1. BACKGROUND AND RATIONALE

1.1 Introduction

In recent years, the rapid increase of mobile computing devices like personal digital assistants (PDA), personal computers and laptops has driven a revolutionary change in the computing world. Internet services have grown rapidly and millions of people are using these services in their day to day life. The proliferation of mobile computing devices with improved processing capabilities allows mobile users to connect to the global Internet. The impact of this phenomenal growth changes the modality of communicating and increases its challenges. The most common issues are management of the wireless communication and poor performance of the protocols over wireless networks. These problems act as the major obstacles for the large-scale deployment of these technologies.

1.2 Transmission Control Protocol (TCP)

The Internet uses the Transmission Control Protocol (TCP) for communications over fixed node networks. Various modifications have been proposed to prevail over the difficulties of using it for mobile networks.

TCP is a set of rules used along with the Internet Protocol (IP) to transfer data between computers over the Internet [Kozierok 2005]. The IP is responsible for handling the actual delivery of the data. The message is divided into individual units of data called packets. TCP keeps track of the packets that are transferred to provide efficient routing through the Internet. TCP is known as connection-oriented protocol and provides a reliable service. Many services like HTTP, FTP, telnet, etc. use the transmission control protocol for communication over the Internet. This research concentrates on the way TCP deals with network congestion. TCP uses various mechanisms to avoid congestion and achieve high performance. These
mechanisms control the flow of traffic and keep the data flow below a rate that would cause the congestion. To avoid the network congestion TCP uses the following mechanisms [Allman 2009]:

**Slow-start:** During the exponential growth phase, TCP increases the window size by number of acknowledgement segments received.

**Congestion avoidance:** Continuously probes the network and checks the numbers of ACKs received, and in case of loss of packet TCP reduces the window size and reset it to slow-start state.

**Fast retransmit:** It determines the lost packet and reduces the time a sender waits to retransmit the lost packet.

**Fast recovery:** During congestion avoidance mode, when packets are not received, the congestion window size is reduced to the slow-start threshold, rather than the smaller initial value.

1.3 TCP in Mobile Computing

In the mobile computing environment there will be a combination of a wired networks and wireless networks as shown in Figure 1.1.

![Figure 1.1 Mobile Networks](image-url)

The wireless network is always prone to frequent disconnections because of high bit error rates and the frequent hand offs.

Since the traditional TCP protocol is designed for wired hosts, any delay in ACK for a transmitted segment of data is normally caused by congestion. To recover from the
congestion, TCP aggressively slows down the transmission of data through network by
invoking a congestion control algorithm. A congestion control algorithm reduces the sender’s
window (meaning it reduces the rate at which data is sent), which in turn will reduce the load
to the network and help to avoid congestion.

In mobile networks the delay in ACK may not be due to congestion, but the packets
might have been lost due to futile transmission of the wireless network or due to frequent
transmission errors suffered by the wireless link.

In a network which consists of wired and wireless networks, the data lost in a wireless
network might falsely trigger the congestion control algorithms in the wired part of the
network which will result in a significant reduction in the throughput and delays in an active
connection [Brown 1997]. A split connection approach is used to separate the wired network
from the wireless network which means there will be two connections, one for the wired and
another for the wireless network [Mondal 2007].

Normally when a data segment is lost in a wireless network no ACK is sent back to the
fixed host in the wired network, so the wired Host TCP thinks that the packets are lost due to
congestion and it starts invoking the congestion control algorithm. It reduces the window size
(i.e. the data being transmitted in the wired network) and tries to retransmit the same packet
until the ACK is received [Choong 2002].

The TCP provides a connection oriented service that depends upon the IP addresses of
the sender and receiver. This feature of TCP leads to several shortcomings. The major
shortcomings with the TCP in wireless networks are wasting of available capacity during
slow-start process, and corrupting packets due to the high error rate. Disconnections are
common in mobile networks which result in a loss of packets. Another problem is serial
timeouts which result from continuous retransmission of packets at the time of connection loss [Brown 1997]. The performance of the TCP protocol in the wired network and wireless network differs as shown in Table 1.1. All these observations point to the fact that TCP is not suited for a growing class of applications in the wireless environment.

Table 1.1 Comparison of TCP/IP over wired and wireless networks

<table>
<thead>
<tr>
<th>Wired networks</th>
<th>Wireless networks</th>
</tr>
</thead>
<tbody>
<tr>
<td>High bandwidth (about 100Mbps)</td>
<td>Reduced bandwidth</td>
</tr>
<tr>
<td>Random packet loss is negligible</td>
<td>Higher loss rate due to its vulnerability to interference and disconnection</td>
</tr>
<tr>
<td>High performance rate</td>
<td>Low performance rate</td>
</tr>
</tbody>
</table>

This project describes a proposed algorithm called Modified Mobile Transmission Protocol which is an enhancement to the Mobile TCP. This protocol retains all the salient and necessary features of TCP related to flow control. It uses a split connection approach to implement the protocol in a mixed wired network and wireless network. The MMTCP improves the performance of the TCP connections when a mobile node uses the services across the Internet. The proposed algorithm provides better performance than the traditional TCP.

1.4 Related Work

Several mechanisms have been proposed to improve the performance of TCP over the wireless network. Different properties between the wired network and wireless network are considered by these mechanisms.
1.4.1 Snoop Protocol

The Snoop Protocol is a TCP protocol designed to improve the performance of TCP over networks that have both wired and wireless links. This protocol deals with the problem of packet loss due to network congestion [Balakrishnan 1997]. The Snoop protocol works by deploying a Snoop agent at the base station and performing retransmissions of lost segments based on duplicate TCP acknowledgments (which are a strong indicator of lost packets) and locally estimated last-hop round-trip times [Balakrishnan 2009].

The end-to-end semantics of the transport layer connection is maintained in Snoop protocol [Amir 1995] [Balakrishnan 1997]. The packets passed across the wired-wireless link are buffered at the base station. The buffered packets are used to retransmit unacknowledged packets and reduce the timeouts by suppressing the duplicate ACKs. When an ACK is received from the mobile host, Snoop distinguishes it as genuine, spurious or duplicate and performs the appropriate action. Snoop protocol avoids time outs and maintains a larger value of TCP’s congestion window, thus resulting in better throughputs [Vangala 2003]. It can improve TCP performance quite well in wireless links but has a problem; when there are no duplicate ACKs, the Snoop protocol cannot notice the packet loss until the local retransmission timer is expired [Cho 2005] and cannot detect the disconnection due to handoff.

1.4.2 Mobile Transmission Control Protocol

With the shortcomings discussed in the above section, enhancements have been made to the TCP on mobile networks. M-TCP is an extension of TCP for mobile networks, which can support multimedia services over high bandwidth [Ferriera 2003]. This protocol provides
good performance in case of frequent disconnections, with changing bandwidths, and low bit wireless links.

The TCP is divided into Mobile TCP (M-TCP) and Supervisor Host TCP (SH-TCP). The mobile hosts (MH) communicate with each node in the mobile station and these nodes are controlled by the supervisor host (SH), which is connected to the wired network and responsible for routing and handles other protocols information. Figure 1.2 shows how the TCP is split. At the senders side the TCP protocol is unchanged and at the supervisor host it uses the modified TCP called SH-TCP. The M-TCP is used for communication between the MH and SH.

The end-to-end semantics are maintained in MTCP approach. The ACK of the packet transmitted between the wired networks is not sent to the fixed host (wired network) unless the packet is transmitted to the mobile host (wireless network). When the packet is received by the Mobile host, an ACK is sent back to the fixed host in the wired network.

M-TCP [Brown 1997] proposes a solution to modify the TCP at the mobile host side called Mobile TCP, when it does not receive the ACK from the mobile host side. M-TCP
sends the sender (fixed host) to the persistent state where it sets its window size to zero and in this state the sender does not suffer from the time out and does not slowly reduce its window size. Instead, it sets it to zero and when the mobile host reconnects it sends a greeting packet with the ACK for the previously sent segment of data. Once the greeting packet is received the congestion window size is set to previous window size.

There are certain drawbacks with the M-TCP protocol. When a packet is lost, it is retransmitted by the wired network, which is retransmitted back by wireless network. This process of retransmission of lost packets is most suitable when the mobile host moves out of the range. This model reduces the throughput when there is a data bit error.
2. NARRATIVE

2.1 Modified Mobile Transmission Control Protocol and its Benefits

Modified Mobile Transmission protocol is an algorithm which improves the performance of TCP throughput over the wireless Internet based on the end-to-end approach. The goal in developing MM-TCP (Modified M-TCP) protocol is to lower the error rate by means of dynamic connection migration. Benefits of the MM-TCP protocol of using the MM-TCP are:

1. Improves TCP performance in mobile networks.
2. Maintains end-to-end semantics.
3. Ability to deal with the problems caused by frequent disconnections.
4. Adaption to dynamically changing bandwidth over the already starved wireless link.
5. Ensuring that handoffs are efficient.

2.2 Approach

The previous section describes the various shortcomings of TCP and M-TCP in the context of wireless networks. This research project proposes a Modified M-TCP protocol that provides a solution to these problems. The drawback with the M-TCP protocol is that, when a packet is lost, it is retransmitted by wired networks. In case of bit errors and data loss in wireless network we don’t have to force the wired network to retransmit because there is no ACK from the mobile host. We could just as well try to transmit the data multiple times so that even if one of segment is in error another might reach the mobile host. The M-TCP model does not consider much about the data lost during the transmission in wireless network. If there is no ACK, it makes the sender to retransmit the packet again. This reduces the throughput.

In Modified M-TCP (MM-TCP), the M-TCP at the mobile host side is modified and this allows transmitting the data multiple times with a small interval. The time interval will be way
less than the timer at the sender side (fixed host). This fixes the problem of high Bit error rates and disconnections invoking the wired network to retransmit the packet again thereby increasing the throughput.

The wired network does not have to retransmit the data lost in the wireless network due to the high bit error rate and frequent disconnections unless the user gets disconnected for a long time i.e., more than the round-trip-time (RTT).

This is a simple client server model suitable for evaluating the performance of the MM-TCP algorithm. This project implements the normal TCP connection with congestion avoidance, MTCP and Modified Mobile Transmission Control Protocols. The performance is evaluated by comparing the results from these three protocols, and tries to prove that the Modified M-TCP increases the throughput. Most network applications are divided into two pieces: a client and a server. A client is the side that initiates the communication process, whereas the server responds to the incoming client requests. In this model, communication involves the server side and client side as the endpoints of the connections. At any point of during the service session, the client might suffer loss in quality of service on the connection associated with the server. This might be due to several reasons like network congestion, server load, and loss of packets. This will reduce the performance of the network.

2.3 TCP Application Development Environment

TCP (Transmission Control Protocol) provides a reliable end-to-end service that delivers packets over the Internet. Packets are delivered in sequence without loss or duplication. TCP/IP applications are constructed using the sockets interface. Figure 2.1 explains the client server communication [Mitchell 2009]. The server application creates a socket and it listens for TCP connection request from the clients. When an application wishes to communicate with a
destination server using a TCP connection, the client creates the socket and opens in active mode. In this mode, a TCP connection will be attempted with the server. Client initializes the TCP connection and transfers the data. On receiving the data segments the server responds back with ACKs for the client. The client sends TCP-ACK to the server and releases the connection.

Figure 2.1: Client Server Conversation [Mitchell 2009]

2.4 User Interface

This is a client server approach. The client talks to the server by using a socket connection. The server will listen for socket connections. TCP connection with congestion

Figure 2.2: Server on the Mobile Host.
avoidance, MTCP and Modified Mobile Transmission Control Protocols are implemented in this project. The server on the Mobile Host side is initiated first and the range of error segments is provided as the input as shown in Figure 2.3. The TCP connection will be established between the client and server. The data packets will be sent from the client to the server. The client on the Fixed Host side will show how the packets are transmitted. The connection will be closed when the transfer is finished.
3. PROPOSED RESEARCH

Modified M-TCP is a solution to improve the performance of TCP throughput over the wireless networks. This protocol is proposed for heterogeneous wired and wireless networks.

3.1 Network Model

Implementation of Modified M-TCP is a simple client-server model suitable for evaluating performance of the Modified M-TCP algorithm. In this TCP protocol, a closed loop network can be considered. Figure 3.1 shows the network model and its network elements. The fixed host (FH) sends the data and uses the ACKs received as a feedback from the network to increase or decrease its congestion window size ($cwnd$). The ACKs received from the network are used in TCP congestion control and flow control mechanisms. Hence, in case of disconnections a signal should be sent as indication. Then TCP could react appropriately by preventing unnecessary deflation of the $cwnd$. In this way the available bandwidth of the network can be preserved for other TCP communications. During the data transfer the WH connects to the intermediate host (IH) through a WLAN link, the IH is connected to the wired network [Brown 1997].

3.2 Split Connection

For implementing M-TCP, the split connection approach is used because it is most suitable with the design as mentioned in the previous sections. It allows modifying TCP on the mobile network to increase the throughput in disconnections and varying bandwidth in wireless network [Brown 1997].

Each of the TCP connections is split into two at the sender host (SH) as shown in Figure 3.1. The TCP is unmodified at the TCP sender on the fixed network while sending the data to the SH. SH uses the modified TCP for delivering the data to the mobile host (MH). The TCP client at the
SH receives segments transmitted by the sender and it passes these segments to the M-TCP client for delivery to the MH.

3.3 Design of the M-TCP

The primary goal in designing M-TCP is to keep the TCP sender's congestion open in case of disconnections at MH and to be able to recover from the losses due to disconnections and to eliminate serial timeouts. In M-TCP, the protocols at both ends can be modified [Brown 1997].

Figure 3.2 shows the state diagram of M-TCP. The TCP client sends the data and waits for the ACK. M-TCP monitors the flow of ACKs from the MH. If the ACK is not received then it assumes that the connection is lost. If the connections are lost then it sets the congestion window to persistent state. This will ensure that the disconnections do not cause the M-TCP to invoke congestion control. This will ensure that the data and acknowledgements are not lost during the disconnections. When it regains the connection, the M-TCP at MH sends a specially marked ACK to M-TCP at the SH. If the ACK are delayed then it asks to retransmit the data. When a retransmit timeout occurs, the FH is set to persistent state. When an ACK is not received, it
assumes that the mobile receiver is disconnected temporarily. Hence the retransmissions are futile until the MH notifies the SH that it is reconnected [Brown 1997].

![State Transmission Diagram of M-TCP](image)

Figure 3.2: State Transmission Diagram of M-TCP [Brown 1997].

### 3.4 Design of Modified M-TCP

In Modified M-TCP protocol, the M-TCP at the SH on the wireless portion of the connection, is modified to improve the throughput. Figure 3.3 shows a state diagram of the Modified M-TCP. When the data is sent from the FH, it waits for the ACK. It makes the sender to send the data multiple times at M-TCP with a small interval. This time interval is less than the timer at the sender. In the process of resending the data packets, the duplicate ACKs are rejected when received. When the retransmission timer expires then the M-TCP is moved to persistent
state. Then it waits for the ACK from the receiver. Since M-TCP at SH only enters the persistent state when the data is not acknowledged, the receiver will send an ACK to remove it from persistent state.

The data is sent from the FH and it waits for the ACK. In the Modified M-TCP the data is sent multiple times from the M-TCP end at the base station to the MH and waits for the ACKs. At

Figure 3.3: State Transmission Diagram for Modified M-TCP [Brown 1997].
the MH end, it will check for the duplicate ACKs. If the data is received by the MH then it will send an ACK back to the FH. If the ACK is not received then it assumes that the connection is lost. If the connections are lost then the FH goes to the persistent state and resends the data until it receives the ACK.

3.5 Designing Issues

The implementation of Modified M-TCP includes the following features:

1. Connection Establishment

   For a connection originating on a fixed network, firstly the TCP connection between the fixed host and supervisor host must be completed [Brown 1997]. Then initiate and complete the connection between the SH and MH. The connection is initialized by calling the connect method of the Socket. Then the remote address is set by the connect method, and sends the SYN packet. At the server side, the application will call the listen method before the server starts listening for incoming SYN packets. The client continues sending the packets and ensures the three-way handshake communication is done. From the connection point of view, the SH is made transparent to the TCP sender in the fixed network. On connection setup, the SH creates sockets with local address bonded to both TCP sender and MH's addresses.

2. Congestion Window Settings

   A timer function determines when to send a window size reduction update to the TCP sender. At a given time, the timer should expire allowing the SH to generate a packet, and for the packet to propagate to the TCP sender before the sender invokes the congestion control and timer expires.

Brown made an assumption that, “the TCP sender retransmission timer is set to the retransmission timeout (RTO) at fixed host. The fixed host RTO is based on the sender’s round-
trip propagation delay (RTT) estimated between itself and the MH. Since the SH is situated between the TCP sender and the MH, it has RTT and hence RTO estimates for both segments of the connection” [Brown 1997]. The researcher will use this assumption to set the congestion window timer.

3. Slow-Start Mechanism

The application will use the slow-start mechanism. The congestion window size is not reduced on timeouts because they do not occur due to congestion. Due to this, the slow-start behavior is limited to the beginning of the connection.

3.6 Major Components

3.6.1 Programming Language Used

Like many other programming languages .Net can be used to write and develop client / server applications. Because of concerns for security, the developers of the popular browsers, such as Netscape, have built into their systems restrictions on the IP addresses to which an applet can connect. Consequently, .Net client/server programs are often written as applications, rather than as applets. Fortunately, there is a whole range of new base classes in System.Net.Sockets that provide a rich set of functionality, abstracting much of the lower - level Windows Sockets API and making it much easier to write code like this [Bromberg 2009]. Writing programs that access the network used to be a relatively difficult task. With .NET, this is no longer the case. The .NET Framework class library includes two namespaces that are full of classes that help you with networking: System.Net and System.Net.Sockets [Kurniawan 2009]. Computers running on the Internet communicate to each other using either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP). Typically, you don't need to concern yourself with the TCP and UDP layers. Instead, you can use the classes in System.Net.Sockets package. These classes
provide system-independent network communication. However, to decide which classes your programs should use, you do need to understand how TCP and UDP differ. The TCP uses socket to communicate over the network.

In client-server applications, the server provides some service. The client uses the service provided by the server. The communication that occurs between the client and the server must be reliable. That is, no data can be dropped and it must arrive on the client side in the same order in which the server sent it. TCP provides a reliable, point-to-point communication channel that client-server application on the Internet use to communicate with each other. To communicate over TCP, a client program and a server program establish a connection with one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection. The server program begins by creating a new Socket object to listen on a specific port. The System.Net.Sockets class that provides a system-independent implementation of the server side of a client/server socket connection [Kurniawan 2009].

3.6.2 The Framework for Computing

In network communications typically the data is sent by using protocols. In this project the Modified M-TCP protocol is used for communication. The Mobile Host is started and the minimum and maximum error segments range is provided as input. The Supervisor host is started. The Fixed Host is also started. The connection is set
Figure 3.4: Starting the Mobile Host(MH).

Figure 3.5: Starting the Supervisor Host(SH)
up between the client and server. After initializing, the connection data is transmitted. The hosts at the three stations are started as shown in Figure 3.4. When the fixed host is started, it asks for the path of the log file to be created as shown in Figure 3.6. The status of the communication is stored in the log file. A graph is drawn with the simulated results and then the comparisons are made between the M-TCP and Modified M-TCP protocols.
4. TESTING AND EVALUATION

This section discusses how the performance of the proposed protocol is evaluated.

4.1 Unit Testing

The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. Unit testing can be used to test the individual components. The first step is to implement the Fixed Host, Supervisor host and Mobile Host, then verify whether they are working correctly or not. Unit tests evaluate the success or failure of design changes to ensure that the client application works and does not introduce errors. To test, one has to follow the following steps [Helix 2009]:

1. Client initiates the TCP connection
2. Transfer the data
3. Client closes the TCP connection

Test Procedures:

In Mobile Host Application Console

• Check if the Mobile Host application console is started.
• Check if the MH asks to input the maximum and minimum error segments information.
• Check if the Socket connection is opened.
• Check if the MH outputs how many bytes of data is received.
• Socket connection should be closed.

Test Cases:

1. Start the Mobile Host.

Result:
2. Give the Input

Figure 4.2: Giving the input.

In the Supervisor Host application

Test Case:

3. Start the SH
Figure 4.3: Output when Starting the SH.

In the Fixed Host application

- Create Client_Socket object and Check if socket connection is initialized correctly.
- The socket change should be changed to "closed".
- Check if the ActiveOpen message handler is called.
- Send the data and check if the message handler is called.
- Check if ACKs are received.
- Send FIN to socket.
- Socket connection should be closed.
- Delete the socket object.

Test Case:

4. Start the Fixed Host

Result:
4.2 Performance Testing

The performance of Modified M-TCP is tested against the M-TCP and TCP protocols. The data of various sizes are transferred and the time taken to transfer the data is taken. The experiment is conducted to get a measure of the times taken by the different data packets. A graph is plotted to compare the performance of the Modified M-TCP with M-TCP.

This project tests the Modified M-TCP performance against M-TCP. The time to transfer a data from the fixed host TCP sender (FH) to mobile host (MH) is measured. A graph is plotted with data transfer time against the maximum error segments accepted range. And this proves that the Modified M-TCP is optimal.

For example, a data of 3MB size is sent from the FH sender to MH. The user enters the maximum and minimum error segments allowed as an input at the MH console.
Figure 4.5: Entering the Range of Error Segments.

Figure 4.6: Enter the Path for the Log File.
Figure 4.7: After entering the Log Path

Figure 4.8: Sending the Data and Receiving the Ack’s
Figure 4.9: Screenshot showing how the Data is Transmitted when an ACK is not Received in M-TCP

Figure 4.10: Screenshot showing the Sending of Data Segments.
The time taken for each data segment is stored in the log file and is totaled to find the time to transfer the complete data. The transfer time is measured with various inputs of number of error segments. A graph is drawn with the output by taking number of error segments on x-axis and data transfer time in y-axis.

The performance of Modified M-TCP is compared with the original M-TCP in various test cases. The performance is evaluated in terms of data transfer time. The tests are designed to validate some of the decisions made during the implementation regarding the algorithms, as this is reflected in the values obtained for the total time for transfer.

The experiment is conducted by sending the data of 3MB. The FH will send data chunks to the MH. At the MH side the minimum and maximum data loss range will be taken as input. Each time the data is transferred between the FH and MH. For each data transfer, the time taken to transfer the data with different error segments ranges is taken down as shown in Table 4.1. The
results are shown in a graph. The results show that the time to transfer a data from the fixed host to mobile host is optimal by using Modified M-TCP protocol.

Table 4.1: Time taken to Transfer the Data.

<table>
<thead>
<tr>
<th>Error segments</th>
<th>100</th>
<th>200</th>
<th>300</th>
<th>400</th>
<th>500</th>
<th>600</th>
<th>700</th>
<th>800</th>
<th>900</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-TCP</td>
<td>182.977</td>
<td>213.377</td>
<td>255.122</td>
<td>255.734</td>
<td>256.838</td>
<td>256.834</td>
<td>261.548</td>
<td>277.135</td>
<td>258.934</td>
<td>265.050</td>
</tr>
<tr>
<td>Modified M-TCP</td>
<td>151.356</td>
<td>188.644</td>
<td>251.143</td>
<td>253.348</td>
<td>254.582</td>
<td>255.736</td>
<td>257.108</td>
<td>263.855</td>
<td>257.110</td>
<td>259.641</td>
</tr>
</tbody>
</table>

Simulation has been done with M-TCP and Modified M-TCP and a graph is drawn with the data. The graph has been drawn by taking the time taken to transfer the data on X-axis and the number of error segments on Y-axis as shown in Figure 4.12. From the above data and the Graph we can see that the modified M-TCP performs better than M-TCP when the bit errors are there in wireless network.

Figure 4.12: M-TCP Vs Modified M-TCP
5. CONCLUSIONS AND FUTURE WORK

5.1. Concluding Remarks

This project describes the design and implementation of Modified M-TCP. Modified M-TCP provides a framework for building highly available services over the Internet. The necessity for such a protocol is perceptible from the promising classes of applications being built over the Internet. A TCP protocol is the best protocol for communicating over the networks. With this project the performance of the TCP can be made better even in wireless networks. The goal in developing Modified M-TCP protocol is to lower the data transmission time. The salient features of the protocol are to improve the TCP performance in mobile networks and adapt to dynamically changing bandwidth, frequent disconnections and handoffs over the wireless link. The Modified M-TCP algorithm reduces the data transfer time, increases the throughput, and reduces retransmission ratio compared to TCP and M-TCP.

5.2 Future Work

In the current design, a basic assumption is the error segments given as the input are continuous. This may not be entirely true in some communication where the packets may be corrupted anywhere in the data. One of the future tasks would be to test the simulation by sending the error segments randomly, and analyze the new possibilities that arise from this synergy. With the help of this simulation with random distribution of error packets we can have better performance.
BIBLIOGRAPHY AND REFERENCES


APPENDIX A

TCP BASICS

Cumulative Acknowledgements:
A new cumulative acknowledgement is generated only on receipt of a new in sequence packet.

Delayed Acknowledgements:
An ack is delayed until
  • another packet is received, or
  • delayed ack timer expires (200 ms typical)

Duplicate Acknowledgements:
A dupack is generated whenever an out-of-order segment arrives at the receiver
A packet is lost

Congestion Window:
  • It is determined by the sender, based on feedback from the network
  • Congestion window size bounds the amount of data that can be sent per round-trip time

Throughput \( \leq \frac{\text{Window Size}}{\text{RoundTripTime(RTT)}} \)

How does TCP detect a packet loss?
  • By Retransmission Timeout (RTO)
    - At any time, TCP sender sets retransmission timer for only one packet
    - If acknowledgement for the timed packet is not received before timer goes off, the packet is assumed to be lost
    - RTO is dynamically calculated (RTO = mean + 4 mean deviation)
  • By Duplicate Acknowledgements
    - TCP sender assumes that a packet loss has occurred if it receives three dupacks consecutively

Timeout Granularity
  • RTT (Round Trip time) is measured as a discrete variable, in multiples of a “tick”
  • 1 tick = 500 ms in many implementations
  • smaller tick sizes in more recent implementations (e.g., Solaris)
  • RTO is at least 2 clock ticks
APPENDIX B

M-TCP Code:

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using System.Diagnostics;
using System.Threading;
using MyTcpSockets;

namespace Sh
{
    public class MTcpShFhBehaviour : MyTcpSockets.DefaultBehaviour
    {
        MyTcpSocket socket;
        public void setMyTcpSocket(MyTcpSocket socket)
        {
            this.socket = socket;
        }
        public override List<int> handleGetACKNums(List<int> ackNums, List<SimpleTCPacket> lastAcks)
        {
            Stopwatch sw = new Stopwatch();
            List<int> receivedAckNums = new List<int>();
            sw.Start();
            while (receivedAckNums.Count < ackNums.Count)
            {
                if (sw.Elapsed.TotalMilliseconds >= (double)Constants.ACK_TIMEOUT)
                {
                    return receivedAckNums;
                }
                #region Checking acks
                lock (lastAcks)
                {
                    List<SimpleTCPacket> successAcks = new List<SimpleTCPacket>;
                    foreach (SimpleTCPacket ackPack in lastAcks)
                    {
                        int ind = ackNums.FindIndex(new Predicate<int>(delegate(int curAckNum)
                        {
                            return curAckNum == ackPack.ack_num;
                        }));
                        if (ind >= 0)
                        {
                            receivedAckNums.Add(ackPack.ack_num);
                            successAcks.Add(ackPack);
                        }
                    }
                    foreach (SimpleTCPacket pack in successAcks)
                    {
```

(34 characters remain to be entered.)
lastAcks.Remove(pack);

#region
Thread.Sleep(10);
#endregion
sw.Stop();
return receivedAckNums;

public override bool SendDataPart(Dictionary<int, byte[]> segments, int count)
{
    List<int> ackNums = new List<int>();
    List<int> receivedAckNums = new List<int>();
    int ackCounter = 1;
    bool success = true;
    foreach (int key in segments.Keys)
    {
        ackNums.Add(key + Constants.PACKET_LEN);
        Console.WriteLine("Sending segments {0}", key);
        socket.SendDirect(segments[key], false);
        // Packet time counter
        socket.writeStatsSendDataPart(key);
        count--;
        if (ackCounter == 3)
        {
            foreach (int item in socket.GetACKNums(ackNums))
            {
                receivedAckNums.Add(item);
            }
            if (receivedAckNums.Count < 3 && count >= 3)
            {
                success = false;
                break;
            }
            ackCounter = 1;
            ackNums.Clear();
        } else
        {
            ackCounter++;
        }
        if (count == 0)
        {
            break;
        }
    }
    if (ackCounter > 1)
    {
        foreach (int item in socket.GetACKNums(ackNums))
        {
            receivedAckNums.Add(item);
        }
    }
    // TODO: Getting ack, which also deletes acknowledged elements
    foreach (int ackNum in receivedAckNums)
    {
        // ...
if (success && !segments.ContainsKey(ackNum - Constants.PACKET_LEN))
{
    success = false;
} else
{
    segments.Remove(ackNum - Constants.PACKET_LEN);
}
} return success && receivedAckNums.Count > 0;
}

using System;
using System.Collections.Generic;
using System.Text;
using System.Diagnostics;
using System.Threading;
using MyTcpSockets;

namespace Sh
{
    public class MTcpShMhBehaviour : MyTcpSockets.DefaultBehaviour
    {
        private IMhAckPacketListener ackPacketListener;
        MyTcpSocket socket;
        public void setMyTcpSocket(MyTcpSocket socket)
        {
            this.socket = socket;
        }
        public MTcpShMhBehaviour(IMhAckPacketListener ackPacketListener)
        {
            this.ackPacketListener = ackPacketListener;
        }
        public override List<int> handleGetACKNums(List<int> ackNums, List<SimpleTCPPacket> lastAcks)
        {
            return new List<int>();
        }
        public override bool onAckPacketReceived(SimpleTCPPacket pack)
        {
            bool isSpecialAck = MyTcpSocket.CheckFlags(pack,
                (int)TCPFlags.MTCP_CONNECTION_REGAINED_ON_MH);
            ackPacketListener.onMhAckPacketReceived(pack.ack_num, isSpecialAck);
            return true;
        }
        public override bool SendDataPart(Dictionary<int, byte[]> segments, int count)
        {
            foreach (int key in segments.Keys)
            {
                Console.WriteLine("Sending segments {0}", key);
                socket.SendDirect(segments[key], true);
            }
socket.writeStatsSendDataPart(key);

    }
    segments.Clear();

    return true;

}

using System;
using System.Collections.Generic;
using System.Text;
using MyTcpSockets;
using System.Diagnostics;
using System.Threading;

namespace Sh
{
    public class ShMTcp : IMhAckPacketListener, IDisposable
    {
        public enum ShrinkState { Noshrink, Shrunk, Recovering };
        public enum FreezingState { Nofreeze, Frozen };

        public int tcpRcvNext = 0;
        public int tcplastack = 0;
        public int mtcplastack = 0;
        public int sndmax = 1;
        private bool IsDisposed = false;
        private Thread timerThread;
        private volatile bool shouldStopTimerThread = false;
        private Object syncObj = new Object();

        public ShrinkState shrinkState = ShrinkState.Noshrink;
        public FreezingState freezingState = FreezingState.Nofreeze;

        private List<AckData> acksForSend = new List<AckData>();

        IAckSender ackSender;
        IRetransmitterOfPackets retransmitterOfPackets;
        IPersistPacketSender persistPacketSender;
        private MyTcpSocket mhSocket;
        private MyTcpSocket fhSocket;

        private MyTcpSockets.Timer shrinkTimer = new MyTcpSockets.Timer(\n            Constants.M_TCP_ACK_RECEIVE_FROM_MH_TIMEOUT);
        private MyTcpSockets.Timer freezeTimer = new MyTcpSockets.Timer(\n            Constants.RETRANSMIT_TIMEOUT);
        private MyTcpSockets.Timer persistTimer = new MyTcpSockets.Timer(\n            Constants.PERSIST_TIMER_TIMEOUT);

        public ShMTcp()
        {
            fhSocket = new MyTcpSocket();
            MTcpShFhBehaviour mTcpShFhBehaviour = new MTcpShFhBehaviour();
            mTcpShFhBehaviour.setMyTcpSocket(fhSocket);
            fhSocket.setBehaviour(mTcpShFhBehaviour);
        }
    }
}
mhSocket = new MyTcpSocket();
MTcpShMhBehaviour mTcpShMhBehaviour = new MTcpShMhBehaviour(this);
mTcpShMhBehaviour.setMyTcpSocket(mhSocket);
mhSocket.setBehaviour(mTcpShMhBehaviour);
}

public void Initialize(int localListenPort, int localSendPort, int window)
{
    fhSocket.Initialize(localListenPort, 512, window, 100);
mhSocket.Initialize(localSendPort, 512, window, 0);
    setAckSender(fhSocket);
    setRetransmitterOfPackets(mhSocket);
    setPersistPacketSender(mhSocket);

    timerThread = new Thread(this.timerDoWork);
timerThread.Start();
}

public void setAckSender(IAckSender ackSender)
{
    this.ackSender = ackSender;
}

public void setRetransmitterOfPackets(IRetransmitterOfPackets retransmitter)
{
    retransmitterOfPackets = retransmitter;
}

public void setPersistPacketSender(IPersistPacketSender packetSender)
{
    this.persistPacketSender = packetSender;
}

public void setPersistTimer(MyTcpSockets.Timer timer)
{
    this.persistTimer = timer;
}

public void setShrinkTimer(MyTcpSockets.Timer timer)
{
    this.shrinkTimer = timer;
}

public void setFreezeTimer(MyTcpSockets.Timer timer)
{
    this.freezeTimer = timer;
}

public void PassiveOpen()
{
    fhSocket.PassiveOpen();
}

public void ActiveOpen(string ipAdress, int portNum)
{
    mhSocket.ActiveOpen(ipAdress, portNum);
}

public byte[] Receive(ref int seqNumb)
return fhSocket.Receive(ref seqNumb);}

public void Send(byte[] data, int seqNumb)
{
    onSendPacketToMh(seqNumb);
    mhSocket.Send(data, seqNumb);
}

public void SendACK()
{
    fhSocket.SendACK();
}

public MyConnectionState FhConnectionState
{
    get
    {
        return fhSocket.ConnectionState;
    }
}

public MyConnectionState MhConnectionState
{
    get
    {
        return mhSocket.ConnectionState;
    }
}

public void onMhAckPacketReceived(int ackNumber, bool isSpecialAck)
{
    lock (syncObj)
    {
        shrinkTimer.reset();
        if (ackNumber > mtcplastack)
        {
            if (shrinkState == ShrinkState.Noshrink || shrinkState == ShrinkState.Shrunk)
            {
                if (ackNumber == tcpRcvNext)
                {
                    tcplastack = ackNumber;
                    mtcplastack = ackNumber;
                    Console.WriteLine("");
                }
                else
                {
                    tcplastack = ackNumber - 1;
                    mtcplastack = ackNumber;
                }
            }
            else
            {
                shrinkState.
            }
        }
    }
}

return fhSocket.Receive(ref seqNumb);
if (shrinkState == ShrinkState.Shrunk)
{
    shrinkTimer.start();
    shrinkState = ShrinkState.Recovering;
}

acksForSend.Add(new AckData(tcplastack, false));
}

if (sndmax == ackNumber)
{
    freezeTimer.reset();
} else {
    freezeTimer.start();
    if (isSpecialAck)
    {
        retransmitterOfPackets.retransmitPackets();
    }
    freezingState = FreezingState.Nofreeze;
    Out.DebugPrintCached("ShMTcp.onMhAckPacketReceived()", "sndmax = " + sndmax + "; isReconnectAck = " + isSpecialAck);
}

public void onSendPacketToMh(int seqNumb)
{
    lock (syncObj)
    {
        freezeTimer.start();
        if (seqNumb + Constants.PACKET_LEN > sndmax)
        {
            sndmax = seqNumb + Constants.PACKET_LEN;
        }
        shrinkTimer.start();
        if (seqNumb + Constants.PACKET_LEN > tcpRcvNext)
        {
            if (ShrinkState.Recovering == shrinkState)
            {
                shrinkState = ShrinkState.Noshrink;
            }
            tcpRcvNext = seqNumb + Constants.PACKET_LEN;
        }
    }
}

private void timerDoWork()
{  
  while (!shouldStopTimerThread)
  {
    checkTimerExpired();
    Thread.Sleep(10);
  }
}

public void checkTimerExpired()
{
  lock (syncObj)
  {
    if (shrinkTimer.isExpired() && shrinkState == ShrinkState.Noshrink
        && (tcplastack != mtcplastack && tcpRcvNext != mtcplastack))
    {
      Out.DebugPrintCached("ShMTcp.CheckTimerExpired()
congestion",
"tcplastack=" + tcplastack + "; mtcplastack=" + mtcplastack
+ "; tcpRcvNext=" + tcpRcvNext + "; ");
      acksForSend.Add(new AckData(mtcplastack, true));
      tcplastack = mtcplastack;
      shrinkTimer.start();
      shrinkState = ShrinkState.Shrunk;
    }
    sendAcks();
    if (freezeTimer.isExpired())
    {
      freezeTimer.reset();
      persistTimer.start();
      freezingState = FreezingState.Frozen;
    }
    if (persistTimer.isExpired())
    {
      persistPacketSender.sendPersistPacket();
      persistTimer.reset();
    }
    Out.DebugPrintCached("ShMTcp.checkTimerExpired() timers
state",
"ShrinkState: " + shrinkState
+ "; FreezingState: " + freezingState);
  }
}

private void sendAcks()
{
  foreach (AckData ackData in acksForSend)
  {
    if (ackSender != null)
ackSender.sendAck(ackData.ackNum, ackData.persistantState);
}

acksForSend.Clear();

~ShMTcp()
{
    Dispose(false);
}

public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

protected void Dispose(bool Disposing)
{
    if (!IsDisposed)
    {
        if (Disposing)
        {
            mhSocket.Dispose();
            fhSocket.Dispose();

            if (timerThread != null)
            {
                shouldStopTimerThread = true;
                timerThread.Join();
            }
        }
    }
    IsDisposed = true;
}

private struct AckData
{
    public AckData(int ackNum, bool persistantState)
    {
        this.ackNum = ackNum;
        this.persistantState = persistantState;
    }

    public int ackNum;
    public bool persistantState;
}
}
MM-TCP Code:

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using System.Diagnostics;
using System.Threading;

namespace MyTcpSockets
{
    public class MTcpMhBehaviour : DefaultBehaviour
    {
        private MyTcpSocket socket;
        private bool isPersistantState = false;
        private Object lockObj = new Object();
        private int lastAck = 0;
        private int receiveAcksTimeout = 0;

        public void setMyTcpSocket(MyTcpSocket socket)
        {
            this.socket = socket;
        }

        public override List<int> handleGetACKNums(List<int> ackNums,
                         List<SimpleTCPacket> lastAcks)
        {
            Stopwatch sw = new Stopwatch();
            List<int> receivedAckNums = new List<int>();

            sw.Start();
            while (receivedAckNums.Count < ackNums.Count)
            {
                waitUntilPersistantState();

                if (sw.Elapsed.TotalMilliseconds >=
                    (double)receiveAcksTimeout)
                {
                    return receivedAckNums;
                }

                #region Checking acks
                lock (lastAcks)
                {
                    List<SimpleTCPacket> successAcks = new List<SimpleTCPacket>();
                    foreach (SimpleTCPacket ackPack in lastAcks)
                    {
                        int ind = ackNums.FindIndex(new Predicate<int>(
                          delegate (int curAckNum)
                          {
                            return curAckNum == ackPack.ack_num;
                          }));
                    }
                }
            }
        }
    }
}
```
if (ind >= 0)
{
    receivedAckNums.Add(ackPack.ack_num);
    successAcks.Add(ackPack);
}
foreach (SimpleTCPPacket pack in successAcks)
{
    lastAcks.Remove(pack);
}
#endregion
Thread.Sleep(10);
sw.Stop();
return receivedAckNums;

private void waitUntilPersistantState()
{
    for (; ; )
    {
        lock (lockObj)
        {
            if (!isPersistantState)
            {
                break;
            }
        }
        Thread.Sleep(10);
    }
}

public override bool SendDataPart(Dictionary<int, byte[]> segments, int count)
{
    List<int> ackNums = new List<int>();
    List<int> receivedAckNums = new List<int>();
    int ackCounter = 1;
    bool success = true;
    foreach (int key in segments.Keys)
    {
        ackNums.Add(key + Constants.PACKET_LEN);
        Console.WriteLine("Sending segments {0}", key);
        socket.SendDirect(segments[key], false);
        // Packet time counter
        socket.writeStatsSendDataPart(key);
        count--;
        if (ackCounter == 3)
receiveAcksTimeout = 

Constants.SEND_THREE_PACKETS_TIMEOUT;

foreach (int item in socket.GetACKNums(ackNums))
{
    receivedAckNums.Add(item);
}

if (receivedAckNums.Count < 3 && count >= 3)
{
    success = false;
    break;
}

ackCounter = 1;
ackNums.Clear();

else
{
    ackCounter++;
}

if (count == 0)
{
    break;
}

}

if (ackCounter > 1)
{
    receiveAcksTimeout = Constants.ACK_TIMEOUT;
    foreach (int item in socket.GetACKNums(ackNums))
    {
        receivedAckNums.Add(item);
    }
}

// TODO: Getting ack, which also deletes acknowledged elements
foreach (int ackNum in receivedAckNums)
{
    if (success && !segments.ContainsKey(ackNum -
    Constants.PACKET_LEN))
    {
        success = false;
    }
    else
    {
        segments.Remove(ackNum - Constants.PACKET_LEN);
    }
}

return success && receivedAckNums.Count > 0;

}

public override bool onAckPacketReceived(SimpleTCPHeader pack)
{
    lock (lockObj)
    {
        if (pack.ack_num > lastAck)
        {
            isPersistantState = pack.cwnd == 0;
        }
    }
    return true;
}
lastAck = pack.ack_num;

return true;

public override void onPersistPacketReceived(SimpleTCPacket pack) {
    socket.sendSpecialMarkedAckPacket();
}

public override void onPacketLoss() {
    socket.sendSpecialMarkedAckPacket();
}

public override void onDataPacketReceived(SimpleTCPacket pack) {
    socket.AddNeedsAck(pack);
}

using System;
using System.Collections.Generic;
using System.Text;
using MyTcpSockets;

namespace Mh {
    class MhMTcpSocket {
        private MyTcpSockets.MyTcpSocket myTcpSocket;

        public MhMTcpSocket(int portNum, int windowSize, int threshold, int seq_num) {
            myTcpSocket = new MyTcpSocket();
            myTcpSocket.Initialize(portNum, windowSize, threshold, seq_num);
            MTcpMhBehaviour behaviour = new MTcpMhBehaviour();
            behaviour.setMyTcpSocket(myTcpSocket);
            myTcpSocket.setBehaviour(behaviour);
        }

        public void PassiveOpen() {
            myTcpSocket.PassiveOpen();
        }

        public MyTcpSockets.MyConnectionState ConnectionState { get
        }

{
    return myTcpSocket.ConnectionState;
}

public byte[] Receive()
{
    int seqNumb = 0;
    return myTcpSocket.Receive(ref seqNumb);
}

public void Send(byte[] data)
{
    myTcpSocket.Send(data);
}

public void SendACK()
{
    myTcpSocket.SendACK();
}

public void ActiveOpen(string ipAdress, int portNum)
{
    myTcpSocket.ActiveOpen(ipAdress, portNum);
}

public void setPacketsForLose(List<int> packetsSeqNumsForLost)
{
    myTcpSocket.setPacketsForLose(packetsSeqNumsForLost);
}

using System;
using System.Collections.Generic;
using System.Text;
using System.Diagnostics;
using System.Threading;

namespace MyTcpSockets
{
    public class TcpBehaviour : DefaultBehaviour
    {
        private MyTcpSocket socket;
        private bool isPersistantState = false;
        private Object lockObj = new Object();
        private int lastAck = 0;
        private int receiveAcksTimeout = 0;

        public void setMyTcpSocket(MyTcpSocket socket)
        {
            this.socket = socket;
        }
    }
public override List<int> handleGetACKNums(List<int> ackNums, List<SimpleTCPacket> lastAcks)
{
    Stopwatch sw = new Stopwatch();
    List<int> receivedAckNums = new List<int>();

    sw.Start();
    while (receivedAckNums.Count < ackNums.Count)
    {
        waitUntilPersistantState();
        receivedAckNums =
        getAckNumsWhichLessOrEqualThanLastReceivedAckNumb(ackNums);

        if (sw.Elapsed.TotalMilliseconds >=
            (double)receiveAcksTimeout)
            return receivedAckNums;
    }
    sw.Stop();
    return receivedAckNums;
}

private List<int>
getAckNumsWhichLessOrEqualThanLastReceivedAckNumb(List<int> ackNums)
{
    List<int> lst = new List<int>();
    int lastReceivedAckNumb = socket.LastReceivedAckNumb;
    foreach (int ackNum in ackNums)
    {
        if (ackNum <= lastReceivedAckNumb)
        {
            lst.Add(ackNum);
        }
    }
    return lst;
}

private void waitUntilPersistantState()
{
    for (; ; )
    {
        lock (lockObj)
        {
            if (!isPersistantState)
            {
                break;
            }
        }
    }
}
Thread.Sleep(10);
}

public override bool SendDataPart(Dictionary<int, byte[]> segments, int count)
{
    List<int> ackNums = new List<int>();
    List<int> receivedAckNums = new List<int>();
    int ackCounter = 1;
    bool success = true;
    foreach (int key in segments.Keys)
    {
        ackNums.Add(key + Constants.PACKET_LEN);
        Console.WriteLine("Sending segments {0}", key);
        socket.SendDirect(segments[key], false);
        // Packet time counter
        socket.writeStatsSendDataPart(key);
        count--;
        if (ackCounter == 3)
        {
            receiveAcksTimeout = Constants.SEND_THREE_PACKETS_TIMEOUT;
            foreach (int item in socket.GetACKNums(ackNums))
            {
                receivedAckNums.Add(item);
            }
            if (receivedAckNums.Count < 3 && count >= 3)
            {
                success = false;
                break;
            }
            ackCounter = 1;
            ackNums.Clear();
        }
        else
        {
            ackCounter++;
        }
        if (count == 0)
        {
            break;
        }
    }
    if (ackCounter > 1)
    {
        receiveAcksTimeout = Constants.ACK_TIMEOUT;
        foreach (int item in socket.GetACKNums(ackNums))
        {
            receivedAckNums.Add(item);
        }
    }
    // TODO: Getting ack, which also deletes acknowledged elements
    foreach (int ackNum in receivedAckNums)
if (success && !segments.ContainsKey(ackNum - Constants.PACKET_LEN))
{
    success = false;
}
else
{
    segments.Remove(ackNum - Constants.PACKET_LEN);
}

return success && receivedAckNums.Count > 0;

public override bool onAckPacketReceived(SimpleTCPPacket pack)
{
    lock (lockObj)
    {
        if (pack.ack_num > lastAck)
        {
            isPersistantState = pack.cwnd == 0;
            lastAck = pack.ack_num;
        }
        return true;
    }
}