High Speed Bulk Data Transfer on the Grid Using the SSH Protocol

Chris Rapier
Pittsburgh Supercomputing Center
300 South Craig Street
Pittsburgh, PA 15213
1-412-268-4960
rapier@psc.edu

ABSTRACT

SSH is a widely used multipurpose application for interactive shells, bulk data transfer, and other network transport needs. However, a design choice in most implementations of SSH reduces its functionality as bulk data transport tool in Grid and other high performance environments. In this paper I will discuss the nature of the design choice, the functional limitations it imposes, a method by which it can be remedied, and introduce a set of patches know as HPN-SSH. Based on the industry standard OpenSSH these patches eliminate this bottleneck and allow for data transfers at very high speed across wide area networks.

Categories and Subject Descriptors

C.2.2. [**Computer-Communication Networks**]: Network Protocols *applications*.

General Terms

Your general terms must be any of the following 16 designated terms: Performance, Design, Security, Human Factors, Standardization.

Keywords

SSH, performance, bottlenecks, buffers, high performance networks, HPN, HPN-SSH, auto-tuning, security, cryptography.

1. INTRODUCTION

In 1995, while working the Helsinki University of Technology in Finland, Tatu Ylönen developed SSH as a replacement for less secure connections protocols like Telnet and rsh. The obvious advantages of cryptographic security in an increasingly insecure Internet environment led to its rapid adoption. In 1996 a new version of SSH, SSH2[1] (SSH and SSH2 are generally used interchangeably), was released incorporating multiplexed connection for channeling of multiple secure data connections over a single TCP connection. As each of these individual channels was now unaware of the underlying TCP flow control mechanism an application layer mechanism was implemented. The authors settled on mechanism analogous to the TCP receive window. This SSH receive window has an effective size of 64KB[1], which was the typical maximum TCP receive window at that time. The result of these layer flow controls is that the effective receive window of any SSH2 connection is the minimum of the SSH and TCP receive windows.

As network speeds increased the importance of properly sized receive windows became clearer and resulted in the wide spread adoption of much larger TCP receive windows[2], adaptive receive buffer windows, and other enhancements. However, the SSH receive window in most implementations of SSH has not kept pace. As such even on well tuned systems with large receive windows would be limited to no more than 64KB of data in transit during any given RTT (Round Trip Time). The impact of this, which will be discussed in more detail in the next section, is to significantly impede performance in high performance networking environments. In some situations it can impose a 95%+ reduction in expected throughput. Even so, in these environments SSH and its associated bulk data transport applications, SFTP and SCP, are widely used to transfer data because of ubiquity, easy of use, and familiarity. Users generally just accept the poor performance or, in many cases, assume that it is a network related problem.

One of the author's responsibilities is to provide network support to users and problems with SSH were frequent. In many cases alternatives such as Kerberized FTP or GridFTP were not viable solutions. Therefore a patch, known as HPN-SSH, was developed to dynamically increase the size of the SSH2 receive window and eliminate this bottleneck. The results were a dramatic improvement in throughput speed – in some cases approaching two full orders of magnitude.

In this paper the impact of receive windows will be discussed, the solution that the author's team developed, and the results that have been observed.

2. IMPACT

2.1 Receive Buffers

In order to understand why the SSH receive window is so important it is critical to understand how receive windows impact network performance. Since the path between any two hosts is not instantaneous there will often be data in transit between the two

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Gridnets 2007, October 17–19, 2007, Lyon, France.
Copyright Placeholder.

[1] SSH, SSH2, and SSHv2 are used interchangeably. Almost all references to SSH in the literature refers to SSH2.

[2] RFC 1323 created a window scaling options which increases the advertised receive window through bit-shifting. TCP receive windows can now be as large as 1GB in size[2].

hosts. Since the receiver has not yet seen this data it is unacknowledged or outstanding data. The amount of outstanding data that can physically exist in any given path is known as the BDP (Bandwidth Delay Product) and is equal to the bandwidth at the narrowest point in the path multiplied by the RTT (Round Trip Time) or,

$BDP = BW * RTT$[5]

For example, a 1Gbit connection on a 60ms path would have a BDP of 7.5MB. In other words, this particular path can transmit up to 7.5MB every RTT.

The receiver limits how much outstanding data is allowed at any one time by advertising the size of a buffer used to hold incoming data; this is the receive window. The available space in this buffer is advertised at connection establishment and through the life of the connection. The sender will transmit up to this amount of data and then wait for an acknowledgement from the receiver. When an acknowledgement is received the sender will resume data transfer. If any particular data is not acknowledged in time the sender will assume it has been lost in transit and retransmit the missing data. This is how TCP maintains reliable data transfer.

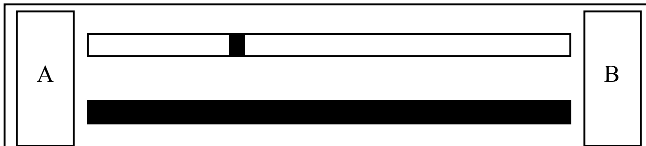

Figure 1. A comparison of undersized versus sufficiently large receive windows.

Two identical connections exist between Hosts A and B. In the first connection an undersized window allows only a small portion of the available network capacity to be utilized. In the second connection a window size that matches the BDP of the path permits full utilization.

In terms of throughput, if the receive window size is less than the BDP the network will sit idle for a portion of each RTT. This will introduce idle periods where the network isn't being fully utilized as illustrated in Figure 1. The size of the TCP receive window can be tuned manually or through automatic buffer tuning algorithms[3].

In the case of OpenSSH[4] the application receive window is statically defined at 128KB. Due to the buffer draining method, no more than 64KB of data will be sent before SSH will pause and wait for an acknowledgement from the receiving SSH application. This occurs on a per channel basis. The result of this is that the effective receive window of the connection as whole is the minimum of the TCP and all SSH receive windows. Specifically,

$RWIN_e = MIN(RWIN_{tcp}, RWIN_{ssh})$[6]

[3] Linux 2.6.9 and higher and Windows Vista have this capability.

[4] We will be using OpenSSH as the example implementation of SSH throughout this paper. It is the most widely used implementation, the de facto industry standard and the basis of the HPN-SSH patch. However, the limitations noted in this paper are common across all implementations of the SSH2 protocol known to this author

Figure 2 illustrates the disparity between the SSH and required TCP receive windows (as dictated by BDP) in a hypothetical 1Gb/s network of varying length. Since the SSH receive window is statically defined no amount of TCP tuning will help improve SSH throughput performance in a high BDP path.

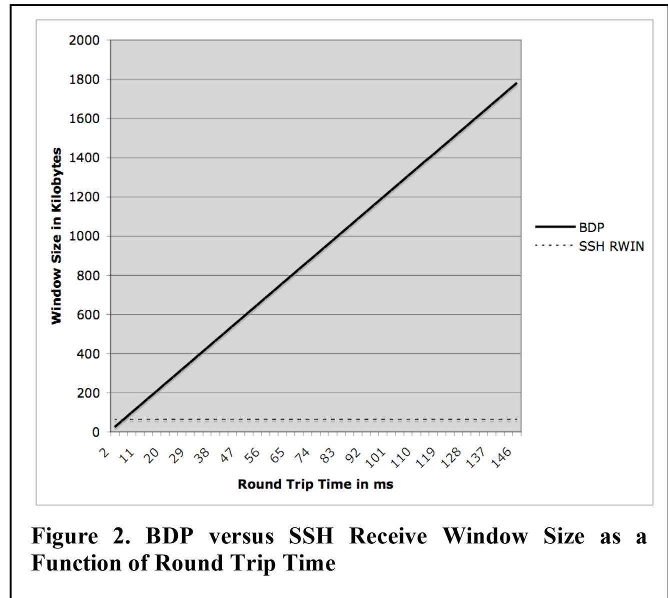

Figure 2. BDP versus SSH Receive Window Size as a Function of Round Trip Time

2.2 Network Utilization

The theoretic maximum throughput can be determined by using the formula for BDP and assuming that the BDP is equal to the effect receive window ($RWIN_e$) and solving for bandwidth (BW). Resulting in,

$BW = RWIN_e/RTT$

Since the receive window of SSH is fixed in size throughput performance will be inversely proportional to the RTT on a given path.

The actual effect of a fixed receive buffer on throughput can be seen in Figure 3 which shows throughput on a hypothetical 1Gb/s path of varying length with a fixed 64KB RWIN. This clearly illustrates a common experience with SSH, and the associated SCP and SFTP applications, where it is fast in the LAN but slow in the WAN. However, as local network speeds increase to 10Gb/s interconnects this receive window bottleneck will be experienced in the LAN as well. Being that SCP and SFTP are commonly used in LAN data transfers this small window size issue will, in time, become and important factor here as well.

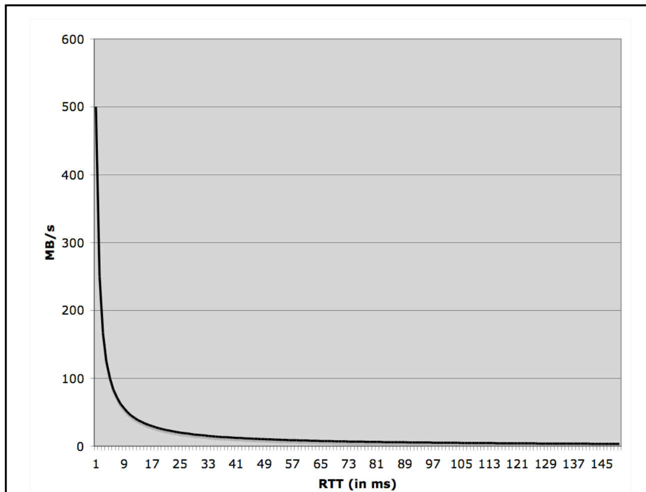

Figure 3. Throughput as a function of RTT.

This figure demonstrates the impact of RTT (Round Trip Time) on throughput for a fixed 64KB receive buffer.

2.3 Security Implications

This experience has proven to be a source of frustration to many users, especially those in high BDP environments like the Grid. Users want to use SSH (especially SCP and SFTP) because its familiar, secure, and easy to use but the speed penalty is substantial. This has led many users to blame the performance problem on either a failing network or the overhead imposed by encryption. However, if the performance problems were being caused by the processor being overloaded one would expect transfers to be slow regardless of this distance between end hosts. Likewise, if the network really was to blame the results would be reproducible with throughput tests like Iperf.

While viable and fast alternatives exist in the form of Kerberized FTP and GridFTP they are unavailable or unknown to a significant portion of users. As such, they have, at times, circumvented security restrictions on unencrypted authentication. While this can provide some relief it is fraught with security problems and is, at best, a half measure. Therefore, the author believes that the SSH performance issue actually creates a security hazard because it may encourage the use of less secure transport methods.

3. SOLUTION

The simplest solution to this problem is to statically redefine the size of the SSH receive window to some appropriately large size. This is, essentially, the approach taken when multiplexing was incorporated into SSH. At that time, while window sizes greater than 64KB were possible through the use of window scaling as described in RFC 1323, the majority of systems were configured with a maximum TCP receive window of 8KB to 16KB. As such, a 64KB window was a reasonable choice. If one used such reasoning today it would be necessary to define the SSH receive window to at least 64MB if not 128MB. This would suffice for high performance and Grid environments but a waste of resources in many others. One of the great advantages of SSH is its ability to function in a wide variety of environments and platforms. Any solution should work to maintain that flexibility even while it enhances performance.

3.1 Layer 4 Awareness

The solution developed by the author, and made available in the HPN-SSH patch[5], is to make SSH aware of the network protocol layer (OSI Layer 4). When the HPN-SSH application, either the server or the client, is instantiated the kernel is queried (through the use of getsockopt()) to determine the size of the TCP receive window. This value is then used to dynamically set the size of the HPN-SSH receive window at run time. By doing this we remove this bottleneck without necessitating the use of statically defined overly large buffers. One of the advantages of this method is that the HPN-SSH patch only needs to be applied in the direction of the bulk data transfer. As such, in heterogeneous connections (SSH to HPN-SSH) users can realize significant performance gains while using standard SSH implementations.

3.2 Autotuning Kernels

A problem arises if the underlying operating system is using what is known as an autotuning kernel. In these systems a process tracks various aspects of individual TCP connections and incrementally grows the receive window to maximize network utilization[4]. Typically the TCP receive window starts out relatively small, approximately 85KB in Linux, but can grow up to 64MB or more in size[6]. If HPN-SSH only queried the kernel once at connectional establishment the SSH receive window may end up remaining undersized. Therefore, the HPN-SSH patched application periodically re-queries the kernel to get the current window size. HPN-SSH will then grow its internal receive window accordingly.

The can, in some instances, such as local LAN transfers, produce problem. Currently the HPN-SSH application will query the kernel once every round trip time. With sub millisecond RTTs these queries can impose additional overhead which may end up reducing overall performance. However, a run time option is available to disable intermittent kernel polling by the HPN-SSH application. This option is also a configuration option for use on operating systems with do not incorporate autotuning kernels.

3.3 Protocol Issues

Protocols such as SCP and SFTP typically, but not necessarily, make use of SSH as their transport mechanism so performance improvements to the SSH code are typically picked up by them. However, as the underlying transports speeds up this can uncover previously hidden bottlenecks in the applications.

In the case of SFTP an additional flow control mechanism was developed to limit the number of outstanding requests between client and server. Since transfers of data blocks (by default SFTP transfers data in 32KB chunks) are considered requests only so many data blocks may be in transit at any one time[1]. This ends

[5] HPN-SSH starts for High Performance Networking SSH and is a patch available for the OpenSSH implementation. Interested readers can find the patch and more information at http://www.psc.edu/networking/projects/hpn-ssh

[6] The out of the box default maximum receive window in Linux is actually around 170KB but it can be set much larger. Interested readers can learn more about high performance TCP tuning at http://www.psc.edu/networking/projects/tcptune

up acting like another receive window where the size is equal to the maximum number of outstanding requests allowed multiplied by the size of the data blocks. In the default configuration OpenSSH allows 16 outstanding requests giving a SFTP receive window size of 512KB. Since the effective receive window of the connection is the minimum of **all** of the receive windows this can prove to be a significant bottleneck in Grid and other high performance environments.

Future versions of HPN-SSH may address the problem by dynamically increasing the number of allowable outstanding requests and/or increasing the size of the data block. However, at this time users will need to use run time options to circumvent this potential bottleneck.

3.4 Encryption Overhead

Cryptographic methods are central to SSH however, most strong cryptographic methods are computationally expensive. SSH actually uses two methods, encryption for privacy and message authentication for integrity. In standard implementations of SSH the amount of data that moves through these routines is largely dependent on network throughput. Until SSH knows it can move more data onto the network no data will enter the cryptography routines. This has the tendency to make CPU load at least partially dependent on the RTT of the connection. Therefore, in low RTT paths CPU load will be greater than that in a high RTT path even if all other factors remain constant.

However, being that HPN-SSH removes a significant network bottleneck CPU loads can be significantly greater in comparison to an unpatched SSH transfer. In fact, with HPN-SSH it is not uncommon for bulk data transfers to be either processor or disk I/O limited. In a multi-user environment, such as data repositories on Grid networks, users may end up competing for scarce processor resources if users run concurrent instances of HPN-SSH. These users will often see highly variable performance even during the span of a single transfer. These performance problems are often mistakenly believed to be intermittent network issues.

To help alleviate this problem HPN-SSH reintroduced the NONE cipher for bulk data transfers. Most users aren't as concerned about the privacy of their data as much as they rely on SSH for its user authentication. Therefore, we introduced a cipher switching routine which allows the client and server to negotiate a different cipher during the connection. In our implementation all authentication data remains fully encrypted however, after authentication the user may drop the use of data encipherment by switching to the NONE cipher. Message authentication is maintained in order to maintain data integrity and protect against man-in-the-middle attacks. This can significantly improve bulk data throughput for some users without sacrificing an unreasonable amount of security.

The use of the NONE cipher switching must be initiated by the user on the command line. A separate configuration option must also be enabled on both the client and the server. Obviously this necessitates HPN-SSH applications at both ends of the connection. Additionally, safeguards are in place to prevent it being used during an interactive session where a user might enter sensitive personal or authentication information. Lastly, a warning is sent to STDERR whenever the NONE cipher is enabled.

3.5 Compatibility

HPN-SSH has been tested over the past 3 years against a wide range of other SSH implementations and versions. Based on the author's testing, experience, and the experience of many users there seem to be no known compatibility problems.

4. RESULTS

The HPN-SSH patch was first developed in 2004 and has undergone significant performance tuning and testing since that time. The results, from the beginning, have been very encouraging and have continued to improve since then. User reports have indicated that the current version of HPN-SSH provides very high performance in WAN environments and performance equivalent to unpatched installs of OpenSSH in LAN environments.

4.1 WAN Performance

High BDP paths, characterized by high bandwidth and high delay, will see the most dramatic performance improvements. It is not uncommon to see an increase in throughput of more than a full order of magnitude. In some cases, such as 10Gb/s networks, improvements of multiple orders of magnitude have been seen. The Figure 4 clearly illustrates the level of improvement that can be achieved with HPN-SSH on a well tuned host.

In many cases the performance of HPN-SSH using the NONE cipher will be very close or equal to throughput tests reported by Iperf. When HPN-SSH is significantly slower than Iperf tests it is often because disk I/O is acting as the bottleneck. When using HPN-SSH with encryption performance is typically less than what Iperf tests report. Generally this is due to one or both of the end hosts having all available processor resources consumed by cryptographic routines. As such, more processor intensive ciphers, such as Triple DES, will be slower than more efficient ciphers like 128Bit AES.

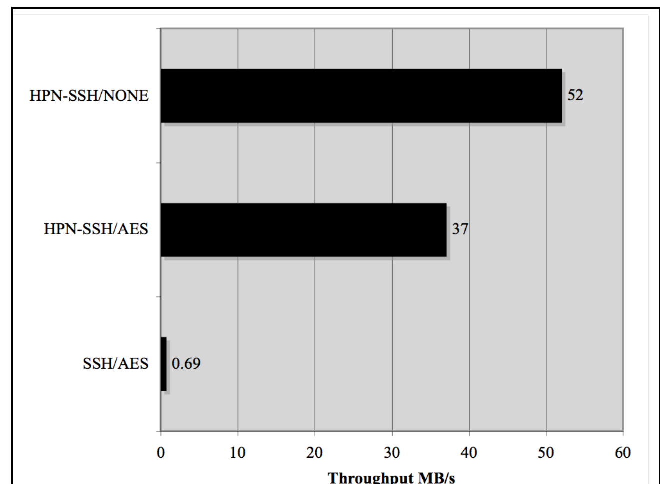

Figure 4. Throughput Comparison of HPN-SSH and SSH.

This figure shows throughput along a 1Gb/s transatlantic path with a 114ms RTT using HPN-SSH and SSH using the default AES cipher. Additionally, throughput of HPN-SSH using the NONE cipher along the same path is shown.

4.2 LAN Performance

Early versions of HPN-SSH showed a disappointing tendency to actually impose a performance penalty in local area networks. This problem, as mentioned earlier, seems to stem in part from increased overhead due to TCP receive window polling every RTT. It may have also been partly caused by managing unnecessarily large buffers. However, replicating these problems proved them to be inconsistent and highly dependent on the particulars of the LAN setup including the type of switch or hub used, the available memory, processor speed and so forth. Unfortunately, the performance problems were verifiable even if they were inconsistent. To address this users have been provided a means to disable all of the HPN functionality in HPN-SSH. This essentially turns the HPN-SSH back into a standard OpenSSH. This will probably not be necessary in the great majority of cases though.

Overall, testing and users indicate that LAN bulk data transfers using HPN-SSH are at least as fast as SSH.

5. CONCLUSION

5.1 User Experience

Early in the development process of HPN-SSH the author realized that the user experience was one of the most critical factors in any application. One of the reasons why SSH has been so widely adopted is because it provides a simple to use, easy to maintain, highly compatible, and consistent user experience. From the user's point of view SSH 'just works'.

With this in mind HPN-SSH was developed so that no fundamental changes were required on the part of the user. It is, in almost every case, a drop in replacement for standard SSH installations. Any previous defined user aliases, scripts, interfaces, and the like continue to work as they always have. The only noticeable difference to the user is that HPN-SSH is significantly faster. The caveat being that the performance improvement will only be seen if transferred data is being received by HPN-SSH.

6.2 Adoption

In the spring of 2007 HPN-SSH became a required component for CTSS 4 compliance in the TeraGrid. It has also been incorporated in to the GSI-SCP patch available from NCSA. HPN-SSH is used by NASA Ames, major research laboratories, many high performance computing centers, agencies within the US Federal Government, financial institutions, technology firms, and is part of the default distribution of HP-UX from Hewlett-Packard. It has also become supported optional components of several Linux Distributions as well as FreeBSD.

As of this writing the OpenSSH development team has not chosen to incorporate HPN-SSH into the main code base. Due to the size of the HPN-SSH patch the volunteer developers simply haven't had the time or resources to fully verify the code. The author is continuing to work with the development team to help provide these resources and address any issues that might arise.

6.3 Final Thoughts

SSH, including SCP and SFTP, are a set of flexible, robust, and eminently useful applications which are unfortunately hamstrung in high BDP environments. This is caused by a flow control mechanism in SSH which is constrained by a statically defined small receive window. HPN-SSH eliminates this bottleneck by replacing this static window with a dynamically defined, expandable, self-adjusting window. Removing this bottleneck transforms SSH by boosting bulk data throughput an order of magnitude or more in high performance environments. It remains an easy to use, simply to maintain, highly secure, and now a high-speed transfer tool well suited for Grid user needs.

6. ACKNOWLEDGMENTS

Our thanks go to Cisco Systems Inc. and to the National Science Foundation for supporting the development of HPN-SSH. This work would not have been possible without the expertise of Michael Stevens who did the bulk of the programming. Lastly, my thanks and deepest appreciation goes out to the OpenSSH development team and Tatu Ylönen.

7. REFERENCES

[1] GALBRAITH, J., YLONEN, T., LEHTINEN, S., SSH File Transfer Protocol. IETF Internet Draft draft-ietf.secsh-filexfer-04. Internet Engineering Task Force (Web site: www.ietf.org)

[2] JACONSON, V., BRADEN, R., BORMAN D., TCP Extension for High Performance. IETF RFC 1323. Internet Engineering Task Force (Web site: www.ietf.org)

[3] OpenSSH 3.8p1 Source Code (channels.c)

[4] J. Semke, J. Mahdavi, M. Mathis. Automatic TCP Buffer Tuning. *Proceedings of SIGCOMM '98 Conference*, Vol 28, No. 4, pp. 315-323, August 1998

[5] W. R. Stevens, *TCP/IP Illustrated Volume 1: The Protocols,* Reading, MA, Addison Weseley Longman Inc. 1994, pp. 289-291

[6] S. Ubik, P. Cimbal, Achieving Reliable High Performance in LFNs, *TNC 2003*. Zagreb, Croatia, May 19-23, 2003.