Instructions for part4: Timers

RSVP needs to refresh the PATH and RESV state is has received. Periodically, new PATH and RESV messages are sent to the downstream and upstream next-hops of the LSP respectively. Such periodic activities are commonly implemented using timers. A timer is a mechanism that allows us to schedule an action for some later time. Quagga provides a timer API that allows timers to be created and destroyed and to have their value set or reset. Each timer is associated with a handler, and when the timer expires the handler is executed. You can find the timer implementation in file lib/thread.c (thread_add_timer()). Also, you can see where timers are served in the main scheduling loop of Quagga in function thread_fetch() in file lib/thread.c. Given that there may be many 1000s of RSVP LSPs/sessions, it is important to implement timers in such a way so that they scale, i.e. we can efficiently create, destroy and schedule 10,000 of timers. Unfortunately, the Quagga timer implementation is not very scalable. All timers are kept in a sorted list, based on their time to expiration. Each time we insert a new timer or modify the value of an existing timer, we have to insert the timer in the right position in the timer list. This is a linear operation that will scale badly when we have thousands of timers. An additional issue is that if many timers expire at the same time, then Quagga may end up spending too much time servicing these timers and neglecting other (potentially higher priority) actions such as events.

First, lets consider what timer functionality is required by RSVP. RSVP needs to periodically refresh both PATH and RESV state. Thus, we will need one timer for generating the periodic refreshes. If updates have not been received then the corresponding state (PATH or RESV) is considered expired and the LSP is torn down. This is also implemented with a timer. Each time we receive a refresh we set a timer to the expiration value (usually called in RSVP time to die (TTD)) and when the timer expires we clean up the LSP. In our implementation so far we have only PATH messages, thus each node will send PATH refresh messages to its downstream node and will receive refreshes from its upstream node. The above would imply that we would need 2 timers for each session (one for sending the refresh and one for the TTD) but it is possible to just use a single timer for both events. (How?). This is important since it will dramatically reduce the amount of timers we need for a given number of sessions. Timers in RSVP are created once, when the session/LSP is created. Then we constantly change their values during the lifetime of the LSP. In theory, we do not care so much about fast create/delete but rather for efficient rescheduling of the timer, i.e. change its value.

As we discussed in class, to avoid synchronization of these refreshes, the time that the messages are sent is **jittered**. Each refresh can be sent at any random point in the interval [R – R/2, R + R/2] where R is the refresh interval, i.e. how often refreshes are sent. We calculate when the state times out using the following formula: TTD = (K+0.5)*1.5*R. R is the refresh interval, and K is the number of refreshes that must be lost in order to declare the LSP down. R is specified in seconds but it is a good idea to further jitter this time down to millisecond resoltuion. For example if we end up having to schedule 500 timers to expired 10 seconds later, it is much better to spread them around inside this seconds (i.e. one of them 10 seconds and 40 msec later, the other 10 seconds and 150 msec) and so on.

What to do in this phase:

- In this stage of the project we will replace the timer management code of Quagga with better code that can achieve fast creation and deletion, fast insertion and controlled execution of expired timers. Clearly we need a better data structure for storing the timers. There are two main options:
    - A priority queue or heap. This is a tree that ensures that the object with the min (or max) key is kept at the root of the tree. Logarithmic complexity for insertion and deletion. Example code can be found in: http://www.csua.berkeley.edu/~ranga/school/cs161/index.html
    - A calendar queue: looks like a hash table where timers are hashed based on their expiration times and put into a short list of events in the same bucket. Depending on the patterns of the timer expiration values it can offer linear time insertion and deletion but may need reorganization and waste memory. The paper (with pseudocode) is in http://portal.acm.org/citation.cfm?doid=63039.63045

  Selecting efficiently the next timer to fire is the same as selecting the next event in an event driven simulation. Looking at the event scheduler of ns-2 you will find a calendar queue implementation. In the following page you can find some more information and a heap implementation: http://www.cs.caltech.edu/~weixl/technical/ns2patch/

  You can implement any of the above and integrate it with the current Quagga timers and keep using the same API for the timers.

- Handle expiration of multiple timers at one go. This can be handled either at the application or at the Quagga scheduler level. In general it is better to let the application (i.e. the RSVP protocol) control the scheduling of the timer execution since it has a better idea of the tasks that need to be executed. Thus we need to add a new API call that will effectively return a list of all the timers that have expired:
    - timer * get_expired_timers();

  Then RSVP can get the list of timers and process them in a controlled way so that it does not starve other protocol operations when multiple timers have expired. A simple way to do this is to configure a TIMER_QUANTUM that will tell you what is the maximum number of timers that can be processed before stopping to check if there is other work to be done. Then you must be able to resume the timer servicing until all expired timers are handled.

- Check the performance of all the above. In particular
    - Setup many (say 10,000 LSPs) with refreshes (every 1 second) and see how the existing Quagga implementation does in terms of CPU time etc…
    - Compare with the more efficient implementation you have done
    - Repeat with different refresh interval values
    - Do you often have multiple timers that have expired at the same time?