SICS Technical Report T2001:10 ISRN: SICS-T–2001/10-SE
ISSN: 1100-3154

DTMsim – DTM channel simulation in `ns`

Henrik Abrahamsson and Ian Marsh

Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, Sweden

November, 2001

Abstract

Dynamic Transfer Mode (DTM) is a ring based MAN technology that provides a channel abstraction with a dynamically adjustable capacity. TCP is a reliable end to end transport protocol capable of adjusting its rate. The primary goal of this work is investigate the coupling of dynamically allocating bandwidth to TCP flows with the affect this has on the congestion control mechanism of TCP. In particular we wanted to find scenerios where this scheme does not work, where either all the link capacity is allocated to TCP or congestion collapse occurs and no capacity is allocated to TCP. We have created a simulation environment using ns-2 to investigate TCP over networks which have a variable capacity link. We begin with a single TCP Tahoe flow over a fixed bandwidth link and progressively add more complexity to understand the behaviour of dynamically adjusting link capacity to TCP and vice versa.

1 Introduction

Dynamic synchronous Transfer Mode [7] or DTM is a ring based networking technology which can dynamically adjust it's bandwidth. DTM has a channel abstraction, where a channel consists of a number of slots. The number of slots allocated to a channel determines it's bandwidth. The slots can be allocated statically by pre-configured parameters or dynamically adjusted to suit an applications needs. In DTM it is possible to allocate a channel to a specific TCP connection but also to multiplex several TCP connections over the same channel.

The Transmission Control Protocol [11] offers a reliable end to end byte stream service. TCP was not designed to operate on a network with variable capacity. It uses an end-to-end congestion control mechanism to find the optimal bandwidth at which to send data and does not expect the capacity of the link to change according to it's needs.

In order to get good throughput with TCP operating over a technology such as DTM it is important to understand the dynamic behavior of the two schemes. TCP is capable of adjusting it's *rate* and DTM capable of changing it's *capacity*. When dynamically allocating bandwidth to TCP flows it is important to avoid couplings with the end-to-end congestion control loop of TCP. One must also be careful not to create oscillations and at the same time avoid allocating too much bandwidth to the flows which would lead to underutilization of the network.

This report is the result of the project 'DTMSIM: DTM Channel Simulation in ns' conducted at SICS with funding from Dynarc and Nutek. The purpose of the project is to investigate solutions for dynamic bandwidth distribution between channels. This is done by creating a channel abstraction, that has dynamic bandwidth properties, in the network simulator (ns-2). This makes it possible to simulate control algorithms for dynamic bandwidth allocation.

The report is organized as follows. First, the dynamics of TCP is described in Section 2 followed by related work in Section 3. The simulation environment is described in Section 4 and the simulation results are presented in Section 5. The report is concluded with a discussion in Section 6.

2 TCP dynamics

TCP has 4 congestion control algorithms, slow start, congestion avoidance, fast retransmit and fast recovery which are specified in [1]. Detailed explanation and examples of these algorithms are provided in [12].

The slow start and congestion avoidance algorithms are used by a TCP sender to control the amount of data that is transmitted into the network. The congestion window (*cwnd*) is a sender-side limit on the amount of data the sender can transmit into the network before receiving an acknowledgement or ACK. The receiver advertised window (*rwnd*) is a receiver-side limit on the amount of outstanding data. The receiver advertised window is a measure of the receivers buffer capacity while the congestion window is a measure of the capacity of the network. The sender is prohibited from sending more than the minimum of *rwnd* and *cwnd* unacknowledged data.

Another state variable, the slow-start threshold (ssthresh), is used to determine whether the slow-start or congestion avoidance algorithm is used to control data transmission. Slow-start [1] provides an exponential increase of the congestion window. This rapid increase goes on until the congestion window reaches the slow-start threshold where congestion avoidance takes over and the congestion window is increased more slowly. During slow-start, TCP increments the congestion window by one segment for each ACK received while in congestion avoidance the window is incremented by one segment per round-trip time (RTT). Congestion avoidance continues until congestion is detected.

If congestion and segment loss is detected by using the retransmission timer, the slow-start threshold is set to one half of the congestion window and the congestion window is then set to one. Therefore, after retransmitting the dropped segment the TCP sender uses the slow-start algorithm to increase the window from one segment to the new slow-start threshold.

If instead segment loss is detected by the reception of three or more duplicate ACKs the fast retransmit and fast recovery algorithms are used. The sender then retransmits the lost segment without waiting for the retransmission timer to expire. The congestion window is halved and also the slow-start threshold is set to one half of the congestion window. So, after the fast retransmit congestion avoidance and not slow-start is performed.

It is known that the fast retransmit and fast recovery algorithms generally only can handle the loss of one or very few packets. If many packet losses occur within one round-trip time TCP has to rely on the retransmission timer.

Figure 1 shows how the congestion window changes with time for a TCP Reno connection on a 1 Mbit link simulated in ns. The plot shows how TCP in congestion avoidance always try to get more bandwidth by increasing the congestion window by one packet each round-trip time. When a packet is dropped fast retransmit is used to recover from the loss and the congestion

[1] It is referred to as slow start because it begins with a window size of 1.

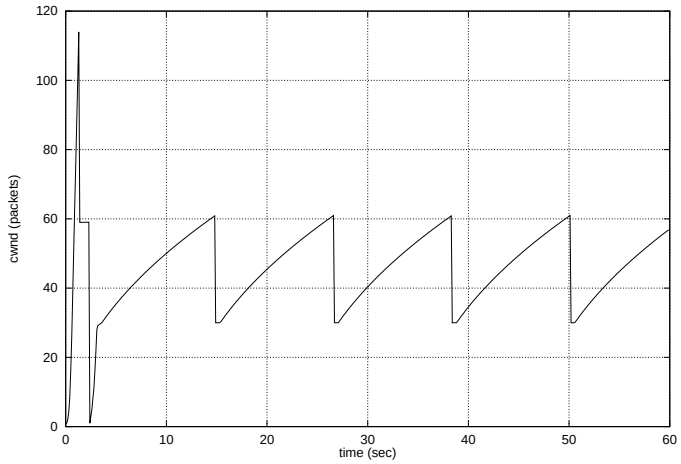

Figure 1: TCP Reno congestion window

window is cut in half. This results in a periodic sawtooth curve.

3 Related work

3.1 Bandwidth allocation schemes and their interaction with TCP

3.1.1 Some open-loop schemes

The PhD thesis *Integration of ATM under TCP/IP to provide services with minimum guaranteed bandwidth* [3] by O. Bonaventure investigates and gives a comprehensive description of each of the ATM service classes with focus on how they manage to support TCP/IP traffic.

The thesis begins by examining the Constant Bit Rate (CBR) service category. This service category is mainly suitable for applications that require a constant amount of bandwidth for the the whole duration of the virtual circuit. When using CBR an application has to specify a traffic contract describing its peak rate (stated as the interarrival time between successive cells). The main conclusion from this thesis is that for CBR the TCP throughput is very sensitive to segment loss. Therefore it is important to generate traffic that is conformant with the traffic contract. If the traffic is not completely conformant, a policing unit discards cell at the ingress of the network, which causes a severe reduction in the performance of TCP hence a low utilization of the link. If the ATM traffic is conformant, TCP behaves correctly and can achieve high link utilization. An additional conclusion is that TCP relies too

much on the retransmission timer, and the high granularity of this timer in most TCP implementations causes large idle times when packet loss occurs.

Most Internet traffic is very bursty and many application don't require a constant amount of bandwidth for their whole duration. It would be a large waste of resources inside the network if these applications had to rely on CBR and specify their peak rate. Instead other service categories such as Variable Bit Rate (VBR) or Guaranteed Frame Rate (GFR) are better suited for bursty traffic. The VBR service category allows an end-system to reserve some average amount of bandwidth inside the network, while still being allowed to transmit at a higher rate during short periods of time. With GFR the end-system can request a minimum guaranteed bandwidth but is able to transmit at a higher rate if the network is not congested.

Similar allocation schemes have also been proposed for IP networks. For instance, in the paper *Explicit Allocation of Best-Effort Packet Delivery Service* [4] D. Clark and W. Fang propose an "allocated-capacity" framework for allocating bandwidth to different users in a controlled and predictable way. The focus of the paper is TCP bulk-data transfers. Feng *et al.* [6] describe an interpretation of the IETF Controlled Load service [13] and investigates its interaction with TCP in the paper *Understanding and Improving TCP Performance Over Networks with Minimum Rate Guarantees.*

The ATM Variable Bit Rate service category (at least according to the conformance definitions VBR.2 and VBR.3), the ATM Guaranteed Frame Rate service category, the "allocated-capacity" framework [4] and the interpretation of the IETF controlled-load service presented in [6] have several things in common. They all rely on traffic contracts, admission control, policing and tagging of packets. Each connection has to specify its bandwidth requirements in a traffic contract. This is used by an admission control algorithm to decide whether enough resources are available in the network to support a new connection. A policing unit is used to control the traffic sent follows the traffic contracts. The four schemes also have in common that they make it possible for end-systems to use additional available bandwidth inside the network. This is done through tagging where packets (or cells) that don't conform to the traffic contract are tagged and treated on a best-effort basis inside the network.

The conclusions of [3], [4], [6] are similar, TCP connections have difficulties to utilize their reservations. A reason for this is the flow and congestion control mechanism used by TCP. The TCP traffic is more bursty than the traffic contracts allow and when the loss of a segment is detected, TCP reduces the congestion window and initiates a fast recovery or a slow start phase. For the fast recovery phase, the congestion window is cut by half whereas in the slow start phase it is set to one. For connections with reser-

vations this decrease of the window size is too drastic since it is insensitive to the reservation that a particular connection may have.

With their allocated-capacity scheme, D. Clark and W. Fang [4] want to guarantee a specified throughput for a TCP stream but they notice that this might be hard to achieve with the rate adjustment mechanisms in TCP, especially if slow-start is triggered. Therefore they attempt to keep the TCP flows in the fast-recovery phase by avoiding dropping several packets of the same flow in the same RTT. This is done by using a RED-like dropping mechanism in the routers as well as using a probabilistic function for tagging packets.

3.1.2 Some closed-loop schemes

The ATM service categories mentioned in the previous section are all open-loop services, which means the network does not provide any explicit feedback to the end systems. In contrast, the Available Bit Rate (ABR) service category is closed-loop. It tries to avoid cell losses inside the network by using a congestion control mechanism which operates in the ATM layer. It modifies the transmission rate of the endsystems in response to feedback received from the network.

Bonaventure summarizes the literature on TCP/IP with the ABR service category. Since ABR provides a loss-free service, the performance of TCP over ABR is mainly driven by ABR and how well the ABR congestion control mechanism works. ABR has the most complex traffic contract of all ATM service categories and the congestion control mechanism makes ABR complex to implement and deploy. The conclusion from [3] is that it is often difficult to select the appropriate values of the parameters in the traffic contract and that the benefits of ABR when carrying TCP/IP traffic is not in proportion to its complexity.

In order to improve the performance of ABR, several researchers have proposed closer interaction between the window-based congestion control mechanism used by TCP and the ABR rate-based mechanism. This could be done by translating the ABR feedback to the TCP sources. For example, by controlling the rate at which the TCP acknowledgements are returned to the TCP source or by modifying the window size in the TCP acknowledgements.

Ideas similar to those proposed for ATM ABR but in a TCP/IP network are used in the *TCP rate control* technique described in [8]. The idea is to calculate and allocate rates for competing TCP flows and then enforce the rates by manipulating TCP header fields and the ACK rate. The real rate allocation algorithm is unfortunately not given in the paper because of commercial reasons (the TCP rate control approach has been implemented

and patented by Packeteer Inc. [10]). The focus in the paper is on how to enforce the allocated rates so for the simulations presented a simpler rate-allocation algorithm was used: Initially, equal rates are given to all competing flows. The sending rates of the flows are then estimated using an exponential average over some rate sampling interval. When a flow does not utilize its allocation it is labeled as a bottlenecked flow. The excess allocation is taken away from all bottlenecked flows and is distributed equally to the non-bottlenecked flows. This step is repeated until there is no residual bandwidth to allocate or all the flows are bottlenecked. Once a fair rate allocation has been obtained the *TCP rate control* approach uses two techniques to enforce the rate. First, using information about the round-trip time the rate is converted into a window value. The TCP receiver window field in the ACK headers is then changed to this new window value. The idea is that, with the receiver window being the limiting factor the negative effects of packet loss should be avoided.

Given a fixed window size, the rate of the TCP source is dictated by the rate of the acknowledgements that it receives. So, the second technique used to enforce the rate is to control and modulate the ack rate in order to smooth out the burstiness in TCP transmissions.

3.2 DTM Related Work

Csaba Antal and Jósef Molnár whilst at Budapest University (now employed at Ericsson traffic lab) investigated the performance of DTM technology as part of their Masters and PhD programs. They investigated a number of channel allocation schemes, including the original KTH ones as well as their own. To test their findings a DTM simulator in C++ was implemented. They published their work in [2] While the work covers DTM, simulation and channel allocation it did not exactly fit the goals for this project. Their investigation did not consider TCP and how to allocate DTM channels according to TCP's bandwidth needs.

Henrik Lundqvist's Masters thesis work looked at the performance evaluation for IP over DTM [9]. The thesis is available as a Linkoping University report (LiTH-IDA-Ex-98/25) and the work was done at NetInsight AB. The goal of the work was to evaluate algorithms which control the bandwidth of DTM channels transporting IP traffic. The algorithms were evaluated using simulations with parameters such as the buffer size and threshold values for adding and removing slots. He looked at both UDP and TCP traffic types with the flow control of TCP together with DTM slot allocation algorithms being specially studied. This makes this work, as far as we know the most relevant to our efforts.

He also investigated how DTM should adjust to the rate of TCP. This is done by placing the incoming packets into a buffer and adding and removing slots if the level of the buffer exceeds continuously maintained threshold values. Secondly he added a dependency on the packet sizes within this buffer to cope with the different sizes of data and ACK packets in TCP. Finally adjustments were made to make the algorithm simple and efficient to implement in a DTM network. The first adjustment was to give fast adaption to a incoming flows and the second, stricter adjustments when changing the actual DTM slot allocation.

Unfortunately this work came to light late in this project so we did not have time to implement or validate the algorithms proposed in the work, however we summarise the findings. One finding relevant to our work is an algorithm which uses the rate of change in the buffer size which could enhance the performance over directly measuring the throughput[2]. However the number of allocations/deallocations needed in the DTM network was high, hence some form of damping is needed. Using damping however requires larger buffers to reduce packet losses, especially if the rate of the incoming flow changes rapidly, but this has the effect that the rate of change in the buffer size becomes less useful. Therefore further investigation is warranted to the usefulness of this particular finding.

4 Simulation environment

We now describe the simulation environment which starts with a short description of the ns simulator. We continue by describing each changeable property in the simulation and where appropriate show a simulation to illustrate the idea.

4.1 The network simulator ns-2

ns-2 [5] is a event driven packet level simulator. It is one of the most widely used packet level simulators due to it's functionality, relative ease of use and open source nature. It is a hybrid simulator using both an object orientated version of TCL called OTCL and C++. Essentially the control is handled by the OTCL, and the data simulation is done by the C++. The mixture of OTCL and C++ was primarily chosen for the convenience of the user, so they have the advantage of being able to change and re-run scripts easily. OTCL was adopted so the class hierarchy of C++ could be mirrored in a

[2]This was also relayed to us by a member of the CNA group

Figure 2: Basic Simulation Topology

TCL. The ns-2 people added some important glue that allows variables to be shared between the OTCL and C++, including during running of script.

More can be read about ns-2 at http://www.isi.edu/nsnam/. The rest of this section discusses the particular features that we use to support DTM like links. The Appendix contains instructions on how to get started with ns-2, run and interpret the OTCL scripts we have used in the simulations.

4.2 Chosen topology

Since the DTM ring will normally be in the centre of a network we chose the simplest possible topology to reflect this. In our simulation topology 2 this means the DTM link is between nodes 2 and 3. The links were initially set to 10Mbits/sec bandwidth and 10ms delay. Our choice of simulation parameters was fairly arbitrary. We chose 10Mbit per second links as they are common in today's networks. Similarly with the delay, we chose 10 milliseconds as the delay of the link. Delay here refers to the transmission delay, *not* the delay experienced to enqueue or dequeue a packet. Additionally the duration of the simulations were chosen between 10 and 60 seconds to reach equilibrium. Once TCP is over slow start it's behaviour is relatively regular and cyclic as is shown in nearly all the plots in the results section.

We vary the bandwidth of the "DTM" link based on measurements of the load leaving node 2. The measurement of this traffic as well the TCP congestion window size in node 1 form the main focus of our simulations.

Different topologies can be simply simulated by using tools such as ping, traceroute and pathchar to ascertain end to end delays and link bandwidths comparable to an operational system.

4.3 Monitoring flows

Measuring the bandwidth of a flow presented to a router is a central part of this project. There are two mechanisms to monitor flows in ns-2. The inappropriately named QueueMonitor can be used to monitor a flow through a node. The more complex Flowmon can be used to gather information about

several flows and is more complex in it's operation. The Flowmon is derived from the QueueMonitor in the simulator class hierarchy.

The throughput of a flow is measured at the ingress of a node and in our simulations at node 2. The actual position of the flow monitor is not too important, since TCP works end to end, eventually one node will see the affect of the flow dynamics. The code snippet below shows a TCL procedure to measure the bandwidth of a flow, it is called once at the start of the simulation and then at each time determined by sampletime

```
set sampletime 0.1 # 100ms
set chan_bw        # 512000 bits/sec

proc monitor_throughput_bytes_qmon {} {

    global ns qmon
    set now [ $ns now ]
    set bytes [ $qmon set barrivals_ ]

    # calc throughput in bits/sec
    set bits_sec [expr [expr $bytes * 8 ] / $sampletime]

    # write to file
    set bits_sec [ expr int($bits_sec) ]
    puts $throughput_qmon_file "$now $bits_sec"

    # call DTM procedure
    dtm_calc_bw $bits_sec

    reset queue monitor
    $qmon set barrivals_ 0

    re-schedule next procedure call
    $ns at [ expr $now + $sampletime ] "monitor_throughput_bytes_qmon"
}
```

The code reads the number of bytes from the Queuemonitor in this case and calculates the the number of bits per second, this is "integerised" value of the throughput in bits per second. Figure 3 shows a plot of the throughout measured by a flow monitor.

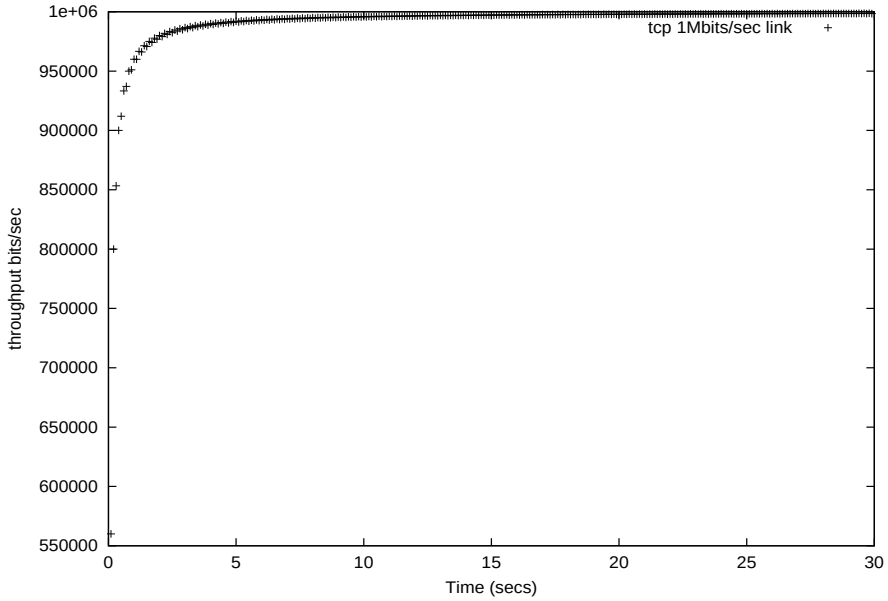

Figure 3: TCP throughput for a 1Mbit constant link

4.4 TCP flavours

ns-2 provides the functionality to simulate nearly all common incarnations of the TCP protocol. We chose to use Reno as this is the most popular flavour of TCP in use today. A study done at ACIRI on the "The TCP Behaviour Inference Tool" so called tbit project reveals that of the nearly 4000[3] showed Reno and new Reno accounting for 66% (36% and 30 %) of the total percentage and Tahoe as 6 % and 26% with and without fast retransmit respectively, approximately 1% were of unknown TCP type. Figure 21 shows TCP tahoe.

4.5 Congestion window monitoring

As stated previously the congestion window (Section 2) a critical component in this work. The throughput is a function of the round trip time, the congestion window, link bandwidths etc. . In ns-2 it is possible to extract the window size with just a few lines of code shown below, we additionally save the time the window was read.

```
# increase size of
```

[3]Web servers were tested

```
# advertised receiver window
$tcp set window_ 500

# record current time
set now [$ns now]

# TCP congestion window and sequence number
set curr_seqn [$tcp set t_seqno_]
set curr_cwnd [$tcp0 set cwnd_]

# Write values and time to file ‘‘tcptrace’’
puts $tcptrace "$now $curr_qsize"
puts $tcptrace "$now $curr_wnd"
```

If the bandwidth of the link is large it may not be possible for TCP to reach the bandwidth of the link so in this case when testing one flow over a 10Mbit link it was necessary to increase the size of the *receiver* advertised window. We chose 500 in this case.

4.6 Packet size

We used 1000 bytes as the packet size, this is the default packet size for TCP in ns-2 however both smaller and larger packets did not reveal significant changes in the simulation results. This is assuming the queue size is specified in packets of course and not bytes, the next section describes the queue sizes.

4.7 Queue length

The queue length can have a dramatic effect on TCP's performance, particularly where the outgoing link is the bottleneck link in a series of routers. A too small queue results in packets being dropped and if too many are lost timeouts and retransmissions must occur. The default queue size is 50 packets.

4.8 Changing the link bandwidth

In order to simulate a DTM like network we need to be able to change the bandwidth of a link dynamically in ns-2. This is done by measuring the bandwidth of a flow, passing it to an estimator function (described in Section 4.11)and calling a function with the new bandwidth for that link,

this is done only when needed. In DTM all the in use channels are scanned every second to see if slots of 512kb can be added or removed depending on the utilisation.

```
proc change_link_bw { bw } {

    global ns n2 n3
    $ns bandwidth $n2 $n3 $bw duplex
}

if { [ expr $now - $savetime_change ]  > $change_time } {

.
. code described in DTM estimator section
.

# bw is in slots
set savetime_change $now
change_link_bw [expr $bw * 512000]
}
```

4.9 Simulation Verification and test

When changing the bandwidth dynamically we need to be sure we do not lose packets as a result Packets could conceivably be discarded from a queue or a link by the simulator when changing the bandwidth. This is important to check whilst lowering the bandwidth. An initial check we made to confirm that the same number of packets are received. The following sections show how TCP can adapt to a link that is changing it's capacity as well as how DTM can adapt to TCP that is changing it's rate.

Figure 3 shows a single TCP connection over a 10Mbits/sec with 10ms delay, 30 second simulation.

We stated at the outset of this work that a difficulty arises due to *both* DTM and TCP being able to control their sending rates. As TCP adjusts to the rate of DTM, DTM adjusts to TCP resulting in a coupling of the two systems. We show one scenerio in the Results section where sending 2 flows through a node with a limited buffer gave rise to a problem in determining the correct bandwidth for the outgoing link of that node. Simply stated the system was not stable.

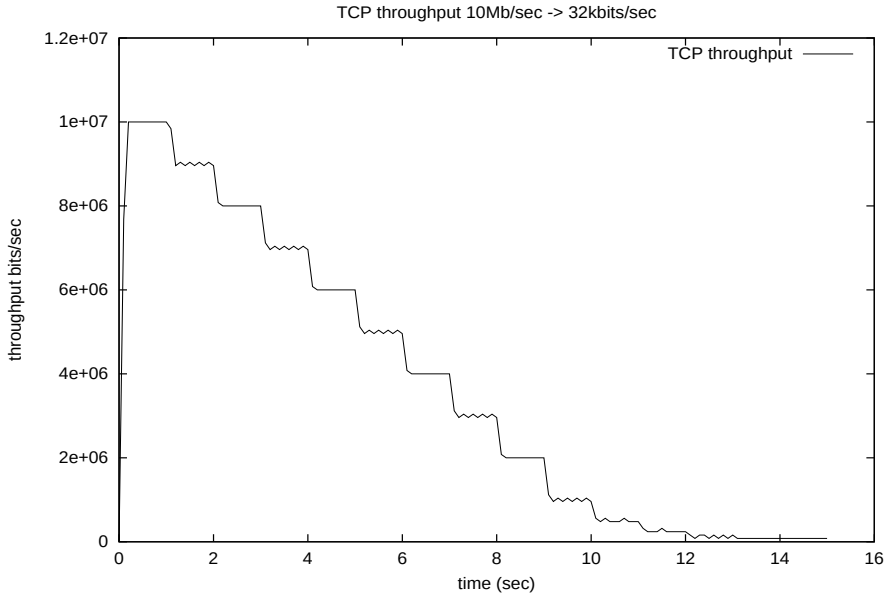

Figure 4: TCP throughput from 10Mbits/sec to 32kbits/sec

4.10 TCP adapting to link bandwidth

Figure 4 shows a single normal TCP connection over 15 secs as the bandwidth is decreased from 10Mbit/sec to 32kbits/sec, the change in bandwidth was initiated manually every second. This shows that TCP can adapt to link bandwidth very well.

A further test was done where the bottleneck link was the first link between nodes 1 and 2, set to 1Mbit/sec. The bandwidth between nodes 2 and 3 was decreased from 10Mbits per second to 32kbits. The goal was to show that decreasing the "dtmlink" would not affect the TCP flow until it reached the same 1Mbit level. This was verify as stated above changing the link bandwidth does not have the effect of ns-2 losing packets and hence TCP's rate being reduced. We would have seen a drop in the rate between 1 and 9seconds if this was the case. Figure 5 shows this scenario.

4.11 DTM estimation

Dynarc AB have a TCP bandwidth estimator. The algorithm estimates the bandwidth by starting at zero and sampling the rate of a given flow. A value is measured and saved. At given intervals, 100 milliseconds, the estimator is called and the previous value compared to the new value and a delta calculated. This delta is 'softened' by a shift value (DTM_RESV_DYNAMIC_SHIFT)

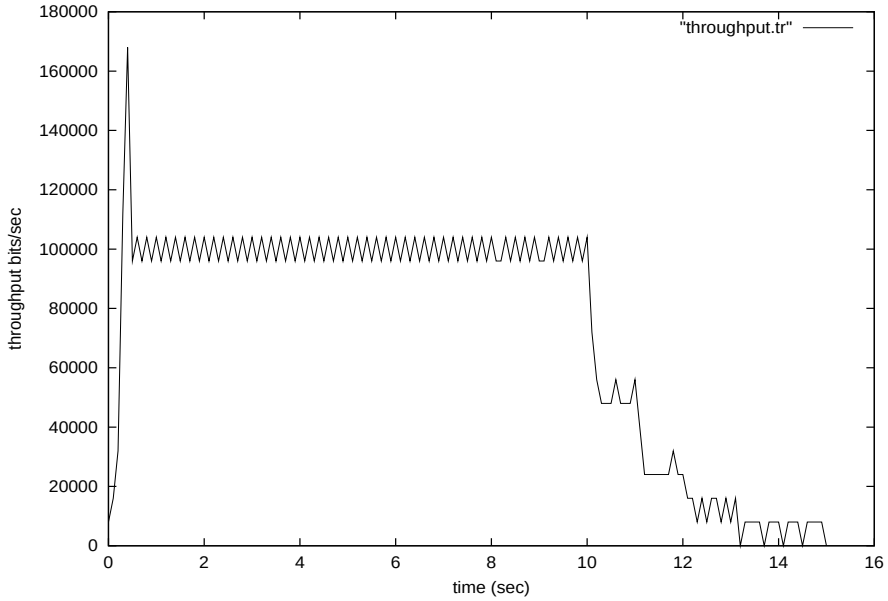

Figure 5: Diminishing DTM link rate (10Mbits/sec to 32kbits/sec)

which divides the delta by a power of 2. So if the shift is 3, which is the value we used the old value of the flow throughput is changed by one eigth towards the new recently measured flow value. The shift therefore determines how aggressively TCP's rate can be tracked. The following Figure 6 shows the affect of halving the delta and dividing by the delta by 8

As can be seen the more aggressive tracking is very good, however one usually does not want to follow the TCP rate too closely. This would result in DTM constantly changing the number of slots for each channel which has an overhead of course. Further mechanisms are employed by Dynarc to soften the dynamicity of DTM's view of TCP which are descried later.

Before we discuss this issue we says a few words about sampling a flow. Figures 7 and 8 (a zoomed version of the first one) show the difference between how often a TCP flow is sampled, in this case we use slow start as the example phase of TCP. Quite clearly to track the throughput of a connection enough sample points are required. This of course depends on the rate of the link where the measuring is being performed (1Mbit/sec in this case) but 100ms would seem reasonable for this capacity of link. It is important to re-iterate that the bandwidth is *not changed* at this frequency merely sampled.

We now show the measured DTM estimation of a TCP connection and the actual bandwidth of a single TCP connection. Figure 9 shows this for a DTM_RESV_DYNAMIC_SHIFT value of 2 which means that the difference

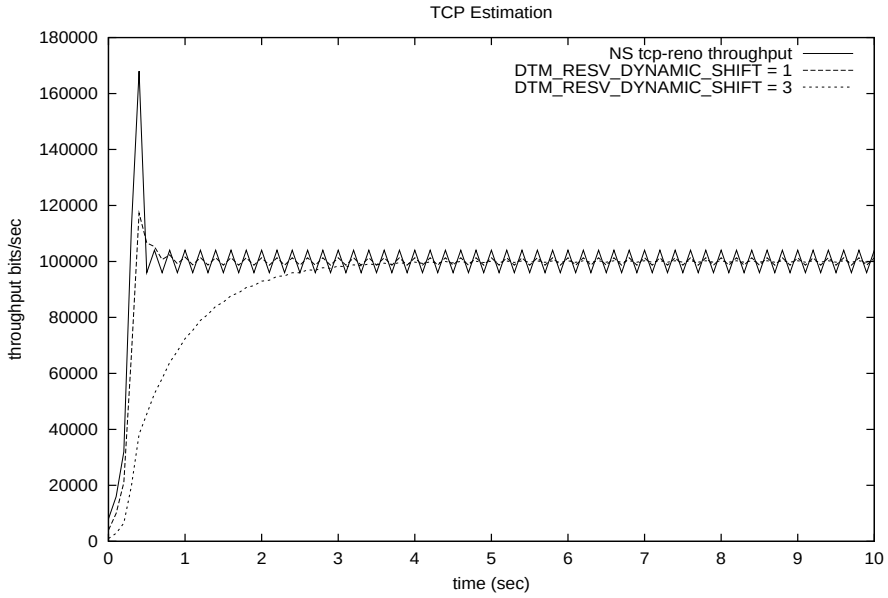

Figure 6: DTM Estimation 1 and 3

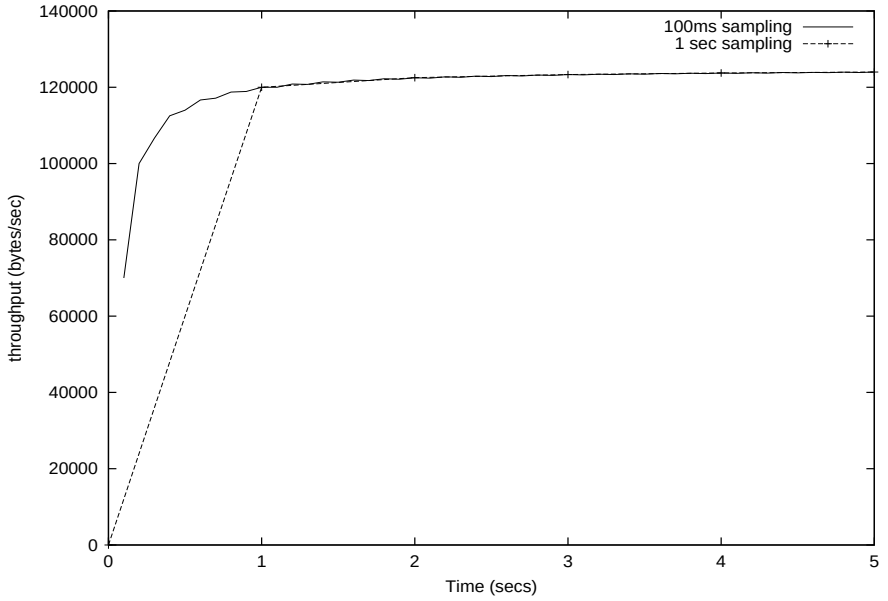

Figure 7: DTM sampling of slow start, 1 sec and 100ms

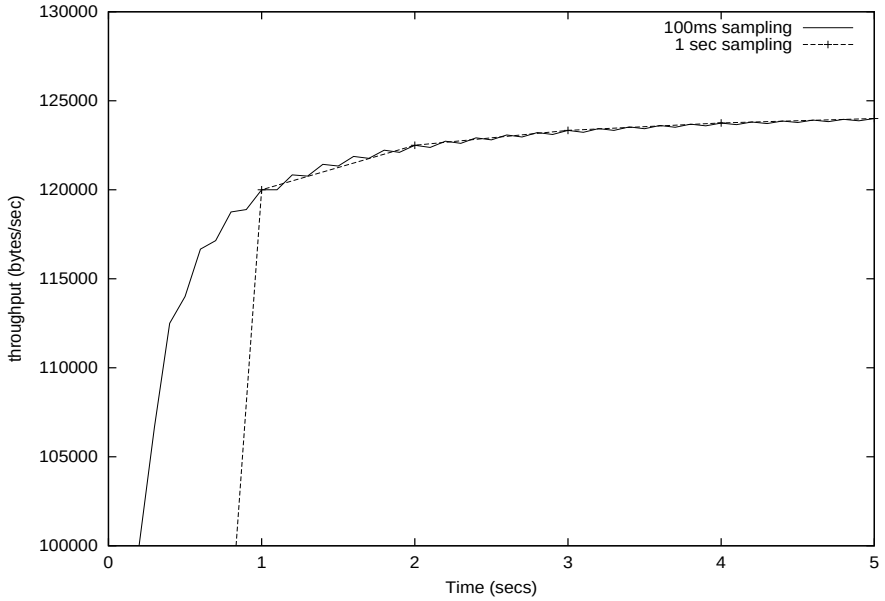

Figure 8: DTM Sampling of slow start, 1 sec and 100ms (Zoomed)

between the previous and new values is decreased by one quarter. Here we can see a problem, although the tracking follows the TCP throughput well it is a little under the rate reported by the flow monitor.

Dynarc use a solution where they add a variable number of slots to the measured bandwidth, one slot is 512kbits. In the following 2 plots the result of this addition of bandwidth is shown, with the measured capacity below under the allocation for the flow. Of course this gives the result of TCP using the bandwidth it needs but also can lead to poor utilisation of the resources.

One would not want to be very generous if one channel were allocated to each TCP flow. As an extra safety margin they add a fixed value of 0.75 of a slot (394kbits) to the measured value as well as the offset. Additionally they define a "corridor" around the measured bandwidth (designated by DynDe in the plots Dynamic Delta) in which the measured bandwidth is allowed to vary before some action is taken to add or decrease the number of slots per channel (per flow). It should be added that although the bandwidth is sampled at intervals of 100ms the bandwidth is changed only every second.

So finally we can show the complete Dynarc algorithm with it measuring and changing the number of DTM slots for a single TCP connection. Figure 12 shows the result for an offset of 1 slot and a corridor of 2 slots. The system seems to working well for this case.

In the case of multiple TCP connections the system seems also to work well as Figure 13 shows.

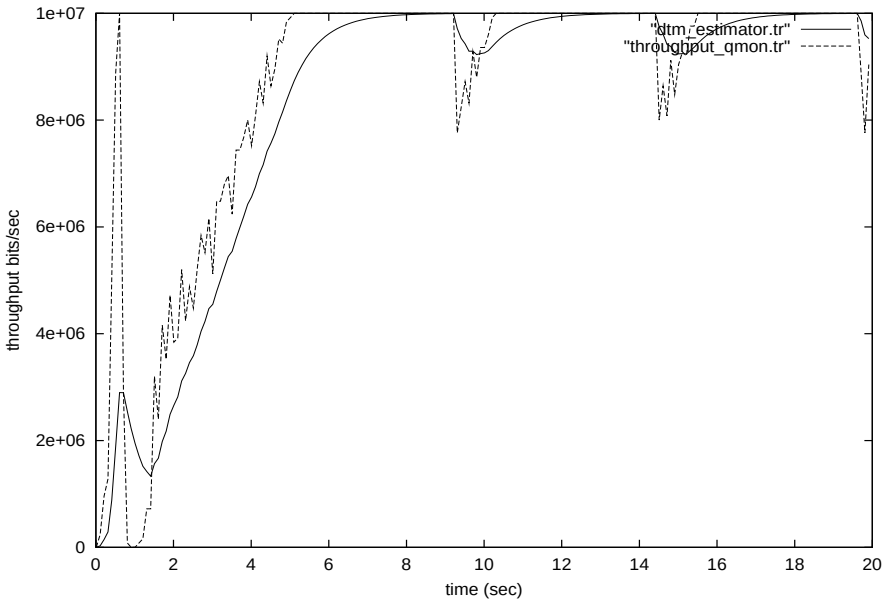

Figure 9: DTM throughput

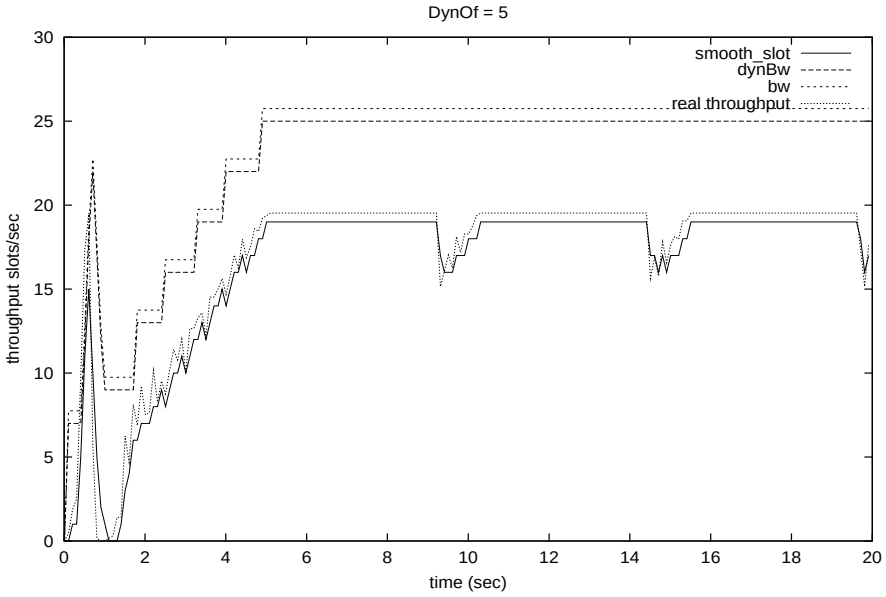

Figure 10: 5 added slots

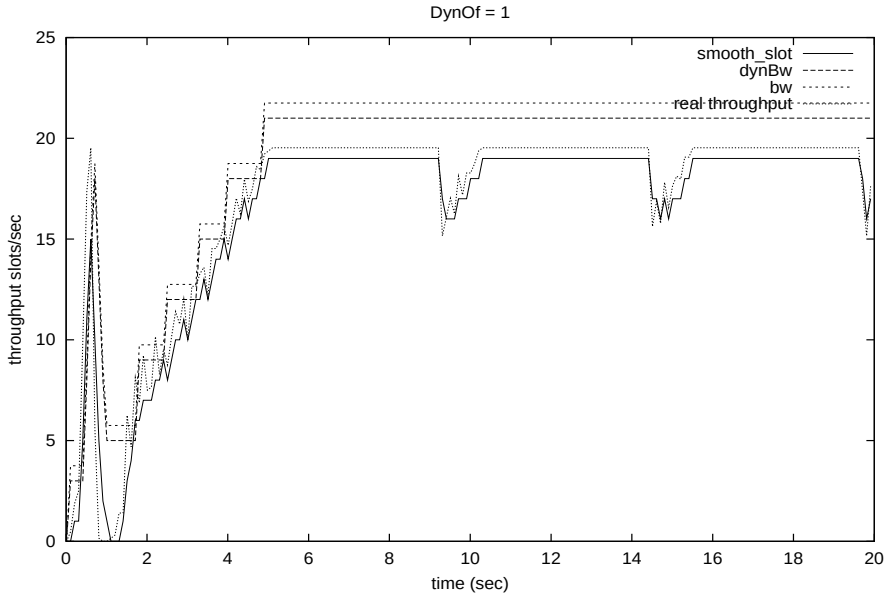

Figure 11: 1 added slot

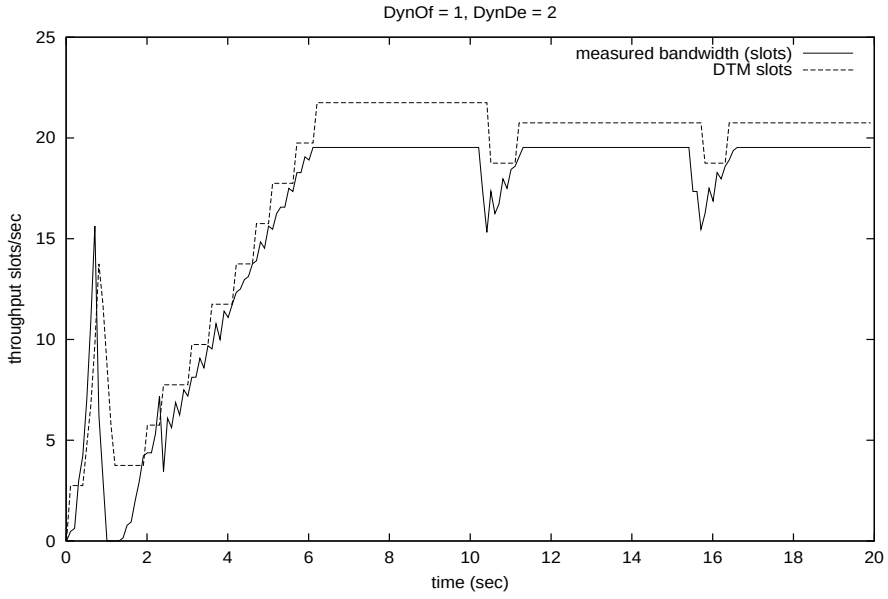

Figure 12: Slots Throughput

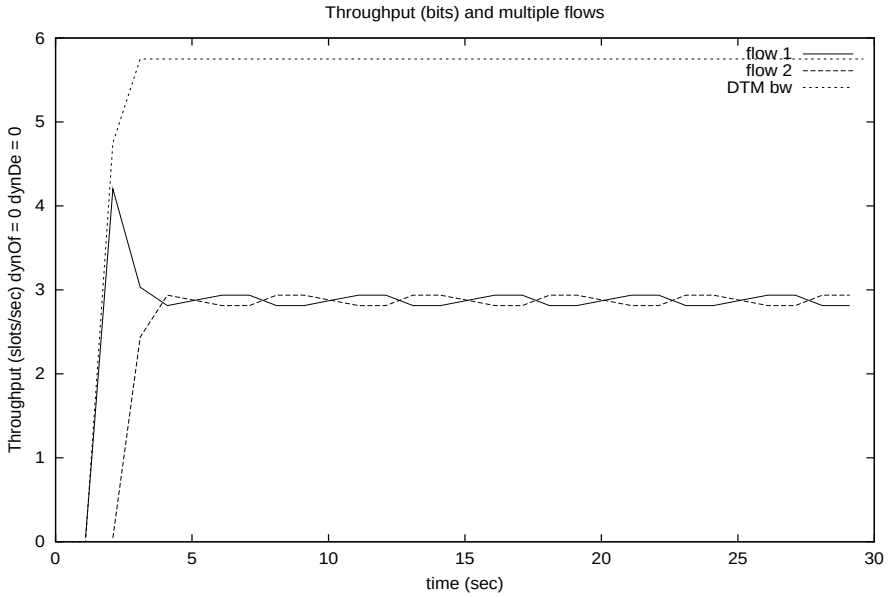

Figure 13: 2 TCP flows per channel

4.12 Final word on the simulation environment

We have presented some parameters of the simulation environment. We tried to be selective and follow certain ideas and strategies however as stated, to cover all possibilities of packet sizes, queue lengths, link bandwidths, window sizes not to mention different types of TCP is nigh on impossible.

5 Simulation results

This section presents simulation results that show how well the "Dynarc algorithm" manages to adapt the bandwidth of the DTM link to that of a bottleneck link. Figure 14 shows the topology used. The 5 Mbit/sec link between node 2 and node 3 is the bottleneck link. The link between node 3 and node 4 is the DTM link with dynamically allocated bandwidth. Initially the DTM link is set to 10 Mbits/sec. The other two links have a capacity of 10 Mbits/sec. A bulk transfer TCP Reno flow which is not limited by the receiver window was setup between node 1 and 5. The throughput was measured at node 4. In the first simulation the queue length was set to 50 packets and subsequent simulations set to 10 packets. The delay on the link between nodes 4 and 5 was altered between 10 and 50 ms.

Figures 15 and 16 show the congestion window and the rate of a TCP Reno flow.

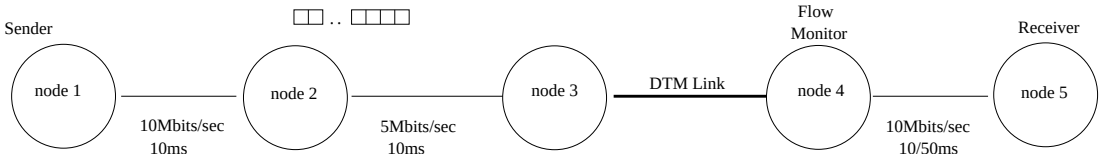

Figure 14: Simulation topology

The paramters of this particular simulation were a queue size of 50 packets at node 2, which is the bottleneck link, and a total of 40 ms link delay, 10 ms for each link. In order to investigate if and how dynamically changing the link bandwidth affects the TCP flow we first kept the bandwidth of the DTM link constant at 10 Mbit. This is shown in Figure 15 This is included for comparison with Figure 16 where the bandwidth is dynamically adjusted. In congestion avoidance the TCP flow increases the congestion window by one packet each round-trip time. This goes on until the TCP flow has filled up all the buffer space at the bottleneck link resulting in a packet drop. The TCP, using fast retransmit and recovery, then cuts the congestion window in half and continues with congestion avoidance. Because of this the congestion window follows a sawtooth curve. As shown in Figure 15, if enough buffer space is available at the bottleneck the rate of the TCP flow is not affected when the congestion window is cut. The third plot in Figure 16 shows the dynamically allocated bandwidth on the DTM link. The algorithm manages to adapt the bandwidth to that of the bottleneck link without affecting TCP.

Figures 17 and 18 show the results when the queue size at the bottleneck was limited to 10 packets. Now the rate of the TCP flow changes with the congestion window, but the changes are too small to affect the dynamic allocation of bandwidth. With a larger link delay the rate variation increases as shown in Figure 19 and the allocated bandwidth on the DTM link now follows these changes as shown in Figure 20. This is even more apparent in Figure 21 and 22 where the simulation with small queue size and a large link delay has been repeated with TCP Tahoe instead of TCP Reno. TCP Tahoe only relies on the retransmission timer and does not use fast retransmit and fast recovery. So, when a packet is dropped the congestion window is set to one and slow-start is invoked.

So far in this report we have shown cases where the dynamic allocation of bandwidth for TCP flows has worked well. In fact after many simulations we performed it was difficult to find a case where the algorithm did not work.

One scenerio where the algorithm has problems was where the queue length of a DTM link was limited.

Figure 23 shows the topology that was used. This topology differs from

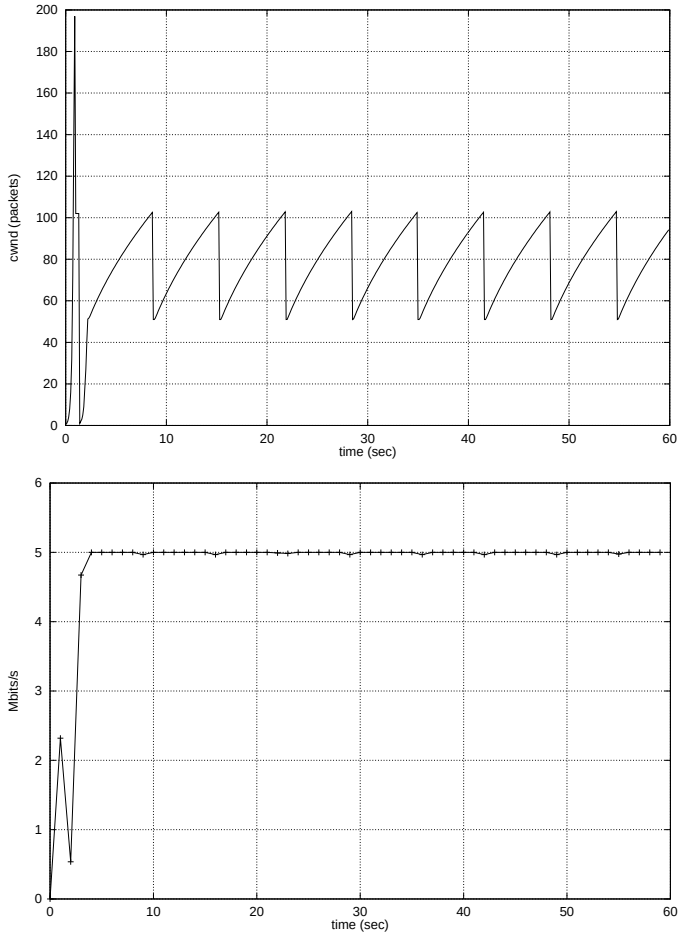

Figure 15: Constant 10 Mbit bandwidth on DTM link. TCP Reno congestion window and measured rate.

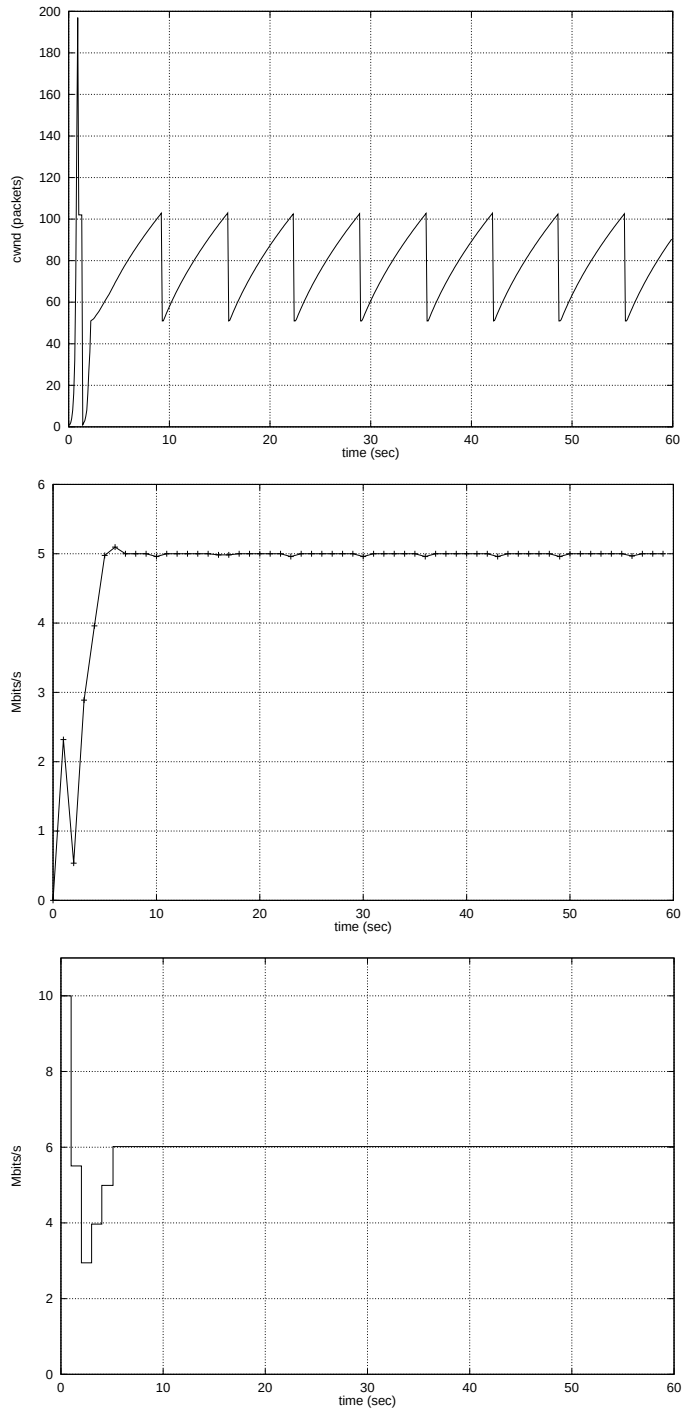

Figure 16: Dynamically allocated bandwidth on DTM link. TCP Reno congestion window, measured rate and the bandwidth on the DTM link.

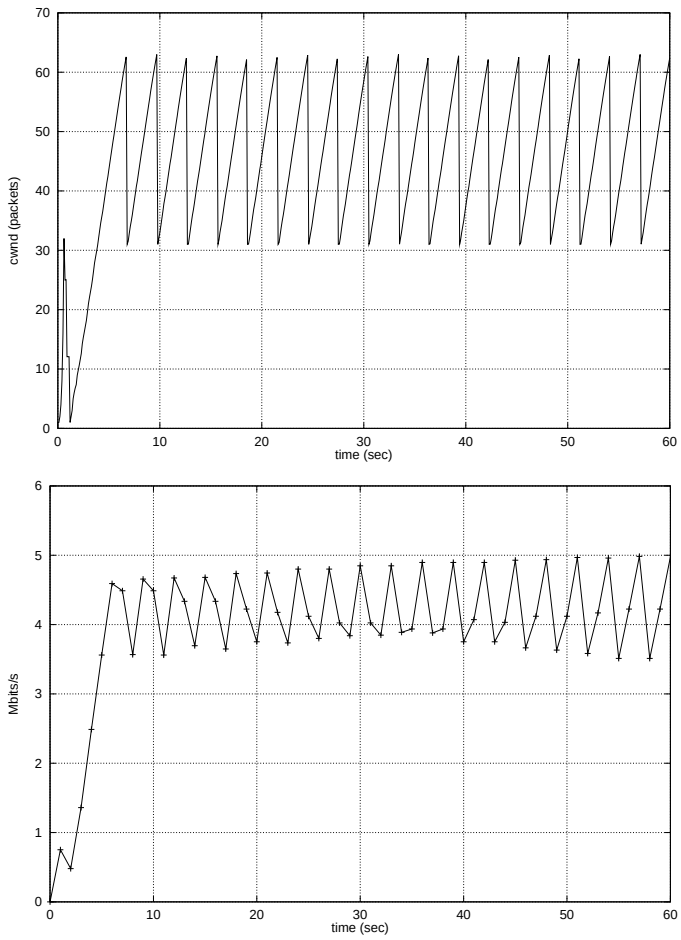

Figure 17: Constant 10 Mbit bandwidth on DTM link. TCP Reno congestion window and measured rate when limited queue-size at bottleneck link

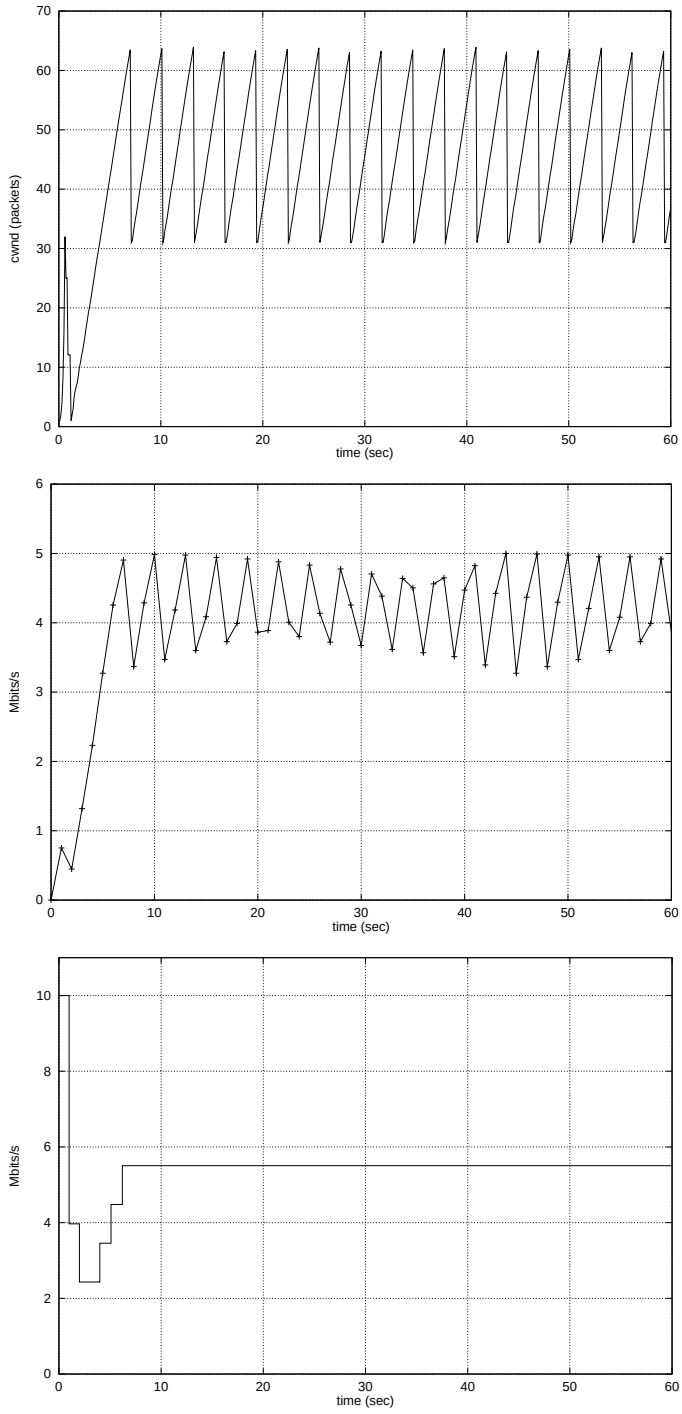

Figure 18: Dynamically allocated bandwidth on DTM link. TCP Reno congestion window, measured rate and the bandwidth on the DTM link when limited queue-size at bottleneck link.

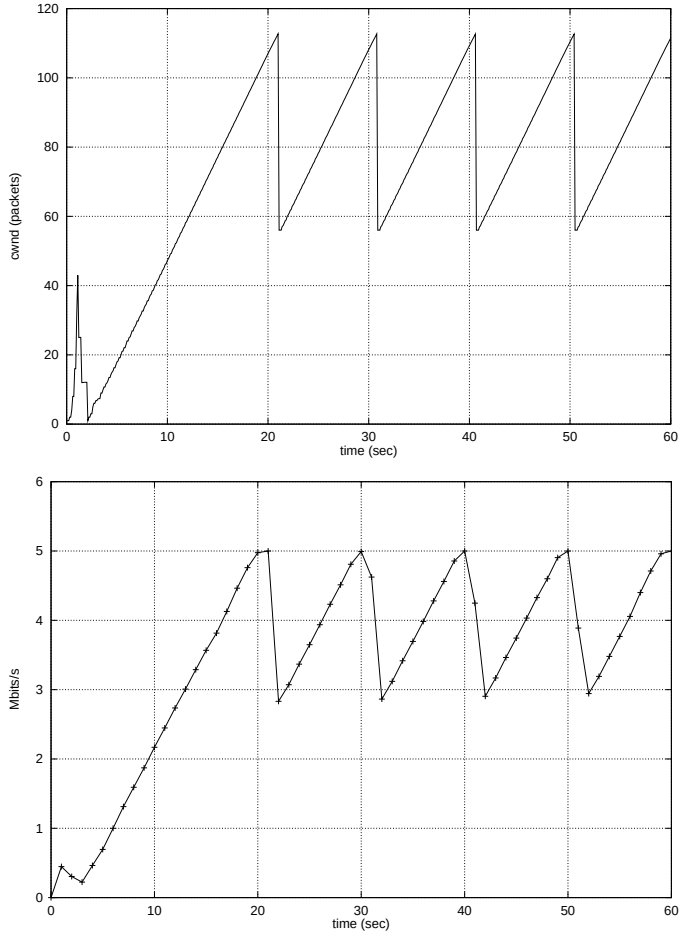

Figure 19: Constant 10 Mbit bandwidth on DTM link. TCP Reno congestion window and measured rate when limited queue-size at bottleneck link and a large link delay.

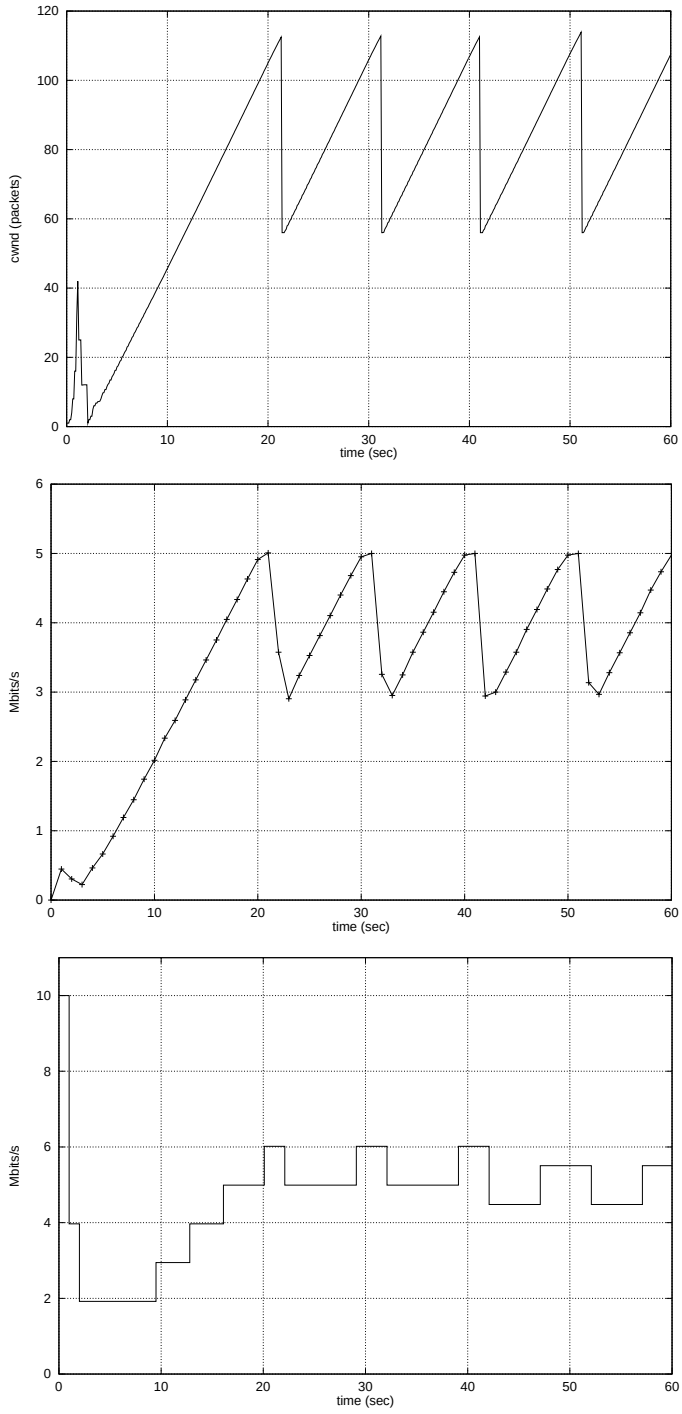

Figure 20: Dynamically allocated bandwidth on DTM link. TCP Reno congestion window, measured rate and the bandwidth on the DTM link when limited queue-size at bottleneck link and a large link delay.

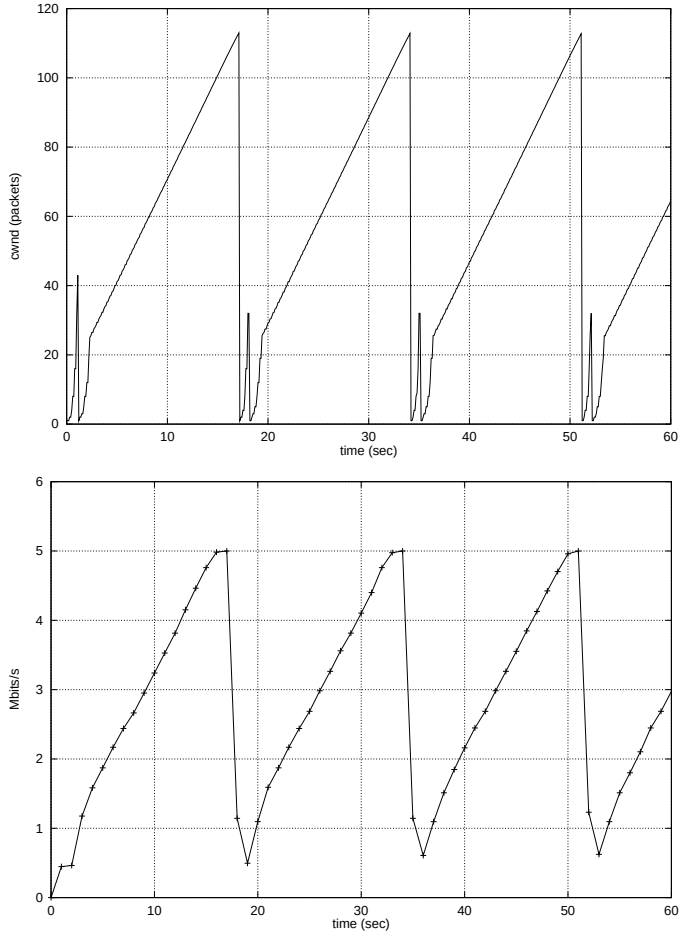

Figure 21: Constant 10 Mbit bandwidth on DTM link. TCP Tahoe congestion window and measured rate when limited queue-size at bottleneck link and a large link delay.

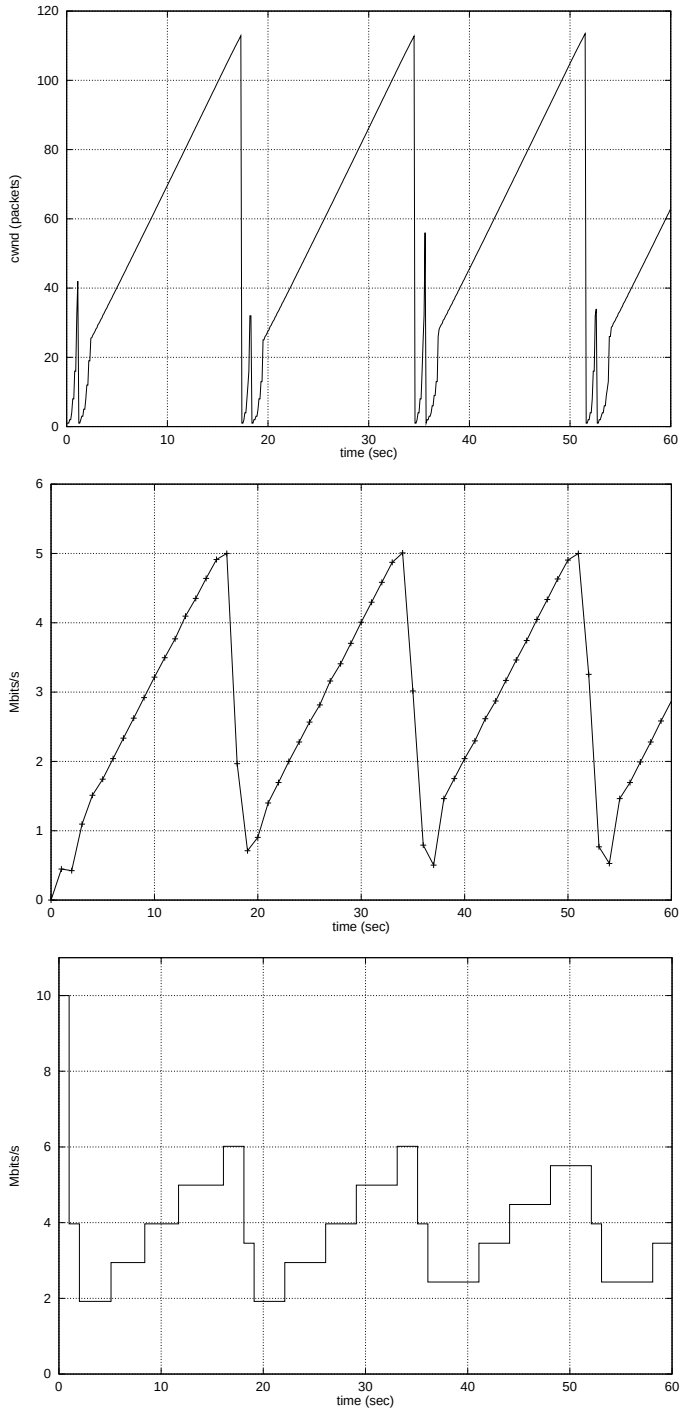

Figure 22: Dynamically allocated bandwidth on DTM link. TCP Tahoe congestion window, measured rate and the bandwidth on the DTM link when limited queue-size at bottleneck link and a large link delay.

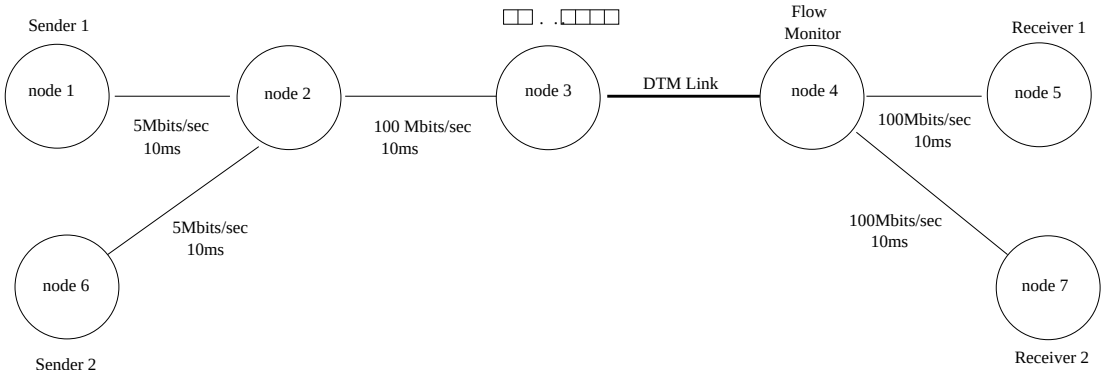

Figure 23: Simulation topology with two flows

previous simulations in that although the capacity of the links between nodes 1 to 2 and 6 to 2 is the same, the flows do share a common output buffer as well as obtain their own input buffer at node 2. Additionally the link feeding the DTM network is 100 Mbits/sec

Figure 24 shows the congestion windows and measured rates for the two TCP flows when the bandwidth on the DTM link is kept constant at 10 Mbit. The TCPs both manage to adapt to the 5 Mbit bottleneck links and are not affected by the limited queue size at the DTM link. But when the simulation was repeated with varying bandwidth on the DTM link the results were quite different. Figure 25 shows the resulting rates and congestion windows and Figure 26 shows the dynamically allocated bandwidth on the DTM link. Neither of the flows manage to reach 5Mbits/sec of their input links.

We hypothesize that the DTM link becomes the bottleneck link and therefore drops packets. If the DTM link has a long queue then the congestion window is not affected as packets are not lost. If one has a small queue, packets are lost which also results in a reduction of the congestion window, hence the rate and thus the estimation of throughput by the DTM algorithm. Some coupling effects of TCP and DTM are not completely understood in this kind of scenerio but warrant further investigation.

6 Conclusions and Future Work

We have created a simulation environment using ns-2 to investigate the TCP protocol over networks which have one variable capacity link. One goal was to verify TCP's operation when one link changes it's capacity. A second goal was to allocate bandwidth for such a link. The main objective was to combine these two systems to assist Dynarc AB in finding effective bandwidth

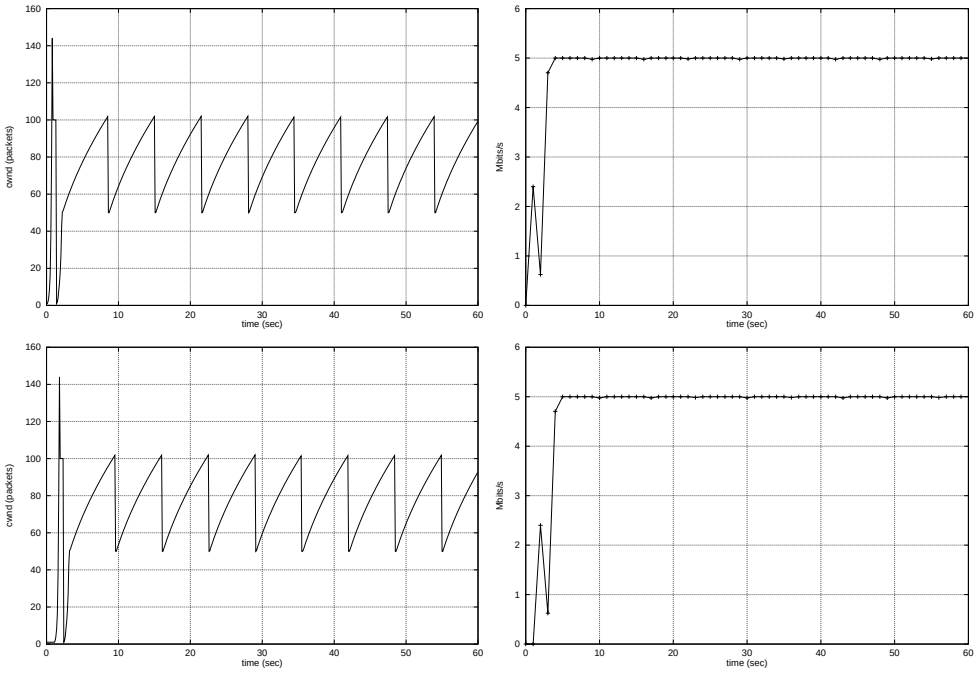

Figure 24: Constant 10 Mbit bandwidth on DTM link. Congestion window and measured rate for two TCP Reno flows when limited queue size at the DTM link.

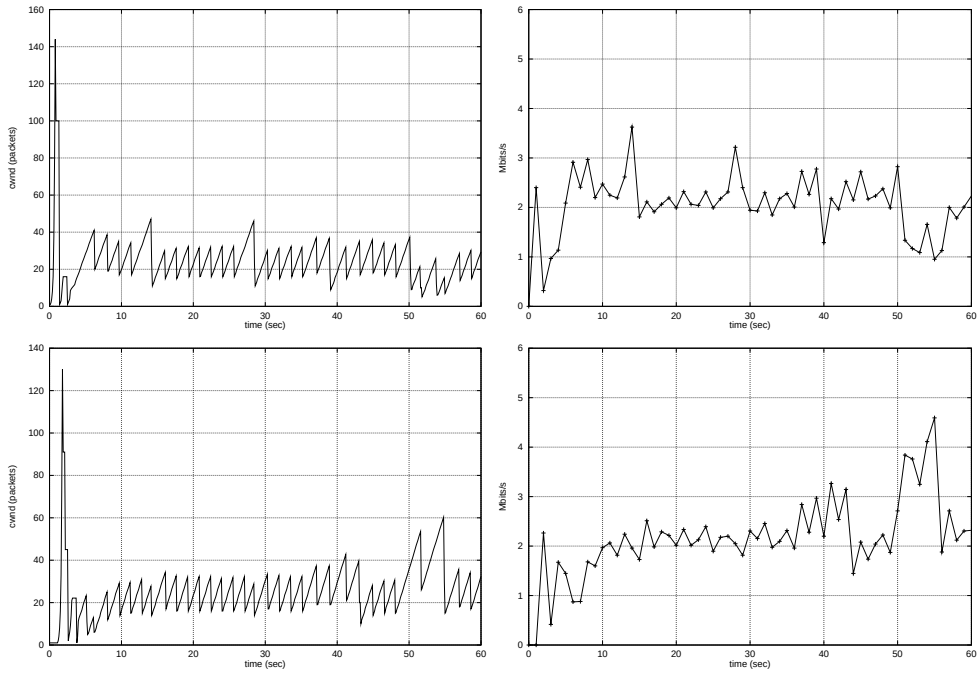

Figure 25: Dynamically allocated bandwidth on DTM link. Congestion window and measured rate for two TCP Reno flows when limited queue size at the DTM link.

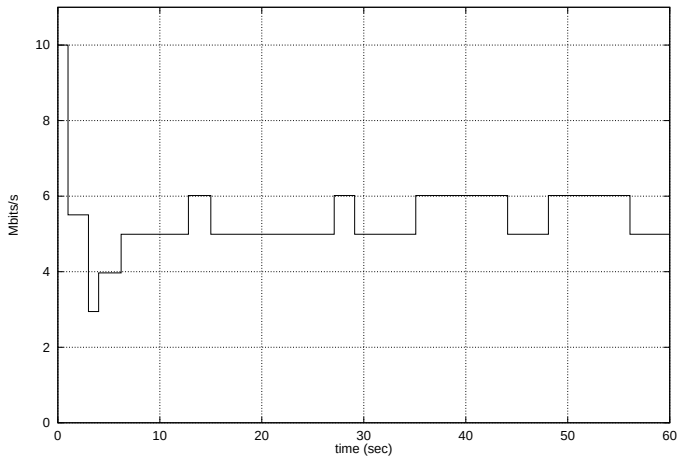

Figure 26: Dynamically allocated bandwidth on DTM link.

allocation schemes for TCP bulk data transfer.

We performed many simulations starting with a single TCP Reno flow over a fixed bandwidth link. We progressively added more complexity to the setup to understand the behaviour of Dynarc's allocation algorithm and its interaction with TCP congestion control algorithm.

The simulation environment enabled us to investigate our stated goals. It allowed us to test and stress the Dynarc algorithm in order to try and make it fail or alternatively to find a scenario where it performed much worse than the fixed link case.

Our overall conclusion is that TCP works extremely work in many different scenarios. This includes networks with both fixed and variable capacity links as our simulation studies have shown. TCP together with the DTM slot allocation scheme worked well in nearly all the simulations we tried.

In our investigations we found one case where the slot allocation algorithm did not perform very well. Figures 25 and 26 show this scenario for a topology shown in 23. As stated we conclude it is due to an interaction of queue size, link bandwidths and interacting TCP flows. These parameters should be investigated as part of further efforts to completely understand such complex interactions and we discuss them in the future work section.

The parameter space, for example queue sizes, link bandwidths, link delays, number of flows, TCP types, network topologies is large and one suggestion for future work are to explore this parameter space. In the case where the slot allocation scheme did not work well it would be advisable to start with these.

Further suggestions include more realistic simulations such as using more flows, using parameters for DynDe and DynOf from real networks as well as build topologies that reflect more real networks. This also includes mixing both UDP and TCP types in the same simulation which we have not performed. It is worth stating this is akin to performing simulations with smaller queue sizes as the UDP takes up a constant fraction of the buffer for it's lifetime.

It could be interesting to look at other algorithms for allocating slots based on measuring traffic, however again we found it more useful in this report to try and test cases where the existing one fails rather than testing and approving a number of possible candidates. Potentially of interest is to look at the queue length in the node rather than the arriving traffic which is simpler to implement and may give a better indication of the usable bandwidth. Henrik Lundqvist's report hinted at this possibility.

Finally the simulation environment we developed should enable further investigations to be performed easily. No extra work needs to be done to start immediately with such investigations, we have developed a test program with

possibilities to change many of the variables we describe as well as functions to process and plot the simulated traffic.

References

[1] M. Allman, V. Paxson, and W. Stevens. TCP congestion control. Internet RFC 2581, April 1999.

[2] Csaba Antal, Jozsef Molner, Sndor Molner, and Gabor Szab. Performance study of distributed channel allocation techniques for a fast circuit switched network. 21(17):1597–1609, November 1998.

[3] O. Bonaventure. *Integration of ATM under TCP/IP to provide services with minimum guaranteed bandwidth*. PhD thesis, University of Liege, 1998.

[4] D. Clark and W. Fang. Explicit allocation of best-effort packet delivery service. *IEEE/ACM Transactions on Networking*, 6(4), August 1998.

[5] Kevin Fall and Kannan Varadhan. ns: Notes and documentation. Technical report, Berkeley University, 1998. Technical Report.

[6] W. Feng, D. Kandlur, D. Saha, and K. Shin. Understanding and improving TCP performance over networks with minimum rate guarantees. *IEEE/ACM Transactions on Networking*, 7(2), April 1999.

[7] L. Gauffin, L. H. Hakansson, and B. Pehrson. Multi-gigabit networking based on DTM. 24(2):119–130, April 1992.

[8] S. Karandikar, S. Kalyanaraman, P. Bagal, and B. Packer. TCP rate control. *Computer Communication Review*, 30(1), 2000.

[9] Henrik Lundqvist. Performance evaluation for IP over DTM. Technical report, Lindkping University, 1998. Masters Report.

[10] Packeteer Inc. http://www.packeteer.com.

[11] J. Postel. Transmission control protocol. Request for Comments 793, Internet Engineering Task Force, September 1981.

[12] W. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[13] J. Wroclawski. Specification of controlled-load network element service. Internet RFC 2211, September 1997.

7 Appendix - ns-2 for dummies

Go to: http://www.isi.edu/nsnam/dist/

and download the latest ns distribution. It's safest to take the 'allinone' package as then one is not depedent on installed Tcl/Tk libraries.

The latest version was on Sun Apr 29:

ns-allinone-2.1b7a.tar.gz

You have wget on your system haven't you ? Good. download the whole package 36Megs of it

```
514 pao [ianm] $ wget http://www.isi.edu/nsnam/dist/ns-allinone-2.1b7a.tar.gz
--10:45:18--  http://www.isi.edu:80/nsnam/dist/ns-allinone-2.1b7a.tar.gz
           => `ns-allinone-2.1b7a.tar.gz'
Connecting to www.isi.edu:80... connected!
HTTP request sent, awaiting response... 200 OK
Length: 38,101,438 [application/x-tar]

    0K -> ......... ......... ......... ......... ......... [  0%]
   50K -> ......... ......... ......... ......... ......... [  0%]
  100K -> ......... ......... ......... ......... ......... [  0%]

some time later .....

37000K -> ......... ......... ......... ......... ......... [ 99%]
37050K -> ......... ......... ......... ......... ......... [ 99%]
37100K -> ......... ......... ......... ......... ......... [ 99%]
37150K -> ......... ......... ......... ......... ......... [ 99%]
37200K -> ........                                           [100%]

10:51:55 (93.87 KB/s) - `ns-allinone-2.1b7a.tar.gz' saved [38101438/38101438]

515 pao [ianm] $ tar zvpfx ns-allinone-2.1b7a.tar.gz
```

another cup of coffee later ...

```
ns-allinone-2.1b7a/ns-2.1b7a/wireless-phy.h
ns-allinone-2.1b7a/ns-2.1b7a/autoconf.h
ns-allinone-2.1b7a/ns-2.1b7a/gen/
```

install the package:

```
516 pao [ianm] $ cd ns-allinone-2.1b7a
517 pao [ns-allinone-2.1b7a] $ ./install
```

You will see alot of output the most important being at the end:

```
Please put

/.amd_mnt/kingkong/host/cna/ianm/ns-allinone-2.1b7a/bin:
/.amd_mnt/kingkong/host/cna/ianm/ns-allinone-2.1b7a/tcl8.3.2/unix:
/.amd_mnt/kingkong/host/cna/ianm/ns-allinone-2.1b7a/tk8.3.2/unix

into your PATH environment; so that you'll be able to run
itm/tclsh/wish/xgraph.

IMPORTANT NOTICES:

(1) You MUST put
/.amd_mnt/kingkong/host/cna/ianm/ns-allinone-2.1b7a/otcl-1.0a6,
/.amd_mnt/kingkong/host/cna/ianm/ns-allinone-2.1b7a/lib,
into your LD_LIBRARY_PATH environment variable.
If it complains about X libraries, add path to your X libraries
into LD_LIBRARY_PATH.
If you are using csh, you can set it like:
          setenv LD_LIBRARY_PATH <paths>
If you are using sh, you can set it like:
           export LD_LIBRARY_PATH=<paths>

(2) You MUST put
/.amd_mnt/kingkong/host/cna/ianm/ns-allinone-2.1b7a/tcl8.3.2/library
into your TCL_LIBRARY environmental
variable. Otherwise ns/nam will complain during startup.

(3) [OPTIONAL] To save disk space, you can now delete directories tcl8.3.2
and tk8.3.2. They are now installed under
 /.amd_mnt/kingkong/host/cna/ianm/ns-allinone-2.1b7a/{bin,include,lib}
```

Do as it says.

```
522 pao [ns-allinone-2.1b7a] $ cd ns-2.1b7a; ./validate
(Validation can take 1-30 hours to run.)
```

If you have a fast PC then it might be worth waiting if not, then it will take long time to run the tests. Come back later.

Refreshed ?, nice film last night ? good, let's continue by showing the last lines of output:

```
Running test onedrop_numdup4_sack_full:
../../ns test-suite-tcpOptions.tcl onedrop_numdup4_sack_full QUIET
Test output agrees with reference output
All test output agrees with reference output.
*** ./test-all-oddBehaviors
Tests: onedrop_reno
Running test onedrop_reno:
../../ns test-suite-oddBehaviors.tcl onedrop_reno QUIET
Test output agrees with reference output
All test output agrees with reference output.
validate overall report: some tests failed:
     ./test-all-webcache
to re-run a specific test, cd tcl/test; ../../ns test-all-TEST-NAME
Notice that some tests in webcache will fail on freebsd when -O is turned on.
This is due to some event reordering, which will disappear when -g is
turned on.
```

The next step is to install the ns binary and man page to a place where it can be accessed.

```
/usr/bin/install -c -m 555 -o bin -g bin ns /usr/local/bin
/usr/bin/install -c -m 444 -o bin -g bin ns.1 /usr/local/man/man1 for
i in indep-utils/cmu-scen-gen/setdest indep-utils/webtrace-conv/dec
indep-utils/webtrace-conv/epa indep-utils/webtrace-conv/nlanr
indep-utils/webtrace-conv/ucb; do cd $i; make install; done

508 pao [ns-allinone-2.1b7a] $ cd ns-2.1b7a/
509 pao [ns-2.1b7a] $ make install
/usr/bin/install -c -m 555 -o bin -g bin ns /usr/local/bin
```

now test to see if it works (deep breath)

```
514 pao [ns-2.1b7a] $ ./ns
% exit
515 pao [ns-2.1b7a] $
```

and NOW! you are ready to run the scripts in the nice package SICS has given you :-)

This is relatively easy, there is a file dtmsim.tcl which you invoke with the ns you just built

```
1237 kio [Tcl] $ ns dtmsim.tcl
```

This will produce some 2 postscript files dtmsim_single.eps for the bandwidth allocated for a single flow and dtmsim_several.eps for the bandwidth allocated for 2 flows. This assumes that you have gnuplot installed and the dtmsim.gnu file in the current directory.

```
1413 kio [Tcl] $ ls -l *.eps
-rw-r--r--  1 ianm  cna  13514 Apr 29 21:23 dtmsim_several.eps
-rw-r--r--  1 ianm  cna  14838 Apr 29 21:23 dtmsim_single.eps
1414 kio [Tcl] $
```

Any other plots that we have shown in this report need to created by adding "puts" commands in the dtmsim.tcl file. Simple knowledge of TCL should suffice to augment the file with some printing commands.

The file is not a good example of TCL program however is fairly modular with most of the key functionality in procedures. The file is given here and when printed fits on a double sided A4 paper in column mode printed with Courier7 font.