Lan Network Watcher

A bandwidth and pps monitoring tool for Air-EISTI

author: funkysayu; reviewers: Titouan Bion, Gaël Berthaud-Müller; status: in review (2017-04-22)

Objective

This project will be part of the AIR-EISTI network infrastructure. Its goal is to provide an easy way to monitor bandwidth usage per host in a local area network (LAN).

We want to provide a tool helping AIR-EISTI system administrators to understand better who is consuming bandwidth and at which rate. This may assists network administrators in the future to better understand who is responsible for high downloading, enhancing the second solution of a high rating system.

This tool will act as a daemon exposing metrics data to Prometheus and eventually later providing an API allowing any client to get metrics of an host bandwidth usage (which can be then plugged to Icinga).

Background

Monitoring is an important part of AIR-EISTI. Knowing why something is going wrong is the daily question of a system administrator. Sometimes, existing tools are able to provide enough information to debug the system. Some other times, we want to provide additional metrics on our infrastructure to have a better understanding of what it is going on.

Since our network is rate limited in terms of bandwidth on several endpoints, we already made a system sharing bandwidth between the hosts of our network. Unfortunately, the recent growth of LAN event participants makes it hard to know the limit between a too low rating system that may lead to per hosts issues (e.g. for downloading games) and too high rating system leading to a global network latency.

Reader of this documentation are expected to have basic knowledge in networking, such as subnetworking and sniffing. Any knowledge on Prometheus and Go are not required but encouraged.

Acknowledgment

The initial solution of bandwidth sharing was proposed by Raphael Ehret with the traffic control configuration of our router. This uses a constant maximum rate for users that they cannot overlap (3Mbit/s).

We already have an implementation of a bandwidth usage monitoring, written by Axel Vanzaghi and plugged in the Lan Party Manager (LPM, available on the internal gitlab) administration panel written by Etienne Dechamps. This project has been designed to export a CSV read in LPM to provide metrics. It does not provide an API endpoint allowing any client / monitoring tool to get a metric for a specific user.

Overview

The program will act as a daemon sniffing the network on a specific network interface.

The user provides a network to monitor and a network interface on which the daemon will sniff the network. While sniffing the network, it will catch all packets from an host within the provided network, either it is sent or received. We only care about three parts of those packets: the host concerned (i.e. its IP/MAC), the size of the packet and either it is sent or received. Sizes will be cumulated, grouped by host (IP, MAC) and direction (Rx, Tx).

Every *t* seconds, we will update hosts metrics by calculating the average bandwidth used by an host, based on the sum of the packets size. Once done we will also maintain a bandwidth load average metric as the CPU load average.

Those metrics will be periodically updated on the Prometheus interface, with a *t* period of time. They will also be available on a gRPC endpoint, allowing any client to fetch the data to extend portability to other monitoring tool such as Icinga or Nagios.

Detailed Design

The whole application is composed of a single daemon collecting data on the users, exposing a HTTP server providing metrics to Prometheus and a gRPC server to collect data through a gRPC client. By default, Prometheus and gRPC server are disabled and at least one must be activated depending of the system administrator needs (otherwise the program would have no utility).

Packet collection

We will use the pcap implementation for Go to collect the data from the network. This provides us a standardized way to filter traffic that should be collected. As a use case, a network administrator may only be interested by monitoring packets going to a specific public subnetwork. The pcap filter `host net $subnet` would allow to filter all packets that has as source or destination an IP within the specific subnet.

By default, the tool will not take into account packets from an host within the specified subnetwork to another host from the same subnetwork. The router can be able to see those packets depending of the network infrastructure (hub based network or switched in master mode for example). We may add an additional option to capture them.

Once captured, the packets will be grouped per host IP. The implementation is fairly simple: if a packet has as source address an IP within the network, it will be grouped as transmission (Tx) from this host; the same is done for destination, this time grouped as reception (Rx). Note that this design may support the previously detailed option watching connection within the subnetwork.

Once a packet is grouped with an host, the tool will sum its size and increment the packet count in the appropriate direction (Rx / Tx). We can then represent the state of an host by a structure:

```
type HostSnapshot struct {
      Timestamp time.Time  // Time when the snapshot was taken
      IpAddress string
      MacAddress string
      Reception NetworkTotalStatistic  // Rx
      Transmission NetworkTotalStatistic  // Tx
}

type NetworkTotalStatistic struct {
      TotalSize int64
      PacketCount int64
}
```

This can be defined as the exact snapshot of an host connection use. However, we are more interested in the statistic within a period of time, as for example the bandwidth use.

Bandwidth and packet per seconds count

In order to calculate bandwidth and packet per seconds, we need to periodically make a difference between a previous snapshot and the current one. The period has its importance and depends highly of the user needs.

Calculation method

The calculation method is fairly straightforward. Let *dt* be the period of time on which we want to calculate the bandwidth and pps use of the watched hosts. Let H_t be the snapshot of the host network use. We then can define the instant bandwidth and pps reception use over *dt* time as following:

rx_bandwidth = (H_t.Reception.TotalSize - H_{t-dt}.Reception.TotalSize) / *d*t
rx_pps = (H_t.Reception.PacketCount - H_{t-dt}.Reception.PacketCount) / *d*t

The same goes for transmission rates.

Choosing the time period of counting

Selecting a correct period of time is a huge deal when monitoring network. Take a too wide *dt* period and you may not see people bursting the connection at high rates. Take a too small *dt* period and you will generate high CPU usage by the tool and a huge quantity of data.

Since generally speaking, the networking uses a per second basis to analyse and manage the traffic rates, we chose to use *one second* as the default period of time. As previously explained, this metric can be changed with a flag, based on the user needs.

Networking load average calculation

Load average calculation is useful to have a better understanding of the collected metrics over the time. As an example, a user may burst the connection from 0 to 100 to 0Mb/s during 1s, using a total of 100Mb, but another user may use the connection at a 25Mb/s constant rate during half a minute, using a total of 750Mb. Being able to detect those two profiles can be important for network administrators working on latency sensitive networks.

Networking load average works as CPU load average. We will analyse network use of hosts on a basis of 1, 5 and 15 minutes, which can be modified through a flag. To do so, a snapshot of every hosts is kept every seconds and the same calculation is done over a 60, 300 and 900 snapshots. We assume a second basis for snapshot saving is enough to get relevant values.

Some monitoring tools such as Prometheus allow to generate a load average like graph based on the previously collected metrics. However, since metrics are collected at a specific period of time that may not match the calculation period of time, some metrics might be forgotten, providing bad load average calculation. This is the reason we decided to directly implement this in the tool.

APIs

We will provide two ways to collect the metrics. One implemented with Prometheus and the second one with gRPC. Both can be enabled, disabled or parametrized through application flag.

Prometheus API

The Prometheus API is directly implemented with the Go Prometheus library. It is designed to be extremely simple, so the metrics are all collected in the same way.

We provide two counters `lnw_bandwidth` and `lnw_pps` updated at the same time metrics are calculated and defined with the following labels:

- `host_ip` and `host_mac` which respectively correspond to the monitored host IP and MAC
- `time_frame` which is the time period of calculation

Disabling prometheus API disable the counter update system.

gRPC API

The gRPC API provides a way for other monitoring tools to implement clients using the application calculated rates. This is highly useful for an integration with Nagios or Icinga, in order to monitor and set alerts based on burst and load average usage of hosts over a subnetwork.

The API will be define by the following proto file:

```
service NetworkWatcher {
      // Collect the statistics of a specific host within the
      // network.
      // If HostQuery message is left empty, the endpoint will
      // return the rates of the whole network. If the host
      // has not been seen during more than 30 minutes, the
      // endpoint raises a NOT_FOUND status.
      rpc Statistics (HostQuery) returns (HostRates);
}

// Message used as the rpc endpoint query.
message HostQuery {
```

```
        // Expected host IP. If left empty, the sum of all hosts
        // rates will be collected.
        string ip = 1;
    }

    // Message used as the rpc endpoint response.
    message HostRates {
        // Observed MAC address. Empty if querying the rates of
        // all the network.
        string mac = 1;

        // Bandwidth rates exprimed in Mbps.
        float rx_bandwidth = 3;
        float tx_bandwidth = 4;

        // Packet per seconds rates exprimed in pps.
        float rx_pps = 5;
        float tx_pps = 6;
    }
```

Note that we might provide a JSON over HTTP endpoint defined by the same proto if this becomes a requirement.

Future work

We want first to create a fully functional and tested sniffer able to listen on an interface and retrieve packets. This will then act as a producer, consumed by another routine handling packet analysis, requirement matching (the packet is from an internal host to an external source or the opposite), the groupement per host and direction and the summation. We will then provide a statistic actuator on top of the last routine, updating metrics of all users on a *t* period of time.

We will then take a look at how to expose those metrics to actual monitoring tool. We will first take a look at providing those to a Prometheus endpoint, periodically fetching them. Afterward, we will provide a gRPC endpoint used by any client to fetch metrics.

References

None.