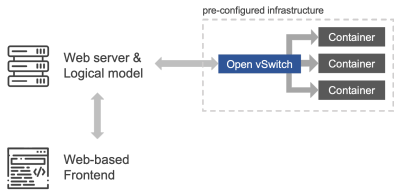
Network Visualisation for the Routing Project

Lina Gehri, Áedán Christie and Marco Di Nardo

As a part of the Communication Networks course, students configure their own mini Internet and have to solve challenges regarding spanning trees, OSPF, BGP and more. While solving these challenges, many problems may arise due to misconfiguration. Due to configuration tools on network devices providing only a limited view of the network, locating and debugging large scale problems can be extremely hard. Visualising the network can help to address this issue, as visualisations allow both to gain a better understanding of the network as a whole, and to find the root of complex problems involving multiple devices. However, creating a visualisation system poses its own problems such as balancing an up-to-date view of the network, while not overloading the network infrastructure by collecting data.
In this group project, we created a visualisation system for the Communication Networks mini Internet. Using smart caching and asynchronous communication between front- and backend, we can efficiently visualise a broad range of network aspects, from BGP advertisement propagation to spanning tree link states. Furthermore, our set up offers a React-based web interface to interactively explore the visualisations and in the future, can easily allow fine-grained access control.

Overview

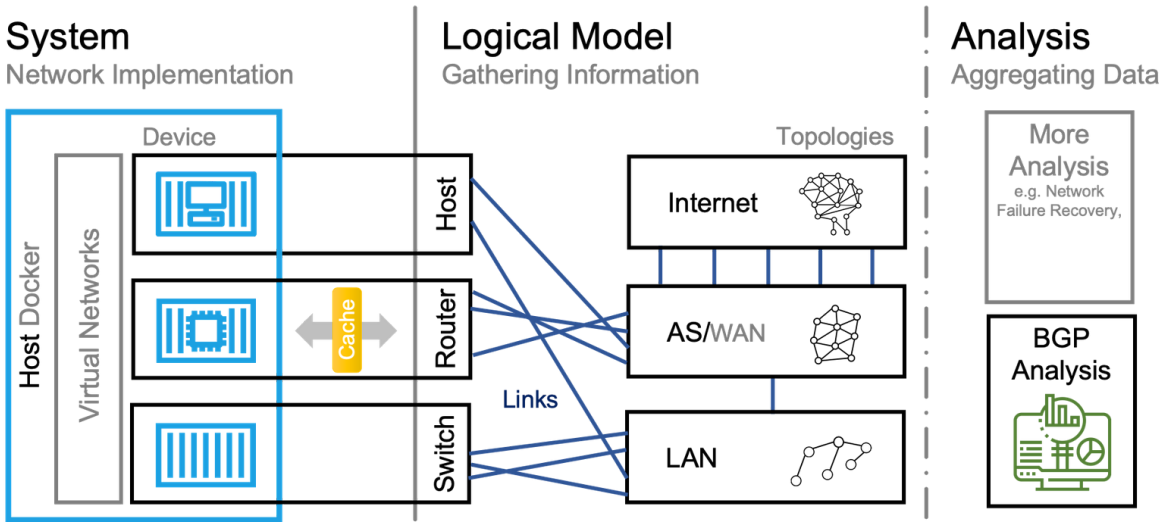


Project overview

The project is essentially split up into two parts, a frontend and backend. The frontend is a single-page web application realised in *React.js* and visualises information retrieved from the backend using RPC calls. The backend is responsible for gathering information from the mini-internet, analysing it and providing the *WebSocket* interface for the frontend.

The backend application runs inside a containerised setup which was built using *Docker* similar to how the docker containers bootstrapped by the *mini-internet* generator work.

Backend



Backend Architecture

Basic architecture

The backend abstracts the existing infrastructure into a logical models with representations of the routers, switches, hosts and links present. We group those devices into topologies to represent the different levels of the *Communication Networks* model in an intuitive way.

Each device is wrapped by a class with corresponding methods. Furthermore, we implemented classes for network links in order to provide a natural interface for gathering link information. We used an indirection we called *ports* such that

devices and links need not be inherently aware of one another and need not know specific implementation details about one another.

The `Topology` class and its descendants are used to represent the hierarchical structure of the *Mini-Internet*. Each topology contains a *NetworkX* graph as a backbone for its data model. *NetworkX* is a *Python* package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. The graph's nodes correspond to the docker devices in the Mini-Internet (Routers, Hosts, Switches) and the graph's links hold pointers to our custom `Link` objects. All of the functionality we provide to work with and manipulate links is executed on these link objects not on the *NetworkX* edge.

Data collection

The most difficult part of the backend was to find a way to retrieve information from the virtualised networking devices in an efficient and non-blocking way. We achieved this by introducing the Python classes `Device` and `DeviceManager` to abstract the connection from the actual devices. Thus, we were able to separate our data model from the specifics of how the devices work and how data is collected from them.

`Switches`, `Routers` and `Hosts` all inherit from a common `Node` class and each hold a reference to their `Device` and can call e.g `(self.device).execute_command()` to use the interface offered by `Device`. `Nodes` are, however, unaware of the specifics of how commands are executed. The `DeviceManager` can add new devices to the network and keeps track of all the devices in the *Docker* namespace. We used the *Docker SDK* for *Python* in order to execute commands via an HTTPS request through docker's internal Linux socket (located in `/var/run/docker.sock`) and return the output. The container needs elevated privileges within the *Docker* system for this to work properly.

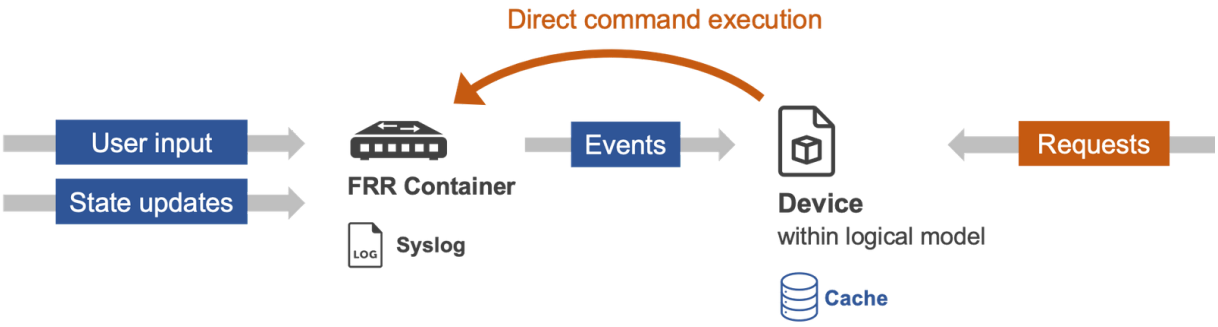
Alternatively, one could connect up the virtual networking devices using e.g. SSH, SNMP or syslog and collect the data that way. A protocol like SSH is would be more versatile and would provide more in-depth access to configuration and runtime information about network devices. Unfortunately, using a direct SSH access seemed rather time-consuming given it blocks the code execution during the call and has to set up a new socket for each command. Using Docker SDK was the most straight forward method which we then used for the rest of our project.

Web server

A web server based on the *Tornado* python framework responds to requests from the frontend and manages the cache connections to the virtual containers. The server manages the *WebSocket* connections and relays RPC requests that come in through the socket from the frontend to the correct object in our virtual model and serves up the response to the client.

In order to profit from *Tornado's* asynchronicity and to allow for many frontend requests in parallel we also made our backend functions asynchronous and used *asgiref's* `sync_to_async` decorator for functions that are inherently blocking. Consequently, the slow and blocking I/O calls to the virtualised networking devices don't block the entire application.

Caching



Cache architecture

In order to reduce load on the containers and especially on the *Open vSwitch* switching fabric, which is implemented using *Linux* sockets and namespaces, we implemented a simple but effective caching mechanism on our interfacing class `Device`. We focused on the virtualised routers as they receive the most traffic and contain most of the information we were interested in. However following a similar scheme, one could implement caches for virtual hosts and switches.

The data is cached in a python dictionary that maps the commands to their last known output. When a cache miss occurs, the command is executed on the device and added to the list of cached commands which resides inside the virtual device. As for cache update strategies, polling is commonly only opted for as a last resort, as it introduces unnecessary traffic and latency for updates. A much better alternative is to use an event-based system with which the event source is able to directly send updates. We chose an architecture based on *syslog* as this is widespread and the virtualised routing software already provided support for it.

More specifically, we used *rsyslog* with its *omprog* module which allowed us to "integrate arbitrary external programs into *rsyslog's* logging" (https://www.rsyslog.com/doc/v8-stable/configuration/modules/omprog.html?highlight=omprog). In our case o*mprog* and a set of custom filter rules trigger a python script that executes the list of registered commands and forwards all the output to our visualisation container. The cache is then updated within the corresponding `device`'s caching dictionary.

Furthermore, some defaults did not provide sufficient logging levels, e.g. OSPF weight changes were not logged by default. Hence, we automatically configure the `log commands` option when setting up the cache which leads to a more conclusive log of students' configuration changes. This feature requires more testing in the future as it is possible that certain state changes or BGP updates are still not being logged appropriately. Unfortunately, *FRRouting* does not document this feature in much detail which makes debugging rather difficult.

Network analysis

Aside from our virtual model, which describes the topology of our network, different types of analysis may be performed on it. In order to provide future users and developers with an example as to how this may be achieved we implemented a class `BGP_Analysis` that provides a testing framework for BGP business relationships which are one of the central topics of Laurent Vanbever's *Communication Networks* lecture. The framework provides two methods `customer_link_test` and `peer_link_test` to test for customer-provider and peer-peer BGP relationships based on rules discussed in the aforementioned lecture.

Problems and difficulties

IXP docker container issues

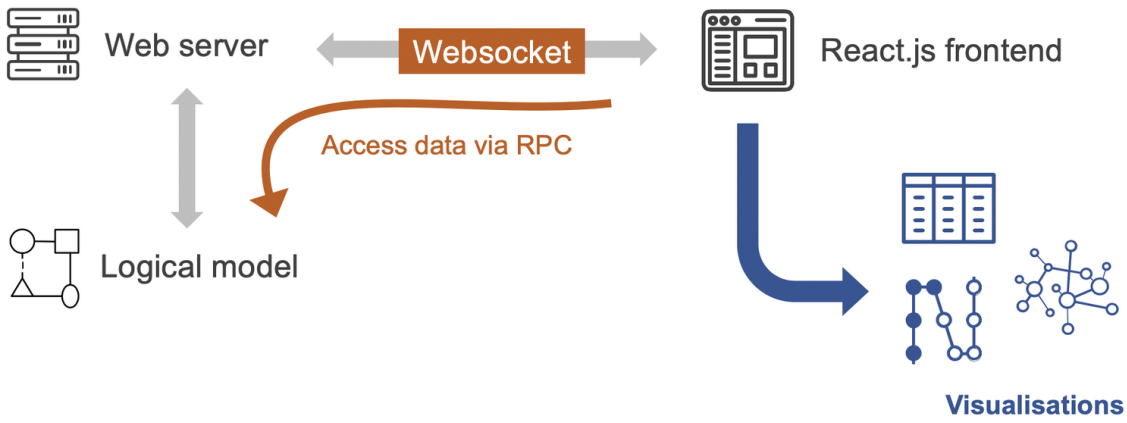
Because of some bugs in the *mini-internet* we had a lot of trouble with every function that involved `IXPRouter` objects as their `vtysh` did not work properly anymore after the upgrade from FRR 6.0.2 to FRR 7.1. Given those constraints, we put workarounds in place to guarantee functionality with IXP router containers. Such workarounds might need to be added or adapted for any future improvements and additional functionality in order for the IXP routers to correctly function within our project's framework.

Ongoing changes to the Mini-Internet

Due to requests for changes from our side and ongoing development of the Mini-Internet project a lot of changes to the environment were made during the time

we built our framework on top of the infrastructure. As a result we had to rewrite functions or even whole classes and wait to have improvements implemented, so we could also program new functions.

Frontend



The frontend consists of a single-page *React.js* application which aims to show visualisations and provide ready access to information from the logical model as well as traversing it. *React.js* has many advantages relevant for complex frontend applications, such as centralised state management and predictable lifecycle methods for components. The application features two sections: a visualisation section and a control section. Using common user interactions the logical model may be navigated through, dynamically updating the visualisation section's view with more information from the backend.

Visualisation section

This main section - the main column within the application - hosts a visual representation of the current view, rendered using *Cytoscape.js*. Currently, only a single level of the logical model may be viewed at any given moment. The visualisation is be programmed to reflect the visual layout that student know from Laurent Vanbever's *Communication Networks* lecture. Even though the *Cytoscape.js* library provides methods to traverse and manipulate the graph - much like *NetworkX* on our backend does - on our frontend we purposefully used only functionality relevant to displaying or animating information on screen.

Control section

This section - the sidebar column within the application - provides a space in which all user controlled actions as well as further information is placed. When nodes and edges are clicked in the visualisation section this view updates automatically to show relevant actions and information. At the time of writing there are three different types of visualisation that we provide the building blocks for: graph animations, tabular data, lists.

Graph animations

This type of visualisation animates parts of the graph within the visualisation section in order to highlight propagations or paths within the graph. We provide two examples for this type: virtual traceroutes and IP prefix propagation. The virtual traceroute uses routing information to infer the effective route taken across the L3 topology. This is then animated on the frontend by visually highlighting nodes and edges sequentially. The IP prefix propagation is launched on the top-level view and shows how a given IP prefix propagates the autonomous systems again visualised by means of staggered animation.

Tabular data

When a lot of data is to be displayed, a table is often the most appropriate visual structure to display that data. We implemented a full screen overlay on top of the visualisation section which opens when needed and displays a large tabular view. An example of where we implemented this is the IP prefix propagation on an edge between two autonomous systems. An action is launched on that edge which then displays the overlay with the set of IP prefixes which are carried on that edge for each direction.

Lists

This type mainly aims at displaying data dynamically and at providing ready access to information about nodes and edges of the mental model. We used this type to show common information such as interfaces, link properties and states.

Improvements and extensions

Selective Cache Update
Whenever something is logged all cached commands are executed also the ones which do not have to do anything with the change. One could do it in a more fine grained way digging into “what exactly is logged at what time and which information depends on it”.

Authentication
It is possible to implement token-based authentication. The best place to do so is in the `frontend_runner` where the function in the backend is called. One could have a list for each token with which functions a person with this token is allowed to execute in the backend.

Bidirectional Updates
Right now, when something in the cache changes the frontend is not notified of this change. One could automatically update the visualisation in the frontend without having to refresh the page.

Pandas
We use regex to parse all configuration files and output. To use pandas would be a bit more readable than regex.

Cache for Switches
Right now the cache is only implemented for routers. The analysis of the L2 network would profit from a cache for switches.