LULEÅ UNIVERITY OF TECHNOLOGY

COMPUTER SCIENCE

D0021E

SIMULATING TRANSMISSION CONTROL PROTOCOL AND TCP RENO CONGESTION CONTROL

Author
S. JONSSON
setjon-7@student.ltu.se
M. LARSSON ANDERSSON
magany-7@student.ltu.se
M. JONSSON
micjon-5@student.ltu.se

Supervisor
A. Prof. KARAN MITRA
SAGUNA SAGUNA

April 15, 2020

Contents

List of Figures

1 Introduction

In laboration 5 for the course we got to choose, in discussion with the supervisor, a network protocol to implement. We decided to implement TCP including a simple congestion control mechanism, in our case TCP Reno.

We have implemented:

- TCP Reno
- RTO calculation
- Three-way handshake
- Four-way handshake
- Retransmission of dropped packets

2 Methods

Our implementation of TCP has taken a lot of inspiration from RFC 793[1], for example for RTO calculations or opening and closing handshakes.

We made a simplified version of TCP where the communication had one receiver and one sender. The simulator was provided with three-way handshake for opening a connection and four-way handshake for closing. We have also implemented TCP Reno congestion control for handling timeouts and 3 duplicate ACK with slow start and fast retransmit.

The simulation start with receiver sending a SYN to the chosen corespondent when corespondent gets the SYN it will create a new TCP connection and return a SYNACK. When the receiver gets the SYNACK it will change the status of the connection to open and return an ACK to the sender then it will construct an new message with and will supply it a data value, an int value representing the number of packet sender should send to receiver, the data it want to retrieve from sender.

When the sender gets the ACK for the SYNACK it will set the connection to open. The sender gets the message requesting the data it will construct packets equal that of value of supplied data in the message and start transmitting the packet to receiver. When the receiver gets the packets it will check if the sequence of the message is the next wanted sequence. If it is, it will add it the a list of which messages has been acknowledged and then return an ACK with the next wanted sequence number. Should it not be the next wanted sequence it will still add the list of acknowledged

sequences but will send an ACK with the same next wanted sequence number as before but attached the acknowledged sequence as data so the sender knows that does not need to transmit again the sequence.

When the sender receive an acknowledge of a sequence during this time it will check if it matches the previous received ACK if it matches it three times in a row it will drop the congestion size in half. When it receives a new acknowledgement it will remove all messages with a lower or equal sequence to the acknowledged sequence, from a list of messages sent that are waiting on acknowledgement, as it could only have sent an the acknowledged of a sequence with the higher value as long as it received the lower sequence value one before it.

After receiver has constructed an ACK it will check if the has gotten all the segments of the requested data. Should that be the case it will construct an FIN to send to the sender, setting the TCP connection in a half closed state. As the sender gets the FIN it will return a FINACK and set the state to half closed. Upon receiving the FINACK the receiver will send an ACK and close the connection, the sender will close the connection when gets ACK.

3 Results

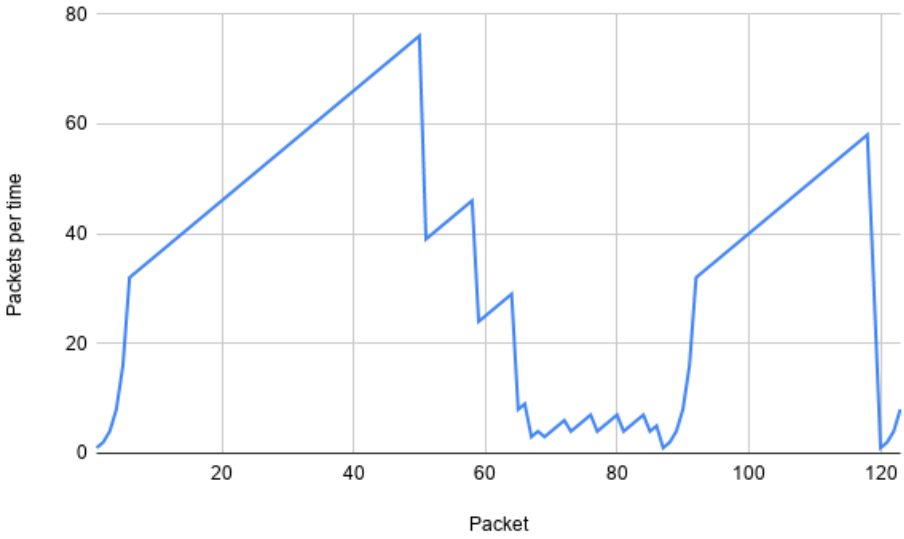


Figure 1: TCP Reno Congestion control, Threshold = 32

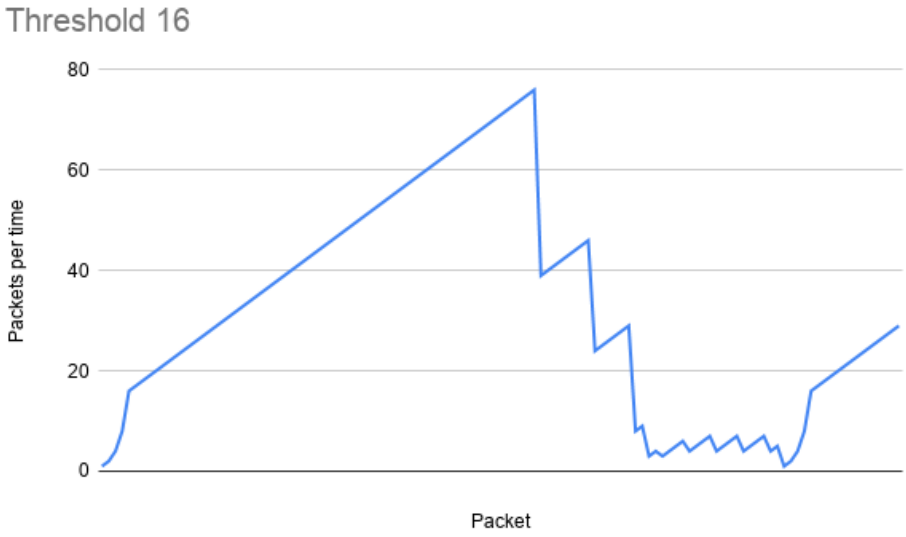


Figure 2: TCP Reno Congestion control, Threshold = 16

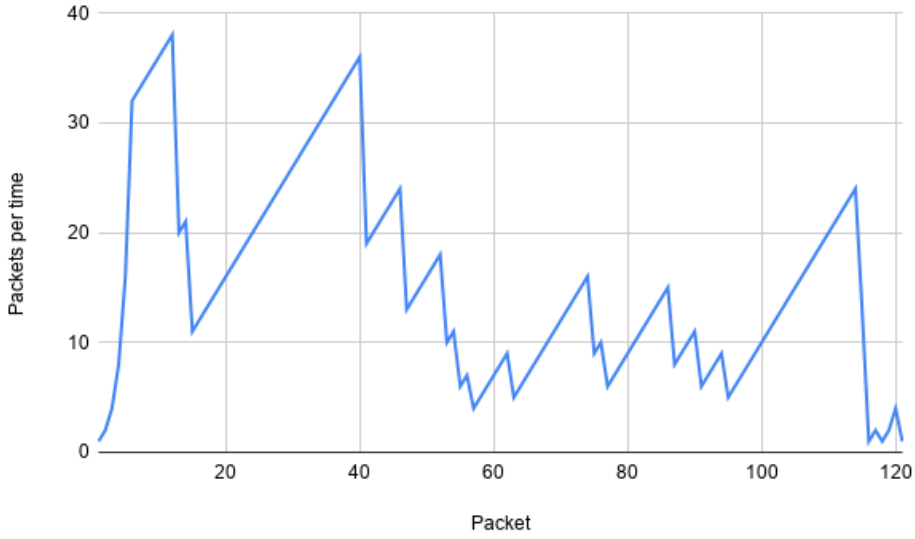


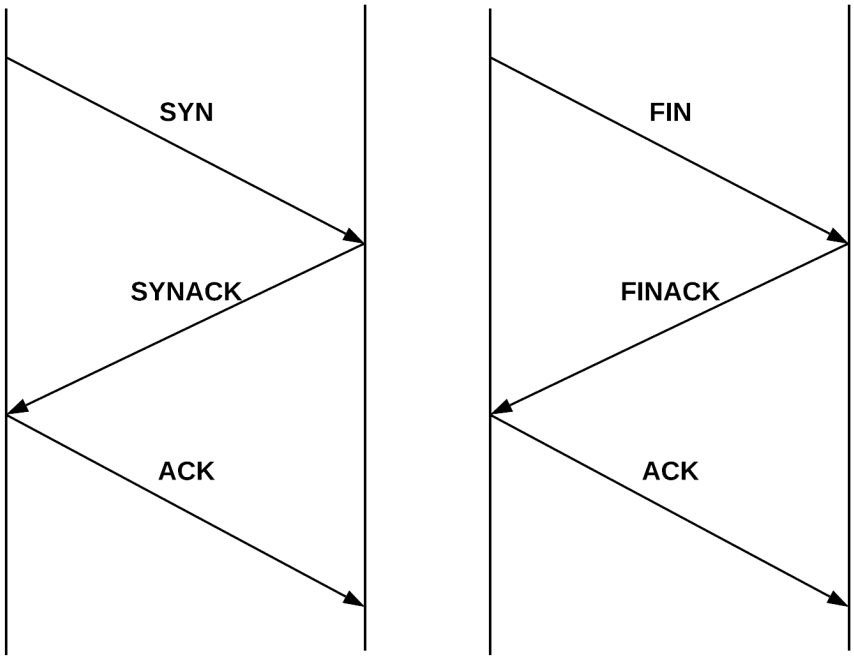
Figure 3: TCP Reno Congestion control, Threshold = 32, Lossylink 10% droprate

```
Node 0.1 send tcp message SYN with seq:0 and ack:-1 to 1.1 at time 0.00
Node 1.1 handling tcp message SYN with seq:0 and ack:-1 from 0.1 at time 0.06
Node 1.1 send tcp message SYNACK with seq:0 and ack:1 to 0.1 at time 0.06
Node 0.1 handling tcp message SYNACK with seq:0 and ack:1 from 1.1 at time 0.12
Node 0.1 send tcp message ACK with seq:1 and ack:1 to 1.1 at time 0.50
Node 1.1 handling tcp message ACK with seq:1 and ack:1 from 0.1 at time 0.56
```

Figure 4: Three-way handshake, opening TCP connection

```
Node 0.1 send tcp message FIN with seq:107 and ack:101 to 1.1 at time 4.96
Node 1.1 handling tcp message FIN with seq:107 and ack:101 from 0.1 at time 5.02
Node 1.1 send tcp message FINACK with seq:101 and ack:108 to 0.1 at time 5.02
Node 0.1 handling tcp message FINACK with seq:101 and ack:108 from 1.1 at time 5.08
Node 0.1 send tcp message ACK with seq:108 and ack:102 to 1.1 at time 5.10
Node 1.1 handling tcp message ACK with seq:108 and ack:102 from 0.1 at time 5.16
```

Figure 5: Four-way handshake, closing TCP connection



(a) Opening TCP connection (b) Closing TCP connection

Figure 6: Establishing TCP connection

```java
public class TCPMessage extends Message {
  public int ack;
  public TCPType type;
  public int data;
  public int segment = 0;
  public int segments = 0;
  public double timeStamp;
  public double ttl;
}
```

Figure 7: TCPMessage.java

```
1  public class TCPConnection extends SimEnt{
2    public enum Config {
3      Sender, Receiver
4    }
5    public enum IncrementStage {
6      Constant, Exponential
7    }
8    public enum ConnectionStage {
9      Opening, Open, HalfClosed, Closed;
10   }
11   private Config config;
12   private ConnectionStage stage;
13   private int seq;
14   private int lastAck;
15   private int nextWantedSeq = -1;
16   private int duplicateAcks;
17   private int dataToFetch;
18   private double ttl = 2;
19   private double rtt;
20   private double srtt;
21   private double congestionSize;
22   private double threshold = 16;
23   private TCPType waitingOn;
24   private IncrementStage incStage;
25   private HashMap<Integer, TCPMessage>waitingOnAck;/// seq, TCPMessage
26   private TCPQueue toSend;
27   private Node self;
28   private NetworkAddr correspondant;
29   private boolean sending = false;
30   private static int slowStartSpeed = 1;
31
32   public TCPConnection(Config config, Node self, NetworkAddr correspondant) {
33     seq = 0;
34     waitingOnAck = new HashMap<Integer,TCPMessage>();
35     this.config = config;
36     toSend = new TCPQueue();
37     this.self = self;
38     this.correspondant = correspondant;
39     congestionSize = slowStartSpeed;
40     incStage = IncrementStage.Exponential;
41     stage = ConnectionStage.Opening;
42     if(config == Config.Sender)
43       waitingOn = TCPType.SYN;
44     else
45       waitingOn = TCPType.SYNACK;
46   }
47
48   public void startConversation() {
49     TCPMessage msg = new TCPMessage(self.getAddr(), correspondant, seq, -1,TCPType.SYN, 0);
50     waitingOn = TCPType.SYNACK;
51     seq++;
52     toSend.addToHead(msg);
53     sending = true;
54     send(this,new TimerEvent(), 0);
55   }
56
57   private TCPMessage getNextMessage() {
58     boolean flag = false;
59     for(TCPMessage msg : waitingOnAck.values()) {
60       if(SimEngine.getTime()>msg.getTimeout()) {
61         System.out.println("Message from "+ self+" with seq: " +msg.seq()+" Timedout at "+
               SimEngine.getTime());
62         timeout();
63         flag = true;
64         break;
65       }
66     }
67     if(flag) {
68       Integer[] keys = new Integer[waitingOnAck.size()];
69       waitingOnAck.keySet().toArray(keys);
70       Arrays.sort(keys, Comparator.reverseOrder());
```

Figure 8: TCPConnection.java 1/5

```
      for(int key : keys) {
        toSend.addToHead(waitingOnAck.remove(key));
      }
    }
    if(!toSend.isEmpty()) {
      TCPMessage msg = toSend.getHead();
      if(msg.type() == TCPType.SYN)
        waitingOn = TCPType.SYNACK;
        else if(msg.type() == TCPType.FIN) {
        stage = ConnectionStage.HalfClosed;
        waitingOn = TCPType.FINACK;
      }
      else if(msg.type() == TCPType.FINACK || msg.type() == TCPType.SYNACK) {
        if(stage == ConnectionStage.Open)
          stage = ConnectionStage.HalfClosed;
        waitingOn = TCPType.ACK;
      }
      else if(stage == ConnectionStage.HalfClosed && msg.type() == TCPType.ACK) {
        stage = ConnectionStage.Closed;
      } return msg;
    } return null;
  }

  public void setDataToFetch(int dataToFetch) {
    this.dataToFetch = dataToFetch;
  }
  private int[]seg = new int[1000];
  public void handleMessage(TCPMessage msg) {
    recv++;
    if(SimEngine.getTime()>=msg.getTimeout()) {
      System.out.println("Drop msg due to past ttl");
    }
    if(!sending) {
      sending = true;
      send(this, new TimerEvent(), 0);
    }
    System.out.println(self + " handling tcp message " + msg.type() +" with seq:"+msg.seq()+" and
        ack:" +msg.ack()+" from " + correspondant + " at time " +SimEngine.getTime());
    if(waitingOn == null) {
      if(config == Config.Sender) {
        messageHandlerSender(msg);
      }
      else if(config == Config.Receiver) {
        messageHandlerReceiver(msg);
      }
    }
    else if(waitingOn == msg.type()) {
      TCPMessage reply;
      switch(waitingOn) {
      case SYN:
        reply = new TCPMessage(self.getAddr(), correspondant, seq, msg.seq()+1, TCPType.SYNACK, 0);
        toSend.addToHead(reply);
        seq++;
        waitingOn = TCPType.ACK;
        send(this,new TimerEvent(), 0);
        break;
      case SYNACK:
        reply = new TCPMessage(self.getAddr(), correspondant, seq, msg.seq()+1, TCPType.ACK, 0);
        waitingOnAck.remove(msg.ack());
        toSend.addToHead(reply);
        stage = ConnectionStage.Open;
        seq++;
        waitingOn = null;
        break;
      case ACK:
        waitingOnAck.remove(msg.ack());
        if(stage == ConnectionStage.Opening)
          stage = ConnectionStage.Open;
        else if(stage == ConnectionStage.HalfClosed)
          stage = ConnectionStage.Closed;
        waitingOn = null;
        break;
```

Figure 9: TCPConnection.java 2/5

```
    case FINACK:
      reply = new TCPMessage(self.getAddr(), correspondant, seq, msg.seq()+1, TCPType.ACK, 0);
      waitingOnAck.remove(msg.ack());
      toSend.addToHead(reply);
      seq++;
      waitingOn = null;
      break;
    default:
      System.out.println("Something went wrong in " + self + " communcatining with " +
          correspondant );
    }
    if(stage == ConnectionStage.Open && dataToFetch > 0) {
      toSend.addToTail(new TCPMessage(self.getAddr(), correspondant, seq, msg.seq()+1,
          TCPType.ACK, dataToFetch));
      seq++;
    }
  }
}
ArrayList<Integer> sentAcks = new ArrayList<Integer>();

private void messageHandlerReceiver(TCPMessage msg) {
  if(msg.type() == TCPType.ACK) {
    if(msg.segments() != 0 && msg.segments() >= msg.segment()) {
      seg[msg.segment()-1] = 1;
      if(nextWantedSeq == -1)
       nextWantedSeq = msg.seq()-msg.segment()+1;
      int temp = nextWantedSeq;
      nextWantedSeq = msg.seq() == nextWantedSeq ? nextWantedSeq+1 : nextWantedSeq;
      int ack = 0;
      if(nextWantedSeq != temp && !sentAcks.contains(nextWantedSeq))
        sentAcks.add(nextWantedSeq);
      if(nextWantedSeq == temp) {
        ack = msg.seq()+1;
        if(!sentAcks.contains(ack))
          sentAcks.add(ack);
        duplicateAcks++;
        if(duplicateAcks>=3)
          threeDupAck();
      }
      else
        while(sentAcks.contains(nextWantedSeq+1))
          nextWantedSeq++;
      TCPMessage reply = new TCPMessage(self.getAddr(), correspondant, seq, nextWantedSeq,
          TCPType.ACK, -ack);
      seq++;
      toSend.addToTail(reply);
      if(dataToFetch==sentAcks.size()) {
        TCPMessage fin = new TCPMessage(self.getAddr(), correspondant, seq, nextWantedSeq,
            TCPType.FIN, 0);
        seq++;
        toSend.addToTail(fin);
      }
      if(waitingOnAck.containsKey(msg.ack()))
        waitingOnAck.remove(msg.ack());
    }
    else {
      if(msg.ack() == lastAck) {
        duplicateAcks++;
      }
      else {
        lastAck = msg.ack();
        waitingOnAck.remove(msg.ack());
        duplicateAcks = 1;
      }
    }
  }
  else if(msg.type() == TCPType.FIN) {
    TCPMessage reply = new TCPMessage(self.getAddr(), correspondant, seq, msg.seq()+1,
        TCPType.FINACK, 0);
    seq++;
    toSend.addToTail(reply);
  }
}

private void messageHandlerSender(TCPMessage msg) {
  if(msg.data() > 0 && msg.type() == TCPType.ACK && dataToFetch == 0) {
```

Figure 10: TCPConnection.java 3/5

```
      dataToFetch = msg.data();
      for(int segment = 1; segment <= dataToFetch; segment++) {
        TCPMessage reply = new TCPMessage(self.getAddr(), correspondant, seq, msg.seq()+1,
            TCPType.ACK, 0);
        reply.setSegment(segment);
        reply.setSegments(dataToFetch);
        toSend.addToTail(reply);
        seq++;
      }
      lastAck = msg.ack();
    }
    else if(msg.type() == TCPType.ACK) {
      if(msg.ack() == lastAck) {
        duplicateAcks++;
        if(msg.data() != 0)
          waitingOnAck.remove(-msg.data());
        if (duplicateAcks>=3) {
          System.out.println(self + " got triple ack on ack " + lastAck);
          threeDupAck();
          duplicateAcks = 0;
          toSend.addToHead(waitingOnAck.remove(msg.ack()+1));
          return;
        }
      }
      else if(msg.ack() == lastAck+1) {
        lastAck = msg.ack();
        waitingOnAck.remove(msg.ack());
        duplicateAcks = 1;
      }
      else if(msg.ack() > lastAck) {
        lastAck = msg.ack();
        Integer[] keys = new Integer[waitingOnAck.size()];
        waitingOnAck.keySet().toArray(keys);
        Arrays.sort(keys, Comparator.reverseOrder());
        for(int key : keys) {
          if(key<msg.ack())
            waitingOnAck.remove(key);
        }
        duplicateAcks = 1;
      }
    }
    else if(msg.type() == TCPType.FIN) {
      TCPMessage reply = new TCPMessage(self.getAddr(), correspondant, seq, msg.seq()+1,
          TCPType.FINACK, 0);
      seq++;
      toSend.addToTail(reply);
    }
  }

  public NetworkAddr correspondant() {
    return correspondant;
  }

  public double getRTT() {
    return rtt;
  }

  public void setRTT(double rtt) {
    this.rtt = rtt;
    if(srtt == -1)
      srtt = rtt;
  }

  public double calculateSRTT() {
    double alpha = 0.8; //between 0.8 and 0.9
    double value = (alpha * srtt) + ((1 - alpha) * rtt);
    srtt = value;
        return value;
  }

  public double getRTO() {
    double beta = 1.3; //between 1.3 and 2.0
    return Math.min(64, Math.max(1, (beta * srtt)));
```

Figure 11: TCPConnection.java 4/5

```
  }

  public void threeDupAck() {
    congestionSize = (int)Math.ceil(congestionSize/2.0);
    incStage = TCPConnection.IncrementStage.Constant;
    if (congestionSize < 2) {
      congestionSize = (2);
    }
  }

  public void timeout() {
    congestionSize = slowStartSpeed;
    incStage = TCPConnection.IncrementStage.Exponential;
  }

  public void reachedThreshold() {
    incStage = TCPConnection.IncrementStage.Constant;
  }

  private void updatingSendingRate() {
    if(incStage == IncrementStage.Constant)
      congestionSize++;
    else {
      congestionSize = congestionSize * 2;
      if(congestionSize >= threshold)
        reachedThreshold();
    }
  }
  private int sent = 0;
  private int recv = 0;
  @Override
  public void recv(SimEnt source, Event event) {
    if(event instanceof TimerEvent) {
      if(stage != ConnectionStage.Closed) {
        sending = true;
        TCPMessage msg = getNextMessage();
        if(msg != null) {
          System.out.println(self + " send tcp message " + msg.type() +" with seq:" +msg.seq()+"
                and ack:" +msg.ack()+" to " + correspondant + " at time " + SimEngine.getTime());
          if((config == Config.Sender && stage == ConnectionStage.Open && msg.type() ==
                TCPType.ACK)||msg.data()>0|| waitingOn != null)
            waitingOnAck.put(msg.seq()+1, msg);
          msg.setTTL(ttl, SimEngine.getTime());
          self.sendTCP(msg);
          try {
            Logger.LogTime(self.toString(), Double.toString(congestionSize));
          } catch (IOException e) {
            e.printStackTrace();
          }
          updatingSendingRate();
          sent++;
        }
        else {
          System.out.print("");
        }
        if(!toSend.isEmpty() || !waitingOnAck.isEmpty()) {
          send(this, new TimerEvent(), 1.0/congestionSize);
          return;
        }
        else
          sending = false;
        if(toSend.isEmpty() && waitingOnAck.isEmpty())
          if(stage == ConnectionStage.Closed)
            System.out.println(self + " communcation with "+ correspondant +" has ended sent "+
                  sent + " and received "+ recv );
          return;
      }
      else {
        System.out.println(self + " communcation with "+ correspondant +" has ended sent "+ sent +
              " and received "+ recv );
      }
    }
    return;
  }
}
```

Figure 12: TCPConnection.java 5/5

4 Discussion

This project in general caused us a lot of headaches, mainly because we didn't start out by designing our overall picture from the beginning, we just started working and dealt with things as they popped up. We also had a hard time knowing how to limit our implementation of TCP because there are so many parts of it, and we could not implement them all.

As can be seen in Fig. 1, the simulation starts with slow start until we reach the threshold of 32, chosen because it was a base of 2 and resulting in the most easily interpreted graph. After reaching the threshold we go to congestion avoidance that increases constantly until it receives a triple duplicate ack and drops the congestion window to half. After the first triple duplicate ack we can observe multiple triple duplicate acks consecutively. This is due to the fact that during the first third of the simulation there are a lot of packets in transmission that have not received acknowledgements yet. After all of the triple duplicate acks we get a timeout which drops the congestion window to 1 and slow start is starting again until the threshold is reached.

In comparison to Fig. 1 we can see in Fig. 2 that they both have a reached a similar speed before receiving the first triple duplicate ack. We can see that threshold affects is the time to reach peak speed.

In Fig. 3 the odds that a packet is dropped is higher, as it is not based on flooding. As we do not use RTT to determine TTL for packets. The time for a packet to timeout is much longer than needed resulting in more triple duplicate acks than would otherwise occur.

References

[1] Information Sciences Institute. Rfc 793: Transmission control protocol.