Implementation and Validation of TCP Options and Congestion Control Algorithms for ns-3

Maurizio Casoni, Carlo Augusto Grazia, Martin Klapez, Natale Patriciello
Department of Engineering Enzo Ferrari, University of Modena and Reggio Emilia
via Vignolese 905, 41125 Modena, Italy
{maurizio.casoni, carloaugusto.grazia, martin.klapez, natale.patriciello}@unimore.it

ABSTRACT

Currently, the ns-3 network simulator include rather limited TCP functionalities. TCP Options are not supported, and it misses models for widely used congestion control algorithms. Thus, simulations can be inadequate for today's standards and unable to represent what happen inside a broad range of networks, from Gigabit Ethernet to high-delay satellite channels. This paper presents an extension of the ns-3 TCP infrastructure, through the addition of the Window Scaling and the Timestamp Options as well as various models of TCP congestion control algorithms, from the widely used TCP Cubic to algorithms tailored for satellite or high Bandwidth-Delay Product links in general, namely TCP Hybla, Highspeed, Bic and Noordwijk. These additions are useful especially for research in high-speed or high-delay networks, filling the gap between real world and ns-3 TCP. Last but not least, this paper also presents some results regarding the validation of the added models, in order to demonstrate their correctness.

Categories and Subject Descriptors

I.6 [**Simulations and Modeling**]: General, Model Development, Model Validation and Analysis;
C.2.2 [**Computer-Communications Networks**]: Network Protocols, Transport Protocols

General Terms

Implementation, Analysis, Testing, Verification

Keywords

Congestion Control, High Delay, Satellite, TCP

1. INTRODUCTION

In the majority of today's research fields, a large portion of numerical results are obtained through simulations. In computer networking research, simulators can provide valuable insights well before the real network topology or technology is built. Moreover, simulations allow to easily reproduce measurements, which permit researchers to easily evaluate and debug protocols and technologies, compare them against other approaches and make or propose enhancements.

The ns-3 network simulator [1] is used by researchers all around the world to simulate a variety of communication technologies and network topologies. It also can be used in real-time, connected to real systems, thus providing a wide range of opportunities.

Until now, ns-3 has suffered poor TCP performance when used to simulate (or when attached to) Long Fat Networks, i.e. networks with large Bandwidth-Delay Product (BDP) channels. This is due to the absence of TCP Options [2], in particular the Window Scaling and Timestamp option, which were born in 1992 to improve TCP performance over such paths.

Another shortcoming of the ns-3 TCP infrastructure is the lack of widely-used TCP congestion control algorithms: to date, the ns-3 simulator includes the implementation of TCP Tahoe, TCP Reno, TCP NewReno and TCP Westwood. The old ns-2 network simulator, instead, provides tested implementations of additional TCP variants; unfortunately, these were never ported to ns-3.

This paper presents and validates the activies performed during the ns-3 summer project of ESA SOCIS (European Space Agency Summer of Code in Space) 2014. The project proposal consisted in the implementation of the Window Scaling and the Timestamp options into the ns-3 TCP layer, as well as the implementation of the following TCP congestion control algorithms: TCP Hybla, TCP Highspeed, TCP Cubic, TCP Bic and TCP Noordwijk. Various properties of these protocols have been analyzed and reported in this paper, to validate such implementations over different propagation delays and error rates.

The remainder of the paper is organized as follows. Section 2 provides a brief theoretical background as well as a survey of related work on the validation of TCP algorithms in ns-3. Section 3 presents the implementation of the options and algorithms, while Section 4 discusses the simulations results along with the implementation's validation. Finally, in Section 5 we summarize and conclude the work.

2. BACKGROUND AND RELATED WORK

In this section, we firstly summarize the theoretical background of either the implemented TCP congestion control algorithms and the TCP extensions of Window Scaling and Timestamps. Then, we present a survey of related work.

WNS3 2015, May 13 2015, Barcelona, Spain
 ISBN 978-1-4503-3375-7/15/05 ...$15.00
DOI: http://dx.doi.org/10.1145/2756509.2756518.

2.1 Background on TCP

In RFC 793 was defined a TCP window header field of 16 bit. Therefore, the largest representable window is 65535 bytes: this limitation can be a problem over channels with a high bandwidth-delay product, because it limits the exploitable bandwidth in such links. To circumvent this problem, RFC 1323 [2] introduced the window scale extension. It expands the definition of the TCP window to 32 bits, using a scaling factor to map this 32 bit value in the 16 bit window field. This scaling factor is carried around in a new TCP option, Window Scale, that is only set in SYN segments: hence, the window scaling factor is fixed in each direction when a connection is opened.

Another extension introduced in [2] is the Timestamp option. It defines a mechanism that allows every segment (including retransmissions) to be timed at a very low computational cost. This way, Round Trip Time (RTT) measurements and Retransmission TimeOut (RTO) calculations can be really accurate, feature often essential for optimal TCP performance. Timestamps have also other uses, like protection from wrap-around of sequence numbers.

Besides options, researchers have also improved the original congestion control algorithm of TCP. When employing high-delay connections (e.g. satellites), one possibility is to use TCP Hybla [3]. The key idea behind Hybla is to obtain for long RTT connections the same instantaneous transmission rate of a reference TCP connection with lower RTT. With analytical steps, it is shown that this goal can be achieved by modifying the time scale, in order for the throughput to be independent from the RTT. This independence is obtained through the use of a coefficient $\rho = \frac{RTT}{RTT_0}$, where RTT_0 is the reference RTT, i.e. the RTT for which we want equal throughput. This coefficient is used to calculate both the slow start threshold (*ssthresh*) and the congestion window ($cWnd$), when in slow start and in congestion avoidance, respectively.

TCP HighSpeed, defined in [4], is designed for high-capacity channels or, in general, for TCP connections with large congestion windows. Conceptually, with respect to the standard TCP, HighSpeed makes the $cWnd$ grow faster during the probing phases and accelerates the $cWnd$ recovery from losses. This behavior is executed only when the window grows beyond a certain threshold, which allows TCP Highspeed to be friendly with standard TCP in environments with heavy congestion, without introducing new dangers of congestion collapse. Standard TCP congestion control is based on an AIMD (Additive-Increase/Multiplicative-Decrease) algorithm, with an additive increase of one segment per RTT ($w = w + \frac{1}{w}$ per ACK received) and a multiplicative decrease of halving the current $cWnd$ ($w = \frac{1}{2} \cdot w$). At the core of TCP HighSpeed there are two functions, $a(w)$ and $b(w)$, which respectively specify the $cWnd$ growth addendum and the $cWnd$ reduction factor when given an actual $cWnd$ value w. Thus, TCP HighSpeed calculates its new $cWnd$ after a single ACK as $w = w + \frac{a(w)}{w}$ while, in response to a congestion event, $cWnd$ is decreased following $w = (1 - b(w)) \cdot w$. As a theoretical comparison, the standard TCP always uses $a(w) = 1$ and $b(w) = \frac{1}{2}$, regardless of the parameter w. When the $cWnd$ grows beyond Low_Window, it always holds that $a(w) > 1$ and $b(w) < \frac{1}{2}$; the specific values depend on the actual window w [4].

The TCP Bic [5] variant aims at mitigating the RTT unfairness (i.e. the unfairness among flows, with different RTTs, that are competing for the same bottleneck link) that may characterize HighSpeed TCP. In particular, in TCP Bic the congestion control problem is viewed as a search problem. Taking as a starting point the current window value (referred as $cWnd_{min}$) and as a target point the last maximum window value $cWnd_{max}$ (i.e. the $cWnd$ value just before the loss event) a binary search technique can be used to update the $cWnd$ value at the midpoint between the two, directly or using an additive increase strategy if the distance from the current window is too large. This way, assuming a no-loss period, the congestion window logarithmically approaches the $cWnd_{max}$ until the difference between $cWnd_{max}$ and $cWnd$ falls below a preset threshold. After reaching such a value (or the maximum window is unknown, i.e. the binary search does not start at all) the algorithm switches to probing the new maximum window with a "slow start" strategy. If a loss occur in either these phases, the current window (before the loss) can be treated as the new maximum, and the reduced (with a multiplicative decrease factor β) window size can be used as the new minimum.

TCP Cubic [6] is a protocol that enhances the fairness property of Bic while retaining its scalability and stability. The main feature is that the window growth function is defined in real time in order to be independent from the RTT. More specifically, the congestion window of Cubic is determined by the function $C(t - K)^3 + cWnd_{max}$, where C is a scaling factor, t is the elapsed time from the last window reduction, $cWnd_{max}$ is the window size before the last window reduction and $K = \sqrt[3]{\frac{cWnd_{max}\beta}{C}}$, with β as a constant multiplication decrease factor.

TCP Noordwijk [7] is a burst-based TCP variant for satellite links. It aims at overcoming issues of standard congestion control algorithms over high BDP channels and consequently maximizing throughput. Noordwijk replaces the standard "window-based" transmission with a "burst-based" transmission. The transmission of packet bursts is characterized by two variables: burst size, which is the number of segments to send in one shot, and burst transmission interval, which is the time between two consecutive transmissions. Both variables are updated according to ACK-based measurements, i.e. *ACK Dispersion* and *RTT Variation*. It has also some drawbacks: it works under the assumption of a controlled environment with well known characteristics and it does not guarantee a fair behavior with competing flows and the efficient resource sharing in case of short transfers. Also, its queue usage (with the resulting delay) is very large [8].

2.2 Related Work

Many ns-3 algorithms and technologies have been validated over the years. With respect to TCP, two research works are pertinent in our overview. The first is about the implementation and the validation of the TCP Westwood and Westwood+ protocols [9]. The authors provide TCP Westwood performance comparison with existing TCP congestion control algorithms in ns-3 (Tahoe, Reno, and TCP NewReno). Furthermore, the effects of various network characteristics such as error rates, bottleneck bandwidth, and propagation delay are studied using throughput as performance metric for the protocols. However, a comparison with a real-world implementation is missing.

The second work describes the implementation and the

validation of the TCP Cubic protocol [10] with respect to the Linux implementation. In fact, authors take the Linux implementation and do a port in ns-3, but this implementation was not included in the mainline of ns-3. To validate the code, a performance comparison is carried out against the Linux implementation (through Network Simulator Cradle) and against the ns-2 version of TCP Cubic.

3. IMPLEMENTATIONS

This section gives a brief survey of all the implementations carried out during the ESA SOCIS summer program. Then, our considerations on the TCP layer of ns-3 are reported. Unless otherwise stated, we will refer to the congestion window with *cWnd*, and to the slow start threshold with *ssThresh*. These values (and all others that depend on such ones) are reported in segments.

3.1 TCP Options

While starting from an extendible design of the TCP Options, we decided to include only two of them (namely, Window Scaling and Timestamps) and relative tests for serialization and deserialization of these into the TCP header. The implementation of other options, like SACK, would have required more time than the timespan of SOCIS: anyway, such implementation (togheter with the dynamic segment size) would be a nice idea for future code proposals.

A new class has been created inside the internet module, called `TcpOption`. All kind of options will be derived from this one; a list of `TcpOption` is kept inside the class `TcpHeader`, extended to serialize and deserialize the known option types thanks to specialized methods of `TcpOption` subclasses. Unknown options are ignored.

The `TcpSocketBase` class has been updated to actively use the values from options Window Scaling and Timestamp (in particular, Timestamps are used for calculating RTTs, while Window Scaling is used to right-shift or left-shift the advertised and the received window, respectively). By the way, options No-op and End have been introduced and actively used to pad the option field. The MSS option is implemented but, because of the inability to dynamically change the MSS of the connections in the actual TCP codebase, it is not used.

3.2 TCP Hybla

As the algorithm aims to remove the performance dependency from the RTT, three values should be kept as class members, and two of them are updated during the transmission: the *minimum* RTT experienced in the connection (referenced as **m_minRtt**), the *reference* RTT (**m_rRtt**) and the ρ parameter (**m_rho**). In particular, the member **m_rRtt** is an attribute of the class `TcpHybla`, called **RRTT**, to make sure that the user can easily access and change such value. By default, it is set to 25 ms.

The `TcpHybla` class is derived from `TcpNewReno`, and it replaces completely the `NewAck` method. When the connection is initialized, *cWnd* and *ssThresh* are multiplied by **m_rho**. When a new ack is received and the experienced RTT is less than the minimum recorded RTT, **m_rho** should be updated. Then, the calculation of the new increment factor for the *cWnd* is made as follows:

if $cWnd < ssThresh$ **then**
  $INC = 2^{m\_rho} - 1$
else
  $INC = m\_rho^2 / cWnd$
end if

The value INC is then added to a counter. A segment-sized increment is done over the *cWnd*, with a loop that increments the *cWnd* until the counter decreases below 1. If the counter is already below 1, the *cWnd* is left untouched. If in slow start, and the resulting value is greater than the threshold, *cWnd* is clamped down to the threshold.

For the methods `DupAck` and `Retransmit`, we update the *ssThresh* calculation by setting the *ssThresh* as the maximum value between 2 segments and the half of the actual count of bytes in flight. We also disabled the increase of the congestion window by 1 segment for every additional dupack received.

3.3 TCP HighSpeed

The goal of the HighSpeed algorithm is to allow TCP to run with high congestion windows in a context of realistic packet drop rates. This is attained thanks to a modification of the standard AIMD algorithm, as explained in Section 2.1.

As values for both the increase function $a(cWnd)$ and the decrease function $b(cWnd)$, the implementation follows the Appendix B of [4]. In particular, the class `TcpHighspeed` has two protected methods, `TableLookupA` and `TableLookupB` which, given the actual congestion window, return the values of $a(cWnd)$ and $b(cWnd)$, respectively. These values are used in the `NewAck`, `DupAck`, and `Retransmit` methods, where *cWnd* is updated as follows:

When a new ack is received (`NewAck`):

if $cWnd < ssThresh$ **then**
  $INC = 1$
else
  $INC =$ `TableLookupA(`*cWnd*`)` $/ cWnd$
end if
$cWnd += INC$

When a loss is detected (`DupAck` and `Retransmit`):

$DEC = 1 -$ `TableLookupB(`*cWnd*`)`
$cWnd = cWnd \cdot DEC$

It is worth noticing that when the window size is less than the value of Low_Window (which defaults to 38 packets), TCP HighSpeed behaves exactly as TCP NewReno, and therefore:

- `TableLookupA(`*cWnd*`)` $== 1$
- `TableLookupB(`*cWnd*`)` $== 0.5$

The slow start and the retransmit phases, instead, are always equal to those of the TCP NewReno, and thus they have not been changed in the implementation.

3.4 TCP Bic

To maintain the performance of TCP Bic as close as possible with the Linux implementation, and at the same time maintain the friendliness with other TCP flavors, the *cWnd* is increased only after a certain number of ACKs are received, following RFC 6356. After the slow start phase, and

after each new ACK, a value is calculated by the method `Update`. This number (**cnt** in the code) represents the ACK packets that should be received before increasing the *cWnd* by one segment. After a trivial check on the arrived ACKs (represented by **m_cWndCnt** in the code), the *cWnd* can be increased and **m_cWndCnt** can be set to zero, or otherwise **m_cWndCnt** can be increased by one and the *cWnd* can be left untouched.

The binary search on the *cWnd* size space is done by varying the returned **cnt**, depending on the internal state of the class (e.g. the last maximum and the current *cWnd* size). That search can be tuned by varying some parameters, added as attributes to the class `TcpBic`:

- *FastConvergence*, for turning on or off the fast convergence after a detected loss.
- *Beta*, which is β for the multiplicative decrease.
- *MaxIncr*, a maximum amount of segments that could be added to the actual *cWnd* during the binary search (i.e. the additive increase threshold as referred in the BIC paper [5]).
- *LowWnd*, which represents the minimum value for the *cWnd* size space used by the binary search algorithm.
- *SmoothPart*, which is the number of RTTs employed from the algorithm to go from **m_lastMaxCwnd-B** to **m_lastMaxCwnd**. We fixed B (which is the range of binary increase) to 4.

When a loss is detected, the new maximum *cWnd* is saved in the member **m_lastMaxCwnd** and then is reduced by the *Beta* attribute. The *ssThresh* is also updated.

3.5 TCP Cubic

The Cubic code in the class `TcpCubic` is quite similar to that of Bic. The main difference is located in the method `Update`, an edit necessary for satisfying the Cubic window growth, that can be tuned with the C (the Cubic scaling factor) attribute.

Following the Linux implementation, we included in the `TcpCubic` class the Hybrid Slow Start, that effectively prevents the overshooting of slow start while maintaining a full utilization of the network. This new type of slow start can be disabled through the *HyStart* attribute. To tune its functionalities, various other attributes have been added to the `TcpCubic` class. Since they are not essential for the functional description of Cubic, please refer to the source code for a detailed description of HyStart.

3.6 TCP Noordwijk

The Noordwijk implementation differs a lot from the previous ones, because it does not use the classical slow start and congestion avoidance phases. Parameters are the attribute *InitialCwnd*, which is the initial burst dimension, *TxTime*, which is the default transmission timer, and B, which is the congestion threshold. When the connection starts, a timer is set with a firing time equals to *TxTime*. Each time it expires, the burst is transmitted over the wire. When the ACKs are received, timing information are calculated over the entire train of ACKs. Thanks to these informations, the burst size and the transmission timer are updated in the `RateTracking` and `RateAdjustment` methods.

Table 1: Simulations parameters.

parameter	value
simulation time	350 seconds
bottleneck type	p2p
bottleneck bandwidth	10 Mbit/s
bottleneck delay	25, 100 and 350 ms
bottleneck loss	0, 10^{-6}, 10^{-4} and 10^{-3}
initial cWnd	10 packets
initial ssThresh	10 packets
MTU	1500 bytes
MSS	1446 bytes
TCP algorithms	NewReno, Bic, Cubic, Hybla HighSpeed, Noordwijk
data to transmit	unlimited

The entire part on retransmission is avoided, as in the specification the channel is considered loss-free. Even with a loss-free channel, there could be losses caused by the queue along the path, and the protocol should be updated to handle such cases. We added a proof-of-concept management of losses, but that code should be refined.

3.7 Issues Encountered

The ns-3 TCP layer was substantially rewritten in 2011, with the introduction of the abstract class `TcpSocketBase`, which provides the TCP socket basic functions, such as the mechanics of its state machine and the sliding window. It is born to be extensible, and in fact it needs to be extended to work: the first extensions that have been released were two TCP flavors, namely TCP NewReno and the basic TCP without congestion control.

This design leads to code duplication, because it contains a fundamental issue: that a congestion control "*is-a*" TCP (e.g. `TcpNewReno` is-a `TcpSocketBase`). This way, each TCP flavor needs to define its own *cWnd* and *ssThresh*. Moreover, each version should reimplement basic algorithms (like fast retransmission) and, even worse, bugs resolved in one subclass may be still present in other subclasses.

Another issue for a ns-3 user is the doubtful consistency of the `TcpSocketBase` API. For example, the initial congestion window is expressed in packets, while the initial slow start threshold is expressed in bytes; these kind of differences could lead to subtle bugs and misunderstandings in the user-written code.

After the development effort made for the ESA SOCIS, the authors claim that a better way to handle the TCP extensibility regarding different congestion control algorithms would be a "*has*" relationship with `TcpSocketBase` (e.g. `TcpSocketBase` has `TcpNewReno`). Proposing a new design for the TCP layer is beyond the scope of this paper, but some steps to simplify the work of writing new (or porting existing) congestion control algorithms to ns-3 could be:

- Merge the *cWnd* and *ssThresh* creation and usage into `TcpSocketBase`.
- Give congestion control subclasses only what they are created for. In other words, they should only manage the connection parameters (i.e. *cWnd* and *ssThresh*), and not the connection functionalities.

In our experiments we had to disable two major components of the TCP protocol: fast retransmit and delayed

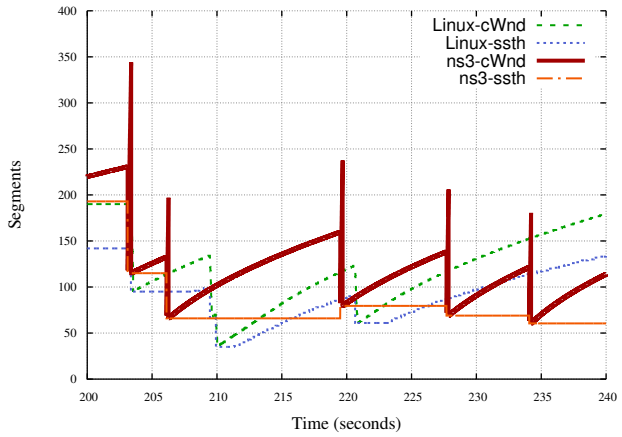

Figure 1: TCP NewReno in ns-3 and Linux, delay 25ms and 10^{-3} packet loss.

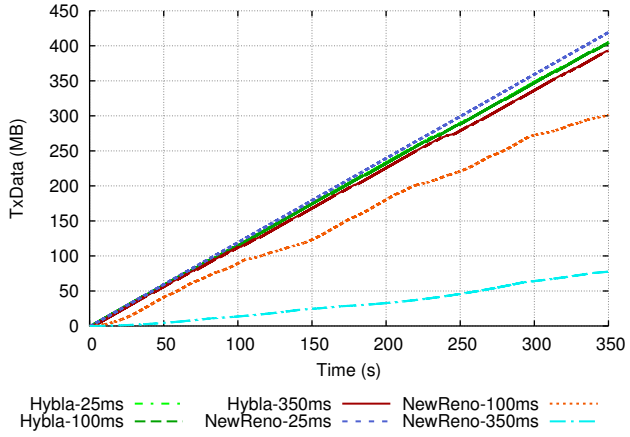

Figure 2: TCP Hybla transmitted data for different RTTs.

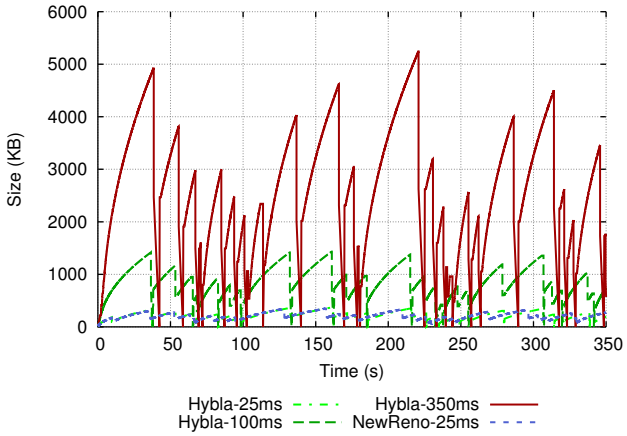

Figure 3: TCP Hybla *cWnd*, for different RTTs.

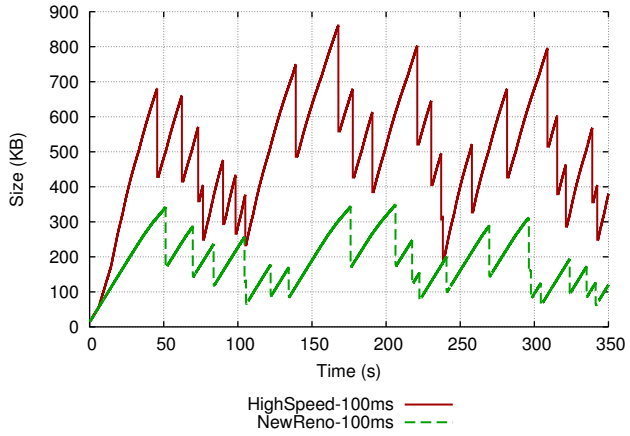

Figure 4: *cWnd*: TCP HighSpeed vs TCP NewReno.

ACK. The first caused major spikes on the *cWnd* for TCP NewReno, but completely disrupted the behavior of TCP Hybla and TCP Highspeed, probably because of their high *cWnd*s. We noted that in the Linux kernel this algorithm seems disabled, as shown in Figure 1, or (more probably) it uses a different approach in carrying out the congestion control by comparing the congestion window with another value rather than with the difference between *SND.NXT* (the sequence number of the next byte of data to be sent) and *SND.UNA* (the sequence number of the first byte of data that has been sent but not yet acknowledged). For what regards delayed ACK, we disabled it (by setting to 1 the attribute *DelAckCount*) in order to clearly see the behavior of the implementations. It is worth to note that the TCP Linux implementation adjusts the timer for the delaying acknowledgements dynamically to estimate the doubled packet inter-arrival time, and it entirely disables the algorithm at the beginning of the connection (in order to speed up the transmission when in the slow start phase). This could be a nice feature to implement on ns-3 in the future.

In general, the TCP layer of ns-3 lacks on tests (functional and RFC-compliance tests), and lacks on comparison with real TCP implementations. So, it is worthless to present a model comparison between real implementations and the ns-3 ones, because it would be difficult to categorize the differencies found (i.e. if a major difference is found, answering the question *is it caused by the model or by the code in* `TcpSocketBase` *?* would not be that easy).

In the next section we introduce our results, based on ns-3 simulations where the behavior of the congestion window and slow start threshold is analyzed with respect to each protocol specification.

4. EVALUATIONS

This section verifies and validates the implementations; our experiments are based on a simple topology, in order to clearly see the behavior of the TCPs and compare it against the theoretical shape.

4.1 Scenario and Parameters

The reference scenario consists of two nodes connected by a point-to-point channel, with a `PacketSinkApplication` on

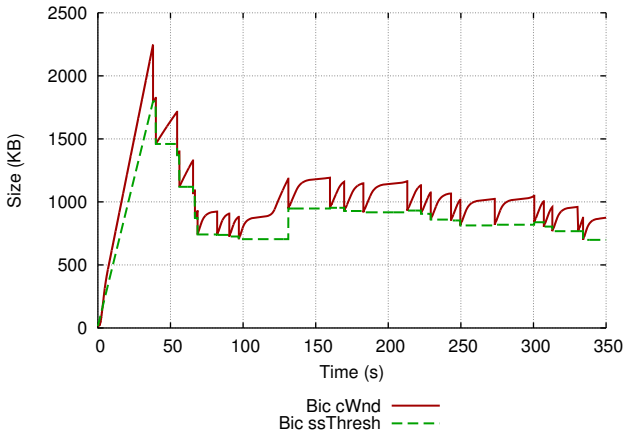

Figure 5: TCP Bic *cWnd* with 100ms delay.

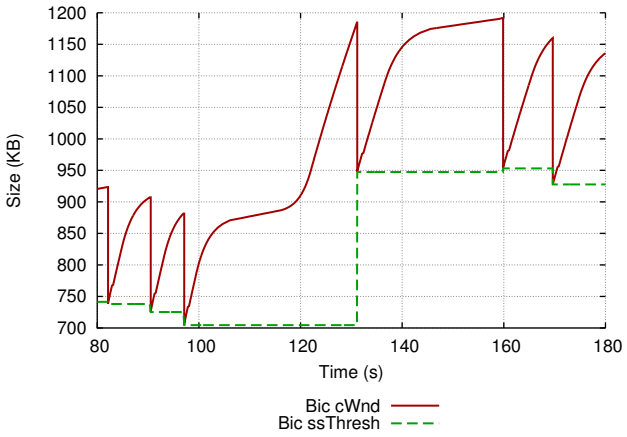

Figure 6: Detail of Bic *cWnd* and *ssThresh*.

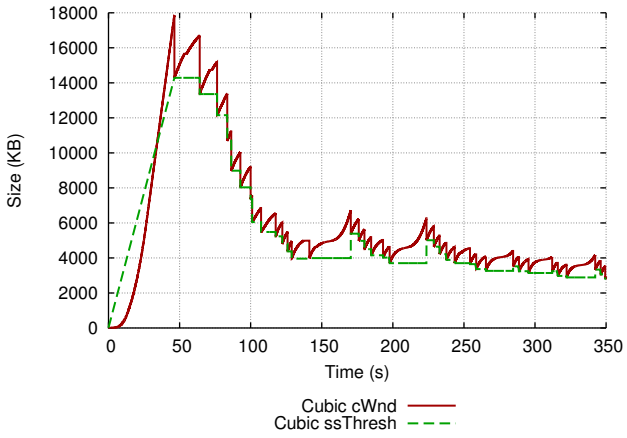

Figure 7: TCP Cubic *cWnd* and *ssThresh* with 350ms delay.

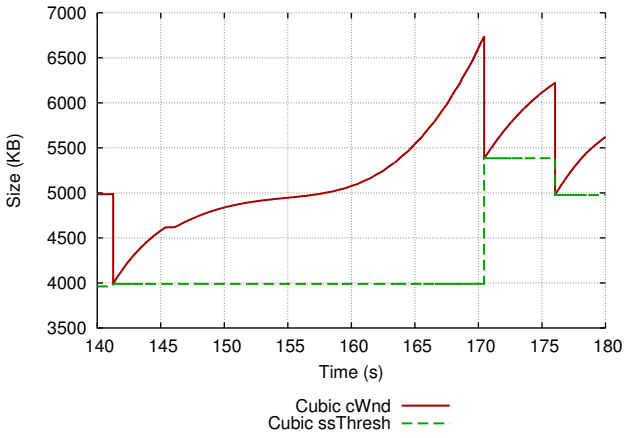

Figure 8: *Detail of Cubic* cWnd *and* ssThresh.

the receiver node and a `BulkSendApplication` on the sender. Even though its simplicity, various parameters need to be considered and explained; these have been summarized in Table 1. First of all, we choose a reasonable experiment duration (350s) with a backlogged sender, in order to see what happens inside the model during each of the key phases, which include slow start, congestion avoidance and recovery from losses. As MTU of the system, we choose a fixed value of 1500 bytes, which leaves for TCP 1446 bytes considering the IP and P2P header length. So, the MSS chosen is exactly 1446 bytes: this avoids fragmentation at the IP layer. The bandwidth of the channel is fixed to 10 Mbit/s in order to have a reasonably fast channel; we are not interested in a comparison between the different TCP versions (i.e. who performs better in a high-bandwidth environment), neither in a comparison with old ns-3 release. In this light, the validation on TCP Options (because of space constraint) is intrinsic to the presented plot, as without Window Scaling the congestion window size would stop at 65535 bytes. The one-way delay of the link and the packet error rate are two variable parameters: for the delay the values are 25 - 100 - 350 ms, while for the packet loss we choose values inside the range going from 0 to 10^{-6}. The rationale behind this choice is to have different delays for different network types (from 25ms for a wired connection to 350ms for a satellite connection). The distribution of such parameters is fixed (i.e. an experiment starts with certain parameters which do not change over the experiment itself). For the initial congestion window and slow start threshold values, both have been set to 10 packets in order to avoid the initial slow start phase (carefully checked but not present in this paper for space constraint) and to focus into the congestion avoidance part, the real core of a TCP congestion control algorithm.

All experiments are conducted with the latest ns-3 release (ns3-3.22) in a Linux environment. Patches and scripts are available in [11].

4.2 Results

Firstly, we report our results for TCP Hybla. We compared its performance against TCP NewReno, as Hybla specifications use NewReno as reference. In Figure 2, we reported the transmitted data of Hybla and NewReno with a delay of 25 ms, 100 ms and 350 ms and a Packet Error Rate (PER) of 10^{-3}. It is possible to notice how NewReno suffers

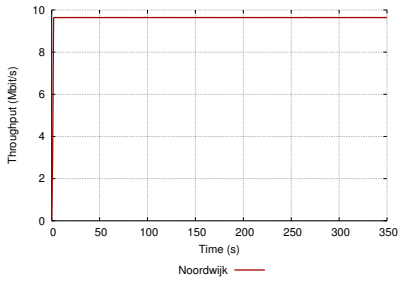

Figure 9: TCP Noordwijk throughput, 1 flow, 100 ms in an ideal channel.

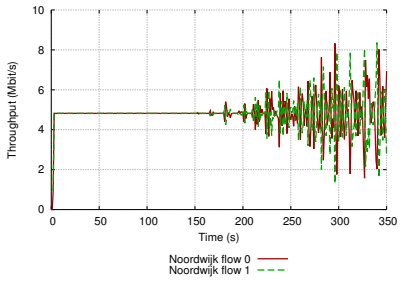

Figure 10: TCP Noordwijk throughput, 2 flow, 350 ms in an ideal channel.

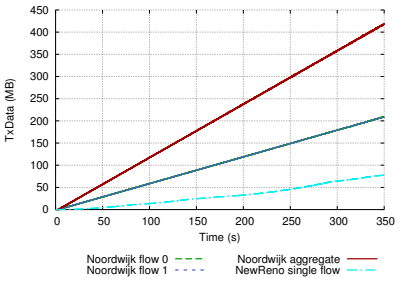

Figure 11: TCP Noordwijk transmitted data, 2 flow, 350 ms vs NewReno.

the increment of the channel delay by a slower growth of its throughput (which can be inferred from the slope of its line) while Hybla do not care at all about the delay increment, achieving performance compliant to those of NewReno at 25 ms. Hybla can achieve performance that are almost RTT independent thanks to its ability in tuning the congestion window during the congestion avoidance phase. For instance, it tunes the $cWnd$ in order to behave, throughput-wise, as NewReno would at a specific RTT value, set to 50 ms in our case. Figure 3 shows exactly this phenomenon by highlighting the differences in the $cWnd$, measured with NewReno (25 ms of delay, i.e. 50 ms of RTT) and Hybla flows at different delays. The chosen PER of 10^{-3} for this experiment permits to appreciate several congestion avoidance phases during the same run and, consequently, to analyze different congestion window growths. As it is possible to notice from Figure 3, Hybla allows the $cWnd$ to grow in proportion with the delay of the NewReno window; in fact, Hybla $cWnd$ at 25 ms of delay is perfectly compliant with the NewReno $cWnd$ at 25 ms, while Hybla $cWnd$ at 100 ms and 350 ms achieve values that are approximately four times and fourteen times higher, respectively, as expected.

To evaluate TCP HighSpeed, we compared the variation of its congestion window to that of TCP NewReno. In this case, we performed the experiment with a delay of 100 ms, the same value that [4] uses as reference. As explained in Section 3.3, we used the coefficients presented in Appendix B of [4]. Therefore, the threshold after which HighSpeed differentiates from NewReno (Low_Window) is equal to 38 packets. The comparison between the two congestion window allows to assess if the simulated TCP Highspeed behaves as it is expected to. Figure 4 shows the results of the simulation in which we introduced a PER of 10^{-3}. As it can be seen, after the Low_Window value is exceeded, the congestion window of TCP HighSpeed grows faster than that of TCP NewReno, as expected. The difference in this growth can be inferred, with some approximation due to the chart limited resolution, from the angle of the respective congestion window growth direction.

With regards to loss event, instead, it is sufficient to pick from the chart any spike and compare its $cWnd$ value with the subsequent lowered one, relating the difference to the respective default $b(w)$ value of TCP HighSpeed. For example, after the 300th second of simulated time, the highest spike represents a congestion window of about 800 KB (794543 bytes); this, divided for the segment size of 1446 bytes, gives 549 segments. From [4], to this value is associated a default $b(w)$ of 0.35, which in turn gives a drop rate of 0.65 (0.50 is the unique drop rate of TCP NewReno). $549 * 0.65$ results in a $cWnd$ reduced to 356 segments, that is equal to 514776 bytes, exactly the value measured from the simulation. This is coherent with the figures presented by Figure 4, where the value of $cWnd$ subsequent to the drop is slightly over the 500 KB line.

Figure 5 presents the TCP Bic $cWnd$ and $ssThresh$, over a channel with 100 ms of one-way delay and a PER of 10^{-3}. There is an initial probing phase, which leads to a big growth of the congestion window, stopped by the first loss event. After each loss, a fast convergence followed by a binary search is done over the $cWnd$. We detailed in Figure 6 what happens around the 100th second of simulated time, where the shape of the $cWnd$ is more clear and resembles the typical BIC function. There is the initial fast convergence, then a binary search until the previous value of $cWnd$ is reached, and after a certain amount of time (determined by the value of the $smoothPart$ attribute, which is set to 5 in this simulation) there is another fast growth of the congestion window.

For what regards TCP Cubic, we show in Figure 7 the $cWnd$ and the $ssThresh$ for a connection with 350 ms of delay (which gives an RTT of 700ms). The connection has a PER of 10^{-3}, which is pretty high. Despite this, the typical Cubic growth of $cWnd$ is clearly visible, as detailed by Figure 8. It is a concave/convex pattern (i.e. concave when approaching $cWnd_{max}$, then convex when exceeding it). At a loss event, Cubic sets the value of $cWnd_{max}$ and reduces $cWnd$ through a multiplication with $Beta$.

To show TCP Noordwijk performance, the only available metric is the connection throughput, due to the different approach to congestion control. There are no congestion avoidance nor slow start phases, and neither a reference implementation to compare it with. For our simulation, we used a default transmission timer of 125 ms and a burst size of 156 packets, each one of 1000 bytes. This gives a default rate of $156 \cdot (\frac{1}{0.125}) \cdot 1000 \cdot 8 = 9984$ kbit/s, which is close to the available channel bandwidth, 10 Mbit/s. The queue size at L2 is set very high: this way, the environment is completely controlled and the Noordwijk should use all the available capacity. In Figure 9 we plot the throughput of one Noordwijk flow in such an environment, and as expected the Noordwijk protocol exploits the entire available bandwidth.

To see what happens in presence of congestion, we simulated a congested environment with two sender nodes, a

Table 2: Simulations recap.

TCP	Validation presented	Result	Open Issues
Hybla	Transmitted data analysis and cWnd comparison	OK	None
HighSpeed	cWnd comparison	OK	None
Bic	cWnd analysis	OK	None
Cubic	cWnd analysis	OK	None
Noordwijk	Transmitted data and throughput analysis	OK	Missing loss management
Window Scaling	Intrinsic validation on *cWnd* size	OK	None
Timestamp	None (for space constraint)	OK	Missing complete RFC 7323 compliance

router, and a sink node. The senders are connected through a p2p link with 100 Mbit/s of bandwidth and 1 ns of delay, while the router is connected to the sink node with a p2p link with the same constrained characteristics as before (10 Mbit/s and the delay in the range 25 ms - 350 ms).

In Figure 10 we report the throughput of the experiment where the bottleneck delay is set to 350 ms. As expected, the two flows share equally the available bandwidth, without any noticeable difference caused by the increased RTT (with respect to the previous experiment with 100 ms of delay). It is worth to note that this result is due to the high queue size; in fact, as we said before, the first burst of both flows is calibrated for a 10 Mbit/s connection. At the beginning, this blind burst means twice as much the data that can be sent over the link. The exceeding data is absorbed by the queue. When the ACKs are received, the Noordwijk protocol adjusts its burst size, thanks to the timing information of the ACKs themselves. With a limited queue size, such behavior would lead to packet losses, which degrade a lot the protocol performance. However, after 150 seconds of simulated time, a little perturbation on a node burst size causes throughput oscillations. Despite this, the flows remain fair in terms of total transmitted data, as it is reported in Figure 11. In the same figure, we plot the total transmitted data for NewReno. As reported in literature, Noordwijk outperforms NewReno in such an environment.

Before conclusions, we summarize the simulations performed in Table 2.

5. CONCLUSIONS

We presented and validated an extension of the ns-3 TCP infrastructure, through the addition of the Window Scaling and the Timestamp Options, as well as through the addition of different congestion controls tailored for satellite or high Bandwidth-Delay Product links in general: Hybla, Highspeed, Cubic, Bic and Noordwijk. We carefully revised the current literature about this algorithms and, through experimental results, we validated their ns-3 implementations by assessing effectiveness and correctness through the comparison of simulation results with the theoretically expected behaviors. Various performance properties of these protocols have also been analyzed over different propagation delays and error rates. As future work, we aim to design and implement the Selective ACK option, mandatory in DVB-RCS systems, that would be extremely useful for a deeper analysis of ns-3 with respect to the Linux implementations, allowing a fair comparison between them.

6. ACKNOWLEDGMENTS

This work was also supported by the European Commission under PPDR- TC, a collaborative project part of the Seventh Framework Programme for research, technological development and demonstration. The authors would like to thank all partners within PPDR-TC for their cooperation and valuable contribution.

7. NOTE

8. REFERENCES

[1] G. Riley and T. Henderson, "The ns-3 network simulator," in *Modeling and Tools for Network Simulation* (K. Wehrle, M. Gunes, and J. Gross, eds.), pp. 15–34, Springer Berlin Heidelberg, 2010.

[2] V. Jacobson, R. Braden, and D. Borman, "TCP Extensions for High Performance." RFC 1323 (Experimental), May 1992.

[3] C. Caini and R. Firrincieli, "TCP Hybla: a TCP enhancement for heterogeneous networks," *International journal of satellite communications and networking*, vol. 22, no. 5, pp. 547–566, 2004.

[4] S. Floyd, "Highspeed TCP for Large Congestion Windows." RFC 3649 (Experimental), December 2003.

[5] L. Xu, K. Harfoush, and I. Rhee, "Binary increase congestion control (BIC) for fast long-distance networks," in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, vol. 4, pp. 2514–2524, IEEE, 2004.

[6] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 64–74, 2008.

[7] C. Roseti, M. Luglio, and F. Zampognaro, "Analysis and Performance Evaluation of a Burst-based TCP for Satellite DVB RCS Links," *IEEE/ACM Trans. Netw.*, vol. 18, pp. 911–921, June 2010.

[8] M. Casoni, C. A. Grazia, M. Klapez, and N. Patriciello, "Reducing Latency in Satellite Emergency Networks through a Cooperative Transmission Control," *IEEE Global Communications Conference (GLOBECOM)*, pp. 2916–2921, December 2014.

[9] S. Gangadhar, T. A. N. Nguyen, G. Umapathi, and J. P. Sterbenz, "TCP Westwood+ protocol implementation in ns-3," in *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, pp. 167–175, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013.

[10] B. Levasseur, M. Claypool, and R. Kinicki, "A TCP CUBIC Implementation in Ns-3," in *Proceedings of the 2014 Workshop on Ns-3*, WNS3 '14, pp. 3:1–3:8, ACM, 2014.

[11] P. N. http://code.nsnam.org/nat/ns-3-dev-socis2014/, 2015.