Routing Protocol Lab

Implementing RSVP-TE on Quagga

Overview

In this project we will implement RSVP-TE from Linux/Quagga. We will use some code from a public domain RSVP implementation from ISI (circa 1994-95) that does not have any of the Traffic Engineering extensions and we will implement the rest from scratch.

We will implement only a subset of the protocol. In particular we will implement:

1. (simplified) packet formats
2. packet send/receive
3. LSP database
4. LSP refresh
5. LSP strict and loose EROs
6. LSP up/down based the state of local interfaces
7. LSP up/down based the state of next-hops
8. MPLS FIB in the kernel and communication with it through netlink
9. Graceful-restart

Our goal in the implementation will be scalability and robustness: we should be able to handle 100,000 LSPs and process their related update traffic, and withstand (gracefully) any amount of RSVP packet incoming rate.

We will start with a simple skeleton code that creates a new Quagga component called **rsvpte.** I will provide the code for the packet parsing (reading all the objects in the packet and the corresponding checks) and you will concentrate on the high level protocol mechanisms.

Architecture

RSVP-TE is used to setup LSPs along a network path requested by the user. The labels needed for forwarding of traffic over the paths are allocated by RSVP-TE and exchanged with RSVP-TE messages. RSVP-TE setups us the LSP by sending PATH messages downstream (towards the endpoint of the LSP) and RESV messages upstream (towards the entry point of the LSP). The PATH message carries the LABEL_REQ object while the RESV messages carry the LABEL object that contains the allocated label. When a nodes receives a PATH message for a new LSP it will perform admission control and if this is successful it will propagate the PATH message further downstream. When last node in the LSP receives the PATH message it will allocate a label, and put it in the RESV message that sends upstream. The RESV message will go through all the nodes in the LSP path. When a node receives the RESV, it will allocate a label, put in the RESV

message and send it further upstream. When the RESV arrives at the first node in the LSP, the LSP is declared UP. If an interface fails, then we tear down LSPs by sending PATHtear messages downstream and RESVtear messages upstream. When a node receives a PATH message but can not forward it further either because of admission control of because the next hop interface is down, it sends a RESVerr message upstream. The originator of the LSP may continue to send PATH messages in this case.

RSVP-TE is a soft state protocol. After the LSP is setup up, it stays up as long as all the nodes in the LSP path refresh it by sending PATH and RESV messages periodically.

The implementation will be organized around the following data structures:

1. Interfaces: these are the real interfaces on the system that we learn through zebra. Zebra will tell us when these interfaces are up, down, and what is their ip address, mask, MTU etc. They must be organized in some data structure so we can look them up based on their ifindex
2. RSVP-TE interfaces (struct rsvp_interface), these are the interfaces used by RSVP-TE (may be a subset of the real interfaces). They also must be organized in a data structure that allows us to look them up based on ifindex.
3. RSVP-TE sessions (struct rsvp_session). One session corresponds roughly to an LSP. Each session has an id that consists of the (ingress IP address, egress IP address, tunnel id). We need to be able to lookup these fast.
4. Path state block (struct rsvp_psb). For each PATH message we receive we create one of these. In our implementation there will be one struct rsvp_psb for each struct rsvp_session (in general there can be multiple psbs for a session). These will be refreshed periodically. The psb will contain information on which is the next downstream hop (nhop). Each time we receive a new PATH message a need to lookup the session it corresponds to an compared with the existing psb for changes.
5. Resv state block (struct rsvp_rsb). Similar to the above. Each rsb contains information on which is the next upstream hop (where did it come from). These are also periodically refreshed and each time we receive a new RESV message we need to find the session it corresponds to and compare with the existing rsb.

Packet reception will be through a packet read thread. Each time we read a packet we check if it arrived from a valid interface where RSVP-TE is running on. Then we queue it for further processing and schedule the read thread again for the next packet. We need to make sure that we do not admit too many packets so as to overload the system.

Packet sending happens through a packet send thread. All outgoing packets for all interfaces must be queued somewhere and the send thread will send them out. We may have too many packets to send (if we have too many LSPs) so the send thread has to make sure that it serves each interface in a fair way, does not prevent other “threads” from running and limits the maximum rates packets are sent out.

If an interface goes down, we need to check all the LSPs that go through it and update them. If the next hop for the LSP was a strict ERO, then we can not reroute it so we will have to tear it down by sending RESVtear messages upstream. If the next hop was a loose one, we will have to ask IGP to tell us how to reaching the next hop after the link failure and we will have to start sending PATH messages to this new next hop. We have to be careful since the best way to reach a next hop may change due to IGP changes, so we will have to monitor for such changes and keep the LSP next hop up to date.

If an interface comes back up, again we locate all the LSPs that want to use this interface and we send PATHs for them downstream.

Stages

1. Create the rsvp-te component in Quagga, basic packet send/receive. (2 weeks)
    i. Interaction with zebra to discover the interfaces in the system
    ii. Organize the interface information in RSVP-TE
    iii. Some elementary CLI
    iv. Start worrying about maintenance/debugging
        1. Understand the logging facilities of Quagga
        2. assert and other tools
    v. Architecture of the I/O subsystem, start worrying about scalability
        1. Internal queues to ensure robustness when handling incoming and outgoing traffic
        2. Packet send and receive tasks that get/put packets from the queues
        3. Quotas and fairness
        4. Shaping of the outgoing packet traffic
2. LSP database, LSP walks per interface, global (1 week)
    i. What is the best data-structure?
    ii. What is the key? What kind of walks we will need to do?
    iii. How to limit the length of the walk to remain responsive?
    iv. How to interrupt and continue the walk?
    v. How to handle deletes in the meanwhile?
3. Refreshes on timer, updates with interface up/down, next-hop up/down, refresh aggregation (1 week)
    i. How to scale the timers?
    ii. One vs. 100,000 timers?
    iii. How to aggregate refreshes?
    iv. How to avoid timer synchronization?
    v. How to monitor for next-hop status? What should zebra support?
    vi. How to change my outgoing interface when the next-hop changes?
    vii. Synchronization with the IGP when we boot?
4. Graceful restart (1 week)
    i. What state should I put in shared memory?
    ii. How to keep it up to date all the time?
    iii. How can rsvp-te tell that is it crashing/restarting?

5. Interface with kernel MPLS FIB (1 week)
    i. The status of the Linux MPLS kernel work
    ii. How to send only what changed?
    iii. The concept of the version walk