An Investigation of Split TCP On a Typical Network

By Zachary Homans, Elizabeth Mahon, and Brendan Ritter

Abstract:

Split TCP is a common strategy used to increase network performance by sending external requests through a proxy client/server. It is most commonly used in environments where round trip times are very large or the packet loss rate is very high. Past studies have sufficiently analyzed these target scenarios. Our goal was to test the performance of split TCP in an environment lacking these severe disadvantages. To do this, we constructed our own proxy and corresponding endpoint code, and used these to run a series of tests to determine the effect of split TCP on an average network. We found that split TCP appears to be beneficial even on a typical network, particularly in cases where very large files are being transferred.

Introduction:

TCP is the vital connection layer that lies beneath the Internet on the transport layer; it is utilized in such applications as including the World Wide Web, E-mail, FTP, Secure Shell, P2P file sharing, and streaming media. It ensures that connections are reliable, utilizing various techniques to deal with any packets that may be lost and packets that may arrive out of order. It does this by issuing a single TCP request to a host for a single large chunk of data, instead of numerous smaller IP requests. TCP detects any of the aforementioned problems, requests retransmission of unreceived data, rearranges misplaced data, and even helps minimize network congestion to reduce the occurrence of the other problems. TCP then passes on this complete and correct data onto the application layer; thus, the application layer does not need to know TCP's actions at all.

However, some of these techniques reduce its effectiveness on unreliable connections; in severe cases, TCP can end up in a perpetual state of congestion avoidance, which generally results in greatly diminished download speeds. For example, TCP solely attributes dropped packets to network congestion, even when they are caused by something completely different such as link failures, which are common in mobile devices. In those cases, one can potentially use "split TCP", which turns one long connection into two or more smaller connections. This may result in more reliable and faster connections. Longer connections require longer transfer time and have more potential for link failure, and conversely, shorter connections require less transfer time and have less of a potential for link failure. Thus, if we send data over two short connections rather than one long one, not only is there less of a chance for link failure, but even if there is, the data and acknowledgement only have to travel half the distance to get to the desired location. Previous studies on mobile ad hoc networks have shown that the use of split TCP has increased network throughput by 30% in typical situations[1], while other studies on high

[1] Swastik Kopparty, S. V. (n.d.). *Split TCP for Mobile Ad Hoc Networks.* Riverside, CA: Department of Computer Science and Engineering, University of California, Riverside.

loss, high Round Trip Time (RTT) networks have resulted in up to an 850% increase in network speed by using split TCP.[2] Even Cloud services could reduce latency by 43% through the use of split TCP.[3]

Our goal is to further test and hopefully confirm the desired network performance increased promised by heralds of split TCP. Unlike most studies, we wish to test split TCP in a completely normal, low loss, non-mobile environment to see if there is any significant performance increase. To do this, we created an opaque proxy in C that would retrieve files given a command, effectively splitting what could be done in a single long connection in two or more smaller connections. Using this proxy, we then ran a series of experiments to test the impact of the split on performance.

Building the Proxy:

In order to test the performance of split TCP, we first created an opaque proxy from scratch, instead of using a prebuilt one. The reason for this was three-fold. First, this exercise is intended to be an educational one; by building our own proxy, we learned a great deal about C programming, especially in relation to networks. Second, using prebuilt code is often difficult from a conceptual standpoint; it's easy to use, but difficult to understand exactly what it is doing. By building it ourselves, we guarantee that we understand the code that we are using. Third, it's possible that using prebuilt code could result in unnecessary code functionalities; we desired to construct a minimal opaque proxy, one that only did exactly what we desired in the least amount of space possible.

Our final split TCP code was built in three discrete pieces; each piece of code was run on a different laptop. For the remainder of this paper, we will identify each laptop in the follow way. One endpoint that allows the user to decide what file they are trying to get, sends the command, and receives the file. This laptop and corresponding code segment will be known as "A". The other endpoint takes the command and retrieves the file, either from the Internet or locally (subsequently, there are two variations of this segment). This laptop and corresponding code will be known as "C". Note that C isn't necessarily an "endpoint," as it still sometimes talks to the Internet; in this case, it is acting as an opaque proxy. Finally, a dedicated opaque proxy sits between the A and C and forwards both the command and the file data. This laptop and corresponding code will be known as "B". Every individual piece of code utilizes TCP sockets. Overall, in an Internet request scenario, A B, and C all act as clients, and B and C act as servers, while in a local request scenario, A and B act as clients, and B and C act as servers. The full code for all three pieces are located below in the Appendix.

Experiment Methodology:

To determine the effects of split TCP on a typical network, our experiment consisted of a series of seven different scenarios, each connecting the above three modules in different ways. These scenarios differ by the inclusion or exclusion of B and C, and by the type of connection between laptops, either wired or wireless. For the wired connections, we used a crossover ethernet cable

[2] Rahul Jain, T. J. (n.d.). *Design and Implementation of Split TCP in the Linux Kernel.* Newark, NJ: Computer Science Department, New Jersey Institute of Technology.
[3] Abhinav Pathak, Y. A. (n.d.). Measuring and Evaluating TCP Splitting for Cloud Services. Purdue University, Polytechnic Institute of NYU, Microsoft Corporation.

to connect our laptops. Note that because our laptops only have one ethernet port, we were unable to do other possible scenarios, such as connecting all three laptops with two cables. We also had issues connecting Module C to another laptop through the cable and to the Internet at the same time. However, we feel that our testing suite is sufficient enough to give us the evidence required for our purposes.

The testing suite is as follows:

- Scenario 1: A connects to C through the cable. C locally retrieves a file.
- Scenario 2: A wirelessly connects to C. C wirelessly connects to the Internet.
- Scenario 3: A wirelessly connects to C. C retrieves a local file.
- Scenario 4: A wirelessly connects to B. B wirelessly connects to C. C wirelessly connects to the Internet and retrieves a file.
- Scenario 5: A wirelessly connects to B. B wirelessly connects to C. C locally retrieves a file.
- Scenario 6: A wirelessly connects to B. B connects to C through the cable. C locally retrieves a file.
- Scenario 7: A connects to B through the cable. B wirelessly connects to C. C locally retrieves a file.
- Control/Scenario 8: A connects wirelessly to the Internet.

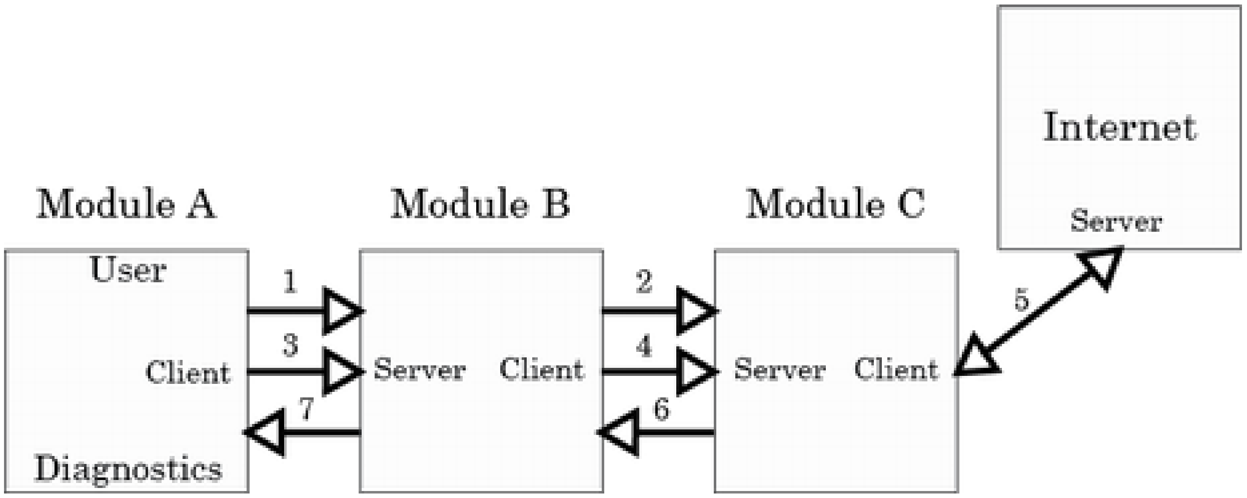


Figure 1: Diagram of Scenario 4. 1: Client A connects to Server B. 2: Client B connects to Server C. 3: Client A sends user request to Server B. 4: Client B forwards user request to Server C. 5: Client C processes request and obtains data. 6: Server C sends data back to Client B. 7: Server B forwards data to Client A. Every scenario is similar; they differ by connection type, locality, and the amount of splits.

We believe that this testing suite will serve our goals for the following reasons. First, we constructed a control scenario, along with several scenarios with one split and several scenarios with two splits. Not only will this allow us to see whether or not split TCP is beneficial, but whether the effects are enhanced when more splits are used. Second, by using the crossover cable in several scenarios, we hope to measure the effects of removing a possible wireless jump between computers that could prove significant. For example, if our results showed that going through a single split boosted performance when the proxy was connected by the cable, but not otherwise, it would allow for further and more specific exploration into the subject.

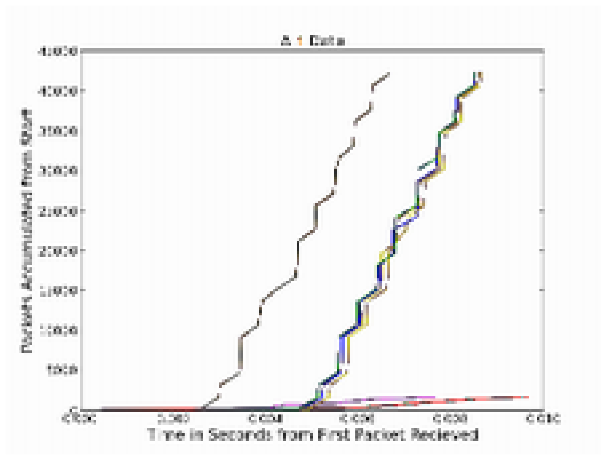
To begin each test, we started the network code on C (if applicable), then B (if applicable), and finally on A. To ensure that our tests were long enough that typical variance would not affect our results unduly, we used a big cat picture:

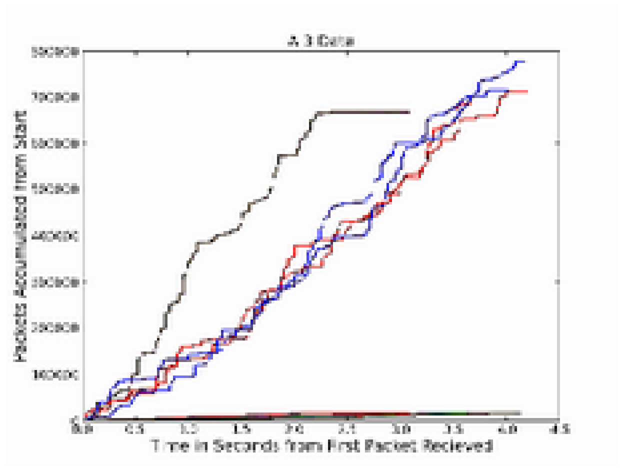
(Source: http://www.hdwallpapers.in/walls/african_lion_king-wide.jpg)
Figure 2: Our download picture (763 KB).

In order to analyze each scenario, we also needed to record the information relevant to the data transfer. To do this, we ran a background process consisting of the Linux command tcpdump on each computer to monitor all TCP data going in and out of the appropriate port. This allowed us to save the time and sizes of packets being sent across the port. We then cleaned the data and used two Python scripts, located in the Appendix. The first modified the data into a usable format, and the second graphed the cumulative data sent against time.

Results and Analysis:

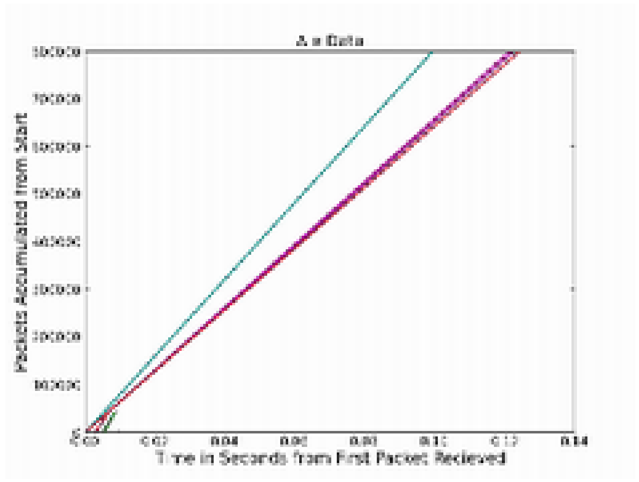
Wired vs. Wireless: Scenario 1 vs. Scenario 3





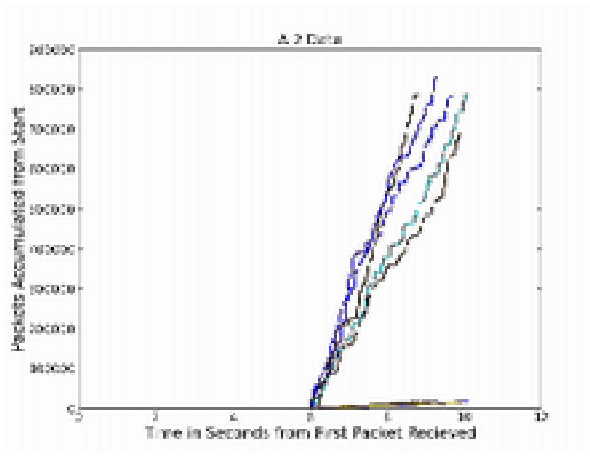
Scenarios 1 and 3 were the same test in terms of split TCP, with neither really functioning as a split network - A was simply a client and C was simply a server retrieving a local file. They only different by connection type - Scenario 1 was wired and Scenario 3 was wireless. This test meant to serve as a control, to confirm that wired connections perform better than wireless ones.

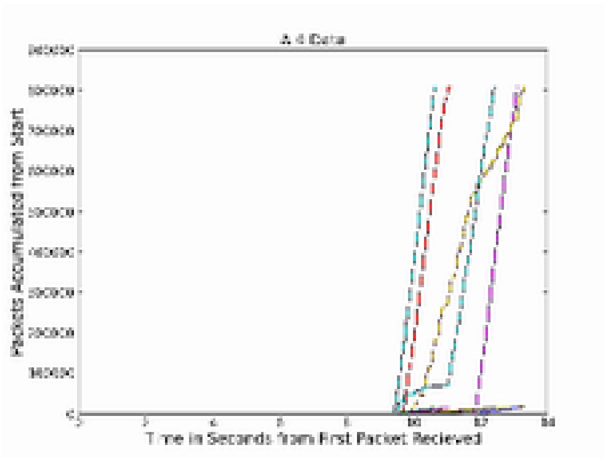
Unsurprisingly, this connection difference made a very large impact, as Scenario 1 completed several orders of magnitude faster than Scenario 3. However, this gap might not be as large as depicted. There appears to be an error in the data collection for Scenario 1: the amount of total data received is far less in Scenario 3 than in Scenario 1 by an order of magnitude. Regardless of this error, if we extrapolate the rate observed in Scenario 1 to the appropriate file size observed in Scenario 3:

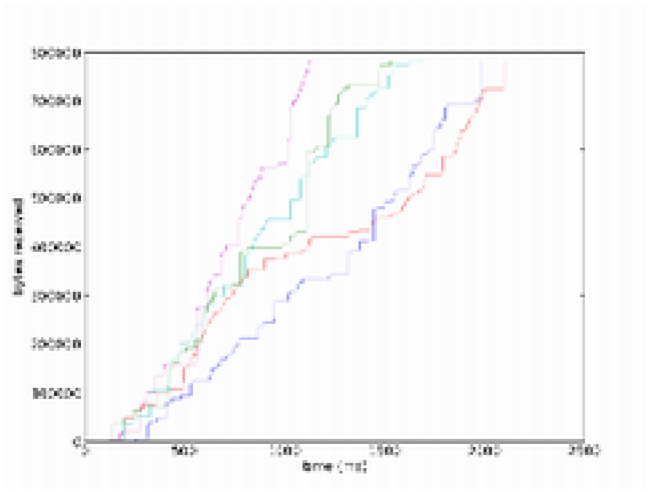


Scenario 1 would still be substantially faster, as predicted.

Presence of Intermediates (B & C) with external file: Scenario 2 vs. Scenario 4 vs. Scenario 8







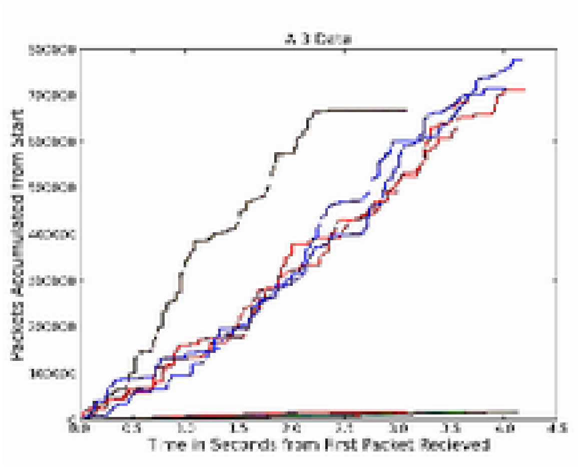
All three of these scenarios were done across only wireless connections. This includes C's connection to the Internet to retrieve our desired data. Note that Scenario 8 uses a different piece of code than the rest of our scenarios, and thus the graph is slightly different; it is a simple measurement of a standard download, obtained through our professor Allen Downey as part of his Software Systems course. The code can be found at: http://allendowney.com/ss12/wget.tgz.

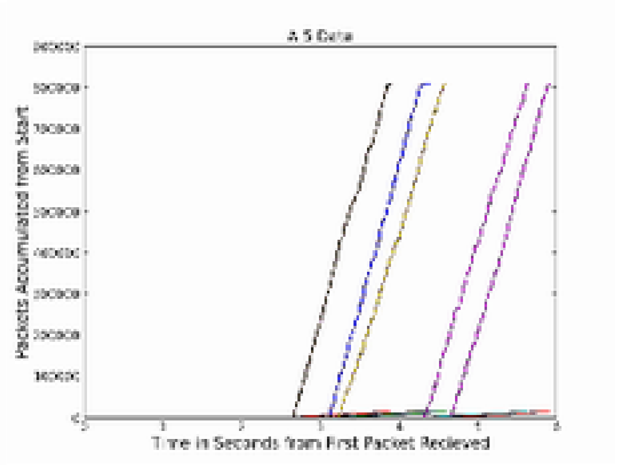
Comparing these three graphs provides extremely interesting results. We can see that in a direct download (Scenario 8), the whole file is downloaded in approximately 1.5 seconds on average. When adding one split TCP layer (C) in Scenario 2, not only does the start-up time for the download increase greatly, as would be expected when going through additional pathways, but the download speed also takes a dive, resulting in a 2-3 second download time.

However, when adding in a second split TCP layer (B) in Scenario 4, the start-up time increases further, but the download speed skyrockets, resulting in a download time of approximately .5-1 seconds. Thus, our data appears to indicate that the utilization of multiple layers of split TCP over wireless connections drastically increases the network performance.

Of course, it's important to note that download speed isn't the only variable to consider; start-up time must be considered as well. Adding additional split TCP layers clearly adds greatly to the start-up time. This tradeoff is vital. If one only wishes to download a small file, there is no reason to wait for a long start-up time to receive slightly better network performance. On the other hand, if one is downloading a very large file that could take a very long time, the start-up time becomes negligible, and thus the use of split TCP becomes a far more attractive option.

Presence of Intermediate (B) with local file: Scenario 3 vs. Scenario 5



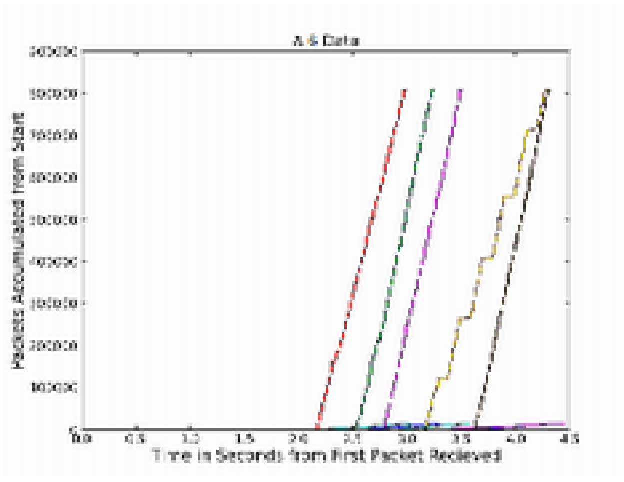


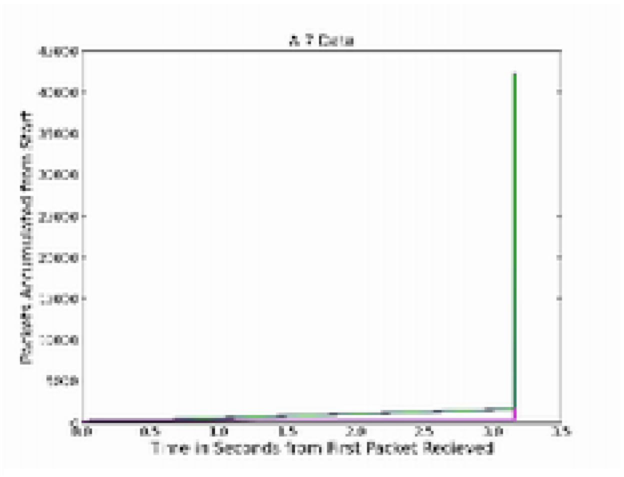
Another major comparison is between Scenarios 3 and 5 and Scenarios 2 and 4. Scenario 5 is similar to Scenario 3 in having communication solely over wireless and retrieving the picture locally. However, Scenario 5 has the intermediate B, just as Scenario 4 did.

In Scenarios 2 and 3, constant packet loss (as evidenced from the plateaus) seems to have slowed down the connection to take a full 4 seconds. However, once the intermediary B was introduced in Scenarios 4 and 5, packets seem to be dropped less frequently, allowing for much greater transfer rates as shown by Scenarios 4 and 5's one second download time. This reduction in download time indicates that indeed, the inclusion of B and thus the utilization of split TCP can be effective for reducing packet loss on even typical networks, just as we showed in the comparison between Scenarios 2, 4, and 8.

Furthermore, a final observation can be made: there is no significant difference between retrieving the picture locally and downloading it when it comes to download speed. However, start-up times are noticeably longer when downloading. Thus, it may be possible that C downloads the file from the Internet on request, and then sends the data to A just as if it had the data stored locally.

Location of Wired Connection: Scenario 6 vs. Scenario 7





The comparison between Scenario 6 and 7 is unfortunately incomplete due to erratic nature of the data collected for Scenario 7. We believe that this is due to a network configuration error caused by the usage of both wired and wireless networks. Whenever a computer has a wired communication, that computer records erratic data as shown by data sets A-1, A-7, B-6 and B-7. C seems unaffected by this problem.

Notes on the data:

- Tcpdump does not return 100% accurate timestamps, as packets are handled by the OS before being sent to tcpdump, and the timestamp is dependent on when tcpdump gets the packet. The time between arrival and tcpdump recording the packet is variable.
- The amount of data received is larger than the picture, most likely due to headers on the packets that contain routing information, packet number, and other metadata.
- Data was recorded leaving and entering all computers. Data for B and C can be found in the Appendix. Scenarios 1 and 6 had to be redone due unsuccessful data capture.

Notes on the Experimental Setup:

- Although consistency was sought, variables beyond our control may have had a factor in the experimental data. All data was collected in one location (the Olin Library) over all tests and over one wifi network (OLIN_GUEST). This network was chosen due to problems connecting to the faster network (OLIN_MH). This proved to be fortuitous since Elkan's netbook proved only capable of connecting to OLIN_GUEST.
- The computers performing the A role changed from test to test. Scenarios 2, 3, 4, and 5 were performed with Chloe Eghtebas' Olin-issued laptop. Scenarios 1, 6 and 7 were performed with David Elkan's Asus Netbook. However, we do not believe that this affected our results in any way.

Conclusion:

Even in a typical system without high packet loss, the use of split TCP appears to be very beneficial. Wireless connections, typically much slower than wired connections, showed speeds that were comparable to wired connections when connecting through the split TCP network. The main drawback is in the increased start-up time created by requiring more hops and therefore more handshakes. However, when transferring an extremely large file, we believe that this start-up time would become negligible. Therefore, split TCP appears to be advisable even in typical conditions when extremely large files are being transferred.

Future work:

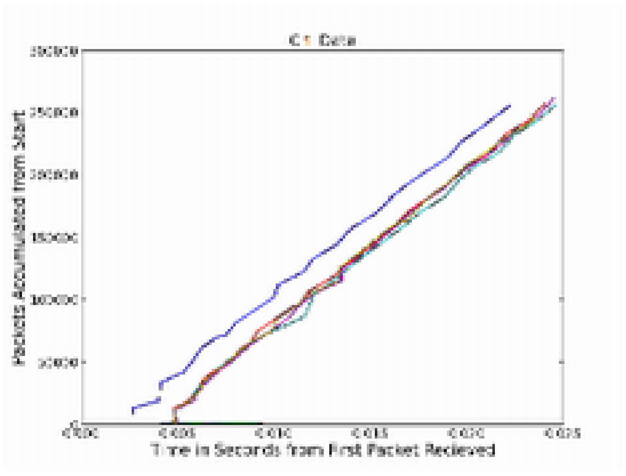
The main flaw in our conclusion is that we have not yet tested extremely large files to see whether or not split TCP truly enhances their overall download time. It is possible that larger files would even further increase the already long start-up times, which would result in split TCP not being as effective as we believe it to be. Further experiments must be made to test split TCP performance in this scenario.

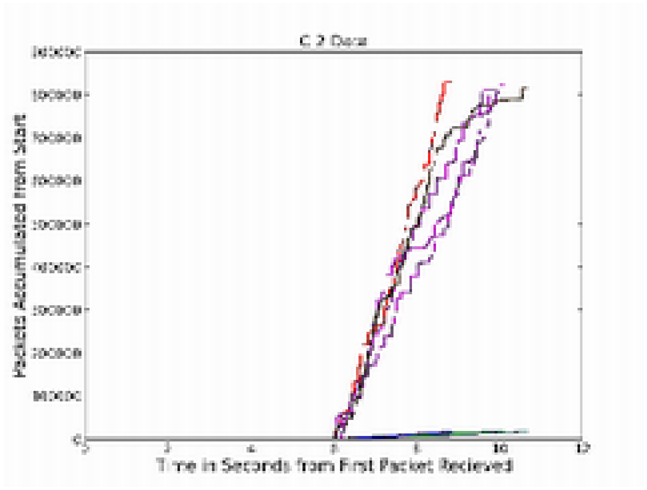
Additionally, more work can be done in regards to testing wired connections. Determining why our wired tests did not return correct data and solving the problem would be the first step to observing the effects of split TCP on a wired network. Furthermore, with the introduction of a switch between multiple wired neworks, it would be possible to test even more possible scenarios, such as one where A, B, and C are all wired together.

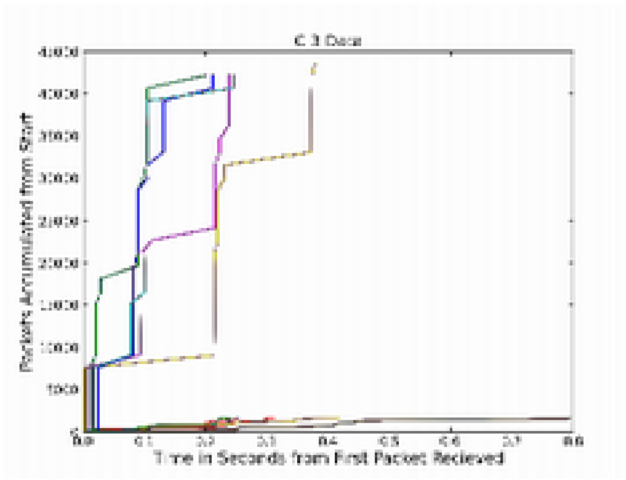
Acknowledgements:

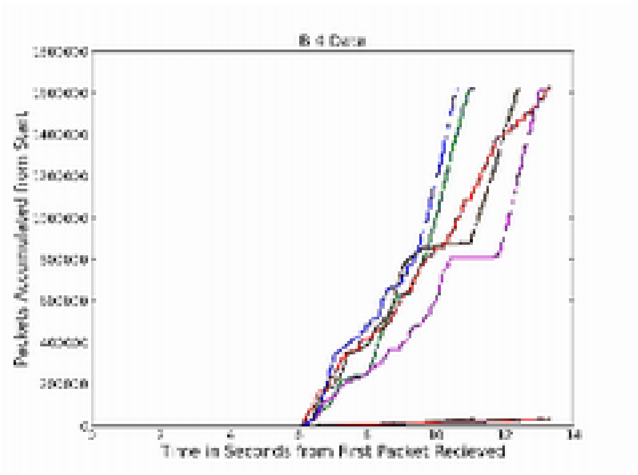
- Allen Downey, for being an excellent professor, inspiring us to explore this topic, providing weekly feedback, and helping us obtain a crossover cable.
- Chloe Eghtebas, for lending us her computer to run our experiments on after Zach’s died.
- David Elkan, for lending us his computer after Chloe needed to use hers.
- Manuel Amaral, for helping to discourage us from doing a different, impossible project.
- Hugh Mahon, for giving us simple example code for a proxy.

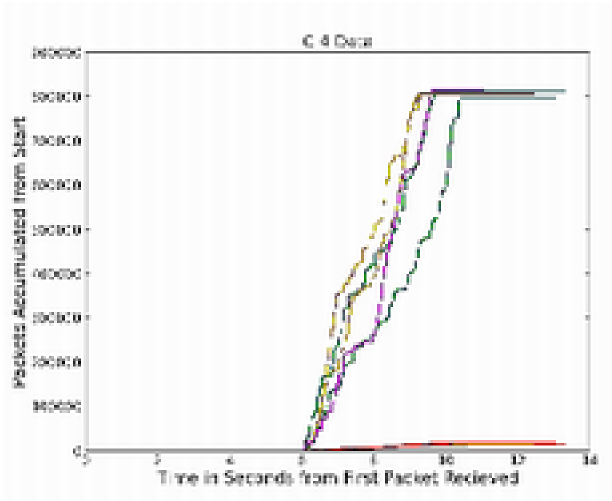
Appendix:

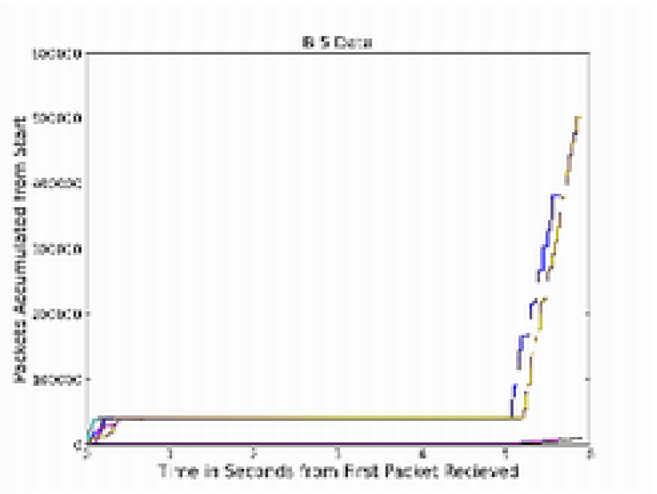

C 1 Data
Packets Accumulated from Start
Time in Seconds from First Packet Recieved

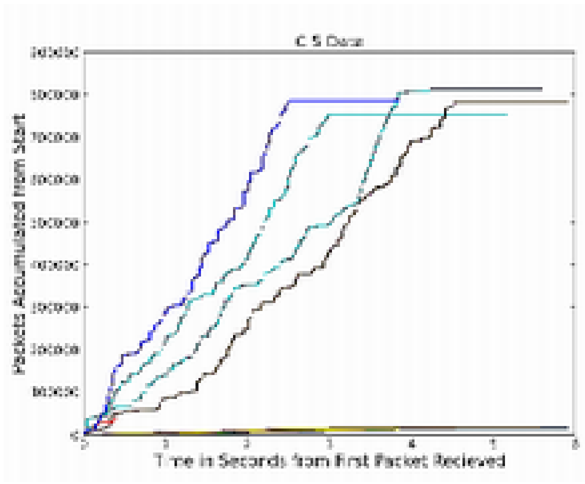

C 2 Data
Packets Accumulated from Start
Time in Seconds from First Packet Recieved

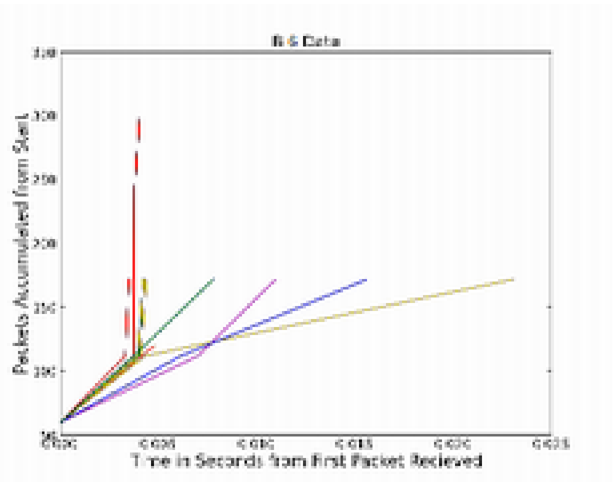

C 3 Data
Packets Accumulated from Start
Time in Seconds from First Packet Recieved

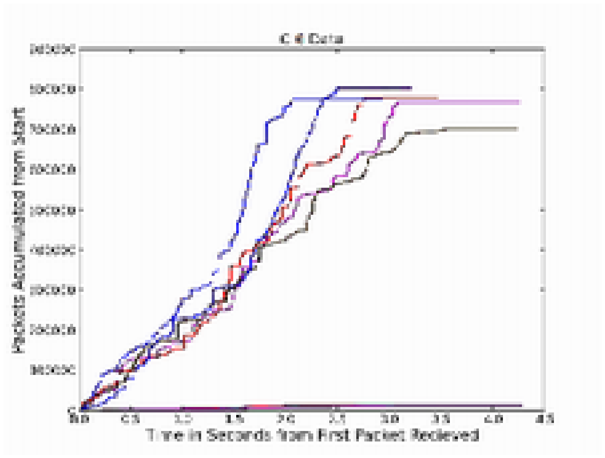

B 4 Data
Packets Accumulated from Start
Time in Seconds from First Packet Recieved

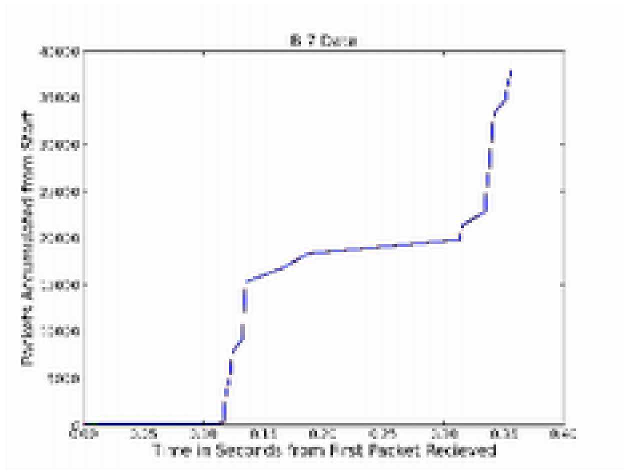

C 4 Data
Packets Accumulated from Start
Time in Seconds from First Packet Recieved

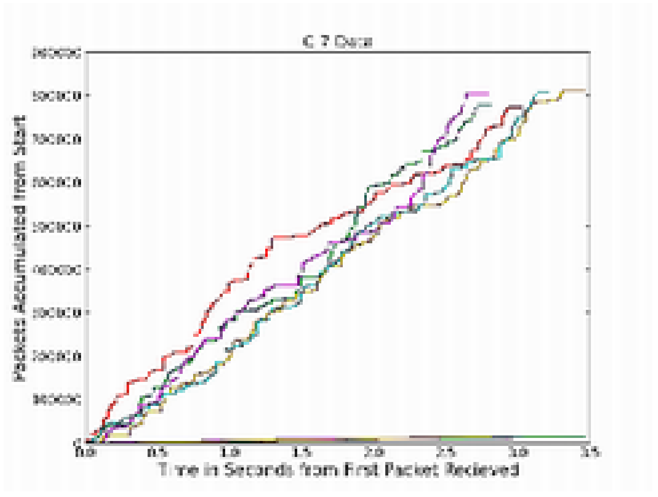

B 5 Data
Packets Accumulated from Start
Time in Seconds from First Packet Recieved











A Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>
```

```
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
        //max size of user input
        #define MAX_COMMAND 80
        //max size of image
        #define MAX_FILE 1000000

        int localsocket; //socket to read data from, write requests to
        struct sockaddr_in connect_to; //address of socket that will send us data, take requests
        void *connectaddr = (void *)malloc(sizeof(struct in_addr));
        int status;
        FILE *file_name;
        double file_len;
        double stuff_read;

        //process input arguments
        if(argc < 2)
        {
                printf("Give me an IP address!\n");
                return 1;
        }

        localsocket = socket(AF_INET, SOCK_STREAM, 0); //makes a simple tcp socket...

        //build the information of the socket we're connecting to
        connect_to.sin_family = AF_INET; //we're talking to a tcp socket
        connect_to.sin_port = htons(8080); //convert computer order to network order, # is port
                        number. Might want to make this variable.
        inet_pton(AF_INET,argv[1],connectaddr); //probably should add error checking here.
                        Copies argv[1] (ip address) into connectaddr as an in_addr struct.
        connect_to.sin_addr = *(struct in_addr *)connectaddr;

        //we could bind here; according to man 7 ip, not doing so means this client gets a
                        random open port. Which sounds okay to me.
        puts("Connecting to B");
        status = connect(localsocket, (struct sockaddr *) &connect_to, sizeof(connect_to));
        if(status < 0)
        {
                printf("problem. %s\n",strerror(errno));
                return 1;
        }
        else
        printf("We're golden! I think I connected!\n");

        write(localsocket, argv[2],MAX_COMMAND); //not sure using max command is right here
        read(localsocket, &file_len, sizeof(double)); //get size of file

        char *file = (char *)malloc(file_len*sizeof(char));

        printf("File length:%f\n",file_len);
        stuff_read = recv(localsocket,file,MAX_FILE,MSG_WAITALL);
        printf("Read: %f\n",stuff_read);
        printf("Opened file: %i\n",file_name = fopen("test","wb"));
        write(fileno(file_name), file, file_len);
        fclose(file_name);
        return 0;
}
```

B Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
        #define MAX_COMMAND 80
        #define MAX_FILE 1000000

        //Server code to talk to ModuleA
        //we want to design it so it just waits until someone connects to it.
        //so all we need is an outward-facing socket, yeah?

        int localsocket; //this one waits for a connection. Once it's connected to, we get...
        int connectedsocket; //what we'll actually be writing to/getting requests on
        char *command = (char *)malloc(MAX_COMMAND*sizeof(char));

        struct sockaddr_in myaddress;
        struct sockaddr_in *conn_addr;
        struct socklen_t *conn_addr_len = (struct socklen_t *)malloc(sizeof(myaddress));
        int option = 1;
        int err = 0;

        localsocket = socket(AF_INET, SOCK_STREAM, 0);
        setsockopt(localsocket, SOL_SOCKET, SO_REUSEADDR, (const void *)&option, sizeof(int));

        //now we need to bind it to an address
        //build said address
        myaddress.sin_family = AF_INET;
        myaddress.sin_port = htons(8080); //maybe make this configurable in the future
        myaddress.sin_addr.s_addr = htonl(INADDR_ANY); //assign the computer's IP to server

        bind(localsocket, (const struct sockaddr *)&myaddress, sizeof(myaddress));

        //now we wait until someone talks to us...
        listen(localsocket,1000);
        connectedsocket = accept(localsocket, (struct sockaddr *) &conn_addr, conn_addr_len); //accepting from A
        puts("Connected to A");



        //Client code!
        int localclient; //socket to read data from, write requests to
        struct sockaddr_in connect_to; //address of socket that will send us data, take requests
        void *connectaddr = (void *)malloc(sizeof(struct in_addr));
        int status;
        double file_len;

        localclient = socket(AF_INET, SOCK_STREAM, 0); //makes a simple tcp socket...
        printf("%i\n",localclient);
        //build the information of the socket we're connecting to
        connect_to.sin_family = AF_INET; //we're talking to a tcp socket
        connect_to.sin_port = htons(8080); //convert computer order to network order, # is port
                                number. Might want to make this variable.
        inet_pton(AF_INET,argv[1],connectaddr); //probably should add error checking here.
                                Copies argv[1] (ip address) into connectaddr as an in_addr struct.
        connect_to.sin_addr = *(struct in_addr *)connectaddr;

        //we could bind here; according to man 7 ip, not doing so means this client gets a
                                random open port. Which sounds okay to me.
        puts("Connecting to C");
        status = connect(localclient, (struct sockaddr *) &connect_to, sizeof(connect_to));
                                //connecting to C
        if(status < 0){
                printf("problem. %s\n",strerror(errno));
                return 1;
        }else{
                printf("In communist Russia module B connected to you!\n");
        }
        err = read(connectedsocket, command, MAX_COMMAND); //read commands from A
        printf("%s\n",command);
        //send command along client
        write(localclient, command, MAX_COMMAND); //write commands to C
        printf("Sent!\n");

        read(localclient,&file_len,sizeof(double)); //read file length from C
        write(connectedsocket,&file_len,sizeof(double));//write file length to A

        char *file = (char *)malloc(file_len*sizeof(char)); //allocate space for file
        recv(localclient,file,MAX_FILE,MSG_WAITALL); //read file from C
        write(connectedsocket,file,file_len);  //write file to A
        return 0;
}
```

C Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>

#define MAX_COMMAND 80

int main (){
        //we want to design it so it just waits until someone connects to it.
        //so all we need is an outward-facing socket, yeah?

        int localsocket; //this one waits for a connection. Once it's connected to, we get...
        int connectedsocket; //what we'll actually be writing to/getting requests on
        char *command = (char *)malloc(MAX_COMMAND*sizeof(char));

        struct sockaddr_in myaddress;
        struct sockaddr_in *conn_addr;
        socklen_t *conn_addr_len = (socklen_t *)malloc(sizeof(myaddress));
        int option = 1;

        localsocket = socket(AF_INET, SOCK_STREAM, 0);
        setsockopt(localsocket, SOL_SOCKET, SO_REUSEADDR, (const void *)&option, sizeof(int));

        //now we need to bind it to an address
        //build said address
        myaddress.sin_family = AF_INET;
        myaddress.sin_port = htons(8080); //maybe make this configurable in the future
        myaddress.sin_addr.s_addr = htonl(INADDR_ANY); //assign the computer's IP to the server

        bind(localsocket, (const struct sockaddr *)&myaddress, sizeof(myaddress));

        //now we wait until someone talks to us...
        listen(localsocket,1000); //but we only want one person talking to us
        printf("???\n");
        connectedsocket = accept(localsocket, (struct sockaddr *) &conn_addr, conn_addr_len);

        //read the command
```

```
		read(connectedsocket, command, MAX_COMMAND); //reads from B
		printf("%s\n",command);

		//The file getting part of the process

		pid_t is_child;
		FILE* file;
		char *file_data;
		double fileLen;
		struct stat *fileStat=(struct stat*)malloc(sizeof(struct stat));
		char* filepath=strrchr(command,'/');
		int file_sent=0;
		is_child=fork();

		if (is_child>=0){ //Fork was successful
			if(is_child != 0){ /*Parent process*/
				file_sent=1;
				execlp("wget","wget",command,NULL);
			}else{
			sleep(6); //wait for file to download
				filepath+=1;
				printf("Sending: %s\n",filepath); //prints filepath
				file=fopen(filepath, "rb"); //opens the file specified by filepath
				if (fstat(fileno(file),fileStat)<0){ //finds its size
				printf("Error in determining file size");
				}
				fileLen=fileStat->st_size;
				printf("Filelength: %f\n",fileLen);
				//printf("Opened file: %i\n",file);
				file_data=(char*)malloc(fileLen+1); //allocates memory for the file
				fread(file_data,fileLen,1,file); //reads the file length
				printf("Read: %i\n",ferror(file));
				printf("File data: %s\n",file_data);
				fclose(file);
				//write(connectedsocket,&fileLen,sizeof(double));
				//write(connectedsocket,filepath,strlen(filepath));
				write(connectedsocket,&fileLen,sizeof(double)); //Writes filelength to B
				printf("Wrote: %i\n",write(connectedsocket,file_data,fileLen)); //Writes file
																to B
				printf("Error? %s\n",strerror(errno));
				puts("Wrote here.");
			}
		}
		return 1;
}
```

Python Script 1 (Makes .csv out of tcpdump text files):

```
import sys
import os
import csv

def time_to_sec(time):
		l=time.split(":")
		hour=int(l[0])
		minute=int(l[1])
		sec=float(l[2])
		minute+=60*hour
		sec+=60*minute
		return sec

def run():
		if len(sys.argv)!=2:
		sys.exit("Must provide directory")
		for directory,dir2,files in os.walk(sys.argv[1]):
		    for thing in files:
            i=0
            if thing.endswith(".txt"):
                inpute = open(os.path.join(directory,thing),'rb')
                outpute = csv.writer(open(((os.path.join(directory,thing)[:-4])+".csv"),'wb'),delimiter=',',quotechar='|')
                liste=[]
                file_len=0
                start_time=0
                for line in inpute:
    liste=line.split()
    if i%2 == 0 and len(liste) != 0:
                    file_len += int(liste[16][:-1])
                    if i==0:
                        start_time=time_to_sec(liste[0])
                        outpute.writerow([0.0,file_len])
                        print start_time
                    else:
                        outpute.writerow([time_to_sec(liste[0])-start_time,file_len])
    i+=1
run()
```

Python Script 2 (Makes graphs out of .csv):

```
import csv
import os
import sys
import matplotlib.pyplot as pyplot

def pretty_pictures():
    data = list()
    if len(sys.argv)!=2:
        sys.exit("Must provide directory")
    for directory,dir2,files in os.walk(sys.argv[1]):
        for thing in files:
            i=0
            if thing.endswith(".csv"):
                masterdata = csv.reader(open((os.path.join(directory,thing)),'rb'),delimiter=',',quotechar='|')
                d = dict()
                for line in masterdata:
       d[float(line[0])] = float(line[1])
       data.append(d)
    for d in data:
        times = d.keys()
        times.sort()
        sizes = d.values()
        sizes.sort()
    pyplot.plot(times, sizes)
    pyplot.xlabel("Time in Seconds from First Packet Recieved",fontsize=15)
    pyplot.ylabel("Packets Accumulated from Start",fontsize=15)
    pyplot.title(directory[-2:-1]+" "+directory[-4:-3]+" Data")
    pyplot.show()

pretty_pictures()
```