CSCI 4969/6966 Project 2

Reduced TCP Implementation and Timing

Team Choices closed Sunday October 29th, 11:59:59 PM.

Mid-Report due Monday November 13th, 11:59:59 PM. Full project due Thursday November 30th, 11:59:59PM.

This assignment has one "mid-report" deliverable and a deliverable at the end. Please read over the mid-report requirements carefully. This is a *team assignment*. Teams were already assigned this time around, but for future iterations I've left the team formation part here.

Team Formation

On Submitty there should be an additional button for Project 2 Mid-Report that says "CREATE TEAM". You can use this to invite someone to a team or manage existing invites. Teams should contain exactly 2 people. If you are not on a team after team submissions close, you will be manually assigned to a team. Since we have an odd number of students, you should still form groups of two but requests can be sent to be a team of 3. These requests should include the user names of all 3 team members. I will keep these in mind when doing manual assignment of any unassigned students.

Only one submission needs to be made per team. You should all be able to see the team submission the same way you would submit a normal assignment. You do **not** need to designate a team leader to handle all submissions. Your teams will automatically be duplicated for Project 2 Final Deliverable when that gradeable is created.

Reduced TCP Implementation

The goal is to make a new TCP implementation that can replace the existing one, to keep our additions visually distinct we will use the prefix **rpitcp** instead of just **tcp**. Our implementation will be fairly minimalistic; you should only implement the features that are required. Your implementation should be done in separate files, and use its own class names but should provide an alternate interface for *inet_connection_sock* that allows an application to invoke your implementation of TCP without any application changes.

You should define your own structures and ultimately your TCP implementation should work when interacting with other hosts/programs that use TCP, even if they advertise TCP options you don't support. You should interact with the existing Berkley sockets and INET sockets, both in terms of general "workflow" and memory usage. You should primarily be using *sk_buff* and pulling from the correct memory pool, ensuring you don't violate the socket read/write memory restrictions.

It is recommended that you first focus on getting your implementation to work with a client and server that you wrote (Assignment 2 may be useful here). You should be able to correctly handle the three-way-handshake, keeping track of ACK numbers, and closing correctly. Ideally the receive window should be adjusted as necessary, and congestion control should be implemented using TCP Tahoe and TCP Reno. You must also implement retransmission.

Do not implement advanced TCP features suggested in other RFCs: SACK, PAWS, F-RTO, Fastopen, etc. Your implementation should be compliant with RFC 793. If you use existing Linux code from the Linux TCP implementation you must justify the necessity of each line. You can account for a code segment in a single paragraph as long as each line is addressed, but you cannot copy a segment and simply say "these five lines handle memory allocation." You must also justify any structures, macros, members, and prototypes.

Once your basic implementation is done, you should focus on robustness. What happens if a connection stalls out? What if bad ACK numbers come in? What happens when you have many connections? What if there's a lot of data? What if there's a lot of congestion events or loss events? (Note: the two types of events will look the same to our implementation, since we are not supporting ECN.) You will also want to look at the socket options and see what you need to support, for example SO_LINGER and SO_KEEPALIVE.

Timing and Testing

Beyond the simple question of "can we implement this," and the additional question of "how much can we learn about the kernel networking subsystem through this hands-on approach," we also want to know how much of a performance impact we suffer. Another way to think of it is, how much of a performance gain do we get through advanced features that are implemented in the kernel. Since there are many features and it can be quite bothersome to target specific ones, you will take whatever timing tests you create and run them on RPITCP-Tahoe, RPITCP-Reno, Linux TCP-Reno, and Linux TCP-CUBIC.

The most direct testing is to do file transfers or server/client interaction such as web page requests, and simply get the system time before and after each transfer and log the differences in times. Another approach is to use the *time* command. A third approach is to use *Wireshark* traces. If you do not have a GUI you can also use *tshark* which is a terminal-based capture that outputs in a Wireshark-compatible format.

Some of your testing should involve lossy situations. You can either simulate loss inside the kernel (in which case I strongly recommend using sysctl parameters or some sort of procfs control) or through external means such as *qdisc* / *tc qdisc*. Network Emulation requires additional kernel modification, but is an option. Keep in mind that application-level losses will not result in TCP-level loss events. Make sure in your submission to explain any additional kernel configuration that is required, or any additional packages needed for testing! Also make sure when describing tests with losses to explain how you induced losses and what those parameters were. Also describe how these do or do not simulate real situations (you may want to look up terms such as "bursty loss" and "correlated loss").

As you write tests, be thorough. Write a sentence or two about the purpose of each test. Keep track of the results, both correctness and timing. Make sure to run each test on your RPITCP implementations and on Linux so that you can always compare the timing and behaviors. Describe what makes the test unique - if you do a bulk transfer from a third-party website, and then a bulk-transfer on localhost via FTP, those might have significantly different behaviors. If you have one instance of the VM up using RPITCP and one instance using Linux TCP this might be very different from two RPITCP versions.

The Mid-Report Requirements

The mid-report should be at least 2 pages double-spaced *per group member*. Code segments, screenshots, etc. do not count towards this length. Describe in detail all of the TCP features you support so far, the testing you've done so far (see the previous section for more details on testing), justification for any code you've "borrowed" from the Linux TCP implementation, issues that you've run into, and the remaining work you have to do before the full project is due.

The Full-Report Requirements

The full-report should be at least 3 pages double-spaced *per group member*. Code segments, screenshots, etc. do not count towards this length. Describe in detail all of the TCP features you support so far, the testing you've done so far (see the previous section for more details on testing), justification for any code you've "borrowed" from the Linux TCP implementation, issues that you've run into, and any functionality limitations of your code. You can reuse portions of your mid-report writing.

Tips

1. You might end up doing *printk* debugging fairly often. Be careful about overflowing the log buffer by printing too much too fast.
2. You may want to disable the check for updates (in Ubuntu with Gnome open Software & Updates, go to the Updates tab, and set "Automatically check for updates" to "Never" - this will help prevent TCP calls you didn't expect.
3. You might consider using version control for this project, but be careful not to add any directories like **.git** or **.svn** to your patch files.
4. You may want to look at some TCP extensions that you don't have implemented and test the situations they're supposed to address. This could expand your test cases and give you more understanding about why the implementation is so complex.

Grading

The mid-report submission will count for 35% of the project grade with 25% of the total coming from writing quality and the remaining 10% being for non-trivial progress with code and testing. The mid-report will **not** be peer reviewed.

The full-report submission will account for the remaining 65% of the grade with 25% being writing quality, 25% being depth of testing and test documentation, and the final 15% being for succesfully implementing RPITCP with both congestion avoidance algorithms. The full report **will** be peer reviewed.

Mid-Report Submission to Submitty

1. **Mid_report.pdf** is the primary part of the submission, and should follow the requirements outlined above.
2. **Mid_patch.diff** is your patch file for your progress so far. It does not need to compile or run perfectly, it's there to show your progress so far. Give some sort of unique identifier to *EXTRAVERSION* in the **Makefile** such as a dictionary word. "-project2" is probably not going to be unique.
3. **Mid_README** should include the division of labor so far (who did what, how much time everyone's spending), and if it's unequal discuss how you'll rebalance it for the remainder of the semester. Everyone should be putting in an approximately equal amount of work. This file can also include any notes you want the instructor to read that you wouldn't want a peer reviewer to read.
4. Any optional files, including any Wireshark traces.

Full Project Submission to Submitty:

1. **Full_report.pdf** is the full report as described above. Do not put your name on the report.
2. **Full_patch.diff** is your patch file for your full submission. Do not put your name in the code or *EXTRAVERSION* of the **Makefile**, but do pick an *EXTRAVERSION* that is unique with the same guidelines as in the mid-report submission.
3. **Full_README** should include a final division of labor (who did what, how much time everyone spent). Ideally this division should be approximately equal. It should also include any notes that you want the instructor to read that you wouldn't want a peer reviewer to read.
4. Any optional files, including any Wireshark traces. You should submit at least one Wireshark-readable trace demonstrating normal operation of local RPITCP against remote TCP server, and of local TCP against remote RPITCP.