Instructions for part 2: I/O lookup structures and walks

In this part of the project we will focus on how to organize the basic information of RSVP and how to process it. We will mainly work on the concept of the *walk.* In RSVP the most important piece of information is the session information (struct rsvp_session) that (in our simplified project) corresponds to an LSP. In a large network there may be many LSPs, thus we have to ensure that our implementation scales well (say 100,000) LSPs. To ensure good scaling we need efficient ways for the following:

Lookups: PATH and RESV messages are sent for a particular LSP, thus for each incoming such message we first have to lookup the session information for this LSP. As discussed, the key for this lookup is the (address of the ingress, address of the egress, lsp-id). Ingress is the node at the beginning of the LSP and egress is the node at the end of the LSP. Lsp-id is a number that is chosen by the ingress node. To ensure fast lookups we need an appropriate data structure. The selection of this structure depends on the patterns of inserts/deletes/lookups that we will have to deal with.

Consider how an RSVP network may work in practice: usually RSVP-TE is used for traffic engineering (TE). Although we have not talked much about TE in the class yet, usually what happens is that the administrator of the network will compute an almost full-mesh of LSPs between the aggregation routers in its network (i.e. between each PoP). These LSPs will be computed with off-line tools and will be setup in the network and will remain active for long time (weeks/months) until they are updated through another run of the off-line tool. If a provider has 100 aggregation routers, it will have 10,000 LSPs in its core routers. As a result:

- LSPs have a long lifetime, after they are established they remain active for weeks. So I will not have frequent insertions/deletions. These will happen mainly when some link or node fails and then LSPs change paths, but these are not very common events. Things would be a bit different if I have LSPs that follow the IGP paths (i.e. without an explicit route). In this case, when the best paths in the IGP change, a lot of LSPs may change paths too and this will cause multiple inserts/deletes in the routers that the LSPs crossed. In general though I can assume that the deletes will be much less frequent than the lookups.
- Inserts will happen all at once or at least close to each other either when the router boots or when the configuration of the LSPs changes. After this initial insertions there will be few additional inserts.
- There is no particular pattern in the inserts, i.e. the 10,000 LSPs will be probably established in an arbitrary order.
- Deletes will be quite rare in general
- If a router has N LSPs, all of them will be refreshed regularly, thus I will need to lookup each of them. I can assume that the lookups will be covering all the LSP set and will in general be uniform.

What does all this tell us about which lookup structure is the best? First, since lookups are uniform, there is no need to do something like a splay tree that tries to optimize access to the most frequently accessed elements. Since there is no pattern in the inserts, it may be possible to get away with an unbalanced tree, but it is risky. If for some reason there is a pattern (or a pattern emerges over time – for example as routers fail and then come back and it establish again all the LSPs that they were the ingress for) then over time we may end-up with a very unbalanced structure. A balanced tree is the generic solution that would apply well here. This treats the key of the session (ingress, egress, lsp-id) as flat. Another approach would be to use the hierarchy in the key, for example have multiple trees, one for each ingress address, where the key in these trees would be (egress, lsp-id). But it is not clear that this additional complexity would improve lookup times. Hash tables could also be used (either a single large hash table or multiple ones, one for each ingress node) but one needs to be careful when selecting a hash function in this case. The (ingress, egress, lsp-id) keys are not totally random, the ingress and egress addresses will be addresses of the routers in the network, so they will come from a small set of 100-200 addresses and the ingress and lsp-id will be correlated since the ingress nodes assigns the lsp-id. Thus, if we are not careful with the hash function we may end up with too many collisions. Thus, overall a balanced tree is the safest choice.

Walking: In certain occasions we will need to find quickly all the sessions that satisfy a condition. For example, when an interface goes down, we want to find all the sessions that were using this interface for their PATH messages. There are two ways to deal with this. The first is to create an additional tree for each interface and enter in this tree pointers to the sessions that are using the interface for their PATHs. Thus, when the interface goes down, we just scan its session tree and we find quickly all the sessions affected. One problem with this approach is its complexity, when a session starts using a new interface for its PATHs, I have to remove it from one interface tree and put it in the other. Also, I may want to find sessions that use an interface for their RESV messages, and maybe some other conditions too, it is not realistic to keep adding more trees for each condition. Increasing the complexity of the code it usually not a good idea when trying to build a robust software system. A more practical (although maybe little bit less efficient) method is to just have a singe tree of sessions, and **walk** it (i.e. visit all the nodes in the tree) to find the sessions that satisfy this condition. The processing of the sessions that satisfy the condition is done during the walk. Usually, we walk the nodes of the tree in-order. When we call the function that implements the walk we usually pass a handler function that is invoked for each node visited by the walk.

Walk scheduling and responsiveness: When we do this walk, we may have to process many sessions. For example, If I have 10,000 LSPs and 4 interfaces, then in a world of uniform distributions there will be about 2,500 sessions per interface, if one interface fails then I will have to do some work for 2,500 sessions and this may take some amount of time. In a single CPU system that is using events, during this time, I can not do any other protocol processing. This may cause delays in sending important packets or servicing some timers that have expired or failing to read arriving packets from the network so that they get dropped by the kernel. The solution is to break the walk in smaller stages, and each time we complete a stage to check and perform any pending

work, and then continue with the next stage, until the processing is done. To be able to do an interruptible walk we need to:

- Continue the walk where we stopped the previous time. This means that we have to remember the node of the tree where the previous stage stopped and return to it the next time and continue the walk from there. To continue the walk we first do a lookup for the node where the previous stage left off and after we find it, we continue the walk from there.
- Deal with changes that can happen during the walk. For example, as we walk the tree and process sessions that are affected by an interface down, we may need to delete a session. But deleting a node of the tree while we walk it may cause a crash. One common solution is instead of deleting the node to *mark* it as deleted, i.e. set a DELETED flag. Then, after the walk is completed, we have to delete all the nodes that are marked as deleted. Another approach is to lock the nodes of the tree that can not be deleted during the walk. When the walk moves to the next nodes we unlock the previously locked nodes and if some is marked as deleted we delete it.
- Deal with changes that can happen to the tree between the stages of the walk. While the walk is interrupted the tree can change with additions and deletions. In principle this should not cause any problem. When we continue with the walk, we will find the node where we left off and we will continue. But what will happen if the node that we left off last time was deleted? It will not be possible to find the node where the previous walk stopped. This could be solved with a variation of the tree lookup function that finds a node with a given key or the next one if that node is not in the tree.

Range walks

In some cases we may want to do a *range* walk, i.e. walk all the elements between a minimum A and a maximum B. In this case, it is more convenient to walk the tree in-order. Of course, a range walk is meaningful in a tree where there is some natural order. In the RSVP session tree, there is no such ordering, so it does not make much sense to walk all session between (source_A, dest_A, lsp-id_A) and (source_B, dest_B, lsp-id_B).

Version walks

In our example with rsvp sessions and interfaces, we know that when an interface goes down, many sessions will be affected so the cost of the walk is justified. Imagine that things were different and each time an interface went down, only very few sessions were affected. In this case, I would have to walk the whole tree of 100,000 sessions just to find 5-10 sessions, which is rather inefficient. A technique for addressing this is to introduce node versions. Each node has a version number, and each time something changes in the node, its version increases by 1. Each time we change a node's version we make sure that we copy the new version to all the ancestor nodes of the changed node, all the way to the root. In this way the version of a node is the maximum version of all its descendants. Now consider that we perform a walk and we want to find nodes that have version larger that a minimum version we specify. If during the walk we reach a node that its version is lower than the requested minimum version, we know there is no need to walk its sub-tree and we save a lot of processing.

What to do in this phase:

- Implement a balanced binary tree for storing the session information. You can use any public domain code you can find for this purpose, see below for the AVL library. You will have to modify this tree to store pointers to rsvp_session objects and use the (source_ip, dest_ip, lsp-id) information as the key in the tree. Assume that the key is in a structure like

```
Struct rsvp_session_key {
        u_int32_t    ingress_addr;
        u_int32_t    egress_addr;
        u_int16_t    lsp_id;
 };
```

- Implement a function

```
int  walk_tree (struct rsvp_session_key  *from,
                struct rsvp_session_key   *to,
                struct rsvp_session_key  *last,
                Int  limit,
                int (handler)(struct rsvp_session *));
```

  That is delete safe, walks the nodes between *from* and *to* and calls function *handler()* for each of the nodes. If it visits more than *limit* elements it stops. Returns the node it left off in *last*. If last is NULL, the walk is done. If *from* is NULL then the walk starts from the first node. If *to* is NULL the walk will continue until the last node. The function returns the number of nodes it visited.

- Similarly, implement

```
int walk_tree_version(int min_version,
                      int max_version,
                      struct rsv_session_key *last,
                      int limit,
                      int (handler)(struct rsvp_session *));
```

  that will walk only the nodes between the two specified versions.

You can use any type of balanced binary tree implementation. A convenient implementation can be found in the AVL library (http://www.stanford.edu/~blp/avl). The red-black tree with a parent pointer is a good choice (file prb.c). This type of tree uses a parent pointer to find the next node in a walk. The tree nodes contain a left and right child pointer, a parent pointer and the pointer to the actual payload (in our case this will be a pointer to a struct rsvp_session). This code implements a traverser which is used to implement walks: it stores the current node in the walk and each time it is called it returns the next node in the tree. You will have to figure out how to make it delete safe and implement the versioning.