

# Application Layer Network Window Management in the SSH Protocol

Chris Rapier rapier@psc.edu  
Michael A. Stevens mstevens@andrew.cmu.edu

## Problem statement

A number of network applications make use of multiplexed channels inside of a single TCP connection to handle data transfer and/or control information. Because these channels cannot make use of the TCP windows for flow control they must implement their own. This means that a second window can be imposed on top of the existing TCP window. The result of this is that even if the TCP window is correctly sized for the current to produce exceptional FTP performance a user may still encounter dismal throughput under one of these applications. This is because the application window, which is often statically defined, is too small for many typical paths. This forces the connection to slow down to the limit of the smaller of the two windows.

The best current example of this is the SSH2 protocol. It is not uncommon for a user to be sitting on a connection they can utilize less than 1% of because of this double window problem. While a user might not experience any issues in interactive sessions it's a very noticeable problem in bulk data transfers (eg SCP, rsync -essh, sftp, etc) and is common source of frustration – especially for users with access to high performance network connections. A researcher trying to transfer a 400GB dataset over a GigE connection is simply not going to be satisfied with a 1.2 Mbit/sec transfer rate.

Alternatives that meet increased security requirements do exist: Kerberized FTP, Grid Services, and even VPNs. However, the infrastructure investment

associated with them often limits their use to larger institutions. Smaller groups and individuals often don't have the time, expertise, or money to make use of these alternatives and are often forced to accept poor performance. Fixing the windowing problem in SSH will remove that infrastructure barrier and help speed adoption, spur demand, and inspire new applications to make use of high speed encrypted data transfers.

Additionally, if the problems facing SSH can be effectively addressed the solution may be applicable to other protocol implementations that make use of internal flow control mechanisms. Without a doubt, there are notable performance problems with NFS and AFS which may be, at least partially, addressed with programming practices developed through the SSH2 research.

## Approach

The fundamental problem, as stated, is that the maximum effective receive window for any SSH connection will be limited to the smaller of the two buffers (internal or TCP). The SSH buffer is defined to a maximum of 128KB. However, because of the way in which it is updated the maximum effective buffer is half that, or 64 kilobytes. If we take the Bandwidth Delay Product formula ( $BDP = \text{bandwidth} * \text{RTT}$ ) and solve for bandwidth using the buffer size of 64KB we see that throughput will always be no more than  $64k/\text{RTT}$ . For example, on a path with a 80ms RTT the maximum throughput is 6.4Mbits/sec. In comparison, a path with a 15ms RTT the throughput would be 34.4Mbits/sec.

A number of individuals have attempted to resolve this problem by statically increasing the buffer size. However, this is often a piecemeal solution that might work well for an individual along some paths but not on a more widespread basis. A home user simply doesn't need an 8MB buffer while a network researcher would be stifled by anything significantly smaller.

Our approach has been to change the buffer from a statically defined value to one dynamically assigned during execution. By performing a `getsockopt()` function when the SSH window goes through an adjustment the application window can be set to match the TCP receive window size. By doing this at each window adjustment and not just when the connection is created, it is possible to make full use of kernels that incorporate an autotuning TCP stack.

### **Results to Date**

Proof of concept tests undertaken between well-connected hosts on high BDP paths have shown throughput rates of over 195Mbps using the ARCFOUR cipher. An unmodified SSH implementation only achieved 15Mbps using the same (or any other) cipher. All tests indicate that the 195Mbps value is strictly CPU limited. Throughput results can be found below in Figure 1.

### **Continuing work**

While it might seem that the work has been completed, what has been done to date is only a partial solution. The RTT typically used to determine window size (either in an autotuning kernel or by

manual calculation) is the host-to-host time but the internal window should be based on the application to application RTT. Research needs to be performed on where to define the endpoints of the RTT in order to fully maximize network usage. Additionally, enabling out of band communication between the client and server will allow queue length information to be exchanged allowing for even finer tuning of the transfer rate.

We are also continuing to develop SSH and SCP to further increase throughput while maintaining protocol compliance and compatibility with the existing installed `sshd` and `ssh` client base. These enhancements include eliminating the pipe between SCP and SSH, parallel file transfers using multiplexed connections, serial files transfers over one persistent connection, and reducing the total `memcpy`s required to transfer bulk data.

Additionally, one of the original goals of this work was to present these enhancements to the OpenSSH group and have them incorporated into the next release. However, the OpenSSH group has not placed as high a priority on throughput performance as hoped. So it is believed that an independent packaging and distribution of a high performance SSH solution would be the best method of making these enhancements available to the public. It is hoped that the final result of our research will be a stable widely distributed implementation of SSH capable of sustained speeds in excess of 500Mbps/sec.

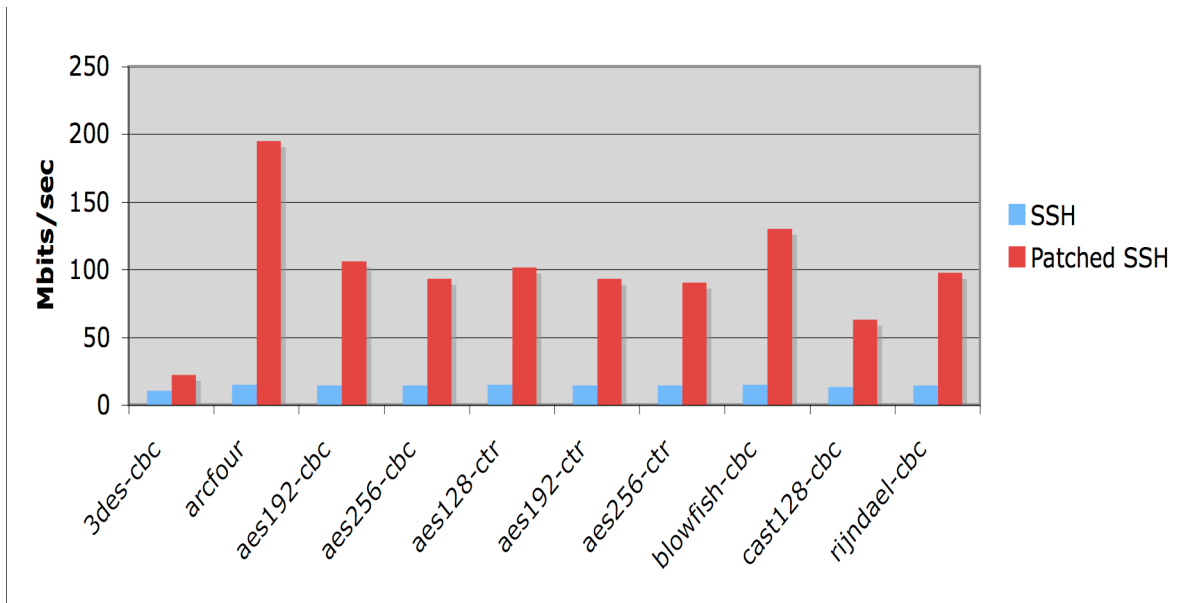


Figure 1: This graph compares the average throughput rate of a standard distribution of SSH (OpenSSH 3.8p1) against SSH with the adjustable window patch. Various ciphers are used. The hosts were a Dual PIII 1Ghz and Quad Itanium 1.3Ghz using linux 2.6. Transfers were from a memory file system to /dev/null.