15-441 Spring 2007: Project # 3
Reliable Sliding-Window Transport

Out: Thursday, Mar 24th, 2007
Due: Thursday, April 19, 2007, 11:59 pm

1 Introduction

In the previous projects, you implemented an application layer service (SMTP), and the routing layer (IP forwarding and routing). In this project you will implement a transport layer. You will implement a reliable transport layer similar to TCP.

IMPORTANT: If your IP forwarding code does not already meet the requirements of Project 2, you must fix the remaining bugs and implement the remaining features in order to successfully complete this project. If you believe your IP forwarding code is hopelessly broken, please bring it to our attention as soon as possible. Your routing daemon need *not* be working to complete this project; static routes will suffice.

2 Logistics

2.1 Handouts

The project directory (which will be referred to as `<projdir>` can be located at

`/afs/andrew/course/15/441-sp07/project3`

The support code has been slightly modified, but it should not affect your Project 2 implementations, and should link with your code seamlessly.

We reserve the right to change the support code as the project progresses to fix bugs :) and to introduce new features that will help you debug your code.

2.2 Groups

Since you will be extending the code written for Project 2, we assume that all groups will remain the same. If you need to change your group for whatever reason, you must come and talk to us by *Friday, March 23*, and no later.

You are not permitted to share code of any kind with other groups. Each member of the group is responsible for both sharing the work equally, and for studying the work of their partner to the point they understand it and can explain it unassisted. Please also indicate a breakdown of what each group member did (see Section 6).

3 Your Assignment

In this assignment, you will implement a reliable, in-order-delivery transport protocol. The transport layer provides the means for delivering data from one network application to another. Your transport protocol must support the following:

- Multiple simultaneous connections concurrently. This means you will need to multiplex and demultiplex using port numbers and use proper synchronization mechanisms to ensure thread-safety.
- Reliability. Your protocol must detect lost packets and retransmit them. You will implement a reliable sliding-window transport protocol that allows multiple packets in flight at the same time. This has better performance than a "stop-and-wait" protocol.
- In-order delivery. No duplication or reordering of packets as observed by the user.
- Flow control through window advertisement

3.1 Read Chapter 5 in the book

Although the book is optional for this course, it would be really helpful to have in order to help you get started on this project. If you don't have a copy, borrow a friend's or head over to Professor Kesden's office to check out his copy.

Section 5.1 (UDP) discusses how to multiplex and demultiplex using ports. You should start your transport protocol by implementing this.

Section 5.2 (TCP) discusses the details of the different components of TCP: Section 5.2.2 discusses the packet format for TCP, which you will use. Section 5.2.3 discusses connection establishment and termination. You will spend a large chunk of time getting this right, so read and make sure you understand this section before attempting to implement it. Section 5.2.4 discusses the sliding window mechanism. This is the second piece that you will spend a lot of time on. Note that you will be using a *packet-stream*, not a *byte-stream* as this section describes. This means sequence numbers will be associated with packets, not bytes and retransmissions will be at the level of packets. You do not need to worry about the sequence number wraparound problem in this assignment (make sure your initial sequence numbers are less than half the maximum value, and we will make sure to transfer less than two gigabytes per connection).

You do not need to worry about Nagle's Algorithm, adaptive retransmission, congestion control, or any subsequent chapters in the book, although these things may come in handy for the extra-credit section.

3.2 Timers

Your TCP implementation will need to use three timers: initial connect timer, retransmission timer, and close timer.

- The *initial connect timer* is used to detect connection failures. This timer is set when a call to connect is made and is canceled when TCP reaches the established state. If the timer expires, the connection should be closed.
- The *retransmission timer* is used to detect a packet loss. If this timer expires, then a packet or its ack has been lost. TCP should retransmit the packet with sequence number one greater than the last packet acknowledged.
- The *close timer* is used to catch any FINs being retransmitted. Chapter 5.2.3 explains this in more detail.

To implement timers you will use the `timeout()` and `untimeout()` functions defined in `include/systm.h`. Please refer to Section 9.1 of the simulator handout for details of the timer API. The prototypes are listed below as a reminder:

```
typedef void (*timeout_t)(void *);
void timeout(timeout_t, void *arg, int ticks);
int untimeout(timeout_t, void *arg);
```

Suggested timer values are: 30 seconds for the initial connect timer, 3 seconds for the retransmission timer, and 75 seconds for the close timer. Remember that a kernel tick in the simulator is 500 ms.

3.3 Hooks

You must register your transport protocol with the socket layer in order for it to receive requests based on system calls issued by user code. Use `sock_register_transport(&proto_tcp)` at initialization time. Please refer to Section 6 of the simulator handout for details. For this assignment, you do not have to implement the `sendto`, `recvfrom`, and `setsockopt` calls, so set these entry points to NULL.

Next, you must call your init function (i.e., `tcp_init()`) from init.c in the template/kernel directory.

3.4 Functions

The template/kernel directory has `tcp.h` and `tcp.c`. Function prototypes are given in these files for the main functions. You are allowed to change the functions, but you must keep the registered functions (`tcp_socket, tcp_close, tcp_bind, tcp_connect, tcp_accept, tcp_write, and tcp_read`) and their arguments the same.

3.5 Other important points

- You must bound the size of the *receive buffer*. The size of the receive buffer determines how much data transport layer queues at the receiving end, waiting for the data to be consumed by the application. The size of the receive buffer also defines the maximum *receiver window* advertised to the sender. Your receive buffer should be sized to hold somewhere betwee 16 and 256 packets.

- Your protocol should avoid retransmitting more data than is necessary. For example, assume a sender transmits data segments n, $n + 1$, $n + 2$, and data segment n is lost. The acknowledgment number sent by the receiver would remain at n (TCP's acknowledgment field identifies the next sequence number expected) even when data segment $n + 1$ and $n + 2$ are received (TCP uses a cumulative acknowledgment scheme). Upon receiving the acknowledgment for n, the sender should retransmit data segment n but *not* data segment $n + 1$ and $n + 2$.

- Conceptually, a receive buffer contains two types of segments: in-order segments and out-of-order segments. In-order segments contain data that are ready to be consumed by the application. Out-of-order segments are waiting for missing segments with lower sequence numbers, and are not ready to be consumed by the application. You can choose to put both types of segments in one receive buffer, or you can break them up into two different buffers. Regardless, the sum of both types of segments cannot exceed `TCP_RCVBUF_SIZE`.

- If the size of the data given by a `Write()` call fits in one transport segment, then send the data as one segment. If the size of the data is too large to fit in one segment, you need to break up the data into multiple segments. The size of segment is limited by interface MTU. Optionally you can pack data from multiple `Write()` calls into one transport layer segment, but this is not required. A `Write()` of 0 byte data is discarded by your network stack and is not sent over the wire.

- a `Read()` can read any number of bytes of data greater than zero. However, you are not required to return the exact size of data requested (the size can be $\leq$ the requested size). For example, let us assume you have two in-order segments in your receive buffer, each with 100 bytes. If an application performs two `Read()` calls, and each call requests one byte, you must return the first byte from the first segment and then the second byte from the first segment. On the other hand, if an application requests 120 bytes, you can choose to return either 100 bytes (the whole first segment) or 120 bytes (the first segment plus 20 bytes of the second segment). While in theory you could also return seven bytes, or eleven, or some other silly number, there is no defensible reason for doing so.

- For simplicity, the sequence number and acknowledgment number in the transport layer header will be units of "packet" rather than "byte". Thus, the transport protocol is a *packet-stream* protocol instead of a *byte-stream* protocol. However, to the application using your transport protocol, it will appear to be a *byte-stream* protocol since the protocol is not required to maintain packet boundaries. The size of the send buffer and receive buffer are defined in units of packets instead of bytes. These maximum buffer sizes (`TCP_SNDBUF_SIZE` and `TCP_RCVBUF_SIZE`, respectively) are defined in `<projdir>/include/systm.h`.

4 Testing your code

In order to test your code, we have provided a reference kernel with packet filtering capability. An ideal way to test the correctness of your TCP implementation would be to setup the standard 3 node network with the reference kernel in the middle.

The format of the filter file is as follows

node-id
filter-flags | packet-range

Here filter-flags refers to the TCP flags that the node should drop (ack, syn, fin). packet-range refers to the range of packets the node should drop. This range refers to any packets that traverse this specific node. So if you specify the range as 2 4, and the node first receives 1 packet from node 3, then receives 2 packets from node 2 and finally 1 packet from node 3, it will drop the two packets from node 2 and the last packet from node 3 only. Note that you can specify either flags or packet range, and the file must be exactly 2 lines long.

So lets say you setup the reference kernel as the middle node (1) in your simulation. If you want the node to drop packets 5-10, you would write

n1
5 10

If you wanted the node to drop all ack packets, you would write

n1
ack

To run the reference kernel with the filter, name your filter file tcpfilter.config and place it in the same folder as the reference kernel and run the kernel as usual. The kernel will automatically read the filter. A sample file and reference kernel has been provided in `<projdir>/refipkernel`.

5 Evaluation

We will be grading this project by scripts and an in-person demo. You will need to sign up for a 30-minute time slot. Late submissions will be handled according to the policy given in the syllabus. We intend not to extend the due date for Project 3. The point breakdown will be approximately as follows.

- **(60 points) Sliding-window transport**

  This part of your grade reflects how well you implemented the protocol features we listed above. We will check if your protocol can handle packet loss, duplication, reordering using test cases and the. Note that correctness is more important than performance. It is better to have solid working features than broken features with optimization.

- **(15 points) Style**

  Poor design, documentation, or code structure will reduce your grade by making it hard for you to produce a working program and hard for the grader to understand it. Compiling errors and warnings will also reduce your style credits.

- **(15 points) Demo**

  We will run a series test cases to check your implementations during the demo. You will be asked specific questions related with your design and implementations. This part of grade reflects how well you understand your implementation choices, and the code written jointly with your project partner.

- **(10 points) 2 Checkpoints**

  There will be 2 graded checkpoints during this lab. Each will be worth 5 points. A brief description is provided in the "Plan of Attack" section of this handout, but please watch the bboard for specific details about the each of the checkpoints as it's time approaches.

  - **(April 3rd) SYN**
  - **(April 12th) Stop and Wait**

6 Handin

6.1 Code Requirements and Restrictions

We will run your program on x86 computers running Linux. We recommend that you use similar machines for development. Such machines are available in the Wean clusters. Additionally, several Linux servers (unix44.andrew.cmu.edu through unix49.andrew.cmu.edu) are available for remote login. If you have your own Linux system, you are welcome to use it for this project. Note, however, that we will test your code on the Andrew systems. Thus, *you must make sure your code runs correctly on the Andrew Linux machines*.

You must write your code in C. In addition, your code must compile with `gcc` using the `-Wall -Werror` flag cleanly on an Andrew Linux machine without any warning message.

6.2 Project Writeup

Each group should create a brief report (README file) describing their efforts, in one of the following formats: plain text, postscript, or pdf. Your report should include the following:

- A description of the design of your sliding-window transport protocol. Note that if there are errors in your implementation of some functionality, we may still be able to give you credit for having worked through the design issues for that functionality.
- A description of what works and what does not (use a table for this). For things that do not work, give your thoughts on what the problem might be.

- A breakdown of what each group member did (use a table for this as well).
- Your thoughts on the project: was anything too difficult? What would improve the project? Were there parts of the project that worked particularly well, and shouldn't be changed?

Also, each group should create a file called TESTS with a description of the test cases you used, and any interesting testing strategies that you used.

6.3 Hand-in Procedure

You should submit the following files:

- Makefile, *.c, *.h
- Project writeup `README` and `TESTS`
- (optional) Code or documentation (`EXTRA`) on any extra credit items you have worked on (see Section 10).

All your submission files are to be placed in

`/afs/andrew/course/15/441-sp07/handin/lab2/andrewid1-andrewid2/`.

Use the same version notation as in previous labs. Note that *submissions by e-mail will not be accepted*.

Similar to Project 2, your `Makefile` should be written such that we can build your binary from source by simply running `make` (with no arguments) in your submission directory. The binary produced must be called `kernel`. If your code does not build according to this procedure, your submission *will* lose points. You do not need to include any library files (such as the C library, or the simulator's libraries) in your submission directory.

7 Resources and Hints

- **Start early!** Not only for your sake but for the common good. Read the handout, think about the issues, ask us questions. There is an inherent tension between giving you freedom of design and having to evaluate your submissions. Where ambiguities arise as to what we require, they are best resolved as early on as possible. If you find yourself plagued by doubts and frustrations in week 3 of a 4-week project, the fault lies partly with you.
- RFC 793 (`http://www.ietf.org/rfc/rfc0793.txt`) specifies the functions of TCP protocol as your reference of a sliding-window transport layer. Note while reading the TCP literature, you will come across many features not required by this project. So we suggest you be selective in reading the RFC. *Please start reading the RFC immediately, as it will take some time and you need to read some parts several times.*

- There are several things to note about the interface between transport layer and socket layer:
  - A server must call `Bind()` to bind to a port number before calling `Accept()`. If not, `Accept()` should return a failure. A `Bind()` to a client socket is optional, and thus you must support both implicit binding (`Socket` → `Connect` → ...) and explicit binding (`Socket` → `Bind` → `Connect` → ...) on a client socket.
  - `Accept()` returns `0` (instead of a new file descriptor as in UNIX) upon success, and `-1` upon failure. Thus, `Accept()` does not create a new file descriptor (unlike the Berkeley Socket specification), and uses the same file descriptor for the subsequent socket calls.
  - Since `Accept()` does not return a new socket descriptor, after `Accept()` returns and the incomming connection is handled, the socket will be destroyed upon `Close()`. The server needs to recreate the socket and bind it with the `Socket()`/`Bind()` sequence before listening for new connection by calling `Accept()` again.
  - The socket starts accepting client connection requests only *after* the `Accept()` call has been successfully made, i.e., SYN packets arriving before the `Accept()` call should be discarded.
  - `Write()` should return almost immediately, except in the case where the *send buffer* is full. This means that if your transport layer cannot deliver the data right away, you should queue the data in a send buffer for later transmission. However, if the send buffer is full, `Write()` blocks until enough space in the send buffer is freed to enqueue another packet.

8 Plan of Attack

The following is a suggested plan of attack intended to help you get started. While you may not choose to do everything in this order, it may provide you with some guidance. There will be one *milestone* and two *checkpoints* during the course of this project.

1. Verify your IP forwarding layer. Throughout the course of the project, we will release reference kernels agaist which you can test your TCP implementation. But first, you need to make sure that your IP layer works against ours. Initially, we will release a reference project 2 kernel that you can use to test. Please verify your IP layer by running a network simulation involving both your kernel and the reference kernel and make sure that they can communicate with each other and agree on all the bits in the IP header.
2. Write the `socket` call, which involves allocating and attaching a control block to the socket, and deallocate it in `close`. For now, your control block can have only two things in it, just the source and destination addresses. Have `bind` and `connect` fill in these addresses. Since you do not yet have port numbers, you can only support one socket per host. However, your protocol does have enough information to send packets, so implement a dumb version of `write` and see if you can correctly send a packet to the destination host. Verify that `tcp_receive` gets the packet that `write` sends out.
3. Implement a simple receive buffer. Note that the pbuf structure contains a nextpkt pointer that you can use to link packets together; therefore, you shouldn't have to write any queue structures. Have

`tcp_receive` enqueue to it while `read` dequeues from it. Also make sure that your receive buffer is thread safe. In addition, implement the signalling mechanism necessary for `read` to block when the receive buffer is empty, and wake up only when a packet has come in. Now figure out a way for `close` to signal 'EOF' to the other side. At this point, you should have enough functionalities implemented to perform a simple file transfer.

4. Add port numbers to your protocol so you can have more than one socket per host. Update your TCP-control block, as well as `bind` and `connect` so they know about port numbers. Make sure that `bind` assigns unique port numbers, and think about race conditions. Also, have `close` recycle the port numbers. Now, every packet being sent sent needs to have its the port number identified, so modify `write` so that it prepends a simple header identifying the port numbers. Modify `tcp_receive` so the port numbers are interpreted and used to demultiplex between sockets. (You will need a datastructure to map port numbers to sockets). Congratulations, you have implemented a complete transport layer protocol (a.k.a. UDP). Now your protocol should be able to handle two simultaneous transfers. **You should reach this milestone by March 27.**

5. From now on, you will be implementing TCP-specific features for reliable transport. Please take some time to read and re-read the textbook sections on TCP, the RFC, and this project handout. Also, it's probably a good time to implement the TCP header and checksums.

6. Implement three-way handshake for establishing a connection reliably. To assist you, we will provide a reference implementation for you to connect against, for use until your TCP implementation can connect with itself. **This is checkpoint 1, due April 3**

7. Implement connection teardown. You also need to properly report 'EOF' to the user according to the socket I/O semantics.

8. Implement reliable packet transfer using stop-and-wait. Now you should be able to transfer a file reliably (even with link-level packet dropping turned on). **This is checkpoint 2, due April 12.** - You will download a rogue kernel. A user program we provide will transmit a key (< andrew ID 1 > < group # > < andrew ID 2 >), to a TCP port on the rogue kernel, which will transmit back a hash string. Then you will send staff-441 the key and hash so we can verify you have completed the checkpoint.

9. Implement sliding window to allow multiple packets to be in flight. Now you will need a send buffer as well as a receive buffer. First, only accept packets that are in-order and silently drop out of order packets. Second, have an out-of-order queue to re-order packets as they come in. (This step will take the rest of the time)

To summarize, there will be one milestone and two checkpoints, and the key dates are as follows:

1. March 20 - Release
2. March 27 - *Milestone:* UDP-like transport
3. April 3 - **Checkpoint 1**
4. April 12 - **Checkpoint 2**
5. April 19 - **DUE!!!!**

9 Getting Help

- Most questions will be of general interest and should be posted to the class bboard. Please make your questions clear and specific to increase the chance that we can solve your problem with one response. You are responsible for reading the bboards to stay up-to-date. We will assume that all students in the class will read and be aware of any and all information posted to the bboards.
- If you have a question for the TAs – a question that is not appropriate for the bboard – please email your question to *staff-441@cs.cmu.edu*. As always, the course staff is available for help during office hours.
- Talk to your classmates. While you need to write your own original program, we expect conversation with other people facing the same challenges to be very useful.
- Come to office hours. This is particularly useful if you have questions about how to structure your code, or questions about other aspects of your design.

10 Extra Credit

Our intent in suggesting extra credit items is to give interested students the opportunity to explore additional topics in depth that will not be covered in project requirement. The primary reward for working on the suggested items is the additional experience and knowledge that they give you, *not* extra credit points. Extra credit will be granted at the discretion of the teaching staff.

If you work on the suggested topics below, please include in your project submission a file called `EXTRA`, describing what you have done.

Transport layer congestion control There are many features related to TCP congestion control that can be implemented based on a sliding-window transport layer. These include *slow start, congestion avoidance, fast retransimission, and fast recovery*. Implement these options in your sliding-window transport layer, and compare the protocol performance with the one without these features.

Two way connect TCP allows the two parties to both connect at the same time.

Test cases We encourage you to come up with interesting test strategies for checking your work.

Your Own Idea We welcome your suggestions for other interesting extensions to the project.