netLab: using Network Engineering to motivate Software Engineering

By

Brad Love
B.Sc., University of North Texas, 2002

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Your file Votre référence
ISBN: 978-0-494-41158-2
Our file Notre référence

netLab: using Network Engineering to motivate Software Engineering

By

Brad Love
B.Sc., University of North Texas, 2002

Supervisory Committee

ABSTRACT

This thesis describes the design and deployment of *netLab*, a self-contained lab environment suitable for use in an upper level networking course. *NetLab* does not require special hardware, special permissions, kernel modifications, or multiple computers. The laboratory was designed to emphasize hands-on programming over device configuration or performance analysis. *NetLab* uses network engineering projects to motivate software engineering principles. The main projects are linkLab and routerLab, the implementations of a layer-2 network protocol and a layer-3 routing algorithm simulation. Both projects use a physical-layer emulator providing controllable impairment for thorough testing. The lab has been shown to be capable of expansion to accommodate different protocols. *NetLab* is a success in that students consistently found netLab to be challenging and exciting, and all ranks of students advanced their skills.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

Many thanks to my supervisor, Dan Hoffman, for being involved with my work throughout my program. His advice, guidance, and financial assistance were essential to the completion of this degree. I am grateful to my parents for their unending support in all of my endeavors, without them none of this would have been possible. Thanks also goes to my ferrets for keeping me company, British Columbia for being so amazing, and my mountain bikes for keeping me sane.

1. Introduction

This thesis presents a multi-part lab designed to use networking problems to motivate Software Engineering principles. Practices such as adherence to precise specification, careful design, and thorough validation are stressed throughout `netLab`. Network Engineering offers an excellent environment for reinforcing these topics. Network protocols require the ability to interoperate with external implementations and the unreliable nature of physical mediums allows extensive testing scenarios.

The `synchLab` is a tutorial on all of the aspects of multi-threaded programming that will be encountered in the later two projects. Minimal examples are provided in every step to show the prime principles that are needed in each project. In `linkLab`, students implement a link-layer protocol in C++. The sliding window protocol Go-Back-N[1] is used currently. A sliding window allows the transmission of a "window" of multiple frames, with the idea that the window size should allow for acknowledgements to be received by the sender before the window has been filled. Interoperability is stressed through strict adherence to a specified protocol. Test suites and multiple test cases are provided for each project for students to gauge their current performance and progress. Test suites are engineered generically to allow for infinite combinations of tests. The final segment, `routerLab`, is a project whose goal is to teach router functionality. Students write code to implement most of the functions of a router. Since `Routers` are connected together when tested for correctness, distributed programming issues come into play. The routing algorithm used in `routerLab` is the Distance Vector protocol[2]. This algorithm has one main drawback that students are encouraged to solve, the problem of count to infinity[3]. Count to infinity happens because of the asynchronous manner of Distance Vector. Figure 1.1 displays the situation that can occur. Routers only keep track of the cost and first hop of destinations in their

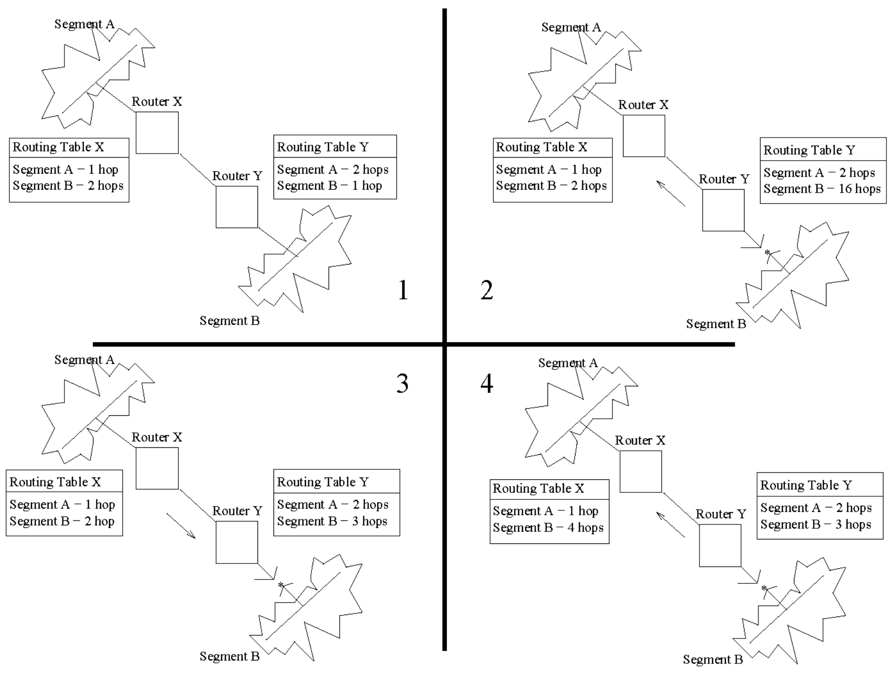


Figure 1.1: Count to Infinity

routing table; for simplicity only the destination and cost are shown. Figure 1.1 part 2 shows a portion of the network becoming unreachable. The count to infinity problem arises when the `Router` connected to the disconnected portion sees that one of its neighbors can get to the disconnected portion. Figure 1.1 part 3 shows the `Router` accepting the new cost to the unreachable section, without realizing that the path is actually through itself. With each subsequent update, the `Routers` will increment their costs to the disconnected section, until one ultimately reaches infinity. Figure 1.1 part 4 shows the first cycle. There are several approaches to handling this problem such as poison reverse, split horizon, and triggered updates. The protocol RIP[4] addresses each of these aspects, but adds more complexity than is desired for a base project. Validation of both projects is handled by extensive automated testing, code reviews, and informal proofs on sections of code.

The `netLab` lab package has been a great success each time it has been deployed. The lab has been used in UVic's csc450/550 three times in its complete state. The minimal synchronization examples helped confused students to understand topics they have seen in prior courses. LinkLab takes up the bulk of the lab, which has students implement a specific link layer protocol.

The best students found the projects challenging enough to require significant thought. The weak students still succeeded, since enough background material and small examples were provided. All ranks of students advanced their skills which can be hard to do with a single set of material. Each term, graduate students in CSC450/550 must complete an advanced project as part of the curriculum. Since the completion of `netLab`, graduate students propose an advanced project based on either `linkLab` or `routerLab`. Several interesting ideas for future development have come from graduate projects.

NetLab is delivered using generic Linux workstations. Students have normal user accounts with no root priviliges. Three-hour sessions are held once a week, staffed by a lab instructor. The labs are capped at 15 students to allow hands-on help with the material. As shown in Table 1.1, there are four main parts to netLab: synchLab, snifferLab, linkLab,and routerLab. SynchLab and snifferLab are covered in Chapters 3 and 4; linkLab is covered in more detail in Chapter 6 and routerLab is covered in more detail in Chapter 7.

Week 1	Lab introduction
Week 2	**synchLab**: Posix threads with C++
Week 3	**linkLab**: introduction
Week 4	**linkLab**: debugging with threads, logfile messages
Week 5	**linkLab**: Beta demo
Week 6	**linkLab**: drop-in
Week 7	**linkLab**: Final demo
Week 8	**snifferLab**: packet trace analysis
Week 9	**routerLab**: introduction
Week 10	**routerLab**: Beta Demo
Week 11	**routerLab**: drop-in
Week 12	**routerLab**: Final Demo

Table 1.1: netLab schedule

2. Related Work

There were several goals to be achieved when creating `netLab`, such as:

- experience designing and debugging multi-threaded programs,
- hands on experience writing network protocol code,
- interoperability of projects, and
- teaching the importance of thorough testing.

A lab was wanted that emphasized protocol specification and writing code, instead of simulations. The framework was desired to be simple enough to describe in one lab setting, yet complex enough to be able to handle a multitude of different low-level network protocols with ease. Projects were meant to be isolated; if you are writing a protocol then you should only focus on the protocol instead of simulation code, initialization, or other network layers. The lab also had to be able to run on a single Linux machine, with no special permissions. Pre-existing network topic labs seemed to have very complex frameworks, were too broad of scope, required networks of machines, emphasized configuration instead of coding, required custom Linux kernels, or stressed simulation and real world statistics. The `netLab` lab package consists of two main projects and the focus was desired to surround those projects exclusively: `linkLab` and `routerLab`.

The student implemented link-layer protocol[1] is called `linkLab`. A *link-layer* is a level 2 Open Systems Interconnection (OSI) protocol, of which there are many variants. Link-layer protocols connect two hosts together for data communication. It is meant for host-to-host transmission over a physical medium and incorporates the ability to handle unreliable networks via retransmission. The protocol chosen for implementation in `linkLab` is Go-Back-N. Other link-layer protocols that were inspected for possible use were HDLC[1] and PPP[5]. HDLC is a Level 2 protocol, which allows transmission of synchronous data over a point-to-point connection. There are several drawbacks to using HDLC though: its multiple different frame types, byte stuffing, and set window size. HDLC has distinct frame types for user data, flow and error control, and link management. The multiple frame types add complexity, which was desired to be kept manageable. Byte stuffing also makes the protocol difficult to use. Frames do not have a length field to describe how long they are; instead byte stuffing is used to pad frames with a specific pattern to indicate the beginning or end of a frame. Byte stuffing means that students would have to constantly monitor the link and recognize bit patterns in the stream. HDLC uses a three bit field in data frames to handle sequence numbers and the sliding window size is directly tied to it. There are seven possible sequence numbers in the range so up to seven frames can be transmitted at once. The complex frame types and limited configurability discounted HDLC as a possible candidate protocol. PPP is an advancemnt of the HDLC protocol, which allows more configurability and the ability to piggyback other protocols. The main difference between HDLC and PPP is that PPP is character oriented, whereas HDLC is bit-oriented. Having the protocol character oriented makes it easier to be implemented as students do not have to continuously monitor a bit buffer, but can instead monitor for a special character. The special character dictates that the current frame has been received in full. This means that PPP frames are always an even amount of bytes, unlike HDLC. Like HDLC, PPP uses multiple subprotocols to handle data, link control, and network control. The link control protocol handles bringing connections up, configuring options, and tearing them down upon completion. Once the link control protocol has set the connection up, the network control protocol configures the network layer options, dependant on the network layer selected. Separate network control protocols are required for each type of network layer that is desired to be used. Since for our testing the link control and network control protocol options would be set at compilation time PPP was deeemd too complex. Adding a link control protocol to Go-Back-N would be a good extension to the project, but this has been left for future consideration.

The `routerLab` project is a student implemented router. The Distance Vector protocol[1] was chosen as the routing protocol to be implemented for the project. Distance Vector works by having each router monitor connections to its neighbors. If a topology change is detected, the router sends updates only to its neighbors, which can lead to a delay in topology changes percolating to the rest of the network. There are multiple virtual router environments but none that were suitable for inclusion in `netLab`. The goal of `netLab` was tight integration of both of the projects, as `linkLabs` intent is to connect student `Routers` together. The virtual router environment should be flexible and abstract without requiring complex configuration or multiple machines. The virtual router environments inspected were: the Stanford Virtual Router[6], the Click Modular Router[7], and VELNET[8]. The Stanford Virtual Router is similar to `routerLab`, although it has been designed to operate on IP packets connected to a physical network. The Stanford Virtual Router requires a server and isolated network, meaning that it cannot be deployed on standard workstations with no modifications. The Click modular Router[7] is another virtual router environment, similar to the Stanford Virtual Router, but requires custom Linux kernel modules running on an isolated network. Since `netLab` needed to run on stock Linux machines this was unacceptable. The University of Western Sydney's VELNET[8] is a self-contained virtual network laboratory, which allows implementation of network protocols and router configuration. VELNET handles its self-containment by placing its multiple hosts in VMWARE[9] virtual servers, and the lab environment is based in windows, which is not useful for CSC450/550. VELNET also places most emphasis on configuration instead of protocol coding.

While there is a multitude of networking lab environments and projects used by universities today, there was nothing self-contained that met the intended goals. The lab should be self-sufficient and not require any special permissions for user accounts. There are several labs designed along similar ideas as `netLab`. These labs attempt to achieve different goals though or are either too broad or narrow of focus to be of use to what was desired. The labs inspected were NIST NET[10], University of Girona's Virtual Laboratory for Learning IP Networking[11], and TinkerNet[12]. NIST NET has useful parts including impairment and full network emulation. All of the emulation and impairment is handled by a custom Linux kernel, which detracts from its usefulness in `netLab`. NIST NET is able to simulate an entire network, but this is geared at testing of protocols on machines connected through the NIST NET machine. Emphasis is on performance statistics and accurate emulation over core protocols, as protocols are unimportant to NIST NET. Since we desired to be able to have `netLab` able to run on any single Linux machine, NIST NET's framework was not suitable as a possible testing framework. The University of Girona's Virtual Laboratory for Learning IP Networks contains several good ideas. They allow students to configure virtual network topologies, choose between a possible combination of IPv4 and IPv6 networks and utilize Linux network commands on the topologies. The lab does not require modifications to the Linux kernel, but does require an isolated multi-computer network. In addition the core of the lab emphasizes configuration of networks and actual Linux networking commands to handle those networks. Underlying protocols can be selected, but there is minimal focus on protocol specification and testing. There are no projects involving coding of protocols or modification of internal components either. Most of the focus is on handling IP networks and the problems that can arise in configuring them. Since the main focus of `netLab` is hands on programming experience with protocols, and not configuring them, the lab was not useful. Harvey Mudd College's TinkerNet comes close to what was desired of `netLab`, but does not cover as wide of scope and was developed after `netLab` was designed. TinkerNet focuses on real Ethernet packets and all the OSI[13] layers from link-layer to application layer. TinkerNet has students implement a range of related projects, with each built on the previous one. Although there is hands on coding experience, the projects are designed to interface with standard protocols on a live network. Students initially implement functions to send and receive ARP, IP, and UDP frames[1] on the wire. The final project has students implement a microprotocol which exclusively deals with fragmenting and reassembling messages. The TinkerNet framework does not have the testing capabilities required for `netLab`, as it is designed to interface with an Ethernet network. Our emphasis is intended to have students program full protocols, in order to give them experience with all the subtleties involved. The broad focus of TinkerNet, which means more smaller projects, its requirement of an external network, and no testing suite puts it outside of the scope that was desired for `netLab`.

3. Threading using synchLab

The topic of synchronization is usually covered in operating systems courses. Students study basic concepts and the inherent problems with race conditions and concurrent access. The aim of synchLab is to present threading concretely using the POSIX [14] thread library. Small examples are utilized for maximum impact. The focus is understanding that multi-threaded programs behave differently from one execution to the next, and tactics for writing reliable code despite the differences.

SynchLab is based on six C++ programs:

- `time.cpp` : introduces the `struct timeval` and the accuracy of the `usleep` system call.
- `createThread.cpp` : details POSIX thread creation and management.
- `interleave.cpp` : displays the arbitrary execution order of statements in multi-threaded applications.
- `shared.cpp` : how unprotected memory shared between multiple threads can be corrupted.
- `mutex.cpp` : accessing and protecting shared memory between multiple threads in a C++ application.
- `classes.cpp` : techniques to utilize threads with C++ classes.

The examples work together to prepare students for practical work with multi-threaded code.

3.1 Evaluation of Machine Timing Accuracy

Some students have never been pressed to have their programs running with fine grained accuracy. This initial exercise measures the execution time of one million iterations through an empty for loop. Timing is handled in `time.cpp` using the `struct timeval` and the `usleep` std library call. A `timeval` contains fields for seconds and microseconds. The `usleep` method delays execution of subsequent instructions in the calling thread for a specified number of microseconds. A library of overloaded timeval operators is supplied by the instructors. The `timevalOperators.h` macros enables students to perform standard arithmetic operations on timevals during execution. A supplementary quiz is given which has students modify `time.cpp` to time the accuracy of the `usleep` std library call. To accomplish this students must replace the empty `for` loop of `time.cpp`, shown in Figure 3.1, with a `usleep(N)` call. The program is then adjusted to accept an unsigned integer on the command line to pass to the `usleep` method. Students are required to find the minimum number of microseconds at which usleep is fairly accurate on average. This is important because `usleep` is not always precise for small values, and students should see firsthand `usleep` is not totally accurate in timing situations.

3.2 Creating Threads

Since many students have never created a POSIX thread, a concise example is provided to explain the topic. The example, `createThread.cpp`, shows how to start a thread, pass it a parameter, and introduces the thread life cycle. The thread, T, is passed an integer N and prints the integers $0, 1, \ldots, N-1$. When the main method terminates, however, T is forced to terminate. As a result, the output is only a prefix of $0, 1, \ldots, N-1$. The length of the prefix varies across executions.

3.3 Statement Interleaving

The `interleave.cpp` example focuses on two questions:

1. What predictions can we make about the execution order of the statements within a single thread?

2. What predictions can we make about the interleaving of the statements in two threads?

Students know the answer to the first question: the order is controlled by the conditional expressions in the if-then-else and loop constructs in the program. They are less sure about the second question. This program shows that the interleavings vary from one execution to the next and that no predictions can be made about the interleavings.

The implementation of `interleave.cpp`, shown in Figure 3.3, is simple. No attempt is made to illustrate typical use; instead the interleaving properties of threads are laid bare with "minimal examples" [15], which focus on a single issue

```
int main(int argc, char* argv[])
{
    timeval t0,t1;

    // retrieve and print the current time
    gettimeofday(&t0, NULL);
    cout << "Current time: " << t0 << endl;

    // to loop one million times and print the elapsed time
    gettimeofday(&t0, NULL);
    for (int i = 0; i < 1000000; i++)
        ; // intentionally empty
    gettimeofday(&t1, NULL);
    cout << "Elapsed time: " << (t1-t0) << endl;

    return 0;
}
```

Figure 3.1: `time.cpp`

```
void* threadFunction(void* ptr)
{
    int *n = (int*)ptr;
    for (int j = 0; j < *n; j++) {
        cout << j << endl;
    }
}

int main(int argc,char* argv[])
{
    pthread_t threadStruct;

    unsigned int n = atoi(argv[1]);

    int r = pthread_create(&threadStruct,NULL,&threadFunction,(void*)&n);
    cout << "return code: " << r << endl;

    usleep(1000); // pause so that the thread gets some time

    return 0;
}
```

Figure 3.2: `createThread.cpp`

using the least statements possible. The `compute` function busy-waits for a random amount of time, to introduce variable delay. The function `threadFunctionA` prints $0, 1, \ldots, 9$, left justified, calling `compute` each iteration; `threadFunctionB` is identical, except that the output is indented one tab stop. The main method creates threads A and B, and then waits for each thread to terminate before exiting. Table 3.1 shows the output from running interleave twice. The serialized output is rare; it is also rare to see two consecutive runs produce the same output.

3.4 Shared Data

Protecting shared data can be key to success in multi-threaded programming. In `shared.cpp`, two threads are started which perform identical actions. Each one repeatedly fetches, increments, and then stores a shared global integer. Most students assume that, with two threads iterating through the operations described

```
void compute() // pause for a random amount of time
{
    long var = rand() % 15000000;
    for (int i = 0; i <= var; i++)
        ; // empty
}

void* threadFunctionA(void* ptr) // print 0..9, left justified
{
    for (int j = 0; j < 10; j++) {
        compute();
        cout << j << endl;
    }
}

void* threadFunctionB(void* ptr) // print 0..9, indented
{
    for (int j = 0; j < 10; j++) {
        compute();
        cout << '\t' << j << endl;
    }
}

int main(int argc,char* argv[])
{
    // create random seed, using current microseconds
    timeval seed;
    gettimeofday(&seed, NULL);
    srand(seed.tv_usec);

    // start two threads
    cout << "A\tB" << endl;
    pthread_t threadA;
    pthread_create(&threadA,NULL,&threadFunctionA,NULL);
    pthread_t threadB;
    pthread_create(&threadB,NULL,&threadFunctionB,NULL);

    pthread_join(threadA,NULL);
    pthread_join(threadB,NULL);

    return 0;
}
```

Figure 3.3: `interleave.cpp`

N times, the final value for the shared data will be $2N$. Due to the random nature of interleaving, however, this is rarely the case. Executions produce a variety of values in $[N..2N]$.

The `shared.cpp` code, shown in Figure 3.4, is similar to `interleave.cpp`. The difference is that, in the former, only the final result of the interleaving is presented, as a single integer. Table 3.2 shows two interleavings. The first interleaving illustrates "safe sharing"; the second is one of many others which can occur in `shared.cpp`.

A serialized interleaving		**Another possible interleaving**	
thread A	thread B	thread A	thread B
0		0	
1		1	
2		2	
3			0
4			1
5		3	
6		4	
7		5	
8			2
9			3
	0		4
	1		5
	2		6
	3		7
	4		8
	5	6	
	6	7	
	7	8	
	8	9	
	9		9

Table 3.1: `interleave.cpp`: output

```
int totalCount = 0;

// pause for a random amount of time
void compute()
{
    long var = rand() % 1500000;
    for (int i = 0; i < var; i++)
        ; // intentionally empty
}

// increment totalCount 10 times
void* incrementer(void* ptr) {
    for (int i = 0; i < 10; i++) {
        int a = totalCount;
        compute();
        totalCount = ++a;
    }
}

int main() {
    // create random seed, using current microseconds
    timeval seed;
    gettimeofday(&seed,NULL);
    srand(seed.tv_usec);

    pthread_t thread0;
    pthread_create(&thread0,NULL,&incrementer,(void*)NULL);
    pthread_t thread1;
    pthread_create(&thread1,NULL,&incrementer,(void*)NULL);

    pthread_join(thread0,NULL);
    pthread_join(thread1,NULL);

    cout << "Total count: " << totalCount << endl;
}
```

Figure 3.4: `shared.cpp`

A correct interleaving		An incorrect interleaving	
thread A	thread B	thread A	thread B
`a=totalCount;`		`a=totalCount;`	
`compute();`			`a=totalCount;`
`totalCount=++a;`		`compute();`	
	`int a=totalCount;`	`totalCount=++a;`	
	`compute();`		`compute();`
	`totalCount=++a;`		`totalCount=++a;`

Table 3.2: `share.cpp`: interleavings

3.5 Controlling Access to Shared Data

The previous example, `shared.cpp`, demonstrates the importance of controlling access to shared data, now controlling the interleavings is addressed. In the `mutex.cpp` code, a semaphore is introduced, along with lock/unlock pairs in each of the two threads of `shared.cpp`. The semaphores ensure atomic access is enforced, therefore the final value is always $2N$.

3.6 Accessing Private Instance Data

The `classes.cpp` example, shown in Figure 3.5, displays how to access private instance data from a static thread context and how to number object instances. This is an object oriented spin on a previous example, `interleave.cpp`. A class is provided that has a single static method which prints values [1..10]. The class is instantiated twice, a static constructor variable is used to determine instance number, and `this` is passed as a parameter to a thread constructor. With `this*` preserved inside the static thread, each instance prints its values [1..10] in a separate column.

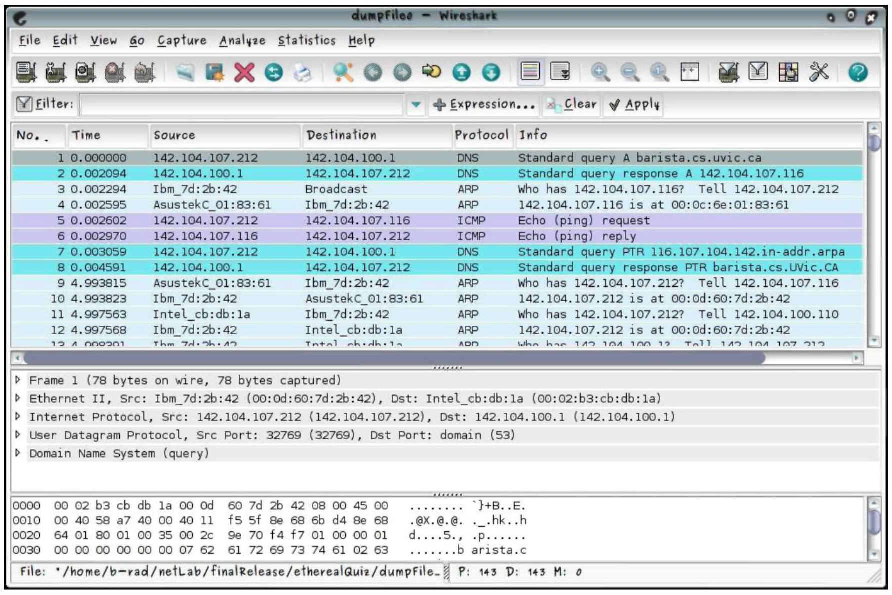
```
class C {
public:
int threadId;
pthread_t thread;
C()
{
    static int threadCount = 0; // total number of C instances
    threadId = threadCount++; // id of this instance for messages
    pthread_create(&thread,NULL,&threadMethod,(void*)this);
}

static void* threadMethod(void* ptr)
{
    C* self = (C*)ptr;
    for (int j = 0; j < 10; j++) {
        compute();
        if (self->threadId == 0)
            cout << j << endl;
        else
            cout << '\t' << j << endl;
    }
}
};

int main(int argc,char* argv[])
{
    // create random seed, using current microseconds
    timeval seed;
    gettimeofday(&seed, NULL);
    srand(seed.tv_usec);
    // start
    cout << "A\tB" << endl;
    C c0;
    C c1;

    pthread_join(c0.thread,NULL);
    pthread_join(c1.thread,NULL);

    return 0;
}
```

Figure 3.5: `classes.cpp`

4. Packet Sniffing using snifferLab

Students are introduced to the WireShark (formerly Ethereal) packet "sniffer" during lectures. WireShark is a program that monitors data traveling over a network. This lab gives hands-on experience using a sniffer to solve a number of problems involving the Internet protocols they have seen in lecture. WireShark's core functionality is briefly reviewed to remind students how to utilize the programs functionality. A small list of related program tips is provided to aid completion of the lab without having to search broadly. The students are given a capture file containing approximately 3000 frames with an assortment of traffic such as ICMP, ARP, FTP, P2P, ssh, and HTTPS. A capture file is a recording of traffic that is received over a network device. Usually to use WireShark to record a live network students would require root permission. However, since a capture file is supplied, root permission is not required.

Figure 4.1 shows one screen of frames. Using the capture students must



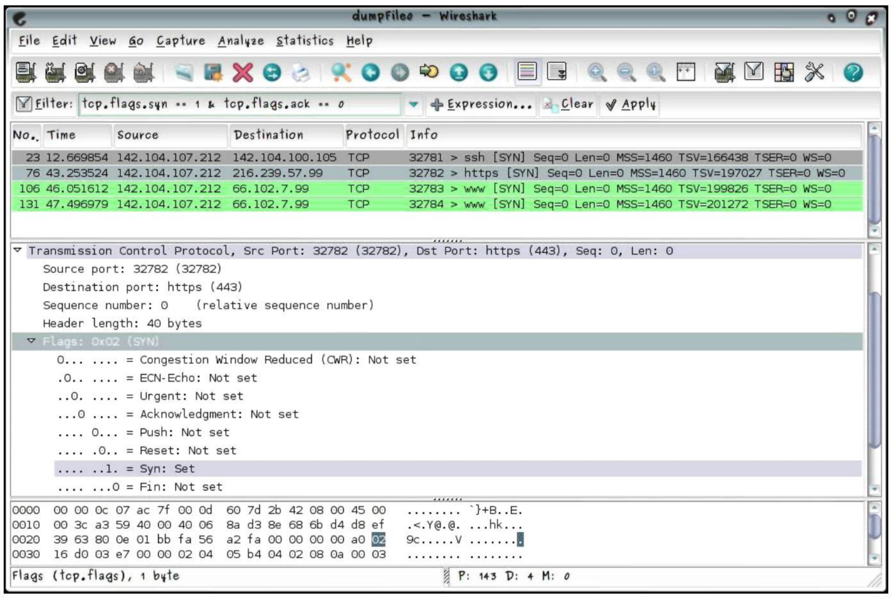
```
No. .   Time      Source              Destination         Protocol  Info
      1 0.000000  142.104.107.212     142.104.100.1       DNS       Standard query A barista.cs.uvic.ca
      2 0.002094  142.104.100.1       142.104.107.212     DNS       Standard query response A 142.104.107.116
      3 0.002294  Ibm_7d:2b:42        Broadcast           ARP       Who has 142.104.107.116?  Tell 142.104.107.212
      4 0.002595  AsustekC_01:83:61   Ibm_7d:2b:42        ARP       142.104.107.116 is at 00:0c:6e:01:83:61
      5 0.002602  142.104.107.212     142.104.107.116     ICMP      Echo (ping) request
      6 0.002970  142.104.107.116     142.104.107.212     ICMP      Echo (ping) reply
      7 0.003059  142.104.107.212     142.104.100.1       DNS       Standard query PTR 116.107.104.142.in-addr.arpa
      8 0.004591  142.104.100.1       142.104.107.212     DNS       Standard query response PTR barista.cs.UVic.CA
      9 4.993815  AsustekC_01:83:61   Ibm_7d:2b:42        ARP       Who has 142.104.107.212?  Tell 142.104.107.116
     10 4.993823  Ibm_7d:2b:42        AsustekC_01:83:61   ARP       142.104.107.212 is at 00:0d:60:7d:2b:42
     11 4.997563  Intel_cb:db:1a      Ibm_7d:2b:42        ARP       Who has 142.104.107.212?  Tell 142.104.100.110
     12 4.997568  Ibm_7d:2b:42        Intel_cb:db:1a      ARP       142.104.107.212 is at 00:0d:60:7d:2b:42

▷ Frame 1 (78 bytes on wire, 78 bytes captured)
▷ Ethernet II, Src: Ibm_7d:2b:42 (00:0d:60:7d:2b:42), Dst: Intel_cb:db:1a (00:02:b3:cb:db:1a)
▷ Internet Protocol, Src: 142.104.107.212 (142.104.107.212), Dst: 142.104.100.1 (142.104.100.1)
▷ User Datagram Protocol, Src Port: 32769 (32769), Dst Port: domain (53)
▷ Domain Name System (query)

0000  00 02 b3 cb db 1a 00 0d  60 7d 2b 42 08 00 45 00   ........ `}+B..E.
0010  00 40 58 a7 40 00 40 11  f5 5f 8e 68 6b d4 8e 68   .@X.@.@. ._.hk..h
0020  64 01 80 01 00 35 00 2c  9e 70 f4 f7 01 00 00 01   d....5., .p......
0030  00 00 00 00 00 00 07 62  61 72 69 73 74 61 02 63   .......b arista.c

File: "/home/b-rad/netLab/finalRelease/etherealQuiz/dumpFile_   P: 143 D: 143 M: 0
```

Figure 4.1: WireShark raw capture file

answer a series of questions. The large number of frames in the capture file prevents manual review to finish the excercise. This allows completion of the lab on any standard machine with a sniffer. Every question can be answered using either options within WireShark or by utilizing the filter function. The filter function allows control over what is displayed to the user. Applied filters can be as broad as an entire network protocol or as narrow as a single bit set in a TCP header field.
The quiz contains questions such as:

- What Network Layer protocols are present?
- What Transport Layer protocols are present?
- What well-known TCP or UDP ports[1] are present?
- How many TCP connections are present?
- For each TCP connection, what is the client initial sequence number?
- How many frames are used in one of the prior connections terminations?
- Is there a relationship between initial sequence number and transmission time?

Figure 4.2 displays how to determine the number TCP connections are present. To answer the question we filter to look only for TCP frames with the SYN bit set and the ACK bit unset, since this is the initial frame in the TCP handshake.


dumpFileo – Wireshark
File Edit View Go Capture Analyze Statistics Help
Expression... Clear Apply

Filter: tcp.flags.syn == 1 & tcp.flags.ack == 0

```
No. Time      Source          Destination     Protocol Info
 23 12.669854 142.104.107.212 142.104.100.105 TCP      32781 > ssh [SYN] Seq=0 Len=0 MSS=1460 TSV=166438 TSER=0 WS=0
 76 43.253524 142.104.107.212 216.239.57.99   TCP      32782 > https [SYN] Seq=0 Len=0 MSS=1460 TSV=197027 TSER=0 WS=0
106 46.051612 142.104.107.212 66.102.7.99     TCP      32783 > www [SYN] Seq=0 Len=0 MSS=1460 TSV=199826 TSER=0 WS=0
131 47.496979 142.104.107.212 66.102.7.99     TCP      32784 > www [SYN] Seq=0 Len=0 MSS=1460 TSV=201272 TSER=0 WS=0

▽ Transmission Control Protocol, Src Port: 32782 (32782), Dst Port: https (443), Seq: 0, Len: 0
    Source port: 32782 (32782)
    Destination port: https (443)
    Sequence number: 0    (relative sequence number)
    Header length: 40 bytes
  ▽ Flags: 0x02 (SYN)
      0... .... = Congestion Window Reduced (CWR): Not set
      .0.. .... = ECN-Echo: Not set
      ..0. .... = Urgent: Not set
      ...0 .... = Acknowledgment: Not set
      .... 0... = Push: Not set
      .... .0.. = Reset: Not set
      .... ..1. = Syn: Set
      .... ...0 = Fin: Not set

0000  00 00 0c 07 ac 7f 00 0d  60 7d 2b 42 08 00 45 00   ........ `}+B..E.
0010  00 3c a3 59 40 00 40 06  8a d3 8e 68 6b d4 d8 ef   .<.Y@.@. ...hk...
0020  39 63 80 0e 01 bb fa 56  a2 fa 00 00 00 00 a0 02   9c.....V ........
0030  16 d0 03 e7 00 00 02 04  05 b4 04 02 08 0a 00 03   ........ ........

Flags (tcp.flags), 1 byte                P: 143 D: 4 M: 0
```

Figure 4.2: WireShark Filtered view

5. The Emulated Physical Layer: PhysicalLayer

In a real network, the Physical Layer provides an unreliable service. The service delivers data packets over a network from one host to another. Figure 4.1 shows the common scenario of data travelling over a network link. Although most packets are successfully transferred from host A to host B, some are lost in transmission. The Physical Layer does not detect transmission errors; it merely transmits and receives bytes.

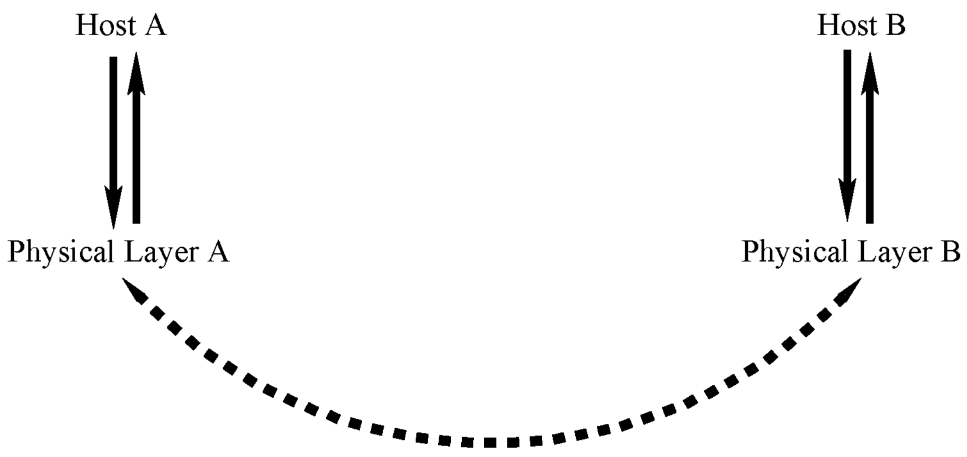


Figure 5.1: Bidirectional Physical Layer

5.1 Requirements and motivation

The CSC450 projects deal with high level network concepts. In labs, root access must not be given for safety reasons; therefore any dealings with low-level networking protocols must be emulated. Two C++ classes were designed to handle the issue, the `PhysicalLayer` and the `NetQueue`. The `PhysicalLayer` was created to allow simulated use of a network without any of the security and permission concerns. The `NetQueue` was created to control everything transmitted through a `PhysicalLayer`.

To handle the unreliable nature of Physical Layers, three types of impairment are used: kill, corrupt, and delay. These three impairments allow most problems inherent with networks to be controlled and manipulated in a precise way. The course projects which use the `PhysicalLayer` are tested thoroughly, so the precision helps to discount uncertainty in the results. Three types of impairment are available to the `NetQueue`: `kill`, `corrupt`, and `delay`. Each endpoint has independent impairment settings for transmission.

5.2 NQ/PL architecture

A full communication service between two hosts is provided using two `PhysicalLayer` instances connected to a `NetQueue`. A `NetQueue` maintains two separate queues, A and B, one for each direction. When a `PhysicalLayer` is instantiated, it is connected with either the A or B side of a `NetQueue`. Figure 5.2 shows two Physical Layers with each inserting into one queue and retrieving from the other queue.

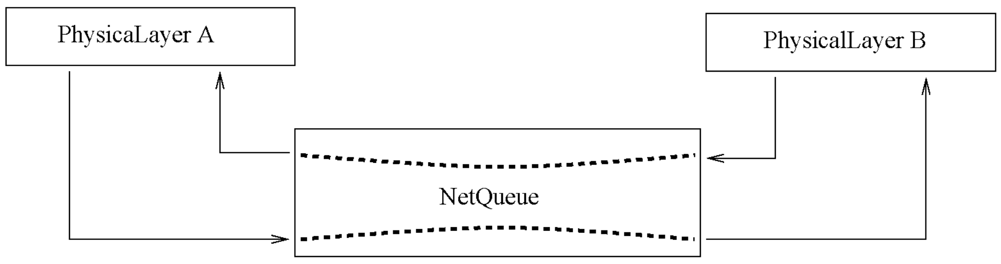


Figure 5.2: PhysicalLayer NetQueue interaction

The following files are supplied:

- `NetQueue.cpp`: unreliable bi-directional communication service which provides two distinct endpoints.
- `NetQueue.h`: method prototypes for NetQueue.
- `PhysicalLayer.cpp`: single sided access to one endpoint of a NetQueue.

- `PhysicalLayer.h`: method prototypes for PhysicalLayer.
- `PLTester.cpp`: minimal functionality tester for Physical Layer / NetQueue connectivity.

5.3 NetQueue interface

The `NetQueue` class is the heart of the bi-directional communication service provided. `NetQueue` manages each direction independently, handles impairment and uniform delay. Figure 5.3 shows the API for the `NetQueue`. To conserve space the figure only shows the method calls for side A, the side B method calls are identical. In each `NetQueue` object, `maxsize` specifies the maximum number of transmissions that can be queued in each direction. If `maxsize` is reached during a `NetQueue`'s lifetime, an `Exception` is thrown. The method `maxSize()` returns the value set in the constructor. Optionally, a microsecond value for uniform delay may be supplied to the `NetQueue` constructor. If the uniform delay is set, each transmission in each direction will have the same end-to-end delay.

A `NetQueue` instantiated, with or without uniform delay, provides first-in-first-out queuing behaviour. Each direction has its own queue to receive from, and transmits to the other direction's queue. A standard FIFO queue has limited functionality and is not used for the implementation in order to handle optional impairment, described later. Instead a minheap is employed, with a release time used as the radix for the ordering.

Each side has an `add` method which inserts the passed data into the opposite direction's minheap. Before a transmission is added to the minheap, a timestamp of the current system time is attached to it. The attached timestamp is used as the release time to determine a transmission's availability. It is the caller's duty to know if the `add` call is legal; the `sizeXtoY` method allows this check without throwing an `Exception`.

When the `current` method is called, the top of the associated minheap's

```
class NetQueue {
 private:
  MinHeap *AtoB, *BtoA;
  struct timeval delay;
  unsigned int frameCountA, frameCountB, maxsize;
  int killNumA, corruptNumA, delayNumA, killNumB, corruptNumB, delayNumB;
  double *killsA, *corruptA, *killsB, *corruptB;
  timeval *delayA, *delayB;

  /** Create an empty Link with maximum size maxsize and millisecond delay.
   *  If maxsize < 1 or maxsize > 100 or delay < 0 then throw Exception.*/
 public:
  NetQueue(unsigned int maxsize, long delay);

  /** Create an empty Link with maximum size maxSize and delay 0.
   *  If maxSize < 1 or maxSize > 100 or delay < 0 then throw Exception.*/
  NetQueue(unsigned int maxSize);

  /** Return the maximum number of entries permitted in either
   *  the A to B portion or the B to A portion of the NetQueue. */
  unsigned int maxSize();

  /** Impair A side * kill, corrupt, and delay masks */
  void setImpairA(double* kills, int killNum, double* corrupt,
                  int corruptNum, long* delay, int delayNum);

  /** Add a copy of buf to the A side.
   * If sizeAtoB() == maxSize() then throw Exception. */
  void addA(unsigned char* buf, unsigned int bufLength);

  /** Retrieve the next buffer from the A side, without removing it.
   * If sizeBtoA() == 0 then throw Exception */
  unsigned int currentA(unsigned char* buf);

  /** Remove a buffer from the A side.
   * If sizeBtoA() == 0 then throw Exception. */
  void removeA();

  /** Determine whether it is legal to remove a buffer from the A side.*/
  bool removeLegalA();

  /** Return number of elements in the A to B portion of the NetQueue.*/
  unsigned int sizeAtoB();
};
```

Figure 5.3: NetQueue specification

release time is checked. If the release time is earlier than the current time, the data is copied into the passed buffer and the length in bytes is returned. The buffer passed to the `current` method must be pre-allocated. It is the caller's duty to know if the `current` and `removeCurrent` calls are legal; the `removeLegal` methods allow this check without throwing an exception.

For controlling impairment, there are three different types for each direction. The constructor is used to specify impairment at instantiation time. The `setImpairment` methods are used to override the current impairment, if any, of a particular side. Both methods accept three arrays; `kill`, `corrupt`, and `delay`. used as an index, modulo the array length, into each array. The `kill` array contains floating point values in [0..1]. For a `kill` array of length N, frame i will be killed if $R <$ `kill`$[i \bmod N]$, where R is randomly chosen in [0..1]. The `corrupt` array is the same as the `kill` array, indicating the probability the ith frame is corrupted. The `delay` array contains integers which specify the number of microseconds to delay the ith frame. Each array can be of any size. A counter is used in each direction which maintains the total number of transmissions which have occurred and is Kill is the first impairment checked; if a frame is to be killed it is discarded and there is no reason to corrupt or modify its release time. If a frame is to be corrupted, a byte in the data buffer is incremented. A delayed frame has the specified delay added to its release time, so that it is not immediately available to the other side.

5.4 PhysicalLayer interface

The `PhysicalLayer` is a wrapper around the `NetQueue` class. While the `NetQueue` provides a queueing service for bi-directional communication, the `PhysicalLayer` provides access to a single side of the communication. Figure 5.4 shows the `PhysicalLayer` API. A `PhysicalLayer` is created by passing it an instance of a `NetQueue` and a boolean value indicating whether it is the A side or B side.

```
/** Provide access to one endpoint of a Link object. */
class PhysicalLayer
{
   private:
      NetQueue* link;
      bool aSide;
      pthread_mutex_t mutexAtoB;
      pthread_mutex_t mutexBtoA;
/** Create a PhysicalLayer. */
   public:
      PhysicalLayer(NetQueue* link0,bool aSide0);

/** Add a buffer. if send sucessfull : true */
      bool send(unsigned char* buf, unsigned int bufLength);

/** Return the next buffer. */
      unsigned int receive(unsigned char* buf);
};
```

Figure 5.4: PhysicalLayer specification

Since the communication handled by a `NetQueue` is usually multi-threaded, two mutexes are used to protect the shared structure between threads. There are only two public methods in the `PhysicalLayer`: `send` and `receive`. The `Send` method accepts an unsigned char array and its length in bytes, then depending on if the instance is side A or side B the proper `NetQueue` method is called and passed the data. The `Receive` method checks which side the instance is and calls the associated `NetQueue` method. If anything is available, it is copied into the address represented by the pointer passed to it and the length in bytes is returned. The `receive` method requires that the buffer passed to it has been pre-allocated.

5.5 Testing the PhysicalLayer

A tester is included that displays the result of communication using `PhysicalLayer` and `NetQueue`. Figure 5.5 shows the `PhysicalLayer` tester. The tester shown will send three transmissions in each direction and attempt to receive them. Side A of the `NetQueue` is set to delay the first by 1000 microseconds,

```
int main(int argc, char* argc[])
{
   NetQueue nq(3);
   PhysicalLayer plA(&nq,true);
   PhysicalLayer plB(&nq,false);
   unsigned int sendCount  = 3, recvCountA = 0, recvCountB = 0, data = 0;
   double killA[] = {0.0,0.0,1.0};
   double corruptA[] = {0,1.0,0};
   long delayA[] = {1000,0,0};
   nq.setImpairA(killA,sendCount,corruptA,sendCount,delayA,sendCount);

   for (unsigned int i = 0; i < sendCount; i++, data++) {
      plA.send((unsigned char*)&data, sizeof(unsigned int));
      plB.send((unsigned char*)&data, sizeof(unsigned int));
   }
   while (recvCountA < sendCount || recvCountB < sendCount) {
      if (plB.receive((unsigned char*)&data) > 0) {
         cout << '\t' << data << endl;
         recvCountB++;
      }
      if (plA.receive((unsigned char*)&data) > 0) {
         cout << data << endl;
         recvCountA++;
      }
   }
}
```

Figure 5.5: Physical Layer Tester

corrupt the second, and kill the third. Side B is set to have no impairment. Once the impairment is set, the three transmissions are sent and then a while loop is entered until all the frames have been received. Figure 5.6 show the output from the described setup. Line numbers have been added to the output on the left.

```
1.              16777217
2.         0
3.         1
4.         2
5.              0
```

Figure 5.6: Physical Layer Tester Output

Lines 2, 3, and 4 show what side A receives. The values 0, 1, and 2 were originally transmitted. Side B, on lines 1 and 5, shows a different output from side A. The first value side B receives is 16,777,217. The value is not in the range of numbers that were transmitted so this must be the corrupted transmission. The second transmission received was the first value, the delay caused it to be misordered. The program must now be terminated manually, since the last frame was killed and will never be received.

6. Implementing a Link Layer Protocol: linkLab

In `linkLab`, students implement a link layer protocol as a C++ class, as shown in Figure 6.1. The implementation is based on a sliding-window protocol[1] which uses pipe-lining for improved performance. The sender keeps a “window” of frames: those transmitted but not yet acknowledged. Until it is acknowledged, each transmitted frame must be stored in a “send buffer.” There is a fixed maximum for the window size, to limit the buffer space required. If the window is full, no additional frames are buffered until the window size decreases. Frames are identified using a sequence number that is incremented with each subsequent frame. When a acknowledgement frame is received, all frames with a previous sequence number are removed from the window.

6.1 Link Layer API

Students are given a skeleton class containing three empty methods: the `LinkLayer` constructor, `send`, `receive`, and an `PhysicalLayer` implementation.

The constructor’s task is to initialize the core components of the protocol:

- initialize the protocol parameters,
- create a window of proper size,
- initialize inbound and outbound sequence numbers to zero, and
- start a thread to handle physical layer activity.

The Network Layer passes messages to the `LinkLayer send` method. If the send buffer is full, the message must be rejected; every transmitted frame must be buffered so that a resend is possible. Otherwise, a frame is constructed to hold the data. A frame consists of a sequence number, an acknowledgement number, a checksum and a Network Layer message. In the send buffer, a retransmit time

```
/** Provides the send side of a simplex go-back-N Link Layer service.
 * If the sequence of byte arrays b1, b2, ..., bN is passed to
 * LinkLayerSend.send then N calls to LinkLayerReceive.receive will
 * return exactly the same sequence of byte arrays.
 */

/* Create a LinkLayer object.
 * Use sequence numbers 0, 1, ..., maxSeqNum.
 * Use a sender's window size with maximum size maxWinSize.
 * For each frame, if an acknowledgement has not been received
 * successfull within timeout microseconds, retransmit the frame.
 * Repeat this timeout/resend cycle until the frame has been acknowledged.
 * If !(0 < maxWinSize <= maxSeqNum) then throw LL_Exception
 */
LinkLayer(unsigned int maxSeqNum, unsigned int maxWinSize,
          unsigned int timeout, PhysicalLayer* physicalLayer);

/* Send buf to the other side.
 * If there is room in the send buffer then
 *    store buf in the send buffer
 *    send buf to the other side
 *    return true, as soon as buf is stored locally
 * else
 *      do not send or store buf
 *      return false
 * If bufLength > MAX_BUF_LENGTH then throw LL_Exception.
 */
bool send(unsigned char* buf,unsigned int bufLength);

/* Retrieve a buffer.
 * If any data is available to be retrieved
 *    copy the data into buf
 *    return the length of the data
 * else
 *    return 0
 */
unsigned int receive(unsigned char* buf);
```

Figure 6.1: LinkLayer specification

is stored with each frame. The `receive` method simply checks to see if there is a message waiting in the receive buffer. If so, the receive buffer is cleared and the message is copied into supplied byte buffer. The `LinkLayer` silently drops any messages received while the receive buffer is full.

Most of the `LinkLayer` complexity is in the thread processing, which loops forever executing the following three tasks:

1. Read the next frame F from the Physical Layer. Inspect the checksum and sequence number. If correct, proceed to task 2, else proceed to task 3.
2. Copy the data portion of F to the receive buffer.
3. Extract the acknowledgement number A from F. Iterate through the frames residing in the sliding-window. Delete all frames which have a sequence number prior to A. Resend all frames which have timed out.

Following is a description of all files supplied:

- `AbstractLinkLayer.h`: Contains the public API for `AbstractLinkLayer` and the API's for ancillary classes:
  - `Frame`: class used in all link layer transmissions. `Frame` contains the following fields: sequence number, acknowledgement number, checksum, data length, and the data segment.
  - `TimedFrame`: a sub-classed `Frame` used by the Link Layer protocol to determine when a transmitted `Frame` has timed out. One additional field is provided to record the transmission time for the `Frame`.
  - `SendBuffer`: a size-restricted `TimedFrame` container, which allows adding to the end and removing from any position. The `SendBuffer` has a linked list as its back end and pre-allocates all memory that will be used at construction time.
  - `ReceiveBuffer`: a storage container for a single buffer.

- `Exception.h`: Used to signal exceptions. The assignment specification describes the circumstances which require an exception to be thrown.
- `LinkLayer.cpp`: Contains signatures for the methods that must be implemented. The bodies to the methods are empty except for a `return` statement.
- `LinkLayer.h`: Method prototypes for `LinkLayer`.
- `tester.cpp`: Sends a specified number of frames between two link layers, with user-defined impairment.

6.2 Instructor Solution

The instructor solution has been fined tuned and is able to pass all tests we have designed 100 percent of the time. Many revisions were made to the solution to achieve correctness. Several different concepts have to be dealt with such as: the interval over which the sender's window is sitting, complexity processing acknowledged transmissions, access to shared data structures between threads, and keeping code as concise and non-redundant as possible.

A sliding window protocol allows a varying amount of seperate data to be transmitted at a single time. The send window is represented as a contiguous range of numbers inside the sequence number range. The size of the send window must be less than or equal to the maximum sequence number, since the sequence number range includes zero. The window size must be at least one less than the total number of possible sequence number values to allow acknowledgement of previous frames without duplicating a sequence number already in the window. Sequence numbers provide the ability to keep track of transmitted data. When data is transmitted, the current sequence number is assigned to it, then the number is incremented. This allows acknowledgement of individual frames, or a range of frames, based on an acknowledgement field. When a frame has been received, its

```
class ModInterval
{
public:
   unsigned int low,high,N;

   ModInterval()
   { }
   ModInterval(unsigned int low,unsigned int high,unsigned int n) :
      low(low),high(high),N(N)
   { }
   /* returns 1 if x is inside of the current interval */
   inline unsigned int memberOf(unsigned int x)
   {
      if (low <= high) {                 // 0  1  2  3  4
         return low <= x && x <= high;   //    [L    H]
      } else {                                       // 0  1  2  3  4
         return low <= x && x < N || x <= high;      //    H]    [L
      }
   }
   /* returns size of current interval */
   inline unsigned int size()
   {
      if (low <= high) {                 // 0  1  2  3  4
         return high-low+1;              //    [L    H]
      } else {                                       // 0  1  2  3  4
         return N-low + high+1;                      //    H]    [L

      }
   }
   /* PRE:  m.memberOf(x) && m.memberOf(y)
   *        where ModInterval m(low,(high)%N,N)
   *  returns true if x
   inline bool precedes(unsigned int x,unsigned int y)
   {
      // for wrapped intervals, map x and y to an unwrapped interval
      if (high < low) {
         if (x <= high) {
            x += N;
         }
         if (y <= high) {
            y += N;
         }
      }
      return x < y;
   }
};
```

Figure 6.2: ModInterval

acknowledgement field is checked to see if the other side is acknowledges receiving a transmission. To acknowledge the sequence number i, $i{+}1$ *mod maxSeq* is placed in the acknowledgement field. Go-Back-N allows only a single transmission to be accepted at a time until the Network Layer has read it. To take care of lost frames, an acknowledgement number acknowledges all frames in the window before it as well as itself. Since the receive end can only accept one single frame at a time, we know that if an acknowledgment is received for a number inside of the window, everything before it has been correctly received. The difficulty with sliding window protocols arises when the sequence number range "rolls over," meaning the values inside of the send window could be of the form $[N\text{-}1, N, 0, 1, ...]$.

To handle all the special circumstances that can arise within the send window's interval, a special class was designed: `ModInterval`. `ModInterval` stores and accesses an interval from the sequence $[0, 1, ..., N\text{-}1]$, with wrapping. The value N here is the maximum sequence number from the `LinkLayer`. Because of the wrapping, the value for *low* will not always be less than high. If $low <= high$ then the interval contains a contiguous range of values, $[low, low{+}1, ..., high]$. If $low > high$ then the interval is of the form $[low, low{+}1, ...N\text{-}1, 0, 1, ..., high]$. Figure 6.2 shows the methods of `ModInterval`. Examples in the code comments are for an interval of size 5, where 'L' denotes *low*, 'H' denotes *high*, and [ ]'s indicate the interval. A method, `memberOf`, is provided to tell whether a specific x is inside of a `ModInterval`. The method `precedes` calculates whether a given x comes before y in the managed interval. The `precedes` method is used to determine the frames affected by a specific acknowledgment number. A `size` method is also provided which returns the size of the current interval. All three methods must handle the case of wrapping in a slightly different way.

When implementing a `LinkLayer` almost all of the work of the protocol happens inside a single running thread. This means that several of the `LinkLayer` data structures can be accessed across multiple threads. Whenever this is possible, care must be taken to ensure that, in case of context switch during execution, the shared structures internal states remain consistent. There are two affected parts of `LinkLayer`: the `SendBuffer` and the `ReceiveBuffer`. Whenever data is sent via the `send` method it must be inserted into the `SendBuffer`, which is scanned to determine re-transmissions. The `ReceiveBuffer` is also accessed by the thread and via the `receive` method. Each of these structures receives its own mutex to handle synchronization issues. Since the code is well organized, the `ReceiveBuffer` must only be protected in two places and the `SendBuffer` must be protected in three places. In addition to protecting the two structures, care must be taken to insure that only the minimal critical sections are protected. Each critical section must be analyzed carefully so that the full scope is covered. The `resendFrames` method determines whether or not there is anything inside of the `SendBuffer` that has timed out and should be re-transmitted. Handling acknowledgements and clearing the `SendBuffer` is also handled by `resendFrames` to keep loop complexity minimal. Figure 6.3 shows the largest critical section in the solution, located inside of the `resendFrames` method. The `SendBuffer` is locked immediately at the start of the method and remains locked throughout the execution of the `for` loop that processes the entire `SendBuffer`. Some students do not see the importance of locking the `SendBuffer` during the whole processing stage, instead locking and unlocking it during every iteration. This can cause the `SendBuffer` to go to an inconsistent state if a context switch happens during an operation on the `SendBuffer`. Important as well is making sure that all paths to exit from the method unlock the mutex so that deadlock does not happen. Since there are two ways to return from `resendFrames` there are two separate unlock calls.

The entire extensively commented instructor solution is 350 lines, including debugging statements. Much thought was put into the solution to make sure there is as little redundant code as possible. The method call complexity was also kept minimal to help when proving correctness. The average length of a student solution is 500+ lines, and is usually fairly obfuscated. Students have

```
bool LinkLayer::resendFrames(unsigned int ack,bool ackReceived)
{
   bool frameSent = false;
   pthread_mutex_lock(&sbMutex); // ***** LOCK

   if (sendBuffer.getActiveFrameCount() == 0) {
      pthread_mutex_unlock(&sbMutex); // ***** UNLOCK
      return frameSent;
   }
   // create a ModInterval: sendBuffer interval plus one more for ack
   unsigned int lastSeq = sendBuffer.last()->seq;
   unsigned int firstSeq = sendBuffer.first()->seq;
   unsigned int n = maxSeqNum+1;
   ModInterval ackInterval(firstSeq,(lastSeq+1)%n,n);
   // use default ack value if no legal ack passed in
   if (!ackReceived || !ackInterval.memberOf(ack)) {
      ack = firstSeq;
   }
   // get base to check for timed out frames
   timeval currentTime;
   gettimeofday (&currentTime, (struct timezone *)0);
   // free ack'd frames
   // resend timed out frames; set frameSent=true if any frames sent
   TimedFrame* tf = sendBuffer.first();
   while (tf != NULL) {
      if (ackInterval.precedes(tf->seq,ack)) {
         tf = sendBuffer.removeCurrent();
      } else {
         if (tf->tv < currentTime) {
            tf->tv = currentTime + timeout;
            tf->ack = nextInboundSeq;
            tf->checksum = 0;
            tf->checksum = tf->genChecksum();
            frameSent =
             physicalLayer->send((unsigned char*)tf,
             Frame::HEADERLEN+tf->dataLength);
         }
         tf = sendBuffer.next();
      }
   }
   pthread_mutex_unlock(&sbMutex); // ***** UNLOCK
   return frameSent;
}
```

Figure 6.3: LinkLayer resendFrames method

```
bool LinkLayer::send(unsigned char* buf,unsigned int bufLength)
{
   if (bufLength > Frame::MAXDATALEN)
      throw Exception("buf too long");

   return addToSendBuffer(buf,bufLength);
}

// pre: buf has room for Frame::MAXDATALEN bytes
unsigned int LinkLayer::receive(unsigned char* buf)
{
   unsigned int size = 0;

   pthread_mutex_lock(&rbMutex); // ***** LOCK
   if (!receiveBuffer.empty()) {
      size = receiveBuffer.remove(buf);
   }
   pthread_mutex_unlock(&rbMutex); // ***** UNLOCK

   return size;
}
```

Figure 6.4: LinkLayer send and receive methods

a hard time realizing that many duties performed by the protocol share much functionality which is where the bloat comes from. The two public methods, `send` and `receive`, should consist of very few lines of code. Figure 6.4 shows how simple the method bodies can be in a concise solution. The functionality that the two methods provide is merely a public request for a service that the protocol does itself behind the scenes in a few places. Because of this, internal methods were designed and are called to keep redundancy to a minimum. In student solutions `send` and `receive` can be three to four times longer.

6.3 Testing

There are excellent opportunities for combining teaching and automated testing[16, 17]. NetLab exploits these opportunities. LinkLab projects are tested using three different methods: self-pairing, interoperability, and multi-processor behaviour.

In the linkLab test code, which is similar between all three testing methods, two `PhysicalLayer` and two `LinkLayer` objects are created. Using tester parameters `AtoBtotal` and `BtoAtotal`, a loop is entered. with the following behavior:

- The A side sends `AtoBtotal` Network Layer messages, with the ith message containing i.
- The B side attempts to receive `AtoBtotal` messages, checking that the ith received message contains i.
- The B side sends `BtoAtotal` Network Layer messages, with the ith message containing i.
- The A side attempts to receive `BtoAtotal` messages, checking that the ith received message contains i.

A single test fails when either side receives an out of order message, an incorrect message, or nothing has been received within a two second time range. Tests are executed ten times each, and all ten runs must pass in order for credit to be given.

In all, each test case is specified by eleven parameters:

- Physical Layer A: `kill`, `corrupt`, `delay`
- Physical Layer B: `kill`, `corrupt`, `delay`
- Link Layer: `maxWin`, `maxSeq`, `timeout`
- Network Layer: `AtoBtotal`, `BtoAtotal`

As a result, a large number of interesting test cases can easily be generated. Student solutions are exercised on tests of increasing difficulty, covering:

- a single network message, without Physical Layer impairment,
- multiple messages without impairment,

- multiple messages with varying impairments of a single type,
- multiple messages with multiple impairments, and
- stress tests with a large amount of message traffic and complex impairment.

In the networking domain, interoperability is both essential and difficult to achieve. In linkLab, a student implementation S is first tested with an instance of S at each Link Layer endpoint. Then, each pair of student implementations, S_0 and S_1, is tested, with S_0 as the A side and S_1 as the B side. The instructor solution is also paired with every student solution. Two ruby[18] scripts are used to accomplish interoperability: `namespaceInjector.rb` and `runAllPairs.rb`. First every source file is prepared using `namespaceInjector` to insert a unique namespace into each student solution. The `#ifndef _LinkLayer_h_` line from every header file is also changed to match the corresponding namespace. These two modifications allow instantiation of different implementations of the same class. A modified `tester.cpp` is used which contains special symbols denoting the two namespaces. The `runAllPairs` script is then executed with the following behaviour:

- Choose a pair S_i and S_j from all possible pairs of solutions.
- Copy a clean `tester.cpp` to the test directory.
- Substitute S_i's for `_Namespace_A_`.
- Substitute S_j's for `_Namespace_B_`.
- Run each test ten times and measure the run-time.
- Write the results to a file.
- Repeat until all pairs have been tested.

Interoperability tests display interesting results. When paired with the instructor's correct solution:

- Some students who cannot pass the most difficult tests solo will occasionally pass them.
- Most students with weak results pass more tests.
- Few students pass fewer tests.

This shows that an implementation which appears to be broken might only in fact be broken on either the send or receive side. Students that pass less tests stray from the protocol and are not adhering to specification.

The last round of testing is to insure correctness with threading issues. Most computers have only a single CPU core, so race conditions and synchronization issues do not always arise. A dual-core workstation and a quad-core server are used to determine if a student solution behaves differently with multiple cores. When the tester described above is executed, two linkLayer instances are created; including the tester thread this gives three running threads. With the quad-core server each thread in the test process receives a single core. Difficulties with student solutions become apparent immediately with just a dual-core processor. Most weak solutions are obvious when the core count is more than one. From a dual-core to a quad-core machine, the test results are not as radically different, but only the strongest of solutions are able to pass tests 100 percent of the time.

An interesting note is that execution time can grow greatly from single-core to multi-core tests. Use of bad synchronization techniques is the typical cause, with livelock causing the growth in run-time. The most extreme cases have single core tests finish in 20 seconds and the multi-core tests finishing in 1000 seconds. Table 6.1 shows a sampling of the test results obtained from the last time netLab was run. The test case used in the table is called throughput. Throughput has each side transmit 10,000 frames to the opposite side, with no impairment. The students chosen for the table passed the throughput test case on single and dual-core machines 100% of the time. The bottom row is the instructor's solution test results. Column A shows the average finish time on a single core machine. Column

singlecore	*dualcore*	*quadcore*	
avgtime	*avgtime*	*numpassed*	*avgtime*
21	216	8	250
20	40	4	72
21	219	6	280
22	258	4	250
20	40	8	145
21	185	8	260
23	241	9	240
20	40	7	180
40	670	10	236
60	60	10	64

Table 6.1: SMP test results for throughput test case

B show the average completion time on a dual-core machine. Columns C and D deal with results from a quad-core machine. Column C is the number of times passed out of 10 and column D is the average completion time. It's easy to see from table 6.1 that many students must have some synchronization issues to cause their solution to slow down by greater than 100% on average. The results displayed are from the throughput test, which is a high traffic-test with no impairment. The instructor's solution is shown as the bottom entry in the table. While being slower than some of the student solutions on single-core, the instructor's has consistent timing throughout all platforms and all tests. The instructors solution does not employ optimizations or tricks to speed up execution, stability is the main goal.

6.4 Code Inspection

In linkLab, testing is supplemented by informal code reviews. The idea is to get each student talking about his/her code in a focused way. The goals are to determine if a student has control of his/her code and to teach disciplined analysis of source code. Reviews are also effective at detecting plagiarism; we have found that most cheaters cannot explain the stolen code. We use two approaches for code review: (1) code walkthroughs based on test scenarios and (2) fault-based

```
tester sent A to B: 0
added to sendBuffer: [0,0,65531,4,0,0,0,0]
PL sent: [0,0,65531,4,0,0,0,0]
PL sent: [0,0,65531,4,0,0,0,0]
        PL received: [0,0,65531,4,0,0,0,0]
        added to receiveBuffer: [0,0,0,0]
        removed from receiveBuffer: [0,0,0,0]
        tester received A to B: 0
        added to sendBuffer: [0,1,65534,0]
        PL sent: [0,1,65534,0]
```

Figure 6.5: Log file sample

inspection.

In linkLab, students are required to implement log file messages, conditionally compiled, for debugging purposes. A log file message must be generated for each `add` and `remove` from the send buffer and receive buffer, and for each `send` and `receive` call made to the Link Layer and the Physical Layer. Figure 6.5 shows a log file for a test case which sends a single Network Layer message from A to B.

Messages from LinkLayer A are shown left-justified; messages from LinkLayer B are indented. The Physical Layer kills the first frame, which times out, and is then resent successfully and acknowledged. In a walkthrough, the students must describe the code on the execution path indicated in the log file. For well-organized code, a walkthrough is straightforward; for sloppy code, it is a struggle.

Ideally, we would ask students to present correctness proofs for their code. Such proofs are infeasible in linkLab; even verification experts would find a `LinkLayer` correctness proof challenging. Instead, we focus on informal proofs for simple necessary conditions.

For example, we often ask a student to prove that a particular pointer dereference is legal, i.e., that it will never cause a segmentation fault. Consider the code fragment in Figure 6.6. Line 6 invokes the `first` method of the `SendBuffer` class. If `sendBuffer` is non-empty, then `first` will return the address of the first frame; otherwise, it will return `NULL`. We know that `sendBuffer` is non-empty on line 2. Because `sendBuffer` is locked on line 1 and remains locked through line 6, we know that the dereference on line 6 is legal.

A more interesting example concerns a simple invariant I on `sendBuffer`: unless the `sendBuffer` is empty, the frames must be consecutively numbered. For example, if `sendBuffer` contains 5 frames, `maxSeq` is 7, and the first frame has sequence number 4, then the 5 frames must have sequence numbers 4,5,6,7,0. The proof is by induction, although that term tends to make the students uncomfortable. The base case is trivial: `sendBuffer` is initially empty. For the induction case, the student must consider each call on the `SendBuffer` `add` and `remove` methods, and show that, if I holds just before each call, then it holds just after the call. The proof is straightforward for well-organized code: the entire implementation will have one or two calls to `add` and one or two calls to `remove`.

For the instructor's solution the proof is quite easy for both methods. There is only one call to `add` and one call to `remove`. The `add` call is in the `addToSendBuffer` method, which is called from two places. The first place is the send method, shown earlier in Figure 6.4. The second is inside of the `Thread` method, in the case where a pure `ACK` is sent. The `addToSendBuffer` method is shown in Figure 6.7. The `SendBuffer` is locked on line 4 and remains locked for

```
    ...
1.  sendBuffer.lock(); // ***** LOCK

2.  if (sendBuffer.getActiveFrameCount() == 0) {
3.      sendBuffer.unlock(); // ***** UNLOCK
4.      return frameSent;
5

6.  unsigned int firstSeq = sendBuffer.first()->frame->seq;
    ...
```

Figure 6.6: Code sample for pointer dereferencing proof

```
1:   // pre: bufLength < Frame::MAXDATALEN
2:   bool LinkLayer::addToSendBuffer(unsigned char* buf,
3:                                   unsigned int bufLength)
4:   {
5:      pthread_mutex_lock(&sbMutex); // ***** LOCK
6:      if (sendBuffer.getActiveFrameCount()==sendBuffer.getMaxFrames()){
7:         pthread_mutex_unlock(&sbMutex); // ***** UNLOCK
8:         return false;
9:      }
10:     // build a TimedFrame to be transmitted immediately
11:     TimedFrame timedFrame;
12:     timedFrame.seq = nextOutboundSeq;
13:     timedFrame.ack = nextInboundSeq;
14:     timedFrame.checksum = 0;
15:     timedFrame.dataLength = bufLength;
16:     memcpy(&timedFrame.data[0],buf,bufLength);
17:     timedFrame.checksum = timedFrame.genChecksum();
18:     // add to sendBuffer
19:     sendBuffer.addBack(&timedFrame);
20:     // increment sequence number for next frame
21:     nextOutboundSeq = (nextOutboundSeq+1)%(maxSeqNum+1);
22:     pthread_mutex_unlock(&sbMutex); // ***** UNLOCK
23:     return true;
24:  }
```

Figure 6.7: LinkLayer addToSendBuffer method

the duration of the method. The method has two seperate exit points; lines 6 and 22 show the `SendBuffer` being unlocked so that deadlock does not occur. The next sequence number is prepared inside the locked portion of the method, on line 21. The current sequence number is first incremented, the remainder of that value divided by the maximum sequence number plus one is the next sequence number. The maximum sequence number is a valid value which is why we find the next sequence number by divide by `maxSeqNum` plus one. In the case where the current sequence number is the maximum sequence number, the next sequence number is 0. Incrementing the sequence number inside the locked body prevents an identical sequence number from being inserted into the `SendBuffer`. If the sequence number is incremented outside of the locked portion, a well timed context switch could cause a duplicate sequence number to be inserted inside the SendBuffer. Since the entire method is locked and this is the only place where there are additions to the SendBuffer, the integrity of the SendBuffer is preserved. Therefore the add half of the proof is satisfied.

To finish the proof we must look at the remove method. The remove method is called inside the resendFrames method, shown earlier in Figure 6.3. The resendFrames is called from the Thread method. The SendBuffer is locked on line 4 at the start of the method. Lines 7 and 42 show the unlock call for both possible exit points. One parameter to resendFrames is ack, which is the acknowledgement number of the latest received frame. Lines 11–14 show the creation of a ModInterval which allows us to determine the validity of ack. The ModInterval is created using the oldest sequence number in the SendBuffer as the low point and the latest sequence number as the high point. The SendBuffer is iterated through to determine if a frame has been acknowledged or needs to be retransmitted. Line 27 shows how we determine if a frame inside of the SendBuffer is to be removed. The check is performed using the precedes method from ModInterval. If the sequence number of the frame currently being inspected precedes ack then it is removed from the SendBuffer. We have already shown that the addToSendBuffer method guarantees that a contiguous range of numbers is inside the SendBuffer. A property of Go-Back-N acknowledgement numbers is that they acknowledge all frames older than the one being acknowledged. Line 28 shows where a frame is removed from the SendBuffer. Since the SendBuffer is scanned in order, we process the sequence range from oldest to newest. This means that any time a frame is removed it contains the oldest sequence number in the SendBuffer. By only removing the oldest sequence number, the integrity of the sequence range is preserved as the rest will still be a contiguous range of values. The proof is complete since we have shown that whenever something is added or removed from the SendBuffer the invariant is preserved.

We have found that it is revealing to have a student talk about his/her code.

```
bool LinkLayer::resendFrames(unsigned int ack,bool ackReceived)
{
   bool frameSent = false;
   pthread_mutex_lock(&sbMutex); // ***** LOCK

   if (sendBuffer.getActiveFrameCount() == 0) {
      pthread_mutex_unlock(&sbMutex); // ***** UNLOCK
      return frameSent;
   }
   // create a ModInterval: sendBuffer interval plus one more for ack
   unsigned int lastSeq = sendBuffer.last()->seq;
   unsigned int firstSeq = sendBuffer.first()->seq;
   unsigned int n = maxSeqNum+1;
   ModInterval ackInterval(firstSeq,(lastSeq+1)%n,n);
      // use default ack value if no legal ack passed in
   if (!ackReceived || !ackInterval.memberOf(ack)) {
      ack = firstSeq;
   }
      // get base to check for timed out frames
   timeval currentTime;
   gettimeofday (&currentTime, (struct timezone *)0);

   // free ack'd frames
   // resend timed out frames; set frameSent=true if any frames sent
   TimedFrame* tf = sendBuffer.first();
   while (tf != NULL) {
      if (ackInterval.precedes(tf->seq,ack)) {
         tf = sendBuffer.removeCurrent();
      } else {
         if (tf->tv < currentTime) {
            tf->tv = currentTime + timeout;
            tf->ack = nextInboundSeq;
            tf->checksum = 0;
            tf->checksum = tf->genChecksum();
            frameSent =
             physicalLayer->send((unsigned char*)tf,
             Frame::HEADERLEN+tf->dataLength);
         }
         tf = sendBuffer.next();
      }
   }
   pthread_mutex_unlock(&sbMutex); // ***** UNLOCK
   return frameSent;
}
```

Figure 6.8: LinkLayer resendFrames method

Lab instructors administer the reviews during the beta and final demonstrations, in about 10–15 minutes per student. It is important to focus the discussion so that the student does not ramble. Students are expected to be able to describe the functionality of every aspect of their code thereby ensuring that students understand what they write. If they cannot explain something they have done during the review, the student loses marks. A list of possible topics to ask the student is taken to the review, such as:

- Why some areas of code are or are not protected by mutex.
- What data structure is used to handle frame store and why was it chosen.
- Detail the run time complexity of a single iteration through the main loop.

In addition if the lab instructor notices an obfuscated spot in the students code, the student is asked to clearly describe what is intended.

6.5 Discussion

While good `LinkLayer` implementations are relatively short—400 to 600 lines of source code—it is challenging to produce a correct one. And, with thorough testing, few bugs go undetected.

There have been several versions of `linkLab`. The project was first written in Java. The `Go-Back-N` protocol was the implemented protocol then, as it is now. Initially the data transmission was unidirectional. Since data was only transmitted in one direction the architecture was quite limited. The receiving side transmitted acknowledgements only. An acknowledgement was an empty frame with the sequence number field used as the acknowledgement number. The send and receive sides were different classes as well, making the project seem disconnected. The forced simplicity of the project made it of little practical use. In addition there was no connection to the second course project, `routerLab`.

Later `linkLab` was extended to bi-directional communication. This sparked more interest in the project, since it started looking more like a useful service. There were several limitations in the second major revision though. The `linkLab` API used a send method with a blocking approach, the method would not return until space was available in the send window for the message. The send method blocking did not affect the original approach. With bidirectional communication, though, this simplified the logic too much as there were no special cases, such as the send windows being full. This ended up being a bit too unrealistic for effect. The sequence number, acknowledgment number, and checksum were one byte fields. For the checksum this is no problem. Complicated tests involving impairment ran the risk of overlapping the sequence number range though. False positives will arise if the sequence number range is allowed to overlap. The class level locks that the Java language utilizes took away from having to fully understand the underlying synchronization issues. Most student solutions would have every method "synchronized"; while being a quick fix this is overkill and causes a dramatic drop in performance. With the issues that arose in this revision a major decision was made to move to C++.

The coversion to C++ went smoothly, and `linkLab` turned out to be a much more elegant project afterwards. Operation was changed to emulate a real link layer service as closely as possible. Non-blocking transmission was used and 32-bit integers were used for all fields in the frames. Without the ability to synchronize entire classes, students were finally forced to understand all the race conditions and be able to identify all critical sections. At this point `synchLab` was born as it was necessary for students to experience first hand the different classic problems inherent with multi-threaded programs, and how to deal with them. C++ also allowed full control over memory so when students inspected frame contents they were able to notice the endianness of their machine. The entire project was modularized at this point and all aspects were made uniform throughout all classes and documentation. At this point the project encompassed three separate network layers: the physical layer, the link layer, and the network layer. There have been minimal changes to the API since it was converted to C++.

Initially an impairment scheme similar to the final version was used. The delay impairment technique was the same as now. Corrupt and kill however used boolean values to specify whether to corrupt/kill a frame. If a particular array element was true, the impairment would occur. Tests always had to be made with care since cycles could occur, in which a correct solution could not pass a test, because progress could not be made. Such could happen if the window size and impairment mask size were the same size, or the mask contained a cyclical pattern. If the first frame in the window is always impaired, the test is impossible to finish. Straightforward cases such as this are fairly easy to avoid, but with the move to C++, we started finding more tests failing. Test cases would not always fail, only occasionally getting stuck and usually towards the end of the test. This turned out to be a case similar to the original, but depended on the interleaving of frames and the timing within the threads performing the protocol work. Sometimes the interleaving would set up a case when the send window flushed itself at the end of the test. When the send windows size reaches the size of the impairment mask, it is possible that the position the mask is in at the moment is true. This means the test cannot finish. Figure 6.9 shows the described scenario. Note how sequence number 24 is always killed, and therefore progress is impeded. The only way to fix this problem for sure is to ensure that the impairment masks are greater in size than the send window. This leads to very messy test cases though as window sizes from 1 to 100 are used. Once this problem arose the offending parts of the impairment scheme were changed. By using a probability value between 0 and 1.0 we can introduce a bit of non-determinism. Tests now contain high values such as .99 where they were "true" originally.

The final polish to `linkLab` came in the form of an instructor developed send window, the `SendBuffer`, given to students as a starting step. In previous

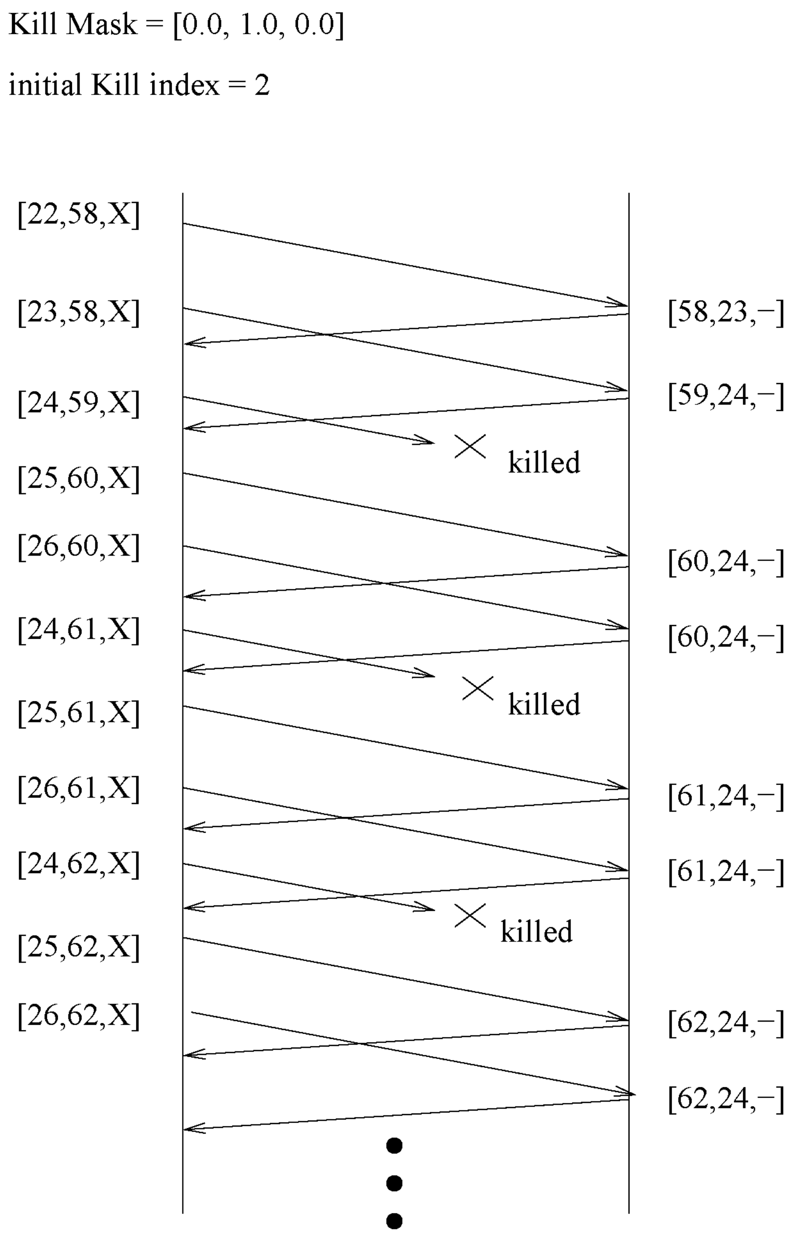


Figure 6.9: Cyclic impairment

terms students were not given the `SendBuffer` specification or implementation, and instead were given a full specification of how a send window should operate. Students had a very hard time designing the data structure though, and many horrible implementations were developed. Most students by this time had taken many design courses so this was a very puzzling issue. Issues ranged from overly complicated code, to spreading functionality over a great area and not abstracting the design. Most students who were strong coders didn't have too much problem making a send window that worked, although some found the send window the hardest part to design. Weak students would occasionaly come up with great solutions, but the major hurdle of having to implement an abstract structure that generically handles a protocol was too much for some to overcome. The send window is a very important part of the protocol, but the emphasis was meant to be on the protocol itself and related issues. To level the playing field, all students now start out with the `SendBuffer` and can focus solely on understanding the protocol.

An odd byproduct of `linkLab` being rather non-trivial arose when the code base was made in C++. Students became quite competitive, working to pass all the test cases that they can develop and trying to come up with special cases that we might test them on during marking. It has given the lab a good atmosphere as the students are taking pride in attempting to finish a project 100% to specification. Students have strived to be the first to produce a solution comparable to the instructor solution, which passes every test we have designed. The final revision of the lab has been run through twice, and so far no student solution has passed every test 100% of the time. Some have come close but due to the severe nature in which solutions are tested any bug of even the smallest size will come out. Many students have said that after the completion of `linkLab` they feel that their overall coding skills have increased more rapidly than through other course projects.

Even though Go-Back-N is used as the protocol to be implemented the framework is flexible enough to adapt to other link layer protocols. The test suite that is included does not required modification to accomodate further protocols. Experimentation has been done to explore the possible use of TCP selective repeat as the chosen protocol for future iterations.

7. Implementing a Router: routerLab

In `routerLab`, students use C++ to implement a router. A router is a device that forwards data packets through a network via a routing table. A routing table contains information on how a router can reach other destinations on the network and the cost to get there. The current implementation is based on the Distance Vector protocol, which uses the Bellman-Ford[1] shortest path algorithm. A network is treated as a weighted graph, where routers are vertices and the links are the edges of the graph. Every `Router` has a unique id and `Router` id's are taken from a contiguous range of integers starting with zero. The Distance Vector protocol uses the Bellman-ford algorithm in a distributed manner where each router communicates its routing table to each of its neighbours at a specified microsecond `updatePeriod`. The routing information transmitted, or Distance Vector, is an array of unsigned integers representing the full delay in microseconds of a path. This Distance Vector is copied directly from the routing tables cost column. Receiving neighbours choose the lowest advertising cost, add this entry into its routing table, then transmits its updated routing table to each of its neighbours upon the next `updatePeriod`. The algorithms completion depends on the diameter of the network, because complete paths are not produced until the shortest-path information from the two farthest routers percolates to each other. The Distance Vector protocol performs the algorithm continuously, to keep itself and its neighbours updated. Each router must communicate with other routers in the network in order to efficiently route packets from host to host. Echo requests are sent to each neighbouring router to determine delay, every `echoPeriod` microseconds. Networks of routers are connected using the `linkLab` link layer. The `LinkLayer` uses a scheme which does not handle impairment, since impairment is not a direct part of the project.

7.1 routerLab API

Students are supplied with a skeleton class containing four empty methods: the `Router` constructor, `thread`, `getID`, and `getRoutingTable`. Figure 7.1 shows the `Router` API. The `thread` is where protocol duties are performed. The `getID` method returns the unique id of the `Router`. A copy of the current routing table is returned by the `getRoutingTable` method. The `getRoutingTable` method is also used to visually determine correctness of a students `RoutingTable` during testing.

The Router's constructor uses the supplied parameters to initialize the core components required by the protocol:

- saving the `updatePeriod`,
- saving the `echoPeriod`,
- allocating arrays of unsigned integers to store a distance vector array from each connected neighbor,
- copying the initial supplied distance vector arrays into local storage,
- allocating an array of unsigned integers to represent direct link costs,
- copying the initial link costs into local storage,
- allocating space to hold a routing table,
- calculating an initial routing table based on the supplied initial distance vectors and direct link costs,
- allocating an unsigned integer array to hold the outgoing distance vector, and
- setting the `echoPeriod` and `updatePeriod` first deadlines.

```
/* Implement a router using the distance vector protocol.
 * A router must:
 * Maintain its routing table using the distance vector algorithm.
 * Route data frames received from its neighbours.
 * Periodically send distance vector frames to its neighbours.
 * FINAL ONLY: Use echo request and reply frames to measure
 * link delays to its neighbours. */
class Router : public AbstractRouter
{
private:
/* running thread where main functionality exists. */
static void* thread(void*);

public:
/* Constructor must be implemented for each AbstractRouter subclass.
 * Initialize the router and start the router thread.
 * id: the identifier of this router.
 * numNeighbors: total number of neighbors for this Router.
 * neighbourIds: neighbourIds[i] contains the id of the router
 *  at the other end of linkLayers[i].
 * linkLayers: the LinkLayers connecting this router to its neighbours.
 * linkDelays: linkDelays[i] contains the initial value for
 *  the time in microseconds to send a frame from this router
 *  to the router at the other end of linkLayers[i].
 * numRouters: total number of routers in the network.
 * initialDistanceVectors: numNeighbors distance vectors,
 *   each with numRouters entries.
 * updatePeriod: the time in microseconds between routing table updates.
 * echoPeriod: the time in microseconds between echoRequest transmissions.
   Router(unsigned int id,unsigned int numNeighbors,
    unsigned int neighborIds[], AbstractLinkLayer** linkLayers,
    unsigned int linkDelays[], unsigned int numRouters,
    DistanceVector** initialDistanceVectors,unsigned int updatePeriod,
    unsigned int echoPeriod0);

/** Return the id of this router. */
int getID();

/** Return the current routing table. */
RoutingTable* getRoutingTable();

};
```

Figure 7.1: Router specification

The `Router thread` is started at construction time and left to perform periodic tasks including:

- forwarding data packets according to the routing table,
- accepting router update packets and using them to update the routing table,
- generating distance vector packets at a specified interval,
- sending echo request packets and monitoring echo reply packets to estimate link delay, and
- replying to echo request packets from neighbors.

Following is a description of all files provided:

- `AbstractRouter.h`: Contains the public API and documentation for `AbstractRouter` and contains API's for ancillary classes:
  - `RouterFrame`: set of class constants which describe one of four different frame types used in the distance vector algorithm.
    - `DataFrame`: consists of type, destination address, and data length fields. The data is piggy-backed onto a `DataFrame`.
    - `VectorFrame`: consists of a type and count field, which holds the number of entries in the attached distance vector. The actual distance vector is piggy-backed onto the `VectorFrame`.
    - `EchoFrame`: consists of a type field and a timestamp field. When an echo request is transmitted, the timestamp is filled with the local time. An echo reply maintains the original timestamp and transmits the value back to determine delay time.
  - `RoutingTable`: the table used for routing. Contains two columns, one representing the link to forward on to reach the destination `Router` and the second is the microsecond delay to arrive at the destination. `Router` id is used to index into the `RoutingTable`.

  - `DistanceVector`: a wrapper for an array and the array's length used for management of distance vectors.

- `AbstractRouter.cpp`: implementations of ancillary classes from `AbstractRouter`.
- `Router.h`: method prototypes for `Router`.
- `Router.cpp`: contains signatures for the methods that must be implemented. All method bodies are empty except for a `return` statement.
- `betaTester.cpp`: tests whether a `Router` can initialize, update itself, and route a data frame.
- `finalTester1.cpp`: tests whether a `Router` responds to echo requests.
- `finalTester2.cpp`: tests whether a `Router` generates echo requests, and uses them to update routing table entries.
- `LinkLayer`: a minimal solution to the earlier project `linkLab`.

Figure 7.2 displays two supplied classes: `RoutingTable` and `DistanceVector`. The `RoutingTable` is the core routing table that a `Router` uses to determine where to forward frames. There are two defined columns and an implied column in the `RoutingTable`. The two pointer fields, `outLink` and `delay`, are the defined columns. The implied column is the `Router` id used when addressing the `outLink` and `delay` columns. The `len` field holds the number of `Routers` in the network, as there must be an entry in the table for every `Router`. The `outLink` field defines the immediate link that a frame would be transmitted on to reach the destination `Router`. The `delay` field is the full path estimated delay of sending a frame to the specified `Router`. So, to forward a frame to `Router` N, we transmit on link `outLink[`N`]` which will reach `Router` N in approximately `delay[`N`]` microseconds.

```
class RoutingTable
{
public:
   unsigned int* outLink;
   unsigned int* delay;
   unsigned int len;

/** Create a routing table with all 0 entries. */
   RoutingTable(unsigned int numRouters);

/** Initializes a routingTable using two integer arrays.
  * Accepts parameters:
  *  ids[i] contains the next hop to get to Router i.
  *  c[i] contains the full path cost to get to Router i. */
   RoutingTable(unsigned int* ids,unsigned int* c, unsigned int num);

/**Initializes RoutingTable using the contents of another RoutingTable.*/
   RoutingTable(RoutingTable& other);
};
ostream& operator<<(ostream& out,RoutingTable& r);
bool operator==(RoutingTable& x,RoutingTable& y);

class DistanceVector
{
   unsigned int* dv;
   unsigned int len;
public:
   DistanceVector(unsigned int* dv0, unsigned int len0);
   unsigned int* getDV();
};
```

Figure 7.2: abstractRouter.h - RoutingTable and DistanceVector

The `DistanceVector` class is supplied to give students an easy way to handle multi-dimensional arrays a bit easier in C++. Due to the dynamic nature of the project, a pure multi-dimensional array cannot be used; we will not always know all of the dimensions of the array. It is used in the `Router` constructor when an array of initial distance vector arrays are required to be passed as a parameter. The `DistanceVector` is a wrapper class for an array of unsigned integers and a length field to determine the contained arrays size. A `getDV` accessor method is

```
class RouterFrame
{
public:
   static const unsigned char DATA_FRAME = 1;
   static const unsigned char VECTOR_FRAME = 2;
   static const unsigned char ECHO_REQUEST = 3;
   static const unsigned char ECHO_REPLY = 4;
   unsigned int type;
};

class DataFrame : public RouterFrame
{
public:
   unsigned int dstAddr;
   unsigned int len;      // length in bytes
};

class VectorFrame : public RouterFrame
{
public:
   unsigned int count;
};

class EchoFrame : public RouterFrame
{
public:
   timeval transmitTime;
};
ostream& operator<<(ostream& out,DataFrame& d);
ostream& operator<<(ostream& out,VectorFrame& v);
```

Figure 7.3: abstractRouter.h - Router frame types

supplied as well to obtain a pointer to the array.

Figure 7.3 shows the different frames that are defined to Routers, supplied in `AbstractRouter.h`. These frames are intended for Router-to-Router communication only. There are four types of frames: `DataFrame`, `VectorFrame`, and two types of `EchoFrame`. The different `EchoFrame` types are `ECHO_REQUEST` and `ECHO_REPLY` as detailed in the `RouterFrame` class definition. Each frame inherits the `type` field from `RouterFrame` to be able to determine its type. A `RouterFrame` is received as an array of bytes and requires a cast to RouterFrame to determine type. Once type has been determined, the byte array is cast to the proper frame type. The received frame can have extra data piggy-backed depending on its type; this extra data is not defined by any class attribute.

The DataFrame encapsulates the generic frames that are transmitted Router to Router. The DataFrame contains two additional fields: dstAddr and len. The dstAddr field is an unsigned integer defining a particular Router id, which is the intended destination for the information. The len field specifies the number of bytes of data in the DataFrame. The size of the original array should be len + sizeof(DataFrame). The bytes of data are located after all the fields in a contiguous area. The data is retrieved by indexing the original array.

Periodic network updates are transmitted to neighbours using the VectorFrame. The VectorFrame works similar to the DataFrame. Since the VectorFrame is only transmitted to a neighbor Router one hop away, it does not need an address field. The len field operates like the DataFrame and specifies the number of bytes of data included after the class fields. The size of the original array should be count * sizeof(unsigned integer) + sizeof(VectorFrame). The attached distance vector is an array of unsigned integers, taken from the delay column of the RoutingTable. Each integer represents the full path cost, in microseconds, from the sender to another Router in the network. Each implied index in the distance vector array is an id of a Router in the network.

The EchoFrame is used for timing single hop delay between neighbor Routers. Periodically a Router will transmit an EchoFrame of type ECHO_REQUEST to all of its neighbours. The transmitTime field contains the transmission time of the ECHO_REQUEST. An ECHO_REPLY frame is the response to an ECHO_REQUEST.

7.2 Instructor's Solution

The instructor's solution is a complete implementation of the project. The solution has not gone through as many revisions as `linkLab` but is considered to be stable, due to less complexity. The instructor `Router` passes all tests that have been designed. The project is in many ways simpler than `linkLab`, but does have some intricacies that many students miss when they are coding the project. A `Router` needs to be coded as a self-contained entity which can discover its network topology regardless of its own or anyone elses `Router` id.

Most of the code for `routerLab` is straightforward. Memory management is one of the few non-trivial issues that arises. Inside the constructor care needs to be taken to perform deep copies of all necessary parameters, because of differences between stack and heap variables. If deep copies are not performed then eventually the data is overwritten on the stack and the program crashes for seemingly no reason. Once the constructor has initialized the proper data the, `thread` method is started.

The `thread` method is where the core of the `Routers` work is performed. A `Router` has two main jobs: processing data frames and maintaining network topology. Figure 7.4 shows the processing half of the duties. Processing frames requires a `Router` to iterate through all of its interfaces to see if a frame is available. If a frame is read, it is cast to a `RouterFrame` to determine its type. Depending on the type, the frame is passed to one of four methods. When a `DataFrame` is received it is first checked if the destination is itself. If a frame is destined for itself it is silently dropped. If a frame is for a different `Router` it is forwarded out on the interface defined in the `outLink` entry of `RoutingTable` for the `dstAddr` field from the `DataFrame`. When a `VectorFrame` is received it is copied into the corresponding distance vector for the interface it is received on. These saved distance vectors are used periodically for `RoutingTable` updates. When a `Router` receives an `ECHO_REQUEST`, the frame type is modified to `ECHO_REPLY` and the frame is forwarded back to the sender, with the timestamp unmodified. When an ECHO_REPLY is received, round trip time is calculated by finding half of the difference between the current time and the transmitTime field. This delay is stored as the single hop time in the linkDelays array.

The second half of a Router's duties includes updating itself and the rest

```
void* Router::thread(void* ptr)
{
  timeval t1;
  Router* owner = (Router*)ptr;
  unsigned char tempFrame[116], tempDV[116], tempPing[sizeof(EchoFrame)];
  unsigned int bufSize;
  RouterFrame* frame;
  while(true) {
     for(unsigned int i = 0; i < owner->numNeighbors; i++) {
        bufSize = owner->linkLayers[i]->receive(tempFrame);
        if (bufSize >= sizeof(RouterFrame)) {
           frame = (RouterFrame*) tempFrame;
           switch (frame->type) {
              case RouterFrame::DATA_FRAME:
                 if(bufSize == sizeof(DataFrame) &&
                  bufSize == sizeof(DataFrame)+(DataFrame)frame->len)
                    owner->processDataFrame(tempFrame, i);
                 break;
              case RouterFrame::VECTOR_FRAME:
                 if(bufSize >= sizeof(VectorFrame) &&
                  bufSize==sizeof(VectorFrame)+(VectorFrame)frame->count)
                    owner->processVectorFrame(tempFrame, i);
                 break;
              case RouterFrame::ECHO_REQUEST:
                 if(bufSize == sizeof(EchoFrame))
                    owner->processPingFrame(tempFrame, i);
                 break;
              case RouterFrame::ECHO_REPLY:
                 if(bufSize == sizeof(EchoFrame))
                    owner->processPongFrame(tempFrame, i);
                 break;
           }
        }
     }
```

Figure 7.4: Router thread method - frame processing

```
timeval curr;
gettimeofday(&curr,NULL);
if (owner->nextEcho <= curr) {
   for (unsigned int i = 0; i < owner->numNeighbors; i++) {
      EchoFrame* echo = (EchoFrame*)tempPing;
      echo->type = RouterFrame::ECHO_REQUEST;
      gettimeofday(&curr,NULL);
      echo->transmitTime = curr;
      owner->linkLayers[i]->send(tempPing, sizeof(EchoFrame));
   }
   owner->nextEcho = curr + owner->echoPeriod;
}
gettimeofday(&curr,NULL);
if (owner->nextUpdate <= curr) {
   owner->mergeDVTable();
   owner->updatedOnce = true;
   for (unsigned int i = 0; i < owner->numNeighbors; i++) {
      VectorFrame* v = (VectorFrame*)tempDV;
      v->type = RouterFrame::VECTOR_FRAME;
      v->count = owner->numRouters;
      memcpy(&tempDV[sizeof(VectorFrame)],
       owner->outDVTables[i], owner->numRouters*sizeof(int));
      tempDV[owner->neighborIds[i]] = owner->updatePeriod * 4;
      owner->linkLayers[i]->send(tempDV,
       sizeof(VectorFrame)+v->count*sizeof(int));
   }
   gettimeofday(&curr,NULL);
   owner->nextUpdate = curr + owner->updatePeriod;
}
usleep(10);
```

Figure 7.5: Router thread method - Router updating

of the network. Figure 7.5 shows the two separate update duties from the `Router` thread: echo requests and routing table updates. The first block is executed every `echoPeriod` microseconds. Inside an `ECHO_REQUEST` frame is built, the current time is included, and it is transmitted on every interface. The update block is executed every `updatePeriod` microseconds. First we execute the Bellman-Ford shortest-path algorithm on the distance vectors saved from the neighbors. The routing table update algorithm is shown in Figure 7.6. Costs are calculated by

```
void Router::mergeDVTable()
{
   pthread_mutex_lock(&rtMutex);
   unsigned int updatedDelay = 2147483647;
   unsigned int updatedNextHop = 2147483647;
   for (unsigned int h = 0; h < numRouters; h++) {
      for (unsigned int i = 0; i < numNeighbors; i++)
         if (inDistanceVector[i][h] + linkDelays[i] < updatedDelay ) {
            updatedDelay = linkDelays[i] + inDistanceVector[i][h];
            updatedNextHop = i;
         }
      routingTable.delay[h] = updatedDelay;
      routingTable.outLink[h] = updatedNextHop;
      updatedDelay = 2147483647;
   }
   routingTable.delay[id] = 0;
   routingTable.outLink[id] = 0;
   pthread_mutex_unlock(&rtMutex);
   for (unsigned int i = 0; i < numNeighbors; i++) {
      memcpy(outDVTables[i],&routingTable.delay[0],
       numRouters*sizeof(int));
   }
}
```

Figure 7.6: Router mergeDV method

taking the lowest full path delay value when comparing the sum of all `Routers` costs to a destination and the single hop cost to get to that `Router` through the pertinent interface. The lowest value and interface which provides this value are saved into the `RoutingTable`. Once shortest paths have been calculated and the `RoutingTable` has been updated, the other `Routers` in the network are alerted. A `VectorFrame` is built by copying the `delay` column from the `RoutingTable` as the data. The `count` field in the `VectorFrame` is set as the number of `Routers` in the network. The `VectorFrame` is then transmitted on all interfaces.

7.3 Testing

Routers are tested by designing arbitrary networks and running various tests on them to insure correct results. Students are encouraged to make their routers as generic as possible to stay away from pitfalls one might encounter designing for a single network layout. Multiple different network topologies are used with each test scenario to make sure this is the case. In addition the network size is usually larger than the amount of instantiated `Routers`. These "virtual" `Routers` are addressable, but are hidden behind the tester and are used for more complicated routing. Students are supplied with simple initial tests and given diagrams describing the networks in the test. The tests contain the bare minimum of functionality and test a small subset of features at a time. Tests run on a timeline with all events for a correct `Router` expected at specific times. Tests rely on visual feedback to determine if a student passes or fails, results are inspected on screen for correctness. There are four main tests in `routerLab`: betaTester, finalTester1, finalTester2, and convergence. The betaTester and two finalTesters are given to the students, while the convergence test is executed by the instructor.

In the first test scenario, betaTester, the most basic `Router` functionality is tested. The focus is on: initialization, updating, and routing. After a `Router` is instantiated, its routing table is obtained and printed. The routing tables correctness is verified using the instructors results. A frame is sent through the `Router` to verify it arrives on the correct interface. The `Router` is then sent a new distance vector frame on one of its interfaces which it should use to update its routing table upon the next `updatePeriod` elapsing. The tester pauses until the update should have occured, then prints the routing table again for verification. A final frame is sent through the `Router` to verify that it will arrive on the new correct interface.

The second scenario, finalTester1, tests whether a `Router` responds to echo requests. The `Router` is started, with very high values for `pingPeriod` and `updatePeriod`. The high values ensure that there are not any echo requests or update frames. After instantiation the tester pings the `Router` on all its interfaces. The tester then monitors the interfaces for a correct response from the `Router`. If a `Router` is not following the specification, then invalid update Frames might be seen.

The third scenario, finalTester2, tests whether a `Router` generates echo requests and uses the echo replies to correct an out of date routing table. This time the `Router` is instantiated with incorrect values in the `linkDelay` array, which contains the delays between itself and its neighbors. The routing table is printed to check initial correctness. The tester replies to all echo requests that are received from the `Router` under test. After an `updatePeriod` has elapsed the tester prints the routing table and forwards a frame through the `Router`. The test succeeds if the routing table shows that the echo reply round trip time was used for routing table recalculation and that the frame was received on the correct interface.

The last and most interesting test is one that subjects student `Routers` to similar conditions they would experience if their `Routers` were booted in the real world. For this test, multiple `Routers` are instantiated and connected together. The total number of `Routers` in the network is greater than the number instantiated, which aids in the diversity of testing scenarios. The "virtual" `Routers` are situated "behind" the tester, which allows for the correct path to a "virtual" router to be validated easily by monitoring the tester links. Convergence testing initializes routers with invalid initial distance vectors and incorrect link costs. All the initialization data is zeroed, so that every `Router` has essentially an empty Routing Table upon construction. `Routers` have to measure delay to their neighbors and slowly percolate this data through the network until all routers have enough information to have identical routing data. The amount of time to converge is found by multiplying the diameter of the "real" network by the `updatePeriod` and adding a small delta. The addition of a small delta to the converge time gives any "slow" routers extra time to converge. When this deadline has elapsed Frames are transmitted to a `Router` on the network and the tester interfaces are monitored. The test fails if a frame is not received on the correct interface. If a frame is received correctly, several path costs in the network are modified. The path delays are updated via `NetQueue` API calls. After the network re-converges, frames are transmitted again to determine if routing information is still consistent. The convergence test has shown bugs in otherwise perfect looking solutions, which has made it very useful.

7.4 Code Inspection

For `routerLab`, code review is used again although not in as much depth as in `linkLab`. The same methodology applied in `linkLab` is employed here again. Three areas are inspected: pointer dereferences of received frames, handling of non-responsive links, and the count-to-infinity problem. Students by now are used to the format of the demonstrations and are usually quite adept at answering the questions.

Students must prove that after a `RouterFrame` has been received, the dereference of its pointer is safe at the forwarding stage. The same pointer dereference proof is required during demonstration from every student. Due to the limited size and scope of the project, this may only require a few checks in the code. For a convoluted solution, this proof can still be very cumbersome as the student is required to show for a single dereference that the pointer is safe through every method it has been passed through. The instructor's solution is quite concise and the proof is very easy with this code. There are two forwarding methods which require the proof: `processDataFrame` and `processPingFrame`. Figure 7.7 shows the four process methods, each frame type is handled by its own method. Each of these methods is called from the `thread` method, so if the dereference is valid in the `thread` method, then the dereference is safe in the frame processing methods. Figure 7.4 shows the `thread` method. After a buffer is read from the `LinkLayer`, first it is determined if it contains at least as many bytes as a `RouterFrame` should. If the `RouterFrame` is valid, its size is again checked to make sure the buffer is at least as big as each frame type should be. If the buffer contains the expected bytes, then dereferencing it is safe as there will be no corrupting or reading of

```
void Router::processDataFrame(unsigned char* tempFrame,int i)
{
  DataFrame* dataFrame = (DataFrame*)tempFrame;
  if (dataFrame->dstAddr == id) // for me
   return;
  else {
   int nextHop = routingTable.outLink[dataFrame->dstAddr];
   linkLayers[nextHop]->send(tempFrame, sizeof(DataFrame)+dataFrame->len);
  }
}
void Router::processVectorFrame(unsigned char* tmpFrame,int intrfceNumber)
{
  tmpFrame+= sizeof(VectorFrame);
  memcpy(&inDistanceVector[intrfceNumber][0],
   tmpFrame,numRouters*sizeof(int));
}
void Router::processPingFrame(unsigned char* tempFrame,int intrfceNumber)
{
  EchoFrame* echoFrame = (EchoFrame*)tempFrame;
  echoFrame->type = RouterFrame::ECHO_REPLY;
  linkLayers[intrfceNumber]->send(tempFrame, sizeof(EchoFrame));
}
void Router::processPongFrame(unsigned char* tempFrame,int intrfceNumber)
{
  EchoFrame* echoFrame = (EchoFrame*)tempFrame;
  timeval curr, diff;
  gettimeofday(&curr, NULL);
  diff = curr - echoFrame->transmitTime;
  linkDelays[intrfceNumber] = (diff.tv_sec*1000000 + diff.tv_usec) / 2;
}
```

Figure 7.7: Router process event methods

data that should not be.

There are some aspects of `routerLab` which are very hard to design tests for, and that have many different ways of being implemented. For these areas the students are asked to describe how they handle the various situations. The handling of non-responsive links is always an interesting topic. Students are told write code to handle the situation, but not to let it interfere with normal operation. This is important because if a student decides his/her link "goes down" too soon, the rest of the network's routing could be adversely affected. Many different ideas have been seen such as:

- Deeming a link down when no pings have been received during an entire `updatePeriod`, at the end of the `updatePeriod`.
- Keeping track of non-returned pings, and marking a `Router` down after a set number of pings have gone unanswered regardless of `updatePeriod` elapsing.
- After a deadline, taking the elapsed time for the oldest transmitted ping, and periodically updating the link cost to this growing value.

While the first two approaches seem fairly good, they do have one drawback: what if the link just temporarily becomes congested or very slow? Since the general approach to marking a link down is to make its cost a relative infinity, these methods will cause immediate updates to percolate network wide. In this case, it is not desirable to cause a huge routing table update across the network immediately because the downed host could possibly "wake up" before the updates have spanned the diameter of the network. The third approach solves this problem; instead of marking the link's cost as infinity, the oldest missing pings transmission time is used as the value. This allows routers to decide gradually how they should modify their routing tables and allows a path cost to increase significantly without triggering any ill side effects.

The classic problem of count-to-infinity is also mentioned in the specification. Students are encouraged to research the problem and integrate a solution into their project. The count-to-infinity problem arises when a link from `routerA` to `routerC` goes down and the shortest path from `routerB` to `routerC` is through `routerA`. If the cost of `routerB` to `routerC` through `routerA` plus the cost of `routerB` to `routerA` is less than any other path to `routerC` that `routerA` has available, then a cycle emerges. This will cause `routerA` to see the shortest path to `routerC` through `routerB`, while `routerB` sees the shortest path to `routerC` through `routerA`. There are many documented approaches to handle the issue. The approach most commonly seen uses split horizon combined with poison reverse[19]. With this approach, when distance vectors are transmitted they are modified. The routing table is referenced to find any destinations which go through the distance vector's intended receiver. Wherever this is found the path cost is increased to a very large amount. This prohibits single hop router loops, but does not do anything for multi-hop loops[20]. We do not test for loops with our test cases and do our best to ensure that loops do not happen. This means any drawbacks of the approach do not adversely affect the student.

7.5 Discussion

The `RouterLab` specification was developed as a way to have students implement a second aspect of networking. In previous offerings of CSC450, the second course project was a networked game of the students choice. Some students did not enjoy this project because it required use of a graphics toolkit and emphasized gaming logic over lower level networking topics. A simulated router seemed like a good candidate, since almost every student owns a router.

There is less code to write for a correct `routerLab` solution than for the previous `LinkLayer` project. Even with less code to write, students have shown they have a harder time visualizing routerLab. Since a router is a self-contained autonomous unit, students must think deeply how to perform the router's duties. Simple generic operations, such as forwarding, are straightforward in the implemention due to the use of unsigned integers. Maintaining network topology transparently requires understanding the abstraction exactly in order to function as any `Router`.

Although it is not as mature a project as `linkLab`, `routerLab` has shown much promise. The first incarnation of `routerLab` had it as a disjointed project, that shared a couple similar looking API's but no actual classes. The goal all along was to have the two projects work together though so API's of the two projects were developed side by side. Thus, the concept of a "Link" from the original `routerLab` ended up as the `NetQueue`, which is now the backbone of `linkLab` as well now.

Similar to `linkLab`, routerLab started out as a project coded in Java. Also like `linkLab`, it used a busy wait technique when transmitting out from an interface. Busy wait was switched to non-blocking in the next iteration when the project was converted to C++ along with `linkLab`. Although the internal workings have been drastically changed, the project itself has not evolved as much as `linkLab`. Since `linkLab` was already a bit mature when `routerLab` was created some of the impending difficulties might have been avoided with experience. The API is still almost exactly the same, Java and C++ differences withstanding. Only one additional parameter has been added to the constructor, the period at which to ping.

There are many areas where routerLab could be enhanced.

- Automatic network generation, which would help during the initialization phase, currently the most time consuming part.
- Using dotted-quad IPv4 addressing schemes for a `Router` and hosts under the `Router`.
- Allow for multiple subnets and gateway protocols for routing amongst the subnets.
- Network visualization showing `Routers`, frames, updates, and routing tables for a configuration and test case.
- Interactivity with the visualization where `Routers` could be "killed" or frames could be sent on the fly.

Several of these and additional ideas were supplied by students, some of which would greatly enhance the project. One student suggested modifying the architecture slightly to be an effective load balancer between several internet connections.

Students were usually very positive about `routerLab`. The fact that `Routers` are connected together by implementation of the same `LinkLayer` service from the first project drew much interest. Many students desired to have further more advanced projects based on `routerLab`, since the architecture is smaller and requires less code for implementations.

The routing protocol used for routerLab has always been the Distance Vector protocol. However the architecture is robust enough to handle any other routing protocol. The link-state protocol has been implemented roughly as a test for future implementations. Some of the supplied test programs will not work as expected with other protocols. The convergence test program, which is not currently supplied to the students, is robust enough to handle any protocol due to its nature of allowing a network to converge.

8. Conclusions

The purpose of `netLab` was to push students towards better software engineering practices using network engineering principles, such as adherence to precise specification, careful design, and thorough validation. This goal was consistently met every term, with some students commenting that their skills advanced faster over the course of the lab than the rest of the curriculum. We think this is because of the forced adherence to strict specification, testing, code review, and unification of API's throughout the lab. Students witnessed firsthand the necessity of adhering to the principles since network interoperability demands it.

The five parts of the lab each contributed a unique learning experience and furthered software engineering skills. The synchronization lab, `synchLab`, was highly regarded by all students. The ability to demonstrate many classical synchronization issues in ways that applied to current programming projects helped them understand complications they might have only seen minimally in an operating systems course. `Snifferlab` gives hands on experience with a packet sniffer, `WireShark`, to teach more about internet protocols and to use that knowledge to solve problems using a capture file.

The `PhysicalLayer` turned into an extremely powerful simulated network service and allowed connection of `linkLab` and `routerLab`. The data link layer project, `linkLab`, reinforced several very important topics: concurrency, specification adherence, interoperability, and execution speed. Students could witness the effect of poor synchronization code or bloat that would cause extremely slow finish times for tests. Competition always arose with `linkLab` as well, which made every student take pride in their code and strive to have better test case performance than their peers. Many of the weakest students that were able to complete `linkLab` gained enough skills by the end of the project to be able to be on par with the top students for `routerLab`.

The final project, `routerLab`, was an excellent closing project. It showed that, even with a much smaller code base than `linkLab`, the complexity could be just as great. Coordinating multiple autonomous entities displayed the importance of timing independent algorithms and interoperability. The convergence tester is the most interesting aspect of this project as it allows the router to be tested in a situation that closely resembles a real network restarting. The convergence tester also allows many different routing protocols to be plugged into `routerLab` and tested with minimal modification.

The lab is ready to be deployed in future courses. Also, `netLab` was designed with future use in mind so different protocols and test suites are very easy to develop. This makes `netLab` an excellent research tool in various levels of computer science courses dealing with algorithms, operating systems, and networking topics.

Bibliography

[1] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, fourth edition, 2003.

[2] RFC 1075 - Distance Vector Multicast Routing Protocol, 1988. http://www.faqs.org/rfcs/rfc1075.html.

[3] J.L. Sobrinho. On the convergence of path vector routing protocols. *IEEE Workshop on High Performance Switching and Routing, 2001*, pages 292–296, 2001.

[4] RFC 1058 - Routing Information Protocol, 1994. http://www.faqs.org/rfcs/rfc1058.html.

[5] RFC 1661 - The Point-to-Point Protocol (PPP), 2002. http://www.faqs.org/rfcs/rfc1661.html.

[6] M. Casado, V. Vijayaraghavan, G. Appenzeller, and N. McKeown. The Stanford Virtual Router: a teaching tool and network simulator. *SIGCOMM Comput. Commun. Rev.*, 32(3):26, 2002.

[7] E. Kohler, R. Morris, B. Chen, J Jannotti, and M Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[8] B. Kneale, A De Horta, and I. Box. V.

[9] T. Rosenblum, M.; Garfinkel. Virtual machine monitors: current technology and future trends. *Computer*, 38(5):39–47, May 2005.

[10] M. Carson and D. Santay. NIST Net: A Linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):111–126, 2003.

[11] L. Fabrega, J. Massageur, T. Jove, and D. Merida. A virtual network laboratory for learning IP networks. *ITiCSE*, pages 161–164, 2002.

[12] M. Erlinger. Lab exercises for computer networking courses. *ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education*, page 305, 2006.

[13] H. Zimmermann. Osi reference model – the iso model of architecture for open systems interconnection. *IEEE Transactions on Communications Review*, 28(4):425–432, 1980.

[14] W.R. Stevens. *UNIX Network Programming, Volume 2.* Pearson Education, 2nd edition, 1998.

[15] D.M. Hoffman and P. Walsh. Teaching programming with minimal examples. In *Proc. Western Canadian Conference on Computing Education.* University of Northern British Columbia, May 1997.

[16] D.M. Hoffman, P.A. Strooper, and P. Walsh. Teaching and testing. In *Ninth Conference on Software Engineering Education*, pages 248–258. IEEE Computer Society, April 1996.

[17] B. Long, D. Hoffman, and P. Strooper. Tool support for testing concurrent Java components. *IEEE Trans. Soft. Eng.*, 29(6):555–566, June 2003.

[18] D. Thomas. *The Pragmatic Programmer's Guide.* Addison-Wesley, 2nd edition, 2004.

[19] RFC 1058 - Routing Information Protocol, 1988. http://www.ietf.org/rfc/rfc1058.txt.

[20] Christian Huitema. *Routing in the Internet.* Prentice Hall, 2nd edition, 2000.