SOFTWARE DEVELOPMENT
Eden Desta and Lucas Calero Forero

Milestone 2
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*

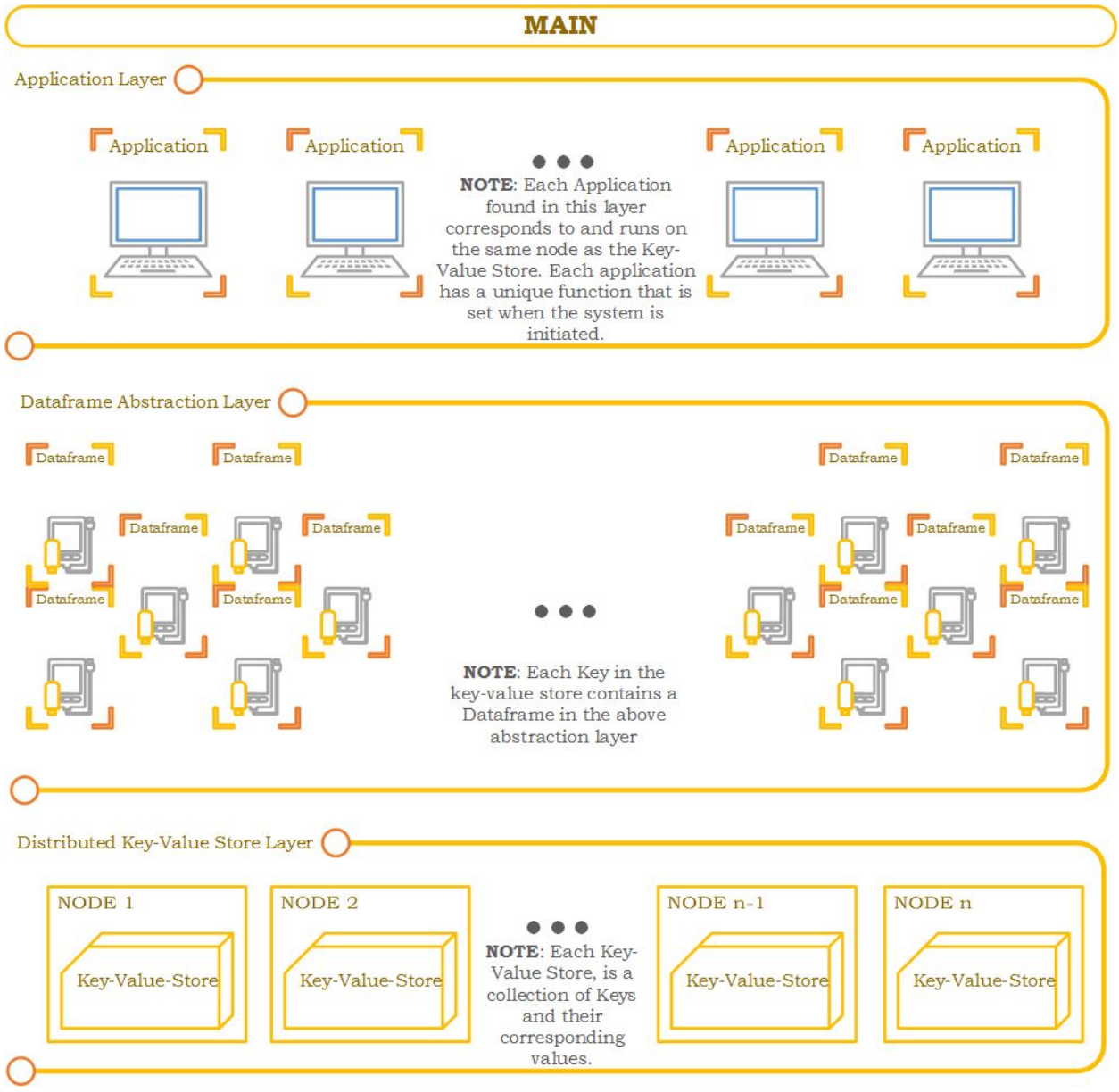

Figure 1: *Top Level System Diagram*

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.

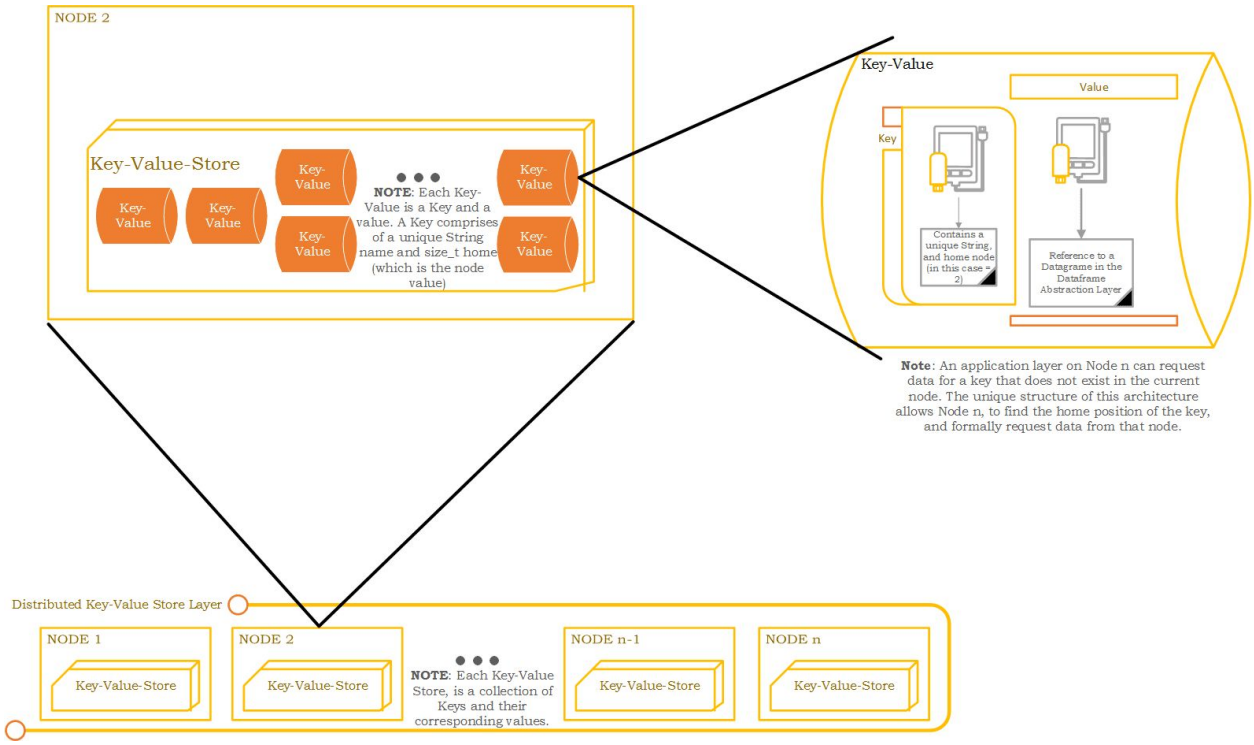

Figure 2: *Distributed Key-Value Store Layer*

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.

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.

Open Questions

1. Is it okay to have a top level main function that does not do anything but deploy the software? It is what we had in mind but was not sure that was something that was okay?

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, the docker tag in the makefile handles the installation of Pyqt5.

We currently have a very simple model in place with an adapter that parsers data from a chosen file. It then allows the user to create a dataframe, from the selected file.

The current KV store has a unique class called a key that stores the name of the key and the home value of the node. The home value is set according to the home node the KV store belongs to. Each key-value pair contains a key and a dataframe. Each key is unique and the data frame is also unique.

A node currently has a unique port that it connects to. It has a very basic receive and accept messages. It is expected that the user sets the port that they want for the new connection.

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.

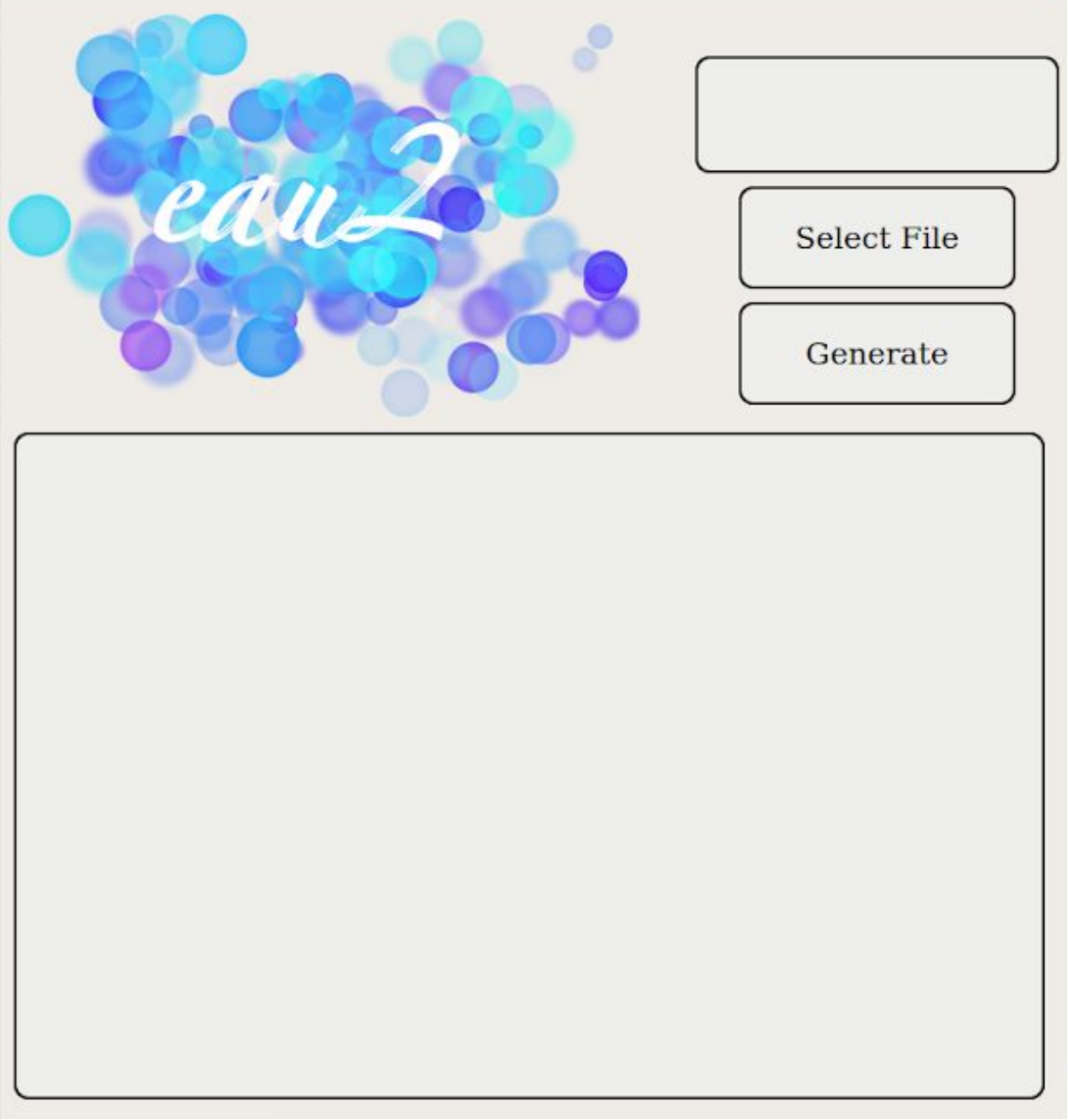

Figure 3: *Current GUI for the system*

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.