15-441: Computer Networks

Recitation 5

P2 Lead TAs: Kartik Chitturi, Ines Potier

Agenda

1. Introduction to Project 2
2. CP1 Starter Code
3. Checkpoint 1
4. Using Vagrant
5. Packet Captures & Wireshark
6. Q&A

Project 2: TCP in The Wild

You've learned about TCP, the Internet's transport protocol.

In Project 2, you'll first implement a transport algorithm very similar to TCP Reno, and then design your own congestion control algorithm!

Project 2: Quick reminder

CP	Grade	Deadline
1	33%	Oct 16
2	33%	Oct 30
3	33%	Nov 6

The deadlines and weights are the same for 441 and 641 this time.

Start early! Do not wait until the last day!

Relevant Textbook Sections

The textbook (*Computer Networks: A Systems Approach*) contains very useful information for Project 2.

For CP1, read **Section 5.2** carefully, especially the on adaptive retransmission (ref. 1,2)

For CP2, read **Section 6.3**, on TCP Congestion Control (ref.3)

CMU-TCP

1. You will build your TCP protocol on UDP sockets, which are not reliable or ordered, have no control over how fast data is transmitted, and do not establish a connection.
2. You will augment those UDP sockets with the missing features to create a reliable, ordered transport protocol with connection establishment and termination.
3. You will implement a congestion control algorithm on top of the transport protocol

Checkpoint 1
Checkpoint 2 + 3

Starter Code

The */15-441-project-2* directory in the handout includes:

1. ***/build*** - empty (build artifacts like .o files go here)
2. ***/inc*** - contains the header (.h) files for the included code
3. ***/src*** - contains the starter code source files
4. ***/tests*** - empty (you will need to write your own tests)
5. ***/utils*** - includes a Lua plugin and shell script for generating packet captures
6. **Makefile** - compiles your project
7. **gen_graph.py** - a Python script to generate graphs from packet captures. (Not needed for CP1)
8. **readme.txt** - empty (describe your project here)
9. **tests.txt** - empty (describe your tests here)

Core TCP Files (writeup pg.3)

1. **cmu_tcp.c/h** - main socket functions required of your TCP socket
   a. Don’t change the function signature of the 4 core functions (socket, close, read, write)
   b. You should change the implementation of the 4 functions and can add more helper functions

2. **backend.c** - the code to emulate the buffering and sending of packets

Important Helper Files (writeup pg.3)

1. **cmu_packet.h** - *DO NOT MODIFY*! Contains basic packet format and header
2. **grading.h** - *DO NOT MODIFY*! Useful constants for your protocol, but will be changed in our testing, so you shouldn't be hard-coding or relying on specific values of these
3. **client/server.c** - Applications using the client and server sides of your protocol. Utilize these for testing but don't place anything important here, as we will not use these for our tests. (but tests will be similar to this).

What the Starter Code Does:

The starter code implements **Stop-and-Wait** transmission!

- It transfers data *reliably* and *in order*
- It is very slow!
- It does not have a handshake or a connection teardown
  - (if 1st or last packet is lost, transfer is incorrect)
- It recovers from loss very slowly (uses a fixed retransmission timeout (RTO) of 3 seconds)

Checkpoint 1 *Due Oct 16, 2019*

What you need to do (writeup - pg 2-5, sec 4):

1. Implement the TCP Handshake and Teardown (ref. 1)
    a. Handshake before data transfer starts, Teardown when the connection is terminated
    b. Should happen in the constructor and destructor for cmu_socket
2. Implement improved RTT (round-trip time) estimation
    a. Implement adaptive RTO by estimating RTT with **either** Jacobson/Karels Algorithm or Karns/Partridge algorithm (ref. 2)

Task #1: You will implement logic for connection establishment and connection termination.

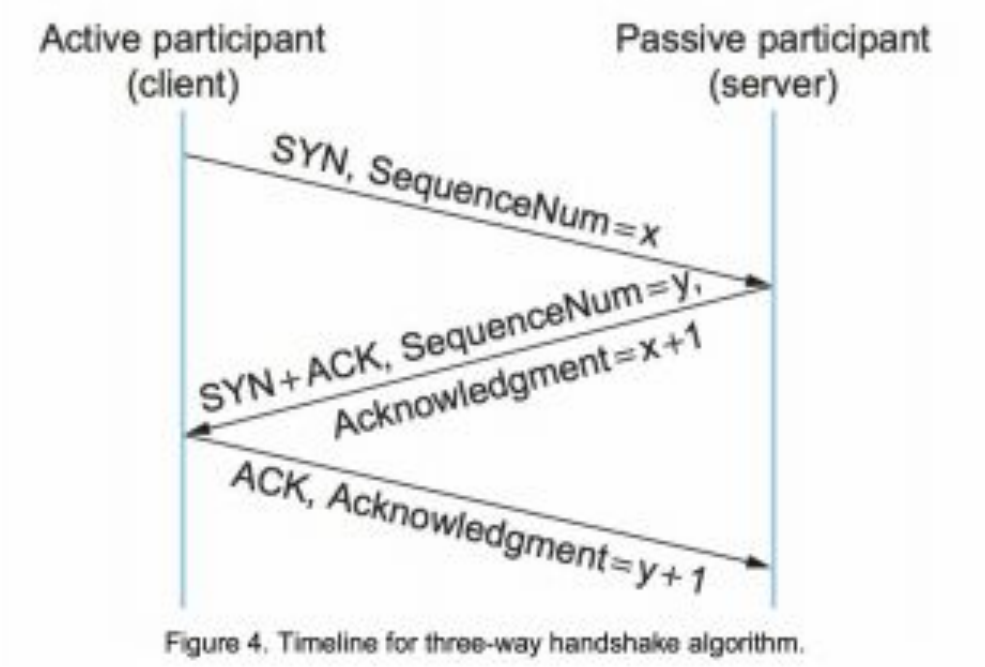
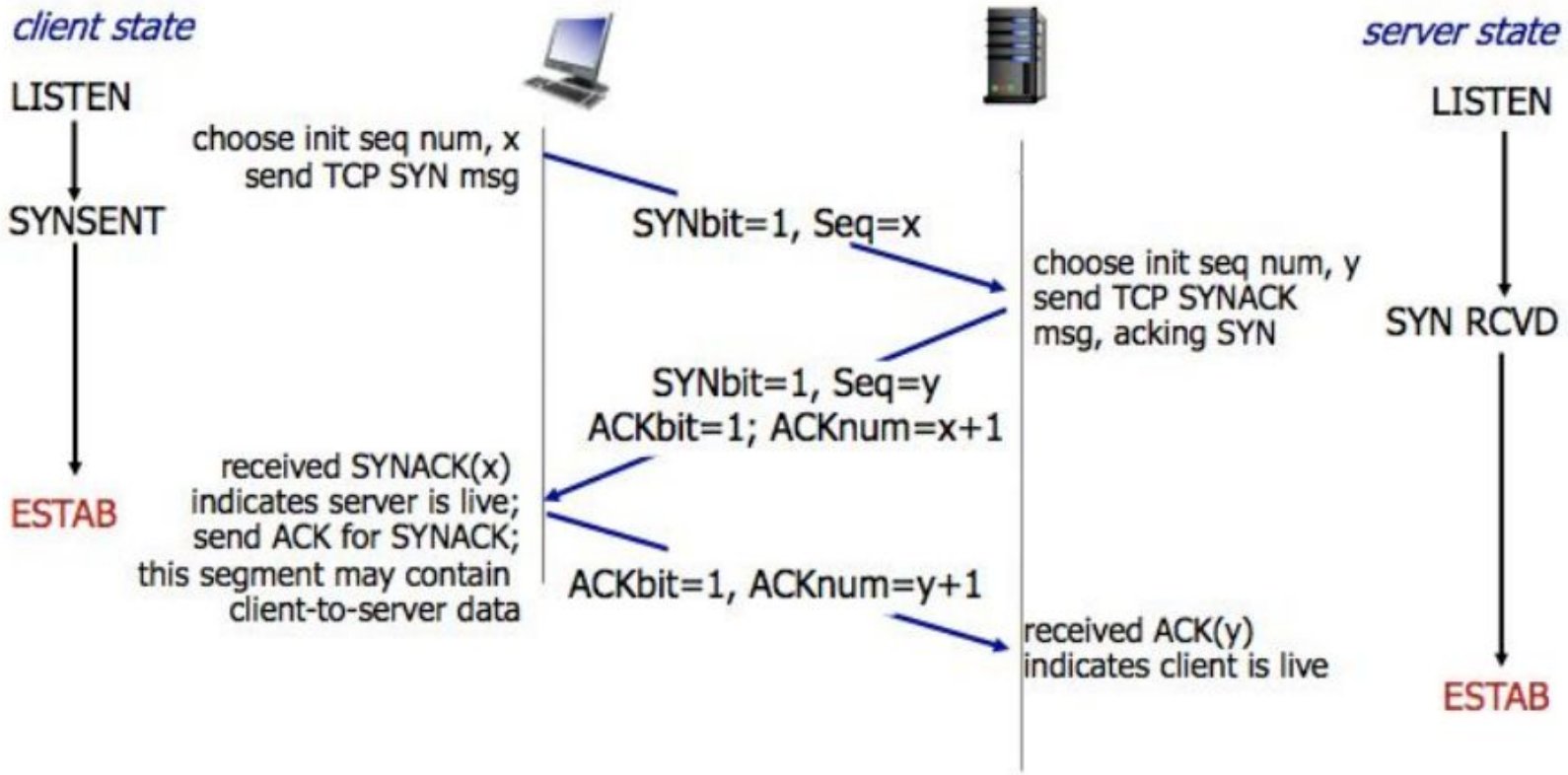
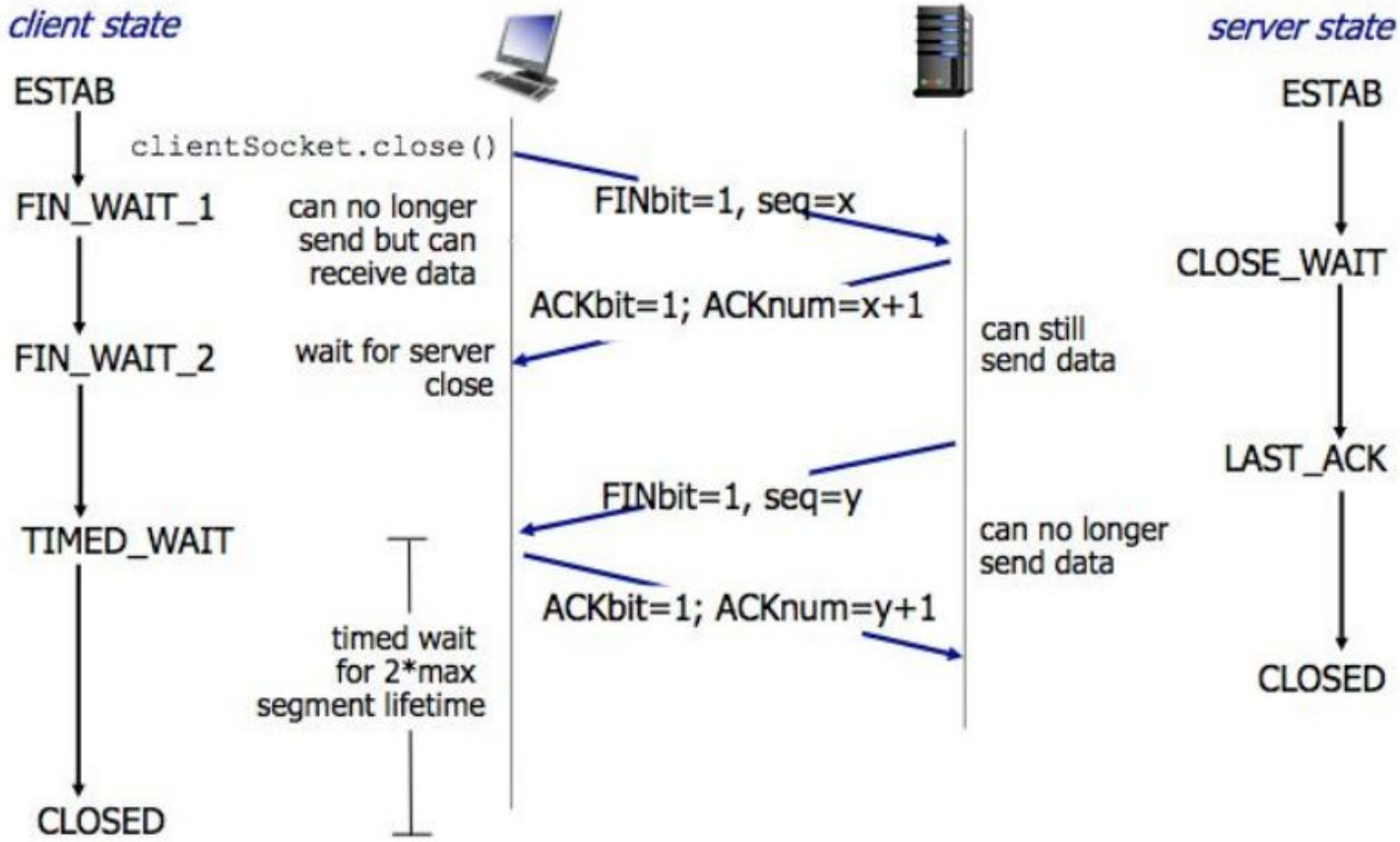


Figure 4. Timeline for three-way handshake algorithm.

TCP 3-way handshake



TCP: closing a connection



Task #2

You will implement Adaptive Retransmisson

- TCP retransmits each segment if an ACK is not received within a certain period of time
  - This is hard-coded in the starter code as 3 seconds
- TCP timeout is a function of the round-trip time (RTT)
- You will use Jacobson/Karls Algorithm or Karn/Partridge Algorithm to implement adaptive retransmission
  - Both of these are described in the textbook (ref. 2)

CP1 Hand-In Requirements

Read Section 7 of the write-up very carefully!

1. Your handin must be a git repo, with the relevant commit tagged with *checkpoint-1*.
2. Your submission tarball should include a top-level directory which must be named *15-441-project-2*
3. That directory will contain your Makefile, readme.txt, tests.txt, graph.pdf and all your source code, organized as they are in the starter code. (Do not submit .o files or executables!)
4. Make sure you compile your code with the gcc flag "*-fPIC*" in your Makefile
5. No files should starter with the word "grader"

Using Vagrant

You will do all of your development and testing *locally* on your own machine using VMs. You will need to install **Vagrant** and **VirtualBox** on your machine. (ref. 4, 5)

The starter code handout also includes a *Vagrantfile*, that defines 2 VM's, 'client' and 'server'. Anything in the same directory as the *Vagrantfile* (which is */15-441-project-2* in the handout) will be synced to the */vagrant* directory in both VM's.

Using Vagrant

Important Vagrant commands to know:

1. ***vagrant up*** - will initialize/boot up the VMs
2. ***vagrant ssh client/server*** - SSH into the client/server VM
3. ***vagrant suspend/halt*** - sleep/power down the VMs when you are done with development
    a. If you shut down your local machine, use *halt*. If your machine is just going to sleep, use *suspend*. Otherwise, you can corrupt your VMs!
4. ***vagrant destroy*** - will completely delete the VMs

Packet Captures

You will want to capture packets sent between the VMs during your tests to analyze and debug your code. 2 tools are installed on the VMs to help you do so: *tcpdump* and Wireshark (in its CLI version *tshark*). (ref. 6,7)
Also, in the */utils* directory of the starter code, we provide 2 files to help with capturing packets:

1. *capture_packets.sh* - A simple program to start and stop packet captures, and analyze the result using *tshark*
2. *tcp.lua* - A Lua plugin so Wireshark can analyze the custom CMU_TCP packet format

Using *capture_packets.sh* & Wireshark

capture_packet.sh provides 3 commands:

1. *start <name>.pcap* - start capturing packets into the given file
2. *stop <name>.pcap* - stop capturing packets into the given file
3. *analyze <name>.pcap* - uses Wireshark to analyze the provided pcap file and generates a CSV containing all the headers of captured packets.

Example

To capture packets using the starter code *client/server.c*:

Server VM

```
vagrant@server$ make
vagrant@server$ utils/capture_packets.sh start cap.pcap
vagrant@server$ ./server
```

Client VM

```
vagrant@client$ ./client
```

Server VM

```
vagrant@server$ utils/capture_packets.sh stop cap.pcap
vagrant@server$ utils/capture_packets.sh analyze cap.pcap
```

This will generate a *cap.pcap* file and a CSV file of the header fields of each captured packet.

Using Wireshark GUI

You can also install the GUI-based version of Wireshark to analyze the PCAPs with a better UI.

The provided Lua plugin */utils/tcp.lua* needs to be placed in Wireshark's plugins folder (ref. 8), and then you can open the .pcap file you created in Wireshark to see all captured packets.

Q & A

References

1. https://book.systemsapproach.org/e2e/tcp.html#connection-establishment-and-termination
2. https://book.systemsapproach.org/e2e/tcp.html#adaptive-retransmission
3. https://book.systemsapproach.org/congestion/tcpcc.html
4. https://www.vagrantup.com/intro/getting-started/index.html
5. https://www.virtualbox.org/
6. https://www.wireshark.org/docs/man-pages/tshark.html
7. https://linux.die.net/man/8/tcpdump
8. https://www.wireshark.org/docs/wsug_html_chunked/ChPluginFolders.html
9.