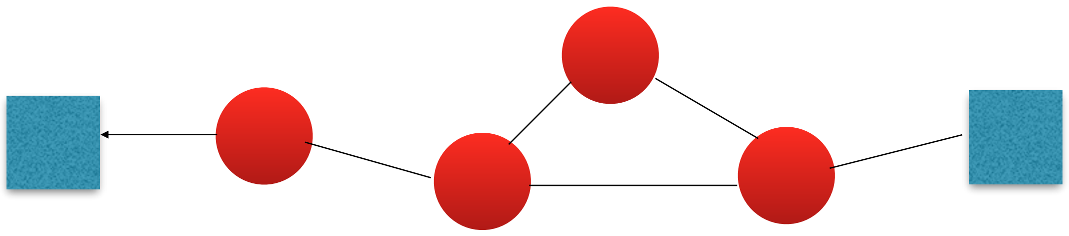
CS 2520
Final Report

Steven Forrest
Yuyu Zhou

Introduction

This report aims to discuss what we learned and what we were able to implement over the past semester. We are proud of what we have accomplished over the past 6 weeks with writing and debugging over 5000 lines of code. Our code is able to successfully route packets between end systems while incorporating problems like bit errors and looping. In order to accomplish this complex routing our routers are able to send routing information between each other including hello, ping, link state advertisement and neighbor acquisition. The code that was developed accomplishes almost all of the tasks that we set out to do in our design phase and we have even done additional work that we were not expecting during our design phase.

Our code has been tested using the AFS system, the unixs.cis.pitt.edu machine, and the element machines in the Pitt CS department. For our experiment, we designed one consisting of four routers and two end systems. The topology of the network is shown below.



The squares represent our end systems and the circles represent the routers. This topology setup can be seen in our config files. In our case each of these represents a different machine in the elements cluster. However, if we were more resource limited we could have multiple routers running on each linux machine. We are confident that our program would also be able to run between domains like unixs and CS since we are utilizing AFS and sockets that only require the IP address and port.

The experiment utilizes a four routers scheme for multiple reasons. First, it allows us to have a loop that the system would have to deal with and allows our cost calculations to be utilized. Secondly, it allowed us to have one link fail all while the packets would still be able to reach their destinations. Lastly, it allowed us to have a varying number of connections from each router. This allowed for test that it would check that it followed the correct path.

The make files located in each directory provide an easy way to build these files if they are transferred to a different machine. The routers require a make clean in order to remove all stored information about the routing data. For more verbose information please look at the logging file located in each directory.

Breakdown of code

Our code consists of two main units: the end systems and the routers. There is some cross over of code as we did not need to reimplement CRC, the library for the logger and the library for the socket on each side. The following is included just as a quick reference and to ensure that you have all the files necessary to successfully run this program.

The end systems consist of the following classes:

- app.c
- sw.c
- app.h
- endSystem.c
- lsrp.c
- sw.h
- checksum.c
- liblog.c
- lsrp.h
- checksum.h
- liblog.h
- commonItems.c
- libsocket.c
- socket.c
- commonItems.h
- libsocket.h
- socket.h

A description of each of these files can be found in the readme located in the /endsystem directory. The main class is endSystem.c.

The routers consists of the following classes. The are broken up into three main directories config, packet, and socket. Config sets up the basic configuration, and logging information for the session. Socket does all the low level socket implementations between the server and the client. Packet does everything else from setting up the different packet types to dijkstra’s algorithm. The lsrp-router is the main class for all of this.

- lsrp-router.c

Config/
- config.c
- libfile.c
- liblog.c

- config.h
- libfile.h
- liblog.h
- lsrp-router.cfg

Packet/
- dijkstra.c
- hello.c
- libqueue.c
- lsa.c
- neighbor.c
- packet.h
- ping.h
- dijkstra.h
- hello.h
- libqueue.h
- lsa.h
- neighbor.h
- ping.c

Socket\
- client.c
- getaddrinfo.c
- libsocket.c
- server.c
- client.h
- getaddrinfo.h
- libsocket.h
- server.h

Items Implemented

We feel that we accomplished the core items listed in the Project Check list. Below we will highlight the items we are especially proud of and the ones where tough decisions made as to what direction made sense given our limited time.

The following are decisions made about the end systems. We decided not to make a GUI interface. We felt that our time was better spent making the core functionality work and if needed in the future we could easily add this functionality. Instead we have for a command line interface that gives the user certain functionality. When the end system is started it does the initial configuration and then waits for commands. Commands can be shown by typing help. We choose to have a static timeout that can be configured in the config files. This was part of our initial design. Our code could easily set up handle the tracking of round trip time since we fork to send a file so we would just need to time the child process. For flow control, Stop and Wait was used because it simplified our design, made it easier to test and was what we decided upon in our initial design.

The following are decisions that were made at the router level. We were able to successfully added packet error to the router level. However, it made the setup of the link state routing protocol extremely difficult so we ended up removing it from our final design. We were able to successfully implement split horizon, poison reverse and link failure.

The link state routing protocol follows what we proposed in our design phase which is based off of the information presented in the project description. Some key choices were the use of the ping pong bit to tell us if the packet needs a response and the use of four bits for our packet life to ensure that the packet did not get stuck in a loop. Our LSA format is based off of this format:

Router ID
Packet Type
Advertising Router ID
LS Age
LS Sequence Number
Length
Number of Links

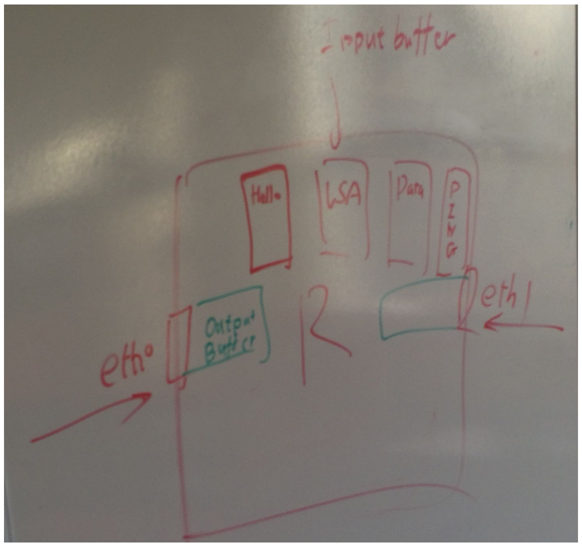
Link_ID_1
Link_Data_1
…
Link_ID_n
Link_Data_n
Packet Checksum

The basic steps that were implemented in our routing protocol:
- Discover neighbors and learn their network addresses;
- Measure the delay or cost to each of neighbors (done by icmp packet delay);
- Construct a packet containing the information just learned;
- Send the packet to all of the other routers;
- Compute the shortest path;

Most of our link state protocol implementation can be found in the packet folder in our router. For example, information about the ping, neighbor and LSA can be found in the ping.c, neigbor.c and lsa.c. When running the router there are numerous show commands that show the routing table and the link state database.

For errors, out network simply drops the packet instead of sending a negative acknowledgement. This simplified our design and allows us to rely on the timeout on the client.

In addition to the requirements we added two key features that allowed us to better test our program and recover from an attack. First, we gave the ability to specify the config files on the command line of the routers. This allowed us to test out of on AFS directory with our experiment design that is described in the introduction. Each router simply refers to another config that has a different topology listed.



The second big item was our design of buffers in the routers. After our initial implementation it became apparent that if the router were slammed with data packets then the hello, ping and other routing information would be stuck in the buffers or even dropped due to buffer overflow. By the time they would be processed they would no longer be applicable. To combat this potential security problem, multiple buffers were utilized so that there would always be space for the routing information and we would be able to reach them in a timely manner. Above is a crude drawing of this setup. There are four input buffers shown and one output buffer per port. The input buffers consist of Hello, LSA, Data and ping.

Experimental Analysis and Tests performed

We have done a wide range of testing and tweaking of our program to try to achieve the best results. We were able to incrementally test our end system by pointing it to another end system with no routers in between and use the network provided by unixs.cis.pitt.edu and elements.cs.pitt.edu.

We found a couple of key limiting factors in the speed of our routing network with regards to our File Transfer Time. The first problem is how we are implementing timeout on our endSystems. We fork the process and have the parent incrementally check to see that child has finished or if the timeout has occurred. The parent sleeps for 100 microseconds in between checks so that the process does not use all of the CPU. However, if and when the child finishes before the 100 milliseconds is up then the client is idle when it could start sending the next packet. So the peak packet rate between two end systems is capped at 1/100 milliseconds if no routing is involved. This formula is from the peak rate that was discussed in class.

We also found that our routers were limiting the speed through the network as well. We believe that this is due to the frequency at which we are updating the link status, acknowledging and sending hello packets. With some tweaking we should be able to improve both the end systems and the routers peak speed. We were not able to come up with a peak rate as the data was not consistent. It depended on the state of the router at the time of arrival and where in LSA/hello/ping cycles it was.

We concluded that two other parameters have large impacts on our network speed: MTU size and timeout value. Since the timeout is static the value may be too large leading to long times to timeout when an error occurs or too short leading to packet retransmission. This can be fixed using an simply exponentially smoothed algorithm to calculate the timeout. The MTU size is a factor in the speed because with each additional packet we send, we must send more header information including CRC, and destination/source IP.

Our bit errors are calculated using the following method:

```
void packet_causeError(struct packet* pkt)
{
    srand(time(NULL));
    int prob = rand() % 100;

    if(errorRate > prob)
    {
        int r = rand() % atoi((*pkt).length);
        ((unsigned char*)(*pkt).data)[r] = '1';
    }
    }
```

If the error rate is above the probability then it will change one bit of the packet. We discovered that if the number of packets being sent is large enough even the packets being resent could contain errors. This drastically increases the time the overall time.

Most of our test cases were not that large and the packet error was small (< 10%). We determined that our average increase was timeout*number of packets*packet_error_rate. This made sense from the way that we implemented our timeouts (dropping of negative Acknowledgements) and the error rate. The tests consisted of passing a 209 byte file with an MTU size of 10. We choose the small MTU so that it would force more packets to be sent with a greater likelihood that a rate of 10% would case an error in the transfer.

Important to note that we only induce errors into our data packets and not our acknowledgements. Even though we generate the CRC for the acknowledgement we ran out of time.

Things Learned

This project was a good refresher of C and remembering how socket implementation worked in C. We were also able to practice using open source functionality (Dijkistra's Algorithm) and practice with tools like git for our code base management. That are commonly used outside of the classroom.

We split the project so that we were able to independently work with one of us focusing on the end system and some of the common functionality (like CRC), and the other working on the router implementation. This allowed us to emulate a real working environment and progress at our own pace.

We found that some of the biggest hurdles were simple things like interfacing between the routers and the end systems so that they were able to talk the same language. In the beginning we both implemented a way of sending the packet stream through a char array. However, the routers than could not read the destination IP that was stored in the header of our packets.

Final thoughts

The challenges that were presented in this project made us think and forced us to strongly consider the design before implementing each item. We strongly believe that we could have finished all of the components given enough time, although we are proud of all the items that we were able to accomplish in this short time. Even though we did not know each other prior to this project we were able to quickly and efficiently get our work done as a team. This was a successful project for us and allowed us to practice a lot of skills that are necessary when working on large projects with multiple team members.