1 Introduction

This document describes in greater detail some of the features of the proposed Pivot utility. It is a summary of our findings from preliminary work done on our software project as of commit `a73842192` in our upstream repo.

2 Data Collection Process

To display useful information about the network, we need to collect the data first.

2.1 OSPF Data Collection

To obtain OSPF data (network topology information from which the network graph is constructed), we will use either:

- BIRD OSPF topology dumps via BIRD CLI,
- Quagga OSPF topology dumps via Quagga management socket (TODO),
- (possibly other routing daemons' interfaces).

We assume that a dedicated instance of either routing daemon will be deployed which will only be used to collect data from and won't partake in the routing process in any way. This setup is preferable, since it stays out of the critical path.

The choice of deployed routing daemon is up to the user. It's arguably simpler and to deploy another instance of daemon x in a network which already uses x for routing information exchange, rather then forcing users to deploy $y \neq x$ solely for the purposes of monitoring.

The dedicated routing instance will be the single source of topology information.

2.2 Traffic Data Collection

To obtain information about the amount of traffic flowing through the network, we will use SNMP daemons on the routers. Basically all dedicated router hardware implements SNMP MIB-II data standard in their SNMP information base (as described in RFC 1213), and thus provides all necessary SNMP metrics, such as ingress or egress traffic flow on specific L2 interface. On Linux routers, `snmpd` can be used to provide the data, as it is also SNMP MIB-II compliant.

The more problematic part of the traffic data collection will be to correctly associate L3 interfaces gained from OSPF-collected data with the corresponding L2 interfaces. From the OSPF (L3) perspective, interfaces are identified by IP address assigned on some L2 interface of the router. The L2 interface can be either physical (Ethernet, wireless, etc.) or virtual (VLAN, bridge, bonding, etc.), but in both cases the same SNMP metrics should be available for data collection. The key problem is to find corresponding L2 interface for the given IP address, because this association is not directly recorded either in SNMP MIB-II compliant information base. Possible solution is to gather this information from ARP table of the given router, as ARP table is also exposed through SNMP MIB-II. The L2 interface can be then found by its MAC address in the SNMP MIB. So the traffic data collection is possible, but the SNMP queries will not be that straightforward and probably some level of caching will be necessary.

There will be a single data collection node for SNMP-enabled devices which will scrape all the devices. This will be our single source of traffic flow information.

3 Data Storage: Graph/Event Database

After researching available choices (mostly general-purpose databases and other custom data storage formats), we have decided to design a simple but powerful data storage format called the Graph/Event Database (GEDB). We're too early into the design process, but the main features and design goals include:

- built around the assumption that we're storing graph data with lots of additional labels attached to the nodes and edges,
- labels can be anything from arbitrary textual content to values of derived metrics, such as TX flow on an edge in bits per second,
- encoded in CBOR,
- events (such as the addition of a node/edge to a graph or a value of a label at the given time) will be serialized as sequential CBOR stream into a file,
- the file will probably contain snapshots, i.e. dumps of complete state of the graph as of some time for efficiency.

The simple database system built around this file format will provide the following features:

- retrieve the entire state of the graph at the given point in time,
- retrieve a network graph which is an accurate representation of a time interval (the "average graph"),
- retrieve next and previous event in the event stream.

We have identified the following problems we will need to solve:

- data rotation scheme has to be established, so that old data can be gradually phased-out automatically without affecting the consistency of the database,
- ideally we should be able to add historical data to the file, possibly at the expense of being somewhat slower in the process.

4 Data Export

The program has the following outputs:

- interactive web-based user interface described below,
- network graph exported to SVG, PNG, PDF and possibly other Graphviz-provided terminals,
- metric collection end-point for Prometheus (`/metrics`) which can be used for additional visualisation (Grafana) and alerting (Prometheus alerts which can monitored by standard tools such as Icinga).

5 User Interface

The user interface (UI) is the main selling point of our software. It's also the pain-point of most existing tools with similar goals. Therefore, significant part of resources will be spent here.

The UI is the primary consumer of data stored in the Graph Event Database (GEDB) files. GEDB itself was designed to provide the features which the UI needs to be useful.

5.1 Overall Description

The UI should communicate the following information to the user:

- Network Graph: how routers and networks are connected. The routers and networks form the nodes of the graph, the edges are the links. Paths in the graph correspond to possible paths of packets sent through the network.

- Traffic: amount of traffic flowing through individual links (edges) in the traffic graph.
- Labels: display other useful information obtained about the network in question. Subject to further research.

Please refer to Section 2 to see how we plan to gather this information.

5.2 Time Dimension

The primary dimension of the data is time, that is, the user has interest in how the topology and traffic changes as time goes by. Both historical data and situation at present time are of interest to the users – the former is useful for post-mortem analysis (e.g. when assessing the effects of a network outage onto the paying customers), the latter for overall monitoring and administration of the network.

To change the time dimension, the user can change the time interval displayed. If a discrete point in time is selected, the UI will display the state of the network at that moment in time. If an interval is selected, the UI will display the "average network graph" which is an accurate representation of that interval. This motivated the features described in Section 3.

5.3 Network Graph Layout and Drawing

The single most important feature of the UI is to display a network graph. This graph consists of routers and networks, where edges connect routers to the networks.

During our preliminary experiments with the UI, we have reached agreement on the following:

- Nodes of the graph need to be placed automatically. For large networks, not having some sort of automatic layout support would be very impractical. We need to provide the user with a reasonable network graph even when the tool is used for the first time. At the same time, the tool should permit manual adjustments of the resulting drawing.
- For drawing of the network graph, we want to rely on the `dot` suite of graph drawing tools, using the `neato` spring-based layout engine. This is a mature program which provides decent drawings for sparse graphs, albeit with some limitations which motivate the rest of this section.
- Similar graphs often result in very different drawings under the spring model. This behavior is easily triggered by adding a single node or edge

to an existing graph. This is at odds with TODO described above, making it hard to track visually how the topology changed over time.

- Ideally, it would be possible to have positions of certain nodes calculated automatically, while other nodes could be "pinned" at the user-provided positions. It is possible to provide the desired location of a node in `neato`, but it is nonetheless subject to further transformations of the drawing, hence not useful in our case.

To provide the automatic layout feature and to overcome the limitations described above at the same time, we have formalized the relationship of the `neato`-obtained layout and the user's manually provided layout in the following way.

Network graph layout can be switched between automatic and manual positioning mode upon user's discretion. In the automatic mode, the position of each node and edge in the graph is calculated using `neato`. In the manual mode, nodes can be moved around and only labels and edges will be placed automatically by `neato`. The automatic mode is fully automatic, i.e. the positions given by the user are disregarded completely.

The user can switch between manual and automatic mode upon their discretion. Automatic mode is the default. When the users switches from automatic to manual layout for the first time, each node defaults to its `neato`-calculated position. If a new node appears in the graph, then in the automatic layout, its position will be calculated by `neato` as that of any other node (possibly resulting in a very different drawing compared to the previous one). In the manual mode, it will be placed in the upper left corner and the user will determine the final position of the node in the manual layout by moving it around as desired.

This approach has the following benefits:

- The user starts with a reasonably drawn graph which they can adjust as their wish, probably to reflect the physical topology of the network in question. If the topology of the network changes significantly, the user can switch back to automatic layout.
- `neato` still takes care of placement of edges and labels, even in the manual mode, taking the burden off the user.

5.4 Display

Display of the network graph can switch among the following display styles which affect how the edges of the graph are drawn:

Recall that an edge is a router-network connection.

- All: display all edges.
- Shortest Path Tree (SPT): display an SPT of the network graph. Since network graphs often purposely contain cycles, their SPTs are not unique. Therefore, we want to allow the user to switch SPTs.
- Traffic: described below.

The Traffic display uses colors to depict the amount of traffic through a link's associated physical interface. The heavier/brighter the edge, the more traffic flows through that link's L2 interface (relative to other links). This results in a drawing where the most distinct links are the ones which are currently used the most, and the least visible are the links which are used little or not at all.

Since network graphs can be quite dense, the goal of this display mode is to take most of the perceived visual complexity out of the drawing, while providing useful information about the traffic flow at the same time. It's different from the SPT display in that it displays all links, albeit somewhat blandly.

5.5 Topology and Link Metric Speculation

Besides our primary goal of displaying network topology and traffic at any given moment in time, we strive to allow the user to project how changes to the network topology will affect the network. This use-case is best understood through examples – we would like to see what happens when:

- link metrics are changed to favor certain links above of others, leading to better resource utilization,
- new links are added to the network, e.g. when redundant links are added to the network to bolster fault tolerance,
- links are removed from the network, as in (partial) network outages.

By default, the network graph contains precisely those edges which were discovered through the data collection process. The user is allowed to switch to an alternate view where edges of the graph can be added and removed and link metrics can be changed.

Most parts of the UI treat this user-provided data as if they were obtained in the data collection process. For example:

- SPTs are calculated in the alternate graph as they would be in the original,

- placement of nodes is subject to the same constraints as described above,
- the resulting graph can be exported as usual.

The major difference is traffic flow calculation. With extraneous and/or missing links in the graph, the distribution of traffic would be affected, and the modifications are often done precisely to see the projected change in traffic.

We are not sure that a practical algorithm exists that will calculate reasonable approximation of traffic distribution in the modified network graph. Further research is needed which is out of scope of this preliminary specification.

5.6 Implementation Details

We have agreed on the following:

- Overall goal: leverage modern web technologies to provide value, not to bother.
- Fundamental operation of the Web UI (e.g. display/export of network map) must operate correctly without JavaScript support. The distinction must be clear to the user.
- Where JavaScript is required, version 8 is assumed (ECMAScript 2017, released June 2017). This version seems to be supported well by modern clients and provides value to the programmer (better support for futures).
- Most of the back-end portion of the application (data collection, data storage, etc.) will be written in Go language. The most performance-critical parts may be written in C language, if we will find it reasonably beneficial.
- Auxiliary tasks (such as setting-up virtual testing environment) will be handled by set of scripts (either shell or Python).
- We believe that the choice of diverse programming languages and technologies in one project is not a sin. On the contrary, it allows us to use the most suitable instruments to solve diverse problems all being part of the same project.