Abstract

IoT devices are becoming part of our daily life. These devices are everywhere from our home to the corporate world and even in grocery stores. These small devices are generating a tremendous amount of data in real time. Currently, the data generated by these devices are sent to cloud for machine learning and business analytics. Today these devices are quite powerful in terms of processing capabilities and the amount of resources they have. We can utilize these resources to do some machine learning within the network of IoT devices. A lot of research is going on to effectively utilizes the computational power of these IoT devices and do initial processing of data within these devices. Many frameworks have also been developed or under research so that developers of IoT devices can leverage these frameworks to build applications on top
of these frameworks. All the complexities behind talking to the underlying hardware or the protocols to communicate between IoT devices is taken care by these frameworks. Calvin is a framework which is an open source project developed by Ericsson research team that can be used to create applications on IoT devices. This paper focuses on integrating the capabilities of Calvin into the IoTX project and to incorporate a Caching layer in the IoTX project to improve performance of the system as a whole.
Acknowledgments

I would like to thank Dr. Peizhao Hu for guiding me throughout the process of my project development and would like to thank Dr. Hans-Peter Bischof for mentoring us for colloquium. I would also like to thank my peers who are working on IoTX project for their help in understanding and helping in overall development of the IoTX project. A special thanks to my wife for supporting me through tough times and believing in me. Lastly I would like to thank my parents for supporting all my decisions in my life.
# Contents

1 Introduction 1

1.1 Stream Processing Frameworks 2

1.1.1 Apache Spark 2

1.1.2 Apache Flink 3

1.1.3 Drawbacks Of Using Conventional Stream Processing Framework In IoT devices 4

1.1.4 Calvin Framework 4

1.1.5 Calvin Architecture 5

2 IoTX Framework 11

3 Data Caching 13

3.0.1 Introduction 13

3.0.2 Data Caching LRU 14

3.0.3 Data caching data structures 14

3.0.4 Data caching algorithm 15

3.0.5 Caching Implementation 15

3.0.6 Data Caching Use Case 20
CONTENTS

3.0.7 Advantages Of LRU Caching .............................................. 20

3.1 Graph showing the system performance relative to the cache size 21

4 Calvin Applications ............................................................. 23

4.0.1 Calvin Applications Using System Actors ......................... 23

4.0.2 Calvin Application Using Custom Actors ......................... 25

4.0.3 Calvin Script And Calvin Application ................................. 26

4.0.4 Steps To Deploy Calvin Application ................................. 28

4.0.5 Advantages Of Calvin Framework ..................................... 29

5 Calvin Actor Model in IoTX Project .......................................... 30

5.0.1 Deploying Application from IoTX Framework ...................... 31

6 Docker And Calvin Control APIs .............................................. 33

6.0.1 Steps Required To Start The Docker ................................. 34

6.0.2 Calvin Control APIs ....................................................... 35

7 Conclusion ........................................................................... 37

7.0.1 Future Work ................................................................. 37
# List of Figures

1.1 Calvin Architecture .......................... 5
1.2 Actor With Multiple Actions [6] .................. 7
2.1 IoTX Framework ............................... 12
3.1 Cache of size four with unique keys k1 to k4 each referring to
an object in doubly linked-list ..................... 15
3.2 Object referred by K3 is accessed ................ 16
3.3 N2 is the object referred by K3 .................. 16
3.4 Move N2 to front of the queue as it is recently accessed .... 17
3.5 N1 will be the second most recently accessed element now ... 17
3.6 Object N5 arrives, Cache is full .................. 18
3.7 Remove the tail element from the queue .............. 19
3.8 Add N5 to the front of the queue ................. 19
3.9 Data caching plot .............................. 21
4.1 Calvin Application Using System Actors [1] ............ 24
4.2 Calvin Application Using Custom Actors .............. 25
LIST OF FIGURES

4.3 Calvin Application Code For Custom Actors .................. 26
4.4 Calvin Script And Application .............................. 27
5.1 REST Post Message To Deploy Application .................. 32
7.1 Calvin Process Flow Model ................................. 39
List of Tables

3.1 **Below table shows how the execution of graph creation in IoTX project varies with different cache size. As cache size is increased the IoTX graph is created in less time. The cache size is a percentage of the total JVM size allocated to the IoTX project.** 22
Chapter 1

Introduction

IoT devices are getting integrated into our lives. We are dependent on these devices so much that we cannot think about a day without these devices. The smart phone is one of the examples of such device. Today many companies are building their products to make our home smarter and thus making our lives more comfortable. But the question is how the companies are able to achieve this? The answer is the tremendous amount of data generated by these IoT devices. Companies collect these data in their cloud services and apply machine learning algorithms and business analytics on this data. This is how a company is better able to understand a customer, what are the needs, preferences, likes, and dislikes of a consumer. Knowing such a minute details of a customer helps companies provide a better or smarter service to the customer. The motivation of this project is to perform some of the computations or machine learning with in the network of these IoT devices by utilizing the computational resources available in these devices.
1.1 Stream Processing Frameworks

There are two very powerful stream processing engines available today in the market.

1.1.1 Apache Spark

Spark is an open source project managed by Apache Software Foundation. Spark is a very powerful framework and has one of the largest open source community of software developers. Spark works on an abstraction called RDD (Resilient Distributed Dataset). The incoming stream of data is divided into a micro batch interval of data known as RDD [4]. This small unit of data is then sent for processing in the Spark cluster. The computation on these RDDs occurs in a parallel distributed system which makes Spark processing lightning fast. The RDDs are immutable which makes them easier to store in RAM. For iterative machine learning algorithms where previous data is needed for next iteration, saving RDDs on RAM makes it a perfect use case. The data retrieval from RAM is very fast as compare to disc reads. Spark uses this feature very efficiently because of which it is ideal for running machine learning algorithms. The failure recovery mechanism in Spark is very fast as Spark maintains the DAG (Directed Acyclic Graph) in the master node. This DAG keeps track of the RDD lineage [8]. It knows how to create an RDD if it is lost due to system failure. As recomputing RDD also occur in parallel, the fault recovery process is very fast and efficient. All these features of Spark make it a very good fit for distributed data processing on a network of IoT devices. But is it good enough? We will talk about it after we discuss another
alternative for stream processing.

1.1.2 Apache Flink

Flink is an open source project managed by Apache Software Foundation. Flink, just like Spark is a stream processing framework which is getting a lot of popularity these days. So what makes Flink different from Spark. The difference is the abstraction. Flink process data as streams rather than batch or micro batch like Spark. To Flink, every data can be realized as unbounded data and can be processed by a continuous stream of operators in a pipeline fashion [3]. Today realizing data as streams is becoming very popular as it does not distinguish between bounded and unbounded data. In a streaming world, every data is treated as streams. Flink takes the advantage of this and processes the data as it arrives at a Flink cluster. You can imagine that Flink will run faster than Spark as it does not wait for micro batches to form. The stream flows between the operators which do operations on running stream. These operators can be distributed across the cluster and can run in parallel. The fault recovery mechanism of Flink is based on saving the global snapshot of the system using Chandy Lamport global snapshot mechanism. The watermarks which act as a checkpoint runs along with the stream and when they reach the operators the Snapshot is saved in a reliable storage system. This is a very light weight process that does not impact the system performance. When ever a system fails, the saved state of the system is read again and the computation is restarted. To start the computation again the stream needs to be restarted again to the point it failed. This is achieved using Kafka. Flink relies on Kafka or other messaging queuing service to replay the
stream again. All these features make Flink a reliable fast stream processing platform. Flink comes with an in-built machine learning library which can be used to perform machine learning algorithms.

1.1.3 Drawbacks Of Using Conventional Stream Processing Framework In IoT devices

Both Spark and Flink are very powerful stream processing frameworks and can be used for machine learning or other data processing in a distributed cluster of machines. The only disadvantage of these frameworks is they need a considerable amount of memory for their installation. These frameworks when run takes a lot of memory to execute. Consider Spark only, it is based on in memory computation. For this to happen it needs a considerable memory to be efficient. These frameworks are good for desktop based application but the IoT devices have a limited amount of memory and CPU. Using these frameworks on IoT devices would neither be ideal nor it will give the required performance that is expected out of them.

1.1.4 Calvin Framework

A group of engineers in Ericsson Research team has been working on developing a framework called Calvin for distributed data processing using a network of IoT devices. Calvin platform allows IoT devices to communicate with each other. It is a framework that hides all the complexities to talk to hardware and the protocols used to communicate between IoT devices. The developers need to only focus on developing applications rather than other complexities involved. Also, Calvin is a light weight framework that can be installed in
resource constrained IoT devices.

1.1.5 Calvin Architecture

Calvin Architecture is composed of many components as shown in Fig.1.1
Below is the description of each component of Calvin Framework:

Calvin Runtime

Calvin Runtime is a container inside which the Calvin Application runs [5].
This layer is responsible for communicating with the underlying hardware
resources and also incorporates the protocols to communicate with other IoT
devices where Calvin application is running [5]. The runtime also manages
the resources like memory usages by the applications built on top of Calvin.

Figure 1.1: Calvin Architecture
CHAPTER 1. INTRODUCTION

Calvin System Actors

Calvin works on the principle of actor model, where each actor is a small reusable computational unit that is responsible for executing a particular task when data arrive at its input port. An actor has input ports, output ports, action, and requirements. Data arrives at an input port of an actor. The actor performs some action on the incoming data and sends the result to its output ports. There are only two conditions when an action is performed by the actor, either an external event triggering an actor or an actor has some data on its input port [6]. An actor can also maintain a state [7]. Actors run within a runtime and can be told to migrate to a different runtimes [7]. In the migration process the actor takes its saved state along with it to the new runtime it is being migrated. The requirements are rules that can be defined to force actors with some constraints. For example, an actor can be told to run on a particular runtime or in a particular building of an organization.
Calvin User Defined Actors

Calvin is a powerful platform and has provided a feature where an application developer can define its custom actor. This is required because not everything can be done using system actors. The system actor store is growing each day but there may be a special use case for an application where user needs to have their own custom actors to perform specific task. Application developer can write custom actors in python and write a configuration file to tell the Calvin framework where these custom actors are located. When an application is deployed, the Calvin framework looks at the location defined in the configuration file and finds the user defined actors. User defined actors

---

Figure 1.2: Actor With Multiple Actions [6]
CHAPTER 1. INTRODUCTION

runs as if they are system actors and can communicate with any actors in the system.

Calvin Tools

Calvin Framework provides inbuilt tools to communicate with the Calvin framework and Calvin runtimes. Below are some of the important and commonly used tools along with their description:

- **csdocs** csdocs is a command line tool which can be used to know about a specific command. It provides a documentation help regarding each command. csdocs can also be used to get the documentation about the actors in the system [2].

- **csruntime** csruntime is a command line tool that can be used to start one or more runtimes on a particular node [2].

  csruntime --host <address> --port <port> --controlport <controlport>

  If port or control port is not specified by the user, the system adds default port and control port value as 5000 and 5001.

- **cscontrol** cscontrol is a command line tool which is used to control a runtime and also to deploy an application. Internally cscontrol uses rest API's to communicate with the runtimes [2]. This command can also be used to define a topology of the application which means adding peers to a node telling it who its neighbors are. The command used to compile and deploy a Calvin application is:

  cscontrol http://<address>:<controlport> deploy <calvinscript file>
• **csweb** csweb is a web based visualization tool. Using this tool an application developer can have a visual of the application running on the runtimes [2]. There are many functionalities provided by the UI. For example deploying an application, destroying an application, looking into the in and out ports of a particular actor to know what data is flowing in the application.

• **csviz** csviz tool can be used to visualize an application as a distributed data flow application [2]. It is a very helpful tool to see how the application is connected logically.

• **csmanage** An application developer can define a reusable component which is a combination of different actors to perform a required task. The reusable component is a very powerful feature of the Calvin framework as it allows to define various reusable components in the system which can be used by the different application. The csmanage tool helps to install these components or custom actors to the actor store [2].

  csmanage install component --namespace <name> --script <calvinscript> --component <componentname>

• **cscompile** This tool is used to compile the Calvin Script file [2]. It does the error checking and syntax checking before converting it to a JSON formatted file.
Calvin Application

Calvin Application is formed by using a set of actors and creating a distributed data flow graph in which each node is an actor. To deploy a Calvin Application two bare minimum files are necessary, the Calvin Script file which composed of actors and how data flows between those actors and the requirement file which are a certain set of rules defined to implement constraints on the actors. Calvin Applications are deployed using the following command:

    cscontrol <host IP> deploy --reqs <requirements file> <calvin script>
Chapter 2

IoTX Framework

IoTX framework aims at serving as an "operating system" to allow user to develop custom applications and to bring some of the data analytics to the IoT devices. This project also provides user a real-time simulation environment to simulate the behavior of IoT devices and to understand how the network of these devices will behave when placed in an actual environment. In summary, the project can be used to study the behavior the devices before actual deployment. The framework consists of various components like the front-end where a user can visually see the IoT nodes, their capabilities, the sensors they have etc. All these information is stored in the back-end database. This framework is based on graph architecture where each node in the graph is an IoT device and connection between the graph is the communication link between the devices. The system also includes the neo4J graph database to store a specific simulation. The user can traverse through the timeline provided in the UI to see simulation changes at each time interval. The graph database has many advantages for future use as it incorporates Graph algorithms. The IoTX
framework is the top most layer that provides application developers to create custom applications and to manage and deploy these applications dynamically. It also aims at serving as an efficient resource manager/scheduler to manage and assign tasks to the IoT nodes depending on the resource availability.
Chapter 3

Data Caching

3.0.1 Introduction

The IoTX project generates a random graph initially for the user. All the information regarding the IoT nodes like the sensors, actuators, capabilities, communication channel link information etc. are stored in the back-end database. Each time during the random graph generation or when a user clicks on an IoT node from the graphical user interface, a request is sent to the back-end to pull the information regarding the selected IoT node and send it back to the front end. Only the basic information about the IoT node (GUID) is stored in the program. Querying the database frequently to access the same information is not very efficient. The performance of the system could be improved if we have some data caching layer in-between the back-end and front-end.
3.0.2 Data Caching LRU

The least recently used data caching algorithm has been incorporated into the IoTX project. This algorithm is based on removing the least recently used object first when the cache is full. The cache is implemented to hold objects of any type because in IoTX project there are many types of objects that are stored in the database for example sensors, actuators, links etc. To utilize the cache effectively it makes sense to store all these types of objects in the cache whenever a user requests them. The cache size is based on the percentage of the total JVM memory allocated to the IoTX project, which means we can configure the cache size depending on how much data we want to store in cache and keep the rest of the memory for the program to run.

3.0.3 Data caching data structures

- As stated above the LRU caches is based on removing the element or object which is least recently used.

- Elements in the cache is stored in a doubly linked-list data structure.

- A HashMap is created which has a key as the object id and value as a reference to the element in the linked-list.

- The combination of HashMap and Linked-List keeps the time complexity of the cache operations as constant.

- The cache needs to be as fast as possible which is why we need constant time operations.
3.0.4 Data caching algorithm

- When an object is referenced in the IoTX project, the algorithm looks up in the cache.

- If the object is found in the cache, move this object to the front of the linked-list.

- If the requested object is not found in the cache, query the database and bring it to the memory and then add it to the front of the linked-list.

- If the cache is full, remove the last object from the linked-list as it would be the object that was least recently used.

3.0.5 Caching Implementation

Figure 3.1: Cache of size four with unique keys k1 to k4 each referring to an object in doubly linked-list
CHAPTER 3. DATA CACHING

Figure 3.2: Object referred by K3 is accessed

Figure 3.3: N2 is the object referred by K3
Figure 3.4: Move N2 to front of the queue as it is recently accessed

Figure 3.5: N1 will be the second most recently accessed element now
Figure 3.6: Object N5 arrives, Cache is full
Figure 3.7: Remove the tail element from the queue

Figure 3.8: Add N5 to the front of the queue
3.0.6 Data Caching Use Case

- When an IoTX project is launched, a random graph is generated. Data caching will improve the launch time of the IoTX project.

- For simulation, a user can use the front-end UI to access the nodes from the database. The caching will improve the user experience as retrieving the requested objects will be faster.

- There are various in-built algorithms implemented in the IoTX project. For example finding the shortest path from the source IoT node to the destination IoT node. This algorithm requires graph traversal. Caching will improve the graph traversal by caching the visited nodes.

3.0.7 Advantages Of LRU Caching

- The insert, update and delete operation in LRU Caching is a constant time operation.

- Faster access to the objects in the cache.
3.1 Graph showing the system performance relative to the cache size

Fig. 3.9 Shows the performance of the IoTX framework with respect to the cache size. The x-axis in the figure is the cache size as a percentage of the total JVM memory allocated to the framework. The y-axis is the time in milliseconds the framework took to create a graph of 500 nodes. As you can see from the x-y plot, when the cache size is increased the graph is created in less time. This proves that with the introduction of a caching layer in the system, we have improved the system performance as a whole. Table 3.1 contains the data that is used to draw this x-y plot.
Table 3.1: Below table shows how the execution of graph creation in IoTX project varies with different cache size. As cache size is increased the IoTX graph is created in less time. The cache size is a percentage of the total JVM size allocated to the IoTX project.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Cache Size%</th>
<th>Time in ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>500</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>500</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>500</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>500</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>500</td>
<td>5</td>
</tr>
</tbody>
</table>

The first row of the table shows the IoTX framework without using caching. As we increase the cache size in the subsequent rows the time taken to create the graph is getting reduced. The testing is done for a graph of 500 nodes.
Chapter 4

Calvin Applications

4.0.1 Calvin Applications Using System Actors

Calvin framework contains a lot of in-built systems actors that can be used to form Calvin applications. This list of system actors are growing with each Calvin release and a lot of applications can be realized using these actors. Calvin applications form a directed graph. The actors act as a node in the graph and they are connected to each other by their input and output ports by which they communicate with each other. Fig.4.1 taken from Calvin Wiki page explains how applications are formed or viewed logically in Calvin Framework.
CHAPTER 4. CALVIN APPLICATIONS

Figure 4.1: Calvin Application Using System Actors [1]

```
#Actor
Trigger : std.Trigger(tick=1, data=null)
sense   : sensor.Temperature()
print   : io.Print()

/* Connections */
trigger.data > sense.measure
sense.centigrade > print.token
```


4.0.2 Calvin Application Using Custom Actors

There are many use cases or applications where built-in actors do not provide the required functionality needed by the user. Calvin Framework provides a feature of writing custom actors where a user can define actions of these custom actors depending on the application they are building. Fig. 4.2 shows a data flow graph composed of custom actors.

Figure 4.2: Calvin Application Using Custom Actors
Fig. 4.3 shows a code example of one of the actors in 4.2.

```python
class InputSum(Actor):
    
    """
    Sums input on port 'first' with input on port 'second'
    Inputs:
    first : integer
    second : integer
    Output:
    result
    """

    def init(self):
        pass

    @condition(action_input=['first', 'second'], action_output=['result'])
    def add(self, first, second):
        result = first + second
        return (result,)

    action_priority = (add,)
```

Figure 4.3: Calvin Application Code For Custom Actors

4.0.3 Calvin Script And Calvin Application

Fig. 4.4 Shows a sample Calvin script and how it translates to a Calvin Application with input and output ports connected. The connections in the Calvin script indicates the connection between the input and output ports for the Calvin application.
**Calvin Script**

/* Actors */
src : std.Constant(data=10)
add: math.InputSum()
out : io.Print()

/* Connections */
src.data    > add.first
src.data    > add.second
add.result  > out.token

---

**Calvin Application**

Actors:
- `src` actor
- `add` actor
- `out` actor

Connections:
- `data` from `src` to `add`.
- `first` from `add` to `result`.
- `second` from `add` to `token`.

Figure 4.4: Calvin Script And Application
4.0.4 Steps To Deploy Calvin Application

- The first step is to write a Calvin Script describing the actors in the application and connections between the actors.

- If you want to use custom actors, write custom actors in Python and create a .config file stating the path of the custom actors so that Calvin framework would be able to locate the custom actors.

- Write a requirements file with file extension as .deployjson. The requirement file contains the information about which actor is bound to which runtime.

- Start the runtimes using csruntime command.

  ```
  csruntime --host localhost --port <portNum> --controlport <portNum>
  --name <runtime name>
  ```

- Define the application topology by adding peers to IoT nodes. This step is required for Calvin framework to know the connections between the runtimes.

- Deploy the Calvin script and the requirement file using the cscontrol command. The cscontrol command compiles and deploys the Calvin application.

  ```
  cscontrol http://x.x.x.x:<control port> deploy --reqs <requirements file name> <calvin script file name>
  ```
4.0.5 Advantages Of Calvin Framework

- Calvin Runtimes are very lightweight which makes them a perfect fit for IoT devices.

- There is another version of Calvin framework which is called Calvin-Constrained which can be ported to any microcontroller which has a communication capabilities.

- Actors in the Calvin framework run independently which helps building asynchronous applications.

- Actors in the Calvin framework can migrate from one runtime to another with very little overhead. This is one of the powerful features of Calvin as it allows the application developer to migrate actors if there are resource constraints in the current runtime.
Chapter 5

Calvin Actor Model in IoTX Project

The goal here is to integrate the Calvin actor model in the IoTX framework. Once the user has created the simulation graph of IoT nodes in this framework, he should be able to create actors and bind these actors to different IoTX nodes.

There is a graphical user interface provided to load the Calvin script from the hard drive. This system is intelligent enough to create the actors by reading the Calvin script and also create the connection between the actors. Right now the actors are randomly bound to the IoT nodes in the simulator. At this point of time, a Calvin application has been created by the IoTX framework. Once the user clicks the run application button provided in the UI a call is made to the back-end.
CHAPTER 5. CALVIN ACTOR MODEL IN IOTX PROJECT

5.0.1 Deploying Application from IoTX Framework

The challenge here is to bind/integrate the Calvin application created by the user with the actual Calvin framework. Front-end sends the following information to the backend to launch the Calvin application:

- The IoTX node id to which actors are bound.
- Name of the actors in the application.
- Application name.
- Information about actors mapping with the IoTX node.

The back-end takes this information sent from the front-end and dynamically creates a requirement file needed to deploy the Calvin application. Back-end starts the simulation by first firing a Docker image for each IoTX node that contains the actor and start the runtime in these docker containers. At this point of time, the system is ready to deploy the application to the Docker containers. Note: Each docker image mimics an actual IoT node in an actual network. To deploy the application, REST API calls are made from the IoTX framework to the Docker containers to talk to the runtimes running on these Docker containers. Calvin allows the user to talk to the runtimes by making the REST API calls. We are leveraging this feature to make our simulator work. Fig. 5.1 shows the format of REST Post message to deploy a Calvin Application.
CHAPTER 5. CALVIN ACTOR MODEL IN IOTX PROJECT

Figure 5.1: REST Post Message To Deploy Application

```json
POST/deploy
Body:
{
  "name": <application name>,
  "script": <calvin script>
  "deploy_info":{
    "requirements": { }
  }
}
```
IoTX project is a layer on top of Calvin framework which will not only manage and deploy applications dynamically but also aims at serving as an efficient resource manager by using actor migration. Calvin exposes REST APIs. In this project, we have implemented an interface that communicates with Calvin running on any machine. IoTX uses REST APIs to control and communicate with Calvin applications. The framework provides a connection to the Docker containers where Calvin is installed to simulate a distributed test application environment. Once the user binds the actors to IoT nodes in the front-end and runs the custom distributed application, the backend collects all the data from the front-end and creates the requirements dynamically to start and run the application in Docker containers.
6.0.1 Steps Required To Start The Docker

The following are the sequence of API’s that needs to be called from the IoTX project to start and run the Docker container:

- Docker.CreateDocker(<name>) API creates a Docker container containing the Calvin framework. Internally this API executes the following command: docker create -t -i –name <name> erctcalvin/calvin-deps:master bash

- Docker.startDockerContainer(<name>) API starts the Docker container. Internally this API executes the following command: docker start <name>. At this point of time docker is started and now we can start the Calvin runtimes on the Docker image we just created.

- Docker.startRuntime(<name>) API starts the runtime on the Docker image. Internally it executes: docker exec -it <name> csruntime –host <Docker’s IP address> –port 5000 –controlport 5001 –name <name>

- To get the Docker’s IP address call Docker.getDockerIpAddress(<name>) API.

A module Docker is provided in the IoTX project which manages the Docker for the user. Another module named DockerHandler is also added to the framework which creates the requirements file dynamically depending on how actors are bounded to the IoTX nodes. This module also take care of starting the Docker and forming the correct message format required to deploy the Calvin application. The functionality of stopping or destroying the Calvin
application is also provided in the framework. When a user stops the application from the front-end, the back-end system maintains the mapping of Calvin application name and the unique application id assigned to an application. Using the application id a REST call is made to the Calvin runtimes to stop the application. Stopping application will not kill the Docker containers and runtimes running on these containers. These containers and runtimes can be used again by the same or different application. If a user wishes to destroy the application completely from the system, a delete application functionality is also provided in the framework. This action will kill the application and the associated Docker containers if there are no other applications using the same Docker containers.

6.0.2 Calvin Control APIs

Calvin Control APIs are REST APIs provided by the Calvin framework to control the Calvin runtimes. IoTX framework uses this API to deploy Calvin applications and to control Calvin runtimes. Once the Docker Containers are started, IoTX framework calls the following Calvin Control APIs to deploy the application:

- CalvinRestHandler.deployApplication(String controlUri, String script, JSONObject requirements, String appName) API will deploy the Calvin application to the Docker container using the IP address passed as the first parameter.

- CalvinRestHandler.deleteApplication(String controlUri, String appId) API will delete the application using the application Id passed as a parameter
by making the DELETE REST API call to the Calvin Framework.

There are more APIs provided in the IoTX framework to manage Calvin Applications. List of Calvin Control APIs are provided in the Calvin wiki page. Most of the useful APIs are already implemented in the IoTX framework. The module CalvinRestHandler is provided in IoTX framework to manage and control Calvin applications.
Chapter 7

Conclusion

Calvin is a powerful light weight framework which is ideal to create distributed data flow applications. Calvin provides two flavors, Calvin-base which is a full fledged framework to create Calvin applications and Calvin-Constrained which can be deployed to resource constrained IoT nodes. Both the frameworks can talk to each other. Therefore, the user can write applications which can consist of a mixture both the Calvin frameworks. There are many tools that Calvin provides which are very useful to handle and deploy applications. Any software can talk to Calvin runtimes via REST API calls. IoTX project utilizes this feature to integrate Calvin actor model within IoTX project.

7.0.1 Future Work

Process Model

Process Model is formed by many processes, where each process can be used in a more complex process chains. A process chain is a collection of processes
which form a directed acyclic graph (DAG) and can be executed one after the other forming a chain to get the expected output. In Calvin, we can create reusable components consisting of several actors. Each reusable component will form a process model where components can connect to different reusable components to form a complex application. A reusable component may be a distributed data flow graph. Fig. 7.1 shows an overview of a process model.

Fig. 7.1 shows two views of IoT device. The topmost figure shows an overall view of an IoT device in an IoTX framework. An IoTX node consists of multiple actors running in Calvin runtime. These actors are connected to process model which internally can be a DAG of actors performing some task based on the input they receive.

The second or bottom figure gives a detailed view of how different IoT nodes are connected and how process model fits in the overall picture. There are two custom applications in this figure. Application 1 is formed by three IoTX Nodes (IoTXNode 1, IoTXNode 2 and IoTXNode 3). The sensor of IoTXNode 1 generates sensor data which is collected by an actor running on this node. This actor forwards the data to the IoTXNode2’s Actor 1. This is not a simple actor. This actor is composed of a Process which is a reusable component and be composed of multiple process to perform a job. In this application it is transforming a data from one unit to other. The output of this actor is sent to IoTXNode3’s actor which collects this data and sends it to the actuator.

The application 2 in this figure is a bit more complex application. In this example the Actor 2 of IoTXNode 2 is composed of process p2. Process p2 uses process p1 to achieve a more complex task here. This figure explains how
these reusable process can be grouped together to form a DAG. The overall application also forms a DAG and the actors which are nodes of this DAG can also be composed of different process which internally forms a DAG inside an actor. The process model can be used to create many applications using existing processes in the framework.

Figure 7.1: Calvin Process Flow Model

This powerful process flow model can be incorporated in the IoTX project. These reusable components can be a set of distributed algorithms that can be used in multiple applications.
Bibliography


