

Qsockets

John W. Heffner
Pittsburgh Supercomputing Center
4400 Fifth Avenue
Pittsburgh, PA 15213
jheffner@psc.edu

December 20, 2004

Abstract

This document describes a unique approach for connecting an existing Terascale cluster to a new high speed nationwide network backplane, using a minimal amount of new hardware, yet providing excellent performance. We use a custom software suite named Qsockets and take advantage of the high speed cluster interconnect already in place on the machine, allowing unmodified binary applications to be used as though there is a high speed IP interface attached to each node.

1 Introduction

In 2002, the NSF funded the Distributed Terascale Facility (DTF), with the goal of connecting various terascale computing and storage resources at different centers with a high speed network. This infrastructure, both hardware and software, is known as the TeraGrid [1]. A year later, the Extensible Terascale Facility (ETF) was funded in part to add the Pittsburgh Supercomputing Center and its Terascale Computing System (TCS, also known as Lemieux) [2] to the TeraGrid.

Lemieux is a cluster of 750 AlphaServer ES45 nodes, each of which contains four 1 GHz Alpha ev68 processors and 4 GB of memory. The nodes are connected by the high speed QsNet interconnect [3] made by Quadrics.

The computing resources at original DTF sites were constructed from the ground up for high speed network connectivity. Each node has a Gigabit Ethernet (GigE) interface, and appropriate switches and cabling in place.

However, the TCS had already been constructed and in full production for some time. Integrating it with the ETF would pose some unique challenges. Retrofitting with GigE to every node was not a desirable solution. First, it would have required substantial additional investment to purchase and install

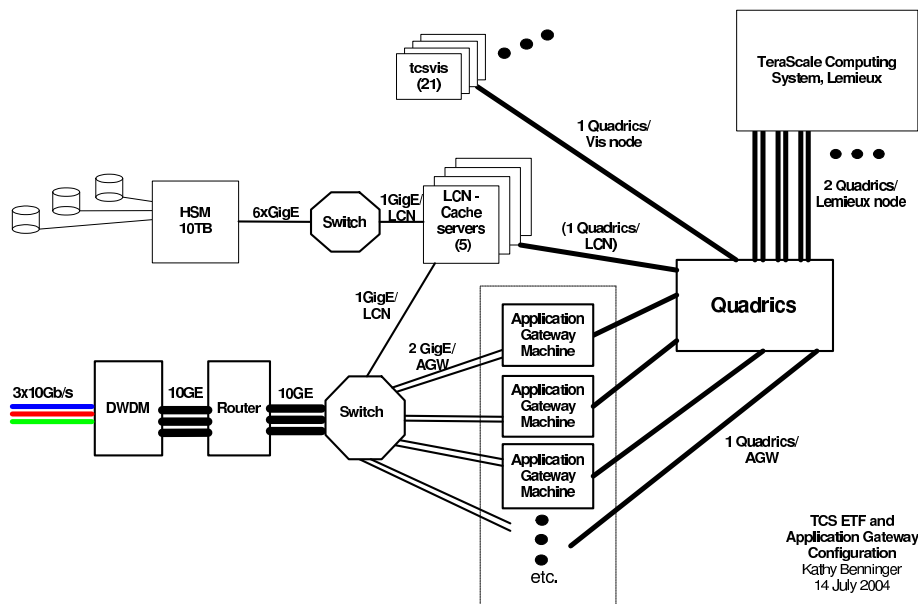


Figure 1: PSC ETF connectivity

750 GigE interfaces, switch ports, and associated cabling. Doing so would also entail substantial down time for the machine.

Instead, it was decided to use the existing QsNet interconnect as the data path into the compute nodes, and use a set of new machines as gateways between GigE and QsNet. Since they do no scientific computation, these gateway machines are relatively inexpensive. The current machines have two Intel Xeon processors and 1 GB of memory, and cost a small fraction of what each compute node does. Each has one Quadrics Elan3 and two GigE interfaces. An additional advantage of using QsNet is that we can take advantage of its remote DMA functionality to reduce memory bus and CPU overhead relative to standard GigE interfaces.

2 Design

The simplest solution considered was to use the IP-over-QsNet driver provided by Quadrics, and do IP routing with the AGW machines. However, this approach was ruled out since performance was not very good, and since the way in which the driver is designed required that both endpoints be running the same operating system on the same hardware. This would have forced us to use expensive Tru64/Alpha machines as AGWs.

To solve these problems, a custom software solution, Qsockets, was designed. It does gatewaying at the sockets level, intercepting calls on client nodes and

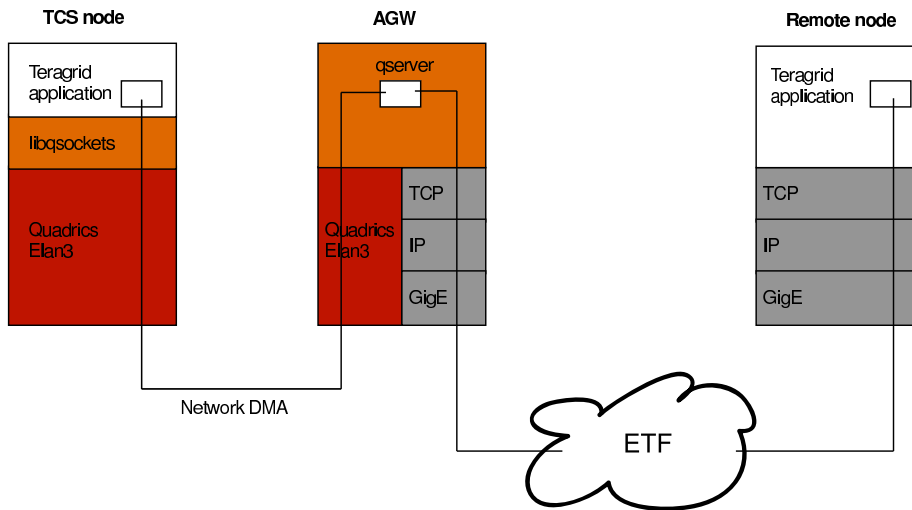


Figure 2: Qsockets layering

executing the calls on the AGW machines. It communicates between the heterogeneous hosts using Elan3 RDMA in a manner inspired by TCSCOMM [4].

On the TCS nodes, Qsockets takes the form of a shared library (`libqsockets`). It contains replacements for each of the standard sockets calls. The library will gateway selected network stream sockets, while passing calls on other file descriptors through to the kernel where they are handled in the normal manner.

A server process (`qserver`) runs on the gateway machines. It interprets the DMA'd requests from clients into the appropriate system calls. The kernel on the gateways handles the TCP/IP networking.

The low level communication that takes place consists of two components: an RPC protocol (called QRPC) for sending commands, and ring buffers for sending and receiving in-band data. The QRPC protocol uses the Elan3 queued DMA functionality to send commands between the server and any number of clients in a reliable manner. The QRPC call types approximately mirror basic socket calls. They are: `socket`, `close`, `shutdown`, `bind`, `connect`, `listen`, and `accept`.

When a `socket` QRPC call is made, the server allocates send and receive ring buffers for that socket in its address space. Also as part of this call, the client and server exchange addresses for front and back pointers into these ring buffers. The nature of a ring buffer allows data input and output to happen entirely asynchronously. When one side completes a read or write on the buffer, it can simply update the appropriate pointer at the other end with an RDMA operation to reflect the change in state.

A memory location for error status is also sent from client to server on the `socket` call. When an error occurs, it is communicated back from the server to the client with a RDMA to this location.

It is worth noting that the `select` call does not require any network communication. To determine if pending data is available for reading, or if space is available for writing, `select` need only look at the local ring buffer pointers. To determine if a socket error is pending, it can read the local error status value.

Somewhat challenging was support for the `fork` and `exec` calls. When these calls are made, open file descriptors are inherited while user space state is segregated or destroyed. Kernels have a separate address space in which to store this shared information; however, Qsockets operates entirely within user space. There is no natural way for Qsockets to maintain proper state across the multiple resulting processes. In order to emulate the behavior of the kernel, Qsockets allocates pre-defined regions of shared memory with the `mmap` call which are inherited in a manner similar to file descriptors (using the `MAP_INHERIT` flag). When the library is initialized, it is able to detect a previous initialization and inherit this state. Multiple processes share the same set of state through use of shared memory locking primitives.

3 Using Qsockets

For an application's socket calls to be handled by Qsockets, the linker must be instructed to search for symbols in `libqsockets` prior to the standard library. For dynamically linked applications, this can be done at run-time by setting an environment variable. (In Linux, the variable is `LD_PRELOAD`; in Tru64, `_RLD_LIST`.) In this manner, an application can use Qsockets without the need to rebuild it or modify it in any way.

Qsockets needs some configuration information from the user. This information is passed to the library through environment variables.

QS_GW: Specifies the QsNet switch port and context number on which the desired `qserver` process is running. Each gateway machine is physically connected to one QsNet switch port; this port number indicates which gateway machine is to be used. Each gateway may run multiple server processes. The context number is used to choose which of the server processes is used. For example, "771.70" would specify the AGW on port 771, and the server with context number 70. This value is required.

QS_V4_ROUTES: A list of network addresses and masks specifying which connections should be gatewayed by Qsockets. If a connection's remote address matches one of these specifiers, it will be gatewayed. This is necessary because an application may need to make connections which should not be gatewayed, particularly connections to hosts that are not connected to the ETF. If this variable is not set, all connections will be gatewayed.

QS_USER_KEY: This should be a string of four decimal 32-bit numbers separated by periods. For example, "53298432.9593824.1092039102.459823417". This 128-bit value is used as a key by the Elan3 communications to prevent

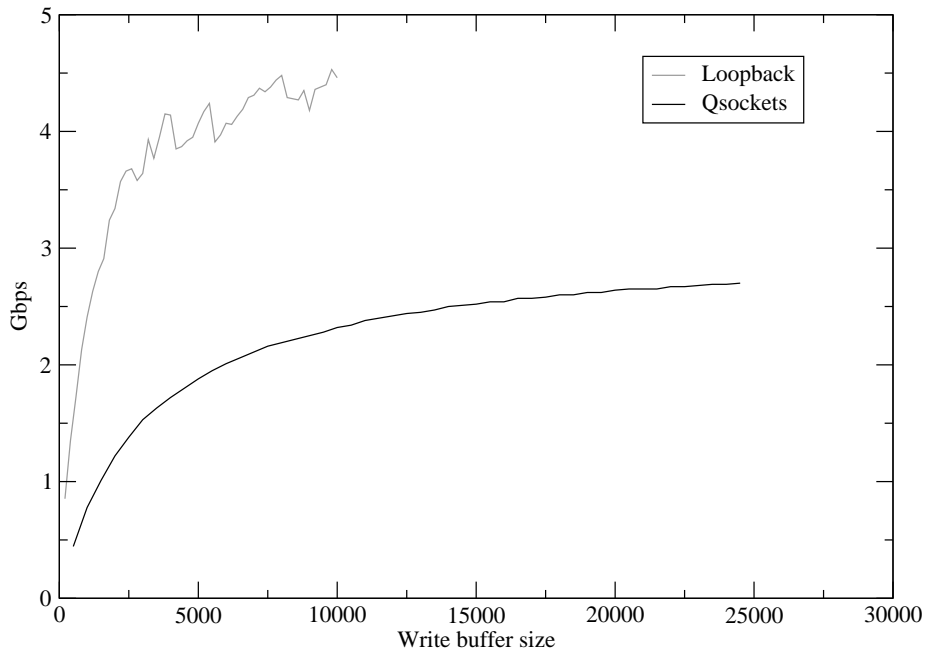


Figure 3: Iperf throughput vs. buffer size.

memory access by unexpected remote processes. The value may be random, but the same value must be used by the `qserver` process. If this variable is not set, a default value will be used.

4 Performance

For the QsNet interfaces, the bandwidth bottleneck is the host PCI bridge. In theory, they run at 66 MHz with a width of 64 bits, for a theoretical maximum speed of over 4 Gbps. In practice, the TCS nodes are capable of about 1.8 Gbps, and the AGWs can reach about 2.5 Gbps. Data transfers rates using `iperf` and FTP applications over Qsockets have reached these speeds.

An important performance consideration for applications using Qsockets is the size of the buffer passed by the application's send and receive calls. Qsockets does not buffer its I/O, instead doing direct DMA from user space. This removes the overhead of a memory copy, but adds a per-call overhead of several microseconds for communication delay. For good throughput, a buffer size of greater than 16 KB is advised. (See Figure 3.) The ring buffer size is defined by a build-time constant, currently 2 MB. Using a buffer larger than this will not increase performance.

Another consideration is the performance of the `select` call. If an application passes only gatewayed socket file descriptors, it is possible to determine the

readiness of these descriptors without any network I/O or syscalls and should be fairly fast – faster than a standard `select` call to the kernel since it avoids the overhead of executing a syscall. There is still some unavoidable overhead in testing each bit in the file descriptor sets, and doing hash table lookups on each file descriptor, but when a descriptor becomes ready, the `select` call should return very quickly since it is polling on values stored in user space.

However, if mixed file descriptors are passed — some handled by the kernel and some gatewayed by Qsockets — the performance of `select` will be degraded. A syscall with its significant overhead must be executed periodically to test the readiness of the kernel sockets. This adds a minimum overhead of one syscall, and may significantly increase the latency when a descriptor becomes ready. If it can be avoided, it is best not to mix file descriptors.

For those in need of a very low overhead means of determining a socket’s readiness, two custom functions have been provided: `qsocket_readable` and `qsocket_writable`. These take as arguments a file descriptor which must be a gatewayed socket, and a cookie handle. The cookie is used to cache a pointer to the socket’s structure and avoid the lookup overhead on subsequent calls. These calls are designed to be made from within a tight loop, and can be executed in tens of nanoseconds in such a fashion. This is useful if an application is polling on readiness of other data, such as an MPI [6] messages.

One significant performance advantage that Qsockets holds is the flexibility it provides for AGW machine configuration. There are many documented problems in high bandwidth-delay networks such as the ETF, and significant research has been put into solutions for these problems. Advances in TCP/IP buffer tuning and congestion control have become available in recent versions of the Linux kernel or are available as patches. The Web100 and Net100 projects provide enhancements to the Linux kernel for instrumentation and tuning of high performance flows [5]. Deploying new or experimental kernel-level changes to a large production system is a difficult and often risky task, and these features are for the most part not even available for the proprietary Tru64 kernel. However, deploying advanced networking features to the AGW machines is relatively simple. These features can greatly aid in diagnosing and correcting performance problems.

Hardware configuration is also flexible. Current AGW machines have GigE interfaces which limit a single flow to 1 Gbps. However, new machines have been purchased to add to the AGW pool which have 10GigE interfaces and two QsNet interfaces, one on each “rail.” The addition of these nodes will instantly add the ability for *any* TCS node to obtain single-flow performance of multiple Gbps.

5 Limitations and future work

One significant operational issue is that misbehaving applications can cause the server problems. For example, an application which unexpectedly terminates due to an uncaught signal will not be able to properly notify the server of its

termination. If the server then tries to issue a DMA to the (now absent) client, the DMA will never complete. Currently, the server will block waiting for this DMA to complete, so that no other connections will be serviced. It may be possible to add a timeout so that if the DMA does not complete in the expected length of time, the connection is considered “dead” and can be closed. However, frequently when an application terminates, there will not be a DMA pending from the server. In this case, the connection will continue indefinitely, bound to its ports and consuming server resources. Because of such complications, and because applications are likely to demand a guaranteed bandwidth, it is likely for the foreseeable future that each job will have its own server processes. The AGW machines will be allocated by the batch scheduler, and the dedicated server processes created and destroyed at job execution time.

One performance limitation is that an application must wait for the RDMA to complete before the send or receive calls return. During this time, it can do no useful computation. This limitation is actually fundamentally due to the sockets API. However, the nature of the low level communications utilized by Qsockets very naturally fits a better model where communications are done asynchronously. It may be useful in the future to implement an asynchronous I/O interface so that computation can be done in parallel with communications. One candidate API would be POSIX AIO. However, at this time fairly few kernels even implement this API, and not many applications are written to it.

One shortcoming of overriding the standard library’s socket calls is that it is not guaranteed to capture all socket operations. If an application or library makes direct syscalls by means of inline instructions, there is no way for Qsockets to trap this. Such an event occurs within `libc` itself. Many versions of `libc` have internal functions which make the actual syscalls. These internal functions are called by the syscall wrapper functions, and by some other functions internal to the library, such as Standard I/O. In order to make Standard I/O work correctly, it was necessary to override these internal functions as well. These functions may vary between different standard libraries. It is also possible to link `libc` in such a way that it is impossible to override these internal functions. In such a case, Standard I/O would not work on gatewayed sockets.

A final issue to address is portability. Qsockets currently works only with Quadrics Elan3 hardware and libraries. While it was designed to be modular so that different low level communications mechanisms might be used, there are currently a number of layering violations for simplification. It may be desirable in the future to properly abstract the lower layer, and add modules for other communications mechanisms. One alternative transport in particular is Sandia Portals [7].

6 Conclusion

Qsockets is a software suite for gatewaying TCP connections over the Quadrics QsNet high speed interconnect via dedicated Application Gateway machines. It replaces the standard sockets API calls on the local end systems (compute nodes

on PSC's Terascale supercomputing cluster). This allows unmodified binary applications high speed access to the ETF despite no accessible IP interfaces. Calls are transported via a custom RPC protocol to the AGWs, and in-band data is transported with a ring buffer using remote DMA. Transport is very efficient, taking advantage of the Quadrics Elan3 hardware to avoid memory copies and data segmentation overhead necessary when using a standard Ethernet device. TCP protocol processing is done on the AGW machines, allowing the end system's CPU to spend more time on useful computation, while also providing the ability to seamlessly add new advanced networking software and hardware features.

Performance has been shown to meet expectations based on host bus speed limitations, and should be adequate for speeds of 1 Gbps. In fact, it should be possible to take advantage of the dual QsNet rail configuration of the TCS and new 10GigE hardware to achieve rates significantly higher than 1 Gbps. The Qsockets architecture also provides some performance optimizations available for demanding applications, if their source can be modified.

The Qsockets architecture was designed specifically for use over the QsNet interconnect, but is also designed in a layered manner so that it may be easily portable to other environments. This technology may be adapted for use in future clusters.

References

- [1] "The TeraGrid: A Primer,"
<http://www.teragrid.org/about/TeraGridPrimer-Sept-02.pdf>, September 2002.
- [2] "Lemieux," <http://www.psc.edu/machines/tcs/lemieux.html>.
- [3] "QsNet High Performance Interconnect,"
<http://doc.quadrics.com/Quadrics/QuadricsHome.nsf/DisplayPages/3A912204F260613680256DD9005122C7>.
- [4] N. Stone, J. Kochmar *et. al.*, "Terascale I/O Solutions," PSC Technical Report CMU-PSC-TR-2003-01, June 2003.
- [5] M. Mathis, J. Heffner and R. Reddy, "Web100: Extended TCP Instrumentation for Research, Education and Diagnosis", ACM Computer Communications Review, Vol 33, Num 3, July 2003.
- [6] "The Message Passing Interface (MPI) standard,"
<http://www-unix.mcs.anl.gov/mpi/>.
- [7] SourceForge, "Project Info – Sandia Portals,"
<http://sourceforge.net/projects/sandiaportals/>.