Achieving Reliable High Performance in LFNs

Sven Ubik, <ubik@cesnet.cz> and Pavel Cimbal, <P.Cimbal@sh.cvut.cz>
CESNET, Prague, Czech Republic

Abstract—

The PC hardware architecture and commodity operating systems such as Linux or Microsoft Windows are dominant in end-host infrastructure in the current Internet. It shows that many performance problems in high-speed long-distance networks (also called long-fat networks or LFNs due to high possible volume of outstanding data between the sender and receiver) are caused by improper behaviour of end hosts, rather than by the network itself. Understanding of the behaviour of networking support in commodity operating systems, networking applications, their mutual interaction and their interaction with a computer network is necessary to identify and resolve these problems. Our goal is to identify the most significant problems and suggest possible solutions. Many advanced Internet users, requiring high network performance, use the Linux operating system. Therefore, we decided to concentrate on networking support in Linux. Particularly, we will show that just setting large TCP buffers and modifying TCP congestion control AIMD parameters is not sufficient to reliably achieve high throughput. We will show that smart window size moderation is required not to allow too large TCP windows. We will also describe an important performance problem in scp, a commonly used network file copy program.

I. Introduction

More than 95% of data transferred over the Internet is currently carried in TCP protocol. Therefore, performance of TCP is essential for overall Internet performance. The current state of networking support in all commodity operating systems after the default installation procedure and system boot is that they perform very poorly over LFNs.

TCP uses two mechanisms to prevent congestion of receiver and network [7]. First, the TCP sender uses flow control, driven by the receiver-advertised window (`rwin`) to limit the sending speed so that the receiver can keep up with incoming packets. Second, the TCP sender uses congestion control, driven by its computed congestion window (`cwnd`) computed based on congestion signals to limit the sending speed so that the network is not congested.

Both windows (`rwin` and `cwnd`) can extend only up to the size of the TCP buffer on the respective side of the connection. The default size of sender and receiver TCP buffers in most commodity operating systems is 64 kB. This is insufficient for LFNs, which require high volume of outstanding data to achieve high performance.

In most cases, increasing the size of TCP buffers helps to increase throughput. Several tools have been developed in the form of kernel patches or daemons to help find buffer sizes required for particular network conditions [1], [2], [9]. However, even with these tools proper setting of TCP buffers is a procedure requiring a lot of manual installation and configuration. Therefore, automatic tuning should be implemented by default in future versions of commodity operating systems.

However, this tuning should not only save memory by not allocating large buffers by default for all connections and care about TCP windows to be large enough. It should also dynamically moderate receiver or sender window so that the transmission does not unnecesarily fill router queues, thus increasing probability of packet loss and decrease of throughput. We will discuss this requirement in section III.

It show that the standard TCP congestion avoidance based on AIMD(1, 0.5) algorithm is too slow for LFNs. There are proposals to modify TCP congestion avoidance by adjusting its aggresiveness and responsiveness according to the outstanding TCP window size [3] or to use another congestion control algorithm [4]. However, the problem of filling up router queues is inherent to AIMD algorithm, which is reactive by nature. We believe that higher performance can be achieved by combination of AIMD with window moderation, as discussed in section III.

II. Linux TCP performance

It appears that TCP implementation in some commodity operating systems include their own modifications with respect to the specification in RFCs. These

modifications should be considered when using these operating systems for performance tests and when studying new mechanisms designed for the use in public Internet. Influence of these modifications on network performance can probably be stronger that various subtle improvements to TCP congestion control proposed in literature.

As we mentioned, we concentrate in this paper on the Linux operating system. Linux kernel 2.2 behaved closely to the specification in RFCs. Beginning with the Linux kernel 2.4 a lot of differences from the specification have been introduced. Most differences are very poorly documented only within the kernel source code. In this section we describe the most important features specific to Linux that influence TCP performance.

A. Sizes of TCP buffers

Actual sizes of sending and receiving TCP buffers are different from values supplied to `setsockopt()` call with `SO_SNDBUF` and `SO_RCVBUF` socket options as well as from values reported by `getsockopt()` call. This should be considered when setting buffer sizes. The internal arithmetics applied by Linux to buffer and winsow sizes is illustrated in Fig. 1. Parts in bold face indicate changes in Linux 2.4 to Linux 2.2. The values supplied to `setsockopt()` call are first multiplied by two and stored in internal variables. The content of these variables is returned by `getsockopt()` call. If `setsockopt()` was not called, system default values (see below) are copied to internal variables. In Linux 2.2 the content of these internal variables is again divided by two before setting the real buffer sizes. Division by two is probably legacy from old TCP implementations that used 16-bit signed variables for window arithmetics. And multiplication by two is probably legacy of patching the previous version to behave as expected. In Linux 2.4 division by two was removed. The real buffer sizes are therefore twice as specified by `setsockopt()`. When an application does not explicitely request buffer sizes by calling `setsockopt()`, the kernel uses heuristics to choose system default values based on `net/ipv4/tcp_[rw]mem` kernel variables and current memory consumption.

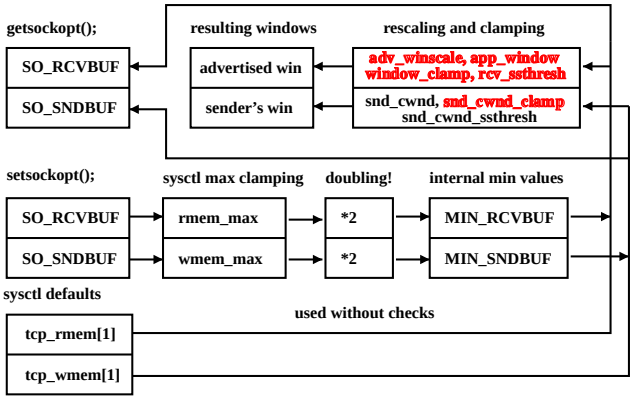

Fig. 1. Internal arithmetics applied by Linux to TCP buffer and window sizes

B. Application buffer clamping

In Linux 2.4, the advertised `rwin` is computed from the internal variable described in the previous section using `tcp_adv_winscale` kernel parameter as: internal variable $*(1-1/2^{tcp\_adv\_winscale})$. The remaining part of the receiving buffer is clamped as application buffer. The purpose is to smooth out advertised `rwin` by putting part of the received data waiting to be read by application out of the receiving buffer.

C. TCP parameter cache

Certain TCP runtime parameters such as `ssthresh` are cached for 10 minutes for individual destination IP addresses and used for subsequent connections to same destinations in an attempt to prevent slow start overshooting available bandwidth at the beginning of the connection. In reality, low `ssthresh` may be consequence of one connection going particularly bad due to temporary overflow of router buffers and may not reflect real available bandwidth of the particular network path. With low `ssthresh`, subsequent connections prematurely switch from slow start to congestion avoidance. The sender congestion window then increases very slowly, resulting in poor throughput for short connections. We can detect this effect by monitoring `ssthresh` and `cwnd` using the web100 kernel extension or by observing development of the sender outstanding window from the captured packet headers. As shown in Fig. 2, the outstanding window stops increasing sharply when `ssthresh` is reached. We can clear TCP cache with command `echo 1 > /proc/sys/net/ipv4/route/flush`.

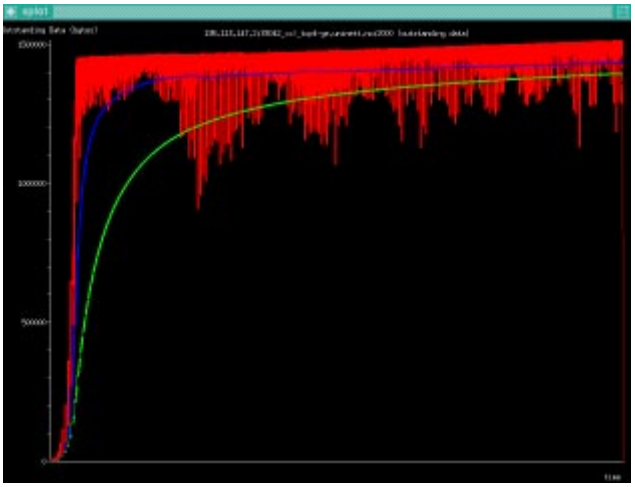

Fig. 2. Outstanding window development with low initial `ssthresh`

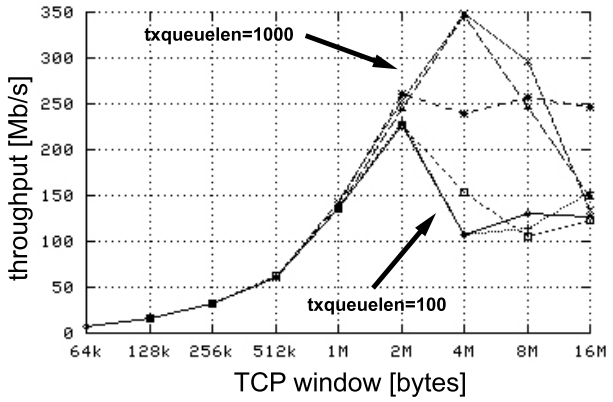

Fig. 3. Throughput for different TCP windows and txqueuelen

D. TxQueueLen

Fig. 3 illustrates achieved throughput over one of our testing network paths for different TCP window sizes. The connection was between two PCs with Gigabit Ethernet interfaces located in Cesnet and Uninett, going over 11 hops of the Géant network, consisting of Gigabit Ethernet and OC-48 circuits. The window sizes were the real ones after applying the internal arithmetics.

Obviously, larger windows resulted in higher throughput. However, beginning with 2 MB windows throughput started to decrease. This effect may have several reasons. One of the most significant factors is the txqueuelen parameter, which sets the number of packets that can be stored in the buffer between the operating system and the network adapter. When an application needs to send a large batch of data and when allowed by current TCP windows, this large batch of data is submitted to the network adapter queue to be sent to the network. The network adapter can send packets only at the speed of its physical interface, for example 1 Gb/s for Gigabit Ethernet. When this queue becomes full, the operating system context is switched to another process. The network adapter can continue sending packets as long as the network adapter buffer is not empty. After that it stops sending packets and waits until the context is switched back to the sending process. With the standard 100 Hz Linux system timer, a process can get context for up to 10 ms. If the network adapter buffer should supply data at 1 Gb/s speed for 10 ms in 1500-byte packets (high volumes of data are normally sent in large packets, so we do not need to consider smaller packets, 1500 bytes is the largest IP packet which can be carried in a standardized Ethernet frame) we must increase the buffer size to at least 1 Gb/s * 10 ms / 1500 bytes = 833 packets. As illustrated in Fig. 3, increasing txqueuelen to 1000 significantly increased throughput for windows set to 4 MB and 8 MB.

E. Runtime window moderation

During the connection, the kernel tries to moderate `rwin` advertised by the receiver as well as `cwnd` computed by the sender. The advertised `rwin` grows from a small initial value to the size of the receiving buffer (without the clamped application buffer). This growth is driven by data being received. If little data is received, `rwin` grows slowly. Once the window reaches its nominal value it stays there even if no packets are arriving unless there is a shortage of memory. The computed `cwnd` is moderated so that it does not grow too much over the outstanding window. The purpose of this moderation is to prevent large packet losses, which could be caused by a sudden burst coming after a period of little sender traffic, when `cwnd` was allowed to grow unimpeded and `rwin` was also high. However, this moderation is insufficient to prevent filling router queues, which can result in unnecessary packet drops. We will discuss window moderation requirements in more detail later.

F. Fast path, slow path

The kernel can process packets in two modes - fast path and slow path. If the connection is purely unidirectional, that is only pure ACKs are sent in one direction and data segment in the other direction, fast path is used. If the connection is bidirectional (one data segment sent in the other direction is sufficient), the kernel switches to slow path, which can influence performance.

III. Finding optimal window size

Traditional advice on configuring TCP sender and receiver buffers is that they should be large enough to allow TCP windows at least equal to the pipe capacity so that the connection is not constrained by small windows and can utilize the available bandwidth. We will show that there are several reasons why it is important not to allow too large TCP windows. We base our analysis mostly on observations of connection traces captured by tcpdump run on a separate PC connected to a switch port configured to mirror outgoing and incoming packets of the sending PC. We found that when tcpdump is run directly on the sending or receiving PC, it either loses lots of packets or it influences the monitored connection, depending on its process priority.

The question is what is the pipe capacity? It is the sum of the "wire pipe" capacity given by the product of available bandwidth and round-trip time of the empty network plus the amount of buffer memory of the TCP receiver and of the free buffer memory of all routers along the route from the sender to the receiver and back. Increasing TCP windows above the size of the wire pipe capacity does not increase throughput. The TCP sender can already send packets at the available bandwidth and further enlargement of TCP windows just increases the volume of outstanding data at the expense of filling up router queues. As the available bandwidth is likely to fluctuate due to changes in cross traffic, TCP windows slightly larger than the wire pipe capacity can ensure that the TCP sender will never need to stop sending packets to wait for acknowledgements. And we have still good chance that router queues will keep up with peeks in cross-traffic, preventing losses and allowing for steady throughput.

When TCP windows are significantly larger than the wire pipe capacity, filled-up router buffers increase probability of losses due to cross-traffic. Moreover, as

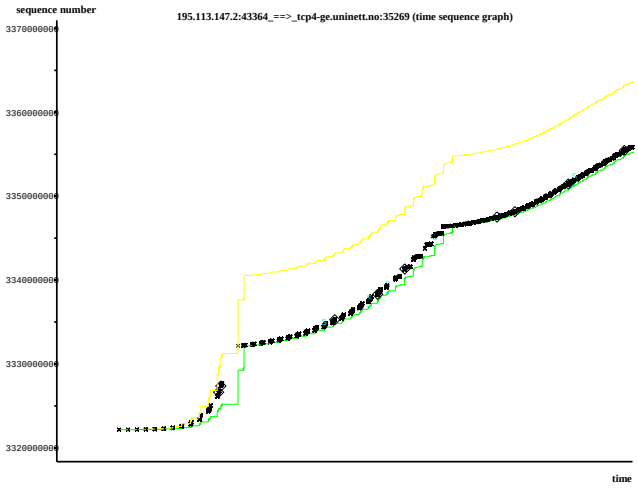

Fig. 4. Multiple losses caused by too large TCP windows

the large volume of data moves among router queues, follow-up losses can occur on routers farther along the route when the TCP connection was in the early part of its congestion avoidance cycle or even in the slow start phase. This causes `ssthresh` to be halved multiple times resulting in very long congestion avoidance and poor throughput. An example of sequence number development of a connection when these multiple losses happened is in Fig. 4. This effect can occur even with little or not fluctuating cross-traffic. When we configure large TCP buffers, the sender congestion window will eventually grow to fill up router queues. TCP is self-clocking in a sense that it can send out packets only as fast as acknowledgements come from the receiver. It is also self-regulating, meaning that a larger round-trip time implies slower congestion window increase. However, it appears that with a large window advertised by the receiver, the sender can overshoot the available bandwidth too much and the decrease in sender speed comes too late to prevent congestion and loss. When other connections use part of the router queues to fill up their limits for TCP windows, this volume of data remains in the queues and reduces available buffer space. The sender outstanding window can be limited either by manual configuration of TCP buffers on receiver or sender side or by dynamic moderation of the receiver-advertised window or the sender-computer congestion window.

For manual configuration of TCP buffers we need to know available bandwidth of the particular network path. If we do not want to use intrusive bandwidth measurement such as with iperf, we can try an avail-

able bandwidth estimation tool based on analysis of delay variation of testing packets. Currently the only publicly available tool of this class is pathload [5]. It is distributed with built-in constants limiting its operation to 120 Mb/s. These constants can be tweaked to allow operation up to the full Gigabit Ethernet speed. However, our experience shows that pathload output is not reliable. Depending on network conditions, the produced results fall into three categories: i) it correctly iterates with variable-rate packet chains to the realistic available bandwidth estimated with 50 Mb/s precision (when that happened on our network path, the indicated available bandwidth was in the range of 850-1000 Mb/s, probably periods of lighter load), ii) it loses lots of packets even in low-rate chains and falsely states the available bandwidth in the range of 50-100 Mb/s and iii) it does not detect any delay increase even in high-rate chains and states an unlikely available bandwidth of 1000 Mb/s. We can make multiple measurements to increase probability that the result is within a specified range.

There are several works that try to configure TCP buffer and window sizes automatically [1], [8], [9]. The goal of these projects was to save memory by not allocating buffers much larger than needed to include the outstanding window or to set the receiver-advertised window or the sender-computed congestion window so that the connection is not limited by them. Linux 2.4 now also by default includes a sort of buffer autoconfiguration and window moderation as described in section II. The goal of this moderation is to prevent sudden bursts that can cause losses, for which purpose it is effective.

However, what we need is window moderation that will keep the outstanding window only slightly above the wire pipe capacity. In principle, this can be achieved by moderating either the receiver-advertised window or the sender-computed congestion window. It is probably easier to do that on the sender. We can check initial RTT of the first SEQ-ACK pairs during connection handshake and during the start of sending data. This RTT will be the sum of empty-pipe RTT and waiting caused by data of other connections queued in router buffers. We can then observe current RTT during the connection. If it tends to increase significantly above the initial RTT we should stop increasing `cwnd`. An example of development of RTT and outstanding window of a connection with too large

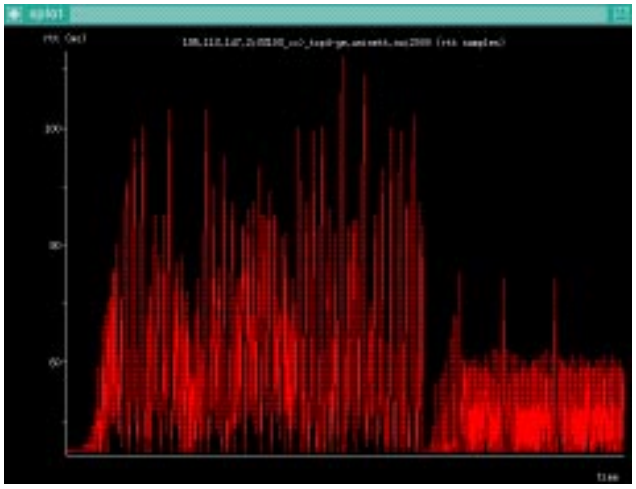

Fig. 5. Development of RTT for too large TCP windows

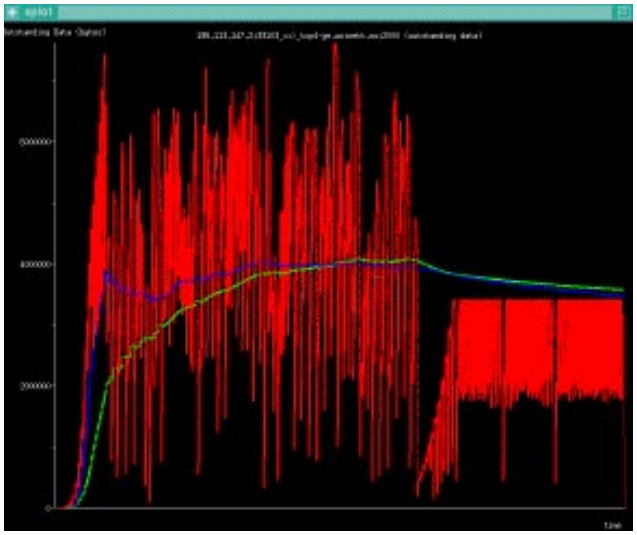

Fig. 6. Development of outstanding window for too large TCP windows

TCP windows as shown in Fig. 5 and Fig. 6, respectively. RTT measured by ping program was 40 ms. Available bandwidth was at most 1 Gb/s (our international link has installed bandwidth of 1.2 Gb/s and router statistics showed that the current load was about 200 Mb/s). Therefore, the empty pipe capacity was at most 5 MB. Large TCP windows allowed the outstanding window to grow up to 8 MB and RTT was as high as 110 ms. Some peaks were caused by fluctuating cross traffic. After about 6 seconds from the connection start a burst of packets has been lost causing slow start.

IV. SSH Performance

It is nice when we achieve high throughput with iperf, but people are not transfering their files with iperf, they use some file transfer application. One of the most popular is scp, which uses an authenticated and encrypted channel established by ssh. We found that the file transfer speed with scp over a long-distance network was much lower than TCP throughput measured by iperf with the same TCP window over the same network path. For example, on our testing path from Cesnet to Uninett, we achieved throughput of 250 Mb/s with iperf using 2 MB windows. The scp file transfer initiated right after iperf achieved throughput of 10.4 Mb/s. Many people tend to blindly blame high CPU load caused by data encryption. We checked CPU load during this file transfer and found that it was approximately 9% on both sides of the connection.

The problem is in internal window maintained by the ssh protocol. The corresponding part of the protocol operation is illustrated in Fig. 7. Applications such as scp or sftp just use a channel created by ssh, so the problem is common to all these applications. The ssh protocol puts data to internal packets, which are by default 32 kB big. Certain maximum number of these internal packets can be outstanding at any given time, creating an internal application window. Default size of this window is four internal packets, that is 128 kB. Therefore, throughput is by default always lower than 128 kB / RTT. The size of internal packets is set in ssh code by compiler-time constant CHAN_x_PACKET_DEFAULT, where x can be SES, TCP or X11 for different channel types. Similarly, the size of the internal window is set in ssh code by compiler-time constants CHAN_x_WINDOW_DEFAULT.

Sequence number development for ssh connection with default internal packets and windows over 40 ms RTT with 2 MB TCP windows is illustrated in Fig. 8. The TCP sender could not utilize the available receiver-advertised window because of slow application-level communication. When we increased the internal window to 40 internal packets (1.25 MB), throughput increased to 48 Mb/s with CPU load increased to 45%. When we increased the internal window to 80 internal packets (2.5 MB), throughput increased to 88 Mb/s with CPU load increased to 85%. Sequence number development of this connection is illustrated in Fig. 9.

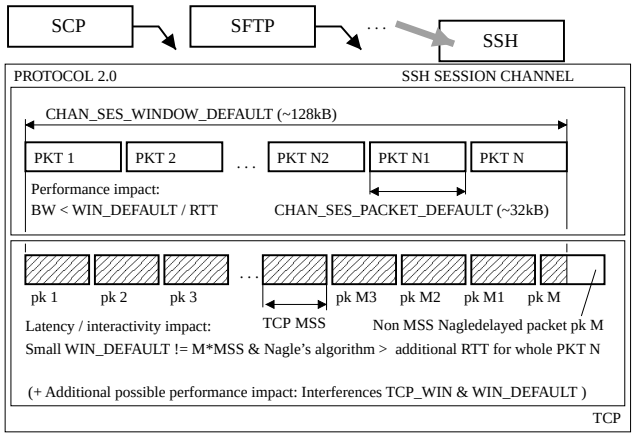

Fig. 7. Internal packets and windows in SSH protocol

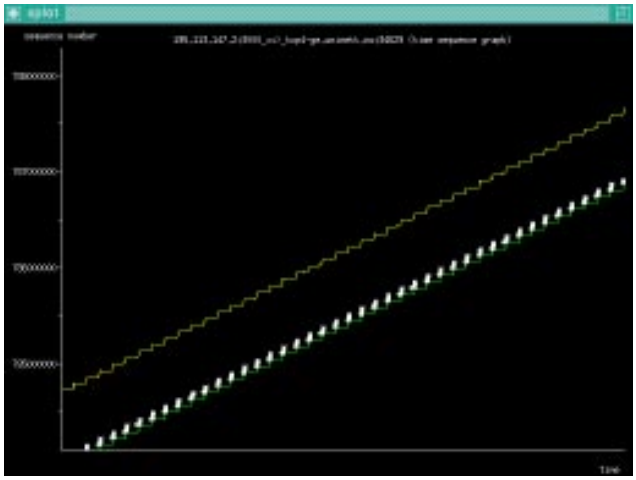

Fig. 8. Development sequence numbers of default SSH connection

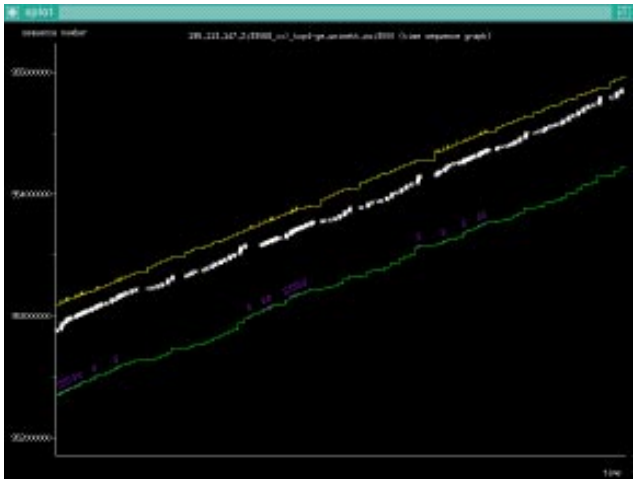

Fig. 9. Development of sequence numbers of SSH connection with internal window increased to 2.5 MB

V. Future work

We are studying implementation of more appropriate sender-computed congestion window moderation.

We intend to do comparison tests to evaluate performance of TCP window moderation combined with dynamic AIMD adjustments.

We are working on QoS and performance monitoring application running on top of the SCAMPI [10] architecture that will help to identify selected performance related problems.

VI. Conclusion

- Too large TCP windows may require other configuration changes for good performance and may deliver worse performance than smaller windows.
- Some common networking applications such as ssh have significant performance problems that must be resolved at the application level.
- The influence of configuration of the networking subsystem on end hosts and the influence of implementation-specific features present in commodity operating systems on the resulting throughput and TCP behaviour can be significant and sometimes stronger than the influence of various enhancements of the TCP congestion control described in literature.
- PERT (Performance Enhancement and Response Team) [11] initiative has started. Can it be the place for interaction of research people with software developers, equipment manufacturers and other communities involved in high-performance networking?

References

[1] W. Feng, M. Fisk, M. Gardner, E. Weigle. *Dynamic Right-Sizing: An Automated, Lightweight, and Scalable Technique for Enhancing Grid Performance*, PfHSN 2002, Berlin, Germany.

[2] *Web100, version 2.1*, `http://www.web100.org`.

[3] Sally Floyd. "HighSpeed TCP for Large Congestion Windows", *draft-floyd-tcp-highspeed-00.txt*, Internet Engineering task Force, June 2002.

[4] Dina Katabi, Mark Handley, Charlie Rohrs. *Internet Congestion Control for Future High Bandwidth-Delay Product Environments*, ACM SIGCOMM 2002.

[5] Manish Jain, Constantinos Dovrolis. *Pathload: a measurement tool for end-to-end available bandwidth*, PAM 2002.

[6] Sven Ubik, Antonin Kral. *Using end-to-end bandwidth estimation tools in high-speed networks*, CESNET Technical Report, work in progress.

[7] M. Allman, V. Paxson, W. Stevens. "TCP Congestion Control", *Request For Comments 2581*, Internet Engineering Task Force, 1999.

[8] J. Semke, J. Mahdavi, M. Mathis. "Automatic TCP Buffer Tuning", *Proceedings of ACM SIGCOMM 1988*.

[9] Tom Dunigan, Matt Mathis, Brian Tierney. *A TCP Tuning Daemon*, SuperComputing 2002, Baltimore, November 16-22, 2002.

[10] "SCAMPI - A Scaleable Monitoring Platform for the Internet", European Research Project, Contract No. IST-2001-32404, `http://www.ist-scampi.org`.

[11] "PERT - Performance Enhancement and Response Team", `http://www.dante.net/tf-ngn/pert`.