Abstract

Nowadays, there are more and more data intensive applications requiring high file transfer over high bandwidth-delay networks. In order to take advantage of the new backbone capacities, which are advancing rapidly to 10Gbps, there is an obvious need for a transport protocol that provides more than 1Gbps of throughput end-to-end.

TCP is the most common protocol used for transferring data reliably over IP networks. Although TCP has shown a remarkable ability to adapt to vastly different networks, it has been shown that TCP is inefficient when the bandwidth and the latency increase.

The purpose of this paper is to present GridDT (Grid Data Transport), a version of TCP that has been modified in order to improve its performance when used over networks with a high bandwidth-delay product. Other approaches to improving TCP, such as FAST will be presented and tested comparatively with GridDT.

The paper is divided into six chapters: *Introduction*, a description of the current TCP and its limitations, *TCP Evolution and actual status*, describing the improvements TCP has suffered over time and other approaches to improving TCP, *Designing GridDT*, a brief presentation of the design guidelines for GridDT, *Implementation*, describing the implementation of mechanisms such as Limited Slow Start and Delayed Congestion Response in GridDT, *Testing and Evaluating GridDT*, presenting test results, and *Conclusions*.

1. Introduction

This chapter will briefly present the current version of TCP (Transmission Control Protocol), and some of the weak points of the current protocol. The purpose of this paper is to present GridDT-Grid Data Transport Protocol (whose implementation consists of a patch for the Linux kernel), and how does this protocol plan to address some of the issues encountered by TCP.

1.1 What is TCP and how does it work?

TCP – Transmission Control Protocol is a connection-oriented data transport protocol used for transferring data reliably.

In order to provide reliability, TCP does the following:

- breaks application data into what TCP considers the best sized segments
- When TCP receives data from the other end, an acknowledgment is sent. TCP uses *expectational acknowledgments;* this means that the acknowledgment number represents the sequence number of the segment that is expected to arrive next.
- when TCP sends a segment, it also sets a timer, waiting for the other end to acknowledge the reception of the segment. If that timer expires, a retransmission is performed.
- TCP maintains a checksum on its header data; if a segment with an invalid checksum is received, it is discarded
- TCP sends data encapsulated IP datagrams, which can arrive out of order at the destination, or can get duplicated. TCP solves this problem by assigning a sequence number to each segment
- TCP also provides flow control. Each TCP connection has a limited buffer space, a TCP receiver will allow the other end to send only as much data as its buffers can hold.

As acknowledging every TCP segment would be inefficient, a windowing mechanism is used. The window defines the amount of data the TCP sender can inject in the network without waiting for an acknowledgment. The window used by TCP is a *sliding window,* which means that it's size can vary. The sliding window mechanism is used by the TCP receiver to impose flow control, as it will correlate the advertised window size with the amount of free buffer space for that connection.

Some TCP related terms will be used extensively throughout this paper:

- RTT – Round Trip Time – the time between sending a segment and receiving the acknowledgment
- Bandwidth-delay product (BDP) – the product between the bandwidth and the delay of a link

- AIMD – Additive Increase Multiplicative Decrease – algorithm used during congestion avoidance; the congestion window is increased additively during periods when congestion is not present, and multiplicatively decreased when congestion is experienced

Also, in order to further understand the operation of the Transport Control Protocol, a short presentation of the header that is attached to each segment is needed.

```
|<------------  IP Datagram ------------>|
            |<------ TCP Segment ------>|
| IP     | TCP    | TCP Data           |
| Header | Header |                    |
```

Figure 1 : The encapsulation of a TCP segment

<table>
<tr><td colspan="8">0   15</td><td>16   31</td></tr>
<tr><td colspan="8">16-bit source port number</td><td>16-bit destination port number</td></tr>
<tr><td colspan="9">32-bit sequence number</td></tr>
<tr><td colspan="9">32-bit acknowledgement number</td></tr>
<tr><td>4-bit header length</td><td>Reserved (6 bits)</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td>16-bit window size</td></tr>
<tr><td colspan="8">16-bit TCP checksum</td><td>16-bit urgent pointer</td></tr>
<tr><td colspan="9">Options (if any)</td></tr>
<tr><td colspan="9">Data (if any)</td></tr>
</table>

Figure 2 : The TCP Header

The *source port and destination port* 16 bit fields are used to identify the sending and receiving application. The association of an IP address with a port number is called a *socket.* A socket pair uniquely identifies a connection.

The *sequence number* field identifies a certain byte in the data stream from the sender to the receiver. TCP assigns a sequence number to each byte.

The acknowledgment number field contains the sequence number of the next expected segment that the sender of the acknowledgment expects to receive.

The header length field contains the length of the TCP header measured in 32 bit words.

There are six flag bits in the TCP header:

- URG – the urgent pointer
- ACK – the acknowledgment number is valid; it is set on all ACKs
- PSH – Push – the receiver should deliver this data to the application as soon as possible
- RST – used to reset a connection
- SYN – Synchronize sequence numbers; used during the connection establishment phase
- FIN – used to terminate a connection

The checksum covers both the header and data.

The urgent pointer is valid only if the URG flag is set. This pointer is a positive offset that must be added to the sequence number field of the segment to yield the sequence number of the last byte of urgent data. TCP's urgent mode is a way for the sender to transmit emergency data to the other end.

As a connection-oriented transport layer protocol, TCP works by establishing connections between a sender and a receiver. A TCP connection's lifetime can be divided into three phases:

- connection establishment
- sending of the actual data
- connection termination

In order to establish a connection between a sender and a receiver TCP uses a mechanism called *three-way handshake*.

During this connection establishment phase, the initial sequence numbers (ISN) are chosen for both the sender and the receiver.

The end initiating the connection sends a TCP segment containing its Initial Sequence Number in the sequence field and the SYN flag set. The receiver will acknowledge this segment with a segment that has its ISN in the sequence field, the SYN flag set, and the sender ISN+1 in the acknowledgment field. The last segment of the three-way handshake mechanism is a segment from the sender with the ACK flag set, and the receiver's ISN+1 in the acknowledgment field.

After the connection establishment phase has ended, the sender starts to send the actual connection data to the receiver. Two TCP mechanisms control how this is done: slow start and congestion avoidance. Related to these two mechanisms are the following TCP variables:

- cwnd – congestion window
- sshthresh – slow start threshold

The congestion window is used in conjunction with the receiver advertised window, the minimum of the two will be used when transmitting. The slow start threshold dictates the end of the slow start phase and the beginning of the congestion avoidance phase.

Initially, cwnd is set to 1 and ssthresh to 65535. During the slow start phase, cwnd is incremented by 1 for each received ACK. The slow start phase will end when cwnd reaches sshthresh. After this, the congestion avoidance phase begins, and during this phase cwnd is incremented by 1/cwnd for each received ACK. Also, the cwnd is never incremented by more than 1 during one RTT.

If packet loss indicated by a timeout is detected, ssthresh is set to cwnd/2, cwnd is set to 1, and slow start is performed. If more than three duplicate acknowledgments are detected, retransmission is done without waiting for the timer to expire (this is called *fast retransmit*) and congestion avoidance is performed instead of slow start (this is called fast recovery).

A TCP connection is usually ended when one end sends a segment with the FIN flag set.

1.2 Shortcomings of the current TCP.

Even though TCP has proven to be very scalable and adaptable during the unpredicted evolution of the Internet, it has been proved that there are some issues related to its performance when used over high bandwidth-delay networks.

It has been shown that when the bandwidth and the delay of a certain link increase, TCP is likely to become unstable.

Inefficiency is another problem that the current TCP will be facing as the capacity of the links in the Internet increases. The current approach to increasing the congestion window is not very efficient when it comes to filling high bandwidths. As presented in the previous section, the current approach is:

- use slow start until the congestion window reaches ssthresh; the congestion window is increased by 1 for every received ACK
- after the slow start phase ends, start congestion avoidance; the congestion avoidance rules state that the congestion window should not be increased by more than 1 during one RTT. On the high bandwidth delay links, this approach can make TCP wait for thousands of RTTs in order to increase the congestion window to a level where bandwidth utilization approaches the maximum value, even though bandwidth is available.

Another potential problem is the aggressiveness of the slow-start phase. When TCP is used over links with a high bandwidth-delay product, the congestion window can reach large values. If the increase in the slow-start phase is too aggressive, a large number of simultaneous drops can occur, drops which will slow TCP down (TCP could enter the congestion avoidance phase with a very small congestion window, and, as a result of this, it will take a large number of RTTs until the original congestion window is recovered). A solution to this problem is implementing Limited Slow Start, as specified in RFC 3742.

Also, there are some issues related to fairness: during the congestion avoidance phase, cwnd is increased by 1/cwnd for each received ACK (and no more than 1 per RTT), this increase does not depend on the characteristics of a certain flow. This generates unfairness between flows with different Maximum Transfer Units (MTU) and different RTTs.

1.3 GridDT

GridDT is a modifed TCP, whose development has started during 2002. The GridDT implementation consists of a patch for the Linux Kernel.The first version of this patch addressed only some of the issues related to the poor performance of TCP, and was developed for the 2.4.20 version of the Linux Kernel.

This paper presents GridDT 3.1, a patch for the 2.4.26 and 2.6.5 versions of the Linux Kernel, which also includes new enhancements.

GridDT v3.1 will implement the following:

- a modified congestion avoidance mechanism, that will increase the congestion window during the congestion avoidance phase with a value which depends on the MTU and RTT of the flow
- a mechanism used during congestion avoidance in order to smooth the increase of the congestion window over one RTT
- Limited Slow Start, the mechanism used to limit the increase of the congestion window during slow-start; implemented as specified in RFC 3742
- DCR – Delayed Congestion Response, a mechanism which is useful in networks where packet reordering is an usual phenomenon; recent studies have revealed that the current Internet is such a case.

A series of comparative tests will also be performed in order to rate GridDT's performance against the performance of other enhanced TCP's.

2. TCP: Evolution and actual status

Looking back in time, it will be noticed that TCP did not initially have the efficiency it has proved during the last twenty years. The tremendous adaptability that TCP has shown was made possible by some small enhancements implemented during its evolution.

TCP has traversed the following stages:

- Vanilla TCP – Slow start and congestion avoidance
- Tahoe TCP – Slow start, congestion avoidance + the Fast Retransmit algorithm was implemented (this algorithm allows a TCP sender to retransmit a segment when more than three duplicate acknowledgments are received, even if the retransmission timer has not expired)
- Reno TCP – also implemented the Fast Recovery algorithm which stated that after a fast retransmit, congestion avoidance, and not slow start should be performed
- New-Reno TCP – it brings a small change to Reno TCP: a retransmission timer expiration will not be waited when multiple packets are lost from a window. The change reflects in a sender's behavior during Fast Recovery, when a partial ACK (an ACK acknowledging only some of the packets that were outstanding at the start of the Fast Recovery period). In Reno TCP, a partial ACK would take TCP out of the Fast Recovery state, by decreasing the usable window to the size of the congestion window. In New-Reno TCP, partial ACKs received during Fast Recovery are treated as an indication that the next packet after the acknowledged segment has been lost and should be retransmitted. When multiple packets are lost New-Reno can recover without waiting for a retransmission timer, by retransmitting one lost packet per RTT, until all lost packets have been retransmitted
- SACK TCP = Selective ACK TCP – also useful when multiple packets from one window are lost. It is implemented as a TCP option, the SACK option field in the TCP header Options contains a number of SACK blocks, where each SACK block reports a non-contiguous set of data that has been received and queued

Attempting to alleviate some of the problems faced by the Transmission Control Protocol as regards high bandwidth-delay networks is currently a common goal throughout the scientific community. Research is being conducted in this field by an increasing number of researchers worldwide. However, there are two main ideas debated by this community. The first group of researchers approaches this problem by maintaining the traditional AIMD approach used in the congestion avoidance algorithm and trying to bring modifications to this algorithm in order to better suit the needs of today's networks. Those who feel that the mechanisms used for congestion detection and avoidance in the current TCP no longer pertain to the network topologies used today form the second group.

2.1 HSTCP – HighSpeed TCP

HighSpeed TCP is a modification to TCP's congestion control mechanism for use with TCP connections with large congestion windows. It was proposed by Sally Floyd and is specified in RFC 3649.

It has been shown that in a steady-state environment, when the packet loss rate is p, the current TCP's congestion window is ~1.2/sqrt(p) segments. This limits the congestion window size that can be achieved by TCP in today's network environments. For example, a standard TCP connection with 1500 byte packets, 100mss RTT, achieving a steady-state 10 Gbps throughput would require an average size of the congestion window of 83,333 segments, which would result in unrealistic requirements: an average packet drop rate of at most 2*10^(-10), which corresponds to a bit error rate of at most 2*10^(-14), which is virtually impossible to achieve with current networks.

HSCTP describes a modified TCP response function (the function mapping the steady-state packet drop rate to TCP's average sending window). Because HighSpeed TCP's modified response function would only take effect with higher congestion windows, HighSpeed TCP does not modify TCP behavior in environments with heavy congestion, and therefore does not introduce any new dangers of congestion collapse. rate in packets per round-trip time) for regimes with higher congestion windows.

The design goals for HSCTP were:

- Achieve high per-connection throughput without requiring unrealistically low packet loss rates.
- Reach high throughput reasonably quickly when in slow -start.
- Reach high throughput without overly long delays when recovering from multiple retransmit timeouts, or when ramping-up from a period with small congestion windows.
- No additional feedback or support required from routers
- No additional feedback required from TCP receivers
- TCP-compatible performance in environments with moderate or high congestion (e.g., packet drop rates of 1% or higher)

In order to specify a modified response function for HSCTP, three parameters are used: Low_Window, High_Window and Low_P. To ensure TCP compatibility, the HighSpeed response function uses the same response function as Standard TCP when the current congestion window is at most Low_Window, and uses the HighSpeed response function when the current congestion window is greater than Low_Window. The HSTCP RFC suggests a value of 38 MSS sized segments for Low_Window, which corresponds to a packet drop rate of 10^-3 for TCP. In order to specify the upper end of the HighSpeed response function, the packet drop rate needed in the HighSpeed response function to achieve an average congestion window of 83000 segments is taken into account. This is the window needed to sustain 10 Gbps throughput, for a TCP connection with the default packet size and round-trip time used in the example above. For High_Window set to 83000, a value for High_P of 10^-7 is specified. This means that a packet drop of 10^-7 will allow HSTCP to achieve an average congestion window of 83.000 segments.

TCP the HighSpeed response function has to be translated into additive increase and multiplicative decrease parameters. The HighSpeed response function cannot be

achieved by TCP with an additive increase of one segment per round-trip time and a multiplicative decrease of halving the current congestion window. Those AIMD parameters are a(w) and b(w). HighSpeed TCP increases the congestion window by a(w)

segments per round-trip time in the absence of congestion, and decreases it to w(1-b(w)) segments in response to a round-trip time with one or more loss events. With Standard TCP, a(w) = 1 and b(w) = 1/2, regardless of the value of w. HighSpeed TCP uses the same values of a(w) and b(w) for $w \le \text{Low_Window}$.

When w = High_Window,

$$a(w) = \text{High_Window}^2 * \text{High_P} * 2 * b(w)/(2-b(w)).$$

The High_Decrease parameter has a suggested value of 0.1, with b(83000)=0.1. For other values w > Low_Window, the following formula is used:

$$b(w) = (\text{High_Decrease} - 0.5)\,(\log(w)-\log(W)) / (\log(W_1)-\log(W)) + 0.5$$

with W = Low_window and W_1 = High_window.

a(w) can then be computed as follows:

$$a(w) = w^2 * p(w) * 2 * b(w)/(2-b(w)),$$ with p(w) being the packet drop rate for congestion window w.

A series of tests has been conducted, comparing the current TCP with HSTCP. The first test presented the behavior of a single TCP flow compared to the one of a HSTCP flow. The results revealed that TCP had a slower growth than HSTCP. TCP needed 300 seconds to reach the bandwidth limit in congestion avoidance, HSCTP reached this limit in 50 seconds. A second test presented the behavior of both TCP and HSTCP with a larger number of flows. The results showed that HSTCP can achieve 100% bandwidth utilization with a smaller number of flows.

Another test has revealed that HSTCP is also better when it comes to bursty traffic. With a level of bursty traffic that reaches 10% bandwidth utilization, TCP flows dropped their bandwidth by almost 70%. HSTCP was also affected, but not as dramatically as TCP. Other tests used include tests carried out on lossy links and links with a constant simulated loss of 10e-5.

However, the results of all of these tests pointed out to the expected conclusion: HighSpeed TCP performs better than TCP for high speed long distance links. Even in the presence of systemic losses, HSTCP flows were able to use almost double the bandwidth used by TCP flows. A single point of concern was discovered during testing: HSTCP has some issues related to being able to insure fairness with TCP flows at low speeds. It has been concluded that a better relation with TCP may be achieved by adjusting the three parameters used by HSTCP, especially Low Window.

2.2 STCP – Scalable TCP

Scalable TCP is an alternative TCP version with improved performance over high-speed networks. Scalable TCP was designed by Tom Kelly, and consists of a simple sender side alteration of the AIMD congestion window update algorithm. It offers a robust mechanism to improve performance in high-speed wide area networks using traditional TCP receivers. A design goal for STCP was to be incrementally deployable and to exhibit the same behavior as traditional TCP stacks when small windows are used.

STCP describes a modified response function with cwnd(T)=a/b * 1/p(T), with p = the packet loss rate. The figure below presents a comparison between STCP's response function and the one used by the current TCP (cwnd=1.2/sqrt(p)).

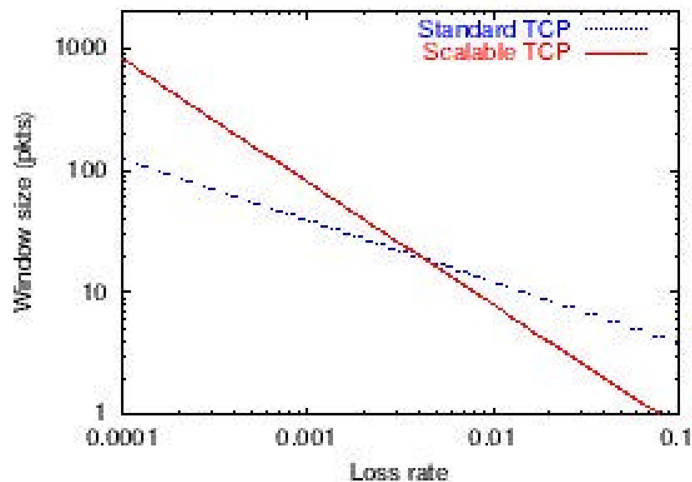

Figure 3 : Comparison between response functions for STCP and TCP

The Scalable TCP algorithm is used only when the congestion window has increased over a certain size. Good resource sharing with traditional TCP connections is insured by choosing the point at which the response curves intersect.

The STCP congestion window algorithm uses two parameters:

- a = the increase in the congestion window
- b = the decrease used when congestion occurs

The equations used by the SCTP congestion window update algorithm are:

cwnd = cwnd + a (per ack)
cwnd = cwnd – b*cwnd

The recommended values for these two parameters are: a=1/100 and b=1/8.

Tests have been conducted over high bandwidth-delay links (1Gbps, 120ms delay) in order to measure the performance gain in Scalable TCP. The results revealed that the performance increase was between 34-175%. The interaction with the current TCP was tested by running Web traffic using the standard TCP in

parallel with several Scalable TCP flows performing bulk transfers. The results showed a negligible impact on the TCP flows. It can be concluded that the design goals of Scalable TCP have been accomplished.

2.3 H-TCP – TCP for high-speed networks

H-TCP is a TCP version modified for high-speed networks, designed by Doug Leith and Robert Shorten at the Hamilton Institute.

Some of H-TCP's design goals include:

- fairness when deployed in homogenous networks
- friendly when competing with conventional TCP sources
- rapidly respond to bandwidth as it becomes available
- utilize bandwidth in an efficient manner

The approach used by H-TCP is the same as STCP's: only change the congestion window update algorithm. The parameters used by H-TCP for AIMD are:

- a = the increase in the congestion window
- b = the decrease used when congestion occurs

In order to meet the requirement according to which H-TCP should perform like conventional TCP when it is not used with high-speed networks, the congestion window update algorithm operates in two modes: high-speed, and conventional. A mode switch is defined, allowing the transition from one mode to the other. The values of the parameters used by the congestion window update algorithm depend on the operating mode. After a predefined period (Δ_L), the transition to using the high performance parameters is done.

The equation defining the values for the AIMD parameters are:

$$\text{If } ?_i = ?^L: a = 1;$$
$$\text{Else if } ?_i > ?^L: a = 1 + 10(?i - ?^L) + [(?i - ?^L)/2]^2$$

where $?i$ is the time since the last congestion event has occurred.,

$$\text{If } |\,[B_i^{max}(k+1) - B_i^{max}(k)] / B_i^{max}(k)\,| > 0.2: b = 0.5$$
$$\text{Else: } b = RTT_{min}/RTT_{max}$$

Where B_i^{max} throughput achieved by source i immediately after a congestion event.

2.4 FAST TCP

FAST is an alternative to the current TCP developed by a team of researchers at Caltech University. As seen by the FAST research team, the weak points of TCP that make it inefficient in networks with large bandwidth-delay products are:

- at the packet level, linear increase by one packet per RTT is too slow, and multiplicative decrease per loss event is too drastic
- at the flow level, maintaining large average congestion windows requires an extremely small equilibrium loss probability
- at the packet level, oscillation is unavoidable because TCP uses a binary congestion signal (packet loss)
- at the flow level, the dynamics is unstable, leading to severe oscillations that can only be reduced by the accurate estimation of packet loss probability and a stable design of the flow dynamics

The FAST design goals map closely to solving those issues that conventional TCP is facing.

All the protocols that have been presented so far belong to the first group, as classified in the introductive part of this chapter (protocols that maintain the loss-based congestion detection mechanisms). FAST belongs to the second group, as its congestion detection mechanism is totally different: it proposes a delay-based solution, using network buffer delay as an implicit congestion signal as opposed to drops. If network buffer delay can be controlled and used as a signaling mechanism, it should be possible to run the network at very high utilizations. Delay-based congestion control has a small advantage over the loss-based approach, but a decisive one at high speed.

It has been pointed out that delay can be a poor or untimely predictor of packet loss, and therefore using a delay-based algorithm in addition to the AIMD congestion window update algorithm of the conventional TCP is the wrong approach. FAST TCP uses a new approach, which exploits delay as a congestion measure in addition to loss information.

Using queuing delay as a congestion measure has two main advantages.

First, queuing delay can be more accurately estimated than loss probability because packet losses in networks with large bandwidth-delay products are rare events and because loss samples provide less granular information than queuing delay factor. Second, the dynamics of queuing delay seems to have the right scaling with respect to network capacity. This helps with maintaining stability as the network scales in capacity.

As regards the architecture of FAST TCP, the congestion control mechanism is separated into four components, which are functionally independent and can be modified so that they can be designed separately and upgraded independently. Figure 4 below presents this separation.

Data Control	Window Control	Burstiness Control
Estimation		
TCP Protocol Processing		

Figure 4 : FAST TCP Architecture

The *data control* component is the one determining which packets to transmit, *window control* decides how many packets to transmit, and *burstiness control* determines the moment when those packets are transmitted. The difference between the last two components is that window control regulates packet transmission over one RTT, while burstiness control is more granular, working at a smaller timescale. All of those components make decisions based on information provided by the *estimation* component. Each of these components will be briefly analyzed in the following part of this chapter.

Estimation

This component provides estimations of various input parameters to the other three decision-making components. It generates a multi-bit queuing delay sample and a one bit loss-or-no loss sample for each data packet. When a positive acknowledgment is received, the estimation component calculates the RTT for the corresponding data packet and updates the average queuing delay and the minimum RTT. If a negative acknowledgment is received (retransmission timer timeout or three duplicate acknowledgments), it generates a loss indication for this data packet to the other components.

Data Control

This component's task is to select the next packet to send from three pools of candidates: new packets, packets that were lost, and transmitted packets that have been acknowledged yet. When there is no loss, new packets are sent in sequence as old packets are acknowledged. This is called *self-clocking*, or *ack-clocking*. During loss recovery, the options are to retransmit lost packets, to keep transmitting new packets, or to retransmit old packets that are neither acknowledged nor marked as lost. The data control makes the decision on how to mix packets from the three candidate pools.

This decision becomes very important when the bandwidth-delay product is large. For example, at a window size of 15,000 packets, a single loss event can lose 7,000 packets, or even more. Those packets have to be retransmitted quickly, but this has to be done in a manner that does not create more congestion and lead to more losses or even timeouts.

A compromise must be done here: the longer we wait, the more certain we are that packets are sent and received, but by waiting we prolong loss recovery. It is

also necessary to transmit a sufficient number of new packets in order to maintain reliable RTT measurements.

Window Control

The Window Control component determines the congestion window based on congestion information provided by the estimation component: queuing delay and packets loss. The congestion control mechanism used with FAST TCP reacts to both queuing delay and packet loss.

Under normal network conditions, FAST periodically updates the congestion window based on the average RTT and average queuing delay provided by the estimation component. When a packet loss is detected, FAST halves its window and enters loses recovery. This is however only a temporary solution, the intended approach is to use the same algorithm for window computation regardless of the sender state.

Burstiness Control

The Burstiness Control component smoothes out the transmission of packets in order to track available bandwidth. It is very important in networks with high bandwidth-delay product, where large bursts of packets may create long queues and even massive losses.

Two burstiness control mechanism are employed:

- Burstiness reduction – decides how many packets to send when an ACK advances the congestion window by a large amount, and attempts to limit the burst size on a timescale smaller than one RTT
- Window pacing – determines how to increase the congestion window over the idle time of a connection to the target determined by the window control component.

An extensive series of tests have been performed by the FAST research team, comparing FAST to TCP Reno, HSTCP, STCP. The results showed the following facts:

- FAST achieved the best overall performance in each of the four evaluation criteria: throughput, fairness, responsiveness and stability
- Both HSTCP and STCP improved the responsiveness of Linux TCP, but they have also presented fairness and instability problems.

FAST TCP has become publicly available for testing since April 2004. It consists of two patches for the Linux-2.4.22 Kernel. Several tests comparing GridDT and FAST will be included in the Chapter 5 of the present paper

2.5 TCP Westwood

TCP Westwood (TCPW) is a sender-side-only modification to TCP NewReno, designed to better handle large bandwidth-delay product paths, with potential packet loss due to transmission or other errors. Its design goals are similar with those of the other protocols:

- improved behavior in networks with high-bandwidth delay products
- only modify the TCP sender
- friendliness to the current TCP.

TCPW bases its operation on information deduced from the received stream of acknowledgments, information, which is used in order to set appropriate values for the congestion control parameters.

TCPW works by estimating an "Eligible rate", used by the sender to update ssthresh and the congestion window when loss is detected. Also , it introduces an "Agile Probing" phase, which is a proposed replacement for the Slow Start mechanism. Also, TCPW uses a mechanism called Persistent Non Congestion Detection (PNCD) is used to detect persistent lack of congestion and induce an Agile Probing phase in order to use large dynamic bandwidth.

TCP Westwood has been tested by the author, but it seems like its performance level is not yet satisfactory.

2.6 GridDT – Grid Data Transport

GridDT is an enhanced TCP version, which keeps the loss-based approach to detecting congestion, but implements several simple mechanisms in order to improve TCP's performance in networks with a high bandwidth delay product.

The development of GridDT started in 2002, and has incorporated the efforts of three: Sylvain Ravot (Network Engineer at Caltech University, currently working at CERN), Adrian Sarbu (Politehnica University Bucharest, Computer Science Faculty), and the author of this paper, student at the Computer Science Faculty, Politehnica University Bucharest.

GridDT is based on modifying the standard TCP, in order to improve the congestion window update algorithm. This algorithm has been modified in order to achieve a more aggressive increase of the congestion window during the congestion avoidance phase. Also, the new algorithm is less aggressive when decrementing the congestion window when congestion occurs. In order to avoid a large number of dropped packets during slow-start, the Limited Slow Start mechanism proposed by RFC 3742 has been implemented.

The GridDT design goals map closely to the problems of the current TCP. They are related to achieving better throughput in networks with high bandwidth-delay products and improving the protocols packet loss handling capabilities. Also, in order to benefit from these improvements, only the sender should be modified, and no additional feedback or support should be required from routers or receivers.

Aspects concerning designing and implementing GridDT, as well as test results, will be explored in depth in the following chapters of this paper.

3. GridDT Design

This chapter describes the steps taken in designing GridDT, the design goals and the decisions taken throughout the design process.

3.1 Design goals

The design goals that governed the GridDT design process are:

- Achieve high per-connection throughput
- Reach high throughput without overly long delays when recovering from multiple retransmit timeouts, or when ramping-up from a period with small congestion windows.
- Improved fairness between TCP flows
- No additional feedback or support required from routers
- No additional feedback required from TCP receivers
- Sender side modifications only
- Smooth interaction with conventional TCP

3.2 Designing GridDT

GridDT consists of a patch for the Linux Kernel. The GridDT version proposed by this paper will result in patches for the Linux kernel versions 2.4.26 and 2.6.5. The first versions of GridDT consisted of patches for Linux 2.4.20.

During the design process of GridDT a series of modifications were proposed. The first one aimed at improving the fairness of the current TCP, which during the congestion avoidance phase, increases the congestion window in the same manner for all flows(the congestion window is increased by 1/cwnd each time an ACK is received, and no more than one per RTT), without taking into account the parameters that characterize that flow (MTU, RTT). GridDT addresses this problem by implementing a modified congestion avoidance function and a function that calculates the increment used during congestion avoidance by taking into account the MTU and RTT of that flow. This function will calculate the increment *add_val* as the product of two variables, which are obtained by comparing the current MSS and the RTT value reference values configurable via sysctl.

Another improvement that GridDT will benefit from is the implementation of the Limited Slow Start mechanism, as proposed by RFC3742. This mechanism is useful in environments where TCP gets to use a large congestion window (thousands or tens of thousands), such as high bandwidth-delay networks. It limits the aggressiveness of the conventional TCP during slow start. With conventional TCP, the current slow-start procedure can result in increasing the congestion window by thousands of segment in a single RTT, increase that can lead to the

loss of a large number of packets in one RTT, which can drastically limit TCP's performance.

Limited Slow Start introduces a parameter "max_ssthresh", and modifies the slow-start mechanisms for values of the congestion window greater than max_ssthresh. The modified algorithm is:

While cwnd<=max_ssthresh, cwnd is increased by one MSS for each arriving ACK during slow-start

When cwnd exceeds max_ssthresh, Limited Slow Start is used:

```
For each arriving ACK in slow-start:
if (cwnd <= max_ssthresh)
        Cwnd+=MSS;
else
        K=int(cwnd/(0.5 max_ssthresh));
        Cwnd+= int(MSS/K);
```

RFC 3742 recommends a value of 100 for max_ssthresh.

Another improvement proposed by this GridDT version is the implementation of a simple mechanism that would limit the burst generated by the new congestion window increase function used during congestion avoidance, by spreading this increase over one RTT.

The latest enhancement added was the implementation of the Delayed Congestion Response (DCR) mechanism. DCR is a simple modification to the TCP congestion control algorithm to make it more robust to non-congestion events. The details of this mechanism are currently specified in a IETF draft.

In the absence of an explicit notification from the network, conventional TCP considers three duplicate acknowledgments an indication of congestion and performs retransmission. This is not always the correct approach, especially in wireless networks with channel errors or networks where excessive packet reordering, as the duplicate acknowledgments can be an just an indication that packets where received out-of-order. It has been shown that the current Internet can sometimes be prone to excessive reordering, contrary to the general belief that packet reordering within the Internet is a rare phenomenon. When this is the case, retransmitting wastes bandwidth, and TCP's efficiency drops. DCR suggests that it would be better to wait longer before responding to congestion. For each duplicate ACK received while delaying the response to congestion, a new packet will be sent and the congestion window will still be incremented.

DCR is based on the following design guidelines:

- improve the robustness of TCP to non-congestion events
- maintain the end-to-end TCP semantics
- require a minimal amount of modification to the network infrastructure
- after the modification, compatibility with existing the flavors of TCP should be maintained

The sender can implement the delay in congestion response (tau) by using either a timer or by modifying the threshold on the number of duplicate acknowledgements to be received before triggering fast retransmit/recovery. The timer-based implementation is quite straight forward, but is influenced by the coarseness in the clock granularity. In the ack-based delay implementation, the sender could delay responding to congestion for the number of duplicate acknowledgements corresponding to the delay required. Thus, if 'tau' is chosen to

be one RTT, the sender would wait for the receipt of 'W' duplicate acknowledgements before responding to congestion, where 'W' is the size of the congestion window when the packet loss is detected. This is the approach that GridDT will use.

It is recommended to use TCP-DCR with TCP-SACK to ensure that the performance can be maintained high even under the conditions of multiple losses per round trip time. When used with TCP-SACK, the only thing modified by TCP-DCR is the time at which the fast retransmit/recovery algorithm is triggered in response to dupacks generated by the first loss within a window of packets. All subsequent losses within the same window (irrespective of whether they are congestion related or non-congestion events) are handled in exactly the same way as TCP-SACK would in the absence of TCP-DCR modifications. If the receiver is not SACK-capable, however, then the sender will have to use TCP-DCR with NewReno.

Another issue that must be considered during the design phase is that when TCP-DCR is used, the receiver will need to have additional buffer space to accommodate the extra packets corresponding to the delay 'tau', when a packet is lost due to congestion. Having these extra buffers allows TCP-DCR to achieve the best performance.

4. Implementation

This chapter presents the implementation details for the mechanisms added to the TCP code in the Linux kernel throughout the development of GridDT. Since GridDT is a patch for the Linux kernel, a brief introduction to the TCP code in the Linux kernel and information on writing a Linux kernel patch will be included. The implementation details for the mechanisms presented in the previous chapter will be detailed later in this chapter, for both Linux 2.4.26 and 2.6.5.

4.1 Linux kernel networking code

The code implementing TCP is thought to be the most complex part of the networking code in the Linux kernel. The files that contain the networking code are located in net/ipv4, net/core, net/sched. The header files are in include/linux, include/net.

Before presenting the actual structure of the TCP code in Linux, a general presentation of the particularities of the Linux TCP implementation is necessary. The first notable difference refers to the congestion window. Traditionally, the congestion window is evaluated against the difference between the highest data segment transmitted and the first unacknowledged segment. The Linux TCP sender determines the number of packets currently outstanding in the network. Then, the sender compares the number of outstanding packets to the congestion window when making decisions on how much to transmit. Another difference is that Linux keeps track of the number of outstanding segments in units of full sized packets. Other implementations compare cwnd to the number of transmitted octets. This makes Linux more conservative compared to the byte-based approach when the TCP payload consists of small segments.

When a TCP connection is established, many TCP values need to be initialized with some fixed-values. In order to improve efficiency at the beginning of the connection, the Linux TCP sender stores in its destination cache the slow-start threshold, the variables used for the RTO (Retransmit TimeOut) estimator, and an estimator measuring the likeliness of reordering after each TCP connection. When another connection is established to the same destination IP address that is found in the cache, the cached values are used to get adequate initial values for the new TCP connection.

Another notable difference is related to the RTO timer calculation. Some implementations use a coarse-grained retransmission timer, with granularities of up to 500ms and the samples used to compute the value for this timer are measured once in a RTT. Linux TCP has a retransmission timer granularity of 10ms, and takes a RTT sample for each segments. Also, the minimum limit for the RTO is 200ms.

Linux TCP also supports the TCP Timestamp option, which helps it to insure accurate RTT measurement for retransmitted segments also.

A state machine with 5 states defines the Linux TCP sender's operation. These states are defined in *include/linux/tcp.h*.

The five states are as follows:

- **Open** – the normal state of the TCP sender. When an acknowledgement is received, the sender increases the congestion window. The way the congestion window is increased depends on weather slow-start or congestion avoidance is performed.
- **Disorder** – this state is entered when duplicate or selective ACKs are detected. The congestion window is not updated, but each incoming packet triggers the transmission of a new segment.
- **CWR** – The TCP sender may receive congestion notifications either by Explicit Congestion Notification, ICMP source quench, or from a local device. When receiving such a notification, Linux does not reduce the congestion window at once, but by one segment for each incoming ACK, until the window size is halved. When the sender is in the process of reducing the congestion window and it does not have outstanding retransmissions, it is in CWR (Congestion Window Reduced). CWR can be interrupted by the Recovery or Loss states.
- **Recovery** – After a certain amount of successive duplicate ACKs were received, the sender retransmits the first unacknowledged segment and enters Recovery state. The default threshold for entering the Recovery state is three duplicate ACKs. During the Recovery state, the congestion window is reduced by one segment for each second incoming ACK, until the window size is equal to sshthresh. The congestion window is not increased during the Recovery state, and the sender either retransmits the segments marked lost, or makes transmissions of new data. The Recovery state is exited when all segments that were outstanding when the Recovery state was entered are successfully acknowledged. After exiting the Recovery state, the TCP sender enters the Open state. The Recovery state can be interrupted by a retransmission timeout.
- **Loss** – When a retransmission timer expires, the senders enters the Loss state. All outstanding segments are marked lost, and the congestion window is set to one segment. Slow-start is performed in order to increase the congestion window. An important difference between the Loss and Recovery states is that in the Loss state the congestion window is increased after the sender has reset it to one segment. Also, the Loss state cannot be interrupted by any other state, thus the sender exits to the Open state only after all data outstanding when Loss state began have successfully been acknowledged.

Before presenting the TCP code in the Linux kernel, a short description of some general structures used in the networking code is necessary.

The networking part of the kernel uses two main data structures: one is used to keep track of the state of a connection (this structure is called *sock*), and another one used for keeping the data and status of both incoming and outgoing packets, called *sk_buff* (socket buffer). A third structure, *tcp_opt*, is part of the *sock* structure and is used to maintain the TCP connection state.

The *sk_buff* structure is defined in *include/linux/skbuff.h*. When the kernel processes a packet, coming either from user space or from the network card, a *sk_buff* structure is created. Changing a field in a packet is achieved by updating a field of this

data structure. In the networking code, virtually every function is invoked with an *sk_buff* (the variable is usually called *skb*) passed as a parameter.

```
struct sk_buff {
/* These two members must be first. */
 struct sk_buff * next; /* Next buffer in list */
 struct sk_buff * prev; /* Previous buffer in list */
 struct sk_buff_head * list; /* List we are on */
 struct sock *sk; /* Socket we are owned by */
 struct timeval stamp; /* Time we arrived */
 struct net_device *dev; /* Device we arrived on/are
leaving by */
```

The first two fields are pointers to the next and previous sk_buffs in the linked list (packets are frequently stored in linked lists or queues); *sk_buff_head* points to the head of the list. The socket that owns the packet is stored in sk (note that if the packet comes from the network, the socket owner will be known only at a later stage). The time of arrival is stored in a timestamp called stamp. The dev field stores the device from which the packet arrived, if the packet is for input. When the device to be used for transmission is known (for example, by inspection of the routing table), the *dev* field is updated correspondingly.

The section referring to the transport control protocol in this structure is a union that points to the corresponding transport layer structure.

```
/* Transport layer header */
union { struct tcphdr *th;
 struct udphdr *uh;
 struct icmphdr *icmph;
 struct igmphdr *igmph;
 struct iphdr *ipiph;
 struct spxhdr *spxh;
 unsigned char *raw;
 } h;
```

The network layer header points to the corresponding data structures (IPv4, IPv6).

```
/* Network layer header */
 union {
 struct iphdr *iph;
 struct ipv6hdr *ipv6h;
 struct arphdr *arph;
 struct ipxhdr *ipxh;
 unsigned char *raw;
 } nh;
```

The link layer is stored in a union called *mac*:

```
/* Link layer header */
 union {
```

```
struct ethhdr *ethernet;
unsigned char *raw;
} mac;
struct dst_entry *dst;
```

Other information regarding the packet length, data length, checksum, packet type, etc.

```
char cb[48];
 unsigned int len;
 /* Length of actual data */
 unsigned int data_len;
 unsigned int csum;
 /* Checksum */
 unsigned char __unused,
 /* Dead field, may be reused */
    cloned, /* head may be cloned (check refcnt to be
sure) */
   pkt_type, /* Packet class */
   ip_summed; /* Driver fed us an IP checksum */
 __u32 priority; /* Packet queueing priority */
 atomic_t users; /* User count - see datagram.c,tcp.c
*/
 unsigned short protocol; /* Packet protocol from
driver */
 unsigned short security; /* Security level of packet
*/
 unsigned int truesize; /* Buffer size */
 unsigned char *head; /* Head of buffer */
 unsigned char *data; /* Data head pointer*/
 unsigned char *tail; /* Tail pointer */
 unsigned char *end; /* End pointer */
```

The *sock* structure keeps data about a specific TCP connection. When socket is created in user space, a sock structure is created.

```
struct sock {
 /* Socket demultiplex comparisons on incoming packets.
*/
 __u32 daddr; /* Foreign IPv4 addr */
 __u32 rcv_saddr; /* Bound local IPv4 addr */
 __u16 dport; /* Destination port */
 unsigned short num; /* Local port */
 int bound_dev_if; /* Bound device index if != 0 */
```

The first fields contain information about the source an destination addresses and ports of the socket pair. The sock structure also contains protocol specific information:

```
union {
  struct ipv6_pinfo af_inet6;
 } net_pinfo;
 union {
   struct tcp_opt af_tcp;
  struct raw_opt tp_raw4;
  struct raw6_opt tp_raw;
  struct spx_opt af_spx;
  } tp_pinfo;
 };
```

The most important part of the sock structure as concerns tcp is the *tcp_opt* field. TCP uses a large number of variables in order to store information about its connections. These variables are stored in the fields of the *tcp_opt* structure.

```
struct tcp_opt {
	int	tcp_header_len;	/* Bytes of tcp header to send
	*/

/*
 *	Header prediction flags
 *	0x5?10 << 16 + snd_wnd in net byte order
 */
	__u32	pred_flags;

/*
 *	RFC793 variables by their proper names. This means you
can
 *	read the code and the spec side by side (and laugh ...)
 *	See RFC793 and RFC1122. The RFC writes these in capitals.
 */
	__u32	rcv_nxt;	/* What we want to receive next	*/
	__u32	snd_nxt;	/* Next sequence we send		*/

	__u32	snd_una;	/* First byte we want an ack for */
	__u32	snd_sml;	/* Last byte of the most recently
transmitted small packet */
	__u32	rcv_tstamp;	/* timestamp of last received ACK
(for keepalives) */
	__u32	lsndtime;	/* timestamp of last sent data packet
(for restart window) */
```

The fields presented above are related to sequence number management. Among them: *rcv_next* – is the sequence of the next expected segment, *snd_next* – the sequence of the segment that will be sent next, *snd_una* – the first byte for which an acknowledgment is expected.

```
/* RTT measurement */
      __u8  backoff;    /* backoff                  */
      __u32 srtt;       /* smothed round trip time << 3       */
      __u32 mdev;       /* medium deviation              */
      __u32 mdev_max; /* maximal mdev for the last rtt period*/
      __u32 rttvar;          /* smoothed mdev_max              */
      __u32 rtt_seq;    /* sequence number to update rttvar    */
      __u32 rto;        /* retransmit timeout            */

      __u32 packets_out;     /* Packets which are "in flight" */
      __u32 left_out;  /* Packets which leaved network      */
      __u32 retrans_out;     /* Retransmitted packets out
      */
```

These fields are related to the measurement of the RTT, which is used by the algorithm that computes the Retransmit TimeOut (RTO).

```
/*
 *    Slow start and congestion control (see also Nagle, and
Karn & Partridge)
 */
      __u32 snd_ssthresh;    /* Slow start size threshold     */
      __u32 snd_cwnd;  /* Sending congestion window          */
      __u16 snd_cwnd_cnt;    /* Linear increase counter       */
      __u16 snd_cwnd_clamp; /* Do not allow snd_cwnd to grow
above this */
      __u32 snd_cwnd_used;
      __u32 snd_cwnd_stamp;
```

The part of the *tcp_opt* structure that is presented above, is used defines *ssthresh*, the limit up until which slow-start is performed, and also contains variables (*snd_cwnd_cnt and* and *snd_cwnd_clamp*) which are used during congestion avoidance, in the *tcp_cong_avoid()* function. *Snd_cwnd_clamp* defines a maximum limit for the growth of the congestion window.

A large number of fields from this structure have been skipped, as the in-depth description of this structure is not within the purpose of this paper. Only the fields directly related to understanding the modifications proposed by GridDT were presented.

The main files that contain the TCP code are located in *net/ipv4* (the headers are located in *include/net*). Those files are:

- tcp_input.c – functions that deal with incoming packets from the network are implemented here
- tcp_output.c – contain functions that deal with sending data to the network
- tcp.c – general TCP code
- tcp_ipv4.c – TCP code that is IPv4 specific
- tcp_timer.c – functions that deal with managing the timers used by TCP
- tcp.h – contains definitions of TCP constants

TCP Input is the largest portion of the TCP code. The sender and receiver code is tightly coupled as an entity can be both at the same time.

Incoming packets delivered to the TCP routines from the IP layer by *ip_local_delivery()*. For IPv4 this routine gives the packet to *tcp_v4_rcv()* which calls *tcp_v4_do_rcv()*. *Tcp_v4_do_rcv()* calls another function depending on the TCP state of the connection.

If the connection is established (state is TCP_ESTABLISHED), it calls *tcp_rcv_established()*.

This is the main case and will be described in depth. If the state is TIME_WAIT, it calls *tcp_timewait_process()*. All other states are handled by *tcp_rcv_state_process()*. For example, this function calls *tcp_rcv_sysent_state_process()* if the state is SYN_SENT.

For some TCP states (e.g., CALL_SETUP), *tcp_rcv_state_process()* and *tcp_timewait_process()* have to initialize the TCP structures. They call *tcp_init_buffer_space()* and *tcp_init_metrics()*. The latter initializes the congestion window by calling *tcp_init_cwnd()*.

The function *tcp_rcv_established()* has two modes of operation: fast path and slow path.

In the code corresponding to the slow path, the seven steps specified in RFC 793 are implemented (plus a few other supplementary operations):

- *tcp_checksum_complete_user()* is called in order to calculate the checksum for a packet . If it is incorrect, the packet is discarded.

- PAWS - The Protection Against Wrapped Sequence Numbers is done with *tcp_paws_discard()*.

The seven steps RFC793 specifies are:

STEP 1: The sequence number of the packet is checked. If it is not in sequence, the receiver sends a DupACK with *tcp_send_dupack()*.

STEP 2: The RST (connection reset) bit is checked. If it is on, it calls *tcp_reset()* and error must be passed to the upper layers.

STEP 3: It checks security and precedence (not implemented yet)

STEP 4, part 1: It checks the SYN bit. If it is on, it calls *tcp_reset()*. This synchronizes sequence numbers to initiate a connection.

STEP 4, part 2: It calculates an estimative for the RTT (RTTM) by calling *tcp_replace_ts_recent()*.

STEP 5: It checks the ACK bit. If this is on, the packet brings an acknowledgment and *tcp_ack()* is called.

STEP 6: It checks the URG (urgent) bit. If it is on, it calls *tcp_urg()*. This makes the receiver tell the process listening in the socket that the data is urgent.

STEP 7, part 1: It processes data on the packet. This is done by calling *tcp_data_queue()*. This function is responsible for giving the data to the user.

STEP 7, part 2: It checks if there is data to send by calling *tcp_data_snd_check()*. This function calls *tcp_write_xmit()* on the TCP output sector.

STEP 7, part 3: It checks if there are ACKs to send with *tcp_ack_snd_check()*. This may result in sending an ACK straight away with *tcp_send_ack()* or scheduling a delayed ACK with *tcp_send_delayed_ack()*. The delayed ACK is stored in *tcp->ack.pending()*.

Another important function is *tcp_ack()*, which is called every time an ACK is received. The first task this function accomplishes is checking if the ACK is valid (that is, if it is within the right hand side of the sending window). If everything is normal, the sender's TCP sliding window is updated with *tcp_ack_update_window()* and/or *tcp_update_wl()*. An ACK is considered normal if it acknowledges the next section of contiguous data starting from the pointer to the last fully acknowledged block of data. If the ACK is dubious (duplicate), fast retransmit is entered with *tcp_fastretrans_alert()*.If the ACK is normal and the number of packets in flight is not smaller than the congestion window, the congestion window is increased by entering slow start/congestion avoidance with *tcp_cong_avoid()*. This function implements both the exponential increase in slow start and the linear increase in congestion avoidance as defined in RFC 793. When in congestion avoidance, *tcp_cong_avoid()* utilizes the variable *snd_cwnd_cnt* to determine when to linearly increase the congestion window.

The fast path is entered in certain conditions in *tcp_rcv_established()*.

The code related to TCP output deals with both data packets and ACKs. The function *tcp_transmit_skb()*, is the most important part in this code section, and executes the following tasks:

- check *sysctl()* flags for timestamps, window scaling and SACK
- build TCP header and checksum
- Set SYN packets
- Set ECN (Explicit Congestion Notification) flags
- Increment TCP statistics
- Call *ip_queue_xmit* (this function is part of the IP code and is the first to process packets coming from the upper layers, TCP in this case) which in turn calls the output part, *ip_output()*.

If there is no error, this function returns. Otherwise, *tcp_enter_cwr()* is called. Errors appear when the output queue is full.

The main files that were modified in the GridDT patches: *include/net/sock.h, tcp_input.c, sysctl.h, sysctl_ipv4.c, include/net/tcp.h* and *include/linux/tcp.h.*

4.2 Writing a Linux kernel patch

Since GridDT is a Linux kernel patch, the present section will briefly explain this concept and the tools and procedures used for creating and applying patches.

A kernel patch is a file that contains the difference between two kernel versions. It is a very convenient way of applying modifications to the kernel source code, because such a file is usually very small and thus easily distributed. However, it should be noted that the patch can only be applied the kernel version it was created for.

Writing a kernel patch consists of a few simple steps:

- first, a copy of the Linux kernel source tree should be created. The desired modifications will be applied to the files in this copy. The original kernel source tree should be kept unaltered.
- The desired modifications will be applied
- The resulting kernel must be tested, and if the results are the expected one, the patch file should be created

In order to create a kernel patch, the *diff* program is used. It's syntax is:

```
diff [options] from-file to-file
```

The most used options when creating a kernel patch are:

-N, or –new-file – in a directory, if a file is found in only one directory, it is treated as present but empty in the other directory

-r – when comparing directories, recursively compare any subdirectory found

-u - Use the unified output format, showing lines (an integer) lines of context, or three if lines is not given. For proper operation, patch typically needs at least two lines of context.

The diff output should be redirected to a file, as it is done in the example below:

Example: diff –Nur linux-2.4.26/ linux-2.4.26-gridDT-lss-dcr > patch-2.4.26-gridDT-lss-dcr

Applying the kernel patch is done using the *patch* program:

patch –p0 < patch-2.4.26-gridDT-lss-dcr

After this, the new kernel should be compiled and used.

4.3 Implementing GridDT

This section describes the modifications applied to the Linux kernel (versions 2.4.26 and 2.6.5) as part of GridDT v3.1. Some of these modifications are new, and the others are enhancements kept from the previous versions. Each implemented mechanism uses some parameters, which are configurable using the Sysctl kernel interface. The sysctl interface is a simple and convenient way of modifying kernel

parameters by using the sysctl command or by writing a value in the file corresponding to that parameter, located on the proc filesystem which is mounted in /proc.

4.3.1 The enhanced congestion window update algorithm

As it has been shown in the previous chapters, the AIMD congestion window update function used by conventional TCP has several issues. First, during congestion avoidance phase the increase of the congestion window is too small (1/cwnd per ACK but no more than one per RTT), which makes it hard for TCP to use high bandwidth-delay product links efficiently. The second issue relates to fairness, and it is raised by the fact that the congestion window is always increased the same, without considering flow parameters such as MTU and RTT. The new congestion window update algorithm implemented in GridDT tries to address those issues. This mechanism was inherited from GridDT version 3. In order to avoid the generating of a burst with this increase, in version 3.1 a mechanism used to smooth the increase of the congestion window over one RTT was also implemented.

Also, the halving of the congestion window when loss occurs is considered too drastic. Another mechanism has been implemented, which allows the congestion window to be decremented by cwnd/x, where x is configurable via sysctl.

4.3.1.1 Modifications for Linux 2.4.26

The first file that was modified was *include/sysctl.h*. The enumeration for */proc/sys/net/ipv4* was modified, and the following lines have been added:

```
NET_IPV4_TCP_MSS_REF=97,
NET_IPV4_TCP_RTT_REF=98,
NET_IPV4_TCP_CWND_DECR=99,
```

Also, *net/ipv4/sysctl_net_ipv4.c* was modified, the following lines were added to the ipv4_table[] vector:

```
{NET_IPV4_TCP_MSS_REF, "tcp_mss_ref",
&sysctl_tcp_mss_ref, sizeof(int), 0644, NULL,
&proc_dointvec},
{NET_IPV4_TCP_RTT_REF, "tcp_rtt_ref",
&sysctl_tcp_rtt_ref, sizeof(int), 0644, NULL,
&proc_dointvec},
{NET_IPV4_TCP_CWND_DECR, "tcp_cwnd_decr",
&sysctl_tcp_cwnd_decr, sizeof(int), 0644, NULL,
&proc_dointvec},
```

The *include/net/tcp.h* file was also modified, by adding the following lines:

```
extern int sysctl_tcp_mss_ref;
extern __u32 sysctl_tcp_rtt_ref;
extern int sysctl_tcp_cwnd_decr;
```

The first two correspond to the sysctl parameters used by the function that calculates the increment during congestion avoidance, *sysctl_tcp_mss_ref* and *sysctl_tcp_rtt_ref*. The third corresponds to *sysctl_tcp_cwnd_decr*, which controls the *tcp_cwnd_down()* function that is called when loss is encountered and TCP is in the recovery state. The default TCP behavior is to decrement the congestion window each second ACK, resulting in a halving of the congestion window at the end of the recovery state. The *new tcp_cwnd_down()* function applies the following formula: *cwnd=cwnd-cwnd/sysctl_tcp_cwnd_decr.*

The default values for these parameters were defined in *net/ipv4/tcp_input.c.*

```
int sysctl_tcp_mss_ref=1500;
__u32 sysctl_tcp_rtt_ref=10;
int sysctl_tcp_cwnd_decr=2;
```

In order to change these values, the sysctl interface is used:
sysctl –w net.ipv4.tcp_mss_ref=*new_value*
sysctl –w net.ipv4.tcp_rtt_ref=*new_value*
sysctl –w net.ipv4.tcp_cwnd_decr=*new_value*

The *tcp_opt* structure was also modified, by adding the following new members:

```
int snd_cnwd_inc;
__u32 snd_cwnd_add_cnt;
__u32 snd_cwnd_div_inc;
__u32 old_cwnd;
```

The first variable, *snd_cwnd_inc* is used to store the additive increment for the congestion window. *Snd_cwnd_add_cnt, snd_cwnd_div_inc and old_cwnd* are used by the mechanism that spreads the increase of the congestion window by *snd_cwnd_inc* over one RTT. The use of these variables will be explained later in this chapter, when referring to the *tcp_cong_avoid()* function.

The function that calculates the additive increment used in the new congestion window update algorithm was included in *net/ipv4/tcp_input.c*, and is called *add_inc*:

```
int add_inc(struct sock *sk)
{
    struct tcp_opt *tp=&(sk->tp_pinfo.af_tcp);
    int mssval, addval, rttval;

   if tcp_current_mss(sk)>=sysctl_tcp_mss_ref)
        mssval=1;
    else mssval=sysctl_tcp_mss_ref/tcp_current_mss(sk);

    /* min RTT over the life of the connection */
    if (tp->srtt<tp->min_rtt || tp->min_rtt==0) tp-
>min_rtt=tp->srtt;
```

```
    if (tp->min_rtt <= sysctl_tcp_rtt_ref) rttval=1;
    else {
            rttval=tp->min_rtt/sysctl_tcp_rtt_ref;
            rttval*=rttval;
        }
    addval=mssval*rttval;
    return addval;
}
```

This function calculates the increment used during congestion avoidance by comparing the values of the MSS and RTT with a set of default values. Those default values are: 1500 bytes for *sysctl_tcp_mss_ref* and 10 ms for *sysctl_tcp_rtt_ref*. The increment used is the product of two components: *mssval* and *rttval*. Mssval will be 1 if the current MSS is greater than the reference value, and s*ysctl_tcp_mss_ref* divided by the current MSS value else; *rttval* takes a value of 1 if the minimum RTT for that connection is less than the reference value, and a value of minimum RTT divided by the reference value in the other cases.

The *add_inc* function is called from the *tcp_cong_avoid()* function, during the congestion avoidance phase. *Tcp_cong_avoid()* is called from *tcp_ack()* when the congestion window must be increased after an ACK has been received. *Tcp_cong_avoid()* separates the state of a TCP connection in two areas: the safe area, when the congestion window is less than the slow start threshold, and slow-start is performed by incrementing the congestion window by one for each ACK received, and the dangerous area, when the congestion window size is above the slow start threshold, and the congestion window is increased by 1/cwnd for each received ACK, but never with more than one in a RTT (this is the behavior of the original function). Both *tcp_cong_avoid()* and *tcp_ack()* have been modified. The original *tcp_cong_avoid()* function was:

```
static __inline__ void tcp_cong_avoid(struct tcp_opt *tp)
{
     if (tp->snd_cwnd <= tp->snd_ssthresh) {
             /* In "safe" area, increase. */
             if (tp->snd_cwnd < tp->snd_cwnd_clamp)
                tp->snd_cwnd++;
     } else {
              /* In dangerous area, increase slowly.
              * In theory this is tp->snd_cwnd += 1/tp-
              >snd_cwnd
                  */
                if (tp->snd_cwnd_cnt >= tp->snd_cwnd) {
                      if (tp->snd_cwnd < tp->snd_cwnd_clamp)
                               tp->snd_cwnd++;
                        tp->snd_cwnd_cnt=0;
                } else
                        tp->snd_cwnd_cnt++;
         }
         tp->snd_cwnd_stamp = tcp_time_stamp;
}
```

The new tcp_cong_avoid function is:

```
static __inline__ void tcp_cong_avoid(struct sock *sk)
{
        struct tcp_opt *tp=&(sk->tp_pinfo.af_tcp);
    if (tp->snd_cwnd <= tp->snd_ssthresh) {
                /* In "safe" area, increase. */
                if (tp->snd_cwnd < tp->snd_cwnd_clamp)
                        tp->snd_cwnd++;
    } else {
             /* GriDT: In dangerous area, increase slowly.
            * In theory this is tp->snd_cwnd+=a(RTT, MSS)/tp-
>snd_cwnd
            * a(RTT, MSS) evaluated by add_inc()
            * Note: Jacobson's congestion avoidance <=> a(RTT,
MSS)=1
             */

             if (tp->snd_cwnd_cnt==0) {
                 tp->snd_cwnd_add_cnt=0;
                 tp->snd_cwnd_inc=add_inc(sk);
                 tp->old_cwnd=tp->snd_cwnd;
                 tp->snd_cwnd_div_inc=tp->snd_cwnd/tp->
snd_cwnd_inc;
              if(!tp->snd_cwnd_div_inc) tp->snd_cwnd_div_inc=1;
             }

               tp->snd_cwnd_cnt++;

               if(tp->snd_cwnd_add_cnt==tp->snd_cwnd_div_inc-1)
{
                               tp->snd_cwnd++;
   #ifdef DEBUG_GridDT
                               monitor_TCP(tp);
   #endif
                               tp->snd_cwnd_add_cnt=0;
                           }

                   if (tp->snd_cwnd_cnt==tp->old_cwnd)
                       tp->snd_cwnd_cnt=0;
                   else tp->snd_cwnd_add_cnt++;


          }
          tp->snd_cwnd_stamp = tcp_time_stamp;
}
```

The *tcp_cong_avoidance()* function has suffered the following modifications:

- it is called with a parameter that is now a pointer to a *sock* structure, because the *add_inc* function calculates the increment for the congestion window based on information contained by the *sock* structure. This did not create any complications because the *tcp_cong_avoid()* was previously called with a *struct tcp_opt* argument, which is a member of the sock structure.

- The increment computed by add_inc is now used to increase the congestion window during congestion avoidance: *tp->snd_cwnd_inc* is used to store the value of the additive increment, as returned by the *add_inc()* function. When entering the *tcp_cong_avoid* function with *snd_cwnd_cnt=0*, *tp->snd_cwnd_inc* is initialized, the value of the current congestion window is stored in *tp->old_cwnd.* (this is necessary, as the value of the congestion window will be increased) and cwnd divided by *snd_cwnd_inc* is stored in *snd_cwnd_div_inc*. Each time *tcp_cong_avoid()* is called during congestion avoidance, both *snd_cwnd_cnt* and *snd_cwnd_add_cnt* are increased. When *snd_cwnd_add_cnt* reaches *snd_cwnd_div_inc*, the congestion window is incremented by one and *snd_cwnd_add_cnt* is reset. This is done until *snd_cwnd_cnt* reaches *old_cwnd*. When this happens, *snd_cwnd_cnt* is reset to zero, and the process starts again.

The *tcp_ack()* function was also changed in order to call *tcp_cong_avoid()* with the new parameter. This was not a problem, since one of the parameters for *tcp_ack()* is a struct sock.

The *tcp_cwnd_down()* function, used in order to reduce the size of the congestion window during recovery, has also been modified.

The original function that standard TCP uses was:

```
static void tcp_cwnd_down(struct tcp_opt *tp)
{
    int decr = tp->snd_cwnd_cnt + 1;
    __u32 limit;

    if (!(limit = tcp_westwood_bw_rttmin(tp)))
      limit = tp->snd_ssthresh/2;

    tp->snd_cwnd_cnt = decr&1;
    decr >>= 1;

    if (decr && tp->snd_cwnd > limit)
      tp->snd_cwnd -= decr;

    tp->snd_cwnd = min(tp->cwnd_cwnd,
tcp_packets_in_flight(tp)+ 1);
    tp->snd_cwnd_stamp = tcp_time_stamp;
}
```

This function decreases the congestion window by one for every second received ACK, resulting in the halving of the congestion window at the end of the recovery phase.

The modified functions is:

```
static void tcp_cwnd_down(struct tcp_opt *tp)
{
    int decr = 0;
    __u32 limit;

    if (!(limit = tcp_westwood_bw_rttmin(tp)))
      limit = tp->snd_cwnd_decr_limit;

    if (tp->snd_cwnd_cnt == sysctl_tcp_cwnd_decr - 1) {
```

```
          decr=1;
          tp->snd_cwnd_cnt=0;
        } else tp->snd_cwnd_cnt++;

        if(decr && tp->snd_cwnd > limit) {

          tp->snd_cwnd-=decr;
    #ifdef DEBUG_GridDT

          printk("\nDecreasing cwnd: snd_cwnd=%d", tp->snd_cwnd);
    #endif
        }

      tp->snd_cwnd = min(tp->snd_cwnd, tcp_packets_in_flight(tp) +
1);
      tp->snd_cwnd_stamp = tcp_time_stamp;
    }
```

The new *tcp_cwnd_down()* function decrements the congestion window by one for every *sysctl_tcp_cwnd_decr* received ACKs. At the end of the recovery phase, *cwnd = cwnd – cwnd/sysctl_tcp_cwnd_decr.*

The *tcp_recalc_ssthresh()* function, that calculates the new *sshthresh* when recovery state is entered, has also been modified. Now it also calculates the new congestion window decrement limit, which no longer is *sshthresh/2*. The original function was:

```
static inline __u32 tcp_recalc_ssthresh(struct tcp_opt *tp)
    {
            return max(tp->snd_cwnd >> 1U, 2U);
    }
```

The modified function is:

```
    static inline __u32 tcp_recalc_ssthresh(struct tcp_opt *tp)
    {
      tp->snd_cwnd_decr_limit = max(tp->snd_cwnd-(tp-> snd_cwnd /
sysctl_tcp_cwnd_decr ), 2U);
      return max(tp->snd_cwnd >> 1U, 2U);
    }
```

The *tcp_complete_cwr function*(), which is called when the CWR (Congestion Window Reduced) state ends, was also modified. The original function was:

```
    static __inline__ void tcp_complete_cwr(struct tcp_opt *tp)
    {
         if (tcp_westwood_cwnd(tp))
           tp->snd_ssthresh = tp->snd_cwnd;
          else
           tp->snd_cwnd = min(tp->snd_cwnd, tp-> snd_ssthresh/ 2);
          tp->snd_cwnd_stamp = tcp_time_stamp;
    }
```

The modified function is:

```
    static __inline__ void tcp_complete_cwr(struct tcp_opt *tp)
```

```
{
   if (tcp_westwood_cwnd(tp))
     tp->snd_ssthresh = tp->snd_cwnd;
   else
     tp->snd_cwnd = min(tp->snd_cwnd, tp->snd_cwnd_decr_limit);
        tp->snd_cwnd_stamp = tcp_time_stamp;
}
```

4.3.1.2 Modifications for Linux 2.6.5

Modifications applied to the Linux kernel 2.6.5 were basically the same. Some minor differences appeared though, as the architecture of the 2.6.5 version of the Linux kernel has slightly changed:

- the definition of the *tcp_opt* structure was moved from *include/linux/sock.h* to *include/linux/tcp.h*
- the format of the ctl_table structure was chaged, so now the definition of a member for ipv_table[] in *net/ipv4/sysctl_net_ipv4.c* looks like this:

```
{
                .ctl_name        = NET_IPV4_TCP_MSS_REF,
                .procname        = "tcp_mss_ref",
                .data            = &sysctl_tcp_mss_ref,
                .maxlen          = sizeof(int),
                .mode            = 0644,
                .proc_handler    = &proc_dointvec,
      },
```

4.3.2 Limited Slow Start

GridDT version 3.1 also implements Limited Slow Start, a mechanism specified by RFC 3742, which limits the increase of the congestion window during slow-start. It works by defining a new threshold, max_ssthresh. While *cwnd < max_ssthresh*, slow-start is performed as usual, by increasing the congestion window with one MSS for each received ACK. When *max_ssthresh<=cwnd<ssthresh*, the following algorithm is used:

For each arriving ACK in slow-start:
if (cwnd <= max_ssthresh)
    Cwnd+=MSS;
else
    K=int(cwnd/(0.5 max_ssthresh));
    Cwnd+= int(MSS/K);

RFC 3742 recommends a value of 100 for max_ssthresh.

4.3.2.1 Modifications for Linux 2.4.26

The first file that was modified was *include/sysctl.h*. The enumeration for */proc/sys/net/ipv4* was modified, and the following lines have been added:
NET_IPV4_TCP_MAX_SSTHRESH=100,

Also, *net/ipv4/sysctl_net_ipv4.c* was modified, the following lines were added to the ipv4_table[] vector:

```
{NET_IPV4_TCP_MAX_SSHTHRESH, "tcp_max_ssthresh",
 &sysctl_tcp_max_ssthresh, sizeof(int), 0644, NULL,
 &proc_dointvec},
```

The *include/net/tcp.h* was also modified, by adding the following line:

```
extern int sysctl_tcp_max_ssthresh;
```

This corresponds to the sysctl parameter used to set the *max_ssthresh* limit. The default value for this parameter was defined in *net/ipv4/tcp_input.c.*

```
int sysctl_tcp_max_ssthresh=100;
```

In order to change these values, the sysctl interface is used:
sysctl –w net.ipv4.tcp_max_ssthresh=*new_value*

Two new members have been added to the *tcp_opt* structure defined in *include/net/sock.h:*

```
__u32 snd_cwnd_slow_start_cnt /* Limited Slow Start: Counter
to update */
__u32 snd_cwnd_slow_start_inc /* Limited Slow Start increment
*/
```

The new *tcp_cong_avoid()* function resulted after the modifications presented in section 4.3.1.1, was modified by replacing the the entire "safe" area with the following lines:

```
   if (tp->snd_cwnd <= tp->snd_ssthresh) {
  /* In "safe" area, increase. */
  if (tp->snd_cwnd < tp->snd_cwnd_clamp) {
   if(tp->snd_cwnd < sysctl_tcp_max_ssthresh)
    tp->snd_cwnd++;
    else{ /* limited slow start */
       tp->snd_cwnd_slow_start_inc=tp->snd_cwnd/(sysctl_tcp_max_ssthresh >>
1);
        if(tp->snd_cwnd_slow_start_cnt < tp->snd_cwnd_slow_start_inc)
           tp->snd_cwnd_slow_start_cnt++;
        else {
              tp->snd_cwnd_slow_start_cnt=0;
              tp->snd_cwnd++;
             }
```

```
        }
}
```

The congestion window will be updated as usual during slow-start, while the congestion window is smaller than the limited slow start threshold, max_ssthresh. After this threshold is reached, the increment used during Limited Slow Start (K) is computed and stored in *snd_cwnd_slow_start_inc*. The Limited Slow Start rules specify that the congestion window should be increased by MSS/K when an ACK is received. The Linux implementation would not permit this, so we count ACKs in *snd_cwnd_slow_start_cnt* and increase cwnd by one when this counter reaches *snd_cwnd_slow_start_inc*.

4.3.2.2 Linux 2.6.5

As regards the 2.6.5 version of the Linux kernel, the differences are those mentioned in section 4.3.1.2.

4.3.3 Implementing TCP DCR – Delayed Congestion Response

Normally, TCP responds to congestion after a timeout is encountered or after three duplicate acknowledgments are received. Treating three duplicate acknowledgments as an indication of congestion is not always correct, especially in wireless networks and network with excessive packet reordering. Recent studies have revealed that the current Internet, contrary to the well established belief, is a network where packet reordering can appear.

The IETF draft that specifies the details of the DCR mechanism state that the limit of three duplicate ACKs is heuristic, and no longer pertains to the network environments today, and concludes that a mechanism used for further delaying the response to congestion is needed.

Two approaches to implementing DCR are possible: the sender can implement the delay in congestion response (tau) by using either a timer or by modifying the threshold on the number of duplicate acknowledgements to be received before triggering fast retransmit/recovery.

The timer-based implementation is quite straight forward, but is influenced by the coarseness in the clock granularity. In the ack-based delay implementation, the sender could delay responding to congestion for the number of duplicate acknowledgements corresponding to the delay required. Thus, if 'tau' is chosen to be one RTT, the sender would wait for the receipt of 'W' duplicate acknowledgements before responding to congestion, where 'W' is the size of the congestion window when the packet loss is detected.

GridDT uses the latter approach. DCR is implemented with a default tau of one RTT, but a mechanism used in order to allow tau to be set to a fraction of RTT is also implemented.

It is recommended to use TCP-DCR with TCP-SACK to ensure that the performance can be maintained high even under the conditions of multiple losses per round trip time. Another issue that must be considered is that when TCP-DCR is used,

the receiver will need to have additional buffer space to accommodate the extra packets corresponding to the delay 'tau', when a packet is lost due to congestion. Having these extra buffers allows TCP-DCR to achieve the best performance. The dimension of the TCP buffers will be modified anyway during testing, as large windows will be used. Modifying these buffers is straightforward and is achieved by modifying some sysctl parameters. The exact procedures will be described in the chapter that refers to testing.

4.3.3.1 Modifications for Linux 2.4.26

The first file that was modified was *include/sysctl.h*. The enumeration for */proc/sys/net/ipv4* was modified, and the following lines have been added:

```
NET_IPV4_TCP_DCR=102,
NET_IPV4_TCP_DCR_DIV=103,
```

Also, *net/ipv4/sysctl_net_ipv4.c* was modified, the following lines were added to the ipv4_table[] vector:

```
{NET_IPV4_TCP_DCR, "tcp_dcr",
 &sysctl_tcp_dcr, sizeof(int), 0644, NULL,
 &proc_dointvec},

{NET_IPV4_TCP_DCR_DIV, "tcp_dcr_div",
 &sysctl_tcp_dcr_div, sizeof(int), 0644, NULL,
 &proc_dointvec},
```

The *include/net/tcp.h* file was also modified, by adding the following line:

```
extern int sysctl_tcp_dcr;
extern int sysctl_tcp_dcr_div;
```

This corresponds to the sysctl parameter used to control DCR. The default value for this parameter was defined in *net/ipv4/tcp_input.c*. The first one, *sysctl_tcp_dcr* dictates whether or not to use DCR. The second one is used as a divisor, allowing the DCR implementation in GridDT to use a value for tau that corresponds to a fraction of RTT.

```
int sysctl_tcp_dcr=1;
int sysctl_tcp_dcr_div=1;
```

In order to change these values, the sysctl interface is used:

sysctl –w net.ipv4.tcp_dcr=*new_value*
sysctl –w net.ipv4.tcp_dcr_div=*new_value*

Two new members have been added to the *tcp_opt* structure defined in *include/net/sock.h:*

```
int dcr_fastretrans_thresh;
int dcr_last_dupack;
```

dcr_fastretrans_thresh is the new threshold, and *dcr_last_dupack* holds the sequence number of the last seen duplicate acknowledgment.

The *tcp_init_metrics()* function defined in */net/ipv4/tcp_input.c* was modified, in order to initialize the *tp->dcr_fastretrans_thresh* to TCP_FASTRETRANS_THRESH, a constant with a value of 3, defined in *include/net/tcp.h*. The value of *dcr_fastretrans_thresh* is updated when duplicate ACKs are received.

The *tcp_ack()* function, which is called when an ACK is received, has also been modified, to be exact the part testing if an ack is dubious. The original code was:

```
   if (tcp_ack_is_dubious(tp, flag)) {
        /* Advance CWND, if state allows this. */
       if ((flag&FLAG_DATA_ACKED) && prior_in_flight >= tp->
snd_cwnd && tcp_may_raise_cwnd(tp, flag))
           tcp_cong_avoid(sk);
       tcp_fastretrans_alert(sk, prior_snd_una, prior_packets,
flag);
   }
```

The modified code:

```
if (tcp_ack_is_dubious(tp, flag)) {
   /* Advance CWND, if state allows this. */
   if ((flag&FLAG_DATA_ACKED) && prior_in_flight >= tp-
>snd_cwnd && tcp_may_raise_cwnd(tp, flag))
       tcp_cong_avoid(sk);
   /* DCR */
   if (tp->snd_una == prior_snd_una && !(flag&FLAG_NOT_DUP))

    if ( sysctl_tcp_dcr &&  (tp->snd_cwnd/sysctl_tcp_dcr_div >
TCP_FASTRETRANS_THRESH) && ack!=tp->dcr_last_dupack)
     {
      tp->dcr_last_dupack=ack;
      tp->dcr_fastretrans_thresh=tp
>snd_cwnd/sysctl_tcp_dcr_div;
   #ifdef DEBUG_GridDT
   printk("\nGot dupack! Setting threshold to %d", tp-
>dcr_fastretrans_thresh);
   #endif
     }
/* DCR */

 tcp_fastretrans_alert(sk, prior_snd_una, prior_packets,
flag);
}
```

The modifications added are simple: we test if we have a duplicate ACK, and if this is true, test if the congestion window divided by the DCR divider is greater than three (TCP_FASTRETRANS_THRESH), as it would be pointless to update the threshold otherwise. We also test if the sequence number of the newly received

acknowledgement is different than the sequence number of the last seen duplicate acknowledgment. This verification is necessary as the update of *dcr_fastretrans_thresh* must be performed only when congestion is detected (when the first duplicate ACK in a series is received), and not for all subsequent duplicate ACKs. If the conditions mentioned above are true, *tp->dcr_fastretrans_thresh* is updated to *tp->snd_cnwd/sysctl_tcp_dcr_div*. With the default value of one for *sysctl_tcp_dcr_div*, congestion delay response is delayed for one RTT.

Another function modified was *tcp_init_metrics()*. This is were *tp->dcr_fastretrans_threshold* is initialized to TCP_FASTRETRANS_THRESH (this is the default value, which is modified when dupacks are received).
The *tcp_time_to_recover()* function has also been modified, by replacing *sysctl_tcp_reordering*(which was previously used to statically modify the fast retransmit threshold) with *tp->dcr_fastretrans_thresh.* This function deals with deciding weather to decrease the congestion window, by differentiating between loss and reordering.

4.3.4 Debugging GridDT

As GridDT is currently in development, it comes with debugging activated. Disabling the debug features in GridDT is done by commenting the macro definition of DEBUG_GridDT in *net/ipv4/tcp_input.c.*

A simple function called monitor_TCP is used:

```
void monitor_TCP(struct tcp_opt *tp) {
printk("0x%x @%d YY cwnd:%d,%d ack:99, srtt:%d cwr:99,99 mrtt:99,99,99\n", (int)tp, tcp_time_stamp, (int)tp->snd_cwnd, (int)tp->snd_ssthresh, (int)tp->srtt);
}
```

It displays: the *tcp_time_stamp*, the value of the congestion window, the value of *sshthresh,* and SRTT (Smoothed Round Trip Time). Some of the graphs presented in the testing section have been drawn based on the output generated by this function.

This function is called inside the *tcp_cong_avoid()* function. Also, there is code inside of the *tcp_cwnd_down* function which prints a message with the new value for the congestion window, each time it is decreased.

5. Testing and evaluating GridDT

This chapter presents a series of comparative tests performed with GridDT and FAST.

5.1 The testbed

The testbed used while testing GridDT resulted as part of the DataTag Project. The main objective of this project was the creation of a large-scale intercontinental Grid testbed involving the European DataGrid project, several national projects in Europe, and related Grid projects in the USA.

The topology of the testbed is presented in Figure 5 below:

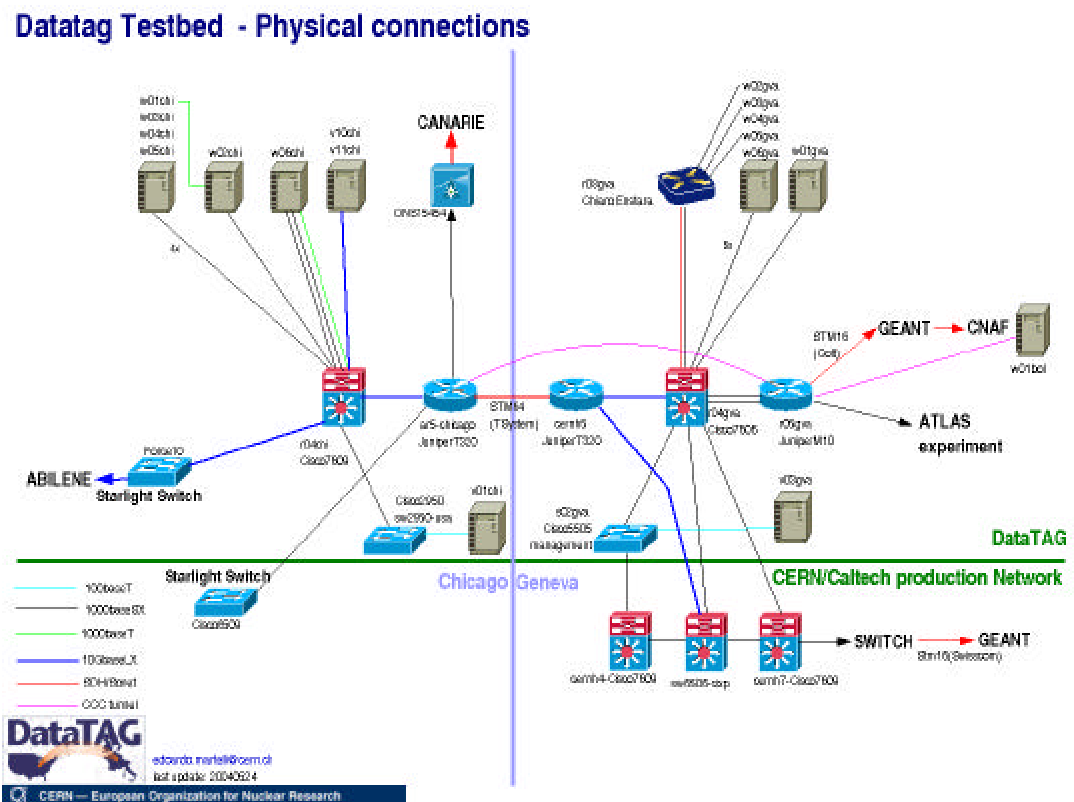

Figure 5: The Datatag testbed

The testbed is based on a high speed transatlantic link (STM-16, 2.5Gbps), with termination points at CERN in Geneva and Chicago (Starlight). This link is dedicated exclusively to network research and intensive data access applications.

The workstations connected to the testbed have 1Gbps network interfaces.

5.2 Testing strategy

During testing GridDT the *iperf* tool was used in order to generate traffic. *Iperf* is a simple traffic generator, with an architecture based on a client-server model, where the client generates traffic and the server receives it. The *tcpdump* tool was used to capture traffic and based on the output of tcpdump, *tcptrace* was used in order to generate the files needed by the graphing tool *xplot*.

Iperf allows the user to set the size of the TCP window. For the testing sessions a size of 128MB was chosen (larger than actually needed).

Some TCP related parameters were also modified, by using the sysctl interface:

```
echo "4096 87380 128388607" > /proc/sys/net/ipv4/tcp_rmem
echo "4096 65530 128388607" > /proc/sys/net/ipv4/tcp_wmem
echo 128388607 > /proc/sys/net/core/wmem_max
echo 128388607 > /proc/sys/net/core/rmem_max
```

tcp_rmem and tcp_wmem are used to set the minimum, default and maximum size of the TCP receive/transmit window . Rmem_max and wmem_max are used to specify the maximum values for the TCP receive/transmit window. For testing GridDT, we only modify the maximum values, as Iperf can set the window size based on a command line parameter.

Also, in order to be able to use large windows and avoid local congestion, the size of the transmit queue was increased to 10000:

```
/sbin/ifconfig eth1 txqueuelen 10000
```

For FAST, the parameters mentioned above have been modified by the tuning script that is part of the FAST distribution. The FAST alpha, beta and gamma parameters have also been modified for some of the tests. The FAST alpha parameter represents the number of packets to be buffered inside the network at equilibrium. The beta parameter represents the maximum number of packets buffered before a FAST TCP flow needs to reduce its window. The gamma parameter is used to monitor the packets buffered during slow start in order to avoid packet loss.

Three series of tests were ran:

- a throughput test
- a test with multiple concurrent flows
- a test where packets were dropped, in order to test how well does the tested protocol handle loss

For these tests, GridDT version 3.1 (without DCR) for Linux 2.6.5 was used. For the DCR version, tests have been conducted separately.

5.3 Actual testing

5.3.1 Throughput tests

The first test in this series measured standard TCP's performance as concerns throughput. A throughput test has been run for all the tested protocols, and the results were compared with those obtained by standard TCP.

The purpose of these tests was to show how each of the tested protocols manages to use high bandwidths, and how fast can it reach maximum bandwidth utilization. Two types of tests have been used, one that ran for 10 seconds and the other for 30 seconds. Tests have also been ran for longer periods (180, 300 seconds), but it has been concluded that a 30 seconds test is relevant as concerns a protocol's ability to use high bandwidth.

The first protocol tested was standard TCP, and the results were used as a baseline for evaluating the performance of FAST and GridDT.

The results for the 10-seconds as output by the iperf tool were:

[ID] Interval Transfer Bandwidth
[5] 0.0- 5.0 sec 457 MBytes 767 Mbits/sec
[5] 5.0-10.0 sec 562 MBytes 943 Mbits/sec
Average throughput:
[6] 0.0-10.5 sec 1019 MBytes 812 Mbits/sec

The graph generated based on these results is presented in Figure 6:

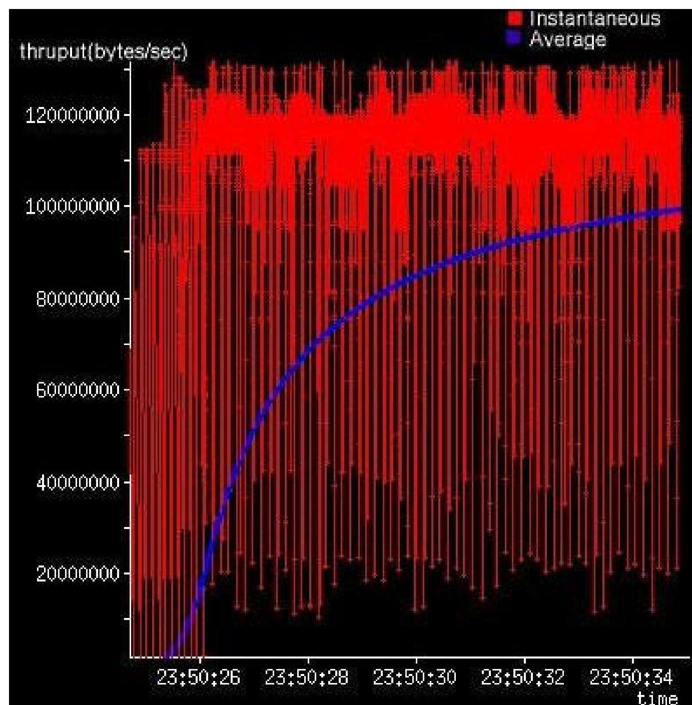

Figure 6: TCP, 10-seconds throughput test

For the 30-seconds test, the results are as follows:

```
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0- 5.0 sec   462 MBytes   776 Mbits/sec
[  5]  5.0-10.0 sec   586 MBytes   984 Mbits/sec
[  5] 10.0-15.0 sec   564 MBytes   946 Mbits/sec
[  5] 15.0-20.0 sec   549 MBytes   922 Mbits/sec
[  5] 20.0-25.0 sec   550 MBytes   923 Mbits/sec
[  5] 25.0-30.0 sec   549 MBytes   921 Mbits/sec
Average throughput:
[  5]  0.0-30.6 sec  3.18 GBytes   893 Mbits/sec
```

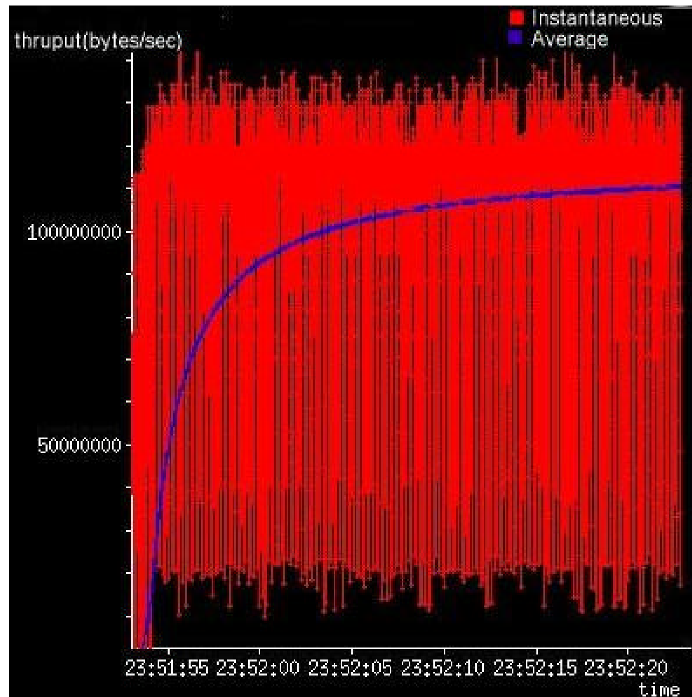

Figure 7: TCP, 30-seconds throughput test

From the results presented above it can be concluded that TCP's performance up to the 1Gbps limit is fair, as it managed to obtain a mean throughput of 893Mbps over 30 seconds and reached the upper limit of its performance in the first 10 seconds of the test (984Mbps).

The next protocol tested was FAST. The results for the 10-seconds test and the 30-seconds test are:

10-seconds test:

```
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0- 5.0 sec  50.7 MBytes  85.0 Mbits/sec
[  5]  5.0-10.0 sec   150 MBytes   252 Mbits/sec
```

The average throughput:
[ID] Interval Transfer Bandwidth
[6] 0.0-10.4 sec 201 MBytes 162 Mbits/sec

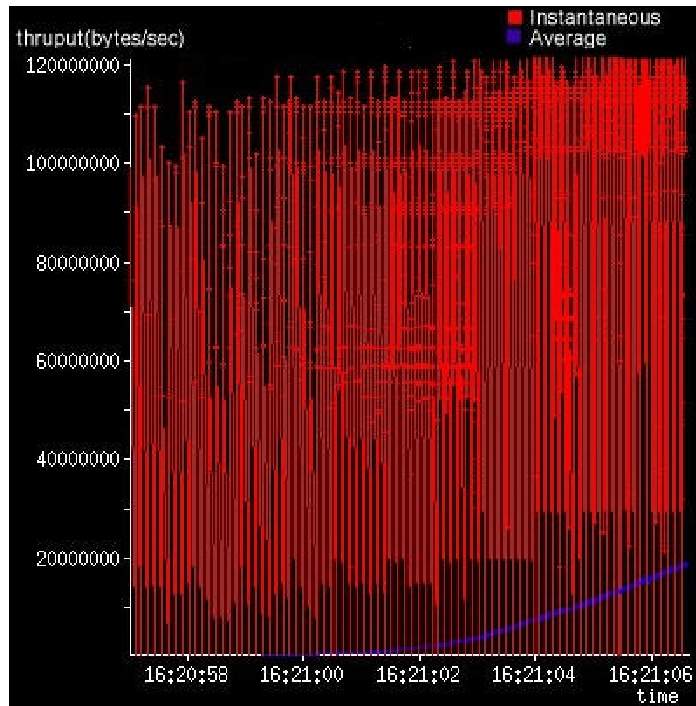

Figure 8: FAST, 10-seconds throughput test

For the 30 seconds test:
[ID] Interval Transfer Bandwidth
[5] 0.0- 5.0 sec 50.7 MBytes 85.0 Mbits/sec
[5] 5.0-10.0 sec 168 MBytes 282 Mbits/sec
[5] 10.0-15.0 sec 435 MBytes 730 Mbits/sec
[5] 15.0-20.0 sec 545 MBytes 914 Mbits/sec
[5] 20.0-25.0 sec 570 MBytes 956 Mbits/sec
[5] 25.0-30.0 sec 559 MBytes 938 Mbits/sec

Mean throughput:
[ID] Interval Transfer Bandwidth
[6] 0.0-30.3 sec 2.27 GBytes 645 Mbits/sec

As it can be easily noticed, FAST increments the congestion window cautiously in the slow-start phase, just like Limited Slow Start. This is why it reaches throughput greater than 900Mbps only after 15 seconds. This cautious approach to increasing the congestion window is very useful in networks with high bandwidth delay products, where during slow start, the congestion window can be increased by thousands of segments in a single RTT. In such cases it is possible to have a large number of packets dropped in a single RTT, which drastically limits TCP's performance.

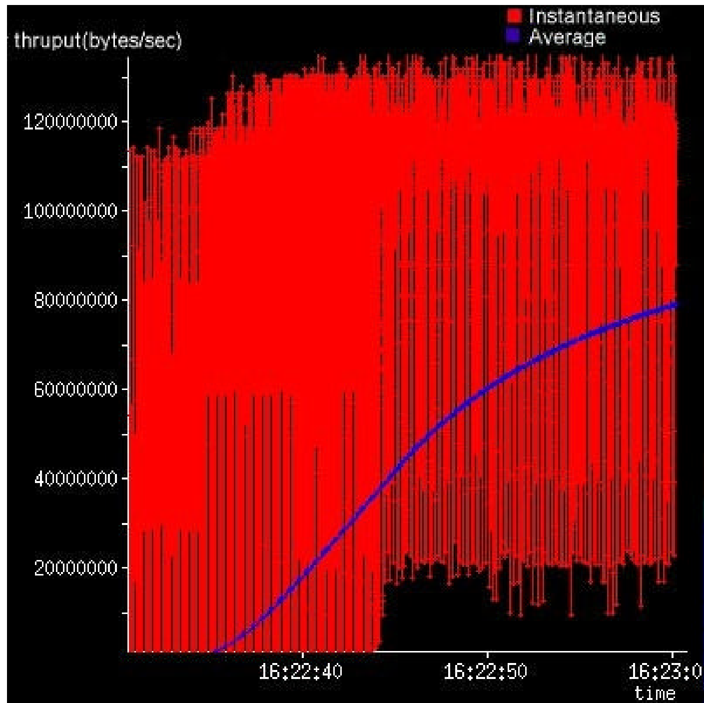

Figure 9: FAST, 30-seconds throughput test

FAST's performance is very good after the throughput reaches 900Mbps, as illustrated by the iperf output for a 60 seconds test:

```
[  5] local 192.91.239.4 port 32812 connected with 192.91.239.34 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0- 5.0 sec  50.7 MBytes  85.0 Mbits/sec
[  5]  5.0-10.0 sec   149 MBytes   250 Mbits/sec
[  5] 10.0-15.0 sec   416 MBytes   698 Mbits/sec
[  5] 15.0-20.0 sec   553 MBytes   928 Mbits/sec
[  5] 20.0-25.0 sec   572 MBytes   959 Mbits/sec
[  5] 25.0-30.0 sec   544 MBytes   913 Mbits/sec
[  5] 30.0-35.0 sec   570 MBytes   956 Mbits/sec
[  5] 35.0-40.0 sec   561 MBytes   941 Mbits/sec
[  5] 40.0-45.0 sec   554 MBytes   929 Mbits/sec
[  5] 45.0-50.0 sec   573 MBytes   961 Mbits/sec
[  5] 50.0-55.0 sec   551 MBytes   925 Mbits/sec
[  5] 55.0-60.0 sec   561 MBytes   941 Mbits/sec
```

The average throughput achieved during this 60-seconds test was 787Mbps.

Considering the fact that FAST is designed for very large file transfer over large BDP networks(transfers which usually take a long time to complete, even when available bandwidth is used efficiently), the fact that it needs 15 seconds to achieve 900Mbps throughput is not such a drastic limitation.

Also, by carefully tuning the FAST parameters, even better results can be obtained. In the example below, the FAST parameter alpha was set to 500 (default 200), and the beta parameter to 512 (default 212):

```
[  5] local 192.91.239.4 port 32813 connected with 192.91.239.34 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0- 5.0 sec  50.7 MBytes  85.0 Mbits/sec
[  5]  5.0-10.0 sec   286 MBytes   481 Mbits/sec
[  5] 10.0-15.0 sec   564 MBytes   946 Mbits/sec
```

The average throughput:

```
[  6]  0.0-60.8 sec  5.81 GBytes   821 Mbits/sec
```

As it can be observed, a faster increase to more than 90% bandwidth utilization was achieved. Also, an average throughput of 821Mbps was obtained during the 60-second test.

The last protocol tested was GridDT. The first test was run using the following parameters:

- *Sysctl_tcp_cwnd_decr* was set to 8, resulting in a window decrease factor of 1/8
- *Sysctl_tcp_rtt_ref* was set to 50
- The default value of 100 was kept for *max_ssthresh*

The results for the 10-seconds test, as output by iperf are:

```
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0- 5.0 sec   136 MBytes   229 Mbits/sec
[  5]  5.0-10.0 sec   147 MBytes   246 Mbits/sec
[  5]  0.0-12.7 sec   283 MBytes   187 Mbits/sec
```

The average throughput during this test:

```
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-12.5 sec   279 MBytes   188 Mbits/sec
```

It is obvious that the default value of max_ssthresh is too conservative. In order to determine the optimum value for *max_ssthresh,* another test was performed. This test set *max_ssthresh* to a very large value(a value highly unlikely to be reached during a 1Gbps test), in order to determine the value of the congestion window at which congestion occurs. Figure 10 presents the evolution of the congestion window during this test:

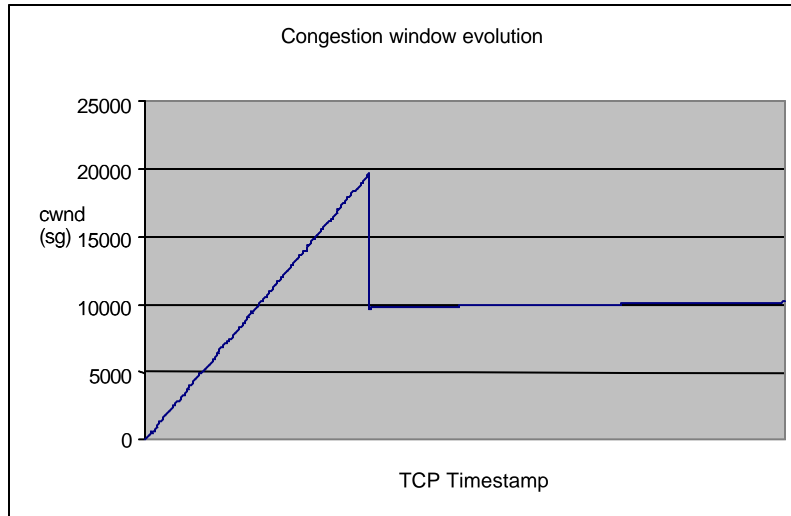

Figure 10: Congestion window evolution, GridDT

It is easily observed that congestion occurs somewhere around the 20000 mark. Considering this, a value of 8000 for *max_ssthresh* was suggested for this link (1Gbps, 120ms RTT). This value will insure both desiderates: a relatively rapid increase to full bandwidth utilization, and a more cautious increase towards the end of slow-start. With this value, limited slow-start will be much less aggressive as slow start after *max_ssthresh* is reached (the maximum increase of the congestion window will be ½ per ACK received).
Another 10-seconds test has been ran with max_sstrhesh set to 8000, with much better results.

```
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0- 5.0 sec   472 MBytes   791 Mbits/sec
[  5]  5.0-10.0 sec   557 MBytes   935 Mbits/sec
[  5]  0.0-10.8 sec  1.00 GBytes   796 Mbits/sec
```

The average throughput:

```
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-10.7 sec  1.00 GBytes   804 Mbits/sec
```

The following test is a 30-seconds test for GridDT, with max_sshthresh set to 8000.
The results as output by iperf were:

```
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0- 5.0 sec   449 MBytes   753 Mbits/sec
[  5]  5.0-10.0 sec   588 MBytes   987 Mbits/sec
[  5] 10.0-15.0 sec   561 MBytes   942 Mbits/sec
[  5] 15.0-20.0 sec   556 MBytes   933 Mbits/sec
[  5] 20.0-25.0 sec   557 MBytes   935 Mbits/sec
[  5] 25.0-30.0 sec   563 MBytes   944 Mbits/sec
```

The average throughput:
[ID] Interval   Transfer   Bandwidth
[6] 0.0-30.7 sec 3.20 GBytes   894 Mbits/sec

As it can be observed, the increase in bandwidth utilization is more reserved during the first 5 seconds, due to limited slow start, but the overall results show good bandwidth utilization.

Fig. 11 below presents the evolution of the average throughput over the 30-seconds test.

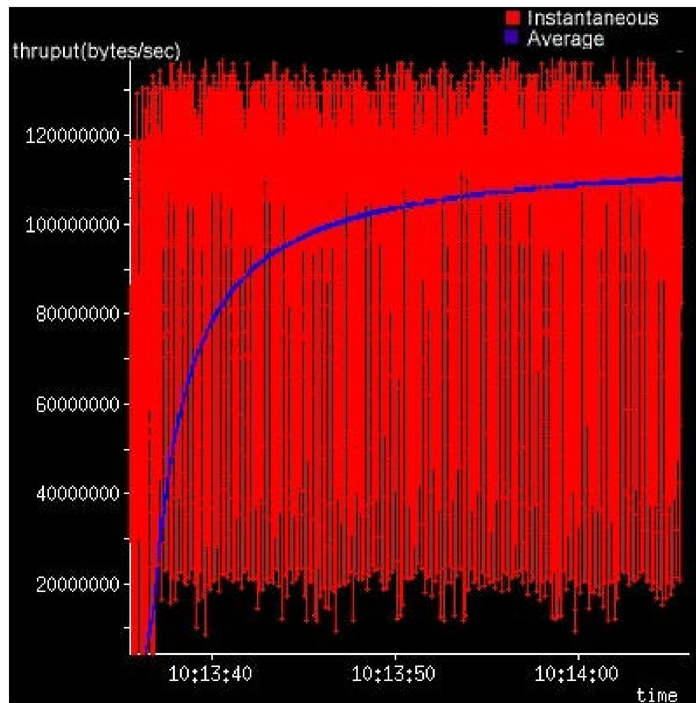

Figure 11: GridDT, 30-seconds throughput test

5.3.2 Concurrent flows test

This series of tests evaluated each protocol's behavior when multiple concurrent flows are competing for bandwidth.

The first protocol tested was the standard TCP. A test with 3 concurrent flows was ran for 30 seconds. The results as output by iperf follow:

[7] 0.0-30.5 sec 1.81 GBytes   511 Mbits/sec
[8] 0.0-30.5 sec 1.26 GBytes   355 Mbits/sec
[6] 0.0-59.5 sec 33.5 MBytes 4.72 Mbits/sec

[SUM] 0.0-59.5 sec 3.11 GBytes   448 Mbits/sec

It can be observed that the first flow started is heavily affected when the other 2 flows start to increase their bandwidth usage, and the first flow never recovers.

The results are also illustrated graphically below Fig. 12.

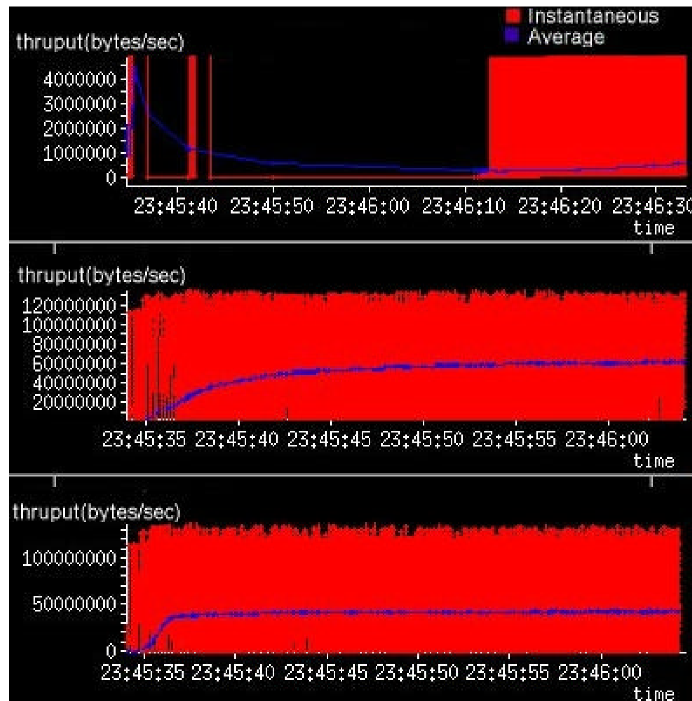

Figure 12: TCP, 30-seconds test, 3 concurrent flows

FAST was the last protocol tested during this test. The iperf output is presented below:

[ID] Interval   Transfer   Bandwidth
[6] 0.0-30.8 sec   936 MBytes   255 Mbits/sec
[8] 0.0-30.6 sec   919 MBytes   252 Mbits/sec
[7] 0.0-30.8 sec 1.11 GBytes   309 Mbits/sec
[SUM] 0.0-30.9 sec 2.92 GBytes   812 Mbits/sec

The graphics generated based on these results are presented in Fig. 13.

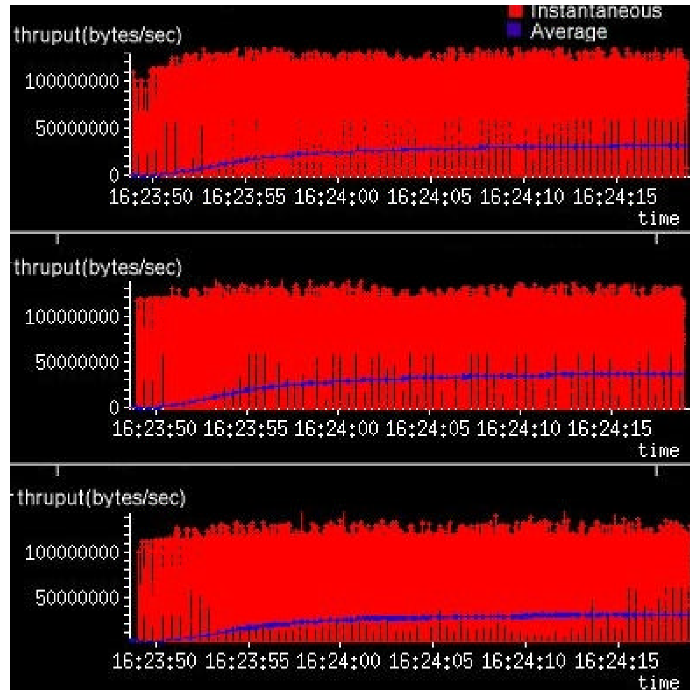

Figure 13: FAST, 30-seconds test, 3 concurrent flows

As it can be easily noticed from the results, FAST exhibited excellent behavior during this test. All three flows got to use bandwidth, and their increase in bandwidth utilization was uniform, with none of the flows starving for bandwidth at any time.

GridDT was also tested, and showed only slightly better results than TCP, due to its more aggressive behavior in the congestion avoidance phase. However, the is still room for improving GridDT in this respect.

Based on these results, it can be concluded that FAST showed the best results during the concurrent flow test.

5.3.3 Packet loss test

This series of tests evaluated the ability to recover from loss for the tested protocols. The way TCP handles loss, together with its slow increase of the congestion window during the congestion avoidance phase are the main factors that contribute to TCP's unsatisfactory performance in networks with a high bandwidth-delay product.

During these tests, packets were dropped using iptables in order to simulate packet loss.

The first protocol tested was standard TCP. Traffic was dropped after 10 seconds (after bandwidth utilization reached 900Mbps) for half a second. The results, as output by iperf are:

[ID] Interval   Transfer   Bandwidth
[5] 0.0- 5.0 sec   483 MBytes   811 Mbits/sec
[5] 5.0-10.0 sec   560 MBytes   939 Mbits/sec
[5] 10.0-15.0 sec   135 MBytes   227 Mbits/sec
[5] 15.0-20.0 sec   291 MBytes   488 Mbits/sec
[5] 20.0-25.0 sec   288 MBytes   484 Mbits/sec
[5] 25.0-30.0 sec   311 MBytes   521 Mbits/sec

The average throughput:
[ID] Interval   Transfer   Bandwidth
[6] 0.0-31.3 sec 2.02 GBytes   554 Mbits/sec

These results are illustrated in Fig. 14.

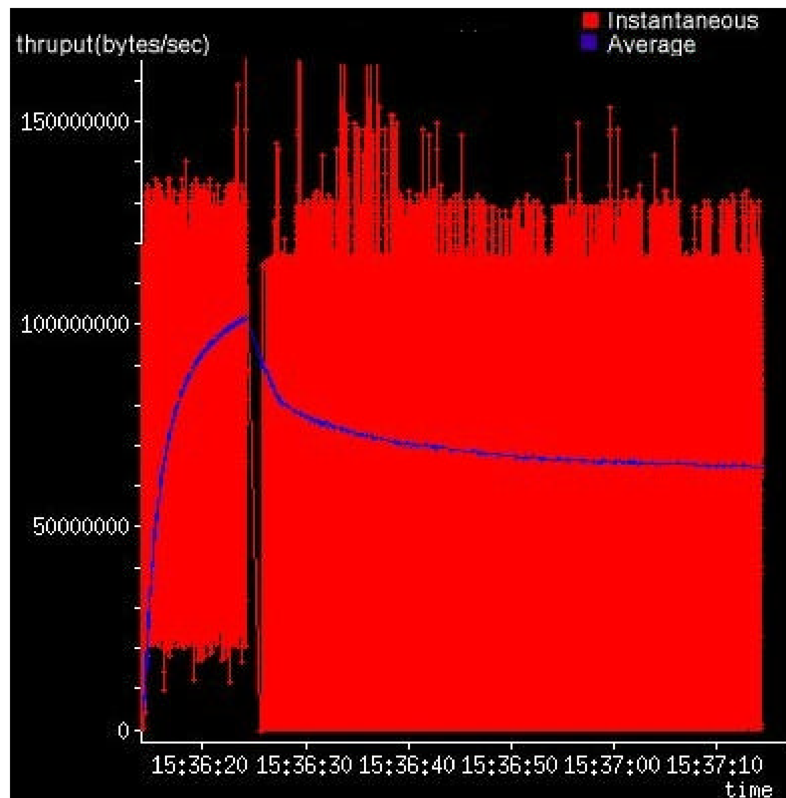

Figure 14: TCP, 30-seconds test, packet dropping for 0,5s

It can be noticed that TCP never recovered to the bandwidth utilization that it reached before the packet drop. The same test was run for 120 seconds, and the graph in Figure 15 was generated, based on throughput values obtained by sampling at every 5 seconds.

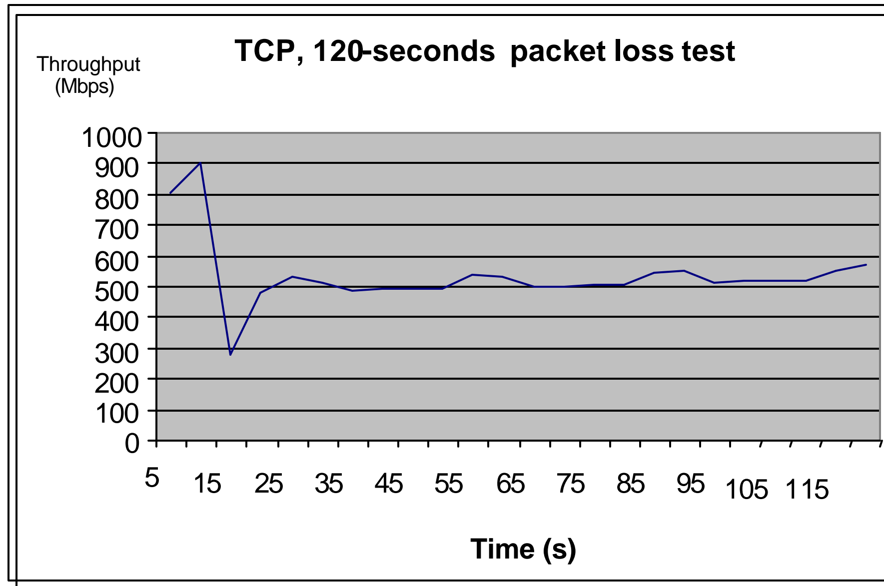

Figure 15: TCP, 120-seconds test, dropping packets for 0.5s

This graph shows that even after 120 seconds, only a small increase in throughput is achieved. From these results, it can be concluded that standard TCP has poor performance when packets are dropped, as it takes a very long time to recover from loss.

The next protocol to be tested during this series was FAST. The same 30-seconds test was performed, but dropping started after 20 seconds (when bandwidth utilization exceeded 900Mbps). The results as output by iperf are presented below:

```
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0- 5.0 sec  50.7 MBytes  85.0 Mbits/sec
[  5]  5.0-10.0 sec   147 MBytes   247 Mbits/sec
[  5] 10.0-15.0 sec   445 MBytes   747 Mbits/sec
[  5] 15.0-20.0 sec   551 MBytes   924 Mbits/sec
[  5] 20.0-25.0 sec  81.3 MBytes   136 Mbits/sec
[  5] 25.0-30.0 sec   260 MBytes   437 Mbits/sec
```

The average throughput:

```
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-30.5 sec  1.50 GBytes   423 Mbits/sec
```

Since dropping packets started after 20 seconds, this 30-seconds test is not entirely relevant. A 60-seconds was conducted in order to see how FAST recovers from packet loss. Based on throughput measurements taken every 5 seconds, the graph in Fig. 16 has been drawn.

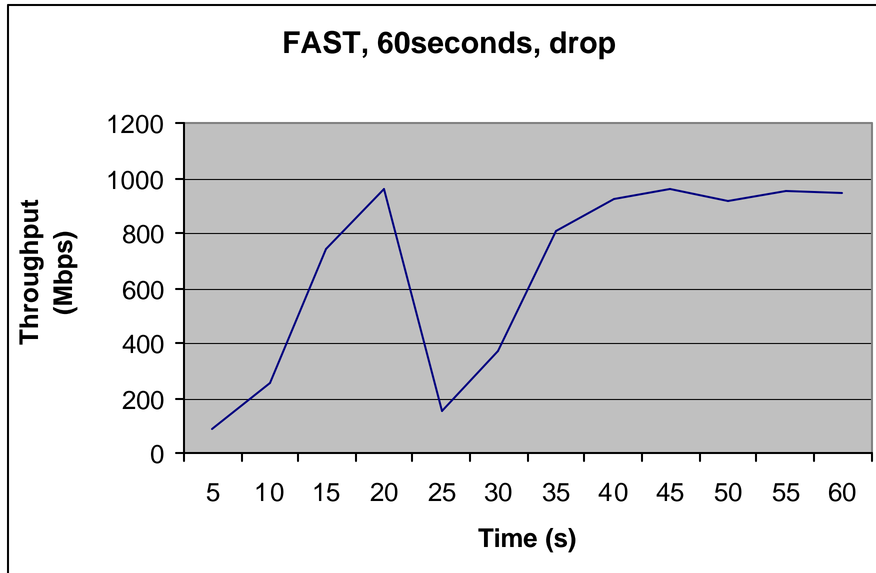

Figure 16: FAST, 60-seconds test, dropping packets for 0.5s

By examining this graph, it can be noticed that FAST recovers very well from packet loss, as bandwidth utilization over 900Mbps is reached only 20 seconds after packet loss is encountered. Also, FAST shows good stability, as bandwidth utilization over 90% is maintained after recovering from packet loss.

The last protocol tested was GridDT. The first test ran for 30 seconds, and packets were dropped for half a second, after bandwidth utilization over 900Mbps was reached. The results as output by iperf were:

[5] 0.0- 5.0 sec   458 MBytes   768 Mbits/sec
[5] 5.0-10.0 sec   565 MBytes   948 Mbits/sec
[5] 10.0-15.0 sec   176 Bytes   295 Mbits/sec
[5] 15.0-20.0 sec   568 MBytes   952 Mbits/sec
[5] 20.0-25.0 sec   567 MBytes   951 Mbits/sec
[5] 25.0-30.0 sec   561 MBytes   941 Mbits/sec
[5] 0.0-30.8 sec 2.83 GBytes   788 Mbits/sec

The average throughput achieved:
[ID] Interval   Transfer   Bandwidth
[6] 0.0-30.7 sec 2.83 GBytes   791 Mbits/sec

The same results are illustrated graphically in Fig. 17 below.

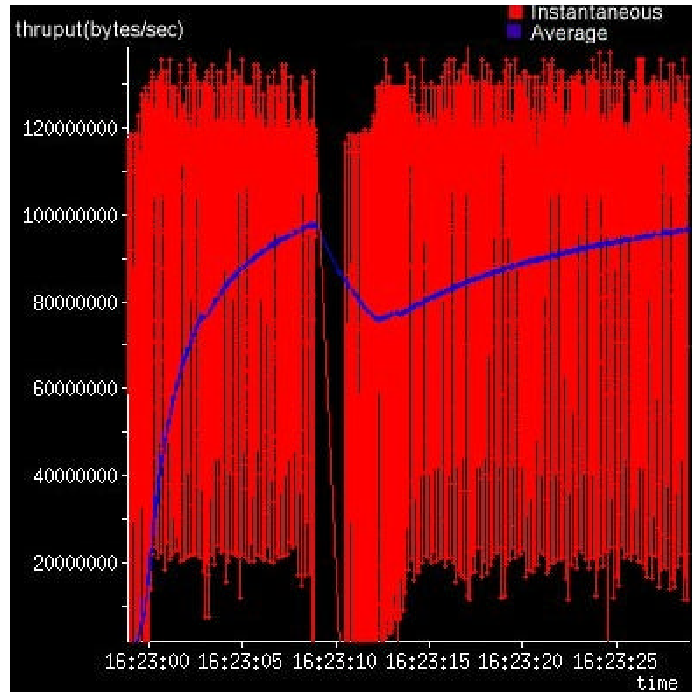

Figure 17: GridDT, 30-seconds test, packets dropped for 0.5s

GridDT showed excellent results, recovering from loss in only 10 seconds. Fig. 18 presents the evolution of the congestion window during this test:

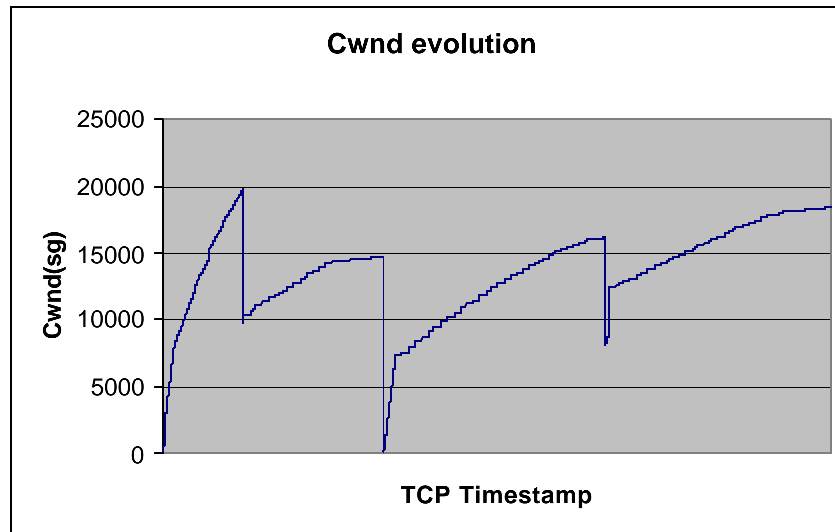

Figure 18: Congestion window evolution, GridDT packet loss test

It can be easily observed that loss indicated by a retransmission timeout expiration has taken place, and GridDT reacted to loss by setting the congestion window to 1.

A second test was ran, that lasted 60 seconds. Packets were dropped after 10 seconds for an interval of 3 seconds. Fig. 19 illustrates the evolution of throughput during this test.

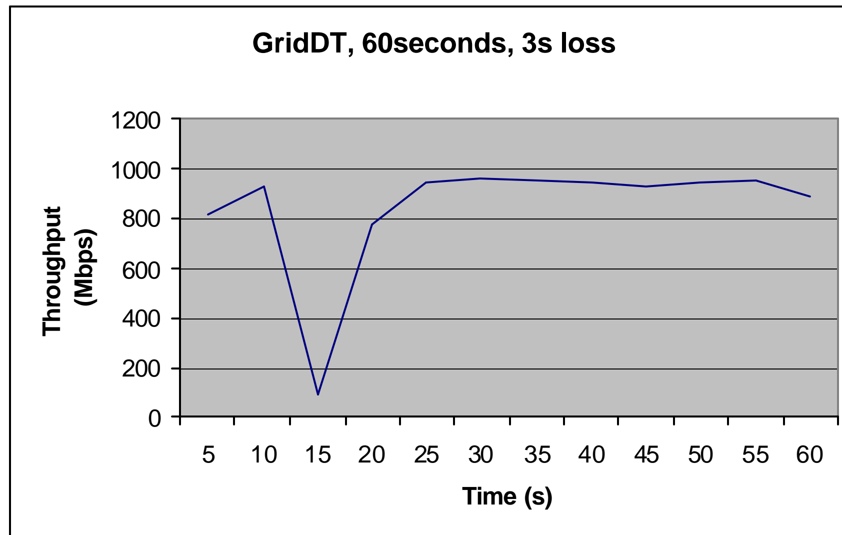

Figure 19: GridDT packet loss, 60-seconds test, dropping packets for 3s

Based on these results, it can be concluded that both GridDT and FAST exhibited excellent results as concerns the capability to recover from packet loss.

5.3.4 Testing DCR

In order to test the implementation of the Delayed Congestion Response mechanism, a tool called *dummynet* was used. Dummynet is a flexible tool designed for testing protocols, which can simulate queue and bandwidth limitations, delays, packet loss, and multipath effects. Dummynet is used on a host running FreeBSD, and it works by intercepting packets in their way through the protocol stack and passing them through structures called queues and pipes. Pipes are fixed-bandwidth and fixed-delay channels. Queues are queues of packets, associated with a weight, which share the bandwidth of the pipe they are connected to proportionally with their weight. In order to define the rules which are used to decide which packets will be intercepted, ipfw (a tool used to configure FreeBSD firewall support) is used.

During the tests performed with GridDT, a multipath effect was generated which in turn lead to packet reordering. Two hosts running Linux were interconnected with a FreeBSD host operating as a bridge and running dummynet. The host that generated the traffic used Linux-2.4.26-GridDT with DCR.

For the test three pipes were created, associated with different probabilities. The different delays associated with each of the pipes insured a multipath effect and hence packet reordering. Iperf was used to generate traffic for 30 seconds. The configuration used during this test was the following:

```
ipfw add prob 0.3 pipe 1 ip from 192.168.2.2 to 192.168.2.1
ipfw add prob 0.5 pipe 2 ip from 192.168.2.2 to 192.168.2.1
ipfw add pipe 3 ip from 192.168.2.2 to 192.168.2.1

ipfw pipe 1 config bandwidth 30Mbps delay 100ms
ipfw pipe 2 config bandwidth 50Mbps delay 80ms
ipfw pipe 3 config bandwidth 20Mbps delay 85ms
```

The results obtained without DCR:

```
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0- 5.0 sec  5.67 MBytes  9.51 Mbits/sec
[  6]  5.0-10.0 sec  12.0 MBytes  20.2 Mbits/sec
[  6] 10.0-15.0 sec  16.2 MBytes  27.2 Mbits/sec
[  6] 15.0-20.0 sec  19.6 MBytes  32.8 Mbits/sec
[  6] 20.0-25.0 sec  23.2 MBytes  39.0 Mbits/sec
[  6] 25.0-30.0 sec  26.2 MBytes  44.0 Mbits/sec
[  6]  0.0-30.3 sec   105 MBytes  29.0 Mbits/sec
```

When DCR was used, a small increase in bandwidth utilization was achieved:

```
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0- 5.0 sec  5.53 MBytes  9.29 Mbits/sec
[  6]  5.0-10.0 sec  14.6 MBytes  24.5 Mbits/sec
[  6] 10.0-15.0 sec  19.3 MBytes  32.4 Mbits/sec
[  6] 15.0-20.0 sec  23.0 MBytes  38.5 Mbits/sec
[  6] 20.0-25.0 sec  24.7 MBytes  41.5 Mbits/sec
[  6] 25.0-30.0 sec  26.3 MBytes  44.0 Mbits/sec
[  6]  0.0-30.4 sec   116 MBytes  32.0 Mbits/sec
```

Based on these results, it can be concluded that DCR brings some performance gain when packet reordering is present, but this increase in performance is not spectacular. Tests have been also conducted with DCR enabled but no packet reordering, and it has been concluded that when packet reordering is not present, DCR does not negatively affect performance.

6. Conclusions

During the testing session presented in Chapter 5, valuable information as regards the performance of standard TCP, FAST TCP and GridDT when used in high bandwidth-delay product networks has been gathered. The main goal of this chapter is to present conclusions reached after examining the test results and to evaluate how does the implementation of GridDT meet GridDT's design goals. Conclusions regarding the performance of FAST TCP are also presented.

As regards the TCP tests, they have only been used as a baseline used in order to present improvements brought by GridDT and FAST. The results obtained showed problems that were already known: slow increase in the congestion window during congestion avoidance, a decrease that is too drastic when congestion is encountered, poor performance in recovering from packet loss.

Based on the results obtained while testing FAST TCP it can be concluded that FAST successfully addresses the limitations of the standard TCP. FAST has also shown the best results in the concurrent flows test, where TCP and GridDT did not perform very well.
FAST also employs a mechanism similar to Limited Slow Start proposed by RFC 3742 and implemented in GridDT, which can be very useful in high bandwidth-delay networks. Very good results have also been obtained when recovering from packet loss. It can be concluded based on these arguments that FAST has accomplished its design goals.

As concerns GridDT, valuable conclusions have been drawn based on the results of the tests. First of all, it has been showed that a value of 100 for the max_ssthresh parameter used during Limited Slow Start, is too conservative. This has implications related to the time it takes GridDT to reach full bandwidth utilization when starting up or when ramping up from periods when the congestion window is small (after loss is encountered). For the links used in the testbed (1Gbps, 120ms RTT), a value of 8000 has been suggested for max_ssthresh. With this value good performance when recovering from loss was obtained and the goal of Limited Slow Start was still achieved: the increase of the congestion window after max_ssthresh is less aggressive than with slow-start.
GridDT has showed the best results when recovering from loss, due to the modified congestion window update algorithm.
DCR has also showed a small increase in performance, but not spectacular. Tests have also been ran with DCR in the lack of packet reordering, and it has been showed that DCR does not negatively affect performance in the absence of packet reordering.

Based on the results obtained during testing it can be stated that the GridDT implementation has successfully accomplished the design goals:

- achieve high per-connection throughput
- Implement Limited Slow Start

- reach high throughput without overly long delays when recovering from multiple retransmit timeouts, or when ramping-up from a period with small congestion windows.
- No additional feedback or support required from routers
- No additional feedback required from TCP receivers

There is however room for improvement, as the results obtained during the multiple concurrent flows test were not nearly as good as the results obtained by FAST TCP. Future efforts made in order to improve GridDT should be aimed towards improving the protocol's behavior in this respect.

Bibliography

[1] W. Richard Stevens , "TCP/IP Illustrated, Volume I", Addison Wesley, 1994

[2] K. Fall, S. Floyd, "Simulation Based Comparisons of Tahoe, Reno & SACK TCP", ACM SIGCOMM Computer Communication Review, Volume 26, Issue 3, 1996

[3] W. Stevens, "RFC 2001: TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms", IETF Repository http://www.ietf.org/rfc/rfc2001.txt, January 1997

[4] M. Allman, V. Paxson, W. Stevens, "RFC 2581: TCP Congestion Control", IETF Repository, http://www.ietf.org/rfc/rfc2581.txt, April 1999

[5] T. Kelly, "STCP: Improving performance in High Speed Wide Area Networks", http://www-lce.eng.cam.ac.uk/~ctk21/scalable, 2002

[6] P. Sarolahti, A. Kuznetsov, "Congestion Control in Linux TCP", http://www.cs.helsinki.fi/research/iwtcp/papers/linuxtcp.pdf, 2002

[7] S. Bhandarkar, A.L. Narasimha Reddy "Improving the robustness of TCP to Non-Congestion Events", IETF draft, http://www.ietf.org/internet-drafts/draft-tcpm-tcp-dcr-00.txt, October 2003

[8] S. Floyd, "RFC 3649: HighSpeed TCP for Large Congestion Windows", IETF Repository, http://www.ietf.org/rfc/rfc3649.txt, December 2003

[9] Cheng Jin, David X. Wei and Steven H. Low, "FAST TCP: Motivation, Architecture, Algorithms, Performance", IEEE Infocom, Hong Kong, March 2004

[10] S. Floyd, "RFC 3742: Limited Slow Start for TCP with Large Congestion Windows", IETF Repository, http://www.ietf.org/rfc/rfc3742.txt, March 2004

[11] R. Shorten, D.J. Leith, "H-TCP: TCP for high-speed and long-distance networks", http://www.hamilton.ie/net, 2004

[12] D.J. Leith, "Linux TCP Implementation Issues in High-Speed Networks", http://www.hamilton.ie/net/LinuxHighSpeed.pdf