Ptolemy: Integrating Cyber and Physical Space and Filtering Data for Intuitive Network Visualization

Anna Lee Barber

Advisor: Y. Richard Yang

May 5, 2016

Abstract

A major challenge facing network developers is visualizing the traffic between different nodes of the network, so that they can make the system most efficient. The purpose of this project is to create an application that will allow network administrators, specifically using the SDN resource OpenDaylight, to visualize and interact with the network data in an intuitive and accessible manner.

Ptolemy displays network topology geospatially, with links colored based on traffic levels. Using a map as a display, and combining cyberspace with a physical representation allows users from many different backgrounds to easily understand the network traffic patterns. Ptolemy provides users the ability to parse and query the network data that is displayed based on both map functionality and custom input, enabling users to isolate and dive deeper into specific parts of the network. This parsing and filtering allows users to more easily view and develop with large and complex networks.

1 Introduction

Software-Defined Networking (SDN) allows network administrators to provide a centralized, simple programming model over a complex set of networks. One notable SDN resource is OpenDaylight (ODL)[1], an open source platform for building programmable, software-defined networks. This project contains a framework and platform intended to accelerate adoption of SDN systems. A major challenge for network developers is visualizing the traffic between different nodes of the network, so that they can make the system most efficient. My goal is to create a system that will allow network administrators, specifically using OpenDaylight, to visualize and interact with the network data in an intuitive and accessible manner. While there are applications that attempt to visualize network data, I found in my background research that they 1) do not show traffic data in an intuitively spatial and visual manner and 2) do not allow for parsing and querying of the network data within the visual framework, making complex networks difficult to show in a comprehensive manner. I want to develop a system that allows the user to display the network topology and data, and interact with the system in order to better understand the data.
2 Background

2.1 Software-Defined Networking And OpenDaylight

Software-Defined Networking aims to make networks agile and flexible, giving network administrators the ability to respond quickly to changing requirements. OpenDaylight is an open-source SDN project that supports several SDN interfaces and protocols, most notably OpenFlow. The OpenDaylight Controller exposes open northbound APIs, which are then used by applications to collect information about the network.[2]

OpenDaylight supports NETCONF network configuration protocol and stores data using the YANG data model. Many OpenDaylight applications use RESTCONF, a REST-like protocol running over HTTP for accessing YANG model data defined by NETCONF datastores. The data accessed is returned in either XML or JSON format.[3]

2.2 Previous Work in Network Data Visualization

There are several projects that attempt to show network data in a visual manner. The four most relevant examples are discussed here.

OpenDaylight DLUX[5] is a Javascript-based user interface that communicates with the backend to provide interactions with the OpenDaylight controller. DLUX displays network topology in an empty space, with host and switch nodes connected by simple lines. Interaction with the topology is limited to dragging the host and switch icons and zoom in/out, allowing the user to clear up the visual space somewhat.

The Open Network Operating System (ONOS)[4] is an alternate SDN operating system option from OpenDaylight. ONOS includes a web GUI with a topology view for its network system. This view shows hosts, switches, and the links between them, optionally overlayed on a simple geographic map, and can show traffic on the links. Selecting a specific node or route causes other nodes and links on the map to fade in the background, while the selected node or route is highlighted. This system does not, however, allow for querying of the network, and relies on the user to input a detailed map.

The NeXT framework[6] provides a UI for network topology with further functionality over DLUX – route visualization and some basic map overlays included. NeXT overlays geographic location on its own simple maps, with traffic visualization. While the presentation allows for many display possibilities, it can still suffer from not being able to show traffic for complex networks in an intuitive way (it shows a single traffic link from a node cluster, for example, without custom filtering of links and nodes). The NeXT toolkit, while visually appealing, does not allow for easy display of subsets of the network on a map. Last, NeXT maps are built by referencing their own map picture, which relies on the user to provide more detailed maps.

The ONOS GUI and the NeXT GUI share many similarities in appearance and functionality. They both show hosts, switches, and the links between them in a simple and visually appealing way. Both, however, rely on a static map input for geographic overlays, making map updates difficult. This system also makes it harder for a user to customize the appearance of nodes and links on the map, outside of what the GUI provides. Though both provide simple zoom functionality, the zoom does little other than focusing in on one area; no filtering or other functionality is provided by zooming in or out. Finally, neither system allows the network to be queried based on user input. At most, selecting a node or link will give information or show that section more prominently on the map (on top of the rest of the network information).
BIERMAN[7] is a web application built on ODL and NeXT UI to manage BIER Networks. Its GUI provides similar map functionality to NeXT, but for a specific context. While its application and UI are not totally relevant to this project’s networks, it provides an example of using AngularJS and Node.js to build its model-view-controller framework on top of OpenDaylight.

3 Overview and Approach

3.1 Spatial Representations of Network Data

In order to visualize network topology, there are two lines of thought for display. One attempts to show the network topology in an open space, laid out with the nodes and links in a clean and organized fashion. The second gives the topology a spatial aspect, placing the virtualized network within a physical context. While there are benefits to the cleanliness of the first approach, this project follows the second.

Though it may be tempting to treat cyberspace as completely separate from physical space, in action this cannot totally be the case. Computer networks have some physical nature as well, and that physical location can help visualize the way the network operates in the real world. The benefits of this spatial approach to data visualization are twofold: first, humans have a natural understanding of geospatial objects - we navigate, plan, and function in day-to-day life by viewing and evaluating geospatial information. Thus, placing graphs and networks in an easily understood context can help with initial understanding of the network. Second, there are many aspects of computer network monitoring that require a whole-network perspective. Particularly in monitoring traffic and traffic patterns, it can be necessary to see how the network relates in space. Again, the geospatial display makes this more intuitive for users: if there is a problem on all links coming from Chicago, it can be easier to conceptualize and then take action on, rather than considering all links from a certain switch within a more nebulous topology display. Finally, geospatial displays of networks have many wide-reaching uses; this project considers the application for software-defined networking and computers, but geospatial representation offers a flexibility within an intuitive display.

3.2 Cyberspace with Usable Physical Locations

Network topology can be represented as a graph with edges and vertices. For some computer networks, that graph might already have an interesting and laid out physical space: hosts and switches with different actual geolocations, over a reasonably large land area, so that it could be easily placed on a physical map. In this case, the proceedings for spatial display of the network are straightforward.

3.3 Creating a Simulated Physical Representation of Cyberspace

In instances where the network does not have reasonably placed geolocations (an entire network with locations in a data center, for example), the challenge for representing the cyberspace in a physical space is creating an intuitive and interesting physical representation of the network graph.
3.3.1 From Graph to Physical Space

The value of physical space is that it is intuitive for users. In everyday life, people interact with physical space: they get more detail by moving closer, see a broader view by moving away, and choose what they see by moving through the area. In representing the network in a physical space, the aim is to allow users to feel the same intuitive functionality that would be present with other objects in physical space. Thus, moving closer should provide more detail, moving away should give a wider picture, and there should be some ability to explore on any level.

Creating a physical representation of the graph would ideally show physical levels that would allow the user to traverse through the graph in a physical space – moving closer shows more vertices and edges in detail, moving away shows a wider view, and some ability to move around the entire area. Therefore, in order to represent the graph in physical space, I propose the following method of display:

The first job is to find the highest level vertices of the graph. Once this is found, the graph can be parsed into levels, with distance from the top determining which level the vertex is placed in. From there, the vertices should be placed in physical space in a way such that within each level, vertices are evenly spaced from each other, and evenly distributed in distance from the parent. Thus, when moving closer to an area, the user will see more detail from that level, with child vertices becoming more visible as the user moves closer to the parent vertex, and moving further away will allow a better view of the higher level.

For example, a simple network graph with one switch and five hosts:

```
For example, a simple network graph with one switch and five hosts:

h1
s1  h2  h3
h5  h4
```

Might be drawn as:

```
For example, a simple network graph with one switch and five hosts:

h1
s1  h2  h3
h5  h4
```

3.3.2 Leveraging the Physical Space

Once the graph has a physical space representation, the physical representation must allow the user to truly view the network in an intuitive way. To leverage the physical space in this way, there are several functionalities that should be present. As discussed
previously, moving closer or further away should provide a more detailed look or better overview, respectively. In order to do this most effectively, aggregation in view makes sense. That is, on a wider view, vertices of the graph should be shown clustered together, and on a closer view those vertices should no longer be clustered (to give more detail). This can be accomplished by clustering different levels based on zoom – on a closer zoom, the next level down can also be shown in detail without aggregation, and so on.

The zoom functionality also allows the user to interact with the physical space in an intuitive way, allowing an easy way to explore the space by moving to a deeper or shallower level, much like the real world gives more detail the closer someone is to the object.

The user should also ideally be able to focus in on certain aspects of the objects in the physical space – filtering to make the view clearer. Much like interactions with objects in physical space, the network can be understood better if parts can be isolated. For the purpose of making understanding the network easier, filtering allows a focus in on a smaller part.

### 3.4 System Architecture of the Application

![System Architecture Diagram]

#### 3.5 Design Approach

This is an OpenDaylight application built with Node/Express/Angular/MySQL and presented, for an intuitive user interface, with Google Maps.

From the user’s perspective, this application displays OpenDaylight network topology on a geographic map, with links drawn to reflect network traffic. The user can interact with the map by zooming and selecting nodes and links. Zooming in and out
filters the nodes and links shown based on user presets, allowing for clear and different
map looks at different levels. A sidebar provides a toolkit for further interaction with
the map: a user can select which nodes and links to display based on custom parameters.

In designing this project, the primary use case involves a user interacting with a
visual system and manipulating data for display. Therefore, it made most sense to
create a model-view-controller system. AngularJS provides a good MVC framework; in
querying data, however, using a database made more sense for fast and reliable results.
Therefore, I chose to run Node.js with Express (for routing/middleware) and MySQL
(the database) on the model and server side, with AngularJS acting as the model and
controller to interface with both the MySQL/Node.js database and the HTML view.
This set up provides a full-stack framework for the application, with the added benefit
of an extensive existing documentation for how they interface.

On the view side, my goal was to create an intuitive map system that would allow
both data querying and simple map interaction. The map should also remain visually
appealing and understandable, even with complex networks. Google Maps provides a
rich API framework for creating maps, along with many different features for displaying
data on the map. Google Maps are also fairly intuitive and user-friendly, making this
an ideal base for the view part of this project. In addition, Google Maps provides added
functionality over a simple geographic map input by the user (like those used in NeXT
and ONOS): Google provides automatic map updates; allows for a large amount of user
customization, especially in data drawn on the map overlay; provides a wide range of
geographic map displays, to customize the look easily; gives an ability to filter the data
view based on zoom level and other map interactions.

4 Design Details

There are three main pieces to this application: the network model – a database,
populated with information about the network; the network controller – getting the
data (including querying) to parse the network data for display; the network view – a
map, to display network data. The following sections discuss the design details of each
of these three pieces.
4.1 The Network Model

This application uses a MySQL database to store network data and easily parse and query that data. The database structure is fairly simple: a table for all of the nodes in the network, and a table for all of the links.

4.1.1 Setting Up the Database

In order for this application to run, there must be a MySQL Database with a table named Nodes and a table named Links.

The table data should follow this structure:

**Node**
- Node_id (int 11)
- Node_name (varchar 100), primary key
- location_lat (float 10,3)
- location_lng (float 10,3)
- type (varchar 100)

The MySQL query to create the Nodes Table:

```sql
CREATE TABLE IF NOT EXISTS Nodes (Node_id int, Node_name VARCHAR(100), location_lat FLOAT(10,3), location_lng FLOAT(10,3), type VARCHAR(100), PRIMARY KEY (Node_name))
```

**Link**
- Link_id (int 11), primary key
- source_node (varchar 100), foreign key references Nodes(Node_name)
- dest_node (varchar 100), foreign key references Nodes(Node_name)
- intensity (float 10, 6),
- src_lat (float 10, 3),
- src_lng (float 10, 3),
- dest_lat (float 10, 3),
- dest_lng (float 10, 3)

The MySQL query to create the Links Table:

```sql
CREATE TABLE IF NOT EXISTS Links (Link_id int, source_node VARCHAR(100), dest_node VARCHAR(100), intensity FLOAT(10,6), src_lat FLOAT(10,3), src_lng FLOAT(10,3), dest_lat FLOAT(10,3), dest_lng FLOAT(10,3), PRIMARY KEY (Link_id), FOREIGN KEY (source_node) REFERENCES Nodes(Node_name), FOREIGN KEY (dest_node) REFERENCES Nodes(Node_name))
```

Note: the four latitude and longitude columns will be set only after topology information about the nodes has been read in. Placing geographic information in both the Nodes and Links tables allows both groups to be treated separately on the map.
4.1.2 Node.js + MySQL + AngularJS

We use Express with Node.js in order to allow an application to make HTTP ‘GET’, ‘POST’, or ‘DELETE’ calls to the server. Express routes different paths to associated server functions. This works with MySQL by allowing AngularJS files to make HTTP requests, which server.js receives and uses to make an associated MySQL action.

Server.js currently supports 6 routes for HTTP requests:

1. POST ‘/nodes’ – Gets the nodes from the table Nodes in GeoJSON format
2. POST ‘/links’ – Gets the links from the table Links in GeoJSON format
3. POST ‘/nodes/insert’ – Inserts new nodes into Nodes
4. POST ‘/links/insert’ – Inserts new links into Links
5. POST ‘/links/update’ – Updates the links table to include geographic location
6. DELETE ‘/all’ – Deletes all rows from both Links and Nodes

4.1.3 Populating the Database

The database is populated by getting network data from OpenDayLight, and then parsing that data. Getting data from OpenDayLight is accomplished through two REST API calls to:

http://{ip-address}:8181/restconf/operational/network-topology:network-topology/
http://{ip-address}:8181/restconf/operational/opendaylight-inventory:nodes/

The REST API returns XML/JSON dynamically generated from YANG models. The AngularJS service odlREST, defined by getrestapi.service.js, performs these REST API calls.

- loadTopology(success, error) calls /network-topology:network-topology/
- loadNetworkInventory(success, error) calls /opendaylight-inventory:nodes/

The AngularJS service parser, defined in parseTopologies.service.js does the bulk of the work in evaluating the returned data from the REST API calls. For each call, we parse the returned data in order to populate the database tables.

Parsing the Network Topology:  The first call is always to /network-topology:network-topology/. This returns the network topology, with basic information about all of the nodes and links. parser.parseTopology() creates a list of all of the nodes, getting their id, type, and geographic location (by latitude and longitude), and a list of links with id, source node, and target (destination) node.

Geographic Location of Nodes:   The YANG models do not return data on geolocation of nodes. Associating the geographic location of nodes with the appropriate node ID or IP address should be done by the network administrator, who has the geographic location of all hosts and switches in the network. Currently, parser randomly assigns a geographic location to each node when the topology is parsed, allowing for testing with virtual networks such as Mininet. Parser does find both the node ID and node IP address, however, should a network administrator wish to use either of those to lookup the geographic location.
Parsing the Node Inventory: The network topology returns all needed data about the nodes, but does not give much information about the links. Therefore, we need to use /opendaylight-inventory:nodes/ to get additional information. /opendaylight-inventory:nodes/ returns a list of the nodes and information about all of the node connectors. parser.parseNodeInventory() iterates through all nodes and collects information on each node connector: link id, source node, destination node, and intensity. For each node connector, two separate links are created: one from node a to node b, and one from node b to node a, as /opendaylight-inventory:nodes/ does not list the two-way links separately. Rather, it gives data on both bytes transmitted and received on a node connector.

Calculating Link Intensity: Link intensity is calculated by finding the link utilization, or traffic over link capacity. This is done by getting bits per second, and dividing by the rate limit (current-speed given by the YANG model).

4.1.4 Updating the Database

The AngularJS service parser returns an object with two arrays: one for nodes and one for links. The Angular controller databaseCtrl, defined by updateDatabase.controller.js, loads the network data with odlREST, parses with parser, and makes two queries to MySQL, one to insert the nodes, and one to insert the links. These MySQL insert calls are done by sending the data to server.js, which connects to the database and inserts through

1. \[ \text{INSERT IGNORE INTO Links SET ?} \]

where ? is the set of links passed in by databaseCtrl. (A similar call is made for the Nodes table.)

databaseCtrl must have an updated userConfig variable, with ctrlHost (host IP address), ctrlPort (port that OpenDayLight runs on), odlUsername (username for OpenDayLight, ‘admin’ by default), and odlPassword (password for OpenDayLight, ‘admin’ by default) set for the individual user.

4.1.5 Resetting the Database

If the user wants to load in a different network, they must first clear out the database so that they do not show old nodes and links. This is also accomplished through the databaseCtrl controller, using resetDatabases(), which calls

1. \[ \text{DELETE FROM [Table Name]} \]

to the MySQL tables, clearing all rows but keeping the table structure.

The GUI includes a button to do this, along with a button to reread network topology and add it to the database.

4.2 The Network Controller

The network controller gets data from the database and puts it in a usable format for the view. This can be done by either getting all data, or parsing the node and link data based on custom input.
4.2.1 Getting Node Data in Map Format

Nodes are loaded into map format through the function loadMarkers() in map.controller.js. This function, in order:

1. Gets nodes from node table in GeoJSON format
2. Creates markers from the GeoJSON returned

**Getting Nodes in GeoJSON format:** An optional queryString parameter in loadMarkers() is passed to server.js as the MySQL query, otherwise the default query is "SELECT * FROM Nodes", which returns all rows in Nodes.

In order to draw the nodes on the map, it is easiest to parse them in a GeoJSON format. Therefore, server.js creates a GeoJSON feature for each node row returned by the HTTP post to '/nodes'. All information about the node (all column values) are set as the "properties" object of the GeoJSON feature, for easy reference.

The following is the structure of each GeoJSON feature:

```javascript
var nodePoint = {
  type: 'Feature',
  geometry: {
    type: 'Point',
    coordinates: [
      rows[i].location_lng,
      rows[i].location_lat
    ]
  },
  properties: rows[i]
};
```

4.2.2 Styling and Filtering the Nodes

Google Maps allows some customization of markers – namely, that a custom icon can be set. By default there is no icon set, but a user could add their own easily, to differentiate hosts and switches, for example, by adding that key and value to the marker object.

When there are many nodes on the map, and especially nodes located close to one another, the map quickly becomes unreadable. To avoid this, particularly with complex networks, I use Marker Clusterer[10], a Google Maps JavaScript API v3 library that creates and manages per-zoom-level marker clusters. Clustering works by taking a user-assigned grid size (as the map display is a large grid) and minimum cluster size, and clusters markers based on these two variables.

A description of the algorithm, from the Google API: "It works by iterating though the markers in the collection that you wish to cluster and adding each one into the closest cluster if it is within in a minimum square pixel bounds." [11]

4.2.3 Getting Links in Map Format

Links are loaded into a map format through the function loadPolylines() in map.controller.js. This function, in order:

- Updates the link table to include geographic location data of nodes
- Gets link data from the database in a GeoJSON format
• Creates polylines from that link data

**Updating the Link Table:** Initially, the link table did not include columns for geographic location. In the programming of this project, however, it became clear that it was too cumbersome to try to reference the geographic location of nodes for each link at every step, especially since links are kept separately from nodes. Therefore, before creating GeoJSON to represent each link, it made most sense to add columns for latitude and longitude of the link’s nodes. Server.js updates the columns to include geographic data when HTTP post ‘/links/update’ is called. This finds the correct nodes in the node table for source and destination, and adds the correct source and destination latitude and longitude (src\_lat, src\_lng, dest\_lat, and dest\_lng) to the link table.

**Get Link Data From the Database:** Like with nodes, server.js makes a query to MySQL, gets a set of rows returned, and parses those rows into a GeoJSON format. An optional queryString parameter passed to mapCtrl.loadPolylines() is passed to server.js as the MySQL query; the default query is ‘SELECT * FROM Links’, which returns all rows.

For links, the GeoJSON geometry used is a LineString, where the starting point is (src\_lng, src\_lat) and the ending point is (dest\_lng, dest\_lat). As with nodes, all other data in the link table (all columns) are included in the properties value of the GeoJSON. The server returns an array of GeoJSON features, and the AngularJS functions can get all information about the link through the properties object.

The returned GeoJSON feature object looks like:

```javascript
var newLink = {
    type: 'Feature',
    geometry: {
        type: 'LineString',
        coordinates: [
            [rows[i].src_lng, rows[i].src_lat],
            [rows[i].dest_lng, rows[i].dest_lat]
        ]
    },
    properties: linkProperties
};
```

4.2.4 Styling the Links

Links are styled using the function styleFeature() in map.controller.js. The user sets two variables here - `maxIntensity` and `minIntensity`. These provide a range for traffic calculations, where intensity is the link utilization (as calculated from the network information and added to the database). Further, because we are showing two-way links, where each direction may have different traffic, we cannot draw the lines on top of each other. In order to present a better visual representation of the network traffic, links are styled alternatingly as a normal polyline (drawn as a straight line), or as a geodesic polyline (drawn with curvature to mimic the curve of the Earth). Because links are placed in the table immediately next to their corresponding opposite (so link A:B is followed immediately by B:A), when two links between the same two points are drawn on the map, one will always have a slight curve, separating it visually from the other.
Links can be further styled in the `styleFeature()` function, as a user can edit stroke weight and stroke opacity to provide different visual outputs for the links.

### 4.2.5 Setting Link Visibility

When a network contains many links, a map of the network topology can quickly become very cluttered. To work around this, the links can be set so that they are not visible when the map’s zoom level is under a certain number, so that zooming out shows only nodes (and node clusters). This number is set with the variable `$scope.linkConfig.zoomShow`, and should be an integer between 0 and maximum zoom level (this can be referenced through the Google Maps API; it is usually around 18). On default, all links are initially set with `visible: true`, which draws them on the map on load. The map references `$scope.linkPaths` set by `mapCtrl` in `map.controller.js`, and sets visibility based on link parameters in `linkPaths`. If a user wishes to set the map’s initial zoom level greater than `zoomShow`, and still hide the links initially, they should set `visible: false` when creating polylines in `loadPolylines()`.

### 4.2.6 Getting Link Information

Displaying link information can also get messy, due to sheer number of links placed in a network. In order to create a clean map and to see traffic better, the GUI does not create `infoWindows` for each link. Rather, there is a section on the sidebar which displays link information for the most recently clicked link.

In `index.html`, a div for displaying the list of link information is formatted as such:

```html
<div id="showLink" ng-model="clickedLinkHTML" style="color: white">(click on a link to see its information)
  <ul style="list-style-type:none">
    <li>Link ID: {{clickedLinkHTML.Link_id}}</li>
    <li>Intensity: {{clickedLinkHTML.intensity}}</li>
    <li>Source: {{clickedLinkHTML.source_node}}</li>
    <li>Destination: {{clickedLinkHTML.dest_node}}</li>
  </ul>
</div>
```

In order to show additional information about a link, the user must follow two steps: 1) ensure that the link table `Links` includes a column for that additional information, so that it will be returned as part of the GeoJSON from the server and 2) add a list item to `index.html` referencing that information.

### 4.2.7 Queries

This app allows a user to query nodes and links on the map, to show specific information and present a clearer picture of the network traffic. In the view, the user can set custom parameters that the controller then uses to return filtered data.

### 4.2.8 Creating a New Query String

The function `createQuery()`, defined in `createSubsetQuery.service.js` returns two strings: one to query the Nodes table, and one to query the Links table. The two strings are created by first parsing the custom queries. Text from the input box for nodes is added
to the node query string, and text from the input box for links is added to the link query string.

createQuery() next adds clauses to the string based on source and destination nodes selected by the checkboxes. If a node is selected in either list, the node will appear on the map. For a link to appear on the map, both its source node and destination node must be selected in the appropriate list. This is accomplished through an “and” statement in the link query, and an “or” statement in the node query.

If a source node list is passed into createQuery(), it assumes that a destination node list is also passed in (even if it is an empty list). An empty list designates that no nodes of that type have been selected.

4.2.9 Validating User Queries

The controller subsetQuery does some checking of the user input for custom queries: it makes sure that the column_in_table aspect of the query is valid by checking it against a list of valid table columns.

For example, checking for Node columns refers to

```javascript
var nodeCategories = [
    "Node_id",
    "Node_name",
    "location_lat",
    "location_lng",
    "type"
];
```

A user can customize their database to include additional columns or different columns, and validate by changing the variables 'nodeCategories' and 'linkCategories'.

If the user input does not have a correct column name, or does not include three elements that are space-delimited, subsetQuery will ignore it. Otherwise, the query string is created, and passed to MySQL. If there is an error in syntax there (trying to compare intensity (a float) to a word like “switch”, for example), MySQL will return no rows.

4.2.10 Output to Map

Once a query string is created by createQuery(), it is passed in as an optional argument to loadMarkers() (for nodes) and loadPolylines() (for links) in map.controller.js. These two functions query the MySQL database with the optional query (or default, which is to return all rows, if no query string is passed in).

Map.controller.js keeps track of a $scope variable $scope.allMarkers, which maintains a list of all markers in the database, and not just those returned by the custom MySQL query. The checkboxes for user source and destination selection reference this variable, to allow users to have a list of all possible nodes and not just those shown on the map at the current moment.

4.3 The Network View

The map is the main aspect of the GUI for this application. It displays the network topology based on geographic location of nodes, and draws the links based on link traffic. This gives the user an immediate view of the network traffic, in a coherent and easily comprehensible format.
4.3.1 Google Maps

The base map is built using Google Maps, which has an extensive Javascript documentation[8]. In order to integrate AngularJS with Google Maps, we use the angular-google-maps directive[9].

Basic map properties are set in map.controller.js, using the function loadMap(). In this function, the user should set map initial zoom level, map center, and the map type (road, satellite, terrain).

In order to use Google Maps, index.html includes

```html
<script src="https://maps.googleapis.com/maps/api/js?key=AIzaSyBJNG9PUlMfEJJLRjI-PMEth8FRgaE5Pju"></script>
```

The string following 'key=' should be replaced with the user’s Google Maps API key (see Google’s documentation).

4.3.2 Node Displays

Nodes are displayed on the map through Google Maps markers. Each marker also has an associated infoWindow, which will show information about the selected node on click.

4.3.3 Link Displays

Links are displayed on the map separately from the markers. Though this means having to do some additional work to connect links and their source and destination nodes, it allows flexibility in displaying and querying. Links are displayed on the map as Google Maps polylines, based on GeoJSON coordinate data. They are styled to show the traffic level on each link. Further, in order to help clear up the view of the map (particularly for large networks, which may have hundreds of nodes and thousands of links), the map can be set to only show links at a certain zoom level. If the zoom is less than the set level, the links will not be visible on the map.

4.3.4 Angular-Google-Maps Multiple Polylines Issue

While angular-google-maps includes a directive for multiple polylines, ui-gmap-polylines, there is a known issue in which polylines cannot be styled dynamically using this directive. Instead, I have used the directive for a single polyline, ui-gmap-polyline, with ng-repeat, which accomplishes the same goal of placing all polylines on the map, in only slightly less elegant code. This enabled me to hide and show links dynamically.

4.3.5 UI for Queries

The application has two main options for users to filter the map: 1) selecting source and destination nodes to show and 2) custom queries on aspects of nodes or links.

Source and Destination Filtering In the UI, the user is presented with two lists of checkboxes: one for source nodes, and one for destination nodes. All boxes are checked initially. When the “Go!” button is clicked, a list of selected source nodes and selected destination nodes is made to help create the new query.
**Custom Queries**  In the UI, the user is also presented with two input text boxes: one for queries for the node table, and one for queries for the link table. At this time, the input queries must adhere to the following format:

\[
\text{column} \_\text{in} \_\text{table comparator value}
\]

where the three aspects are space-delimited. (An example query for the Links table: \textit{intensity} > 0.3). Both the Nodes table and the Links table can be queried via these input boxes.

### 5 Preliminary Evaluation

See the end of this paper for screenshots of the application in action.

In use, the application allows networks of many sizes to be displayed via the physical map. Because it is built on OpenDaylight, and builds a database from JSON gathered by OpenDaylight, networks that are supported in OpenDaylight are easily added to the database. Parsing and filtering functionality helps to clear up the view of the map, though this can take some work from the user when maps are particularly complex - the initial view showing all links and nodes is sometimes unhelpful because of crowding in space. The actual parsing functionality works well to find specific subsets of the data, based on node and link attributes, however.

Load testing of the web application was performed with the Node.js module loadtest. The tests were run for varying times at varying levels of Requests Per Second (RPS), and percentage of requests completed served within a certain time were measured.

Here tree network refers to a tree network of depth 5 and fanout of 2 (63 nodes and 124 links). Linear network refers to a linear network with 7 hosts and 7 switches (14 nodes and 26 links).

These network data files can be found in the code base under the samples folder (one set is named tree\_depth5\_fanout2, the other is named linear\_7).

Average results are recorded in Table 1:
<table>
<thead>
<tr>
<th>Description</th>
<th>Duration</th>
<th>RPS</th>
<th>50% of Requests</th>
<th>90% of Requests</th>
<th>99% of Requests</th>
<th>Longest Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree network from sample data file</td>
<td>20s</td>
<td>500</td>
<td>4ms</td>
<td>121ms</td>
<td>216ms</td>
<td>314ms</td>
</tr>
<tr>
<td>Tree network from sample data file</td>
<td>50s</td>
<td>500</td>
<td>3ms</td>
<td>240ms</td>
<td>8361ms</td>
<td>8583ms</td>
</tr>
<tr>
<td>Tree network, keep alive requests</td>
<td>20s</td>
<td>2000</td>
<td>3ms</td>
<td>6ms</td>
<td>16ms</td>
<td>70ms</td>
</tr>
<tr>
<td>Linear network from sample data file, keep alive requests</td>
<td>20s</td>
<td>2400</td>
<td>3ms</td>
<td>5ms</td>
<td>10ms</td>
<td>35ms</td>
</tr>
<tr>
<td>Linear network on load from OpenDaylight, keep alive requests</td>
<td>20s</td>
<td>1800</td>
<td>4ms</td>
<td>7ms</td>
<td>26ms</td>
<td>258ms</td>
</tr>
</tbody>
</table>

Table 1: Results of Load Tests

Keeping alive the requests so that new connections do not have to be created gives a very reasonable run time. Creating the graph by loading data from OpenDaylight adds a slight amount of time to the latency, as compared to loading data from a local file, but not an unreasonable amount.

When the keep-alive option was not set, new connections had to be created every time. On 20 seconds of duration, this slowed down the longest request time, but still ran reasonably well. On 50 seconds of run time, however, the longest request time was around 8500 milliseconds – a very long time for a request, due to errors based on overloading the application. When not having to recreate connections, however, both the more complex tree network and simpler linear network were able to handle higher RPS and had better latency with no errors.

Doing a total reload and redrawing the graph based on network data took on average about 80ms for the tree network.

6 Future Work

The major point of future work for this project involves specific routes of flow along the network. Ideally, the map would be able to get routes from the network, and display them over traffic (much like Google Maps might display a car route over the traffic of the roads). This would enable developers to look very specifically at the traffic on a specific route. The other major improvement for the application would be to dynamically update just the traffic of each link, without having to reload the entire network topology, in order to enhance performance. Further future work also includes providing a system for geographically locating specific nodes.

In the query system, some future enhancements involve making the possible querying more robust - allowing complex queries with multiple parts, for example. The query system could also provide better validation of user inputs - a functionality which might
require a more detailed list of allowable inputs. If the system supports flows, the query system should also be expanded to optionally include flow overlays for the user.

7 Conclusion

By displaying network topology and traffic using Google Maps, we create a GUI that allows users to understand the network and network traffic in an intuitive visual manner. Filtering and map functionality provide a both a clean user interface and an ability to see complex networks from multiple levels. Further, the project allows the user to query map data in a flexible manner. This both adds usefulness in real time, as a user attempts to parse out what is happening in the network, in both broad overviews and very focused areas, and provides a framework for additional applications to parse and query map data.

References


Figure 1: A sample map with many nodes clustered

Figure 2: The same map on zoom, where now links are shown
Figure 3: Results of a user-input query

Figure 4: A simpler network map