iotX: Modeling Topology and Application using Neo4j Database

by

Vinay Vasant More

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

08 2017
Acknowledgments

I would like to thank Department of Computer Science at RIT and professor Dr. Peizhao Hu for offering an opportunity to work on this interesting project. I am grateful to professor Dr. Carlos R. Rivero for introducing me to Graph Databases world and iotX team at RIT for building a professional environment for sharing exciting ideas and learning different technologies. Last but not least, I would like to dedicate this work to my family and friends for believing in me throughout my academic career.
Abstract

iotX: Modeling Topology and Application using Neo4j Database

Vinay Vasant More

Supervising Professor: Dr. Peizhao Hu

IoT (Internet of Things) devices with limited resources are currently pushing data to cloud for data processing [10]. We are proposing an iotX framework for reducing this dependency on cloud and utilizing resources across distributed IoT devices by carrying out lightweight processing at the Edge with dynamic task allocation.

In IoT world, user can interconnect physical IoT devices and develop all kinds of applications by sharing and processing data based on the requirement. A real world application in an IoT environment can require thousands of IoT devices to be interconnected along with required resources such as sensors, actuators, processing power, etc. In iotX framework, user is provided with a web interface for customizing IoT network topology and launching different applications to test bed to verify the desired outcome before actually implementing the setup in the real world. Thus, there is a need to store initial configuration of every entity like IoT Node, Sensor, etc. along with efficient tracking of user updates at each event. For this project, we are using multiple database technologies for different purposes based on their advantages over the other. In next sections, we’ll learn more about this.
## Contents

**Acknowledgments** ................................................................. ii

**Abstract** ........................................................................ iii

1 **Introduction and Motivation** ...................................................... 1
   1.1 Motivation ........................................................................ 1
   1.2 iotX Framework ................................................................. 3
      1.2.1 Current Scenario ......................................................... 3
      1.2.2 iotX Vision ............................................................... 4
   1.3 Polyglot Persistence ............................................................ 5
      1.3.1 Polyglot Persistence - Overview ..................................... 5
      1.3.2 MongoDB Database ................................................... 6
      1.3.3 Neo4j Graph Database .............................................. 7
   1.4 Calvin Framework ............................................................ 8
      1.4.1 Calvin Overview ......................................................... 8
      1.4.2 Calvin Actor ............................................................ 9
      1.4.3 Calvin Application ................................................... 10

2 **Design** ............................................................................... 11
   2.1 iotX Architecture ............................................................. 11
   2.2 Modeling components of IoT Ecosystem ................................ 13
      2.2.1 iotXGraph Model ..................................................... 13
      2.2.2 iotXNode Model ....................................................... 13
      2.2.3 Sensor Model .......................................................... 14
      2.2.4 Actuator Model ......................................................... 15
      2.2.5 Actor Model ........................................................... 16
      2.2.6 Application Model ................................................... 17
   2.3 Modeling IoT Network Topology Graph ............................... 18
   2.4 Modeling Application Graph ............................................. 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>3 Implementation</strong></td>
<td>23</td>
</tr>
<tr>
<td>3.1 iotX Web Interface</td>
<td>24</td>
</tr>
<tr>
<td>3.2 iotX Data Storage Strategy</td>
<td>24</td>
</tr>
<tr>
<td><strong>4 Application Deployment</strong></td>
<td>27</td>
</tr>
<tr>
<td>4.1 Application Deployment Steps</td>
<td>27</td>
</tr>
<tr>
<td>4.2 Application Deployment Example</td>
<td>28</td>
</tr>
<tr>
<td><strong>5 Conclusion</strong></td>
<td>29</td>
</tr>
<tr>
<td>5.1 Current Status</td>
<td>29</td>
</tr>
<tr>
<td>5.2 Contributions</td>
<td>30</td>
</tr>
<tr>
<td>5.3 Future Work</td>
<td>31</td>
</tr>
<tr>
<td><strong>Bibliography</strong></td>
<td>32</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Data Processing in Cloud</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Data Processing in Cloudlets</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Data Processing in IoTX Framework</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>Polyglot Persistence</td>
<td>5</td>
</tr>
<tr>
<td>1.5</td>
<td>Calvin Architecture</td>
<td>8</td>
</tr>
<tr>
<td>1.6</td>
<td>Calvin Actor [4]</td>
<td>9</td>
</tr>
<tr>
<td>1.7</td>
<td>Calvin Application Script - Example</td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>IoTX Architecture</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>IoTXGraph Model</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>iotXNode Model</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>Sensor Model</td>
<td>15</td>
</tr>
<tr>
<td>2.5</td>
<td>Actuator Model</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>Actor Model</td>
<td>16</td>
</tr>
<tr>
<td>2.7</td>
<td>Application Model</td>
<td>17</td>
</tr>
<tr>
<td>2.8</td>
<td>IoT Network Topology Model</td>
<td>18</td>
</tr>
<tr>
<td>2.9</td>
<td>Network Topology Event Graph - E0</td>
<td>19</td>
</tr>
<tr>
<td>2.10</td>
<td>Network Topology Event Graph - E1</td>
<td>19</td>
</tr>
<tr>
<td>2.11</td>
<td>Application Graph Model</td>
<td>20</td>
</tr>
<tr>
<td>2.12</td>
<td>Application Graph - Before Actor Migration</td>
<td>21</td>
</tr>
<tr>
<td>2.13</td>
<td>Application Graph - After Actor Migration</td>
<td>21</td>
</tr>
<tr>
<td>2.14</td>
<td>Application Graph - Neo4j Example</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>Implementation</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Data Storage Strategy</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>Application Deployment Steps</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Example - Application Script</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Example - Calvin Application Graph</td>
<td>28</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction and Motivation

1.1 Motivation

IoT devices are now being used everywhere from Smart Home to Smart Cities. These devices generally communicate with Cloud for data processing and fetching required details. IoT devices with limited resources are also currently pushing data to cloud. With this, there is a dependency on reliable internet connection and cloud. We are proposing an iotX framework for reducing this dependency on cloud and utilizing resources across distributed IoT devices by carrying out light weight processing at the Edge. This will not only reduce the dependency but also can restrict the bandwidth consumption.

In IoT world, user can interconnect physical IoT devices and develop all kinds of applications by sharing and processing data based on the requirement. Before, setting up such an infrastructure in real world, there is need to design, customize, test the IoT network topology and application before hand. In iotX framework, user is provided with a web interface for customizing IoT network topology and launching different applications to test bed to verify the desired outcome before actually implementing the setup in the real world.

Interconnection of IoT devices and data flow information can be easily imagined as a graph architecture. On our web interface, user can make changes to the exiting network topology at different discrete-time events for running different applications. Thus, we have to store and update connectivities of such huge mesh network based on the user updates.
• **Storing initial configuration of different entities:** We want to store initial common structural information and basic configuration which can be used to build each IoTXGraph i.e. a single infrastructure of IoT devices such as smart home, smart city, industrial application, etc. The initial configuration of different entities like IoT node, sensor, actuator, actor store, links between IoT nodes, location, etc. include basic model or properties of such entities. For efficiently storing and retrieving such initial configuration we need document oriented database like MongoDB.

• **Tracking and Analyzing Events:** We also need to track (store and retrieve) and analyze user updates at discrete time events. We’ll use graph databases such as Neo4j in this case, since maintaining nodes and edges structure is easier in graph databases when compared to other persistent databases. Also, graph databases like Neo4j have built-in algorithms for quick path traversals and problems like pattern matching can be effectively handled here.

• **Data processing and Analysis:** Since, we want to get some data processing back to Edge, we have to let IoT devices communicate with each other for message passing and task allocation. We are using one light-weight open source Calvin framework developed by Ericsson Research team for data processing and analysis. Calvin framework provides a runtime environment for IoT devices to communicate with each other [7].
1.2 iotX Framework

1.2.1 Current Scenario

Most of the IoT devices are sending data to cloud for data processing [12]. This requires reliable internet connection and cloud support. If every IoT device starts depending on cloud even for the light-weight data processing then,

- We are wasting resources available across distributed IoT nodes
- Congesting the network unnecessarily
- Increasing the cost since, cloud usage isn’t free
Consider the above figure, here Cloudlets i.e. small cloud data center available at the Edge is used for light-weight data processing. This setup reduces the dependency on Cloud and utilizes computational power available at the Edge.

1.2.2 iotX Vision

Figure 1.3: Data Processing in iotX Framework

In an iotX Framework, we are trying to utilize resources available at the Edge as much as possible. Based on the scale of data processing task in hand, we’ll decide whether to carry out data processing at the Edge or cloud.

If available resources at the Edge can handle a particular task of data processing, then we are not pushing data to cloud. If the available resources at the Edge can not handle a particular task, then we will be pushing it to Cloud. This framework plans to have dynamic task allocation across distributed IoT nodes to carry out data processing. This approach is ensuring the utilization of resources at the Edge along with support for large computations from cloud.
1.3 Polyglot Persistence

1.3.1 Polyglot Persistence - Overview

What if I want to use multiple database technologies?

![mongoDB](image1.png) ![neo4j](image2.png)

Figure 1.4: Polyglot Persistence

Using multiple database technologies together to take their corresponding advantages is termed as Polyglot Persistence [9]. Many times, based on the different requirements, using more than one database makes more sense.

- In this project we are using Document based database MongoDB for storing initial structured info. of different entities like Actor Store, Sensor, Actuator etc.

- Neo4j is a scalable graph database which is used in this project to maintain relationships between IoT nodes in network topology, Application-to-Actor nodes in application graph at each event and for efficient path-associative queries, path computations since Neo4j graph database excels over other NoSQL databases in these aspects.

We’ll now discuss more about MongoDB and Neo4j NoSQL databases; their basic differences, advantages and limitations in detail.
1.3.2 MongoDB Database

NoSQL databases are different from traditional relational databases like SQL which needs well defined schema before you insert anything in the database. NoSQL databases are schema-less or support dynamic schemas, you can insert data for additional columns any time into the database. That’s why, it is considered as a highly scalable option to store unstructured data. MongoDB is one such NoSQL database which is document (JSON style document) oriented i.e. stores unstructured data in the form of documents not tables (in case of relational databases). User doesn’t have to define document design upfront. MongoDB database is easy to access since, user doesn’t have to know the schema beforehand or learn query language like SQL to fire queries [8].

- **Advantages:**
  - Schema-less and document oriented
  - Highly Scalable
  - Easy to access
  - Better performance over traditional relational databases
  - Driver support for major languages

- **Limitations:**
  - Can’t enforce data integrity rules like relational databases
  - Complexity in handling write operation on multiple documents
  - Not suitable for storing information of node-edges form
  - Difficult to write path-associative queries
1.3.3 Neo4j Graph Database

Graph databases are also one of the NOSQL databases where data is stored in the form node and relationships. The nodes and relationships have required properties i.e. attributes. Other document Oriented, column Store or key-value store based NoSQL databases are not suitable for connected data and graphs. Storing data in graph format allows user to view the same data and connections from different perspectives. This network of data and relationships helps in finding optimal paths, pattern matching or providing recommendations. Neo4j also provides built-in graph computing algorithms thereby providing better results in terms of performance when it comes to path-associated queries and computations [11].

- Advantages:
  - Thinking in connected data graph comes naturally
  - Neo4j provides transactional support
  - Connected data provides different insights
  - Effective in path-associative queries
  - Built-in graph algorithms
  - Comply with ACID properties

- Limitations:
  - Modeling a graph in Neo4j is very crucial, bad modeling the data in Neo4j Graph can’t fetch expected results
  - Partitioning the data in graph is difficult hence, can’t be parallelized easily.
  - Database tables with no relations through foreign key references are not a good fit for transforming into Neo4j Graph database.
1.4 Calvin Framework

1.4.1 Calvin Overview

Calvin is a light-weight open source framework developed by Ericsson Research team with a runtime environment for IoT devices to communicate with each other. We are integrating with this framework to achieve data processing at the Edge where IoT nodes can communicate with each other [7].

Figure 1.5: Calvin Architecture
1.4.2 Calvin Actor

- Actor in Calvin framework is a set of well-defined instructions for performing actions. These are python files with required instructions to be executed [2].

- There are pre-defined actors available in Calvin framework. User can write their own Actors by inheriting Actor class.

- Calvin framework also provides different modules of related actors along with Actor Docs which has the documentation for these actors [1].

- Actors run inside a Calvin Runtime which provides a required environment setup. Actors can be migrated from one runtime to other if needed and actors take their saved state while migrating to other runtime [3].

Application - Defines data flow between actors
Actor - Performs designated actions

Figure 1.6: Calvin Actor [4]
1.4.3 Calvin Application

- Calvin application is formed by set of such actors connected based on the data flow information to carry out certain function [3].

- The application doesn’t change throughout the application lifecycle. The Actor sitting inside a runtime on a particular IoT node can migrate to other runtime but, it won’t affect the application. Each actor and actor instances in an application can be uniquely identified [3]

- Application script captures the Actor nodes and their connections through different ports. First part of the application script has actor, actor type, description and next part describes the data flow through different ports. Here is an example of one such calvin application script, where three sensors are emitting their temperature readings and Actor ‘avg’ is calculating the average and sending it to ‘out’ actor to print.

```c
/* Actors */
tempSensor1 : sensor.TriggeredTemperature()
tempSensor2 : sensor.TriggeredTemperature()
tempSensor3 : sensor.TriggeredTemperature()
avg: avg.calculateAverage()
out : io.Print()

/* Connections */
tempSensor1.token > avg.first
tempSensor1.token > avg.second
tempSensor1.token > avg.third
avg.result > out.token
```

Figure 1.7: Calvin Application Script - Example
Chapter 2
Design

2.1 iotX Architecture

- In iotX Architecture, for large scale data processing data can be pushed to Cloud data processing Engine (Apache Spark) while Calvin framework is used if low-scale data processing is to be executed on available resources at the Edge.
• User can deploy the applications on testbed to check for desired results. iotX front-end provides all kinds of customizations to the user starting from creating an IoT network topology, binding actors, adding and deleting a particular application.

• The information on front-end is extracted from the MongoDB and Neo4j databases in the back-end through web-handler to support different requirements which are discussed in the Implementation section of this report.

• We have modeled each entity related to IoT ecosystem with minute details. The back-end also has implementation different path optimization algorithms.

• Docker containers with Calvin installation are used to mimic each runtime so that user can test the application. Docker containers are fired when user starts the application. This also allows the system to scale up since, many docker containers can be fired based on the application requirement.
2.2 Modeling components of IoT Ecosystem

2.2.1 iotXGraph Model

- iotXGraph is a IoT network setup where user has customized the topology and resources on each IoT node. While generating a iotXGraph, user has to mention number of IoT nodes he wants to have in the network, iotXGraph name by which he’ll uniquely identify the iotXGraph.

- iotXGraph also has unique GUID across the system. Grid Size X, Y, Z (Currently, Grid Size Z is set to ’0’ as we are showing in only with 2 dimensions) values will help us in setting a required layout for the IoT network. Connectivity Ratio signifies, how fully connected your network is.

![Figure 2.2: iotXGraph Model](image)

2.2.2 iotXNode Model

- iotXNode is representing the actual IoT node in our system. Each iotXNode has unique GUID across the system for unique identification.

- Quadkey is a representation of particular location on the grid based on the level of detail. Each iotXNode has this quadkey which helps in locating a particular IoT node on the grid.
• iotXNode type specifies which IoT node type it mimics. It can be Rasberry Pi, Odroid, etc. If we assign Rasberry Pi as the node type then, it will have all the resources and attributes based on the node type model. Currently, We are using Simulated node type as we are testing the setup in our system first.

• iotXNode has different resources such as sensors, actuators, actors, etc. along with list of incoming/outgoing links and constraints.

![Figure 2.3: iotXNode Model](image)

2.2.3 Sensor Model

• Sensor Model captures all the hardware details such as manufacturer, model number, serial number, sensor type, sensing type, voltage, sensitivity, accuracy, confidence, minimum and maximum sensing range, etc. of particular sensor along with unique GUID.

• Sensing type of sensor can be temperature sensor, humidity sensor, wind speed sensor, etc. Based on our requirement, we can add a particular type of sensor to IoT node.
2.2.4 Actuator Model

- Actuator Model captures all the hardware details such as manufacturer, model number, serial number, actuator type, actuating type, voltage, minimum and maximum operating range, etc. of particular actuator along with unique GUID.

- Actuating type of actuator can be LED actuator, servo motor actuator, speaker actuator, etc. Based on our requirement, we can add a particular type of actuator to IoT node.
2.2.5 Actor Model

- As an actor can be used in any application with different port information, we are modeling Actor with only name and operation details.

- Set of such actors captured in MongoDB can also be picturized as an Actor Store. Same Actor can be used in multiple applications since, while modeling we haven’t added any specific port information. Based on the application, the port information can be added.
2.2.6 Application Model

- Modeling of an Application include application name, number of actors involved, entire application script (it will be used while integrating with Calvin framework), actor connections information.

- Each actor connection is a set of Actor to particular iotXNode binding, port information associated to that actor.

- Each port information captures port name, connections to ports of different actors.

![Figure 2.7: Application Model](image-url)
2.3 Modeling IoT Network Topology Graph

- IoT network topology in real world is nothing but IoT nodes connected to each other based on the setup requirement. This network topology will change at different events based on the user actions.

- We are storing all topology changes generated by user updates at each event. While modeling the topology in Neo4j, we are representing each IoT node as iotXNode, the connections between IoT nodes become IoTXNode Connection and finally, we are connecting all IoT nodes to one central event node (e.g. T-0) which helps in identifying the topology for a particular event. In next two diagrams, we are capturing topology changes in Neo4j for different events. User has changed Actor to iotXNode bindings in E0 application graph which is when saved can be seen in event graph E1.
Figure 2.9: Network Topology Event Graph - E0

Figure 2.10: Network Topology Event Graph - E1
2.4 Modeling Application Graph

- Application graph can be represented as set of actors connected to each other through specific ports. In above example, Actor2, Actor3, Actor4 are emitting their respective sensing values to Actor1. Actor1 is collecting this data at different ports and showing the sensed values on the LED.

- We are capturing the Calvin application graph by creating one application node and connecting the actors to that application node through App Connection relation. Actors are connected to each other by Actor Connection relation. Each Actor can be seen inside a runtime. Consider if the Pi-3 IoT node fails, Actor3 can be migrated to Runtime2 which can be seen in next two figures.

Figure 2.11: Application Graph Model
Figure 2.12: Application Graph - Before Actor Migration

Figure 2.13: Application Graph - After Actor Migration
• In Neo4j graph database, application graph is captured like this. The purple node at the center is application node with application name as one of the properties.

• The nodes in yellow color are actor nodes connected to central application node by Application Connection relation.

• Actor to actor nodes are connected by relation with name as follows e.g. mul.result > math.token;
  'mul' and 'math' are actors where as 'result' and 'token' are their respective ports.
Chapter 3

Implementation

![Diagram of IoTX web interface and back-end communication]

Data Transmission in JSON format
CRUD operations for maintaining topology and application graph

Figure 3.1: Implementation

IoTX web interface talks with back-end through web handler. Web handler acts as a RESTful service handler which accepts CRUD operation requests from UI for maintaining topology and application graph. Then, it calls appropriate back-end methods to access MongoDB or Neo4j. Since, we are using Neo4j graph database to store and retrieve IoT network topology and application details at each event, for Update and Retrieve requests Neo4j database will be accessed. MongoDB stores initial structural information for all entities thus, for Create and Delete requests web handler will access MongoDB first. Neo4j database will fetch MongoDB details while updating first graph from initial topology graph.
3.1 iotX Web Interface

- IoTX web interface is designed using HTML and CSS. item JavaScript, jQuery, Ajax are used for event handling, animation and UI interaction.

- Cytoscape.js library is used for graph visualization on web interface.

- Currently, web interface allows user to create, update, retrieve and delete IoT network topology. It also provides create, start, stop and delete options for maintaining applications.

- Web interface also offers an option of accessing topology and Calvin application graph at past events to provide a flexibility of designing and utilizing the topology to the user.

3.2 iotX Data Storage Strategy

<table>
<thead>
<tr>
<th>Memory</th>
<th>MongoDB</th>
<th>Neo4j</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Store basic info. (such as GUID) of IoT nodes</td>
<td>- Store details of IoT nodes, including info. regarding attributes of nodes, links, sensors, actuators, actors and constraints.</td>
<td>- Store IoT network topology and Calvin apps details.</td>
</tr>
<tr>
<td>- Cache frequently used objects to reduce accessing DB.</td>
<td></td>
<td>- Support efficient graph-based queries and computations.</td>
</tr>
</tbody>
</table>

Figure 3.2: Data Storage Strategy

- The amount of data we are processing and storing can’t be handled entirely in main memory. Thus, we are limiting the main memory to only store unique GUIDs of IoT nodes while we are generating MongoDB collection documents for the same.
• Data caching is also implemented to store frequently requested node details to reduce the unnecessary DB calls made to retrieve similar information.

• Since, we have different requirements which can be satisfied efficiently by different database technologies, we making use of MongoDB along with Neo4j.

• To mimic a real IoT node in our system, we have modeled each entity related to a real IoT node like Sensor, Actuator, Actor, constraints considering their fine details in MongoDB.

• Neo4j is a scalable graph database which is used to maintain relationships between IoT nodes, Application-to-Actor nodes at each event and for efficient path-associative queries, path computations since Neo4j graph database excels over others in these aspects.

• When an user creates a network topology of IoT nodes, we instantiate all the resources and objects according to the user requirements. User can update the network topology which can be captured as a discrete-time event in the system using Neo4j graph database.

• Neo4j stores data in graph (node-relationships) format whereas other NoSQL systems are disconnected aggregates which are not considered as a well-thought option if you want to store connected data and graphs. It also provides built-in graph computing algorithms thereby providing better results in terms of performance when it comes to path-associated queries and computations.

• **Batch Insertion:** We are making use of BatchInserters in Neo4j for adding IoT nodes and connections as the IoT nodes count can be as big as 1000 and with their interconnections, it becomes a very complex structure. BatchInserter adds all the nodes and relationships in batches by ignoring transactions and concurrency support to achieve best insertion speed [5].
• **GraphDatabaseService**: With this mechanism we are performing CRUD operations, running cypher queries on our database. It also provides transactional support. For all user updates related to each iotXGraph setup, we are capturing it under one Neo4j file store. Since, user can query for viewing any particular graph from past events, while storing each updated instance of our setup we are adding one central node T-0, T-1, etc. to easily locate it in database [6].

• GraphDatabaseService loads a particular neo4j file store at a time, if we store different files for each update that user does, then well have to load each and every file again thereby increasing time complexity. Also, if a user is working on a same iotXGraph performing different CRUD operations, we prefer not to shutdown the GraphDatabaseService for that file store till user starts working on the other iotXGraph. This also saves the loading time, GrapahDatabaseService takes in general.

• Path computing takes minimal time as we have stored our data in the form of Neo4j graph. This also helps in analyzing the event updates and patterns effectively.
Chapter 4

Application Deployment

4.1 Application Deployment Steps

Figure 4.1: Application Deployment Steps
4.2 Application Deployment Example

/* Actors */
ten : std.Trigger(data=10, tick=0.5)
five : std.Trigger(data=5, tick=0.5)
sum : math.InputSum()
sub : math.InputSubtract()
mul : math.InputMul()
out : io.Print()

/* Connections */
ten.token > sum.first
five.token > sum.second
ten.token > sub.first
five.token > sub.second
sum.result > mul.first
sub.result > mul.second
mul.result > out.token

Figure 4.2: Example - Application Script

Figure 4.3: Example - Calvin Application Graph
Chapter 5

Conclusion

5.1 Current Status

- **Modeling Components:** Current System has basic models for iotXGraph, iotXNode, Actor, Actuator, Sensor, iotXLink, location. iotXNode model also has specialized model for mimicking Rasberry Pi and Odroid.

- **Web Interface:** Currently, web interface allows user to create, update, retrieve iotX-Graph and application graph details. User can currently view simulation or event graph based on the selection. The slider in the web interface provides traversing through past events. Information of components like Sensor, Actuator can be viewed in separate panels. There is a separate panel for listing the applications with start, stop, delete options. User can dynamically add the application with application script. Actors binded to a particular iotXNode can be seen and those can be migrated to other iotXNodes as well.

- **Back-end:** MongoDB and Neo4j are used together to support different requirements. To mimic a real IoT node in our system, we have modeled each entity related to a real IoT node like Sensor, Actuator, Actor, constraints considering their fine details in MongoDB. Neo4j scalable graph database is used to track user updates to IoT network topology and application graph at each event and for efficient path computations.
5.2 Contributions

- Modeling Components - Modeled Actor and Application entities in the code base, updated the existing models for iotXGraph, iotXNode, Sensor and Actuator.

- Neo4j graph Database - Implemented following features:
  - Simulation Graph - Build, Update and Retrieval
  - Event Graph - Build, Update and Retrieval
  - Application Graph - Build, Update, Retrieval and Delete
  - Erasing all or selected graph files
  - Getting shortest path between any two IoT nodes
  - Neo4j database converters for all entities

- MongoDB Database - Building MongoDB collection using structured information for every entity based on their respective models. Adding database converters for few entities.

- Front-end: Designed the user interface using HTML, CSS. All the user interactions, animation and event handling is done using JavaScript, Ajax, jQuery.

- Web Handler: Web handler acts as a RESTful service handler, all the user requests like storing, retrieving IoT network topology at each event, adding/updating/deleting applications, updating Actor to iotXNode actor bindings, etc. are fulfilled. The data is received and sent from the web handler in the JSON format.
5.3 Future Work

- Implementing dynamic task allocation algorithm which will assign data processing tasks based on the resources available on distributed IoT nodes.

- Using Neo4j to do pattern matching for identifying similar applications in network. Customizing heuristics for finding optimal path between given set of nodes.

- Using Neo4j Doc Manager to fetch MongoDB documents and query them when needed to convert it into required Neo4j graph structure.

- Building fault tolerance in the system where the tasks allocated to a node which failed in runtime should be reallocated to other eligible node.

- Implementing generic process models which can couple multiple processes together and can support different functionalities based on the input parameters.

- Adding different layouts such as room layout, floor plan layout on to the grid in UI where we are currently showing iotXGraph.

- Adding more specialized models for supporting different IoT nodes, sensors, actuators, etc. which are currently available in the market.
Bibliography


