MP TCP :: C Programing Implementation

CPRE 489

Final Project by:

Mike Croskey

Table of Contents

1 .) Intro:

2.) Project Overview:

3.) The Program:

4.) Concluding Remarks

5.) Works Cited

Introduction:

The following document contains a description the client and server programs implementing the theory used in MP-TCP type data transfer. The implementation explained throughout this document is for educational purposes only and is designed with the sole purpose of demonstrating the ability to utilize multiple connections for transporting a single portion of data. This document provides the reader with a brief explanation on MP-TCP and the experimental implementation of such protocols. A description of the programing is provided covering the underlying functionality used to emulate the MP-TCP in c. the program portion covering much of the in depth explanation on how the program works and the related methods needed to provide such function. A brief explanation is given on what challenges were faced during the completion of the project and how these were overcame including potential improvements. Hence, this document is meant to give an overview and provide an example of MP-TCP using the c programing language.

An Overview of MPTCP:

MP-TCP is short for Multi Path TCP which is a style of transferring data using multiple TCP connections. An experimental platform on which using multiple "TCP/IP session would improve resource usage within the network" (TCP Extensions 1).

MP connections are capable of normal single mode TCP/IP transfer when a connection is not capable of utilizing MP. Only when a source and destination both have the ability to use MP protocol is it possible to create a connection in this manner. The transfer establishes a tradition two way connection and searches for additional available connection on the connected hosts (please note a host represents a sender or a receiver). The hosts must both ensure "extra paths are available, additional TCP sessions (termed MPTCP "subflows") are "then" created on these paths." (TCP Extensions 6). These subflows are utilized to create multiple data paths and provide the infrastructure for sending allotted segments of data simultaneously between hosts.

To ensure data is kept in sequence MP implements a method of dividing data into segmented packets. This allows MP to add "connection-level sequence numbers to allow the reassembly of segments arriving on multiple subflows with differing network delays" (TCP Extensions 7). These subflows are managed independently by a control structure that assigns the sequences and initiates the passing of data. MP is able to send this data using a "64-bit data sequence number (DSN) to number all data sent over the MPTCP connections" (TCP Extensions 10). These sequence numbers are independent of sub flow sequences that manage error recovery and transmittal of lost packets. The DSN is primarily use to reassemble the data into its original ordering once the transfer of data is completed. In addition to the DSN a Data Sequence Signal (DSS) is used consisting "of the subflow sequence number, data sequence number, and length for which this mapping is valid Once the data transfers are completed the connections are then terminated as would traditional single mode TCP/IP" (TCP Extensions 10). This is responsible for indicating how the subflows should handling the transfer of data. The receiver uses this information in order to capture the correct segments in hopes of reallocating in the correct pattern. Management of subflows is done with the use of flags to indicate the addition or removal of flows but connection establishment and termination is parallel to normal TCP/IP operations.

An important part of MP is the ability to make a connection appear as though it is uninterrupted and behalves as a single data stream. This may sound confusing but think for a moment about a system of pipes. Water is of course pumped into end A of the system and in this case it comes out the other end B. Regardless of how the interconnected pipes make their way from A to B this system pumps the same amount of water in one end as it outputs from the other. MP connections work in a much similar fashion as it will take information from the application and transfer this information across a connection to another application. What goes in one end must come out the other in the same order to ensure the two applications communicate correctly. MP connections provide this connection in a manner that appears to be a single connection, but in fact the MP protocol is communicating over multiple connections between the hosts. The act of multi-TCP/IP connection utilization is of course the primary argument behind using a protocol such as MP because a network can transfer at theoretically higher throughputs and error correction capabilities.

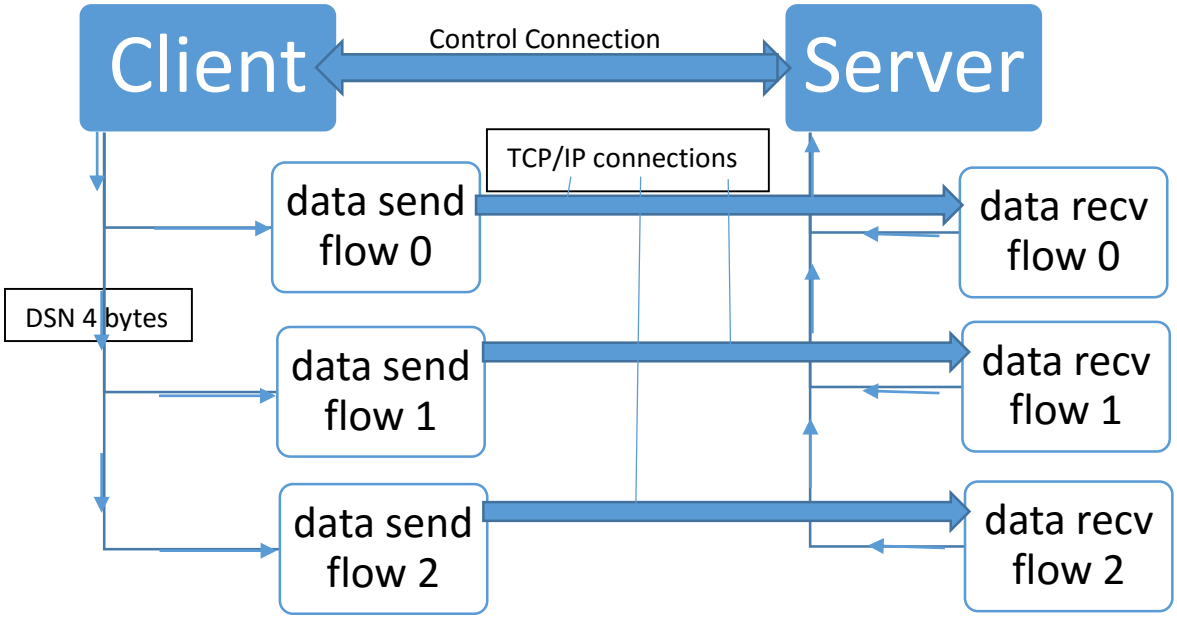
An extensive explanation is available online at the provided work cited web page referenced given in prior in text citations. This covers in detail the exact packet structures and connection procedurals outlined in the above section. The previous overview is design to give enough background for the general understanding behind MPTCP protocols. Individuals are encouraged to look at the provided reference regarding any further question involving the application of MPTCP.

An Overview of the Project:

With the new found understanding of MPTCP an individual is now able to understand the underlying purpose of this project and the code behind the curtains. The intentions of this project are much similar to that of the MPTCP but with a few primary differences. The requirements for this project state a client and server program must be written in c programing language. A server and client program should be pair together to demonstrate the utilization of multiple connections for transferring a single data stream. The programs are to use Berkley sockets as a connection platform and establish a total of four TCP/IP connections between the client and the server. The client must then create a buffer of data consisting of 16 repetitions of 0-9 , a-z, A-Z as representing the theoretical data being transferred.

For simplicity purposes the project implements a separate control connection specifically for managing DSS segments between hosts (no use of flags or additional connection searching). This connection is maintained in addition to providing the establishment of three alternative TCP/IP connections between a client and server host unlike true MPTCPs searching for additional available connections. The three additional connection responsible for transferring DSN segments are forked after connection is established to provide child processes within the client and server. The requirements also state the data management routine must communicate with the children processes via pipes. These pipes are created prior to the forking of the child process to allow communication between the parent process and its children.

The below diagram provides a visual representation of the general flow of the required implementation.



The overall flow of the implementation divides the generated data buffer into 4 byte DSN segments and passes each segment into a child process responsible for handling the individual subflows. Each subflow is passed data in a cyclic manner one fallowing the previous and so on, resetting on each revolution. As each DSN is sequenced the control connection sends a packet with DSS information to the server. The server receives the DSS packet retrieves the appropriate data from the specified connection and finally reassembles the data buffer. A log file is kept by the client and server indicating the DSS information used for each DSN packet.

The Program

"Client Side"

Client.c:

This section covers in detail the implementation of the client side c program used for demonstrating the MPTCP type protocol specified in the project guidelines. The client program is divided into a few main categories each responsible for a specific task associated with the overall client design. These methods consist of an initial data buffer creation routine, connection management routines, child process method for writing data over TCP socket and the main routine operating as a control structure. The controlling structure is needed to manage the state like behavior implemented during execution.

Data Routine

The client handles the creation of the data buffer as an initial step of the overall process. A pointer is allocated with an initial size of 992 characters. This is used to store the generated data buffer consisting of 16 repetitions looping thru sets of 0-9, a-z and finally A-Z. The resulting data structure can be seen in figure 2 below.

Figure 2

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b …… x | y | z | A | B | C …… X | Y | Z | 0 | ……

The main routine runs a few additional initialization steps allocating structures, pipes, threads and sockets used to establish connections. A control connection is established with the provided server IP and control port needed for sending DSN packets. A c programming struct is created to pass socket details and pipe information down into a created thread. This struct is then used to store the initialized pipes as file pointers in a char array.

Connection Management Routines

The main program calls a routine for creating each data connection as a thread and this thread establishes a connection to given server IP and specified port address. The threaded process accepts a pointer to the struct as a parameter during the thread call. The struct pointer is then used to access the socket details and establish a connection. Only after each connection is accepted by the server does the client spawn a child process responsible for writing information piped in from the parent. The threaded connection routine calls a function passing a communication pipe along with a file pointer pertaining to the established connection. These parameters are utilized by the child process in order to receive piped data segment and rely them onto the TCP/IP connection. This finalizes the initialization steps taken to setup the infrastructure needed to transfer multiple data streams.

Main Control Routine

Initialization is followed by a looping structure designed as a state machine splitting the generated data buffer into the desired DSN segments. In addition to creation of segmented data packets a sequence number is generated for each packet to be used in the DSS packet. The DSS packet is constructed using the sequence number, current first bit alignment and data channel used to send DSN segment. The sequence number is global for all DSN segments generated by the control structure. The current first bit alignment is the index from the generated data array coordinating to the first bit in the DSN segment. The data channel byte indicates the data connection used to transmit the DSN segment. Using the generated DSN segment and related DSS packet the control structure transfers the data to the server.

As stated in the previous section the main loop is responsible for communicating the generated DSN and DSS packets at the appropriate times. In order to achieve this the client was originally setup using sleep statements to temporarily delay the timing of the client to ensure proper communication is established and proper steps are completed before reading and writing between the client and server begins. The parent would cause delays in the control connection waiting until all of the connections are established and the children processes have been created before it begins to send information into the pipes or to the server. These timing delays were eventually removed after perfecting the behavior of the child functions.

The general state like behavior of the client can be described as a cyclic push buffer. The client essentially loads a sending buffer with a set of desired characters. After initialization of the needed connections the client begins at the initial state having a sequence number of 0 and starting index of 0. The first state sends the constructed DSN packet into the child pipe for sending data over connection and writes the DSS packet to the control connection indicating to the server that a data segment is transferred on the first data connection. The second and third states are the same as the first except the 2^{nd} and 3^{rd} data connections are used instead. The final state occurs during the completion of the transfer, during this stage any open pipes are closed, child processed are ended and data Sockets used for connections are closed. The parent process closes the children by passing a #FIN DSS packet into the child pipe stream. This is not the best practice as a non-ideal DSS packet could potential contain this sequence and could be passed into the stream prematurely ending the connection. This could be overcome by creating an encoding method that prevent this sequence from occurring or modifying the child with redundancy to ensure a proper fin packet had been received. Hence, each state allows the program to address multiple connections as if acting as a single data stream.

The client iterates thru the entire data buffer incrementing the index at which the data is copied from the complete buffer in real time. For simplicity purposes this implementation does not rely on any error correction acknowledgment of received segments as a method of preventing packet loss. This implementation relies on the fact that TCP has reliable data transfer for the packets sent over the TCP connection in order to maintain synchronization of data flow. Thus, the data is simply feed into each state of the controller and then passed into appropriate data connections maintained by the children processes.

Child Methods

These child process are called after each connection is established and accepted by the server with the purpose of acting as a relay for the parent control process. These children simply loop repeatedly until they receive a packet indicating the data stream has completed. Each loop reads a pointer to the incoming channel of a pipe that is passed into the function from the calling parent thread struct. This pipe contains the 4 bytes of DSS packet data to be sent across the data channel. The pipe only contains the DSS segment because it has no other purpose but to act as a strait forward relay from pipe to TCP socket. As the child process performs each iteration it reads the incoming pipe and immediately writes the data to the outgoing TCP connection socket. This is followed by a brief sleep in order to maintain synchronization, again this is not the best practice but for brevity and simplicity it follows such a delay pattern. Changes were made to the child process to allow this time delay to be removed in the final version of the program design. Therefore, as the child receives data it immediately passes the information on to the server as the data is piped in from the controlling parent process.

Running the Program:

The client program can be ran with the command as follows-

./client –s 192.111.222.3 –a 2200 –b 2300 –c 2400 –d 2500

User must provide server IP, a control connection port and three data connection ports

The following is a list of function definitions and their underlying purpose.

Function Definitions for Client:

Void buildData(char *data) –

This function is responsible for creating the provided data structure containing the data sequence specified in the requirements for this lab. 16 revolutions of 0-9, a-z and A-Z. The program inputs a previously allocated character pointer sized correctly for the required data field. This pointer is used to place the incremented values within the allocated buffer.

Void forkConnection(int dataSocket, int *pipe) –

This function is responsible for creating individual child processes for each individual data connection to be maintained within. The process creates a fork and proceeds to loop while awaiting incoming data from the parent process. The pipe pointer represents a char array containing file pointers to the read and write channels of an initialized pipe sent from the control processes thread. The data socket parameter is used for communicating the received pipe data onto the server. As new data is read from the controller pipe it is immediately written into the socket and sent to the awaiting server connection.

Int conSendSocket()-

This function is very simple and is only responsible for setting up the desired control connection details before returning the created socket for later use in the main controller loop.

Void* conDataSocket(void* val) -

This function is responsible for establishing the actual connection between the server and client program via TCP/IP sockets. This is called as a thread and is meant to separate the established connection as an independent process. This function is passed a pointer to a struct which is given in the provided thread call. The use of threads allows for the connection processes to occur simultaneously with the server. This function is used to establish data connection 1.

Void* conData2Socket(void* val) -

This function is responsible for establishing the actual connection between the server and client program via TCP/IP sockets. This is called as a thread and is meant to separate the established connection as an independent process. This function is passed a pointer to a struct which is given in the provided thread call. The use of threads allows for the connection processes to occur simultaneously with the server. This function is used to establish data connection 2.

Void* conData3Socket(void* val) -

This function is responsible for establishing the actual connection between the server and client program via TCP/IP sockets. This is called as a thread and is meant to separate the established connection as an independent process. This function is passed a pointer to a struct which is given in the provided thread call. The use of threads allows for the connection processes to occur simultaneously with the server. This function is used to establish data connection 3.

“Server Side”

Server.c:

This section of the document includes a detailed explanation for the server side of this MP-TCP implementation program. The server is responsible for accepting incoming data over three connections in addition to accepting a control connection. The control connection accepts DSS packets containing information needed to reassemble incoming data from multiple connections. This program consist of a few separate routines for initialization, handling the creation/bind of sockets, accepting incoming connections for data or control, routine implementing a forked child process and a data parsing routine for capturing and reassembling the incoming data packets.

Initialization

The main program begins by performing a few initialization steps needed for setting up communication between the server and client. The first few steps involve the creation of struct containers for handling data passed into threads. The creation of an empty buffer for reconstructing the incoming data segments. The server initializes a set of four pipes storing the file pointer within the created struct as a char array. The server utilizes four threads during the course of operation, one thread is assigned for each incoming connection. During the initial stages of startup a pointer for the associated struct is passed to each of the threads during creation along with the thread function call. Hence, this the information contained in the created struct can then be passed into the created thread for later use.

Socket Creation

The main function completes the initial steps and moves into the portion of the program dealing with the creation and binding of sockets for accepting incoming connection from the client program. The main function calls a socket creation function which generates the control connection. This socket is stored in the earlier mentioned thread struct. For only the three data connections, main routine sets an indicator within the thread struct to indicate the data connection channel of one, two or three. This indicator is then handled by the underlying routine to accept the proper connection on the provided port.

Accepting Connections

The main program generates the needed threads passing a pointer to the populated struct as a parameter in the thread call. The threaded function is then responsible for accepting the incoming connections on the provided sockets. A connection is accepted and established between the server and the client for each of the four connections. This is followed by a call to initiate the parsing routine function and the three child connection processes. These calls are made from within the threaded processes ensuring independent function for each channel as it accepts the incoming connections. Thus, each function call is provided the socket information passed the thread from the struct and the pipe file pointers need to communicate between child processes and parsing routine.

Child Connection Routine

Each of the data connections utilizes a forked child process in order to read incoming data from the client. This is immediately relayed the through a pipe to the parsing routine awaiting the arrival of DSN data segments. The child process loops until receiving a specialized packet from the incoming data connections indicating the connection is finished. This of course is not the best method in practice as stated in the client explanation. Upon receiving the appropriate packet the connection finishes and closes the child process. Finally, the child process writes the special packet with the termination sequence into the pipe and the parsing routine handles the appropriate closures and deallocations.

Data Parsing and Reassembly Routine

This method could be considered the heart of the server and the main component involved with the overall server function. This routine is responsible for managing the incoming DSS packets and decoding the hexadecimal values representing the sequence number, starting index and data connection channel. A layout of the DSS packet structure can be seen in figure 3.

Figure 3:

4 bytes sequence number | 4 bytes DSN starting index | 1 byte for data channel

The DSS packet could easily implement a CRC routine for checking data integrity but this was omitted due to time concerns. The client encodes the appropriate sequence number and index number in hexadecimal before sending, but the data channel byte is left as a raw data type. The required values are decoded by the parsing routine and stored for use within the parsing method.

The parsing method loops continuously awaiting incoming data on the control connection. The incoming DSS packet provides instruction for the parsing routine. These instruction tell the parsing routine a new DSN segment is available on a specified data channel. The parsing routine reads the data pipe associated with the desired connection and retrieves the data from the child processes. This data is then coupled with the starting index and the construction buffer to place the received data in the proper final data buffer location. This process is repeated for each DSS/DSN segment passed from the client to the server. As the server receives each packet it uses the DNS information provided on the control connection to ensure the data is assembled correctly. This basic routine allows the server to reassemble the packets in the proper order despite possibly arriving out of intended order. This implementation does not provide ACK based error correction due to reliance on the RDT provided under TCP connections. This repeated until all packets for the desired data set have been received and the server handles the appropriate closures and deallocations.

Running the Program:

The Server program can be ran by using the following type command

./server –a 2200 –b -2300 –c 2400 –d 2500

The user must provide the appropriate control connection port and three data connection ports desired for establishing the desired communication.

The following is a list of function definitions and their underlying purpose.

Function Definitions for Server:

Int createControl() –

This function is responsible for creating the control socket on the provided port numbers. After establishing the socket connection it begins listening for incoming connections on the specified port value. On completion an integer file pointer to the bound socket connection is returned for use in the connection accepting function.

Int createData(int val) –

This function is responsible for creating the Data socket on the provide port numbers. After establishing the socket connection it begins listening for incoming connections on the specified port value. On completion an integer file pointer to the bound socket connection is returned for use in the connection accepting function. The input val specifies the values use in creating the data connection. This input val ranging in this case from 1 to 3 to signify the three additional connection implemented in the program design.

void* accData(void* val) –

This is a thread function used to accept the incoming control connection. This function is used by the server for receiving and establishing the connection request from the client. The input parameter is used to pass a pointer to the struct assigned for this thread. The pointer contains a link to a struct with data needed for calling a child creation function.

void* accControl(void* val)-

This is a thread function used to accept the incoming control connection. This function is used by the server for receiving and establishing the connection requests from the client. The input parameter is used to pass a pointer to the struct assigned for this thread. The pointer contains a link to a struct with data needed for calling a child creation function. This function differs from the control connection as the assigned struct contains an integer value indicating the appropriate data channel.

void forkConnection(int dataSocket, int *pipe) –

This function is responsible for creating individual child processes for each individual data connection to be maintained within. The process creates a fork and proceeds to loop while awaiting data bound for the parent process. The pipe pointer represents a char array containing file pointers to the read and write channels of an initialized pipe sent from the control processes thread. The data socket parameter is used for communicating the across the TCP connection receiving data from the server. As new data is read from the data connection it is immediately written into the pipe and sent to the awaiting server parsing method.

void dataParser(int *pipe1, int *pipe2, int *pipe3, int conSocket) –

This process is responsible for managing the incoming DSS and DSN packet received from the child data connections and client control connection. The function accepts an argument for each pipe linked to the

underlying child data processing functions. These pipes allow the data parser to communicate with the receiving functions independently as separate child processes. The control connection uses the conSocket argument to receive DSS packet from the client. This argument is generated with a call to a function used for creating the needed socket. This function is responsible for reassembling the incoming packets into their original ordering. This is done utilizing the received DSN packets for reassembly information.

Project Challenges and Difficulties:

This project had a number of significant challenges ranging from handling the multiple connections to figuring out how to create a system of pipes and sockets capable of communicating with one another. A great amount of difficulty was faced during the initial stages of developing the connections between the client and the server.

The first attempts at creating multiple connections failed repeatedly despite perfecting values and ensuring correctness of methods. The client and server programs displayed inconsistent behavior when attempting to establish multiple connections. The initial control connection would almost always connect successfully and the programs would randomly fail when attempting to connect with any additional data channels. Multiple attempts lead to research on possible causes and the resulting documentation provided some insight on possible reasons. Despite finding information on possible causes time restrictions caused the adoption of a new approach for handling multiple connections.

Multi-threading was adopted as a possible solution to adopting multiple socket connection from a single running process. The idea was developed to create a multi-threaded client and server which might make it possible to establish the needed connections. This discovery lead to the development of a program flow allowing a single process to spawn multiple socketed connections. The client and server utilized this inspiration for creation of independent threads capable of handling the desired tasks.

Implementing the routines needed for disassembling the data and reassembling the received data offered an additional challenge to the mix. This required creating a portable state like machine that could be adapted for use in both server and client systems.

Concluding Analysis and Evaluation:

Certain aspects of this project could be considered for improvement if given the desired time period. Considering this project was intended for completion by two individuals in the provided time period, the current amount of completion under one individual is viewed as an absolute success. Unfortunately, time restriction lead to the lack of ACK based error checking and the reliance of the program implementation on sleeping threads instead of creating flags or interrupt triggers. The original program runs much slower than a fully implemented version of MPTCP would run because of the inherent time delay placement needed to assure data integrity and synchronization of data feeds. The process immediately performs the retrieval of data instead of implementing a time delay to ensure the data is received before attempting to process the incoming data connection. Although this did somewhat hinder the speed at which the program operated this shortcoming does in no way effect the overall goal

of this project. The majority of the effort was spent ensuring the successful utilization of multiple connection when transfer data between two hosts. The time delay was eventually removed after fixing some issues within the child process. In the original time delay implementation the child processes were reading an incorrect amount of data causing overwriting issues and random buffered output. This issue was removed during the final stages of project completion and the resulting runtime is significantly less than the prior implementation.

This project can be consider an overall success when examining the requirements set forth in the project guidelines. This can be seen in the programs ability to send a single block of data characters across multiple TCP connections existing simultaneously between a server and client. This data is transmitted properly and is correctly reassembled by the server. This program behavior is a simplified version of the documented experimental MP-TCP shim protocol. Hence the program can be considered a success because it satisfies the required behavior described as MP-TCP style data transfer.

Works Cited

"TCP Extensions for Multipath Operation with Multiple Addresses." RFC 6824. Ed. A. Ford. Internet Engineering Task Force (IETF), Jan. 2013. Web. 23 Apr. 2016. <https://tools.ietf.org/html/rfc6824>.