BILKENT UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

SENIOR DESIGN PROJECT

Low Level Design Report

10.10.2016

Horus

Group Members

Ali İlteriş Tabak
Hakan Kılıç
Ömer Eren
Süleyman Can Özülkü
Yiğitcan Kaya

Supervisor

İbrahim Körpeoğlu

Jury Members

Özgür Ulusoy, Selim Aksoy

TABLE OF CONTENTS

1.0 Introduction

The Internet of Things (IoT) is a concept of computing which includes multiple interconnected computing devices, provided with identifying tokens, and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction.

The development and adoption of Internet of Things is a critical element of data driven decision making. In the recent years, most of the industrial production has adopted this new concept by digitalizing their production equipment to increase their efficiency and productivity as well as reducing the impact of human error by integrating computer aided control, error detection of the production environment provided by analysis of the data collected.

However, one of the biggest problems in front of the widespread adoption of the IOT in industry, is that it requires serious infrastructure changes. Most of the technology to capture and track sensor and indicator measurements that are developed and installed currently are not connected to the internet and can only be monitored by a designated person. Real-time remote tracking and analysis of the data generated by legacy systems is impossible without changing whole system.

In this project, we propose an interim technology to track legacy control panels, legacy sensors and indicators within a production environment without adding big overhead or introducing any infrastructure changes. HORUS is going to achieve its goals by using a simple camera and image processing techniques in order to analyze the legacy analog

indicators and extract the measurements from them. After the indicator data is extracted and digitized, HORUS will bring power of the cloud computing and big data analytics to help our customers with the analysis and monitoring of their crucial production data that has been accumulated from their sensors and indicators conveniently.

Our system is designed to have two main components, the appliance which operates on client-side, and the server-side. The two-components will be independent from each other, as the client-side will specify the use case of HORUS by collecting data from legacy sensors and indicators, and the server-side will provide application agnostic data storage and data analytics to the data provided by client-side.

The data collection from customers' legacy systems will be handled by a client-side device which we call the appliance and it will be implemented using a Raspberry Pi device with a simple camera that images the legacy sensors and indicators to digitize their data using novel image processing algorithms and a wireless communication module to transmit the data to server-side. On the server-side the authentication and sterilization of the data will be performed, then the data will be analyzed to detect anomalies within a data using unsupervised anomaly detection algorithms, also it will be analyzed to trigger alert based on user defined static rules. The data and the analytics will then be presented to users through a dashboard.

Architecturally speaking, server-side, data-processing component of the HORUS will adopt **Lambda Architecture**. [1] Lambda Architecture is a generic, scalable, fault tolerant architecture designed for data-intensive systems that do data-processing as HORUS. We have decided on Lambda Architecture since it satisfies our requirements while being an easy-to-implement, logical and contemporary architecture that is widely

applied in similar data-driven applications. The low-level implementation details and packages that will compose the Lambda Architecture will be explained in detail later throughout the report.

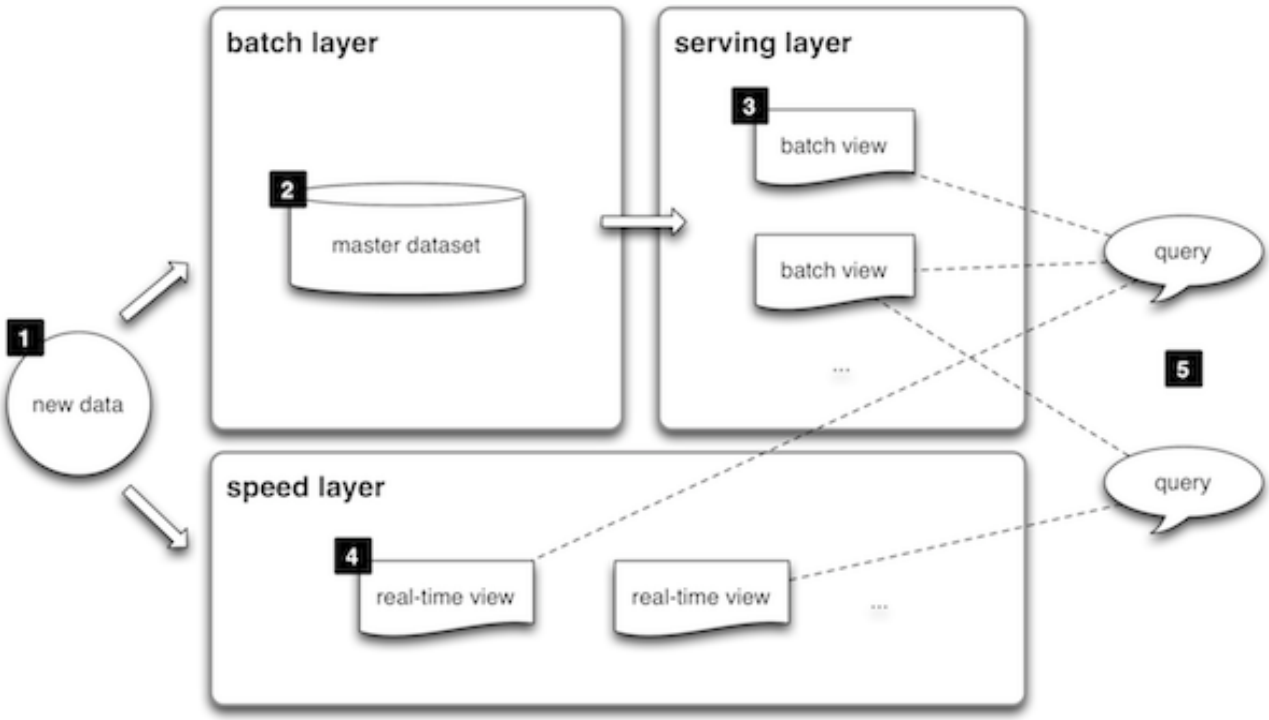

Figure: General implementation of Lambda Architecture. [1]

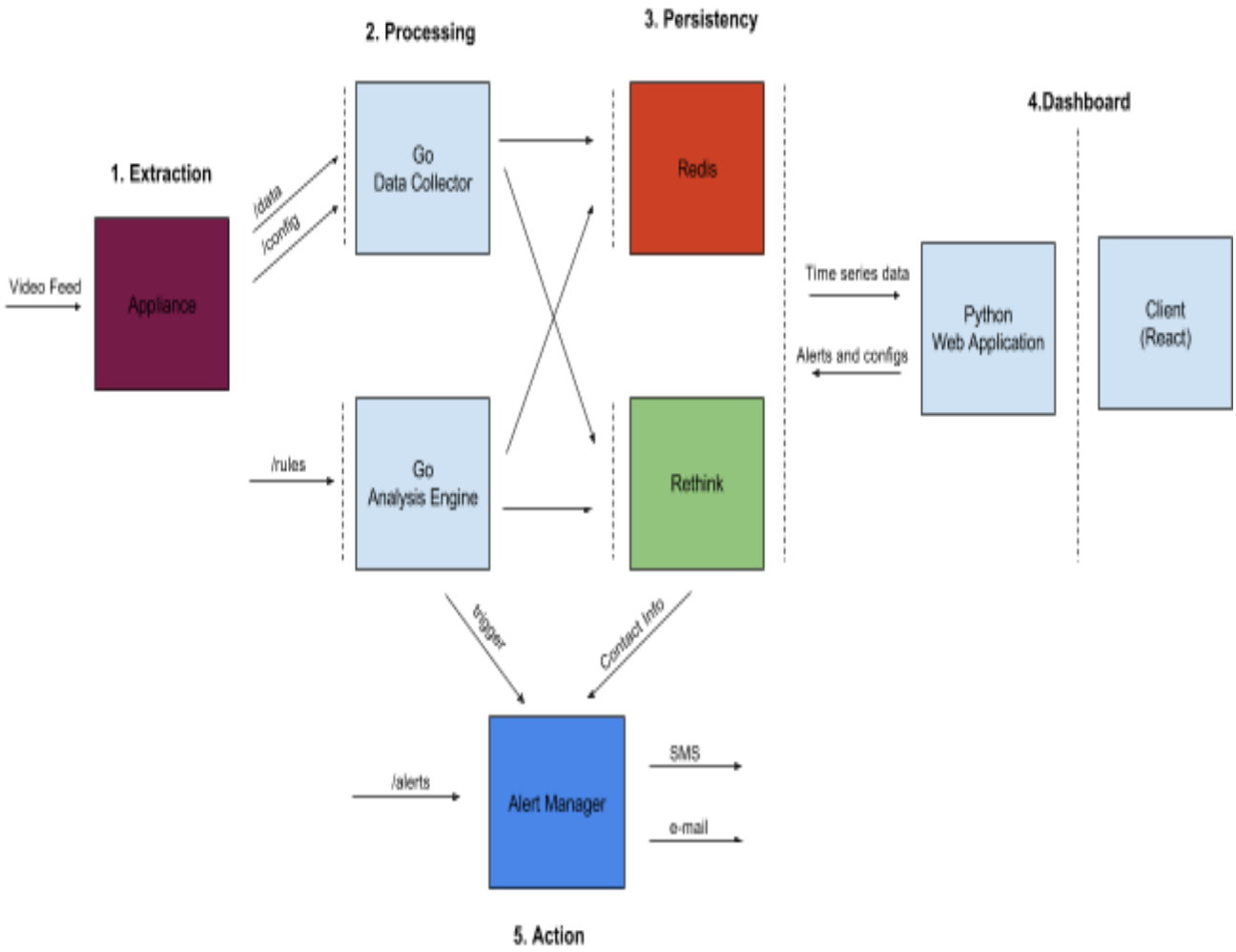

Figure: Lambda Architecture that will implemented for HORUS.

The Lambda Architecture of HORUS will be implemented using the Go programming language. (**Golang**) The main reasons that we have chosen Go over other conventional languages is that Go programming language has;

- Golang has a built-in robust and scalable threading architecture that is easy to program and use and it supports concurrency in language level. An example of this, language level support for concurrency is built-in race condition detection, which no other conventional language has. Since HORUS needs to process data in multiple threads while being scalable, this advantage of Golang will be crucial.

- It is a compiled language, and it compiles very fast. It has evolved from C by simplifying the syntax and removing the gimmicks of C, while still being a low-level systems language. Thus, its runtime performance is high compared to other languages such as Python.
- It has a simple and direct syntax. Also, it has a built-in static analyzer and code styling tools in its toolchain. Thus, writing and maintaining quality code that complies with coding standards is easier.
- Its language support for networking, client-server architectures (HTTP servers etc.) is higher than other languages such as Java.

On the client-side appliances, HORUS will not adapt any high level architecture, in order to keep it simple to implement and make suitable for low processing power hardware. Since the performance is the main concern here, we will adopt **C++** programming language to implement the image processing and communication components of the appliances.

The two most important packages of the HORUS is the Computer Vision package of the appliances, and Data Storage package of the server side, thus the main focus of the project will be on these two packages as they make the HORUS a novel technology that can be applied to industry.
In Computer Vision package, HORUS will make use of widely adopted, rich and de facto computer vision library, **OpenCV** and C++ interface of OpenCV. [2]

In Data Storage package, HORUS will use two main off-the-shelf solutions to implement the data storage layer in the Lambda Architecture that has two parts, Speed Layer and Batch Layer. For Speed Layer, that needs to handle the most recent streaming data

quickly, HORUS will deploy **Redis**. [3] Redis is an on-memory data storage structure that can be used as similar to a database. In HORUS, Redis will be used to analyze the most recent data collected from the appliances before any overhead from an actual, persistent database happens so that a low-latency, almost real-time analysis can be done on the data with high performance and throughput. For Batch Layer, HORUS will deploy a **NoSQL** JSON database, called **RethinkDB** as the data collected from appliances does not require a structured relational database. [4] RethinkDB is an open-source database that is built for reactive, real-time, scalable, data-driven applications, thus, we decided that it is a suitable persistent DB system for HORUS.

In this report, we will mostly focus on most imports structures of our low-level design and the details of off-the-shelf components, and the components that is not the primary focus of the system like the dashboard will be omitted for the sake of brevity.

1.1 Object Design Trade-offs

1.1.1 Scalability, Availability vs. Complexity and Maintainability

In HORUS, since the system needs to be scalable that can support huge data streams from appliances, and huge number of queries from the customers while maintaining almost real-time performance with high throughput, and being available at all times, we adopted a fairly complex architecture, Lambda Architecture, that can satisfy these requirements successfully.

However, by adopting the Lambda Architecture, we need to implement a highly complex system that is hard to develop and maintain, since it consists of multiple high performance layers with many off-the-shelf components.

1.1.2 Programmability vs. Adaptability, Compatibility

Since we already adopted a fairly complex architecture we sacrificed some programmability and ease of implementation. Since adaptability and compatibility are not our focus and not in our requirements with HORUS, we will trade-off some adaptability and compatibility in order to have increased our programmability and ease of implementation, by focusing our implementation on specific, predetermined hardware and software systems such as our DB systems.

1.1.3 Functionality, Developmental Resources vs. Appearance, User Experience

The main novel aspect of HORUS is to be a system that can help legacy sensor and indicator systems to adopt the concept of internet of things, cloud computing and data analytics. That is why, with HORUS we focus on Computer Vision, Data Storage and Data Analytics layers that builds the main functionality of HORUS.

However, because of the limited time and resources we only focus on novel functionality of HORUS, with the expense of the quality of appearance of the user interface, and the user experience provided with our dashboard. Thus, we only plan to implement a prototype system that can provide the main, novel functionality with a very basic dashboard and user interface that can be improved later to provide a better user experience.

1.2 Interface Documentation Guidelines

In the following sections interface documentation guideline will follow the below structure.

Package/Interface	Package/Interface name
Description	Short description of Package/Interface
Operation	Description of functions provided by the package

2.0 Packages

2.1 Appliance Packages

Packages under this subsection denotes the part of the project that will be deployed on the appliance.

2.1.1 Computer Vision

This is the most important part of the project. This package is mainly responsible for extracting the value from the analog indicator. This package will receive the video feed and will extract the current measurement from the screen. This package will only interact with the data transport layer with timestamp and extracted value used as input. We will be heavily using OpenCV image processing library in this package.

2.1.2 Data Transport and Authentication

This package is responsible for communicating the extracted value in a secure manner. It will use several established protocols to achieve security: **Oauth2** [5] will be used for authentication and **SSL** [6] will be used for securing transportation of the aforementioned data. Furthermore, it is important for this package to have fault recovery mechanism to mitigate intermittent network connection.

2.2 Server Packages

Packages under this subsection denotes the part of the project that will be a part of our cloud offerings.

2.2.1 Data Collection and Authentication

This package is the counterpart of the data transport package. It will mainly responsible for validating user input. It will sanitize input for storage layer and will refuse corrupted inputs. However, unlike most web applications, our request load, often calculated with **QPS** [7] metric, will have high throughput. Because of high throughput, we will use a **stateless server** [8] design and a load balancer to divide requests evenly between servers. Stateless servers will allow us to scale easily with respect to the data load.

2.2.2 Storage Layer

Storage layer will consist of two different overlapping layers which will fulfill reliability and performance requirements. For performance we will use Redis which will hold its contents in memory and for reliability we will use RethinkDB which has strong consistency semantics and easy to use sharding and replication. In analysis engine data from redis will be constantly processed in a stream processing fashion. In order to be reliable all the data will also be processed as batches with RethinkDB. This dual mode will allow us to be both reliable and performant at the same time.

2.2.3 Analysis Engine

Analysis engine is mainly responsible for triggering alerts based on results of the processing of the data. It will process the data according to user defined rules, which can be threshold based, anomaly based and behavioral based. It will proceed to trigger an alert if any of the conditions defined by the rules breaks. In order to process data real time and reliable, analysis engine will process recent data using Redis and will combine results from batch processing. Generated trigger will be passed to alert manager to notify user as defined by user itself.

2.2.4 Alert Manager

The reason behind separating the package alert manager is that it will mostly consist of glue code using third party libraries provided by several services. It will help us change the underlying implementation without changing analysis engine and will separate business logic contained in analysis engine from third party code. We will have mail notification system, using **Mandrill** [9] and SMS notification system using **Twilio** [10].

2.2.5 Dashboard

Dashboard will allow users to define alerts based on the customizable rules which are defined in analysis engine. We will be offering users a visual way to define the rules. Also we will offer real time plots for all the devices for manual consumption. We will be using Python for backend of the dashboard and **d3.js** [11] and **React** [12] for the frontend.

3.0 Interfaces

Package	Computer Vision
Description	Enables extracting information from indicator
Operation	**capture(): Image** *Captures image from camera attached to appliance using OpenCV library* **type func(Image): Image** *Function pointers that performs geometrical transformation of image captured to make it suitable for extracting data from image and pipes for other filtering functions* **type extractData(Image): float64** *Extract data from Transformed image and generate a float64 value corresponds to value of indicator*

Package	Storage Layer
Description	Enables organizing collection of data: that are extracted from indicators(points), rules defined by users to detect anomaly, and anomalies detected by Analysis Engine package.
Operation	**savePoint(point, device): bool** *Saves extracted information(point) to database.* **retrieveInterval(int64 start, int64 end, device): timeseries** *Returns timeseries(array of points and deviceID) that are extracted between timestamp start and timestamp end from database* **addRule(rule, alert): bool** *Adds rule and alert for it decided by user to database* **getRules(deviceID): []rule** *Returns array of rules for corresponding device* **getAnomalies(int64 start, int64 end, device): timeseries** *Returns timeseries which contains an array of extracted data from respective(device) indicator. Returned points are detected as anomalous by Analysis Engine.*

Package	Data Transporter
Description	Enables communication between appliance and server
Operation	**Authenticate(): bool** *Authentication* **sendData(float64 value, int64 timeStamp): bool** *Sends extracted point value and timestamp to server*

Package	DataCollector
Description	Initiates communication and collects data from appliances
Operation	**Authenticate(Credentials): bool** *Authentication* **ReceiveData(deviceId, float64 value, int64 timestamp): bool,** *Receives mesured point data from appliances* **HasConfigChanged(deviceId): bool** *Checks if there is change in configuration settings* **GetConfig(deviceId): bool** *Fetches the configuration settings*

Package	Analysis Engine
Description	Processes the collected data and initiates the alarm procedures
Operation	**MonitorRule(rule, deviceID): bool** *Registers rules on monitoring list to be applied on data collected on a device that is specified with* deviceID **DetectAnomalies(deviceID): timeSeries** *Returns timeSeries structure (array of point values) which contains sequence of point values that break assigned rules*

Package	Alert Manager
Description	Provides alerts and warning messages to users based on *Analysis Engine's* outputs.
Operation	**sendSMS(): bool** *Called by Analysis Engine to warn user via SMS* **sendEMail(): bool** *Called by Analysis Engine to warn user via E-Mail*

3.1 Main data structures of the system

- **Point**

```
struct point{
        timestamp int64
        value float64
        isAnomaly bool
}
```

- **Time Series**

```
Struct timeSeries{
        points []point
        deviceId int64
}
```

- **Device**

```
struct device{
        id int64
        config config
        lctn location
}
```

- **Alert**

```
struct alert{
        deviceID int64
```

```
    alert type enum{SMS, EMail}
}
```

- **Rule**

```
struct rule{
    type enum {threshold, anomaly, behavioral}
    Values []float64
    rulef func( timeSeries ): bool
}
```

- **Config**

```
struct config{
    recordInterval float64
    deviceID int64
    visionParameters []float64
    powerStatus bool
}
```

- **Location**

```
struct location{
    latitude float64
    longitude float64
}
```

4.0 References

1. http://lambda-architecture.net/
2. http://opencv.org/
3. http://redis.io/
4. https://www.rethinkdb.com/
5. https://oauth.net/2/
6. https://www.globalsign.com/en/ssl-information-center/what-is-ssl/
7. https://en.wikipedia.org/wiki/Queries_per_second
8. http://orca.st.usm.edu/~seyfarth/network_pgm/net-6-3-3.html
9. https://www.mandrill.com/
10. https://www.twilio.com/
11. https://d3js.org/
12. https://facebook.github.io/react/