

High Speed Bulk Data Transfer Using the SSH Protocol

Chris Rapier, Benjamin Bennett
Pittsburgh Supercomputing Center
300 South Craig Street
Pittsburgh, PA 15213
1-412-268-4960

rapier@psc.edu, ben@psc.edu

ABSTRACT

SSH is a highly successful multipurpose protocol used for both interactive shells and transport layer mechanisms. However, a design choice in most implementations of SSH reduces its functionality as bulk data transport tool in high performance network environments. This paper will discuss the nature of this limitation, the functional barriers it imposes, a method by which it can be remedied, and introduces a high performance implementation based on the industry standard, OpenSSH. Additionally, the authors will introduce a method by which performance on multi-core systems is improved through the use of cipher multi-threading.

Categories and Subject Descriptors

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

General Terms

Performance, Design, Security, Human Factors, Standardization.

Keywords

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

1. INTRODUCTION

1.1 Why SSH?

Users almost always require the means to transfer data into and out of various computers systems. While some methods like anonymous FTP or HTTP allow for the unauthenticated transfer of data many users and systems require some level of protection against unauthorized access. At one, now distant, point clear text authentication methods – like FTP and telnet – provided sufficient protections. However these proved to be easy prey for malicious actors. This forced the adoption of strong, cryptographically secure authentication methods for accessing remote systems and resources.

In the realm of high performance networking there are a number of bulk data transfer solutions available to users and administrators. These include the Kerberos solution kFTP[5], Globus based solutions like GridFTP[1] and glogin[8], security overlays like VPNs, and many others. However, each of these solutions presents their own set of complications. GridFTP can be difficult to configure, costly to maintain, and is limited in its installation to larger institutions. Kerberos is much more common but it still poses problems for smaller organizations and is also limited in its distribution. VPNs, even though relatively common with a wide number of available clients, can incur user management problems, and suffer from poor performance.

SSH, in contrast, is widely available as both an open source and commercial product for almost every combination of operating system and hardware platform in use. It is easy to install and use, painless to configure, and trivial to administer. However, it suffers from a weakness significantly limiting its usefulness: transfer speeds over wide area networks can be intolerably slow.

1.2 Performance and SSH

In 1995, while working the Helsinki University of Technology in Finland, Tatu Ylönen developed SSH as a replacement protocol for less secure network applications rsh, rcp, and telnet. The obvious advantages of strong cryptographic security in an increasingly insecure Internet environment led to its rapid adoption. In 1996 a major version update of SSH was released known as SSH2¹. This version of the protocol mandated the multiplexing of secure session channels over a single TCP connection. As each of these individual channels was now unaware of the underlying TCP flow control and application layer flow control was required. The protocol developers settled on a windowing flow control mechanism analogous to the TCP receive window. This created a ‘layered window’ where the application receive window rode on top of the TCP receive window. The result of this 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 clear and resulted in the wide spread adoption of much larger TCP receive windows², adaptive receive

¹ SSH, SSH2, and SSHv2 are used interchangeably. Almost all references to SSH in the literature and this paper refer 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 1 GiB in size[4].

windows³, and other enhancements. However, until very recently⁴, the SSH receive window has not kept pace and remained at 64KiB – which was the pre-RFC 1323 maximum.

The impact of this, which will be discussed in more detail in the next section, significantly impedes performance in high performance networking environments. Contrary to widely held opinion the throughput issues are generally not a penalty imposed by cryptography. Instead they are overwhelmingly due to this windowing limitation. In some situations it can impose a 95%+ reduction in potential 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, ease of use, and familiarity. Users often simply accept the poor performance as the price of security or, in many cases, assume that it is a network related problem.

Faced with these performance limitations the authors began work on a solution. The result, a set of patches for OpenSSH known as HPN-SSH, opened up the bottleneck by dynamically increasing the size of the SSH receive window. The result was a dramatic improvement in throughput speed – in some cases approaching two full orders of magnitude. At this point the cryptographic computation overhead did become a limiting factor in throughput. To address this the authors reintroduced the NONE cipher as a post authentication option. Additionally, the authors have implemented a method of parallelization so that the workload can be shared among multiple processor cores.

This paper will review the impact of receive windows on network applications, a windowing solution that the authors’ team developed, the cryptographic performance limitations this revealed, the observed results, and directions for further work

2. WINDOWS

2.1 Receive Windows

In order to understand why the SSH receive window is so important it is critical to understand how TCP receive windows impact network performance.

Data from the data source in transit to the data sink is, by definition, unacknowledged data, as the end host has not yet received it. The amount of unacknowledged data that can exist in transit on any given path is known as the BDP (Bandwidth Delay Product) and is equal to the network bandwidth at the narrowest point in the path multiplied by the RTT (Round Trip Time) or,

$$BDP = BW * RTT[10]$$

For example, on a 1 Gb/s path with a 60ms RTT;

$$BDP = 1 \text{ Gb/s} * 60\text{ms}$$

$$BDP = 0.125\text{GB/s} * .06\text{s}$$

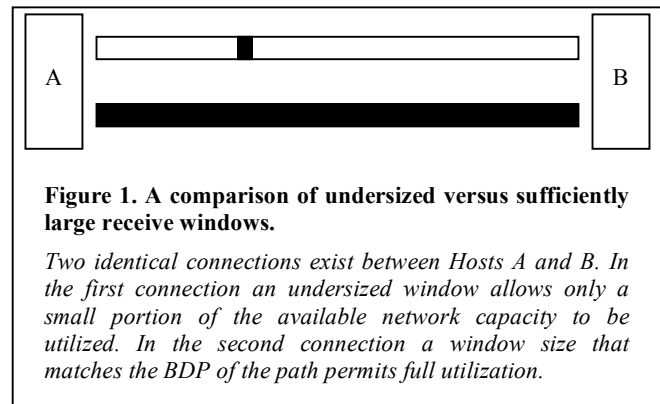
$$BDP = 7.5\text{MiB}$$

³ These are commonly known as autotuning receive buffers and can be found in the TCP/IP stacks of the 2.6.9+ Linux kernels and Microsoft Vista.

⁴ OpenSSH 4.7 has increased the buffer size to 1MB. This is a significant improvement but still limiting for high performance networking users. Also, due to the static nature of the buffer it may cause over buffering in some scenarios.

Therefore, this particular path can transmit up to 7.5MiB every RTT.

The receiver limits how much outstanding data is allowed at any one time by advertising the size of a available space in a buffer used to hold incoming data; this is the receive window. This window is advertised at connection establishment and through the life of the connection as the window size⁵. 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.



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 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 algorithms⁶.

In the case of OpenSSH⁷ the application receive window is statically defined at 128KiB. Due to the buffer draining method, no more than 64KiB of data will be sent before SSH will pause and wait for an acknowledgement from the receiving SSH application[7]. This occurs on a per channel basis so if there are multiple channels each channel may have a full 64KiB of outstanding data. Modern processors can normally perform all necessary cryptographic functions on the incoming data more quickly than the data can be acknowledged over the network. This results in the application often entering a state where it is waiting for more data from the network. In other words, modern machines are so fast that the throughput bottleneck is not the cryptography but the limited receive window size. The resulting application

⁵ Technically the window is the advertised size of the receive buffer and not the buffer itself. However, in practice buffer and window are often used interchangeably.

⁶ The previously mentioned adaptive receive windows.

⁷ 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

window behavior is identical in practical terms to the TCP receive window.

The result of this is that the effective receive window of the connection, from application to application, is the minimum of the TCP and all SSH receive windows. Specifically,

$$RWIN_e = MIN(RWIN_{tcp}, RWIN_{ssh})[11]$$

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

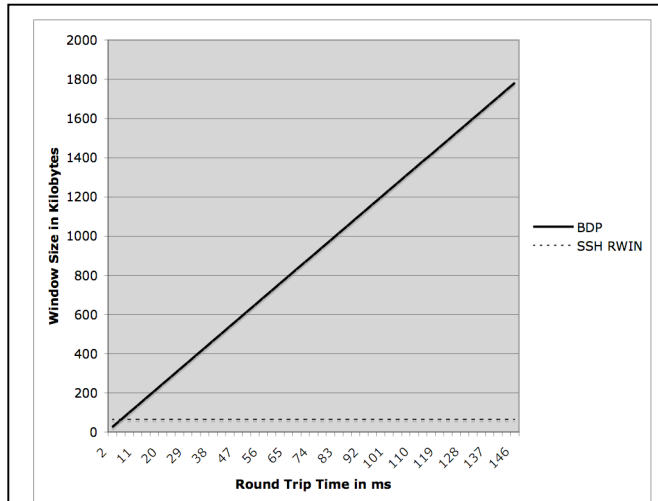


Figure 2. BDP versus SSH Receive Window Size as a Function of Round Trip Time

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 effective receive window ($RWIN_e$). Solving for bandwidth (BW) results 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.

This is illustrated in Figure 3 which shows throughput on hypothetical 1Gb/s paths of varying lengths with a fixed 64KiB RWIN. This clearly illustrates a common experience with SSH where it is fast in the LAN environment but slow in the WAN. However, as local network speeds increase to 10Gb/s this receive window bottleneck may be experienced in the LAN as well.

2.3 Security Implications

This phenomenon of excruciatingly slow SSH performance has proven to be a source of frustration to many users. This is especially true of those in high performance networking environments. Users want to use SSH and its associated

applications, SCP and SFTP, because they are familiar, secure, and easy to use. However, the throughput speed penalty is substantial across wide area networks. 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 processing overhead one would expect transfers to be slow regardless of this distance between hosts. Likewise, if the network were at fault the results would be reproducible with throughput tests like Iperf. In the vast majority of cases neither of these conditions happen to be true.

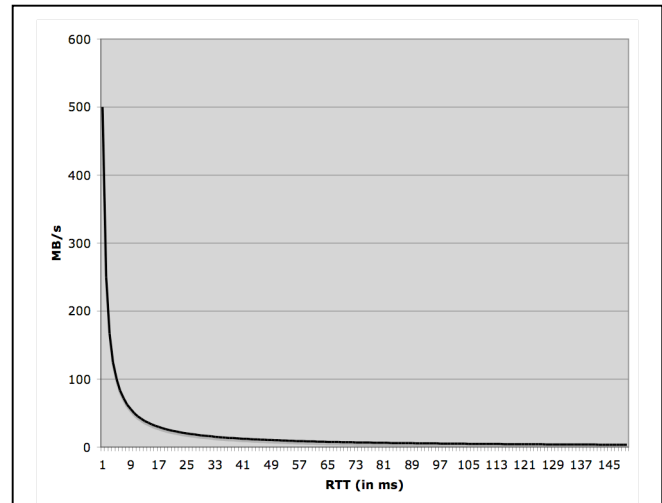


Figure 3. Throughput as a function of RTT.

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

While viable and fast alternatives exist in the form of kFTP and GridFTP they are unavailable or unknown to a significant portion of users. As such, many users have attempted to circumvent best security practices in order to achieve reasonable speeds when transferring bulk data. While this can provide some relief it is fraught with security problems and is not, in any way, a general solution to the problem. Therefore, the authors believe that the SSH performance limitation actually creates a security hazard as it may encourage the use of less secure transport methods. Security depends on functionality and ease of use almost as much as the underlying security methods. If the throughput problem were addressed, SSH would provide users with a fast, secure, and easy to use solution for bulk data transfer across high performance networks.

3. SOLUTION

3.1 Redefining the Window Size

The first method explored by the authors was to simply redefine SSH receive window size to some appropriately large value at compile time.

This is essentially the approach taken when multiplexing was incorporated into SSH. At that time, while window sizes greater than 64KiB 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 8KiB to 16KiB. As

such, a 64KiB window was a reasonable choice. With modern high performance networks it would be necessary to define the SSH receive window to at least 64MiB if not 128MiB.

However, blindly increasing window size may in fact have the opposite effect and cause performance degradation. This can be caused by filled buffers on intervening routers, overly rapid saturation of the path causing excessive loss and a return to slow start before congestion control mechanisms can take effect, or other causes [11]. Additionally, even mild over buffering can cause problems in interactive shell session in some scenarios. Users may still see remarkable improvement when using significantly oversized buffers however the risk of introducing new, and difficult to diagnose, performance problems does exist. As such, any viable performance solution should retain a level of flexibility and economy that is, in many ways, a hallmark of SSH.

3.2 Layer 4 Awareness

In light of the above problems the solution developed by the authors, and made available in the HPN-SSH patch⁸, was to make SSH aware of the transport protocol layer (OSI Layer 4). The underlying idea being that the TCP stack is the best source of information regarding network conditions. 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.

This method necessitates that the host operating system be properly configured for bulk data transfers. Fortunately, over the past several years significant advances have taken place in TCP receive window optimization. These include greater user awareness of the need for right sizing windows, dynamically adjusting receive windows, and applications to diagnose and resize undersized windows.

However, since this may require user intervention and in many cases that of a privileged user, it may still fail. In spite of this minor caveat there are a number of advantages with this method. One of the most significant being that the HPN-SSH application need only be installed at the destination of the bulk data transfer. As such, in some heterogeneous connections (SSH data source to HPN-SSH data sink) users of standard SSH implementations may still realize significant performance gains.

3.3 Autotuning Kernels

A problem arises if the underlying operating system is using dynamically sized receive windows, better known as an autotuning kernel. In these systems a kernel resource tracks various aspects of individual TCP connections and incrementally grows the receive window to maximize network utilization [9]. Typically the TCP receive window starts out relatively small, approximately 85KiB in Linux, but can grow up to 64MiB or more in size⁹. If HPN-SSH only queried the kernel at connection

⁸ HPN-SSH stands 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>

⁹ The out of the box default maximum receive window in some versions of Linux is around 170KiB but it can be set much

establishment the SSH receive window may end up remaining undersized. Therefore, HPN-SSH periodically re-queries the kernel to get the current window size and grow its internal receive window accordingly.

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

3.4 Encryption Overhead

Cryptographic methods are central to SSH and these methods tend to be computationally expensive. SSH uses two distinct cryptographic 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 cryptographic routines. This tends to make CPU utilization 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.

Table 1: Throughput as a function of CPU utilization.

	HPN-SSH		SSH 4.7	
	tput MB/s	load %cpu	tput MB/s	load %cpu
AES128-CBC	41.8	100	17.7	46
RC4	58.3	100	17.7	32
Blowfish	53.6	100	17.7	38
3DES	18.6	100	16.6	92
AES256-CBC	35.6	100	17.5	64
AES128-CTR	34.8	100	17.5	53
None	64.3	58	N/A	N/A

Using a 114ms 1Gb/s transatlantic path a clear relationship is shown between CPU utilization and throughput. In the case of HPN-SSH it demonstrates the relative computational cost of different ciphers. With OpenSSH 4.7 the application buffer limit effective caps throughput and reduces overall CPU utilization.

However, being that HPN-SSH removes a significant network bottleneck CPU loads may be significantly greater in comparison to an un-patched SSH transfer. This is clearly illustrated in Table I where a comparison is made between the load and throughput for both SSH and HPN-SSH. 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 distributed computing networks, users may end up competing for scarce processor resources if users run concurrent instances of HPN-SSH. These users will occasionally see highly variable

larger. More recent versions of the kernel have set the default maximum to 1MB. Interested readers can learn more about high performance TCP tuning for a variety of operating systems at <http://www.psc.edu/networking/projects/tcpune>

performance 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. This led to the development of a cipher switching routine that allows the client and server to negotiate a different cipher at any point during the connection. In HPN-SSH all authentication data remains fully encrypted but the user may switch to the NONE cipher to cease using encryption. 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 users willing to make the described security tradeoffs.

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. 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 Parallelization

In the course of this work the authors realized that multi-threading provides another possible avenue to enhance throughput performance. Referring to Table 1, the CPU utilization shown refers only to a single core on an SMP host. This is because OpenSSH performs all I/O and cryptographic operations in a single execution thread, restricting usage to a fraction of the available computational power. Trends in processor architecture strongly indicate that additional cores will be implemented rather than increasing raw single core processing power, a low number of multiple cores is quite common even today. As such, single-threaded SSH performance may remain stagnant or actually decrease over time.

Taking a closer look at the OpenSSH implementation reveals its logical serial order of required cryptographic operations. When sending, data is removed from a buffer and formatted into an SSH packet by padding to a cipher block size multiple, prepending a packet header, computing MAC, encrypting, and finally appending the plaintext MAC computed before encryption. Upon receiving an SSH packet, the first ciphertext block must be decrypted to obtain the packet length from the header, the remaining blocks may then be decrypted, MAC of the packet plaintext, excluding the MAC contained within the packet, is computed and verified against the MAC contained within the packet. Once decrypted and authenticated the packet is stripped of header, padding, and MAC, and appended to a data buffer.

Several possible multi-threading techniques are apparent. As a sender's MAC and cipher operations depend only on the packet plaintext, these operations may be performed concurrently once a packet is prepared with padding and header. This technique is not a possibility for the receiver as the MAC operation again depends on the plaintext, however on the receiving side the plaintext is available only after the cipher operation completes. Processing cipher blocks in parallel may also be possible, in varying degrees, depending on the cipher mode. For example, it is possible for ciphers in cipher block chaining (CBC) mode to

decrypt multiple blocks in parallel but not encrypt multiple blocks in parallel, whereas for ciphers in counter (CTR) mode [2] both encryption and decryption of cipher blocks may be done in parallel. The authors are not aware of any MAC algorithm that would allow for parallelism within a single MAC computation, however it may be possible to perform MAC computations on multiple SSH packets concurrently.

The initial development in this realm by the authors has focused on enhancing performance for a single cipher, AES [6] in counter mode, due to the high potential benefit and relatively small amount of existing code modification. In addition to parallel cipher block operations mentioned previously, counter mode allows for performing the expensive AES encrypts in advance of the plaintext being ready for encryption or ciphertext being available for decryption. At the time the plaintext or ciphertext becomes available only a low-cost XOR is required. This property of pregeneration implicitly allows for MAC and cipher operations to occur in parallel on both the sender and receiver.

4. RESULTS

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

4.1 WAN Performance

It is impossible to authoritatively state the level of improvement any individual user may experience, as this is highly dependent on path characteristics. However, in a sufficiently high BDP environment it is not uncommon to see performance anywhere from twice as fast to more than a full order of magnitude. The results of comparisons between HPN-SSH and OpenSSH 4.7 can be seen in Table 1. The results of a comparison between HPN-SSH versus OpenSSH 4.6 are illustrated in Figure 4. The performance differential between OpenSSH 4.7 and 4.6 should be noted. The superior throughput of 4.7 is due primarily to an increase to SSH receive window (from 64KiB to 1MiB) made by the developers.

Figure 4 and Table 1 both show the impact that the processor overhead imposed by the cryptographic methods may have on throughput. In many cases the performance of HPN-SSH using the NONE cipher will be very close to the limits of the I/O subsystem; in this case disk I/O. Freed from the constraints of disk I/O, throughput rates of 100MB/s on a transcontinental 1Gb/s path have been demonstrated.

4.2 LAN Performance

Early versions of HPN-SSH showed a disappointing tendency to 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 to be frustrating. The performance failures tended 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. To address this, users have

been provided a means to disable all of the HPN functionality in HPN-SSH. This has been shown to reduce if not eliminate the incidence of this problem. Recent changes in the patch code, such as eliminating unnecessary buffer handling routines, have eliminated some of the possible problem areas. In general, reports of poor performance in local area networks remain rare.

Overall, testing and users indicate that LAN bulk data transfers using HPN-SSH are comparable, if not identical, to OpenSSH

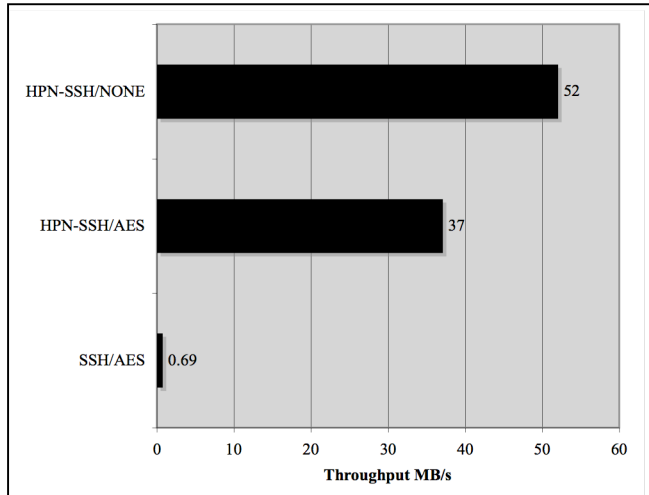


Figure 4. Throughput of HPN-SSH versus SSH 4.6.

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.

4.3 Protocol Issues

Protocols such as SCP and SFTP typically 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 transport 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 32KiB chunks) are considered requests, only so many data blocks may be in transit at any one time [3]. The result of this is the imposition of another receive window on top of the already existent windows. As such, the window 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 512KiB. 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 will 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.

4.4 Multi-threaded Cipher

The authors' current implementation of a multi-threaded AES counter mode cipher (AES128-CTR) employs a variable number of pregeneration threads to perform AES encrypt operations. Threads are created the moment a cipher context is initialized with both key and initialization vector, storing the results in a ring of keystream queues. When encryption or decryption of data is required, the main execution thread reads keystream values from these queues and performs bitwise exclusive ors with the plaintext or ciphertext data.

As there is only a slight amount of work required by this cipher in the main execution thread, throughput with any of its supported key lengths can exceed 99% of the throughput measured with the None cipher (no encryption) for a disk-backed transfer between two 8-core workstations connected via 1Gb/s ethernet, as shown in Table 2.

This increase in performance comes at the cost of a higher memory footprint for buffering of the pregenerated keystream. While the memory usage increase of tens to hundreds of kilobytes is often insignificant for today's generic computing platforms, it can be prohibitive for tightly constrained environments such as some embedded systems.

Table 2: Single-threaded vs. Multi-threaded Cipher

	Single-Threaded		Multi-Threaded	
	tput MiB/s	load %cpu	tput MiB/s	load %cpu
AES128-CTR	61.2	100	107.4	190
AES192-CTR	56.6	100	107.3	202
AES256-CTR	52.0	100	107.0	215
None	107.5	70	N/A	N/A

4.5 Compatibility

HPN-SSH has been tested over several years against a wide range of other SSH implementations and versions. Based on the authors' testing, experience, and the experience of many users there are no known compatibility problems. Additionally, the authors' multi-threaded implementation of AES128-CTR modifies only the internal operations of the given cipher, maintaining full compatibility with others.

5. CONCLUSION

5.1 User Experience

Early in the development process of HPN-SSH the authors 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 previously 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.

5.2 Adoption

In the spring of 2007 HPN-SSH became a required component for CTSS 4 compliance in the TeraGrid. It has also been incorporated into 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-Packard¹⁰. 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 authors are continuing to work with the development team to help provide these resources and address any issues that might arise.

5.3 Future Work

Further performance enhancement will, in many ways, be incremental in comparison to the dramatic increase afforded by right-sized receive windows and multi-threading. However, several interest avenues of work and research remain: striping transfers across multiple servers, like GridFTP, may be a substantial boon to throughput; optimizing the data flow paths in the application may help mitigate concerns over latency in sensitive application; removing the single-core MAC computation bottleneck which affects both encrypted and unencrypted SSH datastreams may further improve multi-threaded performance; use of alternative parallel architectures, such as the Cell Broadband Engine, in concert with a specialized parallel SSH implementation may allow cryptographic performance to reach levels far beyond what typical workstation or server class systems can provide. The authors are also interested in techniques that may help divorce the functionality of SSH from the actual applications either as a fully developed library or even a kernel level module. In all of these avenues the end goal is to create a robust, secure, and high performance transport mechanism that can be easily adapted to existing applications and incorporated into new ones.

5.4 Final Thoughts

SSH, 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, simple 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., the National Science Foundation, and the National Institutes for Health for supporting

the development of HPN-SSH. This work would not have been possible without the expertise of co-implementer Michael Stevens and the support of Larry Dunn of Cisco System, Inc. Lastly, my thanks and deepest appreciation goes out to the OpenSSH development team and Tatu Ylönen for their continued development of the SSH protocol.

7. REFERENCES

- [1] ALLCOCK, W., BESTER, J., et al., "Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing," Proceedings of the IEEE Mass Storage Conference, pp. 13-28 April 2001.
- [2] BELLARE, M., KOHNO, T., NAMPREMPRE, C., The Secure Shell (SSH) Transport Layer Encryption Modes. IETF RFC 4344. Internet Engineering Task Force
- [3] 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)
- [4] JACONSON, V., BRADEN, R., BORMAN D., TCP Extension for High Performance. IETF RFC 1323. Internet Engineering Task Force (Web site: www.ietf.org)
- [5] KOHL, J., NEUMAN C., The kerberos network authentication service (V5). Request for Comments (Proposed Standard) RFC 1510, Internet Engineering Task Force, (Web site: www.ietf.org)
- [6] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)", Federal Information Processing Standards Publication 197, November 2001
- [7] OpenSSH 3.8p1 Source Code (channels.c)
- [8] ROSMANITH, H., KRANZLMULLER, D., "glogin - A Multifunctional, Interactive Tunnel into the Grid," *grid*, pp. 266-272, Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04), 2004
- [9] SEMKE, J., MAHDAVI, J., MATHIS, M., "Automatic TCP Buffer Tuning". *Proceedings of SIGCOMM '98 Conference*, Vol. 28, No. 4, pp. 315-323, August 1998
- [10] STEVENS, W. R., *TCP/IP Illustrated Volume 1: The Protocols*, Reading, MA, Addison Wesley Longman Inc. 1994, pp. 289-291
- [11] UBIK, S., CIMBAL, P., "Achieving Reliable High Performance in LFNs", *TNC 2003*. Zagreb, Croatia, May 19-23, 2003.

¹⁰ Incorporated in HP-UX Secure Shell A.04.40.005

