Multipath TCP
Simplified Implementation

Daniel Limanowski

November 29th, 2018

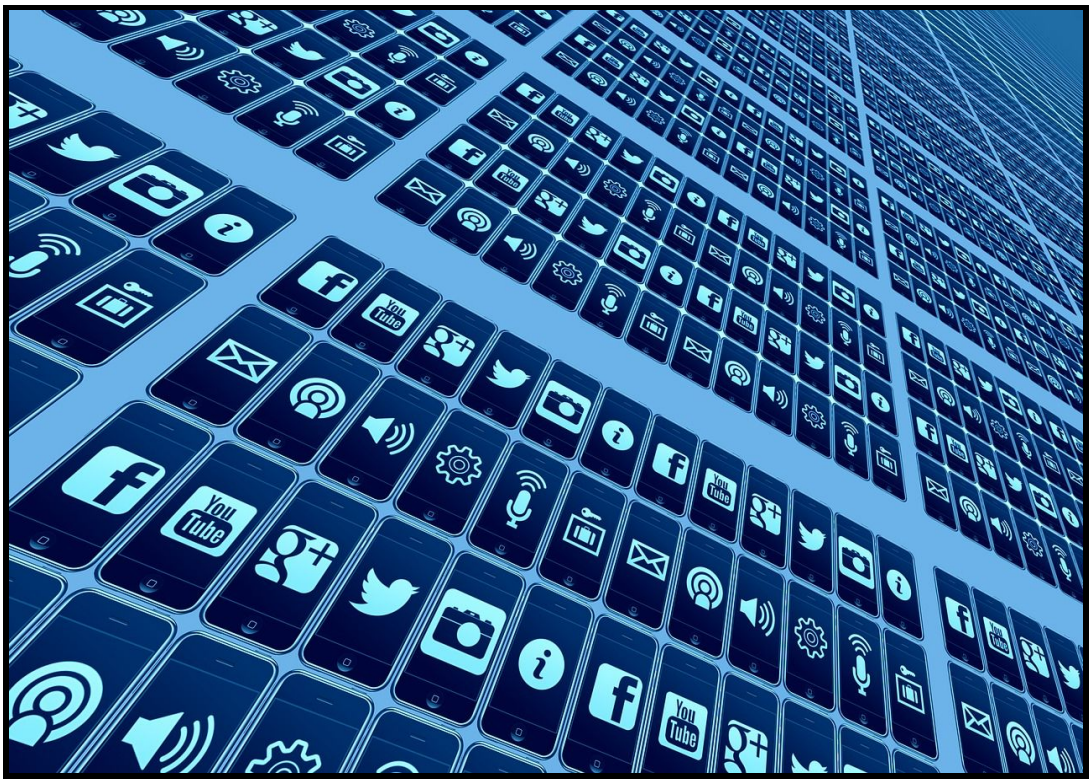

Table of Contents

Overview

This document details the implementation of a simplified version of the currently-experimental networking protocol known as multipath TCP, or more formerly "TCP Extensions for Multipath Operation with Multiple Addresses" (RFC 6824).

Multipath TCP aims to provide multiple paths for data from a single source to travel to its single destination. In current TCP implementations, all TCP traffic travels over one path to get to its destination. By increasing the paths from one to a varied number, the bandwidth of the network connection is proportionally increased because the traffic can flow over multiple paths instead of one fixed path. This essentially means that the traffic can flow to different intermediate routers or networks before reaching its final destination.

The implementation shown in this report is not fully compliant with RFC 6824. There are many features and details left out due to the brevity of this project. The goal of this project is to focus on establishing multiple subflows to send data across a network as well as successfully reassembling the data in order on the server with the help of a control connection.

Design Requirements

This project had the following design requirements for the given simplified multipath-TCP implementation:

Client

The project requires a client application to generate the payload (data to be sent to the server) and send it over multiple subflows to the server, along with a Data Sequence Signal (DSS) to correlate the data from each subflow to the entire conversation/message.

The specific design requirements of the client are as follows:

- Establish three data subflow TCP connections to the server, with each of these connections being forked off as a child process
- Establish a single control TCP connection to send the DSS to the server
- Send a payload of 992 bytes to the server that consists of sixteen iterations of the characters 0-9, a-z, and A-Z
- The parent process sends the children the payload data in four-byte chunks, round-robin style (meaning Subflow (SF) 1 receives bytes 0-4, SF 2 receives 5-9, SF 3 receives 10-14, SF 1 receives 15-19, and so on using pipes
- The parent process sends a DSS to the server over the control connection for each four-byte chunk sent to a child
- The children (subflows) send the data to the server as they receive it

- Log the DSS-to-data mapping in a file

Server

In order to complete a network transfer of a 992-byte payload, a server application had to be written for this project. The server receives Data Sequence Signals (DSSs) on a control port and correlates the DSS to the data received on three TCP subflow sockets.

The specific design requirements of the server are as follows:

- Listening control port socket to accept a control flow connection from the client
- Three listening TCP subflow sockets to accept subflow connections from the client
- Receive a chunked-payload from each of the forked subflow sockets and inverse-map the data to the DSSs via pipe communication to reconstruct the conversation in the proper order
- Display the received 992 bytes in order on the display monitor of the machine running the server
- Log the DSS-to-data mapping in a file

Design Detail

This section describes the different files and functions required for proper implementation of the project. The project was implemented solely using the C programming language.

Files

The project is made up of the following files:

File	Description
common.c	Shared code that the server and client both use.
common.h	Header file for common.c. Contains shared information (such as includes) for the client and server applications as well.
Makefile	Allows for “make” and “make clean” commands to be run on the project to compile and cleanup the project, respectively.
mptcp-client.c	The client application code.
mptcp-client.h	The header file for the client.

mptcp-server.c	The server application code.
mptcp-server.h	The header file for the server.

Functions

The project contains the following functions:

add_to_log (common.c)

Prototype:

- `int add_to_log(FILE *log_fp, unsigned short dsn, char msg[ROUND_ROBIN_SIZE])`

Description:

- Adds a comma-separated line to the logfile given a Data-Sequence-Number * (DSN) and a message

Accepts:

- `FILE *log_fp`: Already-opened pointer to log-file
- `unsigned short dsn`: data sequence number for line in log
- `char msg[ROUND_ROBIN_SIZE]`: Message chunk associated with DSN for log line

Returns:

- 1 if line added successfully, -1 otherwise

main (mptcp-client.c)

Prototype:

- `int main(void)`

Description:

- Forks TCP subflows and sends payload data to each subflow while sending mapping information to the server over control port and logging the info.

Accepts:

- Accepts no input

Returns:

- EXIT_SUCCESS upon clean exit of child processes or EXIT_FAILURE if anything went wrong.

create_subflow (mptcp-client.c)

Prototype:

- `int create_subflow(int port)`

Description:

- Initializes a TCP subflow connection to the server by setting up a socket and connecting to `SERV_IP`.

Accepts:

- `int port`: TCP port to connect to on the server host

Returns:

- File descriptor of the connected socket

subflow (mptcp-client.c)

Prototype:

- `void subflow(int init_pipe[2], int write_pipes[3][2], int read_pipes[3][2])`

Description:

- Child processes (the subflows) call this function to receive data from parent process and send over to server.

Accepts:

- `int init_pipe[2]`: Initialization pipe used to get a child's "ID" so it knows which of the read/write pipes to use
- `int write_pipes[3][2]`: Array (one for each subflow) of pipes for child to write to (writes back the received DSS)
- `int read_pipes[3][2]`: Array of pipes for child to read from (data)

Returns:

- Nothing

main (mptcp-server.c)

Prototype:

- `int main(void)`

Description:

- Forks TCP sockets to receive payload chunks. Maps sequence numbers from the client via control port to the payload chunks on the subflows. Logs the mappings to a file.

Accepts:

- Accepts no input

Returns:

- EXIT_SUCCESS upon clean exit of child processes or EXIT_FAILURE if anything went wrong.

setup_socket (mptcp-server.c)

Prototype:

- `int setup_socket(int port)`

Description:

- Initializes a TCP subflow socket and waits for a connection to it, eventually returning a connected socket
- NOTE: this function will block until a connection is established

Accepts:

- `int port`: TCP port to setup port on for connections

Returns:

- File descriptor of the connected socket

subflow (mptcp-server.c)

Prototype:

- `void subflow(int read_pipe[2], int write_pipe[2])`

Description:

- Child processes (the subflows) call this function to receive data from their socket and pipe it up to the parent process.

Accepts:

- `int read_pipe[2]`: The child reads from this pipe, for example to get its connection socket. The parent writes to this pipe.
- `int write_pipe[2]`: The child writes the connection data to this pipe.

Returns:

- Nothing

Protocol

Figure 1 below describes the simplified Multipath TCP protocol implemented for this project.

The server application is started first, followed by the client. The client establishes four connections in total to the server: one control connection and three subflow connections. The control connection sends a Data Sequence Signal (DSS), consisting of a sixteen bit Data Sequence Number (DSN), from the client to the server. Along with every DSN sent, a subflow sends a four-byte chunk of data that is correlated to the DSN.

The client sends the four-byte data chunks Round-Robin style to each subflow. This means that chunks are sent to subflow 1, 2, 3, then back again to subflow 1 until the entirety of the payload (or message) has been transmitted. To reiterate, every chunk sent is associated with a DSN sent over the control connection.

The server receives a DSN and four-byte chunk of data that is piped up from a child process to the parent process. The parent process associates the bytes to the DSN and reorders the conversation in sequential order according to the DSN.

Data is sent by the client's children processes until the message is completely distributed. At this point, the parent client process sends "STOP" to its children processes which then send that to the server. The client subflows exit once receiving the stop command, as well as the server subflows. The programs both terminate.

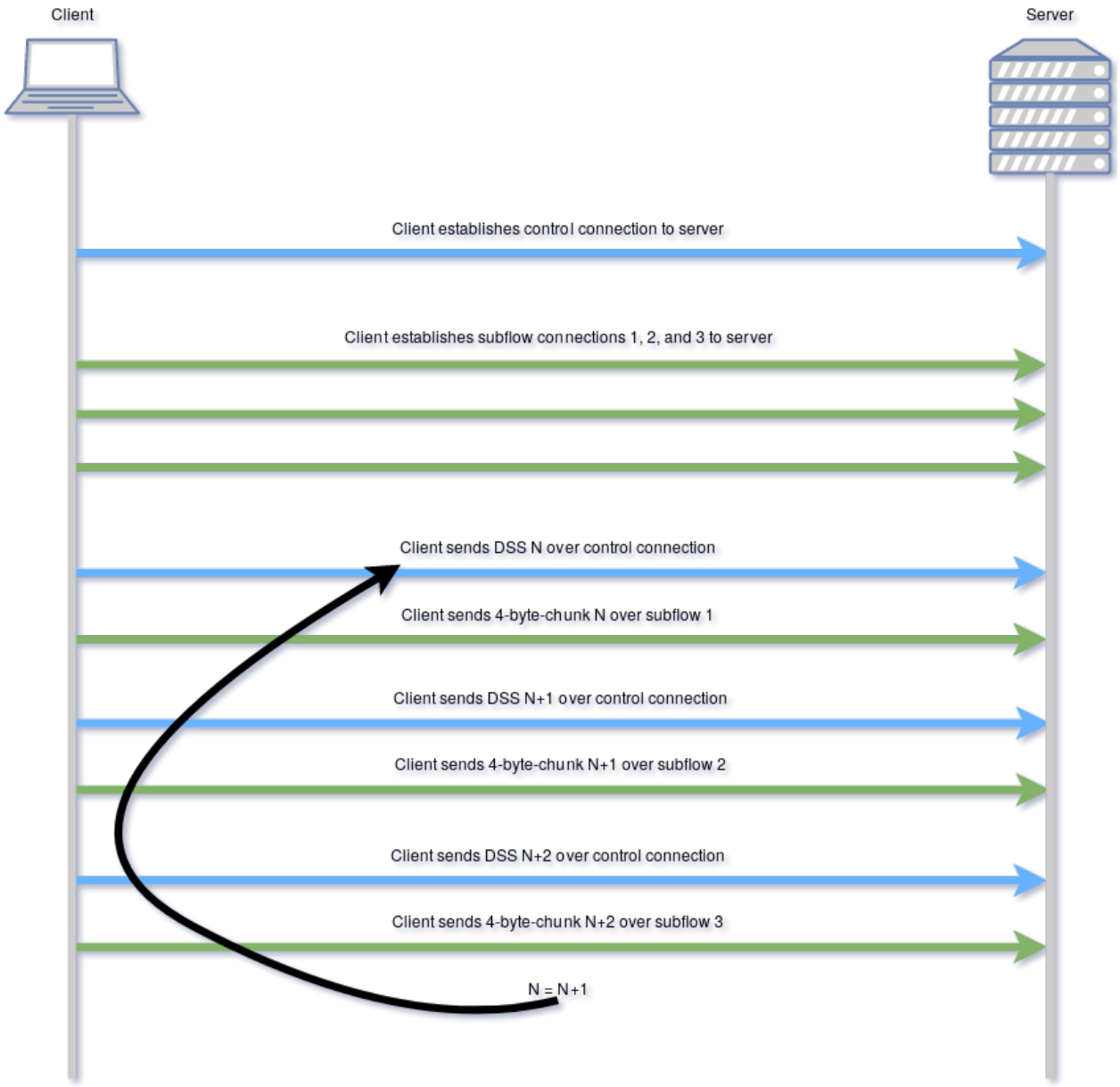

Figure 1: Simplified Multipath TCP Implementation

Innovative Features

Any features added on top of the existing design requirements are considered “innovative” for this project. This section describes those extra features.

Analysis with Wireshark

To prove that the code written for this project and the protocol developed works as detailed in this document, the traffic sent from the client and server was captured and analyzed using the open source packet analyzer, Wireshark.

This first screenshot, shown below, shows the typical TCP three-way handshake of SYN, SYN ACK, and ACK for ports 50000-50003 when the client initially is ran.

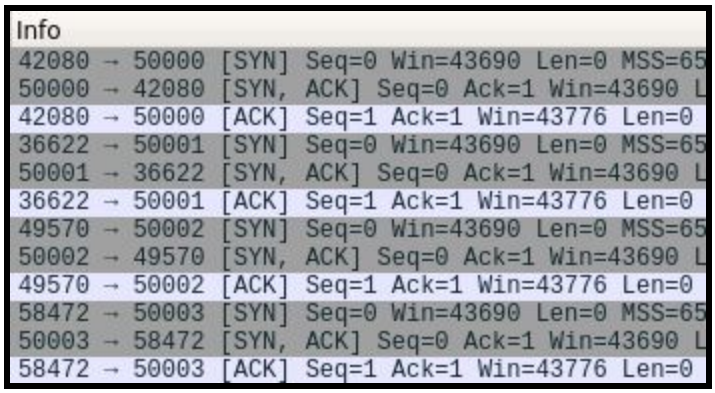

Figure 2: Connection process

After the handshake, the client immediately begins sending subflow data and DSSs. In the screenshot below, the client sends “0123” to port 50001 and the server sends an ACK to acknowledge the message.

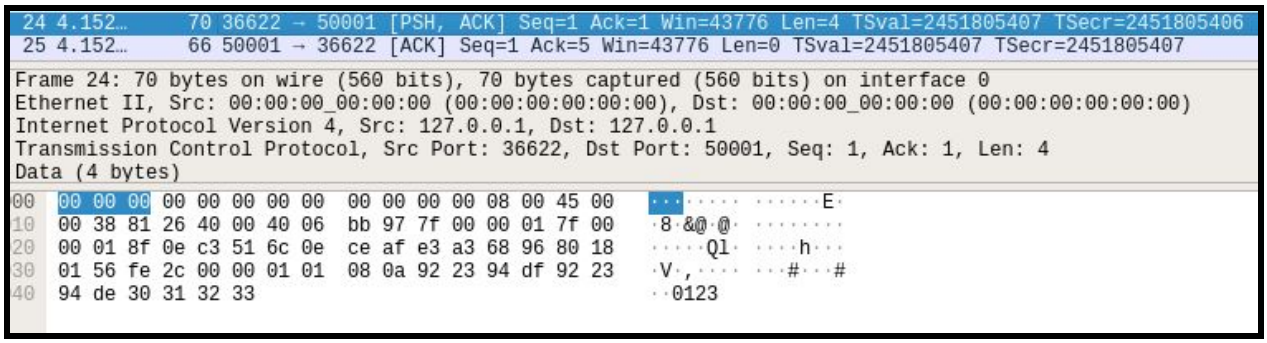

Figure 3: Sending subflow data to the server

The packet sent over port 50000 just before the above data packets contains the simple DSS, which only consists of sixteen bits (two bytes, circled in red) of a number, starting at zero and incrementing by one each iteration.

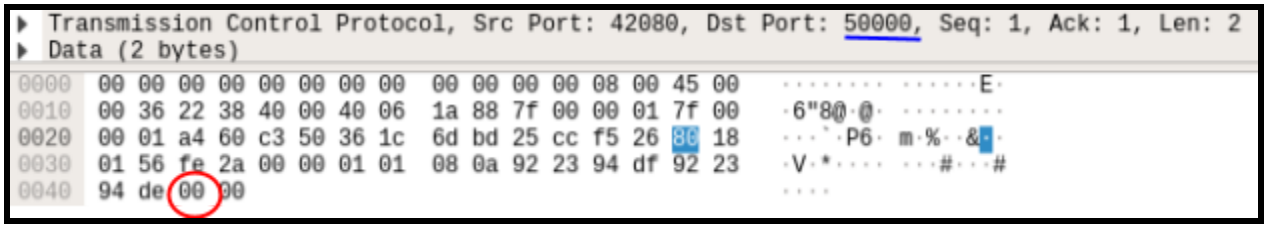

Figure 4: The DSS/DSN shown in red being sent to the server over the control port shown in blue

Once all data is sent to the server, the client pushes out “STOP” to indicate to its children, as well as the server, that it is time to terminate. The TCP traffic begins the FIN, ACK process to close out the connection, as shown in the final screenshot below.

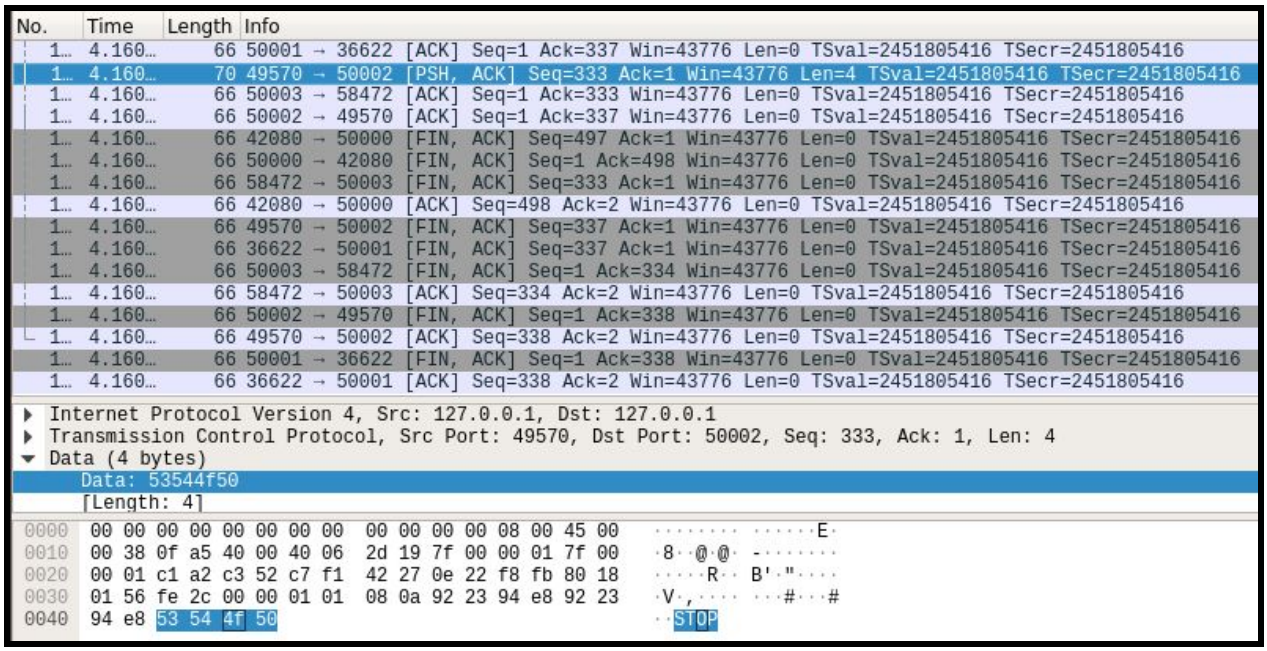

Figure 5: The termination process with STOP over the control connection

Assumptions/Limitations

This section details any assumptions that were made regarding the implementation of this project.

Simple Data Sequence Signal (DSS)

For this simplified implementation of multipath TCP, the DSS was reduced to a simple sixteen bit Data Sequence Number (DSN). The DSN is an unsigned short primitive in C, which is of size sixteen bits. The DSS, which in this case is just the DSN, is incremented by one each time the

client sends a chunk of data on the subflows. The server then uses the DSS to correlate the order of the data chunks it receives on its data connections.

To contrast real implementations of multipath TCP, the DSS is a much larger data field that contains checksums, flags, length information, and more. This extra information was unnecessary to properly implement this simplified implementation.

Iterative and single-use server

The design requirements did not specify that this server had to run concurrently (accepting more than one client). The requirements also did not specify if the server had to keep running “forever.”

In order to focus on successful implementation of the project, decisions were made to conform as closely as possible to the requirements and to keep functionality as simple as possible. That being stated, the server is iterative and quits after servicing one client. This single-use server demonstrates the required collection and reassembly of data as detailed in the design requirements.

Sleeping sockets

In the `create_subflow` function of `mptcp-client.c`, a one-millisecond `sleep` statement was added as a sort of hack to ensure the program functions properly.

The `sleep` statement was added between an `inet_pton` statement and a `connect` statement. `inet_pton` takes an IP address represented in dot notation in the form of a string, and converts it to binary for use with sockets.

When the `sleep` statement is not in place, connect fails with “Connection Refused” presumably because the binary version of the server address is not fully ready for use. To allow for some delay, a minimal `sleep` statement was introduced with no noticeable side effects.

Closing Remarks

This project and report demonstrate a brief functionality of the benefits of multipath TCP. Multiple network paths mean increased bandwidth and better streaming experience on lossy connections.

The benefits of multipath TCP can be seen in cellular networks. Cell phones typically have two major sources of Internet access - via 1) local WiFi and 2) Cellular connection. When a cell phone leaves the connection zone of a WiFi network, it will eventually switch over to use the cellular connection, and vice versa when a phone enters the zone of a WiFi network. This

interruption in switching connections can be seen when a user is actively browsing the Internet, especially in video streams. Multipath TCP allows multiple connections, and therefore can continue sending data even when a source of Internet is lost or replaced.

Multipath TCP is currently an experimental protocol, but with Apple's recent adoption and more companies following suit, it is likely to become mainstream over the next few years, with its improvements benefiting millions of users globally. Before the widespread adoption of 5G networking, multipath TCP can benefit devices that use multiple network interfaces.