Edge Computing Framework for Internet of Things using Calvin

by

Ganesh Rajasekharan

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

May 2017
I would like to thank Professor Peizhao Hu for suggesting the idea for the project and mentoring me until the completion of my capstone project. Also I would like to thank my parents for their support and motivation.
Abstract

Edge Computing Framework for Internet of Things using Calvin

Ganesh Rajasekharan

Supervising Professor: Dr. Peizhao Hu

In the recent years, there has been an explosion in the number of Internet of Things (IoT) devices. The number is estimated to reach about 20 billion by the end of year 2020. Many existing IoT system designs include devices collecting and sending sensor data directly to cloud compute services for performing data aggregation and analytics on sensor data. The monolithic design of IoT devices, offloading all computations to cloud leads to high network latencies and operating costs for maintaining cloud infrastructure. Many IoT applications also require performing real time data analytics such as computing location awareness which is time sensitive. The concept of performing computations at the edge of the network, tapping into the computational resources of the IoT devices themselves is termed as Edge computing.

In order to realize such a system, an open source data flow framework with a controller for IoT is implemented in Python using Calvin which is an open source IoT communication framework. The data is processed as it flows through the network routing nodes and the result is obtained in the network without making additional requests to cloud services.
# Contents

Acknowledgments ......................................................... ii

Abstract ....................................................................... iii

1 Introduction ............................................................... 1

2 Design ......................................................................... 3
   2.1 Calvin - Introduction ................................................. 3
   2.2 Calvin Application - Components ................................. 5
      2.2.1 Describe Component ........................................... 5
      2.2.2 Connect Component ........................................... 6
      2.2.3 Deploy Component ........................................... 7
      2.2.4 Manage Component ........................................... 8

3 Implementation ............................................................ 9
   3.1 Data Flow as Actor Graph ......................................... 9
   3.2 Controller - IoT Coordination .................................... 12
   3.3 Controller - Data Flow Update Algorithm ................. 12
   3.4 Controller - Data Flow Update .................................. 13

4 Analysis....................................................................... 17

5 Conclusions ............................................................... 22
   5.1 Current Status .......................................................... 22
   5.2 Drawbacks and Challenges ....................................... 23
   5.3 Future Work ........................................................... 23

Bibliography ................................................................... 24
# List of Figures

1.1 Existing method for IoT data processing on Cloud ........................................ 2
2.1 System Design of the framework ................................................................. 4
3.1 Data Flow as an Actor Graph ................................................................. 10
3.2 Distributed Actor Deployment JSON ...................................................... 13
3.3 Starting cruntime Shell Script ................................................................. 15
3.4 Actor Map with unique actor hashes ....................................................... 16
3.5 Deployed application printing average temperature results into runtime console ................................................................. 16
4.1 Memory footprint of the framework ......................................................... 17
4.2 Actor graph of Temperature Application - Part 1 ...................................... 18
4.3 Actor graph of Temperature Application - Part 2 ...................................... 18
4.4 Director Web Server listening for new Actor Graphs .................................. 19
4.5 A sample POST request with new graph configuration .............................. 19
4.6 Director acting on new graph configuration .............................................. 20
4.7 New data flow actor graph deployment .................................................. 20
4.8 Updated Actor graph of Temperature Application - Part 1 ....................... 21
4.9 Updated Actor graph of Temperature Application - Part 2 ....................... 21
Chapter 1

Introduction

Nowadays, most of the analytics on sensor data is carried out on the cloud due to the availability of large compute resources as shown in Figure 1.1. A wireless sensor network with temperature sensors connected to components in a power grid or an active security monitoring system for homes, relies on real time analytics on sensor data to trigger events in response to an emergency. An architecture in which data has to flow back and forth between the device and cloud would not be suitable in case of network congestion.

Streams of data generated from individual sensors are often offloaded into the cloud rather than making use of the computational power available at the routing nodes connected to the sensors. The main challenge in designing such systems is the lack of communication and coordination between IoT devices. Additionally, most of the communication frameworks are not suitable for IoT devices since they are resource constrained. Calvin [11] is a communication framework for IoT developed by Ericsson research to enable the communication between IoT devices using the concepts of Actor model [6] and Data flow based programming [10]. Based on the Calvin framework, a data flow processing framework for IoT devices is implemented with a controller which dynamically updates the data flow between sensor nodes based on changes in network conditions.

Many alternative communication frameworks exist for reducing the disconnect between IoT devices and merging it with cloud. NoFlo [1] is a JavaScript framework based on flow based programming. Applications in NoFlo are modelled as graphs and nodes as the routing components named as NoFlo which is similar to actors in Calvin. The NoFlo components do not provide any functionality for migration of actors, whereas in Calvin, actor...
migration is possible between nodes based on the actor graph. Another open-source framework that is developed by Microsoft is Orleans [13] which is based on virtual actors for creating high performance cloud computing applications. Calvin is specifically aimed at creating distributed applications for resource constrained IoT devices to cloud computing services. Calvin also provides a lightweight runtime named Calvin Constrained [12] implemented in C programming language to be run on resource constrained devices and provides most of the important features of a fully blown Calvin runtime.

Figure 1.1: Existing method for IoT data processing on Cloud
Chapter 2

Design

2.1 Calvin - Introduction

Calvin aims to provide the benefits of Mobile Agents which is a widely adopted solution for communication and migration of processes between heterogeneous systems. The benefits provided by Mobile Agents as described by Lange and Oshima [8] are that they can counter network latencies and decrease traffic, encapsulate network protocols, execute in an asynchronous and autonomous manner and also be fault tolerant.

The main idea behind creating asynchronous applications in Calvin is based on the concept of actors which is a mathematical model for achieving concurrency and scalability. An actor in Calvin is a software abstraction for performing a computational task. For example, in a home temperature monitoring system, an actor reads the temperature sensor values and another actor is handles routing the sensor data in the network to an actor in another node which processes the data further. An actor can perform the following tasks when it receives a message from another actor:

1. Create additional actors.
2. Send messages to other actors.
3. Assign a task to be performed to the next message received.

A Calvin application consisting of actors which can be deployed on a platform independent runtime which is responsible for data transport. Calvin aims to bridge the disconnected state of the IoT development environment since solutions has to be cross-vendor
supported for these devices. The application developer does not have to take into account of the heterogeneous state of the IoT application development environment due to the presence of different communication protocols, supported programming languages and service discovery mechanisms.

The software stack of Calvin in figure 2.1 consists of a platform dependent layer in the runtime which is responsible for data transport and abstracting the system dependent components such as sensors, actuators etc to the platform independent layer which is also a part of the runtime. The platform independent layer handles the runtime migration and inter-runtime communication and coordination. The flexibility provided by Calvin to migrate runtime with actors across devices is useful for error handling and it provides algorithms to support all the different kinds of transport layers such as Wifi, Bluetooth etc. Another feature is that the scheduler, responsible for migrations of runtime and data passing which resides on the platform independent layer can also be extensible through plug in mechanisms. The multi-tenant nature of Calvin is helpful for creating a data processing framework for IoT devices since the actors once migrated to another device can share the runtime of actors in that device satisfying memory constraints.

![System Design of the framework](image)

**Figure 2.1: System Design of the framework**
2.2 Calvin Application - Components

The entire life cycle of a Calvin application can be divided into four main components based on the nature of the particular component -

1. Describe - The reusable operative blocks or actors of the application.

2. Connect - The connection between actors modeled as a directed graph.

3. Deploy - The deployment of the actor graph on runtimes.

4. Manage - The migration policy for actors for fault tolerance.

By dividing the application life cycle into four separate components Calvin aims to achieve high scalability and fault tolerance by not sharing states between actors. The actors can modify each others state by exchanging messages and not by sharing states.

2.2.1 Describe Component

Building blocks of Calvin is described by deriving concepts from the actor model. An actor in Calvin is a software abstraction slightly similar to a subroutine in programming languages. The actors receive data tokens in their in-ports and perform some computation on the data received and produce output tokens through their out-ports. The actors in Calvin can be specified to route the output tokens to peers present in the network in a round-robin scheme or any specified routing mechanism. The communication between two actors, active on two separate runtimes is carried out through sockets by sending User Datagram Protocol packets between corresponding runtimes.

The general syntax for writing an actor in Calvin is by the following syntax:

<actor>::<Namespace>.<ActorType>({argument_list})

In the above syntax block, actor tag refers to the unique name of the actor, Namespace denotes the unique identifier for all the available actors and the type of actor available in the namespace through ActorType identifier.
Standard system actors are provided with Calvin for performing basic operations such as random number generation, calculating power of a number, printing to standard output etc. While these actors are helpful, they are not sufficient to create applications which require more complex computations to be performed on data. A custom actor in Calvin can be written as a Python class with Python decorators which invoke functions responsible for certain behavior from the Calvin core system classes.

The three main decorators while writing a Calvin actor are the following:

1. @manage- This decorator marks the fields of stateful actors data to be serialized during migration.

2. @condition - The conditions required for an action to be triggered such as the required input and output data parameters.

3. @stateguard - An action is chosen based on the evaluation of the state of the actor which returns a boolean value.

Calvin actors can be grouped into components and be reused within the same Calvin script. In order to use the component with other applications it has to be installed into a valid namespace in the ActorStore where Calvin looks for available actors.

2.2.2 Connect Component

The actors mentioned in the previous section are deployed on runtimes across multiple machines. The connection between the ports of these actors are modelled as a directed graph. The connections are mapped into the correct ports through a intuitive script with the extension .calvin called as the CalvinScript. An example of a Calvin script is shown below:

```python
rand: math.Random(lower=30, upper=50)
sensor : std.Trigger(tick = 2.0, data=null)
node : math.ComputeAverage()
out : io.Log(loglevel="INFO")
```
sensor.data > rand.trigger
rand.integer > node.temp_sensor
node.result > out.data

In the above code snippet from a sample CalvinScript, the first four lines are actor definitions and the remaining three lines depict the data flow among them. Actor rand generates random integers in the range from 30 to 50 and the actor sensor generates a token on its output port every 2 seconds to trigger a random number from the rand actor. The token generated at the output port of sensor is forward to a custom actor installed in the namespace math in the system which performs computations on the data and logs it into the runtime console using the standard output out actor.

Therefore, by creating a logical connection for data flow between actors, we can create a distributed application. The connections described in the CalvinScript is static when the application is running. Therefore we investigate an approach in the Implementation section to modify the connections or the actor graph when the data flow changes between nodes due to changes in network metrics such as bandwidth, latencies, number of hops etc. Also Calvin does not provide a mechanism to dynamically update the flow changes between the nodes, therefore an algorithm to detect network changes in the actor graph and to update network flows between nodes is described in the Chapter 3 - Implementation.

2.2.3 Deploy Component

Once we have the definitions for actors and the connections between them, the next step is to decide where the actors should reside and how they should exchange data. Calvin provides runtimes which are light weighted and discover each other in the network by maintaining a Distributed Hash Table(DHT) based on Kademlia [9]. Once an application is deployed using the csruntime command the address and port information for a peer runtime is added into the DHT keys. The DHT takes a few moments to build up peer information of runtime adresses through the broadcast address. Runtimes can also be manually linked using the cscontrol command. Kademlia takes O(logn) attempts to find the correct runtime
address from n number of peers.

2.2.4 Manage Component

The application once deployed on a distributed environment is managed by the runtime and actors are migrated by the runtime in case of the failures for error recovery. Management of deployed applications include keeping tracking of resource usage by the actors. These are useful only for providing error recovery and scaling for the application and does not provide features for controlling the data flow between the nodes. Since Calvin does not support any fine grained mechanism to manage the data flow in deployed applications, a controller is implemented on top of Calvin to keep track of applications and to manage and update the data flow between actors.
Chapter 3

Implementation

3.1 Data Flow as Actor Graph

The main challenge while creating distributed applications with data flow model is in updating the state of the system when the existing flows are no longer valid. Similar to the concept of updating the forwarding plane in switches through Open Flow [3] by the controller in Software Defined Networks, we describe a technique in this section to update the data flow between the routing nodes in case of a change in configuration among IoT devices.

A Calvin application which forms a mesh network of runtimes with actors deployed on them is shown below:

```c
/*
The code below specifies the ip->op mappings of the actors and forms an actor graph spanning across multiple runtimes.
*/

#Actor specifications
sensor1 : std.Trigger(tick = 2.0, data=55)
sensor2 : std.Trigger(tick = 2.0, data=60)
node1 : math.ComputeAverage()
node2 : math.DisplayResult()
out : io.Log(loglevel="INFO")

#Actor logical connections
sensor1.data > node1.temp_sensor1
sensor2.data > node1.temp_sensor2
```
# Final result of the data flow across nodes
node1.out > node2.input
node2.result > out.data

An actor graph formed for the above Calvin script is shown in Figure 3.1. The actors are deployed on four separate runtimes and they communicate with each other by passing tokens on ports. The corresponding actor graph of the application can be modelled as a directed graph.

![Figure 3.1: Data Flow as an Actor Graph](image)

We can model more complex applications such as an application for finding average temperature value of a sensor network by performing computations on the nodes. The CalvinScript showing the actors and connections among them is shown below:

```calvin
# Actor specs
rand0: math.Random(lower=30, upper=50)
rand1: math.Random(lower=30, upper=50)
rand2: math.Random(lower=30, upper=50)
rand3: math.Random(lower=30, upper=50)
rand4: math.Random(lower=30, upper=50)
sensor0 : std.Trigger(tick = 2.0, data=null)
sensor1 : std.Trigger(tick = 2.0, data=null)
sensor2 : std.Trigger(tick = 2.0, data=null)
sensor3 : std.Trigger(tick = 2.0, data=null)
sensor4 : std.Trigger(tick = 2.0, data=null)
```
node1 : math.ComputeAverage()
node2 : math.ComputeAverage()
node3 : math.ComputeAverage()
node4 : math.ComputeAverage()
out : io.Log(loglevel="INFO")

#Actor logical connections
sensor0.data > rand0.trigger
sensor1.data > rand1.trigger
rand0.integer > node1.temp_sensor
rand1.integer > node1.temp_network

sensor2.data > rand2.trigger
node1.result > node2.temp_network
rand2.integer > node2.temp_sensor

sensor3.data > rand3.trigger
rand3.integer > node3.temp_sensor
node2.result > node3.temp_network

sensor4.data > rand4.trigger
rand4.integer > node4.temp_sensor
node3.result > node4.temp_network

#final result of all the node data flow
#destination Node
node4.result > out.data

The resulting data of the sensor network is computed by data flow programming and eliminates the need for sending multiple streams of data into the cloud services. The final average value received at the destination node can be used to perform data analytics using a suitable machine learning algorithm. Since computations are performed locally, network resources are relieved from congestion and only computationally heavy analytics on data models need to be performed on the cloud services.
3.2 Controller - IoT Coordination

In the previous section, an application comprising of actors was modelled as an actor graph and deployed across runtimes. It is interesting to note that the data flow between these nodes can change due to different network overheads, and delays would be critical for time sensitive applications. The optimal data flow between nodes or the actor graph can be dynamically computed using a suitable constraint-based path selection algorithm described by Kuipers et al[7]. or manually updated by the user through an interface. The controller which is implemented as a class in Python is deployed on the node in which the runtime is active and performs the following steps to update data flow between the nodes, given a new actor graph with updated data flow.

3.3 Controller - Data Flow Update Algorithm

The following steps are carried out when the Controller is deployed on a node which is connected to other runtimes. Suppose an application is running with a data flow modelled by graph G. When a new data flow graph P is received by the server, the steps of processing are as follows.

1. Start controller web server in daemon mode
2. Listen for new data flow graph P via REST
3. Check for running applications
4. Check if current data flow graph G and received graph P are isomorphic
   4.1 If graphs are isomorphic, continue
   4.2 Else if graph P vertexes equal no. of active peers
       4.2.1 Parse actor graph P and create deployment script
       4.2.2 Redeploy app. with new deployment script
       4.2.3 If redeployment is successful
           4.2.3.1 Update previous actor configurations
           4.2.3.2 Remove existing application from DHT
       4.2.4 If redeployment fails, rollback

Goto 2
The controller web server listens on a port on the node with an active Calvin runtime. The server listens for new data flow information as an actor graph in the form of an adjacency list. The REST web server updates the received graph configuration into a Python dictionary which is processed by the configuration check subroutine in a separate thread. The isomorphism between graph G and P is carried out using VF2 [4] algorithm to check for changes in the newly received graph. Once the graphs are determined to be non-isomorphic, the new graph is deployed by creating a new CalvinScript and deployed into the runtime. This does not affect the already active application and the hash id of the currently executing application is retrieved and is terminated. The controller also switches back to previous actor graph if the newly received graph deployment fails.

3.4 Controller - Data Flow Update

The actor placement on runtimes is mentioned by a JSON file named with the extension .deployjson in Calvin. The figure 3.2 shows the contents of the JSON file for the actor placement across nodes for the temperature sensor application.

```
{
  "requirements": {
    "rand0": [{"op":"node_attr_match","kwars":{"index":null,"name":"runtime-0"}}],
    "rand1": [{"op":"node_attr_match","kwars":{"index":null,"name":"runtime-0"}}],
    "sensor0": [{"op":"node_attr_match","kwars":{"index":null,"name":"runtime-0"}}],
    "node1": [{"op":"node_attr_match","kwars":{"index":null,"name":"runtime-0"}}],
    "sensor1": [{"op":"node_attr_match","kwars":{"index":null,"name":"runtime-0"}}],
    "node2": [{"op":"node_attr_match","kwars":{"index":null,"name":"runtime-1"}}],
    "sensor2": [{"op":"node_attr_match","kwars":{"index":null,"name":"runtime-1"}}],
    "node3": [{"op":"node_attr_match","kwars":{"index":null,"name":"runtime-2"}}],
    "sensor3": [{"op":"node_attr_match","kwars":{"index":null,"name":"runtime-2"}}],
    "node4": [{"op":"node_attr_match","kwars":{"index":null,"name":"runtime-3"}}],
    "sensor4": [{"op":"node_attr_match","kwars":{"index":null,"name":"runtime-3"}}],
    "out": [{"op":"node_attr_match","kwars":{"index":null,"name":"runtime-3"}}]
  }
}
```

Figure 3.2: Distributed Actor Deployment JSON

After the deployjson file is created to describe the actor placements across nodes, they have to be deployed on runtimes. An attempt to start runtimes on multiple machines automatically using a shell script as shown in figure 3.3. Although the starting of the runtime
was attempted to be automated using the script, the Calvin runtimes fail to start on the machines correctly and had to be manually started on each individual Odroid. This could be fixed by piping the script with ssh command and running in the remote nodes.

The figure 3.4 shows the deployed Calvin application and the corresponding actor name and hash id pair which is a map maintained by each node. The application is referenced by the unique id generated using the SHA1[2] hashing technique. Each actor is identified using this unique id for placing the actor on the runtimes and sending tokens based on the specified connections between actor ports. The application hash id is used to terminate the application by the controller using the control URI for the application.

Once the application has been successfully deployed, the runtime with the standard out actor will display the results in the console of the runtime as shown in figure 3.5. This is the average value of all the temperature sensor nodes throughout the entire network. In case a node is shut down, the runtime will migrate the actors to a suitable runtime provided that the scheduling constraints are satisfied with the actor execution requirements.
#!/bin/sh

SCRIPT_NAME=$(basename $0)

function error_desc
{
  # Exit the script when fatal error is encountered
  # Input argument: The error descriptor as String

  echo "ERROR MESSAGE DESCRIPTION!!"
  echo "$SCRIPT_NAME: $1="Unknown Error""
  1>&2
  exit 1
}

# Start Calvin Runtimes on Different Nodes

# csruntime --host 10.10.10.206 --port 6000 --controlport 6001 --name runtime-0 &
# if [ "$S?" = "0" ]; then
#   echo "Created runtime-206."
# else
#   error_desc "$LINENO: ERROR while starting runtime-206...ABORTING"
# fi

# csruntime --host 10.10.10.208 --port 6002 --controlport 6003 --name runtime-1 &
# if [ "$S?" = "0" ]; then
#   echo "Created runtime-208."
# else
#   error_desc "$LINENO: ERROR while starting runtime-208...ABORTING"
# fi

# csruntime --host 10.10.10.209 --port 6004 --controlport 6005 --name runtime-2 &
# if [ "$S?" = "0" ]; then
#   echo "Created runtime-209."
# else
#   error_desc "$LINENO: ERROR while starting runtime-209...ABORTING"
# fi

# csruntime --host 10.10.10.211 --port 6006 --controlport 6007 --name runtime-3 &
# if [ "$S?" = "0" ]; then
#   echo "Created runtime-210."
# else
#   error_desc "$LINENO: ERROR while starting runtime-210...ABORTING"
# fi

# Deploy a Calvin Application on active csruntimes

# cscontrol http://localhost:6001 deploy --reqs avg_app.deployjson avgapp.calvin
# if [ "$S?" = "0" ]; then
#   echo "Deployed avg_app.calvin"
# else
#   error_desc "$LINENO: ERROR during deploying avg_app.deployjson...ABORTING"
# fi

Figure 3.3: Starting csruntime Shell Script
Figure 3.4: Actor Map with unique actor hashes

Figure 3.5: Deployed application printing average temperature results into runtime console
Chapter 4

Analysis

The main challenge in creating a data flow framework is achieving scalability of the executing system during data processing. Therefore, the implementation was tested for a distributed Calvin application processing sensor data. The sensor data was simulated by generating random temperature readings and then used to find the average value among the set of nodes. On top of this application, the controller was executed to test for flow updates using actor graphs. Calvin does not allow the modification of the actor graph. This is because all the peers in the DHT maintain the same actor graph information and a unique id for actors and where to find them using the deployment json file which contains the actor to runtime mappings.

Figure 4.1: Memory footprint of the framework
The following part of this section demonstrates the controller being deployed on the runtime where the application is already running as shown in the previous section. The controller acts on the new actor graph depicting flow changes and performs the data flow update as mentioned in Section 3.3.

The figure 4.2 and 4.3 shows the current actor data flow of the deployed application to find the average of network temperature shown in the previous section 3.4.

![Figure 4.2: Actor graph of Temperature Application - Part 1](image)

![Figure 4.3: Actor graph of Temperature Application - Part 2](image)

The controller is called as Director since it manages actors and is now started as shown in figure 4.4 and listens for new configurations.
For demonstration purpose, a HTTP POST request is made to the controllers web server port with the new actor graph in the form of an edge list with the request content type set to a JSON file as shown in figure 4.5.

Once the server receives the POST request as shown in figure 4.6, it is logged and saved for processing using the algorithm mentioned in section 3.3. The new application and actor ids are generated and placed for deployment on the runtime as shown in figure 4.7.

The figures 4.8 and 4.9. provides the visualization of the newly deployed actor graph, we can notice the change in data flow between nodes compared to figures 4.2 and 4.3.
Figure 4.6: Director acting on new graph configuration

```plaintext
2017-04-24 14:03:06 "POST / HTTP/1.1" 200 12

Newly Received graph:
{
  "graph": [
    {
      "node1": ["node3"],
      "node3": ["node4"]
    }
  ]
}

Current graph:
{
  "node1": ["node2"],
  "node3": ["node4"],
  "node2": ["node3"]
}

Performing config check...

Deploying received actor graph...

```
{
  "graph": [
    {
      "node1": ["node3"],
      "node3": ["node4"]
    }
  ]
}
```

(curr_app_id: deede523-aba0-4f04-9ca9-6bf53f70cc35)

```
app_killer: cstop http://127.0.0.1:6501 applications delete deede523-aba0-4f04-9ca9-6bf53f70cc35
```

kill_code: 0

Application: deede523-aba0-4f04-9ca9-6bf53f70cc35 has been stopped...

dep_app_command: cstop http://127.0.0.1:6501 deploy avgapp_output.calvin

Figure 4.7: New data flow actor graph deployment

```plaintext
kill_code: 0
Application: deede523-aba0-4f04-9ca9-6bf53f70cc35 has been stopped...

dep_app_command: cstop http://127.0.0.1:6501 deploy avgapp_output.calvin
```
Figure 4.8: Updated Actor graph of Temperature Application - Part 1

Figure 4.9: Updated Actor graph of Temperature Application - Part 2
Chapter 5

Conclusions

The data flow based programming along with the actor model provides a great combination for creating an asynchronous data processing framework for IoT. The application performed all the computations on the routing nodes by exchanging tokens with each other. The data flow update approach mentioned in the previous section can be performed for large graphs since isomorphism check is completed in a reasonable time due to the performance provided by VF2 algorithm.

5.1 Current Status

The application has to be redeployed whenever the isomorphism check fails in the algorithm described in Section 3.3. This can be cumbersome for a highly volatile network of large number of nodes. The migration of actors to the corresponding runtimes can be delayed due to constraints to be satisfied. However, the method is suitable for a non volatile network where the changes in network flows are not very frequent.

The actor graph of the Calvin application is static and can only be modified by destroying the application since the actors are deployed across different runtimes. The actors are linked together by the connections in the actor graph which cannot be changed without the redeployment of the application. As Calvin is open sourced and evolving this is definitely a new area for further research.
5.2 Drawbacks and Challenges

Although Calvin provides runtime support for communication, the distributed applications can be difficult to debug. The errors are not descriptive and is time consuming to fix. Also Calvin is written in Python which cannot be executed on resource constrained devices. Calvin constrained which is a lightweight version of Calvin does not have a shared key value store and hence the runtimes have to be manually linked.

5.3 Future Work

The graph isomorphism check can be tested with GraphQL [5] algorithm against VF2 for improvement in performance. Also the scheduling scheme for actors during migrations in Calvin can be researched further for adding support to more network metrics. Another possible area for future work is to maintain an additional distributed data structure for actor connections that can be used by Calvin to make changes to the actor graphs without the need for redeployment of the application.


