CONGESTION CONTROL AND PACKET REORDERING FOR MULTIPATH TRANSMISSION CONTROL PROTOCOL

BY

NIRNIMESH GHOSE

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering
in the Graduate College of the
Illinois Institute of Technology

Approved _____
Advisor

Chicago, Illinois
May 2012

ACKNOWLEDGMENT

This dissertation would not have been possible without the ever-present guidance and the constant help of some individuals who contributed in the best way possible and extended their valuable assistance in the preparation and completion of this research work.

First and foremost, my utmost gratitude and humble acknowledgement to my mentor and adviser, Dr. Tricha Anjali, Associate Professor in the Electrical and Computer Engineering Department, whose sincerity and encouragement I will never forget. Dr. Anjali has been my inspiration as I hurdle all the obstacles in the completion of this study.

I also thank Dr. Sanjeev Kapoor, Professor in Computer Science Department at Illinois Institute of Technology, Chicago for his immense suggestions towards the findings in my thesis work.

My special thanks to Dr. Kui Ren, Dr. Ken Zdunek and Dr.Jafar Saniie, for being in the thesis committee and providing necessary guidance and insights whenever needed.

Last but not the least, my mother Dr. Ranjana Ghose and the One above all of us, the omnipresent GOD for answering my prayers, for giving me strength to plod on amidst every difficulties that came in the way.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Increase in number of connectivity to internet lead to the development of Multipath Transmission Control Protocol or MPTCP. MPTCP, as proposed by the IETF working group mptcp, allows a single data stream to be split across multiple path. This has obvious benefits for reliability, and it can also lead to more efficient use of networked resources. But major problem in MPTCP is the congestion control and packet reordering at the destination. Congestion control for MPTCP is given by the Wischik et al. [9]. But the packet reordering at the destination is not considering which can drastically affect the throughput for the protocol. Therefore, in this work various available packet reordering techniques available for single path TCP are tested for multipath situation. These algorithms are Duplicate Selective Acknowledgement or DSACK, Eifel and Forward Retransmission Timeout or FRTO. A simple topology is simulated in NS-3 and measurement is taken for various path characteristics to see which algorithm works best for the multipath scenario.

CHAPTER 1

INTRODUCTION

MPTCP, as proposed by the IETF working group mptcp, allows a single data stream to be split across multiple path. This has obvious benefits for reliability, and it can also lead to more efficient use of networked resources.

1.1 Overview

Earlier connections generally had only one path to transfer data from source to destination. For the level 3 Transmission Control Protocol was developed. It helps to form a virtual connection between source and destination on level 3 for transfer of data in form of packets. But with advent of new mobile devices which have more than one connections between the source and destination like Ethernet, WiFi, 3G or 4G. Newer technology was required to use them at the same time to improve the throughput. For instance, laptops have usually at least both a wired (Ethernet) and a wireless (WiFi) network adapters. Similarly smartphones and tablet PCs can reach the Internet either through WiFi or through a cellular network (UMTS or 3G+). This lead internet task force to develop the Multipath Transmission Control Protocol or MPTCP [9].

A lot of studies have considered the implementation of multipath capabilities at different layers: at the application layer [2], at the transport layer [3] [8], etc. The last two references shows that transport layer can be the best layer to implement the multipath protocol.

1.2 Multipath Transmission Control Protocol

Multipath transport protocols have the potential to greatly improve the performance and resilience of Internet traffic flows. The basic idea is that if flows are able to simultaneously use more than one path through the network, then they will

be more resilient to problems on particular paths (e.g. transient problems on a radio interface), and they will be able to pool capacity across multiple links. These multiple paths might be obtained for example by sending from multiple interfaces, or sending to different IP addresses of the same host, or by some form of explicit path control.

The design of Multipath-capable flows should be such that they shift their traffic from congested paths to uncongested paths, hence the Internet will be better able to accommodate localized surges in traffic and use all available capacity. Multipath congestion control is that the source and destination take on a role that is normally associated with routing, namely moving traffic onto paths that avoid congestion hotspots. When a flow shifts its traffic onto less congested paths, then the loss rate on the less congested path will increase and that on the more congested path will decrease; the overall outcome with many multipath flows is that the loss rates across an interconnected network of paths will tend to equalize. This is a form of load balancing, or more generally resource pooling [8].

Multipath congestion control should be designed to achieve a fair allocation of resources. For example, if a multipath flow has four paths available and they all happen to go through the same bottleneck link, and if we simply run TCPs congestion avoidance independently on each path, then this flow will grab four times as much bandwidth as it should. In fact, the very idea of shifting traffic from one path to another in the previous paragraph presupposes that there is some fair total traffic rate, and that extra traffic on one path should be compensated for by less traffic on the other. The problem of fairness is made even harder by round-trip-time dependence.

In multipath context, packets may also arrive out-of-sequence as the different paths may have different characteristics (especially the end-to-end delay), or congestion state (and then different queuing delays). The out-of-sequence arrival will create a problem for MPTCP while re-assembling packets at the connection level, and not at the subflow level because subflows are independent.

1.3 NS-3

Network Simulator-3 [6] is a discrete-event network simulator in which the simulation core and models are implemented in C++. NS-3 is built as a library which may be statically or dynamically linked to a C++ main program that defines the simulation topology and starts the simulator. NS-3 also exports nearly all of its API to Python, allowing Python programs to import an NS3 module in much the same way as the NS-3 library is linked by executables in C++. The NS-3 project is committed to building a solid simulation core that is well documented, easy to use and debug, and that caters to the needs of the entire simulation workflow, from simulation configuration to trace collection and analysis.

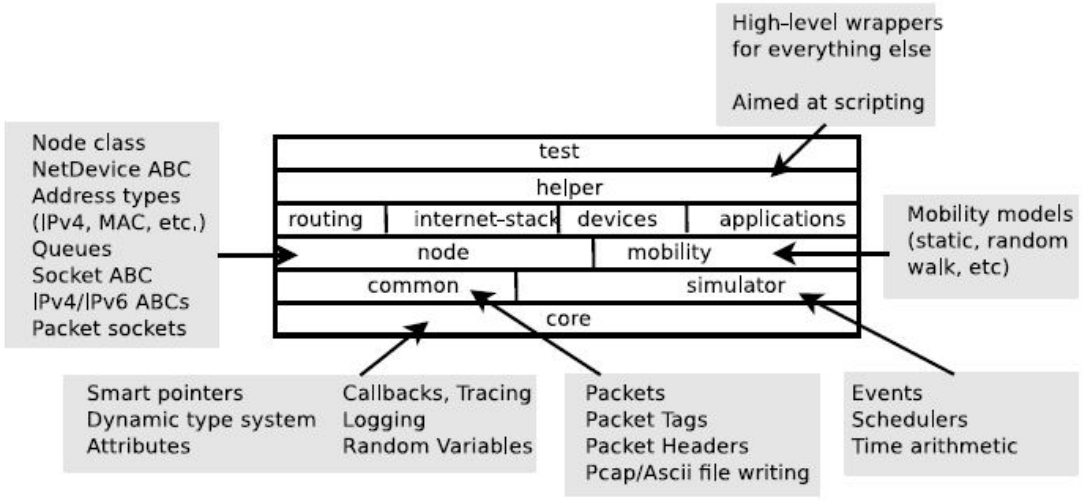


Figure 1.1. Software organization of ns-3

Furthermore, the NS-3 software infrastructure encourages the development of simulation models which are sufficiently realistic to allow NS-3 to be used as a realtime network emulator, interconnected with the real world and which allows many existing real-world protocol implementations to be reused within NS-3. The ns-3 simulation core supports research on both IP and non-IP based networks. However, the large majority of its users focuses on wireless/IP simulations which involve models for Wi-Fi, WiMAX, or LTE for layers 1 and 2 and a variety of static or dynamic routing

protocols such as OLSR and AODV for IP-based applications.

NS-3 also supports a real-time scheduler that facilitates a number of "simulation-in-the-loop" use cases for interacting with real systems. For instance, users can emit and receive NS-3-generated packets on real network devices, and NS-3 can serve as an interconnection framework to add link effects between virtual machines. Another emphasis of the simulator is on the reuse of real application and kernel code. Frameworks for running unmodified applications or the entire Linux kernel networking stack within NS-3 are presently being tested and evaluated.

CHAPTER 2

MULTIPATH TRANSMISSION CONTROL PROTOCOL

Multipath Transmission Control Protocol (MPTCP) is a future internet design which effectively make use of simultaneous multiple paths on the transport layer to transfer data between end nodes. Although present day protocols like TCP Santa Cruz do make use of multi homing to get data across between end nodes, they have certain limitations like usage of single paths at any given moment of time or compatibility with middleboxes. In such circumstances Multipath TCP, essentially used for rate control, promised to be a better alternative. The fact that it is not a TCP alternative but an extension to present day TCP itself should be an added advantage.

2.1 Main Mechanisms

The transport layer in Multipath TCP is divided into two sub layers. The upper layer collects the functionalities for connection management (establishing connections, reordering packets, closing connection etc.). The lower layer controls a set of sub-flows that can be seen each as one single TCP flow. Multipath TCP also manages two spaces of sequence number, one for each sub-layer. Like standard TCP, each subflow has its own sequence space which identifies bytes within a subflow. At the connection level, another sequence space is used to reorder the TCP segments before sending them to the Application layer. The Multipath TCP protocol uses new TCP options to exchange signaling information between peers [1]:

- MPC (Multipath Capable) is used during the three-way handshake to establish a MultipathTCP connection.
- DATA FIN is used to inform the remote peer of the end of data and to close the multipath TCP connections.
- ADD and Remove address are used to inform the remote peer of the availability

of a new address or to ask it to ignore an existing one.

- JOIN is used to initiate a new subflow between a not already used couple of addreses.
- DSN (Data Sequence Number) is used to map between the subflow level and the data sequence space number.

2.2 Connection Establishment

The source application sends a Connect call, the transport layer establishes a connection with the destination peer which was waiting for receiving connection requests. The establishment is TCP-like (three way handshake) with the use of Multipath Capable option to inform the destination that the source is capable of exchanging data using Multipath TCP. To initiate a new subflow, the peers must first exchange their additional IP addresses. The current Multipath TCP draft does not specify how the exchange happens. Maybe additional segments are send having the ADDR (Add address) [1] option to establishment of the Multipath TCP connection.

2.3 Subflow Initiation

Figure 3 shows the initiation of a new subflow and the presence of a JOIN in a SYN segment. To maximize the chance that the subflow under initiation takes a path which is disjoint with previously established path, each IP address is only used by one subflow.

2.4 MPTCP Code Decription

The basic code used to implement using code from [5]. Then I made made changes in the code to implement new techniques and topologies. Main Parts of the MPTCP code are:

- *MpTcpSocketImpl* is a subclass of NS-3 class *TcpSocketImpl*. It provides to

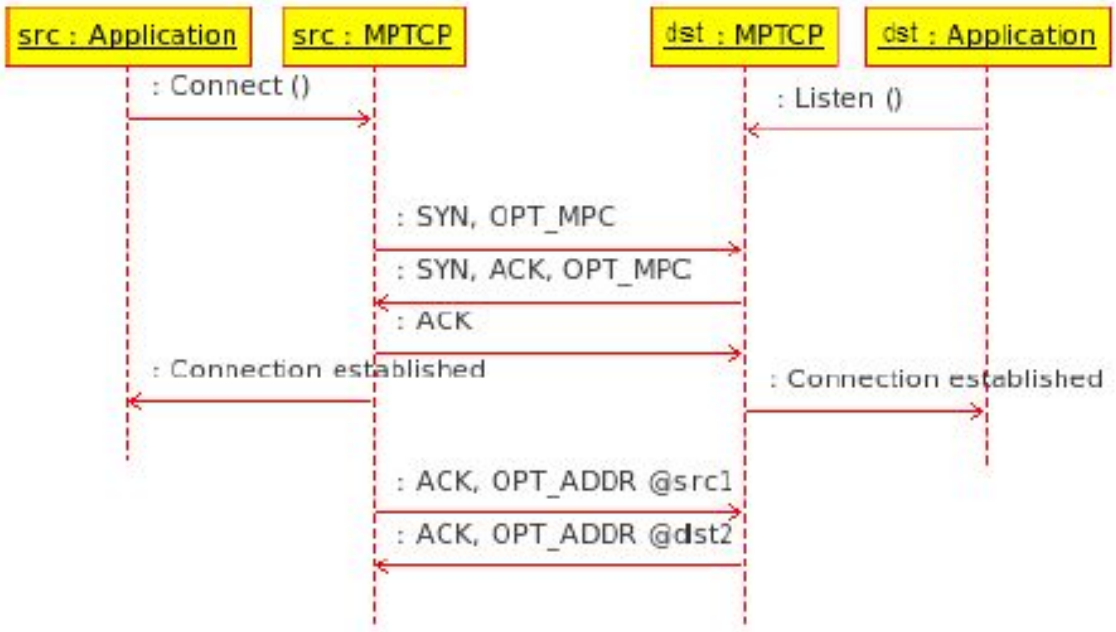


Figure 2.1. Multipath TCP connection establishment

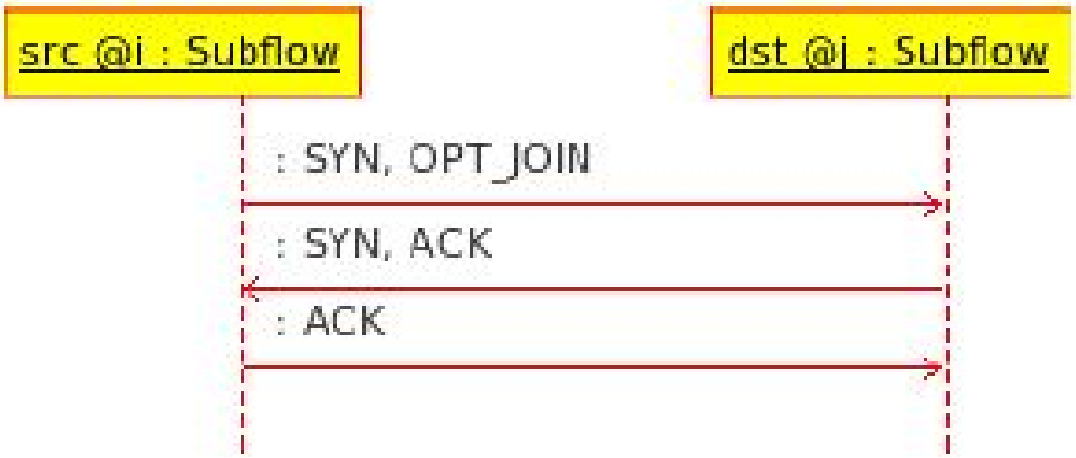


Figure 2.2. Multipath TCP subflow initiation

the application layer a MPTCP API (connect, bind, etc.) to manage Multipath TCP connection. It also implements the packet reordering algorithms described previously.

- *MpTcpL4Protocol* is a subclass of the NS-3 class *TcpL4Protocol*. It is an

interface between the multipath transport layer and network layer.

- $MpTcpSubflow$ represent a subflow of a MPTCP connection.
- $MpTcpHeader$ is subclass $TcpHeader$. An instance of this class is a TCP Header that can handle operation like TimeStamp, MPC, etc.

CHAPTER 3

MPTCP CONGESTION CONTROL

Different congestion control algorithms are needed for Multipath TCP, as single path algorithms have a series of issues in the multipath context. One of the prominent problem is that running existing algorithms such as standard TCP independently on each path would give the multipath flow more than its fair share at a bottleneck link traversed by more than one of its subflow. Further, it is desirable that a source with multiple paths available will transfer more traffic using the least congested of the paths, achieving a property called resource pooling where a bundle of links effectively behaves like one shared link with bigger-capacity.

Bottleneck fairness is just one requirement multipath congestion control should meet. The following three goals as per IETF draft capture the desirable properties of a practical multipath congestion control algorithm:

- Goal 1 (Improve Throughput): A multipath flow should perform at least as well as a single path flow would on the best of the paths available to it.
- Goal 2 (Do no harm): A multipath flow should not take uo more capacity from any of the resources shared by its different paths, than if it was a single flow using only one of these paths. This guarantees it will not unduly harm other flows.
- Goal 3 (Balance congestion): A multipath flow should move as much traffic as possible off its most congested paths, subject to meeting first two goals.

Goals 1 and 2 together ensure fairness at the bottleneck. Goal 3 captures the concept of resource pooling.

3.1 Uncoupled TCP

This is the normal TCP with additive increase and multiplicative decrease. At the start of a connection, an exponential increase is used, as it is immediately after a retransmission timeout.

- Increase w_r by $1/w_r$ per ack on path r.
- Decrease w_r by $w_r/2$ per loss event on path r.

3.2 Linked Increase

In this algorithm [10] the increase of the congestion window size depend on the α which is calculated according to the formula given in above equation. The decrease in the congestion window size is same as normal TCP.

- Increase w_r by α/w_r per ack on path r
- Decrease w_r by $w_r/2$ per loss event on path r

$$\alpha = \hat{w}(\frac{max_r\sqrt{\hat{w}_r}/RTT_r}{\Sigma_r\hat{w}_r/RTT_r})^2 \tag{3.1}$$

3.3 Fully Coupled

This algorithm [10] uses the increase of congestion window size according to normal TCP. While the decrease in the congestion window is dependent on the total window size of all the subflows.

- Increase w_r by $(1/w)$ per ack on path r.
- Decrease w_r to $max(w_r - w/b, 1)$ per loss event on path r; if there is truncation then do a timeout but no slow-start; use $b = 2$ to mimic TCP.

3.4 RTT Compensation

A connection consists of set of subflows R, which may take a different route through the Internet. Each subflow r R maintains its own congestion window. This algorithm as proposed in [10] is used in the MPTCP congestion control as per RFC by Internet Task Force.

- Increase w_r by $min(\alpha/w, 1/w_r)$ per ack on path r.
- Decrease w_r to $w_r/2$ per loss event on path r

$$\alpha = \hat{w}(\frac{max_r\sqrt{\hat{w}_r}/RTT_r}{\Sigma_r\hat{w}_r/RTT_r})^2 \tag{3.2}$$

CHAPTER 4

PACKET REORDEING TECHNIQUES

Standard TCP has jitter with large packets in network. This means that when the end to end delay vary a lot, packets may arrive out of sequence. This may be case for example for wireless network, the mobile devices may change the used hotspot for accessing the Internet. The reordering of a packet makes the receiver responding with duplicated acknowledgements, and this may induce the sender to infer wrongly a packet loss and do re-transmission. To avoid this problem and to distinguish clearly between packet losses due to congestion in the network and reordering due to transmission jitter, many mechanism are proposed in TCP. But all these mechanism are for a single path TCP transmission. Thus it requires a testing of all these mechanism which can be used in the multipath TCP scenario.

In multipath context, packets may also arrive out-of-sequence as the different paths may have different characteristics such as data rate and end-to-end delay or congestion state. The out-of-sequence arrival will create a problem for MPTCP while re-assembling packets at the connection level, and nit at the subflow level because subflows are independent.

4.1 DSACK Algorithm

This algorithm [11]is based on the SACK (Selective Acknowledgement) option. At the reception of a segment that creates a hole in the sequence numbers, the receiver sends back a duplicated acknowledgement containing a SACK option. The first block in the SACK option refers to the segment which triggers this duplicated acknowledgements. After three duplicated acknowledgements, the sender retransmits the missing segments, saves the congestion window value, and then enters a congestion avoidance phase. After that, when the sender detects that the retransmission segment was acknowledged twice, it infers a spurious retransmission and begin a DSACK slow

start to the stored congestion windows value.

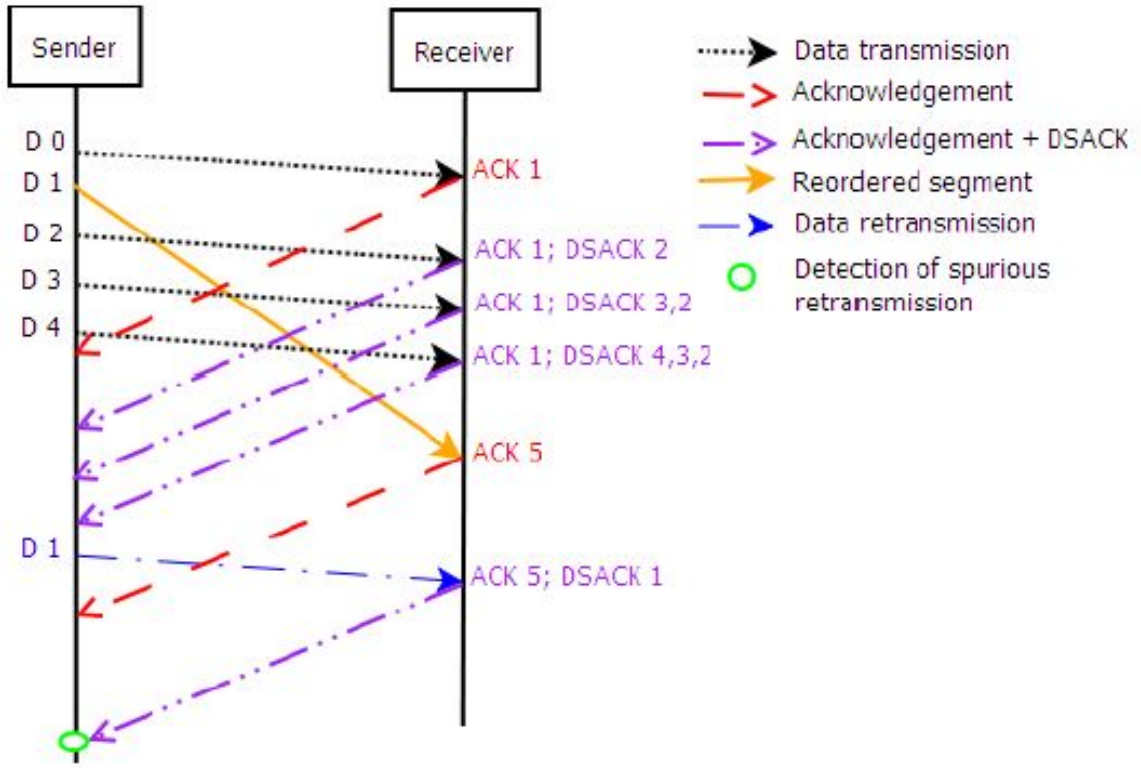


Figure 4.1. DSACK Algorithm

4.2 Eifel Algorithm Acoording to the algorithm [4] the sender inserts a TCP timestamp option in each transmitted segment, and the receiver inserts the timestamp value of the received segment in the corresponding acknowledgement. In case of loss, the sender saves the values of current congestion window and the slow start threshold. Then the sender retransmits the missing segment and stores its timestamp value. When the sender receives as acknowledgment for a retransmitted segment, it compares the saved timestamp value with the one inserted in the acknowledgement. If the first one is greater then the retransmission is considered spurious and the values of current congestion window and slow start threshold are restored.

4.3 F-RTO Algorithm

The guideline behind Forward Retransmission Timeout Recovery Algorithm

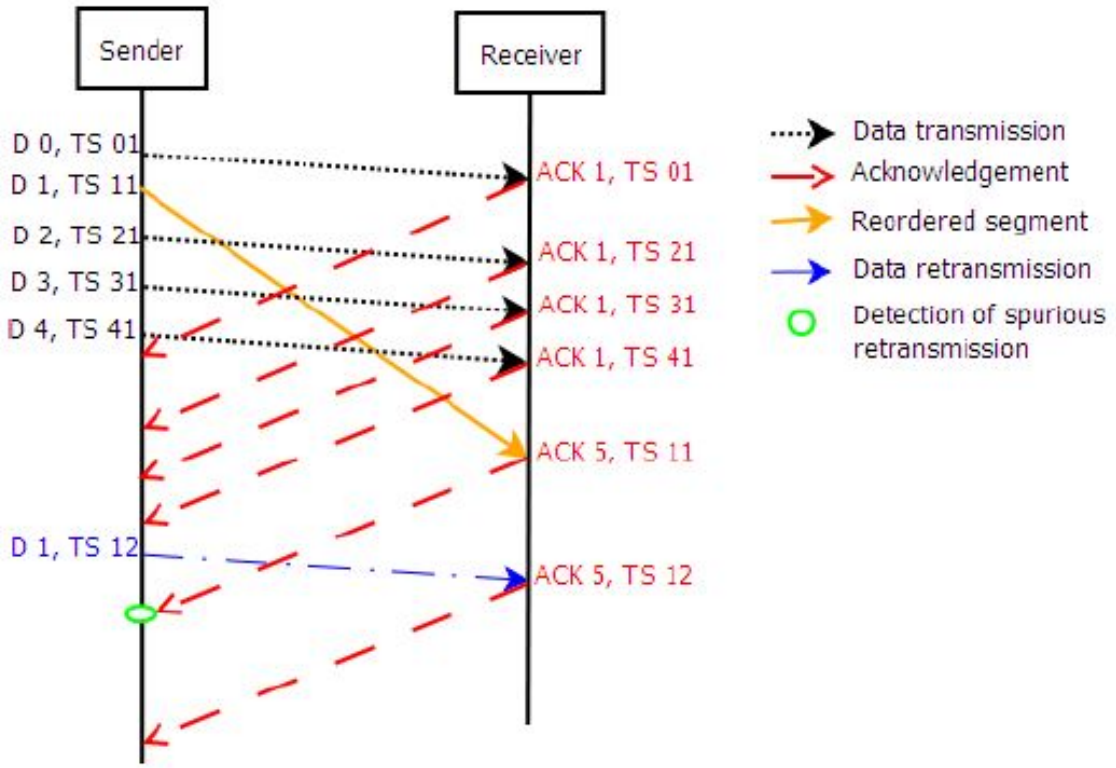


Figure 4.2. Eifel Algorithm

[7] is, that an Retransmission Timeout either indicates a loss, or it is caused by excessive delay in packet delivery while there still are outstanding segments in flight. If the RTO was due to delay, i.e. the RTO was spurious, acknowledgements for non-retransmitted segments sent before the RTO should arrive at the sender after the RTO occurred.

When the retransmission timer expires, the F-RTO algorithm takes the following steps at the TCP sender. In the algorithm description below:

- When the retransmission timer expires, retransmit the segment that triggered the timeout.
- When the first acknowledgement after RTO arrives at the sender, the sender chooses the following actions depending in whether the ACK advances the win-

dow or whether it is a duplicate ACK.

  - If the acknowledgement advances SND.UNA (send unacknowledged), transmit up to two new (previously unsent) segments.
  - If the acknowledgment is duplicate ACK, set the congestion window to one segment and proceed with the conventional RTO recovery.

- When the second acknowledgment after the RTO arrives, either continue transmitting new data, or start retransmission with the slow start algorithm, depending on whether data was acknowledged.

  - If the acknowledgement advances SND.UNA, continue transmitting new data following the congestion avoidance algorithm.
  - If the acknowledgement is a duplicate ACK, set congestion window to three segments, continue with the slow start algorithm retransmitting unacknowledged segments.

CHAPTER 5

RESULT

There are three main topologies on which we tested the various congestion control and packet reordering techniques that are already present. We have used Equal Cost MultiPath (ECMP) in the topology 2 and topology 3 to route the packets in various subflow available. In all topologies a 10mb of data is sent with sender buffer as 14kb and reciever buffer as 20kb. Equal Cost MultiPath is a ipv4 routing which calculates all the path available. We have used it to recalculate the routing table every 0.01ms.

5.1 Topology 1

In this topology there is direct connection between clent and server. There are three subflows namely Subflow 0, Subflow 1 and Subflow 2. The links are 2mbps, 5 mbps and 7 mbps respectively with each having path delay of 100 ms.The worst result is obtained by UncoupledTCP congestion control and no packet reordering and best result is obtained by RTT_Compensator congestion control and FRTO packet reordering.

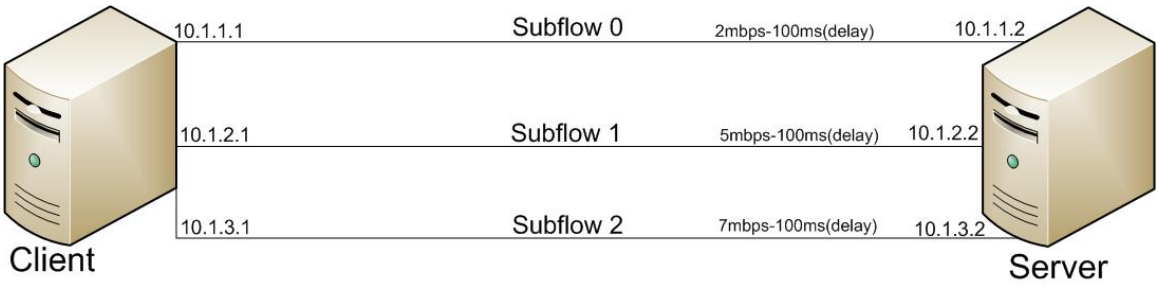
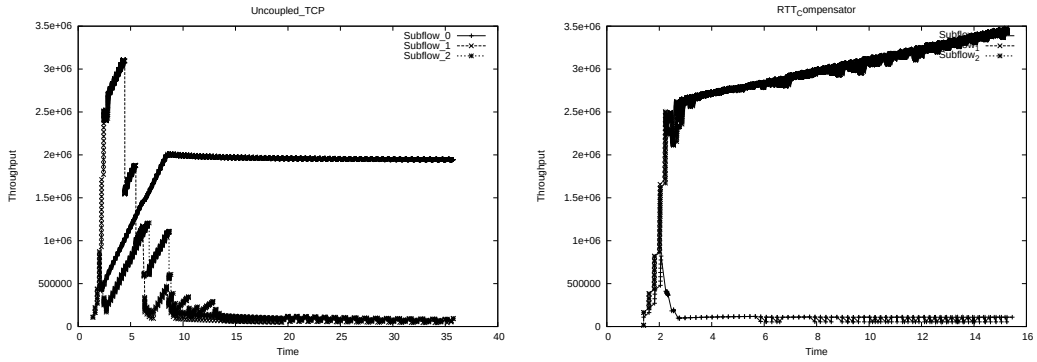
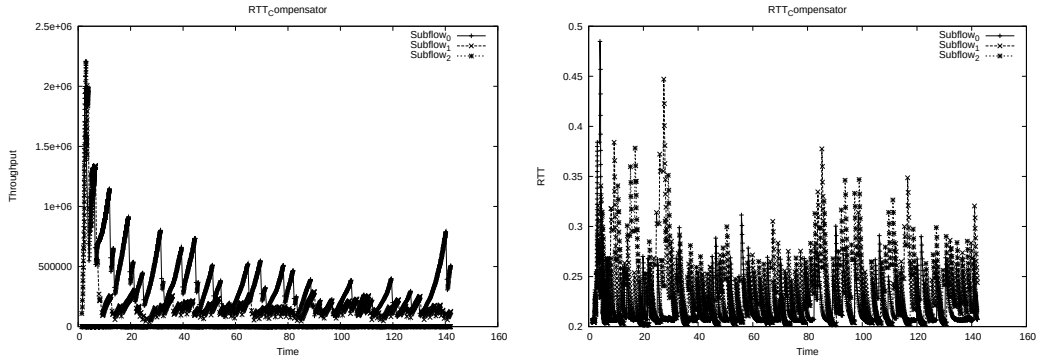


Figure 5.1. Topology 1



(a) (b)

Figure 5.2. Throughput of (a) Uncoupled_TCP with No_Packet_Reordering and (b) RTT_Compensator with DSACK



(a) (b)

Figure 5.3. (a) Throughput and (b) RTT of RTT_Compensator with FRTO

5.2 Topology 2

In this topology there are two intermediate nodes between clent and server. There are two subflows namely Subflow 0 and Subflow 1. In this all the links are 5mbps and have path delay of 100ms. All the links are in differen subnets. ECMP is introduced in this to route packets. The worst result is obtained by UncoupledTCP congestion control and no packet reordering and best result is obtained by RTT_Compensator congestion control and FRTO packet reordering.

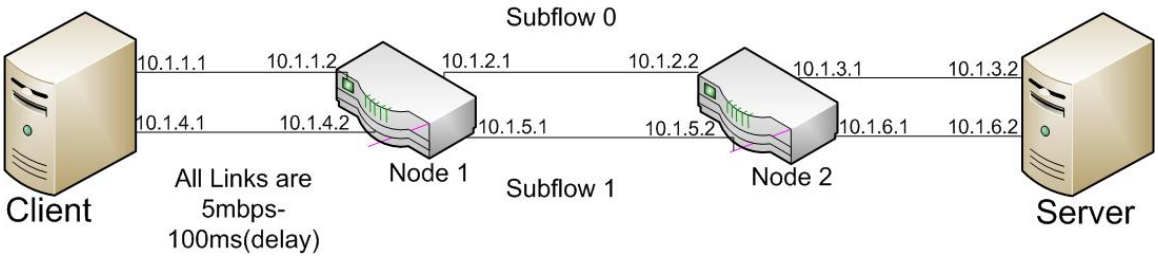
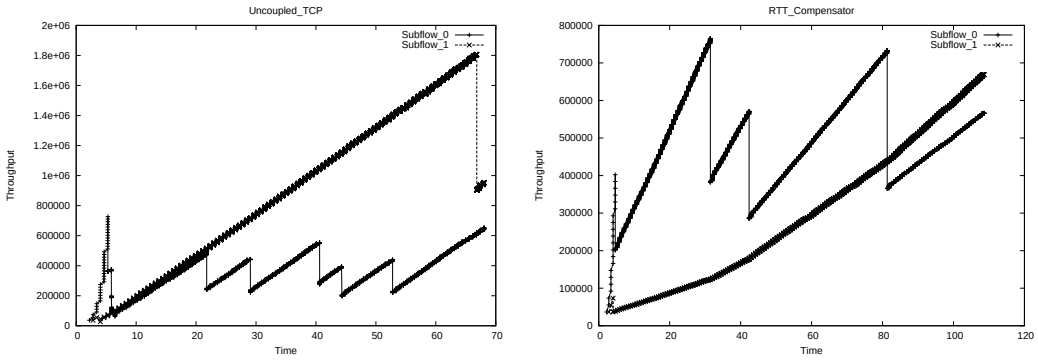
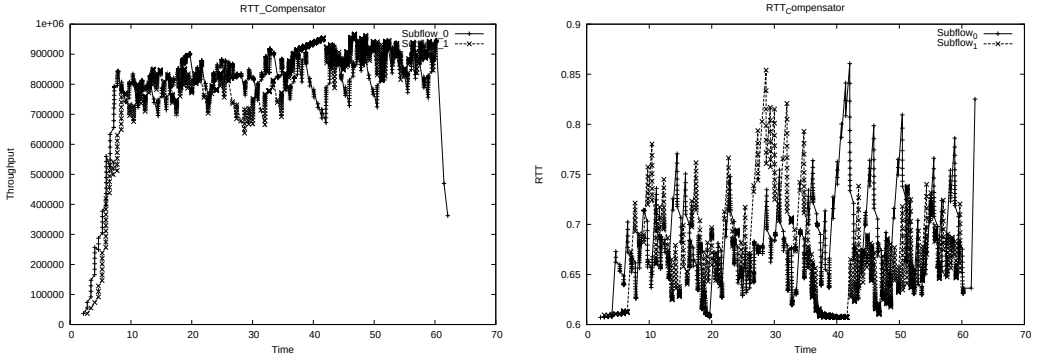


Figure 5.4. Topology 2



(a) (b)

Figure 5.5. Throughput of (a) Uncoupled_TCP with No_Packet_Reordering and (b) RTT_Compensator with DSACK



(a) (b)

Figure 5.6. (a) Throughput and (b) RTT of RTT_Compensator with FRTO

5.3 Topology 3

In this topology there are a pair of client and servers which have one subflow going with two nodes and a link in common. While the other subflow is direct. ECMP is used in this so the total subflow delay is assumed to be same. Hence, all links have data rate of 5 mbps with direct links having path delay of 120 ms and other links with 40 ms path delay. The best is obtained with RTT_Compensator congestion control with DSACK packet reordering. While worst is with RTT_Compensator congestion control with FRTO.

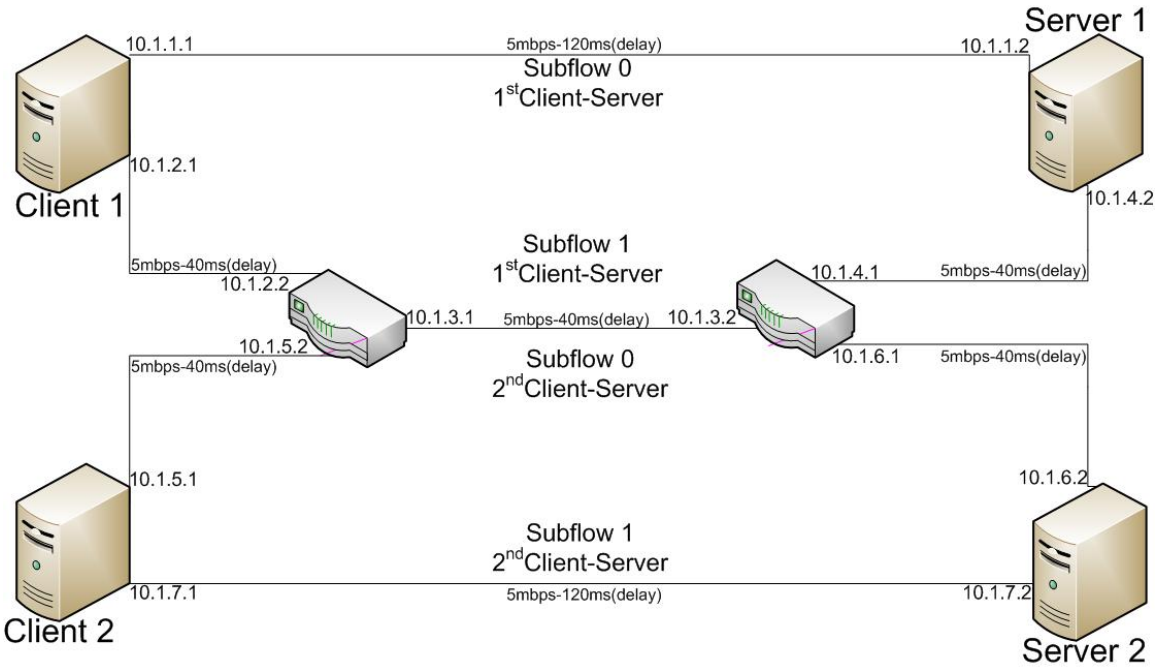
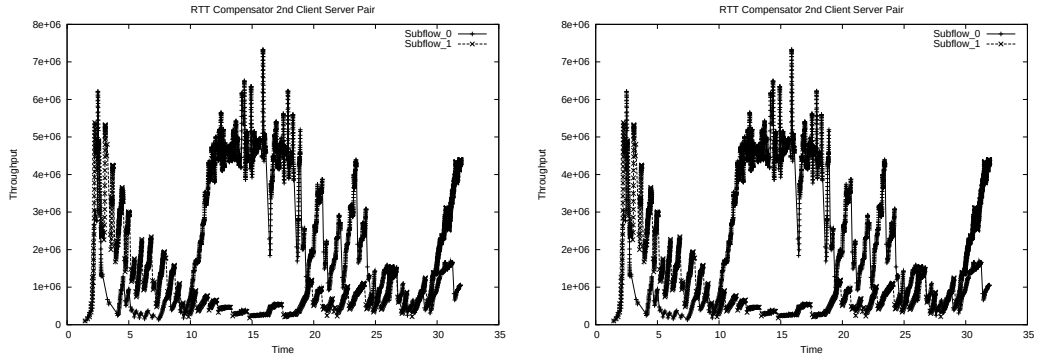
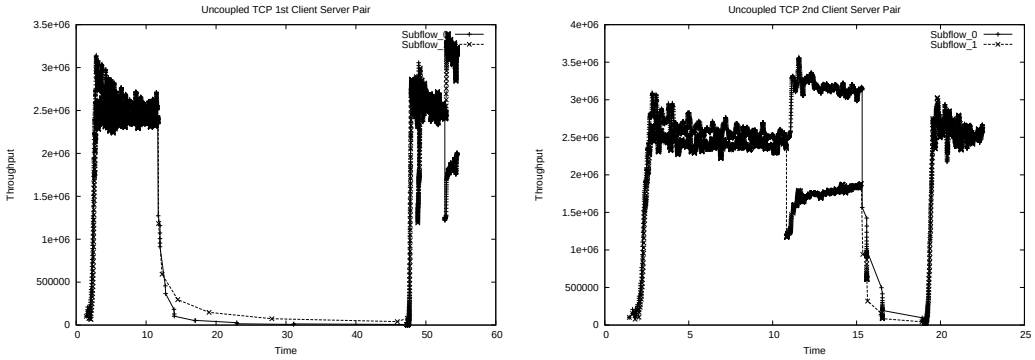


Figure 5.7. Topology 3



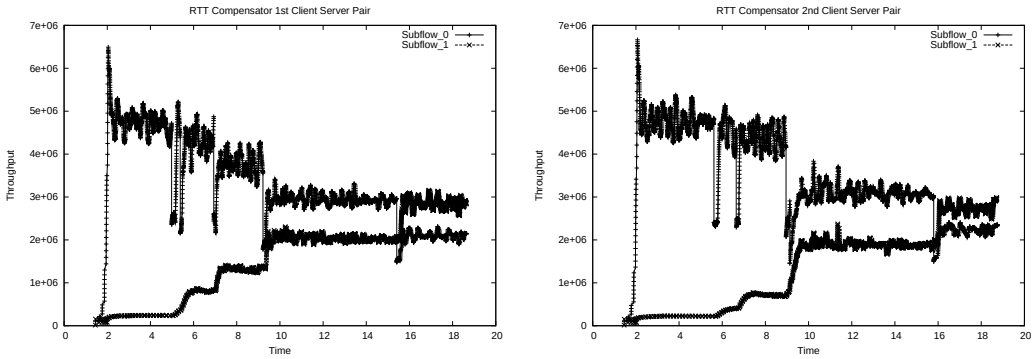
(a) (b)

Figure 5.8. Throughput of RTT_Compensator with FRTO for (a) 1st Client Server (b) 2nd Client Server



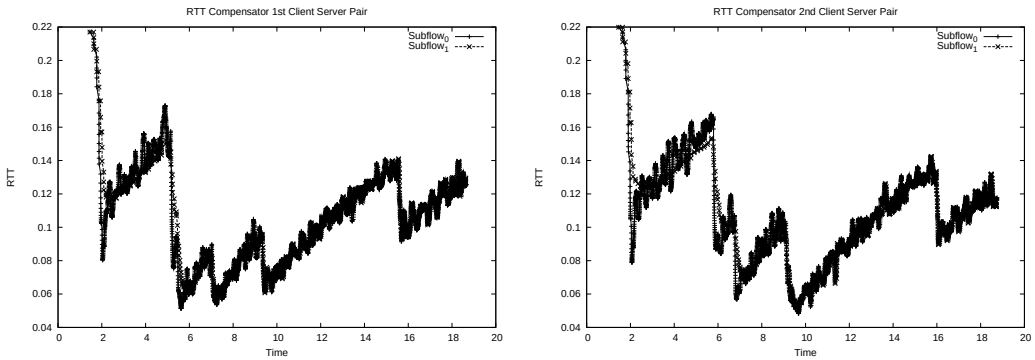
(a) (b)

Figure 5.9. Throughput of Uncoupled_TCP with No_Packet Reordering for (a) 1st Client Server (b) 2nd Client Server



(a) (b)

Figure 5.10. Throughput of RTT_Compensator with DSACK for (a) 1st Client Server (b) 2nd Client Server



(a) (b)

Figure 5.11. RTT for RTT_Compensator with DSACK for (a) 1st Client Server (b) 2nd Client Server

5.4 Topology 4

In this topology there are a pair of client and servers which have pair of subflows going with two nodes and a link in common. While the other subflow is direct. ECMP is used in this so the total subflow delay is assumed to be same. Hence, all links have data rate of 5 mbps with direct links having path delay of 120 ms and other links with 40 ms path delay. The best is obtained with RTT_Compensator congestion control with DSACK packet reordering. While worst is with RTT_Compensator congestion control with FRTO. This topology shows a bug with the mptcp code as we see there are three subflows but we get results which shows it is using two subflows.

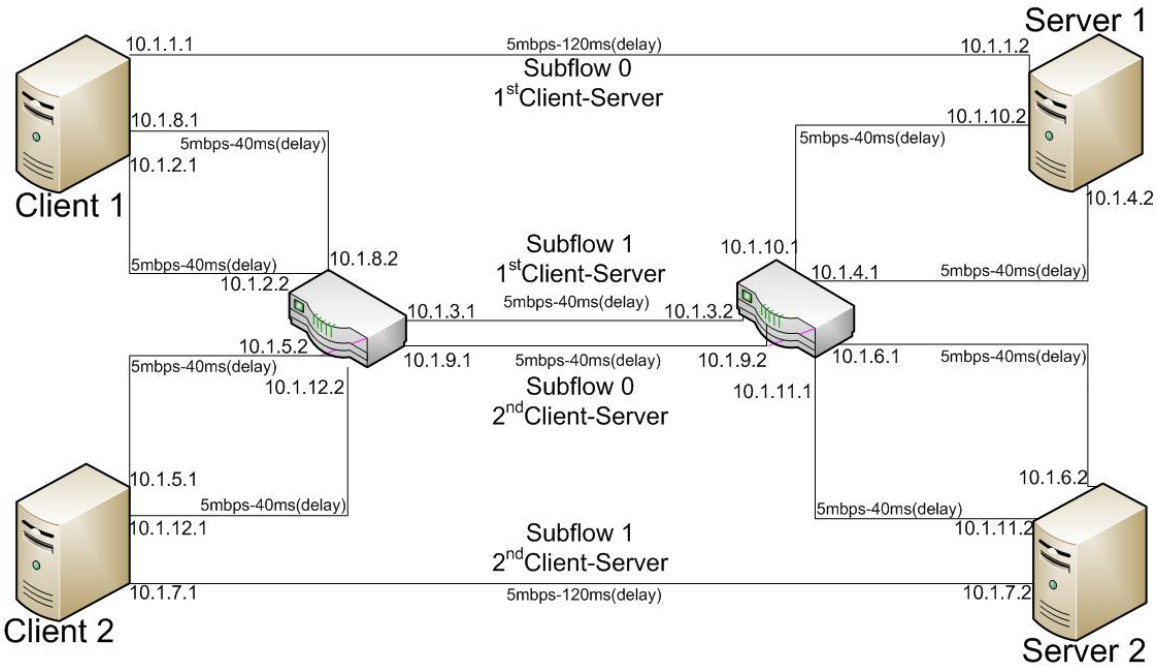
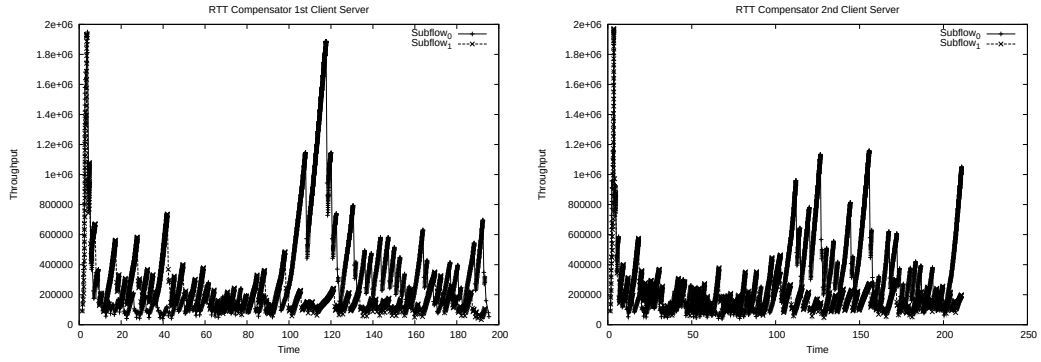
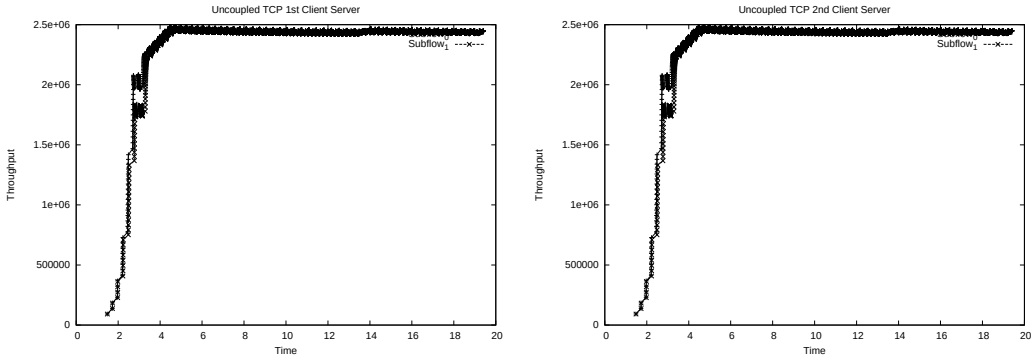


Figure 5.12. Topology 4



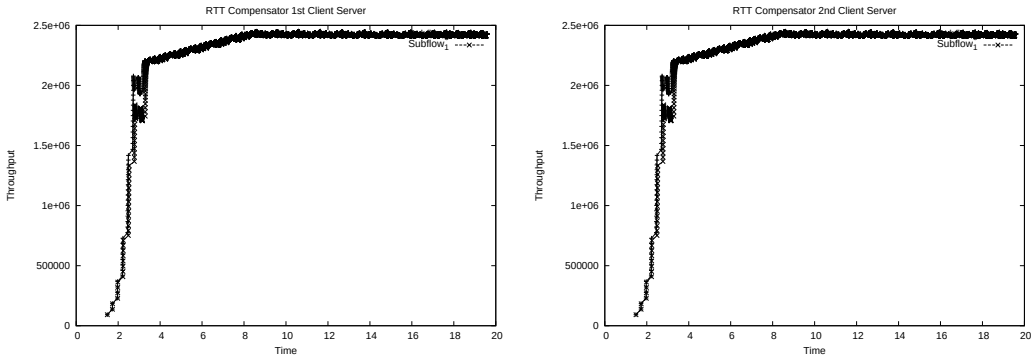
(a) (b)

Figure 5.13. Throughput of RTT_Compensator with FRTO for (a) 1st Client Server (b) 2nd Client Server



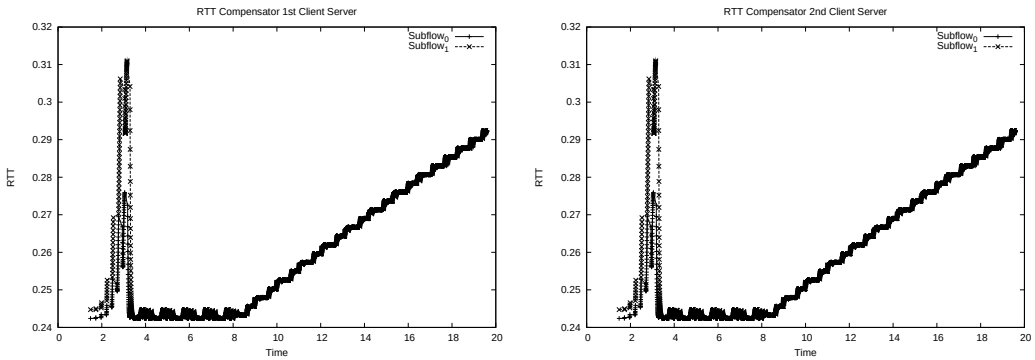
(a) (b)

Figure 5.14. Throughput of Uncoupled_TCP with No_Packet Reordering for (a) 1st Client Server (b) 2nd Client Server



(a) (b)

Figure 5.15. Throughput of RTT_Compensator with DSACK for (a) 1st Client Server (b) 2nd Client Server



(a) (b)

Figure 5.16. RTT for RTT_Compensator with DSACK for (a) 1st Client Server (b) 2nd Client Server

CHAPTER 6

FUTURE WORK AND CONCLUSION

In future work I propose a heuristic packet scheduling algorithm which can be implemented on the sender side to receive packets in order on the receiver side.

6.1 Heuristic Packet Scheduling Algorithm

- Send $(Current + |delay1 - delay2| + 1)$ packet number with maximum transmission unit (mtu) bytes on the slower link first.
- Send $(|delay1 - delay2| + 1)$ number of packets each with $(mtu)/(|delay1 - delay2| + 1)$ bytes on the faster link.
- Send Cumulative Acknowledge on the faster link.
- Buffer on the receiver side save packets received.
- In case of loss of packet send the packet on slower or faster link as the size of the packet and it is rearranged on the receiver side,

6.2 Conclusion

In this work, we described the architecture and implementation of the multipath transmission control protocol (MPTCP) in ns-3.6, with various congestion control algorithms and packet reordering algorithms. We also presented the throughput and round trip time for various topologies using different congestion control and packet reordering algorithms. Our evaluation focused on two key performance aspects: congestion control and packet reordering. The validation of our implementation is supported by the results of our simulation model. We found out that congestion control algorithm given by IETF namely, RTT Compensator and packer reordering technique F-RTO gives good results but not the best results. Therefor, we propose to find better algorithms for the same.

APPENDIX A

MPTCP CODE FOR NS-3.6

```
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include "ns3/core-module.h"
#include "ns3/simulator-module.h"
#include "ns3/node-module.h"
#include "ns3/helper-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mobility-module.h"
#include "ns3/mp-internet-stack-helper.h"
#include "ns3/mp-tcp-packet-sink.h"
#include "ns3/mp-tcp-l4-protocol.h"
#include "ns3/mp-tcp-socket-impl.h"
#include "ns3/point-to-point-channel.h"
#include "ns3/point-to-point-net-device.h"
using namespace ns3;

NS_LOG_COMPONENT_DEFINE("FirstMultipathToplogy");

static const uint32_t totalTxBytes = 10000000;
static const uint32_t sendBufSize  = 14000;
static const uint32_t recvBufSize  = 2000;
static uint32_t currentTxBytes     = 0;
static const double simDuration    = 360000000.0;

Ptr<Node> client;
Ptr<Node> server;

static const uint32_t writeSize = sendBufSize;
uint8_t data[totalTxBytes];
Ptr<MpTcpSocketImpl> lSocket    = 0;

void StartFlow (Ptr<MpTcpSocketImpl>, Ipv4Address, uint16_t);
void WriteUntilBufferFull (Ptr<Socket>, unsigned int);
void connectionSucceeded(Ptr<Socket>);
void connectionFailed(Ptr<Socket>);

void HandlePeerClose (Ptr<Socket>);
void HandlePeerError (Ptr<Socket>);
void CloseConnection (Ptr<Socket>);
```

```
int connect(Address &addr);
void variateDelay(PointToPointHelper P2Plink);

static void
CwndTracer (double oldval, double newval)
{
  NS_LOG_INFO ("Moving cwnd from " << oldval << " to " << newval);
}


int main(int argc, char *argv[])
{
    bool verbose;
    CongestionCtrl_t cc = Linked_Increases;
    PacketReorder_t  pr = F_RTO;
    int arg1 = -1, arg2 = -1, arg3 = -1, arg4 = -1;
    int sf = 3; // number of subflows
    CommandLine cmd;
    cmd.AddValue("verbose", "Tell application to log if true", verbose);
    cmd.AddValue("level", "Tell application which log level to use:\n \t - 0 = ↩
        ERROR \n \t - 1 = WARN \n \t - 2 = DEBUG \n \t - 3 = INFO \n \t - 4 = ↩
        FUNCTION \n \t - 5 = LOGIC \n \t - 6 = ALL", arg3);
    cmd.AddValue("cc", "Tell application which congestion control algorithm to use↩
        :\n \t - 0 = Uncoupled_TCPs \n \t - 1 = Linked_Increases \n \t - 2 = ↩
        RTT_Compensator \n \t - 3 = Fully_Coupled", arg1);
    cmd.AddValue("pr", "Tell application which packet reordering algorithm to use:\↩
        n \t - 0 = NoPR_Algo \n \t - 1 = Eifel \n \t - 2 = TCP_DOOR \n \t - 3 = ↩
        D_SACK \n \t - 4 = F_RTO",  arg2);
    cmd.AddValue("sf", "Tell application the number of subflows to be established ↩
        between endpoints",  arg4);

    cmd.Parse (argc, argv);

    cc = (arg1==-1 ? Linked_Increases:(CongestionCtrl_t) arg1);
    pr = (arg2==-1 ? F_RTO:(PacketReorder_t) arg2);
    sf = (arg4 = -1 ? 3: arg4);

    LogComponentEnable("FirstMultipathToplogy", LOG_LEVEL_ALL);
    if(arg3 == 2)
        LogComponentEnable("MpTcpSocketImpl", LOG_DEBUG);
    else if(arg3 == 6)
```

```
    LogComponentEnable("MpTcpSocketImpl", LOG_LEVEL_ALL);
else
    LogComponentEnable("MpTcpSocketImpl", LOG_WARN);
LogComponentEnable("MpTcpPacketSink", LOG_WARN);
LogComponentEnable("MpTcpHeader", LOG_WARN);
NodeContainer nodes;
nodes.Create(2);
client = nodes.Get(0);
server = nodes.Get(1);

MpInternetStackHelper stack;
stack.Install(nodes);
vector<Ipv4InterfaceContainer> ipv4Ints;
int i,j,k;

for(i=0; i < sf-2; i++)
{
    PointToPointHelper p2plink;
    p2plink.SetDeviceAttribute ("DataRate", StringValue("2Mbps"));
    p2plink.SetChannelAttribute("Delay", StringValue("100ms"));

    NetDeviceContainer netDevices;
    netDevices = p2plink.Install(nodes);

    std::stringstream netAddr;
    netAddr << "10.1." << (i+1) << ".0";
    string str = netAddr.str();

    Ipv4AddressHelper ipv4addr;
    ipv4addr.SetBase(str.c_str(), "255.255.255.0");
    Ipv4InterfaceContainer interface = ipv4addr.Assign(netDevices);
    ipv4Ints.insert(ipv4Ints.end(), interface);
}

for(j=i; j < sf-1; j++)
{
    PointToPointHelper p2plink;
    p2plink.SetDeviceAttribute ("DataRate", StringValue("5Mbps"));
    p2plink.SetChannelAttribute("Delay", StringValue("100ms"));

    NetDeviceContainer netDevices;
    netDevices = p2plink.Install(nodes);
```

```
        std::stringstream netAddr;
        netAddr << "10.1." << (j+1) << ".0";
        string str = netAddr.str();

        Ipv4AddressHelper ipv4addr;
        ipv4addr.SetBase(str.c_str(), "255.255.255.0");
        Ipv4InterfaceContainer interface = ipv4addr.Assign(netDevices);
        ipv4Ints.insert(ipv4Ints.end(), interface);
    }
    for(k=j; k < sf; k++)
    {
        PointToPointHelper p2plink;
        p2plink.SetDeviceAttribute ("DataRate", StringValue("7Mbps"));
        p2plink.SetChannelAttribute("Delay", StringValue("100ms"));

        NetDeviceContainer netDevices;
        netDevices = p2plink.Install(nodes);

        std::stringstream netAddr;
        netAddr << "10.1." << (k+1) << ".0";
        string str = netAddr.str();

        Ipv4AddressHelper ipv4addr;
        ipv4addr.SetBase(str.c_str(), "255.255.255.0");
        Ipv4InterfaceContainer interface = ipv4addr.Assign(netDevices);
        ipv4Ints.insert(ipv4Ints.end(), interface);
    }
    PointToPointHelper::EnablePcapAll("mptcp");

    uint32_t servPort = 5000;
    NS_LOG_INFO ("address " << ipv4Ints[0].GetAddress (1));
    ObjectFactory m_sf;
    m_sf.SetTypeId("ns3::MpTcpPacketSink");
    m_sf.Set("Protocol", StringValue ("ns3::TcpSocketFactory"));
    m_sf.Set("Local", AddressValue(InetSocketAddress (ipv4Ints[0].GetAddress (1), ↩
        servPort)));
    m_sf.Set("algopr", UintegerValue ((uint32_t) pr));
    Ptr<Application> sapp = m_sf.Create<Application> ();
    server->AddApplication(sapp);

    ApplicationContainer Apps;
```

```
    Apps.Add(sapp);
    Apps.Start(Seconds(0.0));
    Apps.Stop(Seconds(simDuration));


    lSocket = new MpTcpSocketImpl (client);
    lSocket->SetCongestionCtrlAlgo (cc);
    lSocket->SetDataDistribAlgo (Round_Robin);
    lSocket->SetPacketReorderAlgo (pr);
    lSocket->Bind ();


    Config::ConnectWithoutContext ("/NodeList/0/$ns3::MpTcpSocketImpl/subflows/0/↩
        CongestionWindow", MakeCallback (&CwndTracer));

    Simulator::ScheduleNow (&StartFlow, lSocket, ipv4Ints[0].GetAddress (1), ↩
        servPort);
    Simulator::Stop (Seconds(simDuration + 1000.0));
    Simulator::Run ();
    Simulator::Destroy();
    NS_LOG_LOGIC("mpTopology:: simulation ended");
    return 0;
}

void StartFlow(Ptr<MpTcpSocketImpl> localSocket, Ipv4Address servAddress, uint16_t ↩
    servPort)
{
  NS_LOG_LOGIC("Starting flow at time " <<  Simulator::Now().GetSeconds());
  lSocket->SetMaxSubFlowNumber(5);
  lSocket->SetMinSubFlowNumber(3);
  lSocket->SetSourceAddress(Ipv4Address("10.1.1.1"));
  lSocket->allocateSendingBuffer(sendBufSize);
  lSocket->allocateRecvingBuffer(recvBufSize);
  lSocket->SetunOrdBufMaxSize(50);

  int connectionState = lSocket->Connect( servAddress, servPort);
  if(connectionState == 0)
  {
      lSocket->SetConnectCallback  (MakeCallback (&connectionSucceeded), ↩
          MakeCallback (&connectionFailed));
      lSocket->SetDataSentCallback (MakeCallback (&WriteUntilBufferFull));
      lSocket->SetCloseCallbacks   (MakeCallback (&HandlePeerClose), MakeCallback(&↩
```

```
            HandlePeerError));
        lSocket->GetSubflow(0)->StartTracing ("CongestionWindow");
    }else
    {
        NS_LOG_LOGIC("mpTopology:: connection failed");
    }
}

void connectionSucceeded (Ptr<Socket> localSocket)
{
    NS_LOG_INFO("mpTopology: Connection requeste succeed");
    Simulator::Schedule (Seconds (1.0), &WriteUntilBufferFull, lSocket, 0);
    Simulator::Schedule (Seconds (simDuration), &CloseConnection, lSocket);
}

void connectionFailed (Ptr<Socket> localSocket)
{
    NS_LOG_INFO("mpTopology: Connection requeste failure");
    lSocket->Close();
}

void HandlePeerClose (Ptr<Socket> localSocket)
{
    NS_LOG_INFO("mpTopology: Connection closed by peer");
    lSocket->Close();
}

void HandlePeerError (Ptr<Socket> localSocket)
{
    NS_LOG_INFO("mpTopology: Connection closed by peer error");
    lSocket->Close();
}

void CloseConnection (Ptr<Socket> localSocket)
{
    lSocket->Close();
    NS_LOG_LOGIC("mpTopology:: currentTxBytes = " << currentTxBytes);
    NS_LOG_LOGIC("mpTopology:: totalTxBytes   = " << totalTxBytes);
    NS_LOG_LOGIC("mpTopology:: connection to remote host has been closed");
}
```

```
void variateDelay (Ptr<Node> node)
{

    Ptr<Ipv4L3Protocol> ipv4 = node->GetObject<Ipv4L3Protocol> ();
    TimeValue delay;
    for(uint32_t i = 0; i < ipv4->GetNInterfaces(); i++)
    {
        Ptr<Ipv4Interface> interface = ipv4->GetInterface(i);
        Ipv4InterfaceAddress interfaceAddr = interface->GetAddress (0);
        if(interfaceAddr.GetLocal() == Ipv4Address::GetLoopback())
        {
            continue;
        }

        Ptr<NetDevice> netDev =  interface->GetDevice();
        Ptr<Channel> P2Plink  =  netDev->GetChannel();
        P2Plink->GetAttribute(string("Delay"), delay);
        double oldDelay = delay.Get().GetSeconds();
        std::stringstream strDelay;
        double newDelay = (rand() \% 100) * 0.001;
        double err = newDelay - oldDelay;
        strDelay << (0.95 * oldDelay + 0.05 * err) << "s";
        P2Plink->SetAttribute(string("Delay"), StringValue(strDelay.str()));
        P2Plink->GetAttribute(string("Delay"), delay);
        }
}

void WriteUntilBufferFull (Ptr<Socket> localSocket, unsigned int txSpace)
{
  while (currentTxBytes < totalTxBytes && lSocket->GetTxAvailable () > 0)
  {
      uint32_t left    = totalTxBytes - currentTxBytes;
      uint32_t toWrite = std::min(writeSize, lSocket->GetTxAvailable ());
      toWrite = std::min( toWrite , left );
      int amountBuffered = lSocket->FillBuffer (&data[currentTxBytes], toWrite);
      currentTxBytes += amountBuffered;

      //variateDelay(client);
      lSocket->SendBufferedData();
  }

}
```

APPENDIX B

ECMP

Equal-cost multi-path routing(ECMP) is ato a single destination can occur over multiple best paths which tie for top place in routing metric calculations. Multipath routing can be used in conjunction with most routing protocols, since it is a per-hop decision that is limited to a single router. It potentially offers substantial increases in bandwidth by load-balancing traffic over multiple paths. Change in file ipv4-global-routing.cc:

```
// -*- Mode: C++; c-file-style: "gnu"; indent-tabs-mode:nil; -*-
//
// Copyright (c) 2008 University of Washington
//
// This program is free software; you can redistribute it and/or modify
// it under the terms of the GNU General Public License version 2 as
// published by the Free Software Foundation;
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
//

#include "ns3/log.h"
#include "ns3/object.h"
#include "ns3/packet.h"
#include "ns3/net-device.h"
#include "ns3/ipv4-route.h"
#include "ns3/ipv4-routing-table-entry.h"
#include "ns3/boolean.h"
#include "ipv4-global-routing.h"
#include <vector>

NS_LOG_COMPONENT_DEFINE ("Ipv4GlobalRouting");

namespace ns3 {
```

```
NS_OBJECT_ENSURE_REGISTERED (Ipv4GlobalRouting);

TypeId
Ipv4GlobalRouting::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::Ipv4GlobalRouting")
    .SetParent<Object> ()
    .AddAttribute ("RandomEcmpRouting",
                   "Set to true if packets are randomly routed among ECMP; set to ↩
                       false for using only one route consistently",
                   BooleanValue(true),
                   MakeBooleanAccessor (&Ipv4GlobalRouting::m_randomEcmpRouting),
                   MakeBooleanChecker ());
  return tid;
}

Ipv4GlobalRouting::Ipv4GlobalRouting ()
: m_randomEcmpRouting (true)
{
  NS_LOG_FUNCTION_NOARGS ();
}

Ipv4GlobalRouting::~Ipv4GlobalRouting ()
{
  NS_LOG_FUNCTION_NOARGS ();
}

void
Ipv4GlobalRouting::AddHostRouteTo (Ipv4Address dest,
                                   Ipv4Address nextHop,
                                   uint32_t interface)
{
  NS_LOG_FUNCTION (dest << nextHop << interface);
  Ipv4RoutingTableEntry *route = new Ipv4RoutingTableEntry ();
  *route = Ipv4RoutingTableEntry::CreateHostRouteTo (dest, nextHop, interface);
  m_hostRoutes.push_back (route);
}

void
Ipv4GlobalRouting::AddHostRouteTo (Ipv4Address dest,
                                   uint32_t interface)
```

```
{
  NS_LOG_FUNCTION (dest << interface);
  Ipv4RoutingTableEntry *route = new Ipv4RoutingTableEntry ();
  *route = Ipv4RoutingTableEntry::CreateHostRouteTo (dest, interface);
  m_hostRoutes.push_back (route);
}

void
Ipv4GlobalRouting::AddNetworkRouteTo (Ipv4Address network,
                                      Ipv4Mask networkMask,
                                      Ipv4Address nextHop,
                                      uint32_t interface)
{
  NS_LOG_FUNCTION (network << networkMask << nextHop << interface);
  Ipv4RoutingTableEntry *route = new Ipv4RoutingTableEntry ();
  *route = Ipv4RoutingTableEntry::CreateNetworkRouteTo (network,
                                            networkMask,
                                            nextHop,
                                            interface);
  m_networkRoutes.push_back (route);
}

void
Ipv4GlobalRouting::AddNetworkRouteTo (Ipv4Address network,
                                      Ipv4Mask networkMask,
                                      uint32_t interface)
{
  NS_LOG_FUNCTION (network << networkMask << interface);
  Ipv4RoutingTableEntry *route = new Ipv4RoutingTableEntry ();
  *route = Ipv4RoutingTableEntry::CreateNetworkRouteTo (network,
                                            networkMask,
                                            interface);
  m_networkRoutes.push_back (route);
}

void
Ipv4GlobalRouting::AddASExternalRouteTo (Ipv4Address network,
                                         Ipv4Mask networkMask,
                                         Ipv4Address nextHop,
                                         uint32_t interface)
{
  NS_LOG_FUNCTION (network << networkMask << nextHop);
```

```
  Ipv4RoutingTableEntry *route = new Ipv4RoutingTableEntry ();
  *route = Ipv4RoutingTableEntry::CreateNetworkRouteTo (network,
      networkMask,
      nextHop,
      interface);
  m_ASexternalRoutes.push_back (route);
}


Ptr<Ipv4Route>
Ipv4GlobalRouting::LookupGlobal (Ipv4Address dest)
{
  NS_LOG_FUNCTION_NOARGS ();
  NS_LOG_LOGIC ("Looking for route for destination " << dest);
  Ptr<Ipv4Route> rtentry = 0;
  Ipv4RoutingTableEntry* route = 0;
  // store all available routes that bring packets to their destination
  typedef std::vector<Ipv4RoutingTableEntry*> RouteVec_t;
  RouteVec_t allRoutes;

  NS_LOG_LOGIC ("Number of m_hostRoutes = " << m_hostRoutes.size ());
  for (HostRoutesCI i = m_hostRoutes.begin ();
       i != m_hostRoutes.end ();
       i++)
    {
      NS_ASSERT ((*i)->IsHost ());
      if ((*i)->GetDest ().IsEqual (dest))
        {
          allRoutes.push_back (*i);
          NS_LOG_LOGIC (allRoutes.size () << "Found global host route" << *i);
        }
    }
  if (allRoutes.size () == 0) // if no host route is found
    {
      NS_LOG_LOGIC ("Number of m_networkRoutes" << m_networkRoutes.size ());
      for (NetworkRoutesI j = m_networkRoutes.begin ();
           j != m_networkRoutes.end ();
           j++)
        {
          NS_ASSERT ((*j)->IsNetwork () || (*j)->IsDefault ());
          Ipv4Mask mask = (*j)->GetDestNetworkMask ();
          Ipv4Address entry = (*j)->GetDestNetwork ();
```

```
        if (mask.IsMatch (dest, entry))
          {
            allRoutes.push_back (*j);
            NS_LOG_LOGIC (allRoutes.size () << "Found global network route" << *j↩
                );
          }
      }
  }
if (allRoutes.size () == 0)  // consider external if no host/network found
  {
    for (ASExternalRoutesI k = m_ASexternalRoutes.begin ();
         k != m_ASexternalRoutes.end ();
         k++)
      {
        Ipv4Mask mask = (*k)->GetDestNetworkMask ();
        Ipv4Address entry = (*k)->GetDestNetwork ();
        if (mask.IsMatch (dest, entry))
          {
            NS_LOG_LOGIC ("Found external route" << *k);
            route = (*k);
            allRoutes.push_back (*k);
            break;
          }
      }
  }
if (allRoutes.size () > 0 ) // if route(s) is found
  {
    // pick up one of the routes uniformly at random if random
    // ECMP routing is enabled, or always select the first route
    // consistently if random ECMP routing is disabled
    uint32_t selectIndex;
    if (m_randomEcmpRouting)
      {
        selectIndex = m_rand.GetInteger (0, allRoutes.size ()-1);
      }
    else
      {
        selectIndex = 0;
      }
    route = allRoutes.at (selectIndex);
    // create a Ipv4Route object from the selected routing table entry
    rtentry = Create<Ipv4Route> ();
```

```
      rtentry->SetDestination (route->GetDest ());
      // XXX handle multi-address case
      rtentry->SetSource (m_ipv4->GetAddress (route->GetInterface(), 0).GetLocal ()↩
          );
      rtentry->SetGateway (route->GetGateway ());
      uint32_t interfaceIdx = route->GetInterface ();
      rtentry->SetOutputDevice (m_ipv4->GetNetDevice (interfaceIdx));
      return rtentry;
    }
  else
    {
      return 0;
    }
}

uint32_t
Ipv4GlobalRouting::GetNRoutes (void)
{
  NS_LOG_FUNCTION_NOARGS ();
  uint32_t n = 0;
  n += m_hostRoutes.size ();
  n += m_networkRoutes.size ();
  n += m_ASexternalRoutes.size ();
  return n;
}

Ipv4RoutingTableEntry *
Ipv4GlobalRouting::GetRoute (uint32_t index)
{
  NS_LOG_FUNCTION (index);
  if (index < m_hostRoutes.size ())
    {
      uint32_t tmp = 0;
      for (HostRoutesCI i = m_hostRoutes.begin ();
           i != m_hostRoutes.end ();
           i++)
        {
          if (tmp  == index)
            {
              return *i;
            }
          tmp++;
```

```
        }
    }
  index -= m_hostRoutes.size ();
  uint32_t tmp = 0;
  if (index < m_networkRoutes.size())
    {
      for (NetworkRoutesI j = m_networkRoutes.begin ();
           j != m_networkRoutes.end ();
           j++)
        {
          if (tmp == index)
            {
              return *j;
            }
          tmp++;
        }
    }
  index -= m_networkRoutes.size();
  tmp = 0;
  for (ASExternalRoutesI k = m_ASexternalRoutes.begin ();
       k != m_ASexternalRoutes.end ();
       k++)
  {
    if (tmp == index)
    {
      return *k;
    }
    tmp++;
  }
  NS_ASSERT (false);
  // quiet compiler.
  return 0;
}
void
Ipv4GlobalRouting::RemoveRoute (uint32_t index)
{
  NS_LOG_FUNCTION (index);
  if (index < m_hostRoutes.size ())
    {
      uint32_t tmp = 0;
      for (HostRoutesI i = m_hostRoutes.begin ();
           i != m_hostRoutes.end ();
```

```
        i++)
      {
        if (tmp  == index)
          {
            NS_LOG_LOGIC ("Removing route " << index << "; size = " << ↩
                m_hostRoutes.size());
            delete *i;
            m_hostRoutes.erase (i);
            NS_LOG_LOGIC ("Done removing host route " << index << "; host route ↩
                remaining size = " << m_hostRoutes.size());
            return;
          }
        tmp++;
      }
  }
index -= m_hostRoutes.size ();
uint32_t tmp = 0;
for (NetworkRoutesI j = m_networkRoutes.begin ();
     j != m_networkRoutes.end ();
     j++)
  {
    if (tmp == index)
      {
        NS_LOG_LOGIC ("Removing route " << index << "; size = " << ↩
            m_networkRoutes.size());
        delete *j;
        m_networkRoutes.erase (j);
        NS_LOG_LOGIC ("Done removing network route " << index << "; network route↩
             remaining size = " << m_networkRoutes.size());
        return;
      }
    tmp++;
  }
index -= m_networkRoutes.size ();
tmp = 0;
for (ASExternalRoutesI k = m_ASexternalRoutes.begin ();
       k != m_ASexternalRoutes.end ();
       k++)
  {
    if (tmp == index)
    {
      NS_LOG_LOGIC ("Removing route " << index << "; size = " << ↩
```

```
            m_ASexternalRoutes.size());
          delete *k;
          m_ASexternalRoutes.erase (k);
          NS_LOG_LOGIC ("Done removing network route " << index << "; network route ↩
              remaining size = " << m_networkRoutes.size());
          return;
        }
        tmp++;
      }
    NS_ASSERT (false);
}

void
Ipv4GlobalRouting::DoDispose (void)
{
  NS_LOG_FUNCTION_NOARGS ();
  for (HostRoutesI i = m_hostRoutes.begin ();
       i != m_hostRoutes.end ();
       i = m_hostRoutes.erase (i))
    {
      delete (*i);
    }
  for (NetworkRoutesI j = m_networkRoutes.begin ();
       j != m_networkRoutes.end ();
       j = m_networkRoutes.erase (j))
    {
      delete (*j);
    }
  for (ASExternalRoutesI l = m_ASexternalRoutes.begin ();
         l != m_ASexternalRoutes.end ();
         l = m_ASexternalRoutes.erase (l))
    {
      delete (*l);
    }

  Ipv4RoutingProtocol::DoDispose ();
}

Ptr<Ipv4Route>
Ipv4GlobalRouting::RouteOutput (Ptr<Packet> p, const Ipv4Header &header, uint32_t ↩
    oif, Socket::SocketErrno &sockerr)
{
```

```
//
// First, see if this is a multicast packet we have a route for.  If we
// have a route, then send the packet down each of the specified interfaces.
//
  if (header.GetDestination().IsMulticast ())
    {
      NS_LOG_LOGIC ("Multicast destination-- returning false");
      return 0; // Let other routing protocols try to handle this
    }
//
// See if this is a unicast packet we have a route for.
//
  NS_LOG_LOGIC ("Unicast destination- looking up");
  Ptr<Ipv4Route> rtentry = LookupGlobal (header.GetDestination());
  if (rtentry)
    {
      sockerr = Socket::ERROR_NOTERROR;
    }
  else
    {
      sockerr = Socket::ERROR_NOROUTETOHOST;
    }
  return rtentry;
}

bool
Ipv4GlobalRouting::RouteInput  (Ptr<const Packet> p, const Ipv4Header &header, Ptr<↩
    const NetDevice> idev,                            UnicastForwardCallback ucb, ↩
    MulticastForwardCallback mcb,
                            LocalDeliverCallback lcb, ErrorCallback ecb)
{

  NS_LOG_FUNCTION (this << p << header << header.GetSource () << header.↩
      GetDestination () << idev);
  // Check if input device supports IP
  NS_ASSERT (m_ipv4->GetInterfaceForDevice (idev) >= 0);
  uint32_t iif = m_ipv4->GetInterfaceForDevice (idev);

  if (header.GetDestination ().IsMulticast ())
    {
      NS_LOG_LOGIC ("Multicast destination-- returning false");
```

```
    return false; // Let other routing protocols try to handle this
  }

if (header.GetDestination ().IsBroadcast ())
  {
    NS_LOG_LOGIC ("For me (Ipv4Addr broadcast address)");
    // TODO:  Local Deliver for broadcast
    // TODO:  Forward broadcast
  }

// TODO:  Configurable option to enable RFC 1222 Strong End System Model
// Right now, we will be permissive and allow a source to send us
// a packet to one of our other interface addresses; that is, the
// destination unicast address does not match one of the iif addresses,
// but we check our other interfaces.  This could be an option
// (to remove the outer loop immediately below and just check iif).
for (uint32_t j = 0; j < m_ipv4->GetNInterfaces (); j++)
  {
    for (uint32_t i = 0; i < m_ipv4->GetNAddresses (j); i++)
      {
        Ipv4InterfaceAddress iaddr = m_ipv4->GetAddress (j, i);
        Ipv4Address addr = iaddr.GetLocal ();
        if (addr.IsEqual (header.GetDestination ()))
          {
            if (j == iif)
              {
                NS_LOG_LOGIC ("For me (destination " << addr << " match)");
              }
            else
              {
                NS_LOG_LOGIC ("For me (destination " << addr << " match) on ↩
                    another interface " << header.GetDestination ());
              }
            lcb (p, header, iif);
            return true;
          }
        if (header.GetDestination ().IsEqual (iaddr.GetBroadcast ()))
          {
            NS_LOG_LOGIC ("For me (interface broadcast address)");
            lcb (p, header, iif);
            return true;
          }
```

```
          NS_LOG_LOGIC ("Address "<< addr << " not a match");
        }
    }
  // Check if input device supports IP forwarding
  if (m_ipv4->IsForwarding (iif) == false)
    {
      NS_LOG_LOGIC ("Forwarding disabled for this interface");
      ecb (p, header, Socket::ERROR_NOROUTETOHOST);
      return false;
    }
  // Next, try to find a route
  NS_LOG_LOGIC ("Unicast destination- looking up global route");
  Ptr<Ipv4Route> rtentry = LookupGlobal (header.GetDestination ());
  if (rtentry != 0)
    {
      NS_LOG_LOGIC ("Found unicast destination- calling unicast callback");
      ucb (rtentry, p, header);
      return true;
    }
  else
    {
      NS_LOG_LOGIC ("Did not find unicast destination- returning false");
      return false; // Let other routing protocols try to handle this
                    // route request.
    }
}
void
Ipv4GlobalRouting::NotifyInterfaceUp (uint32_t i)
{}
void
Ipv4GlobalRouting::NotifyInterfaceDown (uint32_t i)
{}
void
Ipv4GlobalRouting::NotifyAddAddress (uint32_t interface, Ipv4InterfaceAddress ↩
    address)
{}
void
Ipv4GlobalRouting::NotifyRemoveAddress (uint32_t interface, Ipv4InterfaceAddress ↩
    address)
{}
void
Ipv4GlobalRouting::SetIpv4 (Ptr<Ipv4> ipv4)
```

```
{
  NS_LOG_FUNCTION(this << ipv4);
  NS_ASSERT (m_ipv4 == 0 && ipv4 != 0);
  m_ipv4 = ipv4;
}



}
```

BIBLIOGRAPHY

[1] B. Chihani and D. Collange "A Multipath TCP model for ns-3 simulator", , *SIMUTools*, 2011

[2] T. Hacker and B. Athey In "The End-to-End Performance Effects of Parallel TCP Sockets on a Lossy Wide-Area Network," in *IEEE Parallel and Distributed Processing Symposium., Proceedings International (IPDPS)*, April 2002.

[3] Y. Hasegawa, I. Yamaguchi, T. Hama, H. Shimonishi and T. Murase "Improved Data Distribution for Multipath TCP communication," in *IEEE Global Telecommunications Conference (Globecom)*, December 2005.

[4] R. Ludwig and R. H. Katz "The Eifel algorithm: making TCP robust against spurious retransmissions," *ACM SIGCOMM Computer Communication Review*, Volume 30 Issue 1, pages 30-36, January 2000.

[5] "MPTCP code," http://code.google.com/p/mptcp-ns3. May 2011.

[6] "The ns-3 network simulator," http://www.nsnam.org. November 2010.

[7] P. Sarolathi, M. Kojjo and K. Raatikainen "F-RTO: An Enhanced Recovery Algorithm for TCP Retransmission Timeouts," *Computer Communication Review*, Volume 33 Part 2, pages 51-64, 2003.

[8] D. Wischik, M. Handley and M. Bagnulo Braun "The Resource Pooling Principle," *ACM SIGCOMM Computer Communication Review*, Volume 38.5, pages 47-52, October 2008.

[9] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley "Design, implementation and evaluation of congestion control for multipath TCP," in *Proceedings USENIX NSDI*, 2011.

[10] D. Wischik, C. Raiciu and M. Handley "Practical Congestion Control for Multipath Transport Protocols," *University College London, London/United Kingdom, Tech. Rep.*, 2009.

[11] M. Zhang, B. Karp, S. Floyd and L. Peterson "RR-TCP: A Reordering-Robust TCP with DSACK," in *Proceedings of the Eleventh IEEE International Conference on Networking Protocols (ICNP 2003), Atlanta, GA*, November, 2003.