SSH Performance

Allan Jude, *ScaleEngine Inc.* allanjude@freebsd.org

Abstract

The author describes ongoing development and tuning work to maximize performance of bulk data transfer over SSH. Development includes improvements to the HPN patch sets to resolve problems with dynamic window scaling (both TCP and SSH windows), new functionality to manually specify a larger remote send/receive socket buffer for high latency networks, and development of the new NONEMAC feature. The author also presents detailed benchmarks on the performance tuning required to maximize transfer rates over both local and long-haul networks. A comparative analysis of the performance of various ciphers on modern amd64 hardware is also presented.

Motivation

ScaleEngine uses SSH for bulk data transfer because it is the most convenient way to orchestrate the remote server receiving the data. ScaleEngine has three primary use cases for bulk data transfer:

1. ZFS Replication over LAN and MAN connections. This is used to backup customer data between servers in a data center, and to offsite locations. Data is usually "pulled" by the receiver.
2. ZFS Replication over the Internet. This is used to publish specific datasets to remote servers, such as the TrueOS package repositories. Again, data is "pulled" by the receiver.
3. Rsync. Recordings of customers' live streams are recorded locally at various ingest servers around the world, then transferred to the central storage servers. This data is "pushed" from the recording servers to the storage servers.
4. SFTP. Customers upload original copies of their video content to us via SFTP/SCP, and we want to offer the best possible upload speeds without requiring the customer to use a modified version of SSH.

Since 2011 ScaleEngine has made use of the HPN and NONE Cipher patches for SSH to accelerate ZFS replication, especially over LAN. Removing encryption and decryption from the pipeline made it possible to saturate 1 gbps interfaces with a modest CPU. The HPN patches improved performance of SSH over the Internet by using a larger sliding window. We found the best performance came from setting the TcpRcvBuf option at runtime, to compensate for the bandwidth-delay product.

However, for rsync over SSH, the HPN patches provided only a modest improvement, as the fixed SSH window of 2 MB meant that the bandwidth-delay product allowed for only 250 mbps at 60 ms of latency.

Recently the LAN connections between our storage servers were upgraded to 10 gbps, and we noticed a new problem; even with the NONE cipher, performance was limited by the speed of the MAC (Message Authentication Code) used by SSH. With assistance from AES-NI most higher end CPUs can achieve greater speed by using AES128-GCM, an AEAD (Authenticated Encryption with Associated Data) cipher that provides both encryption and authentication in a single pass, compared with no encryption and just an authentication algorithm.

Background

The test environment consisted of Mercat5 and Mercat6 from the FreeBSD Test Cluster hosted by Sentex:

- Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz
- 6 Cores + Hyperthreading (Turboboost Disabled)
- 32 GB RAM
- Chelsio T580-LP-CR 40 Gigabit NICs (connected back-to-back)
- FreeBSD 11.0-RELEASE-p1
- Base OpenSSH (default): OpenSSH_7.2p2, OpenSSL 1.0.2j-freebsd 26 Sep 2016
- HPN OpenSSH (hpn): OpenSSH_7.3p1, OpenSSL 1.0.2j-freebsd 26 Sep 2016
- Patched OpenSSH (fixed): OpenSSH_7.3p1, OpenSSL 1.0.2j-freebsd 26 Sep 2016

HPN - High Performance Networking

The HPN patches for OpenSSH were first developed in 2004[1] to address the issue of bulk data transfer over SSH. The default SSH window size was 64 - 128 KB, which worked well for interactive sessions, but was severely limiting for bulk transfer in high bandwidth-delay product situations. The first patch in the series enabled a dynamic window, allowing standard TCP window scaling to happen, and offered much better transfer speeds over high latency links. The dynamic window feature only worked on HPN-to-HPN connections, so in the case for HPN-to-NonHPN connections, the HPN patches increased the default window size to a configuration setting that defaulted to 2 MB.

The HPN patches also added a client side configuration option, TcpRcvBuf, that allowed the user to manually specify a receive socket buffer size via the setsockopt() SO_RCVBUF. This greatly increased transfer speeds when a client is receiving from a server. Performance for pushing data from a client to a server was still limited by the defined HPNBufferSize option, often suboptimal.

The HPN patches included a number of other features, including a threaded implementation of AES-CTR[2], and the NONE cipher. The NONE cipher feature allowed a standard SSH session to be established, with encryption, then once the login process is finished, and the data transfer begins, the encryption was switched to a null cipher.

OpenSSH later increased the default SSH window size to 2 MB with the release of version 4.7[3] in 2007.

None Cipher

One of the early features of the HPN patch set was the none cipher. Skipping the encryption process can greatly increase the transfer speed of bulk data where confidentiality is not required. In order to preserve the advantages of using SSH instead of a plain TCP connection, the None Cipher does not engage until the login process completes. The connection starts the same as a regular SSH session, and after the key exchange, the user's login credentials are exchanged, encrypted as normal. At this point, if the NoneSwitch command line option is present, and the session does not have a TTY allocated, the session is rekeyed with NULL encryption. The NONE cipher feature contains a number of protections to ensure it cannot be used for an interactive session, and can never spawn a shell. If the -T switch (manually requesting no TTY) is

1: The first HPN patch was for OpenSSH-3.8.1p1 in July of 2004

2: Multi-Threaded AES-CTR was released as part of HPN13v1 in January 2008

3: https://github.com/openssh/openssh-portable/commit/395ecc2bdeefd86a31562dd4145f370b816814bd

present, the NoneSwitch is automatically disabled. These checks ensure an interactive session is never transmitted in the clear. Even when the NONE cipher is used, a MAC is still applied, so the authenticity of the data is still ensured.

None MAC

With modern hardware support for AES-NI, using the AES-GCM cipher is often faster than using the none cipher. When the none cipher is used data is not encrypted, but a MAC is still applied, to detect modification of the data in transit. Whereas AES-GCM is an authenticated cipher and obviates the need to calculate a MAC as a separate pass. The fasted available MAC in OpenSSH is UMAC-64. On our test system, this limited the throughput of the none cipher to approximately 6,000 mbps, while AES128-GCM reached 8,500mbps. By switching to OpenSSL's null MAC, throughput in excess of 15,000 mbps was achieved, and the MAC was no longer the bottleneck. The same safeguards used for the none cipher are also applied to the none MAC. It cannot be used during an interactive session, or when a TTY is allocated.

Figure 1 shows a comparison of the performance of the various ciphers on the test system. Using the NONEMAC test, no encryption, and no MAC, the patched version of OpenSSH was able to reach more than 80% of the performance of the netcat control transfer, while the NONE cipher was limited by the performance of UMAC64, and fell short of AES-GCM. AES-CTR was only ~10% slower than the NONE cipher, as both were constrained by the calculation of the MAC. The tests for AES-CBC and AES-CTR were then repeated with the NONEMAC. CBC mode saw 40% improvement for 128 bit, and 30% for 256 bit, while CTR mode results were improved by 90% and 80% respectively. The multi-threaded implementation of AES-CTR performed quite poorly; this has been reported[4] to the maintainer of the HPN patchset, and is being investigated. Increasing the number of threads from 2 to 4 made only a modest improvement.

Broken Windows

ScaleEngine found it was necessary to use the HPN TcpRcvBuf settings to get acceptable transfer speeds. Recently when this was investigated, it was determined to be because the dynamic window scaling feature of the HPN patches was not working. During both HPN and non-HPN bulk data transfers it was observed that the TCP window rarely grew beyond 256 KB. When investigating, it was determined that the channel_check_window() function slides the SSH window forward each time half of the window has been consumed. In version 4.7[5] an additional check was added, and the window is slid forward if the consumed portion of the window exceeds 3 times the maximum packet size (32 KB in OpenSSH 7.2). We found that this pattern causes the TCP window to never increase much beyond that size, 128 KB.

One of our initial patches changed the condition to: if (session_is_interactive && consumed > 3*max_packet) || remaining < max_window/2

This kept the window small for interactive sessions, but allowed it to grow for bulk data transfer. At this point it was observed that the TCP window would increase until it reached just more than half of the maximum window size. Experimentation revealed that this behaviour remained the same while the amount the window was slid forward was adjusted.

The HPN patch dynamic window feature increases the maximum SSH window to 1.5 times

4: https://github.com/rapier1/openssh-portable/issues/13
5: https://github.com/openssh/openssh-portable/commit/3191a8e8ba454c0cc27fa8a24a9eed87cd111e4b

the difference between the socket buffer and the maximum SSH window, but only if the socket buffer exceeds the maximum window size. Since this condition is never met, and the SSH window never grows, the TCP window never grows beyond half of that size.

Our patch changed this behaviour to grow the SSH maximum window by 1.5 times the difference between the socket buffer and the unconsumed portion of the SSH window. This condition is now met once the TCP window grows to half of the maximum window, and then the maximum window is increased. The TCP window will grow further, to half of the new maximum. This process continues until the TCP buffer no longer needs to grow to maximize bandwidth, or the maximum size of the socket buffer imposed by the operating system is reached.

Manual Buffer Sizing

The HPN patch set includes a client side configuration option, TcpRcvBuf, to set the local TCP receive socket buffer. This skips the OS auto-tuning and allows the user to specify a larger receive buffer to get better performance.

We have extended this concept with a second client side option, to request a larger TCP receive socket buffer on the remote server, to attain maximum throughput when the client is uploading to the server. This was done by creating a new SSH protocol message, in the SSH2_MSG_LOCAL local modifications range. When this message is received by the server, it performs the required setsockopt() to set the buffer size. The buffer size is capped to a new server-side setting, MaxSockBuf, that defaults to SSHBUF_SIZE_MAX. This allows the server administrator to limit the amount of memory that can be consumed by each connection. The Match user directives can be used to limit the large receive buffer feature to only specific users. This limits the risk of the receive buffer being used to exhaust the server's memory resulting in a DDoS.

Socket Buffer Sizing

There are a number of variables that control the sizing of TCP socket buffers on FreeBSD:

- net.inet.tcp.{send,recv}space - Controls the initial size of the TCP socket buffer
- net.inet.tcp.{send,recv}buf_max - Controls the maximum size for auto-scaling the socket buffer
- net.inet.tcp.{send,recv}buf_inc - Controls the size of each increment of the socket buffer
- net.inet.tcp.{send,recv}buf_auto - Enable auto-scaling of the socket buffer
- kern.ipc.maxsockbuf - The maximum size of any socket buffer

There is some danger of memory exhaustion if socket buffers are allow to grow unbounded. If a server is serving many clients concurrently, a smaller maximum socket buffer is likely warranted. Some applications like nginx allow the administrator to specify a send and receive socket buffer size, which is set with setsockopt(), and bypasses the operating system auto-scaling.

For the case of SSH bulk transfer, it is desirable to avoid increasing the maximum size of the auto-scaling socket buffer, as this will impact all sockets on the system. The TcpRcvBuf feature, and its remote counterpart RemoteRcvBuf, allow the user to manually specify a larger static buffer. This size is bounded by kern.ipc.maxsockbuf. This value is the maximum amount of memory that can be consumed by the buffer, not the maximum size of the buffer. 2048 bytes of buffer consumes 2048 bytes plus 256 bytes of overhead, so to support a 64 MB socket buffer, the maxsockbuf must be set to 72 MB. You can tune the maxsockbuf to a very large value, allowing for extremely high bandwidth-delay products, while keeping the auto-scaling buffer at a reasonable

size, to avoid consuming excess memory on a server that also serves many concurrent clients.

In a high latency environment, such as transporting data intercontinentally, a sufficiently large socket buffer is required to overcome the bandwidth-delay product (BDP). Figures 3 through 6 show appropriate socket buffer sizing, and the impact of the lack of dynamic window scaling in unmodified SSH. Each graph is paired with a log scale version of the same data, because the disparity between the numbers is so drastic.

Limits of Tuning

At this point, this work has reached the limits of what can be achieved with minor patching and OS tuning. DTrace flame graphs ([6] and [7]) show that almost all CPU time is now spent in libc (memcpy, memset, realloc, etc). In order to get more performance, it would likely be necessary to make architectural changes to OpenSSH, and this seems excessive considering the tool is already being abused much beyond its intended purpose.

Figure 2 shows that performances across all ciphers scales linearly with CPU clock frequency. Sadly this means that most Intel Xeon E5-26xx processors cannot yet saturate 10gbps network links, because of their lower relative clock speed compared to the E5-16xx processors used in the benchmarks.

Conclusions

With the dynamic window scaling feature fixed, and some minor OS level tuning to grow the socket buffer more rapidly to maximize throughput even across Long-Fat Networks, increased performance can be had with only server-side modifications. With the TcpRcvBuf feature, "pull" type workloads can skip OS auto-tuning and manually specify a socket buffer size to reach peak transfer rates more rapidly. With a slight modification to the SSH protocol, the client can request that the server set a larger receive buffer, allowing "push" type workloads to reach their potential bandwidth more quickly. The introduction of the none MAC feature, in conjunction with the existing none cipher, avoids SSH becoming CPU bound by encryption or authentication processes. In cases where confidentiality is not required, and authentication is provided by other means (ZFS replication checksums), the performance limit becomes how quickly a single CPU thread can shuffle bytes around in memory. For most users, AES-GCM is likely the best choice, as it offers high performance while still providing both confidentiality and authentication.

6: Client flame graph:
http://www.allanjude.com/bsd/ssh.svg
7: Server flame graph:
http://www.allanjude.com/bsd/sshd.svg

Figure 1: Bandwidth by Cipher

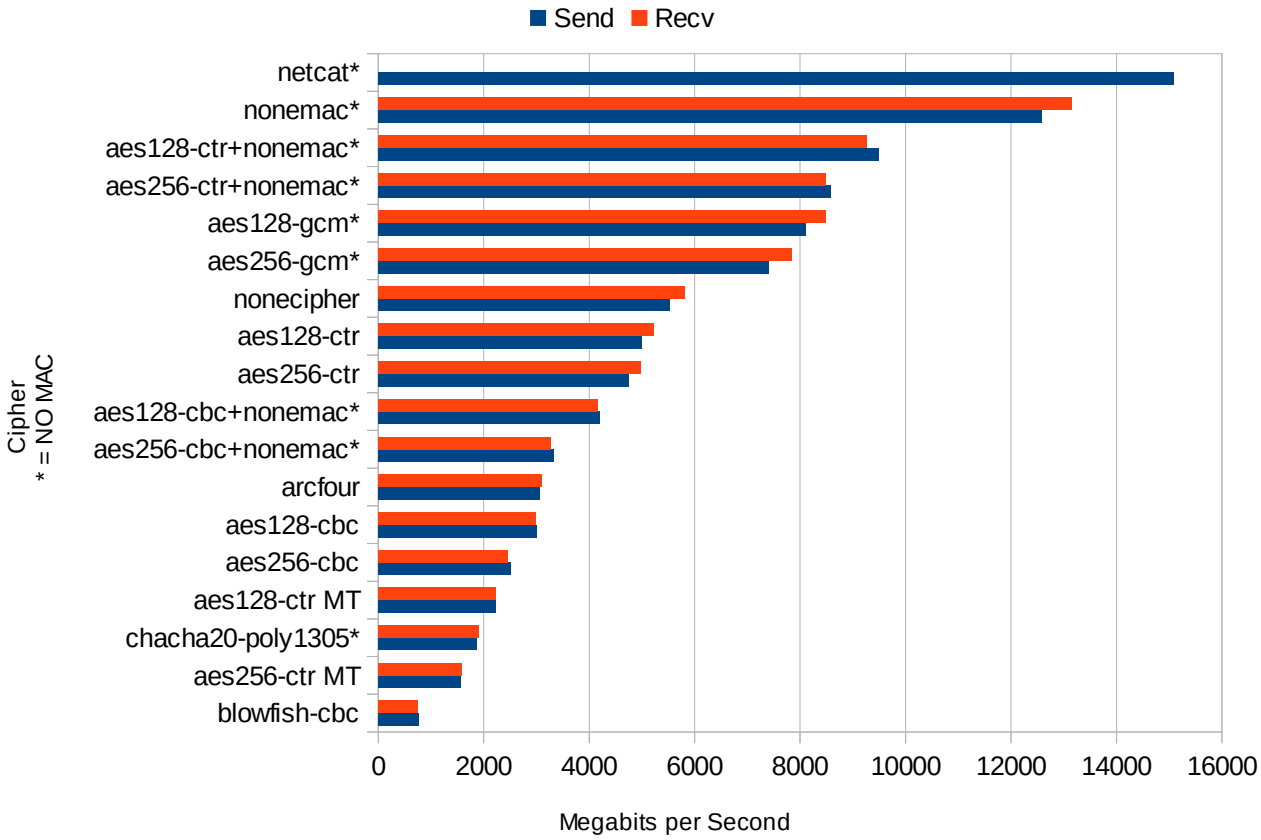

Figure 2: Performance by Cipher vs CPU Frequency

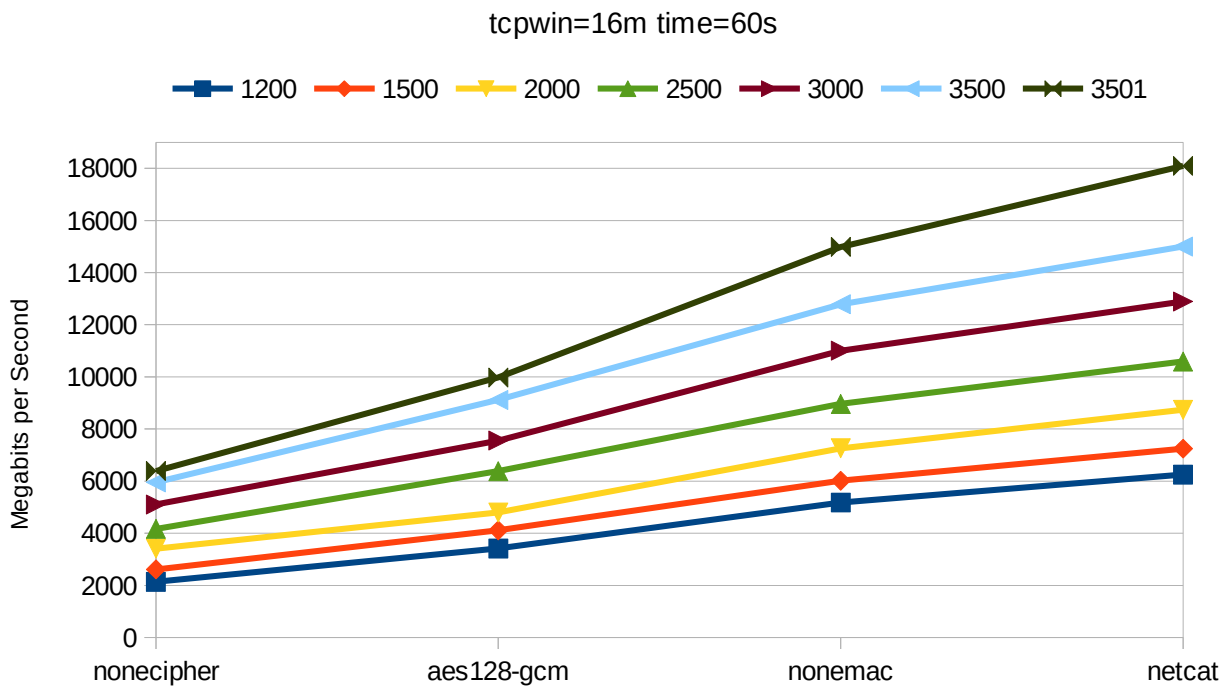

Figure 3: Socket Buffer vs Latency

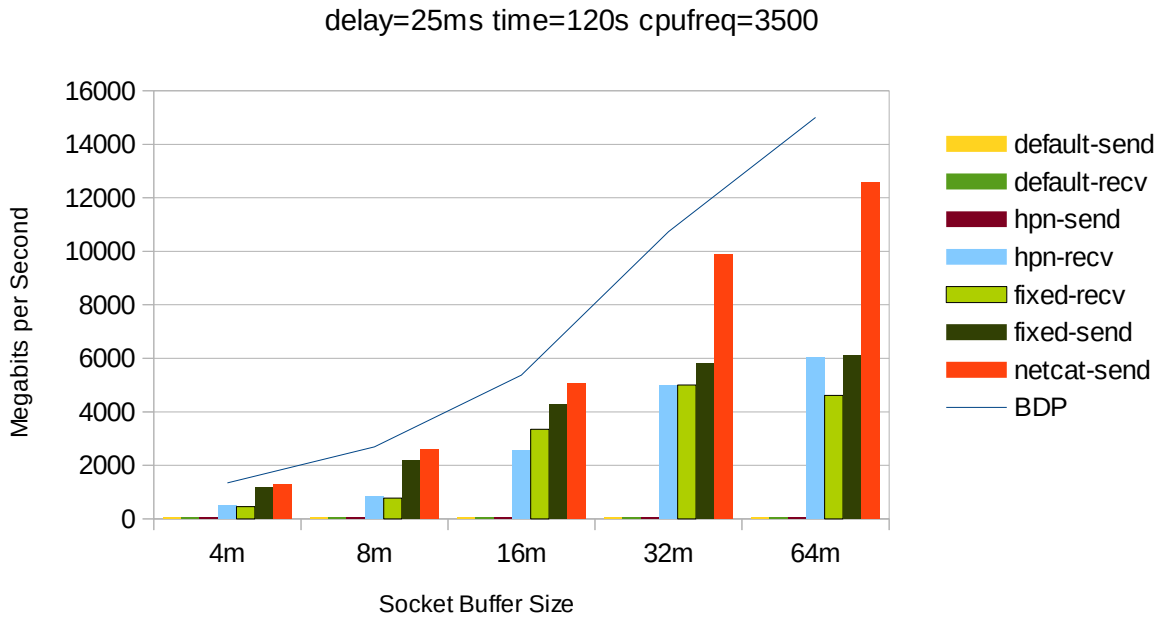

Figure 4: Socket Buffer vs Latency (Log Scale)

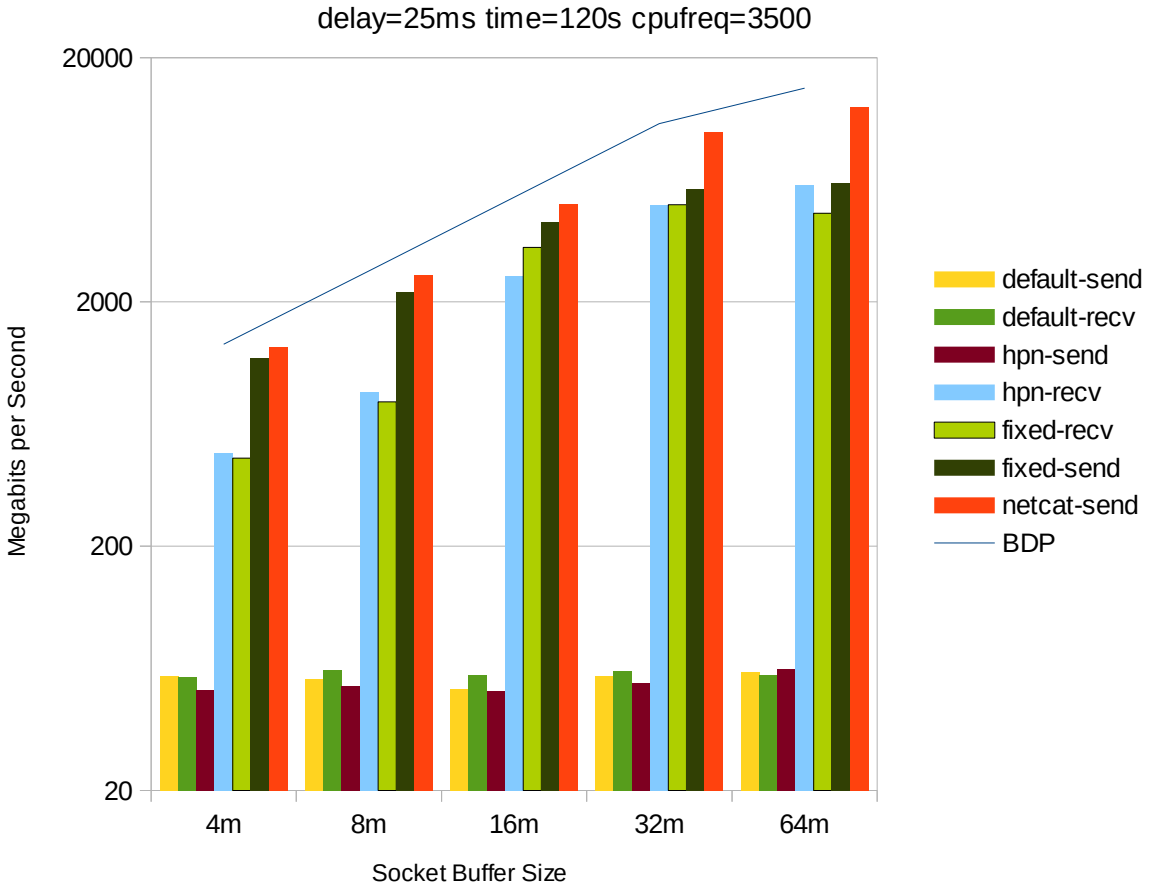

Figure 5: Socket Buffer vs Latency

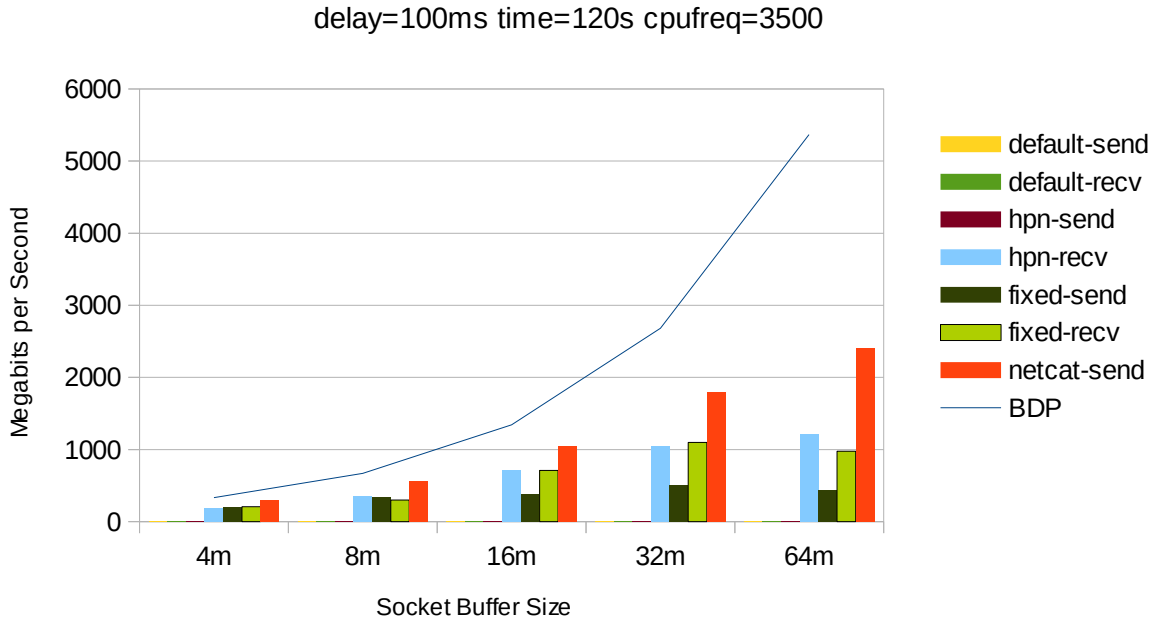

Figure 6: Socket Buffer vs Latency (Log Scale)