Stream Processing on IoT Devices using Calvin Framework

by

Ameya Nayak

A Project Report Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Supervised by

Dr. Peizhao Hu

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences Rochester Institute of Technology Rochester, New York

05 2017
I would like to thank Professor Peizhao Hu for suggesting the idea of this project and mentoring and motivating me to complete all the deliverables. I would like to thank my parents for supporting and motivating me throughout my masters education. Lastly I would like to thank the students in Capstone research group for suggestions helpful in achieving the deliverables of the project.
Abstract

Stream Processing on IoT Devices using Calvin Framework

Ameya Nayak

Supervising Professor: Dr. Peizhao Hu

IoT devices tend to generate a lot of data, most of which is sent to the cloud for storing, processing and analyzing. This approach has two problems associated with it which are dependency on the Internet and additional costs for using cloud services. To avoid these problems we use an open source lightweight framework 'Calvin' which can be used for message passing and data processing among distributed IoT devices and design a pipelining framework architecture for stream processing. Using this design we have built applications to simulate turning the Heater on/off and updating the window blinds status to open, close or half open. We evaluate the performance of these applications with respect to time and memory consumed and make conclusions based on the evaluations.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>ii</td>
</tr>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td><strong>1 Introduction and Motivation</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Apache Spark</td>
<td>2</td>
</tr>
<tr>
<td>1.2.1 Apache Spark - Overview</td>
<td>2</td>
</tr>
<tr>
<td>1.2.2 Why not Spark</td>
<td>3</td>
</tr>
<tr>
<td>1.2.3 Learnings from Spark</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Calvin</td>
<td>4</td>
</tr>
<tr>
<td>1.3.1 Calvin Overview</td>
<td>4</td>
</tr>
<tr>
<td>1.3.2 Calvin Building Blocks</td>
<td>5</td>
</tr>
<tr>
<td>1.3.3 Calvin Onboarding</td>
<td>5</td>
</tr>
<tr>
<td><strong>2 Design</strong></td>
<td>7</td>
</tr>
<tr>
<td>2.1 Stream Processing</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Distributed Machine Learning</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Fault Tolerance</td>
<td>8</td>
</tr>
<tr>
<td>2.4 Architecture</td>
<td>8</td>
</tr>
<tr>
<td><strong>3 Implementation</strong></td>
<td>9</td>
</tr>
<tr>
<td>3.1 AC/Heater</td>
<td>9</td>
</tr>
<tr>
<td>3.1.1 Prediction algorithm in a single actor</td>
<td>10</td>
</tr>
<tr>
<td>3.1.2 Prediction algorithm in a multiple actors</td>
<td>12</td>
</tr>
<tr>
<td>3.2 Window Blinds</td>
<td>14</td>
</tr>
<tr>
<td>3.2.1 Prediction algorithm in a single actor</td>
<td>15</td>
</tr>
<tr>
<td>3.2.2 Prediction algorithm in multiple actors</td>
<td>17</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Spark - Architecture .................................................. 3
1.2 Calvin Architecture .................................................. 4
1.3 Calvin Onboarding Example App ................................. 6

2.1 Architecture of the stream processing framework ............ 8

3.1 Data flow diagram for multiple runtimes ..................... 10
3.2 Left: Before Deployment Right: Output ...................... 10
3.3 Multiple Runtimes before deployment .......................... 11
3.4 Output .............................................................. 11
3.5 Data flow diagram for multiple runtimes ..................... 12
3.6 Left: Before Deployment Right: Output ...................... 13
3.7 Multiple Runtimes before deployment .......................... 13
3.8 Output .............................................................. 14
3.9 Data flow diagram for multiple runtimes ..................... 15
3.10 Left: Before Deployment Right: Output ...................... 15
3.11 Multiple Runtimes before deployment .......................... 16
3.12 Output .............................................................. 16
3.13 Data flow diagram for multiple runtimes ..................... 17
3.14 Left: Before Deployment Right: Output ...................... 17
3.15 Multiple Runtimes before deployment .......................... 18
3.16 Output .............................................................. 18

4.1 Time utilization- Heater/AC ....................................... 19
4.2 Time Utilization- Heater/AC ...................................... 20
4.3 Time utilization-Window blinds .................................. 20
4.4 Memory Utilization-Window Blinds .............................. 21
Chapter 1

Introduction and Motivation

1.1 Motivation

Since IoT devices have low memory and disk space, the data collected from these devices is pushed to the cloud for data processing and analysis. Some drawbacks of performing these tasks are as follows

- If the internet is down, neither data can be pushed to the cloud nor can the IoT devices talk to the cloud services to make important decisions
- Storing the data and performing data analysis on the cloud involves additional cost.

To counter these problems we need to implement a data processing solution running on IoT devices. Most of the big data processing frameworks like Apache Spark would be an overkill since it requires a lot of memory and disk space. This motivated us to use a lightweight framework called Calvin for data processing and analysis on the distributed IoT devices which can run on such devices with low memory and disk space. Also since most of the data generated by IoT devices is stream data so we design a framework which would help automate data flow and serve as a pipeline for parallel stream data processing.
1.2 Apache Spark

1.2.1 Apache Spark - Overview

Spark is an open source large scale distributed system used for data processing and analytics. It is generally used for batch or stream processing [5]. It allows in memory processing which makes it run faster as compared to most other big data processing tools. Apache Spark can be made to run on a standalone cluster, mesos, hadoop, ec2 like clusters on cloud. It can read data from a variety of sources like Cassandra, HBase, Amazon S3, Hadoop File System etc. Spark supports multiple programming languages to build applications like java, scala, python and R. It has built in, read to use APIs which makes it easy to develop applications. it is Highly scalable and fault tolerant [3]. Apache Spark features include

- Spark SQL: to support relational queries and structured/semi-structured data
- Spark Streaming: to support processing of stream data
- MLib: To support ready to use machine learning algorithms for data analysis
- GraphX: To support graph processing
- MLib Pipelines: It provides API for the user to build and optimize pipelines for Machine Learning algorithms.
- RDD: RDD stands for Resilient distributed datasets. It is a fundamental data structure of Spark which is a collection of distributed objects. Its properties includes
  1. Immutable and read-only
  2. Fault Tolerant
  3. Allows in memory processing
  4. Can be processed in parallel

Apache Mesos, a cluster operating system acts as the cluster manager which manages assignment of tasks to nodes. The driver program creates and issues jobs [4].
1.2.2 Why not Spark

- Spark needs a lot of disk space for installation and data processing and IoT devices generally have low disk space

- Spark processes requires a lot of memory for running processes (recommended is 8 GB) and IoT devices are low on memory

1.2.3 Learnings from Spark

- Stream Processing: Convert the stream to batches for processing

- Machine Learning: Break the machine learning algorithm into subcomponents so that work is distributed among multiple nodes

- Fault tolerance: The framework should allow data migration in case of node failures
1.3 Calvin

1.3.1 Calvin Overview

Calvin was developed by Ericsson Research and open sourced in 2015. Calvin combines ideas from Actor and Flow Based Programming models to provide a framework to build distributed applications for IoT devices [6]. Benefits of using Calvin are as follows

- Requires low memory and disk space
- Supports Python
- Users can leverage inbuilt methods for data migration from one device to another in case of node failures
- Users can specify backup nodes in case if the current node fails.
- Allows users to distribute workload among different nodes.

Figure 1.2: Calvin Architecture
Developing a Calvin Application involves 4 phases

- **Describe**: Implementing individual components of the application, that is,

- **Connect**: These components interact with each other in the form of a graph

- **Deploy**: Instantiating the application as a graph

- **Manage**: Automating allocation of resources to the components during the lifecycle of the application

### 1.3.2 Calvin Building Blocks

- **Calvin Actors**: Calvin Actors are essentially Python scripts which follow a pre-defined convention. These actors can receive data from other actors, process data and send data to other actors [1].

- **Calvin script**: A calvin script is where the application is defined in terms of a graph which describes the data flow among different actors. It has essentially has 2 parts. The first part is describing the nodes of a graph. You can do that as follows. The second part is describing the edges in the graph.

- **Calvin Runtime**: Calvin Runtime is an execution environment provided as a part of the Calvin framework. It is used to deploy the application and run the actors.

### 1.3.3 Calvin Onboarding

- **Calvin Actor**: You need to import Actor and decorators from Calvin.actor.actor. Every Script should be a class and it must inherit Actor. If you need to initialize attributes it needs to go in the init() method. Within a Calvin script, you can either implement a calvin method which can take either no or multiple inputs and provide no or multiple outputs or helper methods which are used by these calvin methods. The order of execution is mentioned at the end under action_priority. Any calvin system libraries required goes after the action_priority [2].
• Decorators: Decorators are like annotations in Java which in this case are used while implementing actors. Various Decorators are 1. `@manage` which tells the Calvin framework which attributes to manage while node failures and migrations of Calvin runtime. This is to be done before the init method. 2. `@stateguard` to specify conditions on whether to execute a particular Calvin method or not. 3. `@condition` to specify the input and output variables here, if any.

• Calvin Script: A Calvin script can be considered as a graph which describes the data flow among different actors. It has essentially has 2 parts. The first part is describing the nodes of a graph. You can do that as NodeName : Namespace.Actor(parameters can be 0 or multiple). The second part is describing the edges in the graph. This can be done as follows Node1.output -> Node2.input

• `deployjson` To run the application as a distributed application involving multiple nodes we need to create a json which maps the nodes to runtimes.

![Figure 1.3: Calvin Onboarding Example App](image_url)
Chapter 2

Design

Pre-processing data and machine learning algorithm may be implemented as a single or multiple actors. Once the actor pushes the output data it is ready for processing the next input token even though the data has not been processed the entire framework. To achieve distributed functionality, the mapping of calvin actors to calvin runtimes has to be specified in a .deployjson file Each runtime runs on a different device in the cluster

2.1 Stream Processing

- Since the main objective is stream processing of data on IoT devices, we need to focus first on how do we handle continuous flowing stream data

- We take into consideration the learning from Spark’s handling of stream data. Thus, to handle stream data, we need to create an actor which would take the stream data as an input, convert it into batches and provide it as an output to the next actor in the data-flow graph

2.2 Distributed Machine Learning

- We need to make sure that the computation tasks are distributed among different nodes. To do that we need to divide the machine learning algorithm into subcomponents and implement these subcomponents as Calvin actors.
• To do that we need to encapsulate one or more actors based on the actor to runtimes ratio.

2.3 Fault Tolerance

• In case of network of IoT devices, it is quite possible some of the devices fail and need to reboot. In that case we must make sure that the framework handles node failures.

• To do that we need to make sure that data migrates safely from one runtime to the backup runtime.

2.4 Architecture

![Diagram of the stream processing framework](image)

Figure 2.1: Architecture of the stream processing framework
Chapter 3

Implementation

Based on the design and architecture discussed in the previous section, we have created two applications

3.1 AC/Heater

We create an application to simulate turning the centralized AC/Heater ‘ON’/‘OFF’ based on temperatures from 3 rooms. To implement this, we do the following

- We create an actor to simulate the sensors emitting temperature data as streams. This actor generates random data every second between 0 to 100 and sends it as the output to the next actor

- The next actor in the data flow pipeline converts the incoming stream data into batches

- The next actor takes batches of data as input and computes the average of the batch.

- We create 3 instances of the above 3 actors so that it resembles sensors in 3 rooms

- The output of these 3 instances is then fed to an actor which takes an average of the 3 incoming values and then outputs to the next actor
3.1.1 Prediction algorithm in a single actor

- We have created actors which implement the ready to use Naive Bayesian, Decision Tree and Support Vector Machine (SVM) provided by the scikit-learning library.
- The following diagram represents the entire data flow of the application.

![Data flow diagram for multiple runtimes](image1)

Figure 3.1: Data flow diagram for multiple runtimes

![Code snippet](image2)

Figure 3.2: Left: Before Deployment Right: Output
Figure 3.3: Multiple Runtimes before deployment

```
{"temperature": "48.8883618993"}
{"AC": "[0]"}
{"Heat": "[1]"}
```

```
{"temperature": "41.0617356325"}
{"AC": "[0]"}
{"Heat": "[1]"}
```

Figure 3.4: Output
3.1.2 Prediction algorithm in a multiple actors

We divide the Naive Bayesian algorithm into 4 sub components which are as follows:

- The first actor in the pipeline creates 2 separate sets one for training and other for validation.
- The next actor, the training data is mapped according to the class values (0 for Off and 1 for On).
- The next actor is then responsible to generate mean and std deviation for each class.
- The final actor the calculates class probability and predicts whether the AC should be turned on or off.

Figure 3.5: Data flow diagram for multiple runtimes
Figure 3.6: Left: Before Deployment Right: Output

Figure 3.7: Multiple Runtimes before deployment
3.2 Window Blinds

We create an application to simulate turning the Window Blinds ‘Open’/‘Closed’/‘Half Open’ based on temperatures from 3 rooms. To implement this, we do the following:

1. In the first sequence of actors we create an actor to simulate the sensors emitting temperature data as streams. This actor generates random data every second between 0 to 100 and sends it as the output to the next actor.

2. The next actor in the data flow pipeline converts the incoming stream data into batches.

3. The next actor takes batches of data as input and computes the average of the batch.

4. In the second sequence we create an actor to simulate the sensor emitting light intensity values as streams. This actor generates random data every second between 0 to 1 and sends it as the output to the next actor.

5. The next actor in the data flow pipeline converts the incoming stream data into batches.
• The next actor then computes the average of the batch and sends it for further processing

3.2.1 Prediction algorithm in a single actor

Figure 3.9: Data flow diagram for multiple runtimes

Figure 3.10: Left: Before Deployment Right: Output
Figure 3.11: Multiple Runtimes before deployment

Figure 3.12: Output
3.2.2 Prediction algorithm in multiple actors

Figure 3.13: Data flow diagram for multiple runtimes

Figure 3.14: Left: Before Deployment Right: Output
Figure 3.15: Multiple Runtimes before deployment

Figure 3.16: Output
Chapter 4

Analysis and Results

4.1 Heater/AC

<table>
<thead>
<tr>
<th>Number of nodes in the cluster</th>
<th>Naive Bayesian implemented using Single Actor (in seconds)</th>
<th>Naive Bayesian implemented using Multiple Actor(s) (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Node</td>
<td>Training: 65-70 P0: 0-2</td>
<td>Training: 70-80 P0: 0-3</td>
</tr>
<tr>
<td>5 Nodes</td>
<td>Training: 70-75 P0: 0-3</td>
<td>Training: 80-90 P0: 0-4</td>
</tr>
</tbody>
</table>

Figure 4.1: Time utilization- Heater/AC

- Running the framework using multiple nodes as compared to running the framework on single node involves an additional message passing overhead from one actor to another thereby increasing the time required to run the application

- Running the machine learning algorithm using multiple nodes as compared to running the framework on single node involves an additional message passing overhead from one actor to another thereby increasing the time required to run the application
Running the machine learning algorithm using multiple nodes distributes the computation load on each node thereby reducing memory utilization per node as compared to running the framework on single node.

Running the framework using multiple nodes distributes the computation load on each node thereby reducing memory utilization per node as compared to running the framework on single node.

### 4.2 Window blind

<table>
<thead>
<tr>
<th>Number of nodes in the cluster</th>
<th>Naive Bayesian implemented on a Single actor (in MB)</th>
<th>Naive Bayesian implemented using Multiple Actors (in MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Node</td>
<td>97 - 105</td>
<td>95 - 100</td>
</tr>
<tr>
<td>5 Nodes</td>
<td>95 - 101</td>
<td>93 - 97</td>
</tr>
</tbody>
</table>

Running the framework using multiple nodes as compared to running the framework on single node involves an additional message passing overhead from one actor to another thereby increasing the time required to run the application.
• Running the machine learning algorithm using multiple nodes as compared to running the framework on single node involves an additional message passing overhead from one actor to another thereby increasing the time required to run the application.

<table>
<thead>
<tr>
<th>Number of nodes in the cluster</th>
<th>Naïve Bayesian implemented on a Single actor (in MB)</th>
<th>Naïve Bayesian implemented using Multiple Actors (in MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Node</td>
<td>97 - 105</td>
<td>95 - 100</td>
</tr>
<tr>
<td>5 Nodes</td>
<td>95 - 101</td>
<td>93 - 97</td>
</tr>
</tbody>
</table>

Figure 4.4: Memory Utilization-Window Blinds

• Running the machine learning algorithm using multiple nodes distributes the computation load on each node thereby reducing memory utilization per node as compared to running the framework on single node.

• Running the framework using multiple nodes distributes the computation load on each node thereby reducing memory utilization per node as compared to running the framework on single node.
Chapter 5

Conclusions

5.1 Current Status

- For both the applications we see that the memory footprint per node is reduced when we run the application using multiple nodes as compared to a single node.

- For both the applications we see that the time needed for prediction is increased when we run the application using multiple nodes as compared to a single node.

- Thus we may say that this framework can be utilized for applications with less memory intensive computations which involve crucial computations so that dependency on the internet is reduced.

5.2 Drawback and Challenges

- Does not support programming languages other than python.

- Since all the Calvin runtimes need to be configured before deployment, achieving auto-scalability is difficult.

- Since Calvin was recently open-sourced, the Calvin user community is sparse and except for the wiki, there aren't many tutorials around.

- A lot of time you might come across unexplained errors for example, the application working on one device but not working on other.
• Debugging the code in a Calvin Actor is difficult since the error messages are most of the time vague and not at all helpful

5.3 Future Work

• Testing the framework with actual sensors

• Implement more generic actors which can be used as in-built actors for implementing various applications

• Dividing other machine learning algorithms into subcomponents and implementing these subcomponents on individual actors, thus allowing users to implement such algorithms on single or multiple runtimes

• Making the applications more robust and try to add auto-scaling

• Integrating with other ongoing projects under Professor Peizhao Hu for developing a full fledged IoT application
Bibliography


