TCP enhancements for the integration of satellite links in the Internet. Modeling and simulation study of Tahoe, Reno and SACK TCP behavior.

Hugo Grosmangin
Advisor: John Baras
Abstract

The performances of TCP over satellite can reveal very poor due to the long delay, high bandwidth and BER characteristics. In order to integrate satellites link in the Internet several algorithms and options have been created: among them Fast Retransmit, which avoid waiting for a timeout before a retransmission, Fast Recovery, which avoid doing slow start after a retransmission, and the SACK option, which permits better retransmissions. The overall project consisted in modeling these algorithms and options within TCP with Opnet in order to provide public models. These models were realized with Opnet because it is one of the most used network modeling and simulating software. This report contains a discussion on the problems of TCP over satellite, a description of the models, the algorithms and options they implement, and the results they provide.
Contents

1 Integration of a satellite link: overview of the problem ......................... 4
  1.1 Characteristics of a satellite link ........................................... 4
  1.2 Integration of the link: problems ............................................ 4
  1.3 Spoofing TCP ................................................................. 5
     1.3.1 Principle ............................................................... 5
     1.3.2 Advantages ............................................................ 6
     1.3.3 Simulation example ...................................................... 6

2 TCP enhancements for better throughput ............................................. 8
  2.1 Standard TCP behavior ....................................................... 8
     2.1.1 Slow Start ............................................................... 8
     2.1.2 Congestion Avoidance ................................................... 9
     2.1.3 Retransmission strategy ............................................... 9
  2.2 TCP enhancements ............................................................ 10
     2.2.1 Fast Retransmit ......................................................... 10
     2.2.2 Fast Recovery .......................................................... 11
     2.2.3 Sack Option ............................................................. 11

3 TCP models ................................................................................. 16
  3.1 Opnet overview ....................................................................... 16
  3.2 The standard TCP model of Opnet ............................................ 16
     3.2.1 Process ........................................................................ 16
     3.2.2 Algorithms and options implemented .................................. 19
     3.2.3 Attributes ..................................................................... 19
  3.3 Multi TCP process model .......................................................... 20
     3.3.1 Process and attached files ............................................... 20
     3.3.2 New algorithms and options implemented .......................... 21
     3.3.3 New attributes .............................................................. 21
  3.4 Simulation model ..................................................................... 21
     3.4.1 Goals ......................................................................... 22
     3.4.2 Node model ................................................................... 22
     3.4.3 Network model .............................................................. 22
4 Results

4.1 Single packet loss in the send window .......................... 24
  4.1.1 Parameters ........................................... 24
  4.1.2 Tahoe TCP ........................................... 24
  4.1.3 Reno TCP ........................................... 24
  4.1.4 SACK TCP ........................................... 26
  4.1.5 Throughput comparison .................................. 26

4.2 Multiple packet loss in the send window ......................... 27
  4.2.1 Parameters ........................................... 28
  4.2.2 Tahoe TCP ........................................... 28
  4.2.3 Reno TCP ........................................... 28
  4.2.4 SACK TCP ........................................... 30
  4.2.5 Throughput comparison .................................. 30

5 Future work ................................................... 33

A Code overview ................................................. 35

B Work overview ................................................ 37
List of Figures

1.1 A network integrating a satellite link ...................................... 5
1.2 A network model spoofing TCP .............................................. 6
1.3 RTT seen by the server for the connections with host and hybrid host .... 7
2.1 Slow start and congestion avoidance in Opnet TCP model ............... 9
2.2 Construction of SACK blocks at the data receiver .................... 13
3.1 The TCP manager process ..................................................... 17
3.2 The TCP connection process ............................................... 18
3.3 Node model .................................................................... 23
3.4 Network model .................................................................. 23
4.1 Congestion window of the Tahoe sender with single packet losses within the send window ........................................... 25
4.2 Congestion window of the Reno sender with single packet losses within the send window ........................................... 25
4.3 Congestion window of the SACK sender with single packet losses within the send window ........................................... 26
4.4 Sent sequence number when single packets are lost in the send window 27
4.5 Throughput comparison in case of single packet losses within the send window ........................................... 27
4.6 Congestion window of the Tahoe sender with five packet losses in the same window ........................................... 28
4.7 Tahoe Retransmission with five packets losses in the same window ..... 29
4.8 Congestion window of the Reno sender with five packet losses in the same window ........................................... 29
4.9 Reno retransmission with five packet losses in the same window ...... 30
4.10 Congestion window of the SACK sender with five packet losses in the same window ........................................... 31
4.11 SACK retransmission with five packet losses in the same window ...... 31
4.12 Throughput comparison in case of five packet losses in the same window 32
Chapter 1

Integration of a satellite link: overview of the problem

1.1 Characteristics of a satellite link

Satellite channels have several characteristics that differ from most terrestrial channels. These characteristics can degrade the performance of TCP. These characteristics include:

- Long feedback loop: due to the long propagation delay (approximately 250 ms), it takes a large amount of time for TCP to know if the data sent has arrived successfully.

- Large delay × bandwidth: to fully utilize the bandwidth TCP has to keep a lot of data transmitted but not yet acknowledged.

- Transmission errors: the satellite link have a larger BER than terrestrial links. Corrupted packet are dropped and assumed to be lost because of congestion by TCP. The size of the window is reduced although there is no actual congestion.

1.2 Integration of the link: problems

The throughput of TCP is limited by:

\[
\text{Throughput} = \frac{\text{windowsize}}{RTT}
\]

Let us see the consequences with figure 1.1. If we consider a connection between host 3 and host 4, it will experience a long RTT since it goes over a long delay link. Moreover, in the commercial TCP the receive window size is limited to 4096 bytes. As a consequence, this connection will experience a lower throughput compared to a connection between two hosts of network A (host1, host2).
Moreover, to optimize TCP's behavior over the satellite link, special algorithms and options should be used.

Figure 1.1: A network integrating a satellite link

1.3 Spoofing TCP

1.3.1 Principle

The basic idea is to split the network into three parts: network A, satellite link, network B. The gateway acknowledges packets on behalf of the end-user (except during the establishment of the connection). Indeed, when the gateway receives a packet from host 3, for example, it acknowledges it, puts it into its send buffer for the satellite link and then sends it, but it also keeps copy of it in its retransmission buffer until it receives an acknowledgement for it from the other end. The gateway eventually has to deal with retransmissions. The gateway can also advertise a larger window to the host which means a better throughput for the connection. So big memory capacities are required in the gateway to store all the packets not yet acknowledged.
1.3.2 Advantages

- The use of a satellite link is transparent for host 3. For host 3 it is like a connection with a host of network A. Indeed host 3 sees the same RTT as for a connection in network A.

- Since the satellite link is "isolated", special algorithms and options can be implemented in TCP in order to improve its performance over the satellite link.

1.3.3 Simulation example

A simulation was realised on the network model given in figure 1.2: The server delivers data to the host, on the same LAN and the hybrid host, over a link with long delay (250 ns) and a bandwidth of 10 Mbits. The gateway implements asymmetric TCP which do spoofing. The connection between the server and the host begins a few seconds after the connection with the hybrid host.

We can see in figure 1.3 that the server sees the same RTT for both connections.
Figure 1.3: RTT seen by the server for the connections with host and hybrid host
Chapter 2

TCP enhancements for better throughput

2.1 Standard TCP behavior

We are here quickly describing the standard TCP behavior specified in RFC 793 and RFC 1122. Especially from the point of view of congestion control and retransmission strategy because these are very important points when considering TCP efficiency.

2.1.1 Slow Start

When sending packets TCP has to consider two windows: the window advertised by the remote receiver and its own congestion window. The first one represents the availability of the remote receiver for receiving packets (it is a flow control from the receiver) and the second one is more an estimation of the available bandwidth in the network (it is a flow control from the sender). TCP takes the minimum of the two previous window as its own send window.

The slow start algorithm operates by observing that the rate at which new packets should be injected into the network is the rate at which the acknowledgements are returned by the other end. This algorithm is used at the beginning of a new connection and after a retransmission. The congestion window is initialized to one segment size and each time the sender receives an acknowledgement it increases this window by one segment size. This algorithm provides an exponential increase. At a certain point when the congestion window reaches a threshold called ssthresh slow start stops and then congestion avoidance operates. generally the initial value of ssthresh is 65535 bytes.
2.1.2 Congestion Avoidance

This algorithm operates in the same way as slow start: it modifies the congestion window according to acknowledgement arrivals. The difference is that the growth of the congestion window is slower because when it reaches the ssthresh we do not want to send data too fast in order to avoid congestion but we still want to increase the window size to fully use the available bandwidth of the network. The congestion window (cwnd) is incremented this way each time a new acknowledgement is received:

\[
    \text{cwnd} \leftarrow \text{cwnd} + \frac{\text{acknowledgments}}{\text{cwnd}}
\]

An example of slow start followed by congestion avoidance in the Opnet standard TCP model is given in figure 2.1.

![Graph showing TCP congestion window size over time](image)

Figure 2.1: Slow start and congestion avoidance in Opnet TCP model

2.1.3 Retransmission strategy

Basically, the standard TCP detects a packet drop by not receiving any acknowledgments for it: a retransmission timer is used when expecting an acknowledgement from the other end. When this timer expires the sender retransmits the first unacknowledged segment, set ssthresh to half the current window (the minimum of cwnd and the window advertised) but no less than two segments and then set cwnd to one segment size and do slow start.
In fact TCP always assumes that a packet loss is due to congestion and estimates the available bandwidth of the network according to these packet losses. This is not a serious problem with terrestrial networks since most of the packet losses in these networks are due to congestion but in case of satellite links a lot of packets are lost because of corruption and TCP behaves as if the network was congested and do slow starts. The consequence is very bad for the throughput. Indeed, after the retransmission TCP sends one packet and waits for an acknowledgement which is supposed to arrive one RTT after the send time, and then it will send two packets, wait for an acknowledgement again and so on. With the large RTT due to the long delay and the very small window the effective bandwidth drops to a very small value whereas there is actually no congestion. So the performances of standard TCP over satellite reveal very poor but there are a certain number of options and algorithms that should improve these performances.

2.2 TCP enhancements

2.2.1 Fast Retransmit

The fact is that in the previous description, TCP does not make any difference between a full congestion when the receiver does not receive any segment and a temporary congestion when only one or a few packets are dropped but the following ones are received. The fast retransmit algorithm allows TCP to make the difference between the two previous cases.

Indeed the receiver TCP is required to generate an immediate acknowledgement (a duplicate ACK) when an out of order segment is received. This acknowledgement should not be delayed. The sender recognizes a duplicate ACK with the following test:

- It is a dataseless acknowledgement.
- It has the same window as the last acknowledgement.
- Outstanding data has not been acknowledged.
- It has the same sequence number as the largest number we have seen in an acknowledgement.

A single duplicate ACK is not significant of a packet loss since it could be generated if a segment had been delayed but the sender assumes that a certain number of duplicate ACK received in a row (usually three or four) is a good indication that a packet has been lost. Thus, after several duplicate ACK the sender retransmits the packet assumed to be missing without waiting for a timeout which leads to a higher utilization of the channel and a better throughput. Moreover duplicate ACK show
that the remote receiver keeps on receiving segments so it does not appear to be
necessary to slow start after the retransmission since data is still flowing between
the two ends, this is why the fast retransmit algorithm is usually used after fast
retransmit.

2.2.2 Fast Recovery

The basic idea of fast retransmit is to keep on sending new data after the retransmis-
sion while expecting for a new acknowledgement: the sender estimates the amount
of data in the network and sends a packet when another one has left the network
to avoid overloading the network. Indeed, a duplicate ACK means that the receiver
has buffered an out of order segment and, thus, that a segment has left the network.
When a new acknowledgement is received (acknowledging new data) the sender goes
back to congestion avoidance from a value of cwnd equal to half the value of the
current window just before the retransmission. Here is how fast retransmit and fast
recovery are usually implemented together:

- When the third duplicate ACK is received, set ssthresh to one-half the mini-
  mum of the current congestion window and the receiver’s advertised window.
  Retransmit the missing segment.
  Set cwnd to ssthresh plus three times the segment size (three segments have
  left the network).

- Each time another duplicate ACK arrives, increment cwnd by the segment size
  and transmit a packet (if allowed by the new value of cwnd): an other segment
  has left the network.

- When the next ACK arrives that acknowledges new data, set cwnd to ssthresh
  (the value set in the first step). This should be the ACK for the retransmission
  of the first step, one RTT after the retransmission. Additionally, this ACK
  should acknowledge all the intermediate segments sent between the lost packet
  and the receipt of the first duplicate ACK. This step is congestion avoidance
  since we are slowing down to half the rate we were when the packet was lost.

Fast retransmit and fast recovery should improve TCP performance on a satellite link
since we avoid slow start and so we don’t drop abruptly to a window of one segment.
However, in case of several packet losses from the same window, TCP enters and
exits several times fast recovery until its usable window equals the amount of data
not yet acknowledged. In this case TCP can only wait for a retransmission timer
to expire and this will cause slow start. In order to avoid this case and unnecessary
retransmission the SACK option should be used.

2.2.3 Sack Option

What is commonly named SACK option is composed of two options in fact: the
SACK permitted option and the SACK option. The SACK permitted option is used
to establish an agreement between the two TCP in connection for the further use of the SACK option.

**SACK Permitted Option**

This two bytes option may be sent on a SYN segment by a TCP that has been extended to receive and process the SACK option once the connection has opened. It must not be sent on a non-SYN segment. The first byte of this option contains the kind which is four, the second byte the length in bytes which is obviously two.

**SACK Option**

The SACK option is to be sent by the data receiver to the sender to inform the sender of the non-contiguous blocks of data received and help him in case of retransmission. The cumulative acknowledgement principle is kept, the receiver expects its holes in sequence space to be filled by incoming packets to send an acknowledgement for new data. A SACK does not stand for an ACK, it is just advisory.

The first byte of this option contains the kind of the option which is 5. The second byte contains the length of the option which is variable according to the number of blocks reported. Then follow the list of blocks. A block is described by two 32 bits unsigned integers in network byte order. The first one is the left edge of the block which is the first sequence number of the block. The second one is the right edge of the block which is the first sequence number expected after the block. A block stands for the contiguous data received in sequence space. As a consequence an option of n blocks will have a length $8 \times \alpha + 2$ bytes. Since the option field is limited to 40 bytes, the SACK option cannot contain more than four blocks. Moreover, the SACK option is generally used in conjunction with other options (as Timestamps which is two bytes long) which often limits the number of blocks to three.

**Data receiver behavior**

If the receiver has received a SACK permitted option on the SYN segment for this connection it should generate a SACK option on each ACK that does not acknowledge the higher sequence number in the receiver's queue. If not, the receiver must not generate a SACK option in any case. Thus, every duplicate ACK should bear a SACK option. If the receiver chooses to send a SACK option the sent blocks should respect the following rules (an example is given in figure 2.2):

- The first SACK block must specify the block of data containing the segment which triggered this ACK, unless that segment advanced the acknowledgement number in the field. This assures that the ACK with the SACK option reflects the most recent change in the receiver's queue.

- The data receiver should include as many distinct blocks as possible in the SACK option. Note that the available space may not permit to report all the
blocks of the receiver's queue.

- The SACK option should be filled out by repeating the most recently reported blocks (based on the first block reported in the previous SACK options) that are not subsets of a SACK block already included in the SACK option being constructed. This assumes that, in normal operation, any segment remaining part of a non-contiguous block of data held by the receiver is reported in at least three successive SACK options, even for large window TCP implementations. After the first SACK block, the following SACK blocks in the SACK option may be listed in arbitrary order. However, SACK blocks must not report any old data which is no longer actually held by the receiver if it has renegoted.

```
SACK blocks to be sent

TCP A
SEQ 1512
SEQ 518-5024
SEQ 925-4236
SEQ 1537-2048
SEQ 2562-2560
SEQ 2561-972
SEQ 3978-5584
SEQ 3382-8996
SEQ 4097-6068

TCP B
No block
No block
Block 1537:3040
Block 2561:3073
Block 2561:3584
Block 4397:4069

TCP A
TCP B

Figure 2.2: Construction of SACK blocks at the data receiver
```

**Data sender behavior**

*Interpreting the SACK option*

When receiving an ACK containing a SACK option, the data sender should
record the selective acknowledgement for future reference. If the data sender performs re-packetization before retransmission, the block boundaries may not fall on boundaries of segments in the retransmission queue; however, this does not pose a serious difficulty for the sender.

Let us suppose that for each segment in the retransmission queue there is a flag bit "SACKed" which indicates that this segment has been reported in a SACK option. When the SACK option is received by the sender it must turn on this bit for every segment in the retransmission queue which is fully contained within a block. This requires straightforward sequence number comparisons. TCP must not flush SACKed data from its retransmission buffer before having received an ACK for it.

Congestion control and retransmission strategy

We describe here the behavior TCP should have to stay close to fast retransmit and fast recovery. This is Sally Floyd's algorithm using the `pipe` variable which is an estimation of the amount of data in the network.

This algorithm is usually implemented as follows:

- When the third duplicate acknowledgement is received, set `ssthresh` to one half the current window.
  Retransmit the missing segment.
  Initialize `pipe` to `cwnd` minus three times the segment size
  Set `cwnd` to `ssthresh`
  Record the recover ACK which is the acknowledgement expected for all the data being sent before the retransmission.

- each time another duplicate ACK is received decrement `pipe` by one segment size.
  If `pipe` is less than `cwnd` retransmit a segment that has not been SACKed and increment `pipe` by one segment size. If all non-SACKed segment have already been retransmitted, send new data.

- If a partial ACK (an ACK for new data but not as much as for the recover ACK) is received decrement `pipe` by two segment size. And perform retransmission as in the previous step.

- when the recover ACK is received TCP returns to normal behavior and performs congestion avoidance.

The `pipe` variable is decremented by one segment size when a duplicate ACK is received because it means that a segment has left the network. The `pipe` variable is decremented by two segment size when a partial ACK is received because it means that the segment that triggered this ACK has left the network and that the next expected segment by the receiver has been lost (which makes two
segments out of the network).
In case of a time out, all previously SACKed data must be "unsACKed".
Chapter 3

TCP models

3.1 Opnet overview

The simulation has been done within OPNET, Optimized Networks Engineering Tools, which is a hierarchical modeling and simulating package. OPNET is an event-driven simulating software. That means that time advances whenever scheduled events occur, modeling is done in a hierarchical fashion from the very lowest elements which are the Finite State Machine to the network as a whole. The fundamental concepts within OPNET are Packets, Interrupts and Inter-process communication. The way a node works in a particular sub-network is programmed in Finite State Machines with Proto-C. Proto-C is very close to C language. Proto-C just adds to C a large library of routines implemented within OPNET and dealing with the handling of packets, interrupts and inter-processes communication. OPNET offers besides the modeling and the simulating tools powerful tools to analyze the results of the simulation.

3.2 The standard TCP model of Opnet

3.2.1 Process

The standard TCP model of Opnet invokes two processes: TCP manager (see figure 3.1) and TCP connection (see figure 3.2). The first one, TCP manager, is a root process which works continuously. The second one TCP connection is invoked whenever a new connection is created. TCP connection is a child process of TCP manager which handles a TCB, Transmission Connection Table and routes packets toward the adequate TCP Connection process. There are as many TCP Connection processes as connections. In that way, each TCP module can handle as many
connection as made possible by the available resources (up to 32 connections), in a
dynamic fashion.

![TCP Manager Process Diagram](image)

**Figure 3.1: The TCP manager process**

The TCP manager process

The TCP manager receives all commands from the application and lower layers
and directs them to the connection process for which the command was received.
It also routes the received packets toward the right TCP connection process. The
identification is done with the socket which respects the usual format: IP address (not
number and node number) and port number. From the socket, TCP manager issues
a connection ID which identifies the corresponding TCP connection process.
The ICI interfaces are the following with the application:

- "tcp command" ICI is used by the application for issuing a command to the
TCP module. The ICI is used to identify the target TCP connection, so that
the command can be forwarded to the correct connection process. The ICI is
also used to pass command arguments and parameters to TCP.

- "tcp status ind" is used by TCP to notify the application of important events
(if the connection aborts for example). The connection creates the ICI but is
The packet format (tcp seg) used to encapsulate application data for processing by the network has 17 fields which specify addresses, sequence numbers, maximum segment size and status flag used by the protocol.

Connection to the lower layer should be made on input and output streams 0. Any other input and output streams (numbered 1 or greater) may be connected to the application. Application may have one pair of packet streams connected to the manager.

TCP connection process

The TCP connection process is based on the TCP specified in RFC 793 and RFC 1122. It must be declared as a child process of TCP manager.

The ICI Interfaces are the following:

- "ip encap req" is used in conjunction with packet transfers to the lower layer. It contains the fields "dest node" and "dest net" for specifying the packet's destination node and network number.

- "tcp status ind" ICI is used to alert the application layer about the status of the connection. It contains fields "conn ID" which specifies the connection ID,
“status” which specifies the status of the connection and “urgent” which is used for urgent data transfer.

It uses the same packet format as the manager.

3.2.2 Algorithms and options implemented

The Opnet standard TCP model sticks to the specification of TCP in RFC 793 and 1122. The following options and algorithms are implemented:

- Three way handshake protocol for opening and closing the connection.
- Slow start, congestion avoidance and flow control.
- Computation of the smoothed round trip time using Jacobson’s algorithm in order to update the value of the retransmission timeout.
- MSS option.
- Persistence timeout.
- Karn’s algorithm.
- Nagle’s algorithm.
- Delayed ACK.
- Resequencing

Still there are some unimplemented features: The Quiet Time mechanism is not implemented and TCP checksums are not computed.

3.2.3 Attributes

There are a certain number of attributes that can be set by the user in the model in order to adapt TCP to the topology of the network or in order to modify parameters that can be relevant for performance evaluation. Here is a complete list of these attributes:

- RCV BUUFF: size of the receive buffer.
- Maximum ACK Delay: time expected before sending a dataless ACK if the ACK has not been piggybacked.
- Maximum Segment Size: MSS that the underlying network can carry without fragmenting.
- Initial RTO
- Minimum RTO
- Maximum RTO
- RTT Gain: for RTT measurement
- Deviation Gain: for RTT mean deviation
- RTT Deviation Coefficient: used to determine the effect of mean deviation on the final calculated RTO.
- Nagle SWA Avoidance: enables or disables Nagle’s algorithm
- End-to-End Delay Measurement: enables or disables measurement of ETE delay.
- Persistence Timeout: Duration of the persistence Timeout.
- Karns Algorithm: enables or disables the use of this algorithm.
- RCV BUFF Usage Threshold: usage of the buffer expected before transferring segments to the socket buffer.

### 3.3 Multi TCP process model

#### 3.3.1 Process and attached files

The Multi TCP process is an extension of the Opnet standard TCP. As a consequence, its architecture is the same: two processes, the root process multi TCP manager and the child process multi TCP connexion. These processes have the same ICI interfaces as the standard model.

They use the same packet format “tcp seg” plus two new ones: “tcp seg1” for the transmission of SACK permitted option and “SACK tcp seg” for the transmission of the SACK option.

These processes include the header files “oms-pr.h”, “multi.tcp.h” and “multi.tcp.api.h” which are actually the same as those in the Opnet TCP library except for “multi.tcp.h” which contains the declaration of the data structure “TcpT.Block” used for representing SACK blocks.

In order to manipulate easily retransmission a list of the unacknowledged segments is created and updated along the process: the “scoreboard”. Still the retransmission buffer has not been suppressed and is updated with the scoreboard.

The multi TCP connexion offers three functionalities. The user must choose one them before any simulation according to the desired behavior: Tahoe, Reno or SACK type.

As a consequence, for all networks and node models using the Opnet standard
TCP model in the transport layer, it is easy to switch to this model by just replacing the TCP manager by the multi tcp manager and setting the functionalities and new attributes. For more information about the code see appendix A.

3.3.2 New algorithms and options implemented

These models implement the previously described algorithms and options of the Opnet standard model. For each functionality different new algorithms and options are implemented.

- Tahoe TCP: the fast retransmit algorithm followed by slow start are implemented.
- Reno TCP: the fast retransmit algorithm followed by fast recovery are implemented.
- SACK TCP: the SACK option and SACK permitted option are implemented.

The implementation of these algorithms and options follow exactly the description made in 2.2 except on the following point.
In the SACK mode the field for the SACK option is set in every segment. When there is no block to send this field is filled with empty blocks.
The number of blocks is limited to 3.

Note that if the SACK mode is used and that no SACK permitted option is received the default mode is Reno.

3.3.3 New attributes

This model adds new attributes to those already existing in the Opnet standard TCP model:

- Maximum Duplicate ACK: which is the maximum number of duplicate ACK expected by the TCP before fast retransmit.
- Tahoe.
- Reno.
- SACK.

The three last attributes are used to set the functionality. The desired functionality should be set to one and the other ones to zero. If more than one functionality is set multi TCP connexion will return an error at the beginning of the simulation.

3.4 Simulation model
3.4.1 Goals

The simulation has two goals. The first one is to validate the model. Then, once it is validated, the simulation should permit us to compare the performance of the different functionalities.

Fast retransmit, fast recovery and the SACK option should improve TCP's performances in case of packet losses. This is the reason why the simulation should show the model's behavior in case of packet losses. Moreover, since the spirit of the project is the behavior of TCP over a satellite link, the simulation should show the performances over a link with long delay and high bandwidth characteristics.

In this case the complexity of the network is not relevant. Basically, we simulate a connection between two applications over a link with a high bandwidth and a long delay. The connection uses the TCP/IP stack.

3.4.2 Node model

- The application module provides the data to be sent and the ICI interfaces requested by TCP.
- The IP module also provides the right ICI interfaces to TCP and routes the packets.
- The TCP module uses the multi TCP model.
- The pk.kill module simulates critical cases of packet losses in order to test the TCP model.

3.4.3 Network model

- The delay of the links between the nodes was set to 250 ms.
- The bandwidth of the links was set to 10 Mbits.
Figure 3.3: Node model

Figure 3.4: Network model
Chapter 4

Results

4.1 Single packet loss in the send window

4.1.1 Parameters

This test was realized with the previously described models. There is data flowing in both directions between the two hosts. But packet are lost only for data flowing from host A to host B.

In this test a single packet loss in a window was simulated every 50s for a 700s long simulation.

We check for the congestion window behavior in each mode (Tahoe, Reno and SACK) and then we compare the amount of data sent by host A and the global throughput in the different cases. We set the size of the receive buffer to a large value so that the window size is actually the congestion window. The window size is not limited by the available space in the receiver's buffer.

4.1.2 Tahoe TCP

As expected, we see in figure 4.1 that, each time a packet is lost, the sender slow starts after the retransmission. Moreover sendtime decreases with the retransmission because in this case the window size does not have the time to double between the packet losses. The consequence is a short average window size which limits the throughput.

4.1.3 Reno TCP

We can see in figure 4.2 that, when a packet is lost, during fast recovery the conges-
tion window increases with the reception of duplicate ACK from the new sendtime plus three MSS. Then, when a new ACK has been received TCP performs congestion
Figure 4.1: Congestion window of the Tahoe sender with single packet losses within the send window.

Figure 4.2: Congestion window of the Reno sender with single packet losses within the send window.
avoidance from $athresh$. The consequence is a better average window size which should improve the throughput compared with Tahoe.

4.1.4 SACK TCP

The figure 4.3 shows us the congestion window of the sender using SACK. Basically, the behavior is the same as Reno in case of single packet loss. The congestion window doesn't grow with the reception of duplicate ACK because during fast recovery TCP manages retransmission using the pipe variable. But after recovery, as Reno, congestion avoidance is performed from $athresh$.

4.1.5 Throughput comparison

As we have seen, the average window size is low for Tahoe. Reno and SACK have roughly the same average window size, better than Tahoe. Since the delay on the link is the same for each simulation, the throughput should be the same for Reno and SACK, better than Tahoe. That is what we observe on figure 4.5. We also observe the better behavior of Reno and SACK on figure 4.4: during slow start the amount of data sent is very low and Tahoe behaves badly. However if we look at the characteristics of the satellite links, we observe that, when the channel gets bad, it gets really bad and several packet can be lost in the same window.
Figure 4.4: Sent sequence number when single packets are lost in the send window

Figure 4.5: Throughput comparison in case of single packet losses within the send window

4.2 Multiple packet loss in the send window
4.2.1 Parameters

We realized a 100s simulation, still with the same models. In this simulation data is only flowing from host A to host B, the return link only carries acknowledgements. We simulated a five packet loss in the same window at time around 10s.

4.2.2 Tahoe TCP

The behavior of Tahoe is close to the case of one packet loss. In figure 4.6, we see

![Graph showing Tahoe TCP congestion window size (bytes) over time (sec)](image)

Figure 4.6: Congestion window of the Tahoe sender with five packet losses in the same window

that the congestion window also drops to one MSS, then slow start is performed. However we shall note in figure 4.7 that Tahoe retransmits seven packets whereas only five were lost. These unnecessary retransmissions use bandwidth for nothing.

4.2.3 Reno TCP

The behavior of Reno is really bad in this case: for the first packet loss it performs fast retransmit and recovery (see figure 4.8. Then, when a new acknowledgement is received, it goes out of fast recovery but it has been unable to send new data because its usable window is null. It has no other solution than waiting for a timeout to retransmit each of the following missing segments. Moreover, after a timeout the backoff factor is multiplied by two and thus the following timeout is twice longer than the previous.
Figure 4.7: Tahoe Retransmission with five packets losses in the same window

Figure 4.8: Congestion window of the Reno sender with five packet losses in the same window
The consequence is a very long time with only a couple of packet sent. We see on figure 4.9 the the window stays to one MSS for a very long time. The consequence on the throughput is catastrophic.

Figure 4.9: Reno retransmission with five packet losses in the same window

4.2.4 SACK TCP

The behavior of SACK is excellent. We see in figure 4.10 that it retransmits the good packets without any unnecessary retransmission. The congestion window only drops to one-half the value it was before the retransmissions and then it performs congestion avoidance (see figure 4.11).

4.2.5 Throughput comparison

For the throughput comparison the packets were lost at time 30s. As expected the performances of Reno in this case go far beyond Tahoe and SACK. SACK is better than Tahoe because it avoids slow start after the retransmission. Moreover SACK avoids unnecessary retransmission.
Figure 4.10: Congestion window of the SACK sender with five packet losses in the same window

Figure 4.11: SACK retransmission with five packet losses in the same window
Figure 4.12: Throughput comparison in case of five packet losses in the same window
Chapter 5

Future work

Concerning the model itself, the option field problem should be solved. For now, we use different packet formats according to the option we want to use. Moreover, the length of the fields used for the option is set and cannot be changed. The consequence is that, in the SACK mode, the segments always carry a three block size option field even if it is not required and in further enhancements of the model mixing options will be hard.

A real model of a satellite link should be designed for further more complex simulations. It should be integrated in a terrestrial network.

Other algorithms and options in order to enhance TCP should be implemented. In particular the timestamp option should be implemented with SACK over a satellite link (with variable RTT). The Vegas congestion control, SNACK option, window scaling option..., should also be modeled, which could lead to a model of SCPS.
Bibliography

Appendix A

Code overview

Multi TCP is an extension of the Opnet standard TCP model, here is a list of what I added, where and what it is supposed to do.

Data structure and list:

- scoreboard: defined in the state variable block, this list is used to store all the unacknowledged packet and tag the SACKed ones.

- TcpT.Block: defined in the multi.tcp.h file, this data structure is used to represent a SACK block.

- TcpT.Src_Scb: defined in the header block this structure is used to store packets in the scoreboard.

- TcpT.Block_Record: defined in the header block, this structure is used by the receiver to make a list of the SACK blocks to be sent.

- TcpT.SACK_field: defined in the header block, this is the structure carried by the SACK option field of the TCP segment (SACK_tcp_seg).

- The new global variables are defined in the state variable block.

New functions:

- block_init(): takes an array of SACK blocks in argument and initialize it with empty blocks (0,0).

- tcp_dup_ack_check(): checks if the ack received is a duplicate ack (returns 1 if so, 0 else). Called in the tcp_ack_check() function.

- tcp_ack_recover_comp(): during recovery, checks if ack received is a recover ACK. If so returns 1, else returns 0 and eventually updates the pipe variable or the congestion window. Called in tcp_ack_check().

- tcp_scb_ack_update(): update the scoreboard according to the received ack.
• tcp.sack.sack_update(): takes the received segment in argument, extracts the SACK option and updates the scoreboard. Called in tcp.seg.receive().

• tcp.sack.opt_set(): takes the segment to be sent in argument and sets the SACK option inside. Called in tcp.seg.send().

• tcp.block.update(): takes in argument the sequence the last out of order segment received and an integer set to one if this is a new out of order segment, else it is set to 0. Creates a list of the blocks of contiguous data received and extracts the blocks to be sent. Called in tcp.seg.send.

• tcp.SACK.dup.ack.process(): process the duplicate ack in the SACK mode (schedules fast retransmit and recovery).

• tcp.reno.dup.ack.process(): process the duplicate ack in the Reno mode (schedules fast retransmit and deals with fast recovery).

• tcp.tahoe.dup.ack.process(): process the duplicate ack in the Tahoe mode (schedules fast retransmit and Tahoe recovery).

• tcp.SACK.fast.retrans(): fast retransmit in SACK mode. Scheduled by tcp.SACK.dup.ack.process().

• tcp.reno.fast.retrans(): fast retransmit in Reno mode. Scheduled by tcp.reno.dup.ack.process().

• tcp.Tahoe.fast.retrans(): fast retransmit in Tahoe mode. Scheduled by tcp.tahoe.dup.ack.process().

• tcp.fast.recovery(): fast recovery in SACK mode. Scheduled by tcp.SACK.dup.ack.process().

• tcp.Tahoe.recovery(): recovery in Tahoe mode. Manages packet retransmission according to the ack received and slow starting. Scheduled by tcp.tahoe.dup.ack.process().

Changes in the former code:

• Whenever data is sent, it is stored in the scoreboard.

• In the listen state and the syn sent state, check for the SACK permitted option when a SYN is received in SACK mode.

• In tcp.seg.send() set the options in SACK mode.

• In tcp.timeout.retrans(): in case of timeout, stop fast retransmit and fast recovery.
Appendix B

Work overview

The first part of the internships consisted mainly in reading papers about TCP and TCP enhancements for satellites. I also learned the basics of Opnet 3.0. Then, I had access to the hybrid network model built by Ahmed Gaid. Unfortunately it was in Opnet 2.5 version, which is quite different. Moreover when I got the model it was not running. Therefore I had to make it run in Opnet 2.5 (most of the problems were due to wrong libraries and a few lines of code). Then I could convert it to 3.0 version and add a few things: a host on the same LAN as the server, to compare with the hybrid host, a buffer with finite capacity at the hybrid host to model the modem buffer. This also helped me to develop my Opnet skills.

After that I could start to make my own models: I had to develop new TCP models: Tahoe, Reno and SACK TCP. I used as a base the Opnet standard TCP model, so, first, I had to make sure to understand perfectly this model (and it is a huge amount of code!). Then, when I knew exactly its behavior, I could write the code to enhance it.

At first I made three separate models and I tested them in the appropriate network mode to make sure they have the appropriate behavior. Then, for an easier use, I integrated these three models in one: "multi-tcp" in which you can set the functionality you like. I also tried to make my enhancements as modular as possible for easier future integrations.

Making my own model developed my Opnet skills that I tried to share with the other people working on this project. This way, I could follow the building of other models as RED and Window Scaling Option.

I have learned a lot of interesting things during this internship, especially Opnet. I also improved a lot my knowledge of TCP. The background of the project was also very interesting: it is a very current subject.