SOFTWARE DEVELOPMENT
Eden Desta and Lucas Calero Forero

Final Code Proposal
Software Architecture

Introduction

Eau2 system is a three layered system. The first layer (also referred to as the bottom layer) is a Distributed Key-Value (KV) store. The second layer (also referred to as the middle layer) is a data-frame abstraction layer. Finally, the third layer (also referred to as the top layer) is the application layer. The system has multiple nodes from 0 to num_nodes. In the architecture section of our report, we will delve down into the details of the system and how we expect to build it from the ground up. This section also includes the restrictions that we have placed in our design.

Architecture

There are several restrictions of our architecture that can be found in *Table 1* below. These restrictions will go into the final design we will propose to the team. As mentioned previously, our architecture consists of three layers, Bottom Layer: Distributed KV Store, Middle Layer: Dataframe Abstraction, and finally the Top Layer: Application Layer. We will delve into the details of each layer below. *Figure 1* outlines the general architecture of the system.

Data is read only once at the beginning of our system • data contained within the data-frame is read only
There will always be less than 100 columns contained in our dataframe
We are only supporting 4 types of data 1. Integer

2. Double 3. Boolean 4. String
There will be a fixed number of nodes (identified by a number between 0 and num_nodes)

Table 1: *System Restrictions*

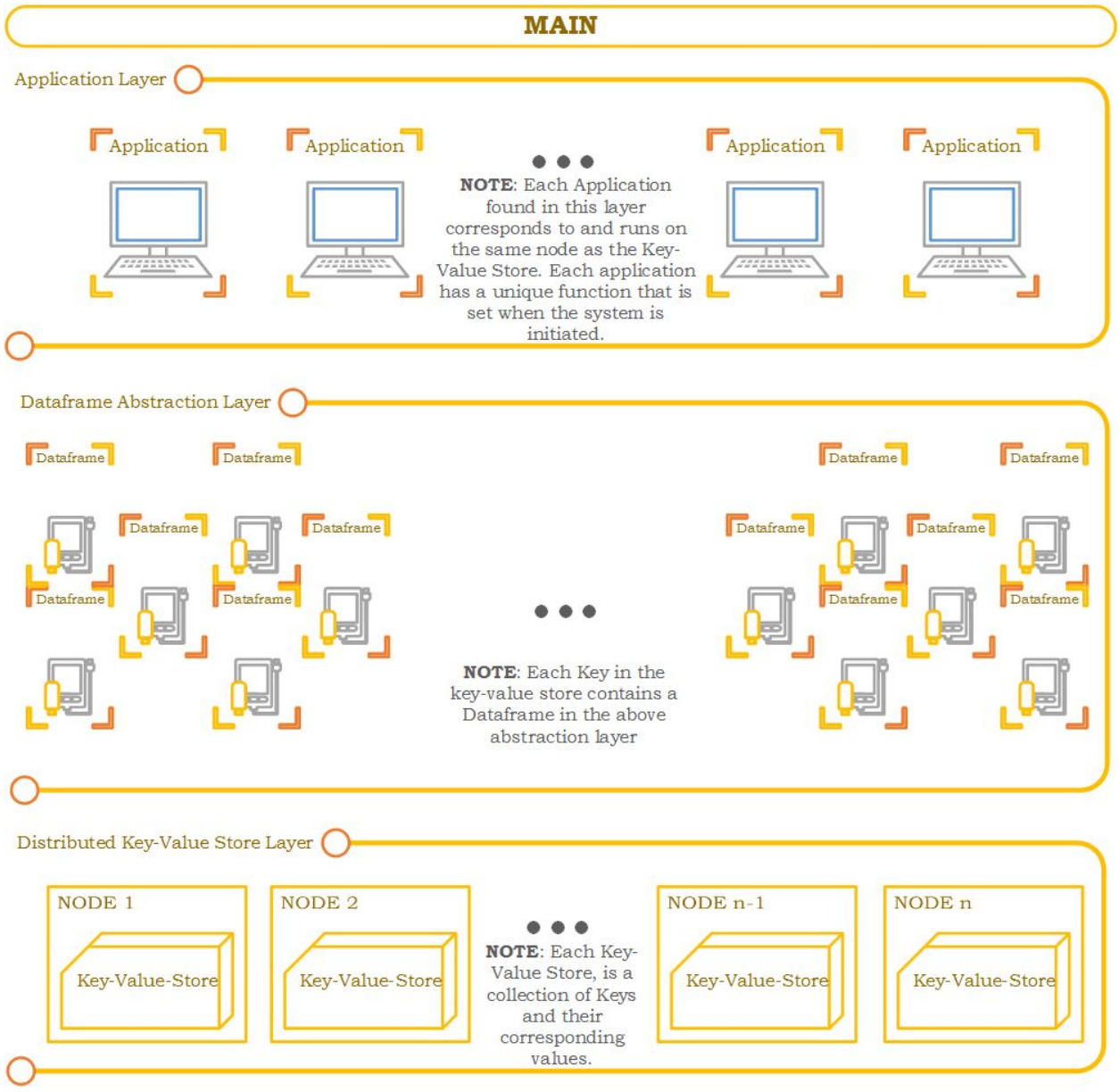

Figure 1: *Top Level System Diagram*

As seen in *Figure 1*, the top level system diagram breaks down into three sections. The Distributed Key-Value Store Layer, runs on multiple nodes that communicate with one another. These nodes are constantly aware who exists. The Implementation layer outlines the details of each three layers and the classes that make them up.

Implementation

A key-value store is depicted in a graphic in *Figure 2.* Essentially, the Key-Value store hosts a known set of key-values. The Key corresponds to a unique value that can be used to identify the Value contained within it.

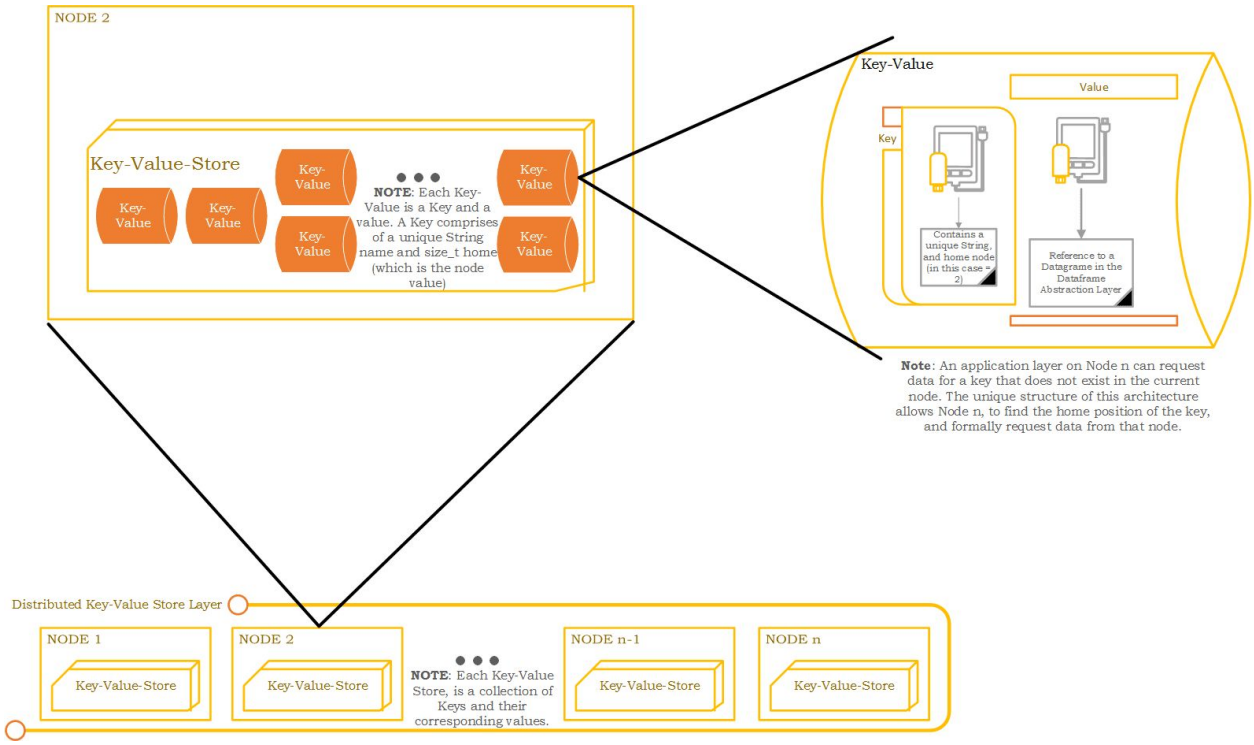

Figure 2: *Distributed Key-Value Store Layer*

As depicted in *Figure 2*, a key-value store contains multiple key-values. A key always has a unique String name that identifies it. There will never exist two keys with the same name. A key also contains a size_t that indicates which node the pair can be found. This is because each node in the entire application has a singular Key-Value Store, and every store has a

plethora of Key-Values. This allows easy interfacing in the case where an application on a separate node sends down a command to its distributed key-value layer. KV store is able to assess that the home of the key its application is looking for is elsewhere and formally request the value through the network.

The Data-frame abstraction layer is unique. This is because the Key-Values contain a reference to a specific Data-frame. A Data-frame has its unique set of functionality, however this layer, is a data container that can be referenced by the Distributed Key-Value Store Layer. Each Value in a KV has a value to a dataframe. A reference to a dataframe is not necessarily unique. Dataframes are not editable, they are only written to at the beginning of the program and viewed whenever called upon.

The top layer, the Application layer is very similar to the Distributed-Key Value layer. The unique aspect of the application layer is its ability to perform a singular function that is set in the main (when the system is deployed). The application layer has no knowledge of the KV-Store and its contents. It only knows what key it wants to request and requests it in the store that is on the same node . The Distributed layer handles the rest of the logic to get it back to the application layer. The application then performs a unique operation on the data that it has received.

Running

In order to run the code. You must configure the “config.ini” file found in the home directory of the software. The system uses that config file to determine the ip address of the server and the name of the client node.

When running the system. The Makefile should be able to run the main server and the respective clients by running the “run_main” and “run_client”. These two commands build and run their own aspects of the software.

When the main begins running it spins up the socket that handles all communication going in and out. This allows the system to constantly be aware of the clients connected and those who are not. This ensures a tight run of the space when the system is attempting to send information to a client that might not exist. The server (also know as the rendevous server, is one of a kind). There is only one rendevous server in the

entire system. The clients however can populate from one to however many PCs are willing to connect.

The clientUI, builds a simple user facing UI that allows the user to interact with the system underneath. The client UI, first spins up and attempts a connection to the server, if the server does not respond, it immediately shuts down. If a connection has been successfully implemented the client runs and connects to the server while also sending its node number and the port it is connected to. This way, the server keeps track of all the ports that are being used. The client then waits for either the server to send a key value, or for the user to create one within itself.

From the client facing side of the system, a user can create a Key-Value pair to add to the store. As mentioned above, there only exists a store per client. It can get a key value pair (this is strictly for outside of the node itself). With this call you can request to retrieve values that exist within the node, however, it is counter intuitive as there is a retrieve value for a user to quickly retrieve the value without having to go through the network. Finally the system has the ability to remove a key value pair as long as they know the key they are working with.

In order to create a new key value, the server can either create one from the file that was read in upon running, or the client can create a dataframe from array or scalar. The file that is specifically read, can be configured in the config file of the system PRIOR to running. The server is the only aspect of the system that has access to the contents of the file. The file can be much larger than the ram, however, the value extracted to be given to a client must be equal to or less than the RAM.

Data larger than the given size are not processed unless they are being parsed into different parts of the client. A client cannot store more than the RAM of the system they are running on. Excess data is handled prior to distribution.

To check validity of the software, running the application has been implemented. This allows for the user to run a demo application. The demo application creates two key-values one, a generated data frame filled with floats increasing from 0 to the given size. And a second value of the system that is the sum of all the floats previously created. The second node, requests the initial data frame constructed from the floats and also retrieves

their sum and stores it to a new key-value pair sent to the first node. Finally the third node checks if the last two values from the key value pairs are equal. If they are, the system is operating as expected.

Use Cases

Use Cases:

➔ **3 nodes exists (therefore 3 KV-stores exists)** ➔ **KV-Store 1:** ◆ **K1-V1** ◆ **K2-V2** ➔ **KV-Store 2** ➔ **KV-Store 3:** ◆ **K3-V3** ◆ **K4-V4** ◆ **K5-V5**
➔ There are 3 applications that also exist -- they all perform Summation ◆ Application 1: ● Requests K1 ● Request received by KV-Store 1 ○ Checks home value of K1 ○ Retrieves Value ◆ Value sends reference to Dataframe ○ Dataframe reference is sent to Application 1 ● Application 1 sums the contents of the data frame ◆ Application 2 -- currently doing nothing ◆ Application 3 -- currently doing nothing
➔ Application 1 -- determines if the value is even ◆ Requests K2 ● Request received by KV-Store 1 ○ Checks home value of K1 ○ Retrieves Value ● Dataframe reference is sent to Application 1 ◆ Application 1 iterates through the entire dataframe and determines if all the values are even ➔ Application 2 -- determines if the value is odd ◆ Requests K3 ● Request received by KV-Store 2 ○ Checks home value of K3 ○ Home value is not equal to node value

        - Places a request to Node 3 for K3's key value
        - K3 process request and retrieves Value
      - Dataframe reference is sent to Application 2
    - Application 2 iterates through the entire dataframe and determines if all the values are odd
- ➔ Application 3 -- sums the contents of two dataframes
    - Requests K5
      - Request received by KV-Store 3
        - Checks home value
        - Retrieves value
      - Dataframe reference is sent to Application 3
    - Requests K1
      - Request received by KV-Store 3
        - Checks home value
        - Home value not equal to node value
        - Request sent to node 1 for value
        - Value retrieved
      - Dataframe reference is sent to Application 3
    - Application sums the contents of the corresponding data frames together by index. If a data frame sizes are not equal, the larger indices are added to 0.

Status

Currently, we have translated our dataframe and our adapter to Python3. We created a GUI for easier reference. To see the adapter actively work and be presented to the user. We used Pyqt5, an executable is included with all the dependencies to remove the hassle of pyqt5 installation.

We implemented a KV store that communicates between nodes through socket connections. We have also implemented several UI views of our application to enhance the user's experience when creating dataframes and for easier debugging and ease of use.

Milestone 1 Walkthrough

As mentioned previously, our Milestone has a GUI that makes it easier to interact with the user. *Figure 3,* displays the current very basic GUI that we have made. There are two

buttons, “Select File” and “Generate”. *Select File* opens up an interactive file explorer for a user to find the file they are searching for. It then fills out the text box above the button. The larger text box, is where the new data frame is placed after the *Generate* button has been selected.

Figure 3: *Current GUI for the system*

Milestone 2 Walkthrough

Milestone 2 introduced a few new classes that have been implemented with the existing design of milestone 1. It introduces the KV store with its unique addition of the key class as mentioned before. It also introduces nodes for future implementation.

Milestone 3 Walkthrough

Milestone 3 introduces threads. It implements a threaded kv store that imitates a small functional ability of the network layer of the kv store. We also implemented a threaded client and server, so the server was able to receive and send messages at the same time. We updated the dataframe to take in a key and a key value store so that once a dataframe is created, it adds it to the key value store as long as the key's home value is the same as the home value of the key value store. Currently we are considering serializing the dataframe, so we can send it to another node in the case the key does not match with the home value of the key value store. The demo application given in milestone 1 was run and tested for functionality of our current system. We also wrote more tests for the newer functionality of our system. We added a readme in the directory for clarity on all the files and their respective functionality.

Milestone 4 Walkthrough

Milestone 4 introduces a threaded node. The node can either be the fundamental rendezvous node that initialises the entire system, or a node that includes a KV store. The rendezvous node does not have a kv store. All nodes are the same class. It was important to ensure this distinction to reduce redundant code. Currently, the system comes setup with a kv store. Eventually the node class will be able to handle creating a kv store with a settable database that is stored. Commands will be set in an enum that will be decoded with the rest of the message that is received. We serialize our dataframes by taking advantage of the fact that python has dictionaries represented as objects. So we created an object serializer that was passed into the json (de)serializer and that was able to (de)construct objects for us.

Milestone 5 Walkthrough

Our time was spent securing all the components we had built up til this point, and trying to demo the API described in Milestone 1. We created an extensive UI to interact with our application. We changed the way that our application creates and stores Dataframes. Instead of having a dataframe API that is stored in a KVStore. We have clients that have KVstores and have the ability of creating dataframes. Another big time crunch that happened for this milestone was testing: we attempted to create unit tests for most essential parts of our codebase.

The code fully implements networks. It has a central rendevous server that handles all communications going in and out. The server filters all the messages its clients sends and packages it to be sent to the correct socket. Though this system is proven to be slower as the application increases, it was a better solution to opt with this to regulate the traffic of the application.

The architecture of our system is as can be seen below:

UI <- Application <- Client <- KV-store <- Keys and Dataframes

This ensures that the application layer is the layer the user interacts with, whilst client is the one that handles all the requests that are sent over from the user. The KV-store as previously mentioned can only exist once per node. It ensures that all the key-value pairs stored within can be retrieved by the functions available to the user via the application class.