Instructions for part 1: I/O subsystem – packet reception and transmission

In this part of the project we will create the basic even structure of RSVP-TE. As we discussed in class, we want to break the packet processing into multiple smaller stages so that we can overlap them. We have three stages for now, packet read, processing, and packet write. The event handlers for each state communicate with each other through queues of packets.

Tasks/Event handlers

Each interface will have a read_queue and a write_queue. The read_queue will contain incoming packets that arrived at this interface and we have not processed yet and the write_queue will contain packets that we need to sent out of this interface but we have not sent them yet.

Read task reads incoming packets from the socket, checks the incoming interface and queues the packet in the read_queue of the interface. Packets are read into a static buffer so that for each incoming packet we need to allocate memory and copy its data. Memory is allocated in Quagga using the MALLOC/MCALLOC macros (we will revisit the memory allocation issue later). The read task limits the amount of packets it reads in one go to MAX_PKT_READ_QUOTA, and it limits the length of the read_queue of each interface to MAX_INCOMING_QUEUE_LEN. If the read_queue becomes too large the packet is dropped. The read task maintains intf_read_list a circular list of interfaces that have non-empty read_queues. When a packet is put to the read_queue of an interface, if this interface is not in the intf_read_list, it is added there. When there are no more packets to read or when the task has read MAX_PKT_READ_QUOTA, the task reschedules itself calling thread_add_read() . If there are more packets waiting in the socket then zebra will call the task again, if not the task will not be called again until there are new packets that arrived at the socket.

The process task gets packets from the read_queues of interfaces. Since all the interfaces with non empty read_queues are in the intf_read_list, the process tasks scans this lists to find which interfaces to read packets from. It processes up to MAX_PKT_PROCESS_INTF_QUOTA from each interface before moving to the next interface in intf_read_list. If there are not more packets in the read_queue of the interface, it is removed from the intf_read_list, else it is put at the end of the list. The process task processes up to MAX_PKT_PROCESS_QUOTA each time it runs. Right now since we do not have the code for the process task, the task will just produce a packet for writing for each incoming packet. These outgoing packets will be queued to the write_queue of the interface that they are to be sent out from. The length of these queues can not be more than MAX_PKT_OUTGOING_QUEUE_LEN. If the write_queue becomes to large, outgoing packets will be dropped. The task will produce up to MAX_PKT_WRITE_QUOTA and then it will reschedule itself. The task will maintain intf_write_list, a circular list of interfaces that have non-empty write_queues. When the

task can not continue any more (because it has written MAX_PKT_WRITE_QUOTA packets) it reschedules itself calling thread_add_event(). If there are no more packets to read the task terminates without rescheduling itself. The process tasks will be scheduled by the read task (using thread_add_event()) where there are new packets to be processed.

The write tasks picks up packets from the write_queues of the interfaces in intf_write_list and writes them to the RSVP socket. It writes up to MAX_PKT_WRITE_INTF_QUOTA packets from each interface before moving to the next interface in the intf_write_list and can not write more than MAX_PKT_WRITE_QUOTA packets each time it runs. If after writing MAX_PKT_WRITE_INTF_QUOTA packets from an interface there are more packets in the write_queue, if puts the interface at the end of the intf_write_list, else the interface is deleted from the intf_write_list. When the write task has to stop (because it has written MAX_PKT_WRITE_QUOTA or a write to the socket returns EWOULDBLOCK) it reschedules itself using thread_add_write(). If there are no more packets to write then the write task does not reschedule itself. The process task will schedule it (using thread_add_event()) when new packets have to be written.

Packet pacing

As we discussed, most times we do not want to send protocol packets to our neighbors at very high rates because this may overload them and make him drop packets. Thus, we pace the packet sending, i.e. we impost a maximum packet sending rate. This can be done per socket (i.e. pace all the packets sent by RSVP independent of their destination), pace packets going out of the same interface, or even pace packets going to the same next hop (remember If I have an Ethernet interface, there may be multiple RSVP routers behind this interface).

You should implement a packet pacing mechanism. Ideally one would want to control precisely the sending rate of packets (e.g. 1 packet/1 msec) but this may not be practical, because we will need a timer for each packet we send and we will need a timer with a very high resolution. Having timers fire every 1 msec may burn too much CPU. So, it is more practical to have instead 50 packets/50 msec, so the timer will fire every 50 msec and will send up to 50 packets. Of course these 50 packets may be sent all together but the burst will be small enough so that the receiver will have time to process the packets until the next burst comes. You should allow for certain types of packets to go through without pacing, for example the HELLO packets.

Data Structures

Interface data structure: you need to organize the RSVP interfaces in a data structure so you can look them up fast. The key for the lookup will be the ifindex of the interface. This ifindex is given to RSVP by zebra when it notifies its about the interfaces in the system (function rsvp_interface_add() in rsvp_zebra.c) and also when a packet is received (function rsvp_recv_packet() in rsvp_io.c). In general there will be few rsvp

interfaces and additions/deletions of interfaces will be infrequent. Thus, the data structure you will use does not have to support efficient additions/deletions, just fast lookups. A hash or a simple tree will do fine here.

You will need a linked list for the read_queue and write_queue of the interfaces and the intf_read_list and intf_write_list. Any kind of linked list will be fine. Quagga implements multiple types, for example see the ospf_fifo_* functions in ospf_packet.c

Interesting experiments

You can use two RSVP processes running on different systems, one will be the sender and the other the receiver. You can have a loop inside the receiver sending RSVP packets to the sender. You can control the rate that packets are sent by changing the parameters of the packet pacing. If you disable packet pacing packet will be sent as fast as the sender can send them.

It is interesting to observe the impact that the settings of various parameters have on the CPU consumption of RSVP. Some experiments:

1. Change the granularity of the packet pacing timers (say from .1 msec to 50 msec) and see if you notice any difference in the RSVP CPU consumption
2. Change the granularity of work the three tasks do (for example reduce the MAX_PKT_READ_QUOTA and MAX_PKT_WRITE_QUOTA) and see what difference it makes in RSVP CPU consumption, if any. One would expect that if we do very little work in each handle and we keep rescheduling the handlers we may burn too much CPU in the scheduling of events. Can you see something like this? Put a sequence number inside the RSVP packets you send so you can count how many are getting lost. Do you see any difference to the amount of packets getting lost (you will see losses when the CPU gets saturated).
3. Change the size of the socket buffers at the receiver and see if you notice any difference in the amount of packets dropped with and without packet pacing. Can you determine what is the maximum size burst of RSVP packets (packets send back-to-back from the sender, i.e. without any packet pacing) that this receiver can handle before dropping packets?