# 15-441 Fall 2004: Project # 3
# Reliable Sliding-Window Transport

Out: Fridav, Oct 29th, 2004
**Due: Friday, Dec 3rd, 2004, 11:59 pm**
Lead TA: Aditya Ganjam

## 1 Introduction

In the previous projects, you implemented an application layer service (TFTP), 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 Project 2 code is hopelessly broken, please bring it to our attention as soon as possible. The routing daemon does not need to be working to complete this project.

## 2 Logistics

### 2.1 Handouts

All files for this project can be retrieved from

`/afs/cs/academic/class/15441-f04/project3`

In the rest of the document, we refer to this directory as `<projdir>`.

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, November 5th*, 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 5).

# 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. This means you will need to multiplex and demultiplex using port numbers
- 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

5.1 (UDP) discusses how to multiplex and demultiplex using ports. You should start your transport protocol by implementing this. 5.2 (TCP) discusses the details of the different components of TCP: 5.2.2 discusses the packet format for TCP, which you will use. 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. 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.

You do not need to worry about Nagle's Algorithm, adaptive retransmission, congestion control, or any subsequent chapters in the book.

## 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 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`.

```
timeout(timeout_t ftn, void *arg, register in ticks);
untimeout(timeout_t ftn, void *arg);
```

`ftn` is a pointer to the function the timer should call when the timer expires. `arg` is the pointer to the argument sent to the function. `ticks` is the duration of the timer in 500ms ticks.

Suggested timer values are: 30 seconds for the initial connect timer, 3 seconds for the retransmission timer, and 75 seconds for the close timer.

### 3.3 Hooks

You must implement a number of hooks to get your TCP implementation to talk to the kernel. (These should be done for you and are mostly for reference)

First, you must register your transport layer by calling `sock_register_transport(&proto_tcp);`, where proto_tcp is a struct of type transport_proto defined in `include/ksocket.h`. This struct defines the properties of this transport protocol (AF_INET, SOCK_STREAM) and the functions that it supports. The files must be filled in with the functions names or NULL if no function is defined. You will not be implementing sendto, recvfrom or setsockop for TCP, so these fields in the struct will be NULL. All others must have valid function pointers.

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. For the receive window, you should use a value between 16 and 256.

- 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 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 read by the application. Out-of-order segments are the ones waiting for missing segments with lower sequence numbers, and are not ready to be read 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. 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, each call requests one byte, you must return the first byte and then the second byte in the first segment. On the other hand, if an application requests 120 bytes, you can choose to return either 100 bytes (the first segment) or 120 bytes (the first segment plus 20 bytes of the second segment).

- 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`.

### 3.6 Tracing the Transport Layer

In real UNIX implementation, the TCP Control Block (TCPCB) maintains the state of a TCP connection. As an example, we provide you with a sample TCPCB data structure `p_tcpcb` in

`<projdir>/tmplate/kernel/tcp.h`

for your sliding-window transport layer. Note you are free to define your own transport layer control blocks.

For grading purposes and to help you debug your sliding-window transport protocol, we provide four functions (defined in `<projdir>/include/systm.h`) that you can use to generate a transport layer trace file. The trace file name is `tcp<node_id>.trace`. The trace file will also help us understand your code if some problem arises. So please insert these functions in the appropriate places of your code. The first three function calls are mandatory (for our grading purpose) and the last function call is optional. Note that you don't have to use the exact data structure we provided in the sample `p_tcpcb`, but the arguments to the functions should have the same semantics. Since we will use the generated traces for grading, inappropriate use of the required functions will be penalized.

- `print_tcpcb()`: call this function whenever there are changes to the relevant fields in the TCPCB data structure (such as send and receive window). The relevant fields are listed in this function's arguments.
- `print_tcp_snd()`: call this function whenever you send a transport layer packet.
- `print_tcp_rcv()`: call this function whenever you receive a transport layer packet from the IP layer.
- `print_tcp_debug()`: call this function whenever you want to insert your own trace information to the trace file. This function takes the same type of argument list as `printf`.

# 4 Evaluation

We will be grading this project by demo. You will need to sign up for a 20-minute time slot. See the project web page for detailed information. Late submissions will be handled according to the policy given in the syllabus. Note there will be NO extensions to this project!

- **(40 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 generated transport layer traces. Note that correctness is more important than performance. It is better to have solid working features than broken features with optimization.

- **(10 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.

- **(10 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.

# 5 Handin

## 5.1 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-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 implement your code in C or C++. You may only use the following libraries: `libc`[1], `libstdc++`[2]. (You must contact us in advance if you want to use other libraries.) Your code must compile with `gcc` or `g++` using the `-Wall` flag cleanly on an Andrew Linux Machine without any warning message.

### 5.2 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.

### 5.3 Hand-in Procedure

You should submit the following files:

- Makefile, *.c (or *.cc), *.h
- Project writeup `README` and `TESTS`
- (optional) Code or documentation (`EXTRA`) on any extra credit items you have worked on (see Section 8).

All your submission files are to be placed in `/afs/cs/academic/class/15441-f04-users/group-<groupid>`. These folders will be locked after submission deadline. Note that *submissions by e-mail will not be accepted*.

[1] Specifically, the GNU C Library.
[2] The GNU C++ library. This includes the Standard Template Library.

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.

# 6 Resources and Hints

- The most important advice we can offer is to finish the forwarding layer from Project 2 before moving on to project 3. This means ensuring that your forwarding layer implementation is relatively bug-free.
- **START NOW!** 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 partly lies on 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 refereed RFC document.
- 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.
  - The socket starts accepting client connection requests only *after* the `Accept()` call has been successfully made, i.e. 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.

# 7 Getting Help

- If you have any question regarding this project, please post your question 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 on these changes. 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.

# 8 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.

**Half-open connections** When one end of a TCP connection gets a close() call, it sends a FIN to the other end. The other end can continue to send data until it gets a close() call.

**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.