David Chanady
David Erickson
Filip Paun

CS344 Router Design

Our goal was to build an easily extensible and testable router. To accomplish this we had two priorities we have when building our code.

Firstly we wanted to ensure that we keep our methods short and to the point. Long methods tend to build up excessive amounts of logic and make it difficult to extend and maintain down the road. Shorter methods that accomplish one specific action make it easier to add more steps in the functionality down the road. An example of this is the tiered hierarchy of our router's packet flow. Each tier could be its own method which asks the questions specifically on that tier, then calls further methods if they are correct. That way at any tier it is very easy to add further logic or processing in.

Secondly we put as much code as possible in to our own header files, and made calls from the default project files in to our own source files. In this way we made our own interface that we used to call and be called from within the default project files. This allowed us to abstract away our logic from the underlying platform that is actually sending and receiving traffic, be it VNS, NETFPGA, or something else.

I believe we were very successful in these goals. We were easily able to test different sections of the code, and easily able to change/update functionality due to our modularity. Further we were able to seamlessly switch back and forth between VNS and the NETFPGA platform when necessary.

To facilitate modularity we created a core router_state structure (stored as the 'subsystem' in the sr variable). This structure was passed around to virtually all our methods so we were able to keep state in one structure, and keep things clean by not using global variables. This structure kept handles on all locks, threads, and global state variables. We further kept all structure definitions and defines in a single header file, so anytime modifications were needed we knew exactly where to go to.

Nearly all lists of structures in our system are stored as linked lists. This allows software to run completely independent of any limitations imposed by hardware. Thus software has no limitation on the number of routing table entries, arp cache entries, etc. Limitations are only imposed when writing these out to hardware, but have no effect on the operation of software itself.

We chose to heavily thread our architecture (~17 threads), which provided some clean abstractions of functionality into their own workspace, but also forced us to use a very disciplined locking system. This we did, and have had great success with this architecture. A few of the functionalities of the various threads in our system include: PWOSPF hello packet broadcasting, PWOSPF link state broadcasting, PWOSPF link

state timeout enforcement, arp queue timeout enforcement, Dijkstra calculations, NAT entry maintenance, port stats tracking, and webserver threads.

Basic routing functionality was cleanly implemented in layers of abstraction. We followed the discipline of naming all our source files as the category of work they performed, and sticking to it. For example ip processing is performed in "or_ip.h/c", arp processing in "or_arp.h/.c", etc. As packets came in we would get more and more specific on what they are and what we need to do with them. For example once a packet arrives, if it is arp it heads to processing in the or_arp files, if its ip it heads to or_ip. Further if its ICMP it goes to or_icmp, etc. This method of laying things out allowed us to easily interpose further logic (like NAT) without drastically modifying the way processing is performed.

We spent a fair bit of time working on our CLI. We wanted to make it as easy as possible for us to add/remove commands, and make the code as clean as possible. To facilitate this we created an interface method "register_cli_command", which takes the command, and a function pointer that will be called when that command is executed. Then when the client issues a command to the cli, we simply do a longest prefix match of the client's command against all the registered cli commands, and then call the function pointer associated with the command. It has greatly simplified our code, and we feel it is a very elegant solution.

Initially we were required to implement our CLI over telnet, which we did. After awhile however we got very tired of having to telnet in and type commands so we moved on to create a fully operational web-based interface. To make this happen we created a mini webserver that runs entirely inside the router. It is a thread pooling webserver, so we spawn 5 threads to begin with, and as http requests arrive they are put into a queue, and the 5 threads pull requests off the queue and service them. We have the ability to serve virtually any type of file residing in our www subdirectory. The webserver appears to be very performant (assuming the LWIP/LWTCP options are set properly), and stood up to a very long torture test without leaking any memory.

While creating the web interface we did not want to break backwards compatibility with the telnet interface, and we also wanted to be able to add a command and have that command immediately available via both the web and telnet. To make this possible we created a url that takes as a parameter the command you want to enter, runs it through the same processing pipeline a telnet command runs through, and then returns the results to the web. We use AJAX (Javascript and XML) to drive the web interface. That is, when you click on any of the links, or enter a command, the result is fetched in the background via Javascript, and as soon as the response is received it replaces the existing result with the new result. This allowed us to create automatically refreshing links on the left side that could execute commands at a set periodic delay (IE if you want to watch the HW as it updates), and have the screen seamlessly update so it appears you are running the application locally. This is in comparison to having to click the refresh button and watching the entire contents blank out and refresh. We further harnessed Javascript to create a nice stats overlay if you click on the "Our Router" logo, it will pop up a small

overlay window showing aggregate in/out traffic on each hardware queue by number of packets and throughput.

One element of our interface that was required to be implemented was the sping command. Our implementation is as follows: we dispatch the ping, then we put that thread to sleep on a conditioned timed wait. When an ICMP echo reply comes in to the server, we add it to a queue of ICMP echo reply packets, then notify all the threads sleeping that are waiting on a reply that a packet has arrived. The threads then wakeup and attempt to grab the mutex then examine the list to see if the reply is for them or not. The threads waiting for a reply will wake up periodically and when packets arrive, eventually if no reply is received they will time out and report back to the user. There is a chance that an ICMP echo reply will not be handled and deleted by an sping thread, so they are timed out as well and there is a thread that does this cleanup work.

PWOSPF functionality can be broken down into two sections, one section for handling the HELLO packet operations, and one for the LSU packet operations. HELLO packets are identified in our IP processing method and passed off to our methods in our pwospf source file. We augmented our existing interface structures to contain the information needed to be tracked by the HELLO packets. When we receive one, we update as necessary, and potentially trigger LSU database updates/floods/Dijkstra. Further, we have a thread running whose job it is to time out any expired entries, as well as handle broadcasting of HELLO packets for our interfaces on a set schedule.

To handle LSU and Dijkstra processing we have a database of structures that is made up of parent router structures, each containing link structures, representing the advertisements in LSU packets. This database is updated from incoming LSU packets, HELLO updates or timeouts (for keeping track of active pairs of links between routers), LSU timeouts, and interfaces coming up or going down. We have a number of new threads running to handle this processing, specifically threads to handle LSU broadcasting, LSU timeouts, Dijkstra processing, and one that sends out queued LSU packets. The last thread was required because some of our locking decisions did not make it possible to send packets directly out from where it needed to happen due to deadlock possibilities. Many new PWOSPF related CLI commands were added as well, available from the “pwospf ?” command. Note at this time we do not currently support adding or removing the default route from the CLI, as we were unable to test against a topology with two connections out to the rest of the internet, try it at your own risk.

When we integrated with hardware we added a large number of commands to print and poll values from hardware so we were sure the things we are setting were correct. As mentioned earlier we abstracted away anything relating to the way the underlying interface works. For example we added a flag on our router_state structure “is_netfpga”, thus anywhere in our code we may need to do something specific to the underlying interface, we query this flag. We are very confident about our interaction with the reference hardware.

The advanced feature we chose to implement was Network Address Translation (NAT). NAT added a few new processing steps to our packet processing. Firstly when packets arrive on the designated Wide Area Network (WAN) interface, after we identify the packet is IP, but before any further processing, we need to look in to our NAT table and check if we need to rewrite the destination address/port. Similarly when packets are determined to be exiting on the WAN interface, and were not generated locally, we have to rewrite the source ip/port if an entry exists, if not we need to create a new NAT table entry and perform the rewriting. We had to be careful to deal with edge cases such as ICMP echo requests and replies that did not have a port, as well as unwrapping ICMP error messages and carefully rewriting the enclosed packet headers in their payloads. Further when working with hardware we had a limitation on the number of entries that could reside in the hardware table, so we had to track how often our entries were being used or 'hit', and ensure the 'hottest' entries were kept in hardware and kept out of the slow path (software).

Testing methodology proved to be a crucial element, as expected, during our development. We were as careful as possible to test all edge cases we could imagine, because they always come up sooner or later. We were also very careful to perform regression testing on previously implemented features to ensure we did not break anything. This was very important when working with the hardware and drastic things were being done under the covers that software did not have control over. Initially we had planned on creating test scripts to cover all cases but the time constraints of the project just proved too tight to be able to dedicate time to something such as this. So we became very good at auditing each other's code and testing to ensure everything worked properly. We also spent a significant amount of time with valgrind ensuring we did not have any leaking memory (within our control), as if you even leak a little bit with each packet it adds up quickly when you are a router. A bulleted list of test cases is beyond the scope of this document.

In conclusion, the time we spent up front in architecting the system, and sticking to these disciplines throughout the rest of our time with it, proved to be a huge win for us in not only having the code actually work and be successful, but also truly be maintainable.