Bachelor Research Assignment

# TCP synchronisation effect in TCP New Reno and TCP Hybla

Author: Bert Baesjou
Date: 30/08/05
Supervisor: Geert Heijenk
University of Twente
Program: B.Sc.Telematics
Chair: DACS

# Abbreviation list

| | |
|---|---|
| ACK | Acknowledgement |
| ARPAnet | Advanced Research Project Agency-net |
| BSD | Berkeley Software Design |
| CA | Congestion Avoidance |
| DARPA | Defence Advanced Research Project Agency |
| FOSS | Free and Open Source Software |
| FTP | File Transfer Protocol |
| IP | Internet Protocol |
| ISI | Information Sciences Institute |
| MSS | Maximum Segment Size |
| MTU | Maximum Transmission Unit |
| Ns2 | Network Simulator 2 |
| RED | Random Early Drop |
| RFC | Request For Comment |
| RTT | Round Trip Time |
| SACK | Selective Acknowledgement |
| SS | Slow Start |
| Ssthresh | Slow start threshold |
| TCL | Tool Command Language |
| TCP | Transmission Control Protocol |

## Abstract

Due to the nature of the TCP protocol, it is possible a number of flows are synchronising with each other. This means that flows influence each other in such a way that possibly one flow benefits from the synchronisation by claiming a large share of capacity, while an other flow is not able to receive a reasonable portion of the capacity. This problem is most likely to occur in (simple) network simulations and possibly in actual networks with large data transfers.

The synchronisation effect is caused by the relation between the round trip time (RTT) of packets of certain flows and the moment when a new slot comes available at a saturated DropTail queue (from which multiple flows make use of). When a certain RTT is in sync with the time a slot comes available at the queue, the flow having that RTT is able to be the first to fill the available slot at that queue, leaving the other flow with a constantly filled queue leading to starvation of these flow(s).

Both TCP New Reno and TCP Hybla (a TCP variant especially aimed at adjusting RTT behaviour for wireless high RTT connections) were researched and both suffer from the same synchronisation effect. With TCP New Reno the synchronisation effect was mapped, and with TCP Hybla some more extensive tests were done. This because TCP Hybla has some RTT adjusting techniques embedded, which might have led to less synchronisation effect. As it turned out, none of the adjustments were sufficient enough to rule out the synchronisation effect.

A solution to the synchronisation effect could be inserting random packets into the network. This packet insertion makes sure that it is hard for the RTT to be exactly the same every time, thus not being able to come in sync with the queue. An other solution is using other queueing schemes. It seems like the RED queueing scheme is an effective solution due to its more dynamic nature of queueing, creating variations in the RTT and more fairness when dropping packets of flows. RED however is not widely available in actual networks, so this would mainly be a solution for network simulations.

# Table of Contents

# 1 Introduction

For the understanding and prediction of the behaviour of protocols and data streams in networks, simulations are often used. Besides the financial argument that networks can be tested without the need of having all the equipment used in the actual network, one can also alter the parameters of this network very fast. This allows one to run multiple simulations for different setups by using scripts altering the parameters.

Downside of this approach is that a simulation is only a theoretical representation of how the reality could be. Despite all the efforts to create a simulator as close to reality as possible, it is almost impossible to create a precise simulation of reality. While in the simulators protocols are implemented as how they are described in references, in operating systems implementations may have slight variations. Therefore there are always differences between the results of the simulations and the performance of the actual network.

This report is about a set of simulations which gave results other than initially expected. As it turns out in the end, the exactness and precise implementation of the algorithms gave strange but explainable results. The probability of seeing the encountered behaviour in actual networks is low, this is due to unpredictability, complicity and randomness of most networks.

The simulation runs where done using the ns2 [1] network simulator. The behaviour described in this report is specific for TCP and TCP based protocols. Other protocols might show the same behaviour, but this is outside the scope of this report.

## *1.1 Problem formulation*

During simulation runs on TCP Hybla [2], a conceptual TCP variant aimed at improving TCP performance for mobile and wireless devices, inconsistencies were found with expectations. Aim of this report is to solve the main research question: “What is the cause of synchronizing effect in TCP flows and what can be done to prevent this effect to occur?”. This is done by stating a number of objectives:

- State the intentions of using the ns2 simulator
- Describe the expected behaviour and the behaviour found in the simulator
- Provide a plausible explanation of this behaviour
- Do research of simulation behaviour of both TCP New Reno and TCP Hybla

## *1.2 Research approach*

First a basis of theory is provided by papers about TCP [3], TCP Friendly [7] and TCP Hybla [2]. These papers give a theoretical basis of what we

would expect from TCP and TCP Hybla.

Simulations will be done with the ns2 simulator version 2.27 with one network setup, but changing parameters of the nodes and links. The theory will be used as reference to compare the simulation results with what would be expected based on the theory. Given these results there will be an attempt to give an explanation to the differences found between the theory and the simulation results. The tests and comparisons are done for both normal TCP New Reno and TCP Hybla.

## *1.3 Outline of this report*

The first part of the report will start with giving an introduction to ns2 in chapter 2, after which an introduction into TCP and variants of TCP will be given in chapter 3. In chapter 4 the network on which the simulation runs were done will be described. The last part of the report, chapter 5 for TCP and chapter 6 for TCP Hybla, discusses the simulation runs on the network as described in chapter 4, the questions which arose from these simulations and explanation of the simulation results. Finally in chapter 7, an overall conclusion will be given.

# 2 Introduction to ns2

NS2 is a network simulator which is maintained at the Information Sciences Institute (ISI) at the University of Southern California [1]. The ns2 project dates back to 1989 and at a certain point in 1995 DARPA started to support the project. NS2 contains code from various researchers from all over the world, working at corporations like Sun Microsystems and universities. This chapter will give some basic information about ns2.

## *2.1 Source code*

The ns2 project is released under the BSD “as-is” license. Due to the project being under this license, the source code is distributed as well, enabling for the user community to participate in the development of this simulator. Downside of this model is that various parts of the code are written by different people. These people might have written it for a special project, leaving the code abandoned without any documentation when finishing their project. If one would want to know more about such a piece of code the original developer may be long gone, leaving only the community to ask if someone would know anything about that specific part. During this research it happened multiple times that there was no feedback at all from the community.

Like almost any Free/Open Source Software (FOSS) project, there is a lack of documentation. Not only does the source code often miss comments, but also the documentation about the use of the program is lacking information about a lot of parameters and functions. This seems to be a trend in Academic contributed code, nice concepts are delivered with reasonable code, but comprehensive documentation almost always lACK's. Even more than with general FOSS projects.

This lack of documentation in combination with lack of community support often means one has to find out how the program works by opening up the source code. There are also a lot of (undocumented) options one can set, making the simulator highly adjustable ad flexible. But it is almost impossible to do a representative simulation without knowing what all the options exactly do, because this often means that the source code has to be looked into. This is not desirable way of operation for any program.

## *2.2 TCL and C++ usage*

The simulator makes a distinction between two layers of code. One layer is C++ and is the core of the simulator, it has the modules implementing the protocols like TCP and IP, the actual simulator, queueing systems, etcetera This could be called the framework which is the basis of this simulator.

On the other hand a TCL file is used for the more dynamic changing part of the program, namely the configuration file of the actual network. Although it is possible to write complete programs or scripts in this TCL file, the main intent is small scripts an creation of objects from the core program.

There is an interface between the TCL and C++ to exchange values of object parameters. The idea behind this setup is to have a fast (precompiled) C++ core of the simulator and still be able to make and use intelligent configuration files which are parsed and built run-time. Appendix 2 will give an introduction into the basics of the TCL file used in our simulations, the entire TCL file used for the simulations in this report is given in appendix 3. Appendix 1 gives the code used to gain insight in the development of the congestion window of individual flows, which was used to create most of the figures in this report.

## 2.3 Documentation and manuals

For a solid introduction into ns2, the documentation and tutorials available online are a nice place to start. But if one wants to do slightly more complex things than the very simple examples found in the tutorials, the documentation totally lACK's support. Of 90 parameters which could be set for TCP alone (not talking about variant dependant parameters for for example TCP New Reno), only 22 are described in documentation. The variant dependant parameters are not even mentioned in the documentation and may have sometimes, when in luck, a comment in the default configuration file. This default configuration file is the file where all of the parameters are set to default (and often “standard”) values.

There are a few websites trying to fill up some gaps in the documentation, but these are not always up-to-date or complete due to a specific purpose of these websites. These pages can best be found via search engines, because links from the ns2 site are almost not provided. When being new to ns2 it is best to have a contact which has experience in using ns2.

## 2.4 Support and maintenance

A final word about support an maintenance, as indicated previously in this document, this is not top of the bill. How the code exactly is maintained is not entirely clear to the outer world. ISI officially maintains it, but a few errors in some makefiles are in the source code long after people on the mailing list came with patches. ISI did not document for the public that there was a patch for this problem, nor did they do a rebuild of that release with the right makefiles.

From the community itself support is also very low. On an average one could say that maybe one out of the ten questions on the mailing list gets a reply. Answers were mostly given to the simple questions from which the information could also be found in a first-timers tutorial. Questions more in-depth were mostly left un-answered. This might be due to the issue that a researcher or organization developing a certain piece of code for a specific project and afterwards has no team offering support of some kind thus leaving the code for what it is. On the other hand there might be researchers working for a longer time with this simulator and knowing all its strengths and weaknesses, leading to no need for those people to have

documentation or create it. When providing modules to ns2 it also seems like no documentation is required by ISI to be taken into the main distribution, leaving no pressure at all for code writers to provide documentation.

## *2.5 Conclusion*

NS2 is probably a great tool for doing research, especially when this research takes up a longer period of time an many simulations. In such a situation it is worth of digging into the source code an finding out how the internals of the program exactly work. However for smaller research with non sufficient knowledge of ns2, there are just too many options and “got to knows” to be able to do simulations with conclusive results.

# 3 Introduction to TCP

The Transmission Control Protocol was created in the 1970s to make connections across the Advanced Research Projects Agency-net (ARPAnet) and to replace the Network Control Protocol. In 1978 the Internet Protocol (IP) was added to TCP to take over the routing of messages, which resulted in the TCP/IP protocol suite. In 1981 the RFC 793 [3], Transmission Control Protocol, was published.

TCP is a communication protocol which is connection-oriented and has a reliable delivery. The application layer sends a byte-stream to transport layer, where TCP divides the stream into segments. TCP then sends this segments to the Network layer, where IP handles the sending across the network. To keep track of the packets and keep them in order on the other side, TCP gives each packet a sequence number, which upon reception by the receiving TCP module is acknowledged. If no acknowledgement is received by the sending side within a reasonable time, the sender presumes the packet is lost and resends the packet. With the usage of a checksum, which is computed by the sender and included in the header of the TCP-packet, the receiver checks if a packet is damaged by computing the checksum and comparing its own with the one sent along with the packet.

## *3.1 "Standard" TCP operation*

In RFC 2001 [9], four TCP algorithms are described (which at the time the RFC was written were already implemented in most operating systems): slow start, congestion avoidance, fast retransmit, and fast recovery. This paragraph discusses those procedures. This paragraph is called **"Standard" TCP operation** because the procedures discussed here are almost standard and exactly the same for all TCP implementations and variants. Most variants have alterations in specific parts of the "standard" TCP operation, that meet specific needs. An example will be discussed in paragraph **3.3 TCP variant: TCP Hybla**.

### 3.1.1 TCP congestion control procedure

When a TCP connection between two nodes is established, the sender probes for the available capacity of the link used. This is done by increasing the congestion window (W), being the number of packets a TCP flow may have in the network at any time. In the initial slow start (SS) phase this is from an initial window ($W_0$), which commonly is one or two times the maximum segment size (MSS), from which the congestion window is increased by one MSS per received non-duplicate acknowledgement (ACK). At the time W reaches the slow start threshold (ssthresh), the sender switches to the congestion avoidance phase (CA), during which the window is increased by MSS/W bytes per non-duplicate ACK received. This rise continues until the advertised window (the size of the buffer of the receiver) is reached, or if the sender notices segment loss. In the latter case the sender enters an other phase in which a recovery procedure is

started that is TCP version specific.

These rules can be expressed as formulas defining the window update rules, the size of the window at a specific time and the transmission rate [2].

The congestion window update rules can be expressed as a formula (1) when we have an index 'i' which denotes the reception of the i-th ACK.

$$W_{i+1} = \begin{cases} W_i + 1 & SS \\ W_i + \frac{1}{W_i} & CA \end{cases} \quad (1)$$

Rewriting this formula for the time domain gives more insight in the performance of TCP. For the SS phase this results in a discrete exponential increase with RTT, as the congestion window is doubled at every RTT. For the CA phase the growth is one segment per RTT and therefore has a linear increase with time. Denoting $t_\gamma$ the time at which the ssthresh,$\gamma$, is reached. This gives

$$W(t) = \begin{cases} 2^{\frac{t}{RTT}} & 0 \le t < t_\gamma \quad SS \\ \frac{t - t_\gamma}{RTT} + \gamma & t \ge t_\gamma \quad CA \end{cases} \quad (2)$$

where $t_\gamma = RTT \log_2 \gamma$

From (2) it can be seen that the lower the RTT, the higher the congestion window increase rate.

It is also possible to calculate the amount of data transmitted ($T_d(t)$) by a standard TCP source from the moment of transmission initialization. For this formula the expression of the segment transmission rate is given (3) (the amount of segments transmitted per second).

$$B(t) = \frac{W(t)}{RTT} \quad (3)$$

$$T_d(t) = \int_{x=0}^{x=t} B(\tau)\, d(\tau) = \begin{cases} \frac{2^{\frac{t}{RTT}} - 1}{\ln(2)} & 0 \le t < t_\gamma \quad SS \\ \frac{\gamma - 1}{\ln(2)} + \frac{(t - t_\gamma)^2}{2 * RTT^2} + \frac{\gamma * (t - t_\gamma)}{RTT} & t \ge t_\gamma \quad CA \end{cases} \quad (4)$$

With this formula we can create the formula representing the amount of data transmitted from the standard TCP source since the start of the transmission $T_d(t)$ (4). It can be seen from this formula that the amount of transmitted data is heavenly dependant on the RTT. The lower the RTT is, the more data can be sent. This is logical because a lower RTT connection in ideal situation is:

- Increasing the W on each received ACK, a connection with a lower RTT is able to increase the W faster. This means that this connection is able to reach the maximum of the available capacity of the communication channel faster, thus being able to send at full rate earlier than the connection with a higher RTT.
- Able to send more packets overall, even if the congestion windows are even. This because the low RTT connection is able to send the next stream of packets earlier than the high RTT connection. Per second the low RTT connection is able to send more packets in this case.

Note that the formulas above are only valid for ideal communication channels without loss. In other situations the formulas are valid until the first loss occurs in the communication channel.

It must also be noted that there are some different flavours of TCP such as Reno, New Reno, Tahoe, etcetera. The main differences in these versions are mostly located in the loss recovery mechanism. This is discussed in the next paragraph.

### 3.1.2Fast Retransmit and Fast Recovery procedure

In TCP New Reno a duplicate ACK is generated for packet n-1 for each packet arriving when packet n is not yet received. After the reception of three duplicate ACK's, the sender starts the Fast Retransmit and Fast Recovery procedure. First segment n is retransmitted and next the sstresh is updated to half of the the value of the W before the loss was detected. The W is reduced to sstresh plus 3 MSS. Each additional duplicate ACK increments the W by MSS and triggers the transmission of a new segment if the current W exceeds the value of the W before loss was detected. When a non-duplicate ACK is received there can be two consequences depending on whether the ACK is only partial, or the ACK covers an entire window. If the ACK is partial, the W is deflated to the amount of data acknowledged and the recovery phase is not terminated. If the ACK confirms all the packets, the recovery phase ends, the W is deflated to sstresh and the transmission restarts in CA phase.

Other TCP protocols have (partially) other procedures in place, but are not discussed here because they are outside the scope of this report.

## *3.2 TCP Friendly*

An algorithm was developed to provide programs which are not using TCP at the transport layer some “friendliness” in the network. The formula (5) is derived by looking at the “normal” TCP behaviour[7]. This makes this formula a mathematical approximation of how TCP theoretically should perform.

$$Capacity\,share = \frac{1.22 * MTU}{(RTT * \sqrt{(Loss)})} \quad (5)$$

When having packets of size MTU (Maximum Transfer Unit) and a particular RTT for a network which is dropping packets when the connection congestion size increases to W packets. Next the TCP would cut the congestion window in half after which the congestion window is increased by one until it reaches W again. This means that in average the congestion window in on ¾$^{th}$ of W. The maximum speed we can send at is W*MTU/RTT. So we define S in bytes/second as the average speed (6).

$$S=\frac{0.75*W*MTU}{RTT} \tag{6}$$

Because when a loss occurs the W is reduced to ½W and the again builds up to a new W, the loss for a connection is 1 over all the possible window sizes from ½W to W (7). This is almost 1 over ((3/8)W$^2$), from which we can derive the W.

$$\begin{aligned} Loss&=\frac{1}{W/2+(W/2+1)+\ldots+W} \\ Loss&\approx\frac{1}{(3/8)\,W^2} \\ W&\approx\sqrt{\frac{8}{3*Loss}} \end{aligned} \tag{7}$$

We can fill this W in into (6) which gives us formula (5).

If we consider two flows, with both the same loss and MTU but different RTT, we can calculate the theoretical share of each flow on the total capacity. Because MTU and loss is constant and the same for both, they eventually can be eliminated. As can be seen in (8) the share of the $RTT_x$ is $RTT_y$ divided by $RTT_y$ plus $RTT_x$.

$$Capacity\ share\ (\%)\ Flow_x=\frac{\frac{1.22*MTU}{RTT_X*\sqrt{loss}}}{\frac{1.22*MTU}{RTT_X*\sqrt{loss}}+\frac{1.22*MTU}{RTT_Y*\sqrt{loss}}}*100 \tag{8}$$

*Which can be rewritten as:*

$$RTT_x\%=\frac{\frac{1}{RTT_x}}{\frac{1}{RTT_x}+\frac{1}{RTT_y}}*100=\frac{RTT_y}{RTT_y+RTT_x}*100$$

To plot with this formula we take a $RTT_y$ fixed to 10 units of time, while increasing the $RTT_x$ from 1 to 100 units of time (the exact unit is not important because a ratio is calculated). Figure 1 shows the plot of TCP friendly and thus the theoretical performance of two TCP flows on a link versus the ratio between $RTT_x$ and $RTT_y$.

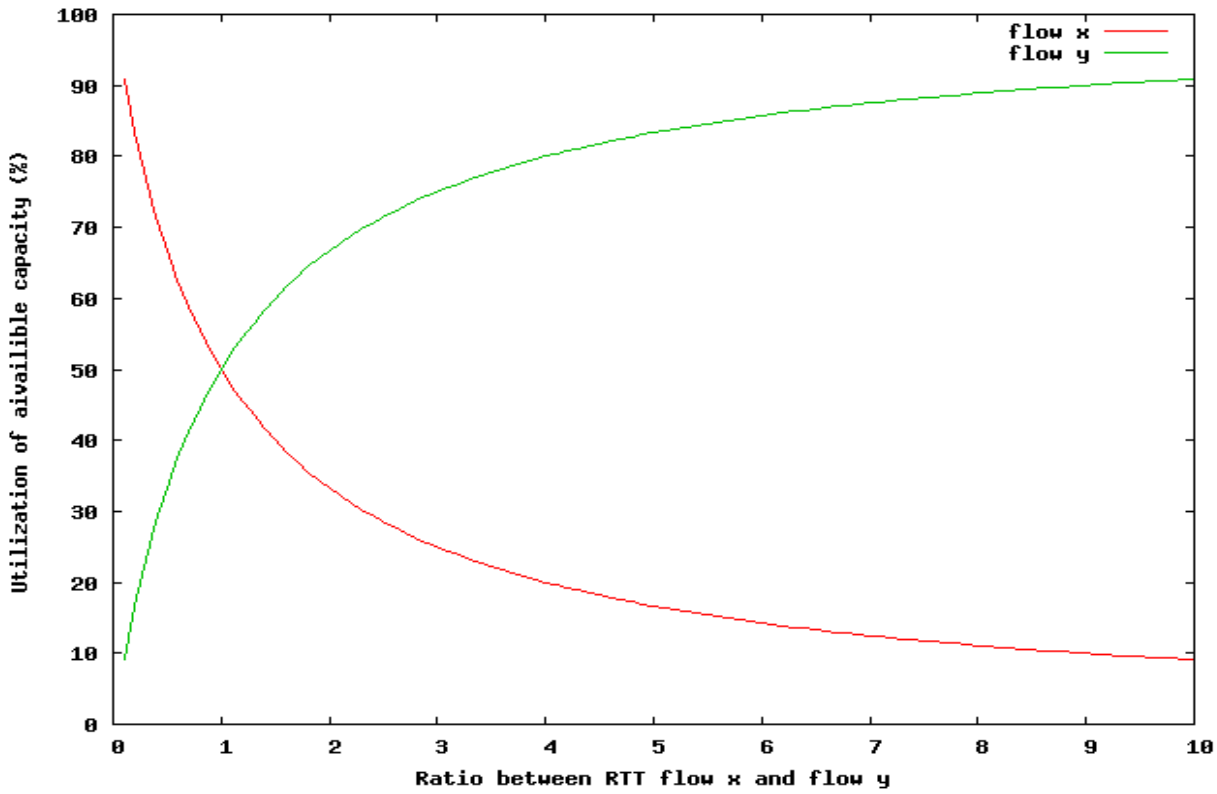


Figure 1, Theoretical performance of two TCP flows.

As can be seen in the figure that even if the RTT of flow x is ten times bigger, it still gets a reasonable amount of capacity and does not starve out.

## *3.3 TCP variant: TCP Hybla*

In this report there is some special attention for TCP Hybla, this is due to the original research assignment which was focused on researching the behaviour of TCP Hybla. This chapter describes the operation of this altered TCP protocol, based on the paper: " TCP Hybla: a TCP enhancement for heterogeneous networks" [2].

### 3.3.1 The objective of TCP Hybla

As the title of the paper indicates, the objective of TCP Hybla is to enhance the TCP behaviour in heterogeneous networks, mainly focused on preventing connections having a high latency being "punished" by this high latency. In figure 1 it can be seen that when the RTT of normal TCP links becomes higher, their capacity share on the network goes down.

This mainly has to do with the time it takes the node to increase W. On a high latency link competing with a low latency link, the low latency link builds up the W much faster after collisions, thus getting a larger share of the overall capacity. When looking at the number of packets successfully delivered at the receiving node, we are talking about goodput. Even in the case when the W of the high latency link and the W of the low latency link are exactly the same, the goodput of the high latency link is still lower. This

because you can have only W packets out in the network during a RTT, the shorter the RTT, the more packets you send per second which in almost always leads to a better goodput.

Objective of TCP Hybla is to provide a way where two channels, with the same available connection capacity but different RTT's, are able to have the same capacity share. We saw that the goodput is heavily dependant on the RTT, TCP Hybla aims at filtering out the RTT an introducing a parameter for regulating the W.

### 3.3.2 Operation of TCP Hybla

In paragraph **3.1** we saw how TCP basically operates. The idea of TCP Hybla is to make the congestion window development independent of the RTT, this is done by using a normalized RTT (ρ). This ρ is calculated by dividing the RTT by a reference RTT ($RTT_0$)(9).

$$\rho = \frac{RTT}{RTT_0} \tag{9}$$

The $RTT_0$ is the round trip time of a connection (the reference link) with a lower RTT than the current connection (the TCP Hybla link). It is desired in TCP Hybla to be able to compete with this low RTT connection goodput wise.

To equalize the window growth with the reference link, the RTT variant part of the equation in (2) is multiplied by the ρ. This results in an equation where the congestion window growth is determined by the $RTT_0$, giving the connection the ability to grow its $W^H$ just as fast as the reference link. Next this equation is multiplied by ρ to compensate for the fact that the window can only update each RTT and not $RTT_0$. Applying these rules result in the formula given in (10).

$$W^H(t) = \begin{cases} \rho\, 2^{\rho\frac{t}{RTT}} = \rho\, 2^{\frac{t}{RTT_0}} & 0 \le t < t_\gamma \qquad SS \\ \rho\left[\rho\dfrac{t-t_\gamma}{RTT} + \gamma\right] = \rho\left[\dfrac{t-t_\gamma}{RTT_0} + \gamma\right] & t \ge t_\gamma \qquad CA \end{cases} \tag{10}$$

Now an $W^H(t)$ is achieved without any dependence on the RTT, when rewriting this formula to congestion window update rules, the “normal” TCP congestion window update rules (1) are replaced by (11).

$$W_{i+1}^H = \begin{cases} W_i^H + 2^\rho - 1 & SS \\ W_i^H + \dfrac{\rho^2}{W_i^H} & CA \end{cases} \tag{11}$$

Just as with normal TCP the transmission rate can be expressed by dividing the $W^H(t)$ over the RTT, applying (10) to this gives (12) which shows that the transmission rate has become independent of the RTT. Instead the $RTT_0$

determines the transmission rate, which means the theoretical transmission rate of the TCP Hybla link has become even with the reference link. Theoretical because these formulas, just like the “normal” TCP formulas, only apply on ideal communication channels without loss. In other situations the formulas are valid until the first loss occurs in the communication channel, which we will see later on in **3.3.3**, is an issue with TCP Hybla.

$$B^H(t)=\frac{W^H(t)}{RTT}=\begin{cases}\dfrac{2^{\frac{1}{RTT_0}}}{RTT_O} & 0\le t<t_\gamma \quad SS \\ \dfrac{1}{RTT_0}\left[\dfrac{t-t_\gamma}{RTT_0}+\gamma\right] & t\ge t_\gamma \quad CA\end{cases} \tag{12}$$

$$T_d(t)=\int_{x=0}^{x=t} B(\tau)\,d(\tau)\begin{cases}\dfrac{2^{\frac{t}{RTT_0}}-1}{\ln(2)} & 0\le t<t_\gamma \quad SS \\ \dfrac{\gamma-1}{\ln(2)}+\dfrac{(t-t_\gamma)^2}{2*RTT_0^{\,2}}+\dfrac{\gamma*(t-t_\gamma)}{RTT_0} & t\ge t_\gamma \quad CA\end{cases} \tag{13}$$

When we now fill (12) into $T_d(t)$ to get the number of segments sent since the start of transmission, it can be seen in (13) that the RTT is no longer present en only the $RTT_0$ remains. Thus giving the same performance as the competing TCP flow.

Finally, the initial value of the initial congestion window and the slow start threshold must all be multiplied by $\rho$ .

**Loss recovery**

To deal better with loss recovery on lines with a longer RTT, TCP Hybla adopted the selective ACK (SACK) option [RFC 2018]. With the SACK option the sender sends an ACK for each packet received, allowing the sender to recover more than one packet per RTT. This option is currently widely spread and available in most TCP implementations and is also often turned on by default in the most operating systems.

## 3.3.3 Known issues with TCP Hybla

TCP Hybla is mostly a theoretically protocol, thus needing more testing and having some known downsides. The most common problems with TCP Hybla are described below.

- Choosing of the right $RTT_0$, when this variable is wrongly determined it could jeapordize the fairness of capacity sharing. The TCP Hybla connection could be far more aggressive in taking capacity. Especially when there are multiple connections with multiple round trip times, it is almost impossible to find a $RTT_0$ which is not aggressive against any of the the flows.

- Calculating the current RTT is also an issue. If the network congests at a certain point, the RTT goes up meaning that the ρ also becomes higher. This leads to a more aggressive TCP Hybla connection, congesting the network even more. The effective goodput goes of TCP Hybla degrades instead of improving it (which was the initial objective of TCP Hybla).
- Due to the larger congestion windows adopted by TCP Hybla with a relative long RTT, there might be a chance that the network becomes congested when suddenly the TCP Hybla connection sends an entire window (bursting) after receiving an ACK. After this sending there might be a time where there is nothing to send until the next ACK is received. A possible solution for this problem could be spreading the entire window over the RTT.

### 3.3.4 Current status of TCP Hybla

As creator of TCP Hybla. the University of Bologna is mainly working on TCP Hybla. They have made a Linux 2.6 kernel implementation [4] and have submitted papers for numerous conferences.

TCP Hybla is also available as ns2 module, which was made by Assed Jehangir at the University of Twente. This module allows one to test the properties and performance of TCP Hybla.

# 4 Setting up a simple network

This chapter describes in detail the network simulated along with the configuration file used. As one will see in this chapter, the network is a very simple one. This is done to make problem analysis as simple as possible. The network for the original assignment in researching TCP Hybla consisted of far more nodes, but the essence (and results) was the same.

## *4.1 Network layout and objectives*

The first basics of the network is to create a bottleneck, forcing data streams to adapt to a congesting network. We also want to be able to have competing flows from different hosts. The most simple set-up with these requirements is the network stated in figure 2. The first two hosts (node 0 and 1) are the sending nodes, these connect to the first router (node 2). This router connects to a second router (node 3) via a relatively very low capacity link, after which the second router connects to receiving hosts (node 4 and 5).

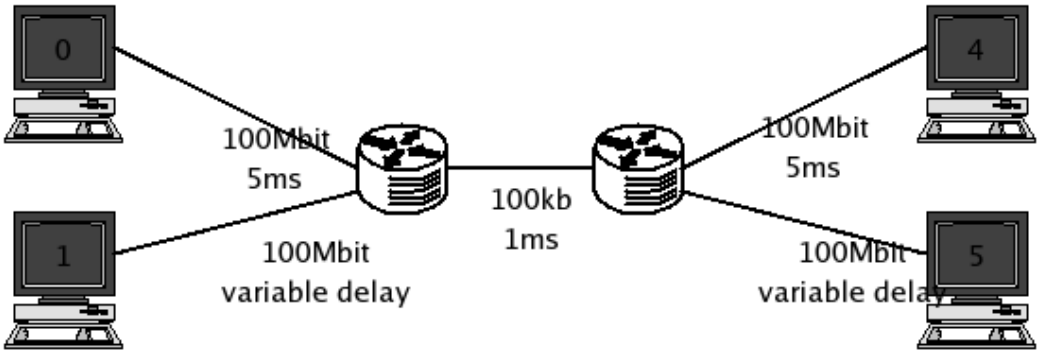


Figure 2, Layout of the simulation network

By giving the links from the senders to the first router a 100 megabit speed and the link from the first to the second router a capacity of 100 kilobit a bottleneck is created.

Delay is also introduced in the network, which is an important parameter for our simulations because the delay at the links has a large impact on the RTT. The RTT is the time it takes the packets to get processed in the network, meaning the total sum of delays on links and processing times at the different nodes. To investigate the behaviour of data streams in the network with different delays, one path (via which a data stream is routed) is chosen with a fixed delay in the links, namely the path from node 0 to node 4, while the other path from node 1 to 5 has a delay which is set to a specific value at the start of each simulation run. During the simulation the variable delay does not change, for example a simulation is done where the variable delay is set to 20 ms giving an overall delay in the path from node 1 to node 5 of 41 ms (20 ms + 1 ms + 20 ms). This value of 20 ms does not change during the entire simulation run. Next a simulation can be done for

21 ms of delay after which the performance of those two simulations can be compared.

## *4.2 Configuration of the network/node parameters*

This paragraph will give an overview of the parameters of the nodes and the network elements, as set in the simulations. The parameters are set in the TCL configuration file given in appendix 3. An other note is that TCP has many (almost 90) parameters which can manually be set in ns2. All parameters have some default values initialized when no value is given in the TCL file. Some parameters set the specifics of generic TCP behaviour such as segment size and advertised window, while others set what kind of TCP implementation should be simulated. Only the important TCP parameters for the simulations of this network are given in table 1, assuming the default values of the other (sometimes undocumented) parameters to be reasonable for these simulations. The values given here are used in all the simulations done with ns2.

The queues at the gateway nodes operate as DropTail queues, first come, first served. When the queue is full, the next packet to arrive is dropped until a new slot is vacant. Queue sizes were all chosen with a length of 10 packets.

All connections used in the simulations use the TCP SACK option, but no significant differences in performance were seen in simulation results of runs with and without SACK.

| ***NS2 parameter*** | ***Value*** | ***Description*** |
|---|---|---|
| WindowOption_ | 1 | This setting defines the type of window update rules of the TCP connection. From 8 different available implementations in ns2, this one does exactly as defined in paragraph **3.1**. |
| ttl_ | 64 | Time To Live set to 64 ms which is default value for most (UNIX) systems. |
| use-scheduler | Calendar {default} | Type of scheduler for ns2, Calendar is default scheduler |
| window_ | 20 {default} | Maximum advertised window |
| packetSize_ | 1000 {default} | Size in Bytes of the packet |
| tcpip_base_hdr_size_ | 40 {default} | Size of the header of the TCP/IP packet |
| ssthresh_ | 0 {default} | Slow start threshold initially zero, but is set to window_ on startup op the ns2 simulator by default. |
| cwnd_ | 0 {default} | This number is updated by 1 by the simulator, when initiating a connection. |
| p_algo_ | 0,1 or 2 | TCP Hybla specific, sets the algorithm used to calculate the RTT. 0 current, 1 smoothed, 2 minimum. |
| r_rtt_ | Competing RTT | TCP Hybla specific, sets the reference RTT to the RTT of the competing network link. |
| queue-limit | 10 | Number of packets (of any size) which can be held in the queue. |
| duplex-link | - | We used only duplex-links in the simulations |
| DropTail | - | All queues are DropTail |
| Agent/TCPSink/Sack1 | - | All connections are using the SACK option. |

Table 1, important ns2 setting for the simulation network

As said before, the entire TCL file with some more explanation about Tcl can be found in appendices (TCL file in appendix 3 and short explanation of TCL in appendix 2).

# 5 TCP New Reno simulation results and analysis

In previous chapters we have seen the configuration of the network at hand. The objective of this chapter is to provide some output of simulations and to discuss these results. This chapter will start with some results from different simulations, after which the results are discussed more in depth.

## *5.1 Simulation results*

In the starting simulation we have a queue of 10 packets. We have two flows:

- Flow 1, starting at 0 seconds, this uses the link with the variable delay
- Flow 2, starting at 20 seconds, this flow uses the fixed delay link

Flow 2 is started after 20 seconds to ensure that flow 1 is in CA phase.

**Variable delay set to 15 ms:**

When doing simulations with the variable delay set to 15 ms, the flows behave like expected, a reasonable fair distribution of throughput is reached. This can be seen in figure 3 which shows how throughput is fairly distributed. After flow 2 starts after 20 seconds, flow 1 adapts by decreasing its congestion window. Both flows sync because they both have to deal with the queue reaching its limits at the same time.

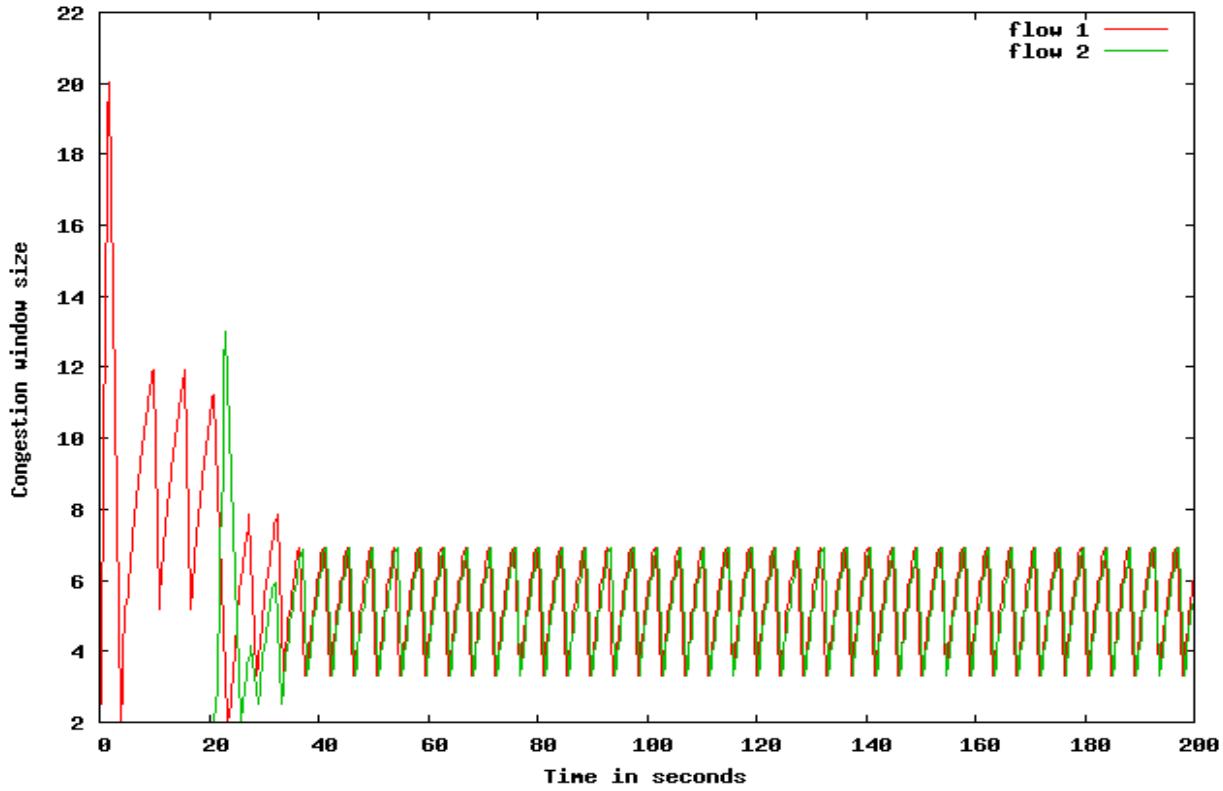


Figure 3, congestion window development for variable delay set to 15 ms

Each simulation also calculated the throughput for each of the flows and within this simulation both flows got around 50% of the available capacity.

This fairness in sharing of capacity is exactly how we would expect TCP to behave in this situation. The observant reader would however argue that the ratio between the delays is about 1:3 (5 ms : 15 ms) and thus one could claim that the appropriate capacity share should be around 75% for the flow with 5 ms delay and 25% for the flow with 15 ms delay (see figure 1). However, the total RTT should be taken into account when comparing ratios, the total RTT includes processing time at queues and nodes. As can be seen further on in paragraph **5.2**, just the queue at node 2 alone takes 830 ms to process. Therefore the total ratio between the actual RTT's becomes close to 1:1 and therefore an equal capacity share is what would be expected.

**Variable delay set to 21 ms:**

When doing simulations with the delay set to 21 ms the output is totally different and seemingly unexpected, as can be seen in figure 4. It can be seen that flow 2 is not able to initiate a normal congestion window behaviour and that flow 1 dominates the link. Remind that flow 1 actually has a larger delay, which is normally a bad thing for flow 1, but still is able to get almost all the throughput on the link.

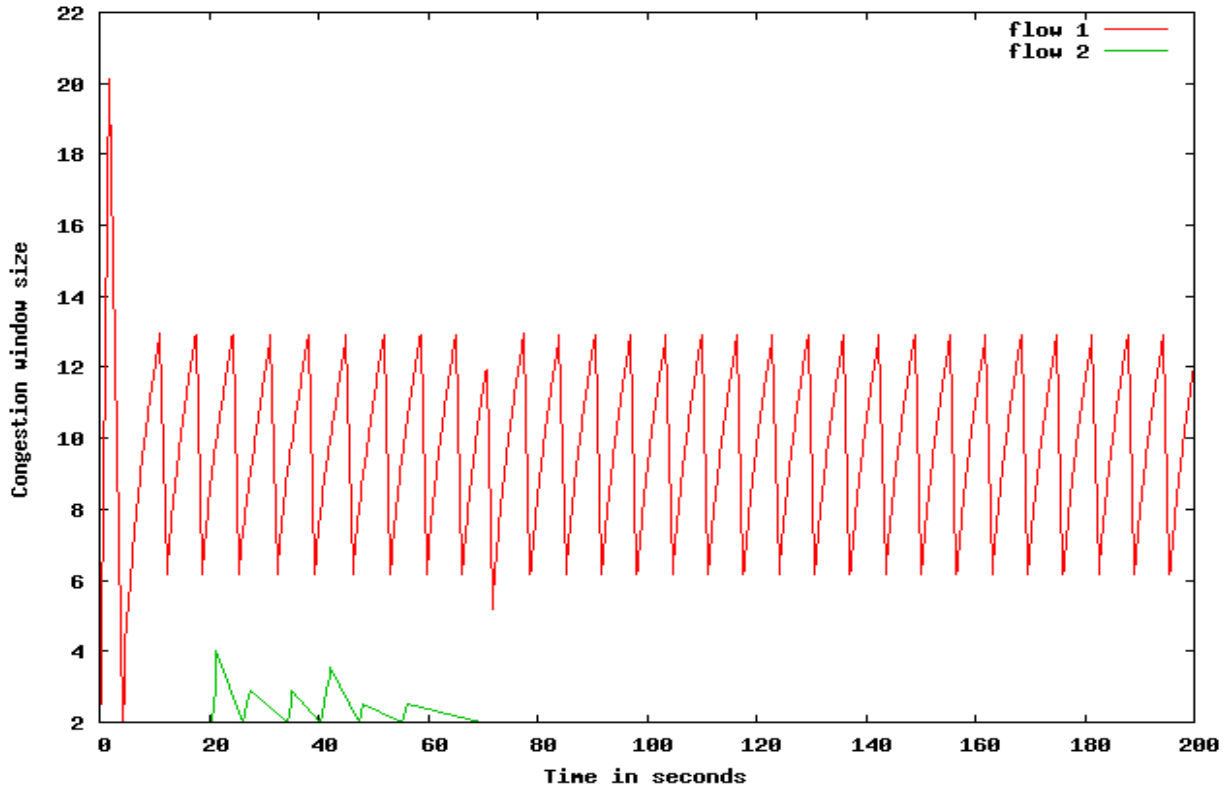


Figure 4, congestion window development for variable delay set to 21 ms

When looking at the goodput data, flow 1 has around 99% goodput, while flow 2 has around 1% goodput. This is not the output what we normally would expect from TCP, since the protocol was designed to distribute the capacity of a link reasonably fair amongst flows.

### Results for multiple variable delays

To have a better understanding of the problem we look into the results of simulations along a wide range of variable delays. Figure 5 describes the utilization of the available capacity (on the bottleneck link) against the variable delay for flow 1.

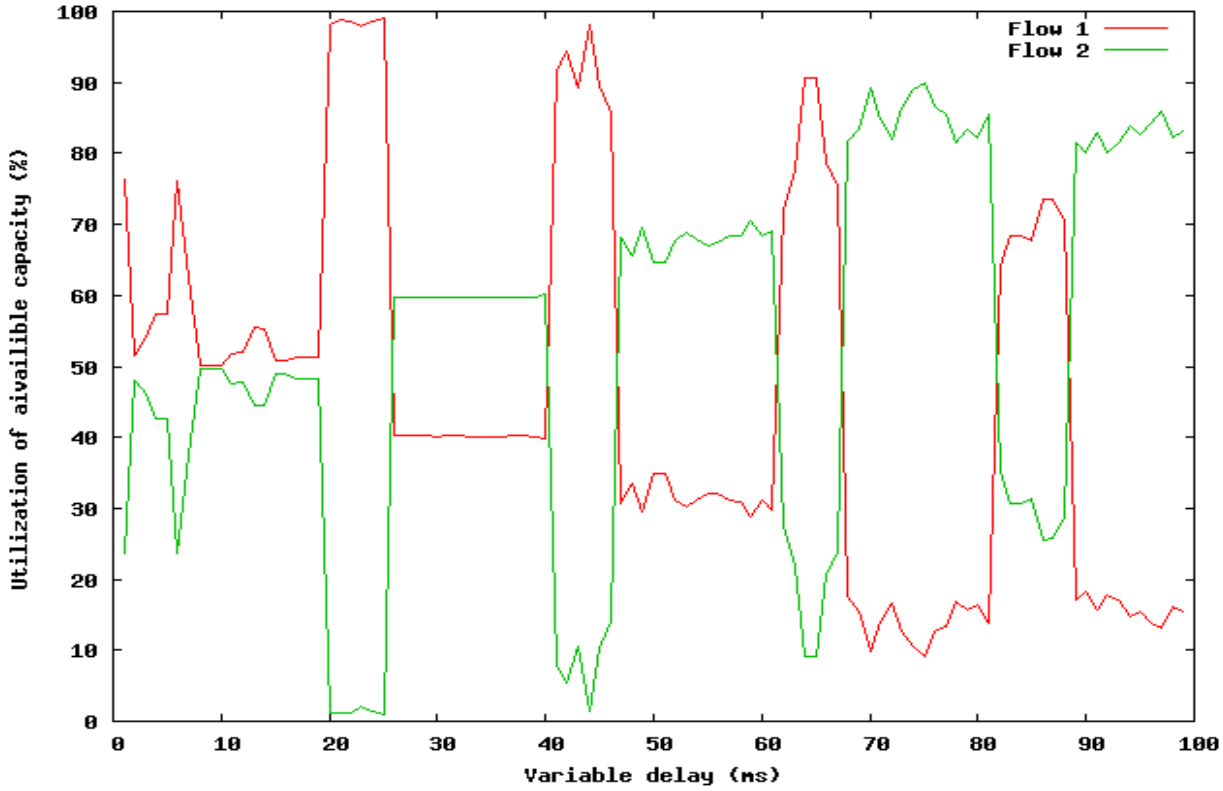


Figure 5, utilization of capacity with a queue of 10 packets

What we see is a repeating pattern. Every 20 ms a period of around 6 or 7 ms the performance of flow 2 drops significantly. This is totally not what was expected on forehand from TCP, the idea behind TCP is to balance the load between flows in a fair way approximating TCP friendly (figure 1).

## *5.2 Result analysis*

When the variable delay is set to a higher value at the start of a simulation run, the overall performance of flow 2 increases. This because the RTT of flow 2 is two times the link latency of 5 ms + 1 ms + 5 ms (two times because the ACK has to be send back trough the same link thus having the same delay), plus the time to wait in the queue at the first router, which is the bottleneck of the network (other queues in the routers have practically no queueing and processing time in the simulation). The queue takes 83 ms to set each packet of 1040 bytes on the 100 kb/s line. With a filled queue of 10 packets this means that when a packets gets in the queue it takes 830 ms before this packet gets on the bottleneck line. Since we send as much data as possible so the queue is most of the time filled, this means the total RTT of flow 2 is always around 852 ms.

Flow 1 however got a variable delay, which is competitive better in the beginning when its RTT is still low, but this delay is clearly affecting the performance when it becomes higher and higher. If we would use TCP Friendly as a sort of reference for how the flows should look like, it would look like figure 6.

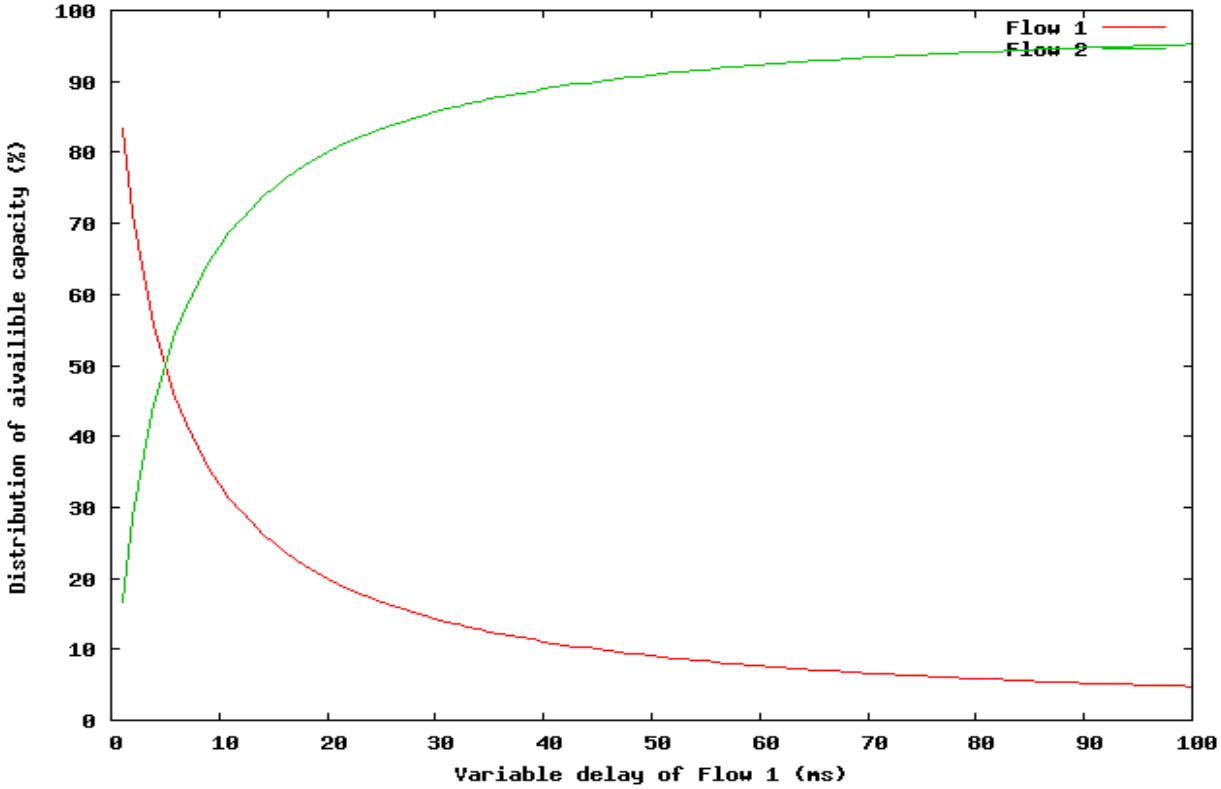


Figure 6, TCP Friendly for simulation network

In this figure we use a fixed RTT for Flow 2 while the RTT for Flow 1 varies. The bandwidth distribution is calculated by setting the MTU and the Loss as a fixed value.

We have seen in formula (8) that the segment size and Loss do not matter for our calculations and is only the RTT part of variance in our figure.

As we can see figure 5 and 6 have in common that Flow 2 eventually gets the overhand and thus Flow 1 decreases. In the basic both pictures look like each other, but the actual performance of the flow is quite different from that of the mathematical reference.

## *5.3 Problem analysis*

The effect we see in figure 5 is described in [8] and is called the Traffic Phase Effect. Basically it comes down to synchronisation in the network.

In networks where we have periodic sources, which is often the case in simulation networks, it could be that window flow control protocols have a periodic cycle equal to the connection RTT. Especially DropTail gateways in

a network with strong periodic traffic can have a systematic discrimination against certain flows. The explanation is as follows: in figure 7 we see a simple network with periodic (bulk) traffic.

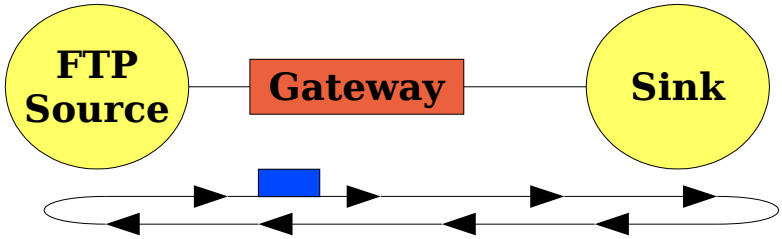


Figure 7, Periodic traffic

For a connection the number of outstanding packets is controlled by the rate ACK's arrive back at the source. When an ACK is received the next packet is immediately sent. This RTT can be called the traffic “period”. In case of bulk traffic, which is a constant stream of packets of the same size, this period is always the same + the time it takes to process queues in gateways.

Queues in gateways decrement by one if the entire packet is sent and increment by one on arrival of the next packet at the queue. Since bulk traffic would always try to optimise the use of capacity, this queue is always filled or has one vacancy. This vacancy is available until the next bulk packet arrives. If we would now have a node trying to send some (telnet) packets (while the bulk connection was sending) it has to arrive after the slot came available but before the next bulk packet arrives, leaving the queue to drop the next bulk packet. If it however reaches the queue to late, the telnet packet would be lost.

In our simulation setup we have two nodes trying to send bulk (FTP) packets. The phenomenon we see in figure 6 between 20 and 27 ms of variable delay can be explained by the Traffic Phase Effect. When we assume the queue to be constantly filled Flow 2 has a fixed $RTT_2$ of 852 ms (14). Flow 1 however has a $RTT_{1\ (variable\ delay)}$ based on the delay chosen between the sending/receiving nodes and the gateways. If we would choose this delay 20 ms, the $RTT_{1\ (20)}$ would be 912 ms (15).

$$RTT_2 = 2*(5\,ms + 1\,ms + 5\,ms) + 830\,ms = 852\,ms \quad (14)$$

$$RTT_{1(20)} = 2*(20\,ms + 1\,ms + 20\,ms) + 830\,ms = 912\,ms \quad (15)$$

When a packet of Flow 1 passed (getting enqueued and finally dequeued to be send) the queue (which took him 830 ms of his $RTT_{1\ (20)}$), it takes 82 ms to: have the packet delivered (1 ms + 20 ms), let the sink send an ACK to the sender (+ 20 ms +1 ms +20 ms) and have Flow 1 delivering the next packet to the queue of the gateway (+ 20 ms = 82 ms). But in the simulator this is not strictly 82 ms but a few milliseconds more due to the fact that it takes (even empty) queues always some time to set a packet on a link. The main factor here is the bottleneck queue at node 3, where it takes the ACK packet 3 ms to be set entirely on the link to node 2. Remember that the bottleneck queue at node 2 takes 83 ms to send one packet, therefore at

this point this node 2 opened up a vacancy just a few milliseconds ago. Flow 2 only has a very small probability to line up in the queue with a slot that is only available for a few milliseconds. Therefore flow 1 “steals” almost all the available capacity in this case.

More generic for our case, when the delay is a little bit longer than the multiple of 83 ms (the time the queue can set one packet on the line), there is a large possibility this flow “wins” in competition for the queue. This because when the new packet arrives at the queue, a packet has just been sent. This multiple of 83 ms is reached each multiple of the 20 ms variable delay (since 2*(20+1+20) = 82 plus a little bit extra delay).

## *5.4 Solution*

A number of solutions are named for preventing this from happening in simulation networks, but they all come down to introducing randomness into the network.

One solution is to include some nodes sending random dummy data into the network. Downside of this dummy data is that it is random and therefore hard to predict what the effects on simulation results are, or how to take the randomness of this dummy data in the appropriate way into account. Taking averages of multiple simulation results could be a solution in such cases.

The other is changing the queue types at the bottleneck link. An in [8] suggested Random Drop policy is not sufficient to decrease the synchronisation effect. But also suggested by [8] is the use of the Random Early Drop (RED) scheme, however further study is needed according to this paper. In the next chapter there will be some promising simulation results for a TCP Hybla connection using a RED scheme.

It is possible (but not very) that this synchronisation takes place in an actual network, when this happens often the previously discussed solutions could be applied. Note that these solutions are intended mainly for simulation purposes, generating for example extra dummy traffic into an actual network to have better performance of this network is probably not something one would pursue.

## *5.5 Conclusion TCP New Reno*

When doing network simulations with any simulator, it is important to check if the results are part of TCP/IP biases. When dealing with larger simulation networks there might also be a smaller chance the synchronisation effect occurs, this because there probably is more randomness introduced in those networks. Some nodes in those networks might provide (automatically without knowing or intention) the same function as the dummy data needed to solve the problem in networks where synchronisation occurs. Downside of larger simulations is that there are more factors to keep track of while analysing simulation results, especially when one is interested in only a specific part of the results.

In practical the chance of this effect ever occurring on real networks is small, because the behaviour of most actual networks is more random. Nodes sending from different TCP implementations for different kind of applications to different parts of the network, create a sort of “natural” randomness. However, especially with large file transfers this behaviour is actually possible in networks.

# 6 TCP Hybla simulation results and analysis

It is suggested in [8] that an adjusted protocol without a dependency on the RTT could be a way to decrease the impact of the synchronisation problem we saw in the previous chapter. Since TCP Hybla is such a protocol, this chapter shows results of simulations done with one TCP New Reno competing with one TCP Hybla flow. It also shows promising simulations with TCP Hybla and TCP New Reno in combination with a RED gateway.

## *6.1 The simulation network*

Basically the network used in these simulations is the same as the network in the previous chapters. Because TCP Hybla is created to enhance the performance of “slow” (high RTT) links, we choose the link with the variable delay as the one using TCP Hybla for its connection. The fixed RTT link just stays using TCP New Reno.

But a small alteration had to be made to ensure the validity of the measurements, this alteration is the capacity of the bottleneck link. Remember that the determining factor in TCP Hybla is the ρ, this ρ is determined by dividing the current RTT by the $RTT_0$. The $RTT_0$ is the RTT of the link which is competed with (in our case the fixed link). At each collection of a new RTT, the ρ is recalculated.

In previous chapters the RTT of the link with variable delay was around the 852 ms (with a queue size of 10 at the router and a variable delay of 20) and the fixed link had a RTT of 830. This would mean that ρ would almost always be around 1 (16). Because a ρ of 1 results in normal TCP New Reno behaviour, it would create no different results as the previous simulations.

$$\rho_{100\,kb/s} = \frac{RTT}{RTT_0} = \frac{912}{852} \approx 1{,}07 \qquad (16)$$

Therefore the capacity of the bottleneck link is set to 1000 kb/s (instead of 100 kb/s). This means that one packet takes 8 ms to get dequeued at the router on the bottleneck link. Theoretically the delay at the queue has a maximum of 80 ms, thus giving a maximum RTT to the variable link of 162 ms and a $RTT_0$ of the fixed link of 102 ms. Now ρ becomes (17).

$$\rho_{100\,kb/s} = \frac{RTT}{RTT_0} = \frac{162}{102} \approx 1{,}59 \qquad (17)$$

These values are much more representative for simulations with TCP Hybla. For the results of the simulations this change has no effect, since synchronisation effects have to do with the moment of a packet leaving the critical queue and the arrival of the next packet from the same sender. The time spent in the buffer of the queue is irrelevant.

## *6.2 Simulation results*

We want TCP Hybla to compete with the fixed link, so we set our reference

$RTT_0$ to the RTT of the fixed link, $RTT_0$ = 22ms + (8ms * queue length). For a queue of 10, this will mean $RTT_0$ would become 102ms.

### 6.2.1 Current RTT TCP Hybla operation

In this first paragraph the ρ is calculated with the last measured RTT. In the next paragraphs it can be seen that there are also other ways to determine ρ, namely smoothed RTT and minimum RTT which are discussed in paragraph **6.2.2** and paragraph **6.2.3**.

Results of simulations with a maximum queue of 10 packets are given in figure 8.

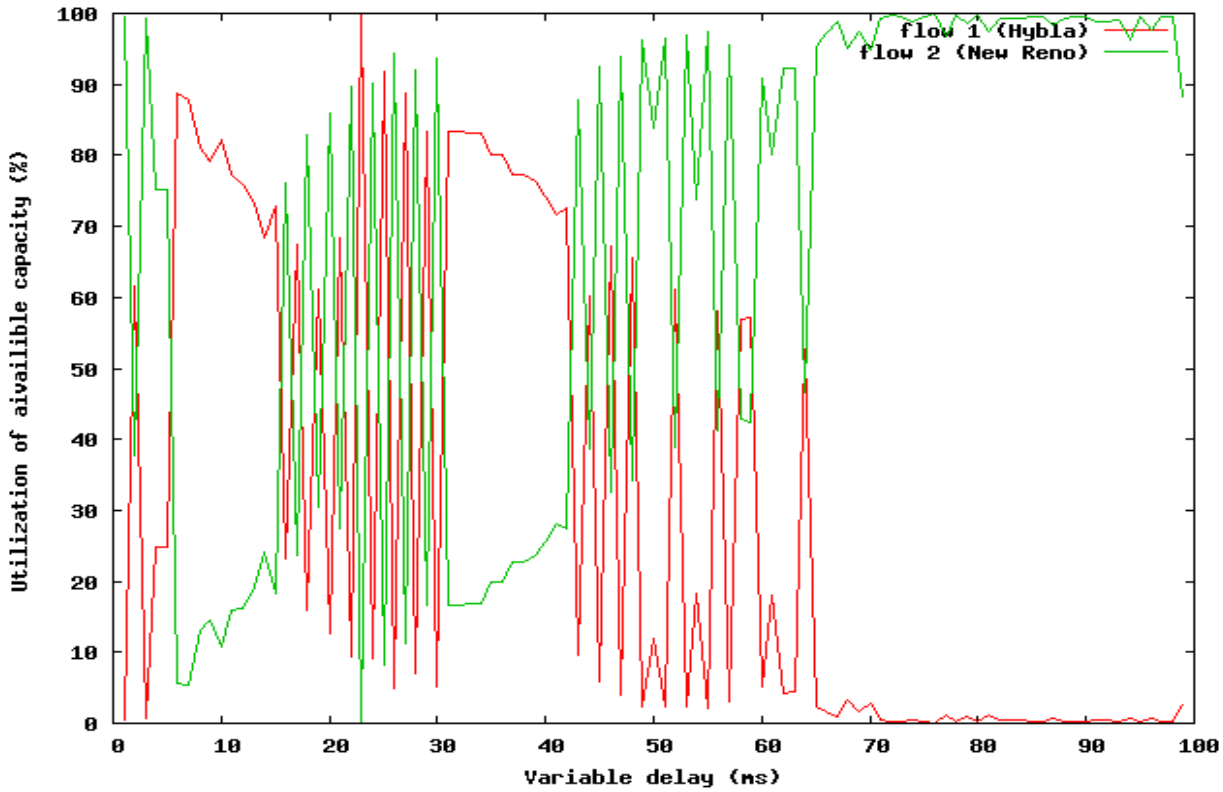


Figure 8, utilization of capacity with a queue of 10 packets

It can be seen that there is still a synchronisation issue, although the figure looks some different. Due to the more variable behaviour of TCP Hybla, the queue is filled in a more random way. Since randomness is a “cure” for the synchronisation problem of flows, synchronisation is slightly less evident in these results. However, there is not enough randomness to ensure that no synchronisation is happening. At for example variable delay is 3 ms and 26 ms, this extremely happens for the Hybla link. For a variable delay of 23 ms this is the case for the New Reno link. Blocks of periods where the synchronisation effect is occurring can be seen clearly between 16 and 30 ms and between 43 and 63 ms. After 65 ms of variable delay, the ρ value grows to large for normal operation.

At for example 70 ms the RTT becomes 262 ms and with a $RTT_0$ of 102 the

ρ becomes 3,5. With such a large value of ρ, the slow start phase becomes very aggressive with an increase of W = 11 (18).

$$W_{increase} = \begin{cases} 2^{\rho} - 1 = 2^{3.5} = 11 & SS \\ \frac{\rho^2}{W_i} = \frac{3.5^2}{W_i} = \frac{12.25}{W_i} & CA \end{cases} \quad (18)$$

So in the first window, eleven packets are sent flooding the queue immediately, leading to multiple packet drops and thus multiple time-outs on packets. The flow is unable to start up normal behaviour.

With an increasing queue size, the behaviour becomes better, but synchronisation can still be observed. This all has to do with the fact that the queue length itself has no influence on the synchronisation effect.

As conclusion we can say that the synchronisation affects the results of the simulation is such a way that a conclusive statement about TCP Hybla cannot be made except for the part that a very large ρ cripples TCP Hybla at a certain point.

### 6.2.2 Smoothed RTT TCP Hybla operation

In smoothed operation the smoothed RTT is used. This is a RTT partly based on the current, and partly on previous RTTs. The idea is to not let the TCP Hybla algorithm overreact on temporarily variations of the RTT. This formula is given in (19) in which α usually has the value of 0.125.

$$RTT_{smoothed} = (1 - \alpha) * RTT_{previous\,smoothed} + \alpha * RTT_{currently\,measured} \quad (19)$$

It can be seen that in case of a spike in the $RTT_{currently\ measured}$, only one eighth of this measurement is taken into the $RTT_{smoothed}$ while the previously smoothed RTT is taken into account for most of the part. In this way the RTT increases only significantly if multiple subsequent high RTT's are measured. The idea behind using $RTT_{smoothed}$ in TCP Hybla is not to let TCP Hybla over react to possible minor changes in the network. In case of a single high RTT, normal TCP Hybla would create a high ρ and a (larger) increase of the congestion window. This overreacted increase could overload the network, leading to a real problem. $RTT_{smoothed}$ should prevent this problem.

In figure 9 the simulation results can be observed.

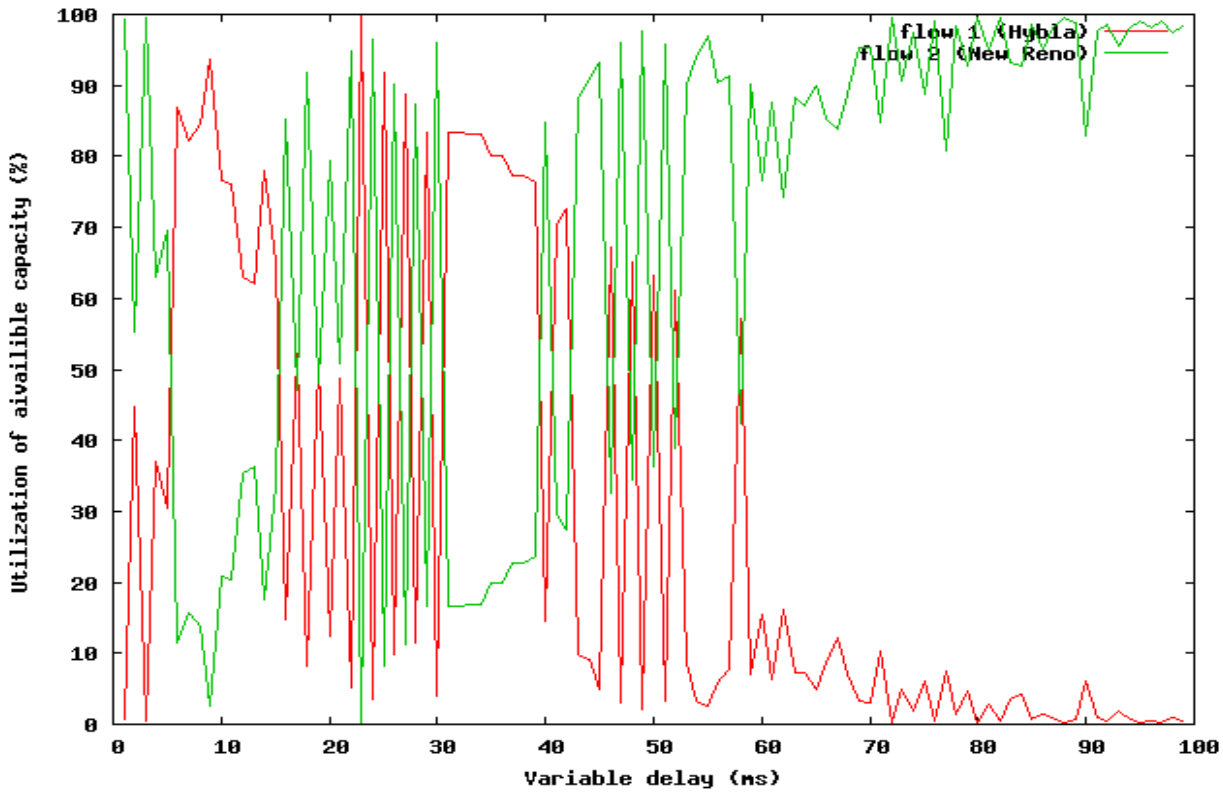


Figure 9, utilization of capacity with a queue of 10 packets

It can be seen, that the graph of the performance is a little bit smoother than TCP Hybla using the current RTT (and thus a little bit better performance), but the synchronisation effect is still very much present. When the queue size grows to 30 packets the overall performance gets better, but synchronisation is still present.

### 6.2.3 Minimum RTT TCP Hybla operation

Minimum RTT is a principle where the minimum value of the RTT ever measured in the connection is taken as the overall RTT. If a lower value would occur this would be taken as the RTT. There are however some downsides to this principle.

- In TCP the first packet of a connection is to set up the connection and is just 40 Bytes. This RTT is therefore always smaller than that of any normal 1040 Bytes packet, links are just faster in processing such a packet. It could also occur that in routing schemes where there are queues for different packet sizes, this packet gets in a queue for the small packets and is processed in favour of the larger packets.
- In our simulation set-up the TCP Hybla flow starts sending without any other flow running. This means the first packet finds an empty queue, giving it a very low RTT which is not representative for the RTT at the moment when the queue is filled.
- In general, a quiet moment in the network would set the minimum RTT to

a low value, which does not represent the RTT when a network becomes saturated. This would lead to a too aggressive TCP Hybla operation.

Therefore TCP Hybla using minimum RTT for ρ calculation is discarded.

### 6.2.4 Current RTT TCP Hybla operation with RED gateway

In this paragraph we consider TCP Hybla with the use of a RED gateway at the bottleneck link. According to [8] this should be promising.

Basically a RED gateway computes an average size of a queue using a weighted exponential running average. When a certain threshold is exceeded by the queue size, a random packet is chosen and dropped. Next the threshold is increased, after which the threshold slowly decreases to the previous value. The chosen packet **n** is determined by the interval 1 to **range**. The $\mathbf{n}^{th}$ packet to arrive at the queue is now dropped. The **range** is set by the congestion rate, at high congestion the value of **range** is low while in low congestion this value is high. This way the drop probability of a certain flow is proportional to the average share of packets on this queue.

The simulation results are showed in figure 10.

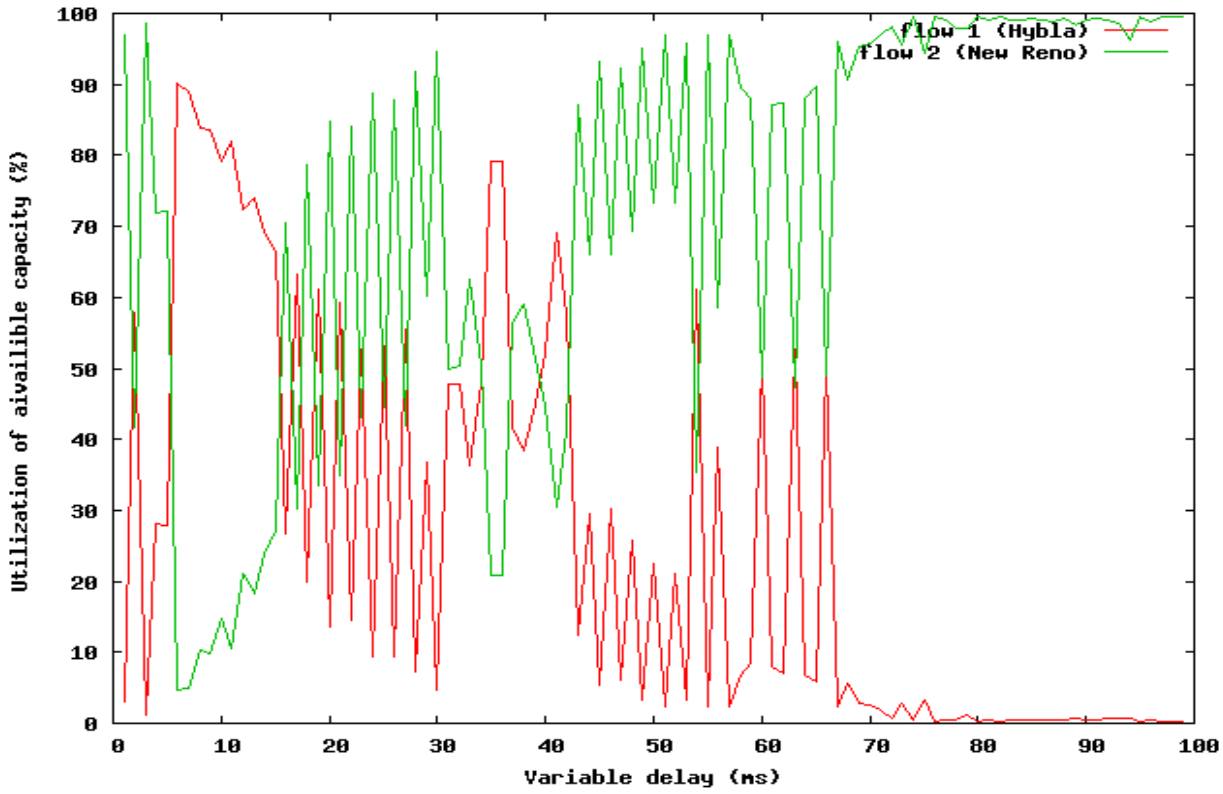


Figure 10, utilization of capacity with a queue of 10 packets

As can be seen in this figure, the synchronisation is still an issue, but a little less than in the initial simulations.

But when applying a queue size of 20 it seems like there is no synchronisation effect. Results of simulations are more like how one would expect from the theory. Results can be seen in figure 11.

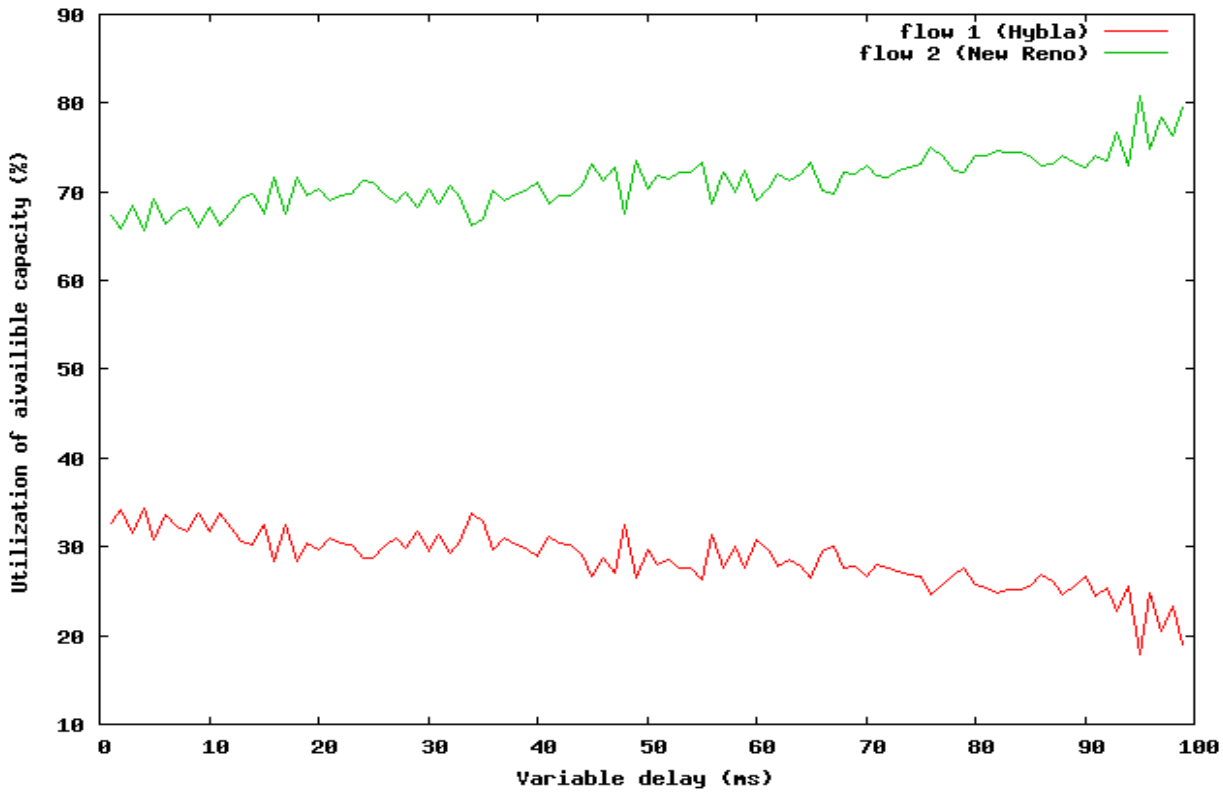


Figure 11, utilization of capacity with a queue of 20 packets

In some more test runs between queue sizes of 10 and 20 packets, it seems like when the queue size becomes 13 or larger, there is no more synchronisation effect. Probably RED does not work very well for smaller queues due to the fact that the flexibility of RED depends on the queue size.

But the utilization of capacity is still not evenly distributed as what the theory would let us expect. This is due to the bursty nature of TCP Hybla, the entire window is sent at once, creating an overflow at the queue. This makes the queue dropping multiple packets from the window. Thus spreading the sending of the packets in one window among the entire RTT, could increase the performance of TCP Hybla to the a more evenly share of the network capacity.

The effect of Hybla getting too aggressive when the $\rho$ becomes to large, also is far less noticeable. This is because RED is better able to handle bursty traffic in larger queue sizes.

## *6.3 Conclusion TCP Hybla*

As conclusion for TCP Hybla it can be said that non of the standard options of the TCP Hybla protocol helps against the synchronisation effect. This is also due to the nature of this effect. The effect has to do with the absence of randomness in the network resulting in a static RTT between the departure of a packet at a queue and the arrival of the next at the same queue.

When bringing some randomness in the gateway by introducing the RED queueing scheme, results become more satisfying. RED has as downside that it is not widely spread in the real world. Therefore for simulations where a real world scenario have to be simulated, using RED is not an option.

# 7 Conclusion

In this paper we saw a synchronisation effect occurring in the TCP protocol. The theory behind the effect is simple, in our situation two flows compete with each other for a place in a saturated queue. While competing, one of the flows has such a round trip time that most packets of that flow arrive just after a spot opens up in the queue. In this way, the other flow is not able to get packets into the queue, which leads to starvation of this flow.

The synchronisation effect seen is however not likely to occur in read life networks, this due to more random traffic on actual networks and implementation differences between operating systems. However in very homogeneous network with large data flows, this behaviour could occur.

In simulation environments however, the synchronisation effect could occur quite easily. One should make sure that readings from simulations say something about the behaviour of the issue at hand in stead of how a (TCP) protocol behaves in a certain environment. Our simulations were severely influenced by the synchronisation effect and therefore were not able to make a contribution to the original research, which was gaining more insight into TCP Hybla.

This report shows that there is almost no difference in performance of TCP New Reno and TCP Hybla. This due to the fact that the synchronisation effect mainly has to do with variation in RTT's and the randomness of the network. Some improvements can be registered for using TCP Hybla in combination with smoothing of the RTT, but these improvements are mainly making TCP Hybla less aggressive and thus improving performance.

There are a number of ways described in [8] to prevent synchronisation, including introducing randomness in the network by introducing nodes, sending random data or changing the type of queues from DropTail to other such as Random Drop or Random Early Detection gateways. We have seen in simulations that RED gateways with a queue size large enough rules out the synchronisation effect, however RED gateways are not very common in actual deployed networks. This report also only made some assumptions why RED only worked after a certain point, a more detailed research should give more insights into why RED started only to work after a certain point.

There is also a downside to introducing randomness in network simulations, results of simulations might be harder to analyse and reproduce due to the more random behaviour. A possible solution is taking the average of the results of multiple simulations.

Since the synchronisation effect can take place in the most simple networks, a more general approach of how to build decent simulation networks would not be misplaced in the manual of any network simulator (thus also on the website of ns2).

## References

[1] The Network Simulator – ns – 2, Information Sciences Institute, Marina del Rey, 09/19/2004,
http://www.isi.edu/nsnam/ns/
[2] Carlo Caini and Rosario Firrincieli, International journal of satellite communications and networking, Volume 22, Issue 5, pages 547-566, 31 August 2004, TCP Hybla: a TCP enhancement for heterogeneous networks,
http://www3.interscience.wiley.com/cgi-bin/fulltext/109604907/ABSTRACT
[3] Defense Advanced Research Projects Agency, Transmission Control Protocol, Arlington, September 1981,
ftp://ftp.rfc-editor.org/in-notes/rfc793.txt
[4] Daniele Lacamera, Implementazione del Protocollo TCP-Hybla su Kernel Linux, Bologna, 27 July 2004,
http://www.danielinux.net/projects/tesi.pdf
[5] Jae Chung and Mark Claypool, NS by Example,
http://nile.wpi.edu/NS/
[6] Trace file formats,
http://k-lug.org/~griswold/NS2/ns2-trace-formats.html
[7] Jamshid Mahdavi & Sally Floyd, TCP-Friendly Unicast Rate-Based Flow Control, 01/1997,
http://www.psc.edu/networking/papers/tcp_friendly.html
[8] Sally Floyd and Van Jacobson, Traffic Phase Effects in Packet-Switched Gateways, Lawrence Berkeley Laboratory, Berkeley, 26-42,
http://portal.acm.org/citation.cfm?id=122419.122421
[9] Network Working Group, TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, January 1997,
http://www.faqs.org/rfcs/rfc2001.html

## Bibliography

Jochen Schiller, Mobile Communication, second edition, Addison Wesley, Harlow, 2003

## Appendix 1 C++ code adjustments to ns2

Alterations were done in the file: $NSHOMEDIR/ns-2.27/tcp/tcp.cc and $NSHOMEDIR/ns-2.27/tcp/tcp.h , the main class of the TCP part of the simulator. These alterations provided the possibility to watch the development of the congestion window.

The header file tcp.h only added the following lines to the class TcpAgent.

```
        int other_;
        FILE *ffp;
```

If one suspects that the integer other_ has something to do with the parameter other_ in the TCL configuration file, one can be said to be right. These two are bound to each other in the tcp.cc file.
The other file, tcp.cc, has the following additions

```
  bind("other_", &other_);
  delay_bind_init_one("other_");
  if (delay_bind(varName, localName, "other_", &other_ , tracer)) return TCL_OK;
```

Here the other_ parameter from the TCL configuration file is bound to the other_ variable in the C++ class.

```
        FILE *ffp;
        char str1[50] = "Window_growth_flow_";
        char *str2 = new char[10];
        sprintf(str2,"%d",other_);
        strcat(str1,str2);
        if ((ffp = fopen(str1,"a")) == NULL){
                fprintf(stderr,"Cannot open file.\n");
                exit(2);
        }
         fprintf(ffp,"%f %f\n",Scheduler::instance().clock(),(double)cwnd_);
        fclose(ffp);
```

The code above is inserted in the function: void TcpAgent::opencwnd() . This function is called whenever an ACK is received for a not already acknowledged packet. The inserted extra code creates a file called Window_growth_flow_{other_} (each flow gets its own file) and prints the clock time and the size of the congestion window (W).

## Appendix 2 A short introduction to ns2 TCL

This appendix gives a short overview of the most TCL syntax used in the configuration file issued in appendix 3. See [5] for a more elaborate overview of TCL configuration files.

```
set START 0
```

This statement sets the value of variable $START to 0

```
set ns [new Simulator]
```

Creates a new simulator object.

```
set n0 [$ns node]
```

Create a new 'node', which can be a router, host, hub, etcetera in the simulation.

```
$ns duplex-link $n0 $n2 100Mb 5ms DropTail
```

Attaches node $n0 with node $n2 with a duplex link of 100Mbit per second an a delay of 5 milliseconds. The sending queue at $n0 will be a drop tail queue.

```
$ns queue-limit $n2 $n3 12
```

This means that the number of packets in the queue of sender $n2 on the port to $n3 will have a maximum capacity of 12.

```
set tcp [new Agent/TCP/Newreno]
```

Creation of a new TCP New Reno object.

```
$tcp set windowOption_ 1
```

Changes a parameter (windowOption in this case) from the default value (every parameter of an object has a pre-defined default value) to 1.

```
$ns attach-agent $n0 $tcp
```

Attaches a node ($n0) to a certain type of sending protocol used.

```
set sink [new Agent/TCPSink/Sack1]
```

Creates a sink object, this object only receives packets and generates ACK's.

```
$ns attach-agent $4 $sink
```

The sink is attached to a certain node.

```
$ns connect $tcp $sink
```

The source and destination are here coupled to each other. The flow will find its own way trough the network.

```
$ns at 20 “$ftp start”
```

Starts the flow $ftp after 20 seconds from the moment the simulator is started.

```
$ns at $END “finish”
```

Runs a predefined finish procedure in which files are finalized, memories are flushed, etcetera.

```
$ns run
```

Always the final code, if the parser reaches this part it starts running the simulator. If one would issue this code on the first line of the script it could be that not all objects are filled and parameter are set, which could give buggy results.

## Appendix 3 TCL configuration file

```
Class TraceApp -superclass Application
#This application helps us to calculate the goodput

TraceApp instproc init {args} {
	$self instvar bytes_
	$self set bytes_ 0
	eval $self next $args
	puts "in init"
}

TraceApp instproc recv {byte} {
	global ns
	$self instvar bytes_
	#puts "Dbyte added to totals $byte"
	set bytes_ [expr $bytes_ + $byte]
	#puts "RD [$ns now] $bytes_"
	return bytes_
}

TraceApp instproc get_bytes {} {
	$self instvar bytes_
	return $bytes_
}

TraceApp instproc reset {} {
	$self instvar bytes_
	set bytes_ 0
}

set START 0.0
set END 200.0
set RCVST 20.0
set LAT [lindex $argv 0] ;#use values from stdin
set QUEUE [lindex $argv 1]
set RRTT [lindex $argv 2]
#Setting the variables of TCP/Hybla
#Agent/TCP/Hybla set p_ $P ;#used to set p_ to static value
Agent/TCP/Hybla set p_algo_ 0  ;# 0 uses current RTT, 1 uses smoothed RTT, 2 uses
minimum RTT.
Agent/TCP/Hybla set r_rtt_ $RRTT ;#Reference RTT which is the competing RTT

set ns [new Simulator]
$ns use-scheduler Calendar ;# different schedulers Heap, List, Callendar, real time

#Define a 'finish' procedure
proc finish {} {
	global ns file1 file2 file3 file4 ftp ftp1 START END recv_application
recv_application1 RCVST
	$ns flush-trace
	set rate [expr ([$recv_application get_bytes]*8) / (10000*($END - $RCVST))]
	set rate1 [expr ([$recv_application1 get_bytes]*8) / (10000*($END - $RCVST))]
	puts "Goodput link 0 (Hybla): $rate kb/s with totally [$recv_application
```

```
get_bytes] bytes send"
        puts "Goodput link 1 (Reno): $rate1 kb/s with totally [$recv_application1
get_bytes] bytes send"
        set effective [expr (($rate+$rate1)*100)/1000]
        puts $file4 "$rate"
        puts $file4 "$rate1"
        close $file4
        puts "Effective $effective %"
        exit 0
}

#Create six nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
set n5 [$ns node]

#Create links between the nodes
$ns duplex-link $n0 $n2 100Mb 5ms DropTail
$ns duplex-link $n1 $n2 100Mb $LAT DropTail ;#20ms DropTail
$ns duplex-link $n2 $n3 1000kb 1ms DropTail
$ns duplex-link $n3 $n4 100Mb 5ms DropTail
$ns duplex-link $n3 $n5 100Mb $LAT DropTail ;#20ms DropTail

$ns queue-limit $n2 $n3 $QUEUE

#Setup a TCP connection
set tcp [new Agent/TCP/Newreno]
$tcp set windowOption_ 1;# Set the TCP behaviour to most default
$tcp set other_ 1;#for internal use, see appendix 1
$tcp set ttl_ 64 ;# simulate *NIX like environment
$ns attach-agent $n0 $tcp
set sink [new Agent/TCPSink/Sack1] ;#receiving node
$ns attach-agent $n4 $sink

set recv_application1 [new TraceApp] ;# create new Tracing Application
$recv_application1 attach-agent $sink

$ns connect $tcp $sink

#Setup a FTP over TCP connection
set ftp [new Application/FTP] ;#The data we send over the link
$ftp attach-agent $tcp

#Setup a TCP connection
set tcp1 [new Agent/TCP/Hybla] ;#use [new Agent/TCP/Newreno] here for newreno
network
$tcp1 set windowOption_ 1
$tcp1 set ttl_ 64

$tcp1 set bvar_ 0 ;#Same use as other_ variable
```

```
$ns attach-agent $n1 $tcp1
set sink1 [new Agent/TCPSink/Sack1]
$ns attach-agent $n5 $sink1

set recv_application [new TraceApp]
$recv_application attach-agent $sink1

$ns connect $tcp1 $sink1
$tcp1 set fid_ 2

#Setup a FTP over UDP connection
set ftp1 [new Application/FTP]
$ftp1 attach-agent $tcp1

#starting the application
$ns at $RCVST "$recv_application start" ;#start the trace application
$ns at $RCVST "$recv_application1 start"
$ns at 0 "$ftp1 start"
$ns at 20 "$ftp start"
$ns at $END "$ftp stop"
$ns at $END "$ftp1 stop"

$ns at $END "finish"
$ns run
```