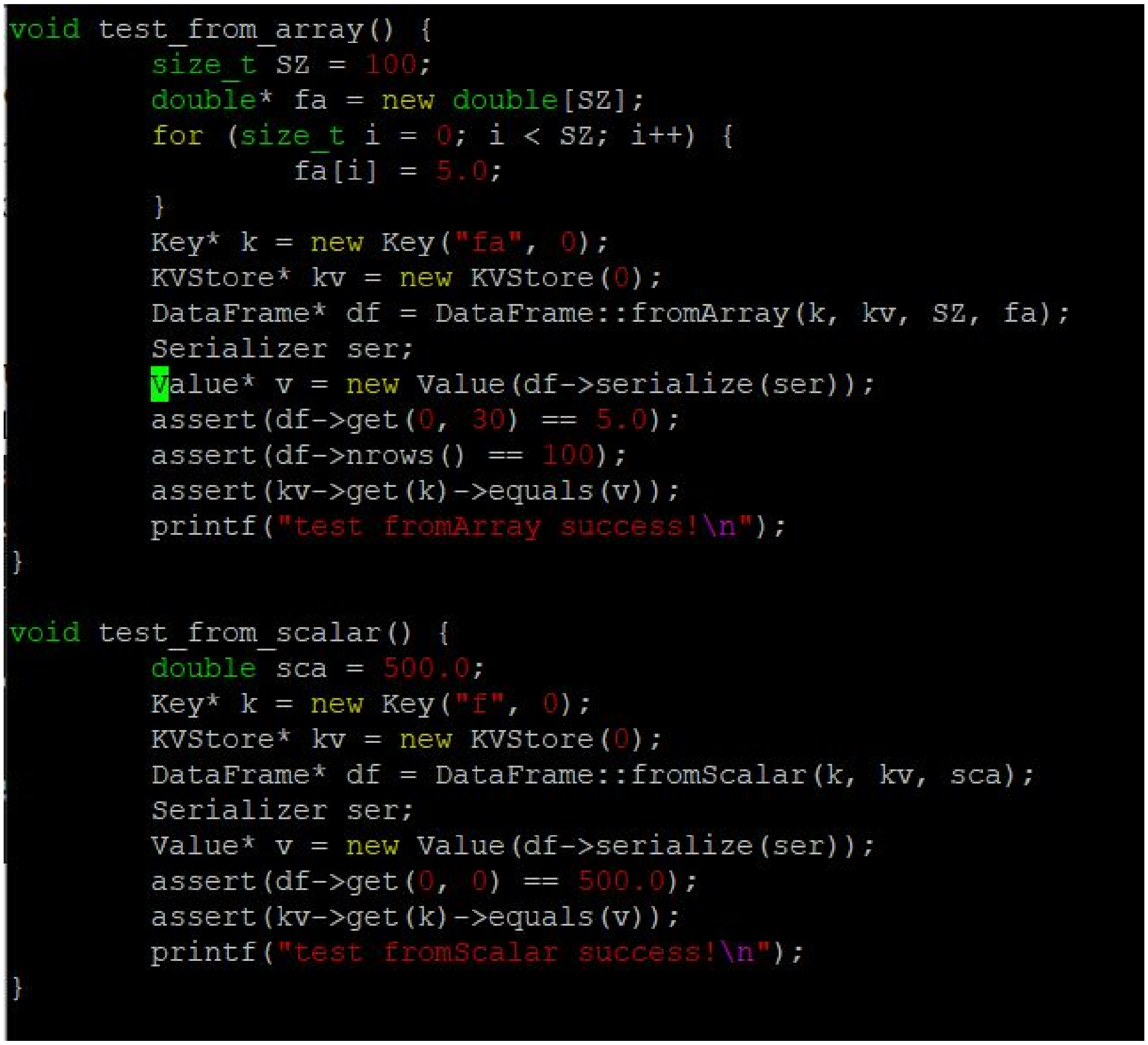
Use cases: examples of uses of the system. This could be in the form of code like the one above. It is okay to leave this section mostly empty if there is nothing to say.

```
FILE *f = fopen("../src/data.sor", "r");
   SOR* sor = new SOR();
   char* schemaFromFile = sor->getSchema(f, 0, 1000000);
```

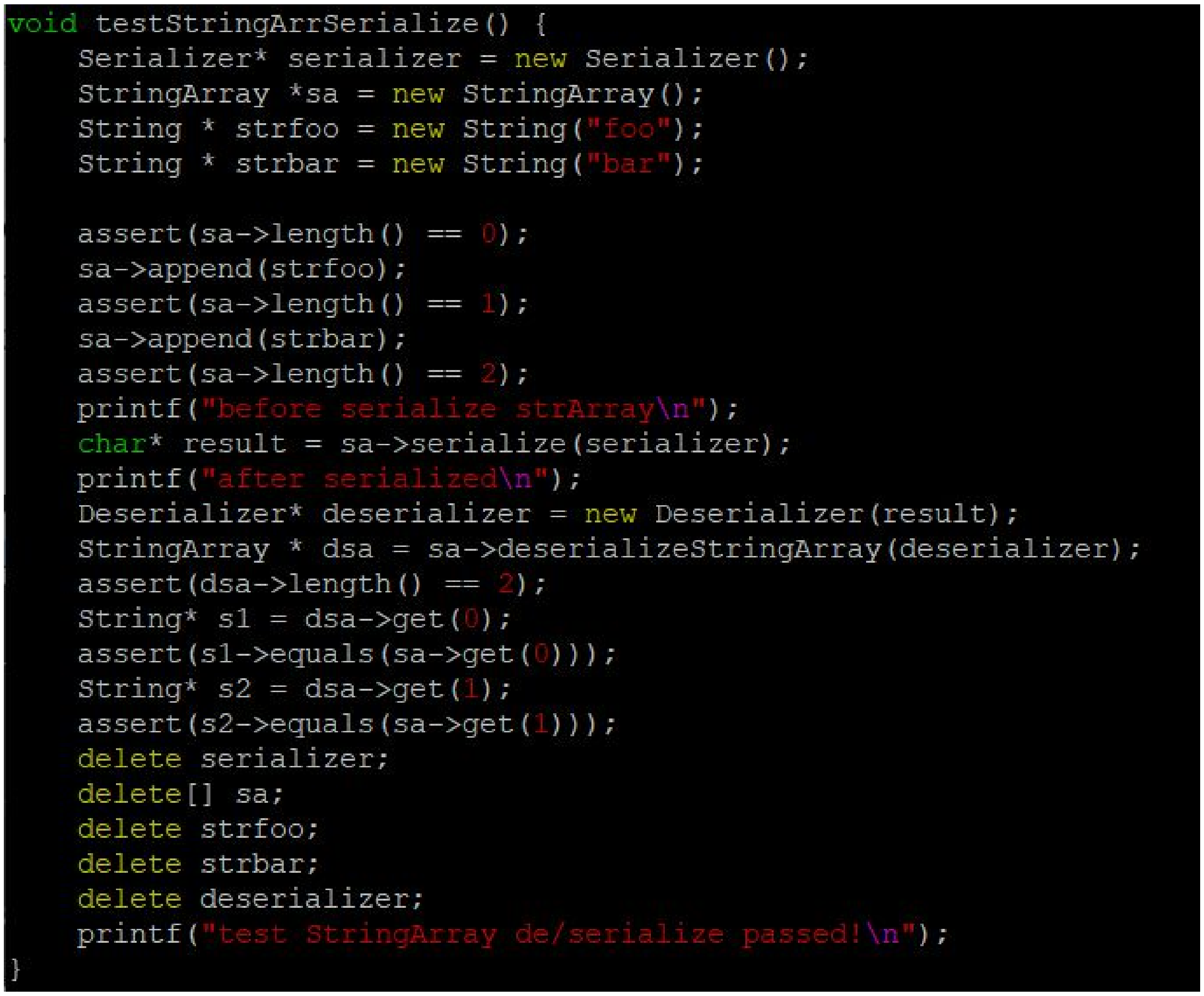
```
printf("schema is: %s\n", schemaFromFile);
Schema s(schemaFromFile);

DataFrame* df = sor->setDataFrame(f, 0, 100000);
df->print();
```



```
void testGet() {
        Schema s("II");
        DataFrame* df = new DataFrame(s);
        KVStore* kv = new KVStore(0);
        Key* key = new Key("df_key", 0);
        assert(!kv->get(key));
        Serializer ser;
        Value* v = new Value(df->serialize(ser));
        kv->put(key, v);
        assert(kv->get(key)->equals(v));
        printf("kv store get success!\n");
}

void testWaitAndGet() {
        Schema s("II");
        DataFrame* df = new DataFrame(s);
        Key* key = new Key("df_key", 0);
        Serializer ser;
        Value* v = new Value(df->serialize(ser));

        KVStore** all_nodes = new KVStore*[3];
        for (size_t i = 0; i < 3; i++) {
                all_nodes[i] = new KVStore(i);
        }
        // we don't yet have a network layer, but this would be our expected res
ult
//      assert(all_nodes[1]->waitAndGet(key)->equals(df));
//      all_nodes[0]->put(key, v);
}
```



```
void test_from_array() {
        size_t SZ = 100;
        double* fa = new double[SZ];
        for (size_t i = 0; i < SZ; i++) {
                fa[i] = 5.0;
        }
        Key* k = new Key("fa", 0);
        KVStore* kv = new KVStore(0);
        DataFrame* df = DataFrame::fromArray(k, kv, SZ, fa);
        Serializer ser;
        Value* v = new Value(df->serialize(ser));
        assert(df->get(0, 30) == 5.0);
        assert(df->nrows() == 100);
        assert(kv->get(k)->equals(v));
        printf("test fromArray success!\n");
}

void test_from_scalar() {
        double sca = 500.0;
        Key* k = new Key("f", 0);
        KVStore* kv = new KVStore(0);
        DataFrame* df = DataFrame::fromScalar(k, kv, sca);
        Serializer ser;
        Value* v = new Value(df->serialize(ser));
        assert(df->get(0, 0) == 500.0);
        assert(kv->get(k)->equals(v));
        printf("test fromScalar success!\n");
}
```



```
void testStringArrSerialize() {
    Serializer* serializer = new Serializer();
    StringArray *sa = new StringArray();
    String * strfoo = new String("foo");
    String * strbar = new String("bar");

    assert(sa->length() == 0);
    sa->append(strfoo);
    assert(sa->length() == 1);
    sa->append(strbar);
    assert(sa->length() == 2);
    printf("before serialize strArray\n");
    char* result = sa->serialize(serializer);
    printf("after serialized\n");
    Deserializer* deserializer = new Deserializer(result);
    StringArray * dsa = sa->deserializeStringArray(deserializer);
    assert(dsa->length() == 2);
    String* s1 = dsa->get(0);
    assert(s1->equals(sa->get(0)));
    String* s2 = dsa->get(1);
    assert(s2->equals(sa->get(1)));
    delete serializer;
    delete[] sa;
    delete strfoo;
    delete strbar;
    delete deserializer;
    printf("test StringArray de/serialize passed!\n");
}
```

Use cases for running the networking layer (with one server and one local node):

start_server:

```
g++ -g -Wall -pedantic -std=c++11 -o server test/server.cpp
./server 127.0.0.1 8000 0
```

start_client:

```
g++ -g -Wall -pedantic -std=c++11 -o client1 test/client.cpp
./client1 127.0.0.1 8000 127.0.0.1 8001 1
```

Status:

- Out of all the demo applications, Trivial is able to run on a scale of 1000 by 1000 values of floats. Unfortunately, due to time constraints and pressure from other classes (which we worked on per the advice of Professor Vitek), we were not able to complete the Demo application. We have found we are able to at least register our clients to the server and send and receive our wait and get messages, as well as send KVStore keys across the network to the node we wish to get from, as well as send data back, but we ran into some errors deserializing any Values we sent across the network in a Message (we suspect due to our readChars method in Serializer).