Data Modeling And Visualization For The iotX Project

by

Ajeeth Kannan

Supervised by

Dr. Peizhao Hu

Submitted to the
B. Thomas Golisano College of Computing and Information Sciences
Department of Computer Science
in partial fulfillment of the requirements for the
Master of Science Degree
at the Rochester Institute of Technology

Abstract

Internet of things (IoT) has been widely used in many applications in the recent days. The usage of IoT devices have increased the research interest on IoT among many computer scientists. The current IoT devices do not talk to each other. They are heavily depending on cloud APIs to exchange information. But, this approach has the following issues like higher bandwidth usage, dependency on network connectivity and data processing latency. iotX is a scalable framework for IoT which overcomes the above issues by implementing in-network processing using Calvin framework. IoTX also has a web application which supports user customizable application deployment. This project focuses on creating a visualization framework for the IoTX project to visualize the IoT network in the browsers.
Acknowledgments

I would like to thank Professor Peizhao Hu for suggesting the idea of this project and supervising me to complete all the deliverables. I would like to thank Sai Ashwin Parakkal, Vinay Vasant More and Sahil Jasrotia for tipping me with their ideas to achieve the deliverables for the project. Lastly, I would like to thank my parents for supporting and motivating me throughout my education.
Contents

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

2 Background ........................................... 3

  2.1 Calvin ............................................. 3

  2.2 Spring Data ........................................ 4

    2.2.1 MongoDB ...................................... 5

    2.2.2 Spring Data MongoDB ......................... 5

  2.3 Jersey – RESTful Web Services in Java ............. 7

    2.3.1 RESTful API .................................... 7

    2.3.2 JAX-RS ........................................ 9

    2.3.3 HTTP Server Using Spring Boot ................ 9

  2.4 Cytoscape JS ....................................... 9

  2.5 Other Front-end Frameworks and Libraries .......... 10

    2.5.1 Asynchronous JavaScript and XML .............. 11

    2.5.2 jQuery – A JavaScript Library ................ 11

    2.5.3 Bootstrap ..................................... 12

    2.5.4 qTip .......................................... 12
CONTENTS

3 Overview Of iotX 14

4 Implementation 17

4.1 MongoDB ......................................................... 17

4.2 Front-end .......................................................... 21

4.2.1 Welcome Page .................................................. 21

4.2.2 Topology Page .................................................. 23

5 Conclusion 35

5.1 Future Work ....................................................... 35
List of Figures

2.1 An example of document data structure. 6
2.2 RESTful API Design. 8
2.3 An example of qTip. 13
3.1 Overview of iotX. 15
4.1 Architecture of MongoDB design in the back-end. 18
4.2 Architecture of iotX topology database. 20
4.3 Welcome page of iotX web application. 21
4.4 Modal to create graph. 22
4.5 An example of iotX topology after loading on to Cytoscape JS. 25
4.6 Navigation bar to select the mode. According to the mode listeners to the event-graph will change. 25
4.7 Left figure: The list of actors in a selected node. Right figure: List of actors in math application that have bound to the selected node. The actor names are specified along with the application name. Application name and actor name are separated by dot symbol. 26
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.8</td>
<td>An example of displaying a qTip with information about iotX node on right click or two finger tap action. 27</td>
</tr>
<tr>
<td>4.9</td>
<td>Left figure: The topology which was stored in the back-end. Right figure: The topology after user moving the positions of the nodes. The newly edited topology will be saved in the back-end. 28</td>
</tr>
<tr>
<td>4.10</td>
<td>The event-time navigation button for viewing the event-graphs. 29</td>
</tr>
<tr>
<td>4.11</td>
<td>The application table in topology page to the details of Calvin applications. 30</td>
</tr>
<tr>
<td>4.12</td>
<td>An example Calvin application which predicts the probability of failure of a wind turbine using linear regression. 31</td>
</tr>
<tr>
<td>4.13</td>
<td>An example IoT network topology and the corresponding mapping of actors for figure 4.12 to the runtimes created on the selected IoT nodes. 32</td>
</tr>
</tbody>
</table>
List of Tables

4.1 The RESTful API details for welcome page . . . . . . . . . . . 23
4.2 The RESTful API details for topology page . . . . . . . . . . . 34
Chapter 1

Introduction

Internet has been evolved in the past two decades drastically. Initially, messages were sent between the machines in a network with the support of human interactions. As internet had grown exponentially, researches wanted to minimize the human support between the network devices. This is the place where Internet of things (IoT) comes into picture. IoT is a system of interrelated computing devices that are able to exchange data. Most importantly, they have the ability to function with the minimal support of human interactions. Smart cities, smart homes and connected cars are few applications of IoT.

In the recent days, lots of research is being done on IoT devices interacting with each other. Currently IoT devices use cloud APIs to talk to each other. So, IoT devices heavily depends on the cloud. This approach has problems like heavy dependency on network connectivity, data processing latency and higher bandwidth usage. iotX is a scalable framework for IoT which attempts to solve the above issues by implementing in-network processing using Calvin framework [1]. iotX is an open source project which aims to provide edge
computing by reducing the cloud usage among the network. It also provides user customizable application deployment. So, user can create any kinds of IoT network of his/her wish and run the applications within the network. In the current implementation user can specify the number of nodes, types of sensors and types of actuator to create an IoT network. But in future, we are aiming to create a fully customizable user defined topology. A topology is represented using graphs where nodes specifies each IoT device and edge specifies the connection between the devices.

iotX also has the web support where the user can visualize the network topology in the desktop and mobile browsers. It also has databases to persist the topologies, so that the user can retrieve the topology any time. My major contribution towards the iotX project is to develop a visualization framework along with web support for the topologies that are running in iotX. And, I have also contributed towards storing data in the back-end. This report focuses on explaining the user interface components in the front-end, frameworks that are used to visualize the network topology, data storage structure, creating an application in iotX and REST API calls that are made to the back-end.
Chapter 2

Background

iotX is built using different frameworks. But, this report would focus only on the background of the frameworks in detail that I have worked.

2.1 Calvin

Calvin is developed by Ericsson research and it is an open source project since 2015. Calvin is an application environment which simplifies the communication between the devices. It has a development framework for application developers and a runtime environment like JVM for handling the applications. Calvin is build using the well-established actor model. Each Calvin application will have different actors communicating to each other according to the specification (Calvin script) in an application. Instead of developing a framework of our own, we have used Calvin Framework to create applications. The user can create and run any number of Calvin applications on a topology. The actors can be bound to any of the nodes in an iotX network. While running an
application, iotX will create separate runtime using docker for each node that had bound to actors. All information about the application including actor-node binding will be persisted in the back-end. The Calvin framework also has a web support named csweb which is used to debug the Calvin applications. We have integrated csweb with our UI to view and edit the runtime and application information in the browser. More details about Calvin framework could be found in Calvin GitHub.

2.2 Spring Data

iotX uses databases in the back-end to persist the data. The reasons are as follows

- User can retrieve the data whenever he/she needs.

- While running a larger topology, there are possibilities of crashing when the program runs. The reason behind this is, the program consumes lots of in-memory to store the details about each component of a topology.

Because of the above reasons, every single detail about each component will be stored in the back-end and retrieved whenever it is required. We have used both MongoDB and Neo4j in the back-end. Neo4j is a graph database which does not store all the details about a node. It just stores the basic detail of a node to capture the changes made by the user on the topology. Neo4j also has implementations for various graph algorithms which can be used for path computations [3]. But, Neo4j stores data as a graph but our implementation required a database which allows structural data storage. So, we decided to
also use MongoDB [4]. As I have implemented functionalities to store details in MongoDB, this section will only explain about MongoDB.

### 2.2.1 MongoDB

MongoDB is exceedingly available, highly performable, automatic scaling and an open-source document database. Each record in MongoDB is stored as a document [5]. Document is a data structure which is composed of field-value pair and it is similar to JSON objects. A document can have another document array inside. Figure 2.1 shows an example of a document. Documents are stored inside collections. Collections in MongoDB are analogous to tables in relational database [5]. NoSQL databases were created to overcome the shortcomings of the relational databases. Reasons for using a NoSQL over SQL database(MongoDB) are as follows [6]:

- NoSQL database supports structured, semi structured and unstructured data.
- Flexible, easy to use, agile and iterates faster.
- The architecture can be scaled out any time instead of costly and inflexible architecture.

### 2.2.2 Spring Data MongoDB

Spring Data for MongoDB [7] is part of the umbrella Spring Data Project which provides spring based programming model. Storing and retrieving data using Spring data MongoDB is simple and easy which provides repository
style data access layer. Because of the easy implementation models, we have used Spring Data MongoDB in our iotX project. The features of Spring Data MongoDB are as follows [7]:

- It supports Java based Querying and Criteria.

- Metadata is mapped using annotations and it also widely supports other metadata formats.

- It has MongoTemplete helper class which efficiently performs mongo operations.

- It also has GeoSpacial and Map-Reduce integrations.

- Java based @Configuration classes are used to support Spring Configuration.

```
{
    NodeID: N1
    NodeType: Simulated Node
    SensorList: [
        {
            SensorID: S1
            SensorType: Simulated Sensor
        },
        {
            SensorID: S2
            SensorType: Simulated Sensor
        }
    ]
}
```

Figure 2.1: An example of document data structure.
2.3 Jersey – RESTful Web Services in Java

2.3.1 RESTful API

RESTful API services has been implemented in iotX to communicate from the back-end to the front-end. API (Application Programming Interface) defines a set of rules to interact between applications. API is flexible which can return the data in the format (e.g. JSON or XML) that the client needs [8]. REST (Representational State Transfer) is not a protocol or a standard. It specifies the approach or the architecture for writing the API. Combining both the definitions of REST and API; REST is an architectural style, RESTful is an interpretation. If the back-end server has REST API and a request is made using API in the client-side, then it is RESTful. Figure 2.2 shows the design for RESTful API. Creating a new data, updating the data that exist in the back-end, retrieving the data and deleting the data are four essential operations of RESTful API. REST is dependent on HTTP. CURD i.e. Create, update, retrieve and delete uses its own HTTP methods [8].

- Retrieve uses GET method from HTTP.
- Create uses POST method from HTTP.
- Update uses PUT method from HTTP.
- Delete uses DELETE method from HTTP.

Each request that made from the client have a status code. The status codes are divided into five classes. The first number of the status code indicates which class it belongs to [8].
CHAPTER 2. BACKGROUND

- 1XX - informational.
- 2XX - success.
- 3XX - redirection.
- 4XX - client error.
- 5XX - server error.

Figure 2.2: RESTful API Design.
CHAPTER 2. BACKGROUND

2.3.2 JAX-RS

Java API for RESTful Web Services (JAX-RS) is a Java programming API specification which provides support to implement web services according to REST architectural style [9]. Jersey is an implementation of JAX-RS which supports for the annotations defined in JSR 311 [10]. Jersey makes the programmers easy to build RESTful Web Services. So, we have used Jersey to support RESTful Web Services in IoTX.

2.3.3 HTTP Server Using Spring Boot

Spring Boot is embedded with Tomcat and Jetty which helps for easy creation of a HTTP server. Spring-boot-starter-web can be used to create and run an application quickly [11]. Spring Boot also supports Jersey. So, we have used Spring boot to start a HTTP server for web support.

2.4 Cytoscape JS

The iotX topologies are visualized in the front-end using Cytoscape JS. The features of this framework are mentioned below [12].

- A graph library written completely in JavaScript.
- It is compatible with all the modern browsers.
- Graph analysis is also possible by utilizing the graph theory algorithm implementation in Cytoscape JS.
- The the graphs viewed using this framework can be completely wrapped and unwrapped using JSON.
• Filtering and querying of particular graph elements among all the elements in the graph can also be performed.

• The viewport and graph elements supports animation.

• The standard gestures for both desktop and touch is also provided.

• As it is an open source project, the features of Cytoscape JS can be enhanced according to the need.

By considering all these features we have decided to use Cytoscape JS to view the graphs (topology) in the front-end. The front-end requires graph visualization in three places

• Viewing and editing the topology graphs.

• Viewing and migrating the actors to iotX nodes.

• Viewing the applications.

We can customize size, shape, visuals, connections, filtering, querying and animations for nodes, elements and viewport of a graph using Cytoscape JS. The usage of this framework in iotX project will be explained in detail in the next chapter. Cytoscape JS is a vast implementation for graph library, the readers might not get exposed to all the functionalities in this report. So, to get more information Cytoscape can be followed.

2.5 Other Front-end Frameworks and Libraries

This section covers the couple of few other most commonly used frameworks in the front-end.
2.5.1 Asynchronous JavaScript and XML

Asynchronous JavaScript and XML (Ajax) is a web development technique in client-side to create asynchronous web applications [13]. Using Ajax a client can send and retrieve data without interfering the display and the property of the existing page. All the Ajax actions are run in the background (asynchronously). Even though Ajax represents XML, the modern day implementations uses JSON (advantage of being native to JavaScripts) instead of XML. All the communication from the browser to the iotX web-server is done using Ajax calls.

2.5.2 jQuery – A JavaScript Library

jQuery is one of the most commonly used JavaScript libraries [14]. The features of jQuery are

- jQuery’s syntax is designed to be simple and it is fast and feature-rich [14].

- It supports HTML document traversal, event handling, animation and Ajax calls much simpler.

- It works on multiple modern browsers.

- Developers can create plug-ins on top of JavaScripts library.

The modern web developments are done using jQuery. Because of the simple function calls and easy implementations, most of the JavaScript functions in this project were written using jQuery.
CHAPTER 2. BACKGROUND

2.5.3 Bootstrap

Bootstrap is a front-end web framework and it is used for designing web applications. It contains design templates for forms, buttons, navigations and many other user interface components [15]. It also has JavaScript extensions. The JavaScript components in this framework come in the form of jQuery (A JavaScript library). Bootstrap is supported in almost all the modern browsers. The user interface components in the iotX front-end is implemented using this framework.

2.5.4 qTip

qTip is used to provide additional messages when the cursor is placed or clicked on a component. In this project, qTip is used to show details about the node. Figure 2.3 shows an example of qTip, that appears on a click action on a node [16].
Figure 2.3: An example of qTip.
Chapter 3

Overview Of iotX

The figure 3.1 shows the overview of iotX. The iotX framework can be divided into five modules (i.e.) Front-end, back-end, docker, cloud and Calvin framework.

**Front-end**  The iotX web application provides the user with the complete control over the IoT network from the front-end. The front-end module takes care of the user customizable application deployment and iotX topology deployment.

**Back-end**  The back-end manages the databases, API services and core model of the iotX.

**Core Model**  The iotX Model has the implementation for nodes, links between the nodes, sensors, actuators and actors.
**Database**  The graph data is stored in MongoDB and Neo4j to utilize the corresponding advantages of both the databases.

- In this project we required a document database for storing the structured information of iotX topology. So, We are using MongoDB to persist the data.

- We required a database which supports graph theory algorithms for path computations and maintains the relationship between the sensors, actuators, nodes, applications and actors. Neo4j is used as it is scalable graph database and maintains relationships between the iotX components efficiently.

![Figure 3.1: Overview of iotX.](image-url)
API service

- The web handler is implemented using Jersey (RESTful API) which takes care of the communication between the front-end and the back-end.

- The dockers are handled from the back-end using docker API. It has functionalities to create dockers for the IoTXNode, run Calvin applications, stop applications and destroy dockers.

- The Calvin control library is a RESTful API service for the to call the Calvin functionalities from the back-end.

- Cloud API is not currently implemented as the current work does not have support with the cloud data processing. But in future all the APIs for cloud will be implemented in Cloud API.

Calvin Framework  The iotX components are integrated with Calvin framework to support data processing in the edge devices.

Docker  Dockers are used to get a real-time experience with launching applications. While running an application, iotX will create separate runtime using docker for each node that had bound to actors.
Chapter 4

Implementation

4.1 MongoDB

The architecture of MongoDB design is shown in figure 4.1. We wanted to design a structure which requires simple queries to update and retrieve details from the database. So, we have used separate database for each topology. The topology database has a list of collections. The collection details are shown below.

- **IoTXGraph** - The IoTXGraph collection stores the details like number of nodes, list of nodes, list of added applications, length and width of room plan. GUID will be an unique id for each component in a topology.

- **IoTXNode** - Each document (IoTXNode) in the IoTXNode collection stores details like GUID of the node, node type (example: simulated node), position of the node, list of sensors, list of actuators, list of actors bound, list of incoming links from other IoTXNode, list of outgoing links...
to other node, capabilities of the node (example: WiFi range) and list of constraints to a node. Positions are stored in quadkey format of Bing Maps Tile System.

- IoTXLink - IoTXLink collection is referenced from IoTXNode collection. Each IoTXLink will also have a list of constraints. Every document stores details like GUID of the link, GUID of the source node, GUID of the destination node, type of the link (wired or wireless) and list of constraints.

- Sensor - The sensor collection is referenced from IoTXNode collection.
CHAPTER 4. IMPLEMENTATION

Details like GUID of the sensor, manufacturer, model number, serial number, sensor type, sensing type (example: light, distance, temperature), voltage, confidence, accuracy, sensitivity, minimum sensing range and maximum sensing range.

- Actuator - The actuator collections is referenced from IoTXNode collection. Each document has details like GUID, manufacturer, model number, serial number, actuator type, actuating type, voltage, minimum operating range and maximum operating range.

- Constraint - Constraint collection is referenced from IoTXNode collection and IoTXLink collection. Constraints can be used to analyze a link or a node by giving a constraint value on their capability. This collection stores details like GUID, constraint type (node constraint or a link constraint) and relation (greater than, less than, equal to, greater than or equal to, less than or equal to).

- Application - Application collection will have the details about the applications that were added to that particular topology on which we are running the experiment. This collection is referred from IoTXGraph collection. The details like actors on this application, nodes with which the actors are being binded and application script would be stored in this collection.

- Actor - Actor collection is referenced from IoTXNode collection and application collection. Details like GUID (GUID of the actor from actor store along with the application name on which the application is run-
Acting (ing), Actor name and list of outgoing ports to other actors along with the actor name are stored.

The architecture of iotX topology database is shown in figure 4.2. We also have a database named iotX which stores the list of topologies and basic details like number of nodes, grid size for the floor plan and id of each nodes. We required a separate database to store the actor details as Calvin actor scripts are common for all the topologies. Neo4j takes initial information of all the components from MongoDB. When user edits the topology, each save after the edit is stored as an event. All these event information is stored in

Figure 4.2: Architecture of iotX topology database.
Neo4j also. This report will not cover about Neo4j as I have not contributed to Neo4j.

4.2 Front-end

The iotX front-end is written in HTML, CSS and JavaScript which includes three pages. All the calls from the front end is done using Ajax and the data is sent in JSON format. Each component in the front-end is implemented using Bootstrap framework. The pages are welcome page, topology visualization page and csweb page (Calvin debugger). This section will cover the implementation details and flow of the interface.

4.2.1 Welcome Page

The iotX server runs on port number 8080. The user can type <ipaddress of the server>:8080 to start the welcome page. Figure 4.3 shows the user interface
of the welcome page. The functionalities of welcome page are discussed below.

Creating an iotX topology. When user clicks on the Create Graph button, a model will appear. The modal is shown in figure 4.4. The user can specify name, number of nodes, height and width of the room plan and comments about the topology. The user can also optionally insert any number of Calvin applications in the application section. When user clicks Create button in the modal, a JSON object (With the details retrieved from the forms in the modal) will be sent using Ajax call to the server (REST call using URL). The current options to create a topology will be extended according to the future implementation and needs.

Figure 4.4: Modal to create graph.
## CHAPTER 4. IMPLEMENTATION

<table>
<thead>
<tr>
<th>URL</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/iotX/iotxgraph/get_all_iotxgraphs_details</td>
<td>GET</td>
<td>Views the list of graphs stored in MongoDB.</td>
</tr>
<tr>
<td>/iotX/iotxgraph/post_user_defined_iotxgraph_elements</td>
<td>POST</td>
<td>Creates a new topology with the given specification from the user.</td>
</tr>
<tr>
<td>/iotX/iotxgraph/post_delete_iotxgraph</td>
<td>DELETE</td>
<td>Deletes a list of selected topology from the MongoDB and Neo4j.</td>
</tr>
</tbody>
</table>

Table 4.1: The RESTful API details for welcome page

As shown in Figure 4.3, the list of topologies stored in the back-end database will be shown in a table. This table will be populated as soon as the page has been loaded or when a new topology is created. User can click on any of the rows in the table to view the topology. This click action will load a new page (topology.html) by sending GUID of the topology as a query string along with the HTTP address. Query strings are additional information that are appended to the HTTP address after '?'.

User can delete one or multiple graphs that are stored in the back-end (both MongoDB and Neo4j) by selecting the check box. Table 4.1 shows the RESTful API details for welcome page.

### 4.2.2 Topology Page

The page needs a query string for GUID of the topology along with the URL to get the details of the topology in the back-end. The page also needs event-time as one more query string to get JSON details, this will be explained in coming sections when event-time is explained in detail. Using these query strings, an Ajax call will be made to get the details. On success of this call,
A JSON object of following details is sent from the back-end.

- Position, list of actors added, number of sensors added, number of actuators added and GUID for each node will be sent. All other details about the node is not added to this JSON, as the JSON size would grow bigger and front-end would need a lot of memory to store these details.

- All the links between the nodes are also added with this JSON which has GUID of the link, GUID of the source node and GUID of the destination node.

- List of applications added to this topology along with the actor names that are running on this particular topology.

The node details and link details will be loaded into Cytoscape JS. The figure 4.5 shows an example of an iotX topology that is shown in the web page after loading on to Cytoscape JS. The label on each node between two '/ refers to number of sensors added to the IoTXNode, number of actuators added to the IoTXNode and number of actors added to this IoTXNode respectively. From now on, this topology will be called as an event-graph in the following sections of the report. The reason for calling it as an event-graph will be explained in the coming sections.

The listeners to the event-graph is added according to the mode on which user is. The two modes are view mode and edit mode. As shown in the figure 4.6 user can navigate to these two modes and listeners for event-graph will change according to the mode selected.
View Mode

On view mode user cannot change the position of the nodes. The user can only perform two actions i.e. click and right click or two finger tap.

Figure 4.5: An example of iotX topology after loading on to Cytoscape JS.

Figure 4.6: Navigation bar to select the mode. According to the mode listeners to the event-graph will change.
Cytoscape JS allows compound node where a parent node can have any number of child nodes \[12\]. So, the clicked node will be the parent node and actors binded to that node will be the child nodes (list of actors binded to a node are retrieved while getting the details about the topology initially). Using this information a graph will be shown on a separate panel. The left figure of 4.7 shows the actors that are bound to a selected node (rectangle box). The right figure of 4.7 shows the actors under math application that are bound to selected node. User can filter the actors with application name using select option in the panel heading. The label on the actor represents the application name followed by a ‘.’ and the actor name. The user can also click on the parent node of actor panel (rectangle box) to know the iotX node that is been selected. An animation sequence from Cytoscape JS is added to highlight the parent node in event-graph to know which node is selected.

A qTip is added to show the information about iot nodes in the event-graph on right click or two finger tap. When user right clicks or two
finger tap action, a Ajax call will be made with keeping node GUID (selected node) and topology GUID as query strings. The back-end will send information by querying the node GUID into the topology using the topology GUID in database. The information will be viewed on qTip as shown in the figure 4.8. Sensor and Actuator section inside qTip is a drop-down list which displays sensing type and actuating type. On clicking the list details of sensor and actuator will be shown on a separate panel.

**Edit Mode**

In the edit mode, user can perform all the actions that was able to perform in view mode. Along with these actions, user can perform the following actions.

- User can drag the nodes to change its position. When user moves the node, user can view the adding and removing of links from the nodes.

Figure 4.8: An example of displaying a qTip with information about iotX node on right click or two finger tap action.
Links are calculated using the communication range between the nodes. When user moves a node a listener will be fired to get Euclidean distance of this node with the other nodes in the topology. This distance will be compared with the communication range and links will be added or removed accordingly. All these changes will be stored in a JSON file to send the updates to the back-end. The left figure of 4.9 refers the actual topology that was stored in the back-end. The right figure shows the topology after user changing the positions of the node. These changes will be stored in the back-end.

• Initially when an application is added to a topology, the actors of that application will be bound to a node randomly. So, in the edit mode user can migrate the actors. On the edit mode, a migrate actor button would appear. A modal will appear when the migrate actor button
is clicked. User can select a node from event-graph and the selected node will appear in the actor panel with the actors already bound to it. User can click on the migrate actor button and migrate the actors to this selected node. Once the actors are migrated, the changes will be updated in both event-graph and actor panel graph. All these changes will also be sent to back-end.

When user shifts from edit mode to view mode, all the changes that were made will be sent in the JSON format and stored in Neo4j as a separate event. For example, initially user will be by default in event-time zero. After editing, when the user shifts back to view mode the event-time will move to one from zero. And, all the changes will be stored in event-time one. So, user can again navigate to any event-time and view the changes. All these changes will not be stored in MongoDB. As event-time and graph mapping can be efficiently implemented using Neo4j, the event-time changes are only stored in Neo4j. Figure 4.10 shows the buttons to navigate to different event times. User can navigate to different event-times only in the view mode.

Application Graph

A separate table is added in the topology page to view the details about Calvin applications that are added to the topology. The application table is shown in figure 4.11 where the topology has two applications named WindTurbine and
• Application Name - Specifies the name of the Calvin application.

• Status - Specifies whether the Calvin application is started or stopped on a docker. The red led light is displayed when the application is not running in the dockers. Yellow led light is displayed when the dockers are created for a Calvin application. Green led light is displayed when the application is running in the dockers.

• View - This view button is used to view the actor-node mapping into the IoTXGraph. A GET method request is sent to the Neo4j with event-time to get the actor binding details of the particular application that has clicked to view. On receiving the details, the details will be loaded into Cytoscape JS. Once the details are loaded, a panel will open automatically with the application graph. Even here, the compound node

<table>
<thead>
<tr>
<th>Applications</th>
<th>Add Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Name</td>
<td>Status</td>
</tr>
<tr>
<td>WindTurbine</td>
<td>red</td>
</tr>
<tr>
<td>math</td>
<td>red</td>
</tr>
</tbody>
</table>

Figure 4.11: The application table in topology page to the details of Calvin applications
CHAPTER 4. IMPLEMENTATION

The concept of Cytoscape JS is used. The IoTXNodes are parent nodes and the actors bound to the nodes would be the child node. The figure 4.13 shows the IoTXGraph and its corresponding mapping of actors to the runtimes created on selected nodes for figure 4.12.

- **Start** – When the start button is pressed, the docker creation will be fired. The docker will be created for each IoTXNode that has actors for that particular application. The application will run until the user stops the application.

- **Stop** – When the stop button is clicked, the application that is running in the dockers will be stopped. But the docker will for that particular node will not be destroyed.

- **Delete** – The application, actors bound the IoTXNodes in that particular application will be deleted from MongoDB, Neo4j and front-end when

![Calvin Application Diagram]

Figure 4.12: An example Calvin application which predicts the probability of failure of a wind turbine using linear regression.
delete button is clicked. The dockers will also be destroyed for the nodes that do not contain actors from other applications.

User can add application to the IoTXGraph any time by clicking the add application button (Placed in the application-table panel). When user creates an application, the list of actors and application name will be stored in MongoDB and Neo4j. Actors will be bound a IoTXNode randomly. The table 4.2 shows all the Ajax call with the URL and the HTTP method used for calling the web handler in the back-end. The Calvin framework have implemented their own web application to debug the Calvin applications. The Calvin we application (csweb) is also integrated with iotX web application. The Calvin (csweb) in the Calvin tab can be clicked in the navigation bar to direct to Calvin csweb.

The UI also has slider and a radio button for simulation option at the top. These buttons are disabled as of now. The initial aim of iotX project is to
build a simulation framework. Once the project is stable with actor bindings, application creation in Calvin framework and user customizable iotX graphs, we would start developing this framework to support simulation environment also.
### Table 4.2: The RESTful API details for topology page

<table>
<thead>
<tr>
<th>URL</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/iotX/iotxgraph/get_emulation_iotxgraph_elements</td>
<td>GET</td>
<td>Gets the node details, edge details and List of applications added to this topology along with the actor names from the back-end. On which node details and edge details are loaded into Cytoscape JS.</td>
</tr>
<tr>
<td>/iotX/iotxnode/get_iotxnode_elements</td>
<td>GET</td>
<td>Retrieves the details about an IoTXNode.</td>
</tr>
<tr>
<td>/iotX/iotxnode/get_sensor_details</td>
<td>GET</td>
<td>Retrieves the details about a sensor.</td>
</tr>
<tr>
<td>/iotX/iotxnode/get_actuator_details</td>
<td>GET</td>
<td>Retrieves the details about an actuator.</td>
</tr>
<tr>
<td>/iotX/iotxgraph/post_emulation_iotxgraph_edited_details</td>
<td>PUT</td>
<td>Updates positional changes of the nodes, added edges and removed edges because of the positional changes and actor migrations in the nodes to Neo4j.</td>
</tr>
<tr>
<td>/iotX/iotxgraph/get_application_details</td>
<td>GET</td>
<td>Gets the actors and node binding for viewing the application graph in Cytoscape JS.</td>
</tr>
<tr>
<td>/iotX/iotxgraph/post_run_application</td>
<td>POST</td>
<td>Sends a JSON with the details of application name and actor bindings to start the application in the dockers.</td>
</tr>
<tr>
<td>/iotX/iotxgraph/post_stop_application</td>
<td>POST</td>
<td>Sends a JSON with the details of application name and actor bindings to stop the application in the dockers.</td>
</tr>
<tr>
<td>/iotX/iotxgraph/delete_application</td>
<td>DELETE</td>
<td>Deletes the application information from MongoDB and Neo4j</td>
</tr>
</tbody>
</table>
Chapter 5

Conclusion

IotX project is still under development phase where we are trying to minimize the Cloud dependency and increase the in-network processing using Calvin framework. The iotX web application is aiming to provide an user-friendly application creations and user customizable IoT network creations. The current back-end and front-end provides basic functionalities iotX with which the new features can be added according to the future needs.

5.1 Future Work

- The current work does not hold state information of physical device and the IoTXNode objects. When physical devices fail, the current work does not have any mechanism to handle this scenario. So, we should work on fault-tolerance among the physical devices.

- Extending the current customization techniques on binding sensors, actuators and actors to a IoTXNode.
• Learning more about Neo4j’s support for graph theory algorithms to do path computations on randomly binding actors to an IoTXNode while creating graphs.
Bibliography


Available: https://github.com/cytoscape/cytoscape.js-qtip