Accelerating Web Protocols Using RDMA

To appear in the Proceedings of NCA'07, Cambridge, MA, July 2007.

Dennis Dalessandro
Ohio Supercomputer Center
1 South Limestone St., Suite 310
Springfield, OH 45502
dennis@osc.edu

Pete Wyckoff
Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212
pw@osc.edu

Abstract

High-performance computing, just like the world at large, is continually discovering new uses for the Internet. Interesting applications rely on server-generated content, severely taxing the capabilities of web servers. Thus it is common for multiple servers to run a single site. In our work, we use a novel network feature known as RDMA to vastly improve performance and scalability of a single server. Using an unmodified Apache web server with a dynamic module to enable iWARP (RDMA over TCP), we can handle more clients with lower CPU utilization, and higher throughput.

1 Introduction

The web has emerged as the dominant mechanism for communication on the Internet at large. While traffic that is internal to a machine, or a cluster, may use an exotic technology, such as InfiniBand, information that leaves the machine is almost always carried via TCP/IP packets over Ethernet. High-performance computing (HPC) continues to move towards more distributed interactions, with the development of grid portals [14], web services [13], remote data repositories [18], and program composition based on component architectures [2].

A significant example of the current need for high-throughput web-based communication is to access remote databases. The National Cancer Institute hosts a Specimen Resource Locator [18] from which researchers can search and download information about a variety of tumor and tissue samples. This is provided solely as a web service, transporting all data across HTTP or HTTPS. Another need in the biomedical field involves distributed images used in virtual microscopy [24], where a remote machine generates composite high-resolution images on demand for any number of clients in clinical settings.

An example of a high-performance computing project that relies on efficient web data transfers is GridChem [11]. This project provides an infrastructure for computational computing in a distributed environment. The GridChem client application communicates with middleware services to launch and monitor parallel calculations on remote supercomputers using CGI [23] scripts. The software is evolving to take advantage of existing Web Service frameworks, where the transport layer will be XML-encoded documents over HTTPS. Formatting, compressing, and delivering or receiving large files quickly, requires a large amount of processing power on the server.

Today it is common for web content to be in the range of a few kilobytes to a megabyte. It is a sure bet that HPC needs will require larger data transfers and use more processing power to create dynamic content. For the rest of the world, applications such as streaming video and online game play, as well as the sheer increased usage, will drive the processing needs of web servers beyond what is available in a single server.

With the prevalence of data transport via web protocols and the abundance of high-speed networks, sources of performance bottlenecks are often found at the web server itself. Techniques to improve server performance exist, such as round-robin DNS or active monitoring for load balancing, and the use of functional decomposition among multiple cooperating tiers of machines. However, the data transport interface itself serves as a major performance limitation. Improvements at this interface are often overlooked due to the entrenched nature of HTTP over TCP/IP in web communications. The iWARP protocols, or RDMA over Ethernet, offer a way to reduce server load, increase data throughput, and hence overall scalability, while still using the existing TCP/IP transport that is capable of communication in the commodity Internet.

Unfortunately, deployment of new protocols is difficult. As evidenced by longitudinal studies of IP and TCP behavior in web servers [17, 20], adoption of new features is painfully slow. New protocols are even less likely to see usage in the Internet at large, as illustrated

by the slow adoption of IPv6 and SCTP, for instance. Proposing that all web clients switch to a new protocol such as iWARP, especially where that adoption may not directly help an individual client's performance, is clearly not a viable approach.

On the other hand, certain environments can take advantage of iWARP as a web transport today. In high performance computing, communicating clients and servers are often part of the same organization, or part of a collaborative environment like a grid. These machines are likely to share software stacks for authentication, data management, and computation, thus making it easier to promulgate changes across the group. We showcase this by implementing an iWARP module for the popular Apache [1] web server.

2 Background

The Apache HTTP Server [1] has been the most popular web server for the last decade. According to the January 2007 Netcraft survey [19] consisting of over 100 million web sites on the Internet, 60% of sites use the Apache server. It is highly configurable and built around an extensible framework that allows the use of third-party modules to affect the behavior of the server in many ways. Modules have access to the web serving pipeline by registering "hooks" at any of around 30 different locations in the code. Our work takes advantage of these hooks to add configurable support for the iWARP protocols.

2.1 RDMA and iWARP

Remote Direct Memory Access (RDMA) is a technique that has been used, for many years in HPC environments, but recently has seen a growth in popularity. Progressing from the introduction of hardware using the Virtual Interface Architecture (VIA) [4] almost a decade ago, then InfiniBand, and recently 1 Gb/s and 10 Gb/s Ethernet cards based on the iWARP specifications [21], the broader computing market has been recognizing the advantages of RDMA.

The need for RDMA in networking is motivated by the cycle-hungry nature of traditional packet processing, and the multiple copies of data that load the memory bus, limiting scalability and interfering with applications. To avoid these CPU and memory bottlenecks, RDMA allows network adapters to move data directly from one machine to another without involving either host processor. RDMA-capable adapters also bypass the operating system, meaning that user applications interact directly with the network cards to initiate transfers, and designate locations for incoming messages. This avoids the overhead associated with system calls, hardware interrupts, and context switches. As a result, latency and overhead of data movement is significantly reduced.

The iWARP protocol stack consists of an application programming interface, called a verbs layer due to the flexibility of the suggested interface specification [15]. Below verbs are three layers that each add separate functionality to the protocol stack. The RDMA protocol layer [22] provides read and write services that allow data to be transferred directly to and from application buffers without intermediate copies. It is built on a Direct Data Placement (DDP) layer [25] that provides a mechanism for incoming data to be placed directly into user buffers. To function across the byte-oriented streaming TCP protocol, DDP uses Marker PDU Aligned (MPA) Framing [5] to insert markers into the stream that can be used by the receiver to discover frame boundaries.

2.2 Devices and Programming Interfaces

Devices that support the iWARP protocol, known as RNICs, are beginning to appear on the market. An RNIC generally has two important characteristics: operating system (OS) bypass, and zero-copy data movement. The OS bypass mechanism enables a user application to communicate directly with the RNIC device, without invoking operating system functions. The zero-copy aspect means that the device is able to move data directly to and from memory, without the need to make intermediate copies of the data.

Our current software targets the NetEffect RNIC, and builds on our previous work with iWARP [6, 7, 8]. As for the API, we have chosen to use OpenFabrics. Formerly known as OpenIB, the OpenFabrics project supports an open source API and software device interface for a variety of RDMA-capable devices. The goal of the OpenFabrics project is to realize this API in the context of the Linux kernel and to encourage an infrastructure built around this interface. By using the OpenFabrics API, we enable other vendors' RNICs, and even InfiniBand devices, to use our Apache module code unmodified.

2.3 Software iWARP

There is a natural evolutionary path toward using RDMA in a commodity Internet environment. While devices that support RDMA over TCP in hardware are becoming more common and less expensive, more than one software implementation is available. Implementing the RDMA protocol in software is not as efficient as using hardware, but it provides a method to deploy RDMA protocols using traditional Ethernet devices. This means that servers equipped with RNICs can take advantage of that hardware even if clients do not have hardware iWARP.

In previous work, we discussed two related software iWARP implementations, and continue to make the code

```
GET /index.html HTTP/1.1
Host: www.osc.edu
User-Agent: Mozilla/5.0
Connection: Keep-Alive
RDMA: server-writes, ip=10.0.0.15, port=3242,
  stag=642, to=0, maxlen=1048576
```

Figure 1. Example GET request headers.

available to the open source community [16]. One exists purely in user-space, and can be used by any application by linking in a library that generates the appropriate RDMA packets using the standard sockets interface [9]. Another method involves implementing the RDMA calls as a kernel service [10]. Both user-space and kernel-resident applications can invoke RDMA services directly through the operating system, with slightly better performance. One other implementation has been reported [3], though it provides only a sockets-based API and has not yet been made available to the community.

The availability of software implementations of iWARP provides the bridge between current commodity Internet devices and devices that will be able to handle higher throughput networks. While implementing iWARP on a client machine does not improve performance for that particular client, a server with an RDMA-enabled network device can offload its communication work to the RNIC and serve more clients at a faster rate.

It is not necessary to deliver specially-compiled versions of applications to use software iWARP. Web browsers, in particular, often provide a plug-in interface that can be configured at runtime to extend the browser. Software iWARP can be delivered as a plug-in module and effectively turn a regular Ethernet device into a virtual RNIC.

3 Design

The most fundamental design decision we had to make was which web server to use. There are numerous possibilities, including writing our own from scratch. Implementing a trivial web server can be accomplished in just a few hundred lines of C code. However, we decided to use the Apache HTTP Server [1]. This decision was motivated largely by the fact that Apache is the most popular web server in use today. The modular architecture of Apache also made it possible to implement the changes we would need to establish RDMA data transfers. The other web servers we considered would have required substantial changes to their architectures, which would have resulted in a basic rewrite of the entire web server.

3.1 Header Fields

Another important design decision is the changes that would be required to the HTTP protocol. Our goal was to keep any changes to a minimum and to ensure backward compatibility with non-RDMA web clients and servers. At first we considered adding a new method, similar to the existing GET, only implemented for RDMA. It was determined that adding a new method would require rather extensive changes to the Apache code base and duplication of much of the existing functionality that implements GET. It is possible to use the existing GET method and just add a single new entity-header field. With this new header field it is possible to express the information needed to build up an RDMA connection, and conduct the RDMA data transfer.

The new header field we have added is simply "RDMA." The potential information that can be included in this header is: the RNIC IP and port number, the steering tag (STag), the tagged offset (TO), choice of RDMA mechanism, and maximum length. An example GET request is shown in Figure 1. The RNIC IP is necessary as RNIC devices may have two IP addresses, one for the usual TCP data path and another for the RDMA data path. Supplying a port number avoids the need to assign a well-known port number, and permits multiple RDMA-enabled clients or servers on the same machine.

The STag and TO are identifiers used to describe a region in memory for an RNIC. These are explained in more detail in the RDMA specification [22]. An STag is a 32-bit opaque value that maps to a registered memory region. The TO is a 64-bit value that specifies either the exact address in memory to access, or an offset from the memory pointed to by the STag, depending on the type of memory region used. While the RDMA specifications support the use of physical addressing for kernel-resident processes, all our addresses are virtual.

The maximum length parameter in the RDMA header field specifies the size of the buffer allocated for RDMA transfers. For one of our data transfer schemes, discussed in the next section, the server will write data to the client before the client knows the expected content length.

3.2 RDMA Data Transfer Schemes

As mentioned in the previous subsection, there are two RDMA mechanisms for GET requests, although only one is supported in our current implementation. As of now, only the GET method is supported; however, possible RDMA mechanisms for the POST method will be described in this section as well.

Supported in our current implementation is the server-writes scheme. The server-writes case is shown in Figure 2a, for the GET method. As the name implies this is when the server takes the initiative and pushes the data to the client. A client issues a GET request, with an RDMA header that specifies all of the information outlined above, it is assumed by the server that the memory to write data into on the client is already allocated and

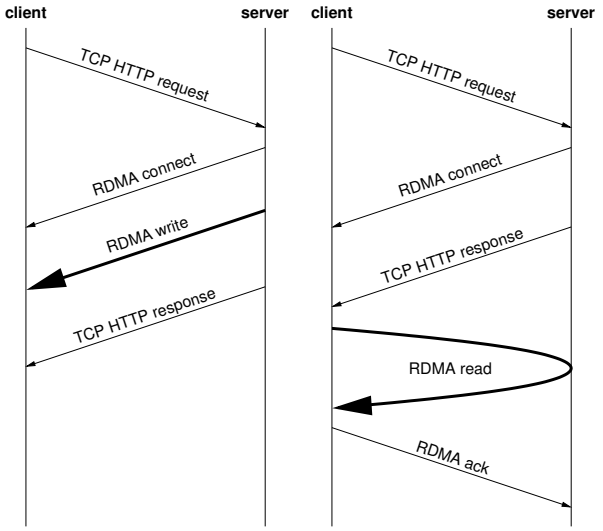

Figure 2. RDMA data transfer schemes: a) server-writes, b) client-reads.

pinned at the time of the request, issues related to memory pinning are explored in section 3.4. Upon receipt of the request the server writes the data to the client. Then to alert the client that the RDMA write has finished, the server sends the usual 200 OK response described in the HTTP RFC [12].

The other RDMA transfer scheme, which is unsupported in our current implementation, for the GET method is the client-reads, shown in Figure 2b. In this case the client issues the GET request omitting the TO and STag. The server allocates and pins the memory needed as well as preparing the buffer. The server then replies with the usual 200 OK response, this time including an RDMA header to signify to the client where to read the data from. After sending the response the server must wait to receive a message from the client signaling that the RDMA read has completed. This message is an untagged message and its acceptance is determined by polling a completion queue. By utilizing the completion queue mechanism it is possible to either implement a blocking poll or a non-blocking poll. For details on completion queues and untagged messages refer to [15].

A POST method would offer two similar choices for RDMA transfers, except instead of server-writes, we now have client-writes, and instead of client-reads we would have server-reads.

A client-writes case would mean the client issues a POST request. The server would then respond with a 200 OK response, that includes the RDMA header, with the information needed for the client to do an RDMA write to the server's buffer. The server then waits for an untagged message to arrive from the client to signal that the client is done doing the RDMA write.

The other RDMA mechanism for the POST method is server-reads. The client issues the POST request, with all of the information in the RDMA header needed for the server to read the data. The server then responds with the usual 200 OK reply to signify that the RDMA read has completed.

3.3 RDMA connection establishment

It is important that the client listen for an RDMA connection before sending its request on the active HTTP connection. The reason for initiating the listen prior to sending the request is to ensure that the client is ready for the connection as soon as the server is. The server actively connects to the client at the RNIC IP address and port number supplied in the HTTP request. Implementing the RDMA connection request from the client to the server would require an extra round-trip communication in the server-writes case: the server must respond with its RDMA port number before the client can initiate the connection. The client-reads case could be modified to let the client initiate the RDMA connection without extra communication steps, but it offers no advantages, except perhaps to bypass overly restrictive firewalls or work around isolated hosts that communicate through NATs.

If a client issuing a GET request specifies a maximum length that is smaller than the requested resource the server can respond with an error message. Likewise for the client if a server specifies the maximum length and the resource the client wants is larger, the client knows not to bother attempting an RDMA GET request.

3.4 Memory Pinning

One of the biggest design questions is how to deal with pinning and unpinning of buffers on the server end. When the client makes a request we allocate and register the memory. The question is how long to hold this? In the server-writes case, write completion indicates that it is possible to release the buffer. In general, there are a number of options, such as waiting for a certain time limit, or letting the client reply with a special DONE message as is the case with the client-reads RDMA GET operation. To minimize the need to send any extra messages, which in the WAN could be especially costly, we decided it best to utilize the completion mechanisms to determine when an RDMA write had finished. This was easy to do and fit well with the HTTP scheme. Once the RDMA write was done we send a successful HTTP reply. The problem was for RDMA read from the client's point of view, there is no way for the server to know when the client has finished reading. So there must be some kind of message sent back from the client. We can not use the HTTP reply as in the server-writes case because the HTTP reply has to be used to inform the client of the STag and TO to RDMA read from. We are left with two options: use the RDMA channel to send a DONE message from the client, or use the TCP channel

to do this. It is less of an impact to simply delay Apache while waiting on the RDMA channel. If we were to use the TCP channel, additional work would be required including possible modifications to the Apache core.

4 Implementation

As mentioned previously, our software takes advantage of hooks within Apache to enable new functionality. The following paragraphs explore where our RDMA module, which we refer to as mod_rdma, interacts with the Apache server.

4.1 Apache Interactions

Child Init The term "child" means a separate server process, not a separate thread. This hook allows for per-process initialization. In mod_rdma, we open the RNIC device and initialize structures for each process to use to represent its state, including queue pair and connection queue descriptors, remote client information, and buffers.

Pre-Connection Each new TCP connection from a client causes this hook to be called. We build a per-connection state structure and register a handler that will clean up the connection state when the server terminates communication with the client.

Insert Filter This hook is invoked by Apache once for each request (many of which may occur per connection), after the request fields have been validated, file system access checks have been performed, and a handler has been chosen for the request. We use this opportunity to look for the RDMA header, and if present, initialize an RDMA connection to the client if one does not already exist, and attach an output filter to Apache's processing chain.

Output Filter Apache uses a "bucket brigade" data structure to pass all data through a series of configurable filters. Most modules use input and/or output filters to modify incoming requests and convert outgoing responses. Our filter holds onto the header of the response, then uses RDMA to send the data (or register it for the client to read), then finally sends the headers down the filter chain once the end of the request has been detected.

4.2 Dynamic Content

Dynamic content, such as that created by PHP or CGI scripts, is a vital component of today's web servers. Due to the design of Apache and its interaction with PHP and CGI processing elements, we need to do nothing special in our RDMA implementation. Apache invokes PHP or Perl to process the request from the client and returns plain text HTML content which our RDMA layer handles as usual.

There are some issues worth mentioning when considering dynamic page content. First, the dynamic nature of PHP and CGI content means that caching is pointless for the server, in most cases. Secondly, the benefit of RDMA really becomes prevalent as shown later in Section 5. Since dynamic page creation requires more CPU activity, the host-resident TCP stack suffers. Since RDMA does not need the CPU to conduct network transfers, it can continue unhindered by the extra load on the CPU. A similar issue arises with encryption (TLS/SSL), in that the CPU must deal with these expensive conversions, starving the network.

4.3 Limitations

There are a few limitations brought about by the implementation. The extra cost of the RDMA connection, as mentioned previously, is one. Another is that our module, as with any real-world RDMA based application, must deal with memory registration. In order to register memory, a fair amount of CPU overhead is needed. We have two options currently for memory registration: pre-register a static buffer, or dynamically register a new buffer for every transfer. Doing the dynamic registration is straightforward, but introduces some overhead in the critical path.

It is possible to register a large chunk of memory ahead of time, for each process. This would occur during startup and not during the critical path for serving files, then the buffer can be reused for every transfer. The downside is that each server thread or process has to register enough memory to serve its largest file size. These issues are explored further in the following section.

The issue of moving the data into a pre-registered buffer is also a problem. Often the data comes from disk, which appears in our module in the form of an open file descriptor. We have to then either read the contents of the file into an RDMA buffer, or we have to memory map the open file descriptor and then register that memory. These both come at a cost, and are a function of file size.

On the other hand, if data is not coming from disk, as is the case with dynamically generated content, there is still a memory copy required to move the data to the RDMA buffer. This puts mod_rdma at a disadvantage, because now we are introducing an extra memory copy, exactly what RDMA strives to avoid.

5 Experimental Results

Next we present the experimental results gathered to test our RDMA-enabled web server. The test environment consists of 15 compute nodes in OSC's Network Research Cluster. These dual AMD Opteron 250 nodes have 2 GB of DDR RAM and an on board Tigon 3 Gigabit Ethernet NIC. For client software we run the popular web-fetch utility, wget, which links in our soft-

ware iWARP library [9]. The clients are connected to a Cisco 6506 switch, which has a latency of around 20 microseconds. Connected to this Cisco switch via 10 Gigabit SR Fibre is a Fujitsu XG1200 switch. The Fujitsu switch, when running in cut through mode has a latency of less than 500 nanoseconds. The servers that we use, are equipped with a NetEffect 10 Gigabit iWARP adapter, and a 10 gigabit Ethernet adapter from Intel. Both cards are PCI-X, although the NetEffect card is in PCIe form factor; it has a PCI-X bridge chip. The servers are also dual AMD Opteron based, and have 2 GB of DDR RAM. Naturally the servers run Apache, with our mod_rdma module linked in.

Since we are interested in improving server performance when the server is under heavy load, we only run on 1 CPU, and we utilize two background processes which calculate a large number of trigonometric computations non stop. By changing the *nice* value of these processes, we ensure they consume nearly 100% of the CPU at all times. In effect we simulate a web server which is completely bogged down by a large number of clients.

5.1 Hidden Costs

As mentioned previously, one of the limitations to our approach is that we have to build up an RDMA connection from the server to the client. In order to avoid this, we would be required to have a full iWARP version of Apache which would require serious internal changes, rather than an add-on for the already existing web server.

In our experiments we have determined that on average, in our LAN it takes around 500 μs to establish an RDMA connection, on an unloaded server. If the server is under heavy load it can take anywhere from 600 μs to a full 3 seconds just to establish the RDMA connection, depending on the number of outstanding clients.

The good news is that this connection cost must only be endured one time. A client can make multiple requests for files reusing the existing connection, according to the HTTP protocol.

Size	No Load	Full Load
2 kB	26.7 μs	1.56 s
4 kB	33.7 μs	1.56 s
500 kB	619.7 μs	1.40 s
1 MB	1350.2 μs	1.56 s
8 MB	12606.6 μs	1.88 s

Table 1. Memory registration costs.

Another hidden cost is memory registration. From previous experience we know that memory registration is very CPU intensive. The data shown in Table 1 illustrates this point. When the CPU is not loaded, it is only a matter of microseconds to register even a large chunk of memory. On the other hand when the CPU is under heavy strain, the time to register memory is significantly increased. To register even a small amount of memory requires more than a full second when the processor is heavily utilized.

Due to the fact that memory registration is so costly when the processor is burdened, this could have an impact on performance, and is the reason that in the following graphs we show two cases, one which we refer to as the static case and the other as the dynamic case.

In static registration, we allocate a single buffer, to be reused for each file requested. As mentioned before this may not be a desirable scheme for the real world, but part of our ongoing research is into a new way to transfer data that would enable us to register memory in a more efficient manner, where the effects would not be seen during the critical path here. The static scheme is meant to show what the expected performance could be in a best case scenario. In dynamic registration, a buffer is registered for each request. his certainly adds extra cost to the time to service a client, but it is a more efficient way to manage system resources.

5.2 Single Client Performance

We now turn our attention to the performance of a single client. We compare performance of a server that is unloaded, and a server that is under heavy CPU load, as we described previously. Figure 3a shows the performance of the unloaded scenario. We see from this figure that TCP takes less time to get a single file. The reasons for this are that the iWARP case must handle issues of memory registration, copying and moving data from Apache buffers to RDMA buffers. On the client side, there is extra processing to handle the iWARP protocol stack in software.

What is somewhat surprising is that the dynamic registration case is slightly faster than the static registration case. One other detail about the static and dynamic schemes should shed light on this. In the static scheme since we already have the buffer allocated and registered, we must copy the contents from the open file descriptor that Apache gives to our module. To do this, we have to memory map the file, then to avoid the memory registration we have to do a memcpy. With the dynamic case, we simply memory map the file and register that buffer. There is no memory copy, but there is a memory registration. What this data shows is that when the CPU is under light load it is less costly overall to register memory than it is to copy it.

Figure 3b shows the same single file retrieval test, but now with a large artificial load on the server. First, we note that TCP outperforms both iWARP cases for 2 kB and 4 kB messages, but as we increase beyond that TCP performance suffers dramatically. The reason for this is that TCP requires much more use of the CPU to trans-

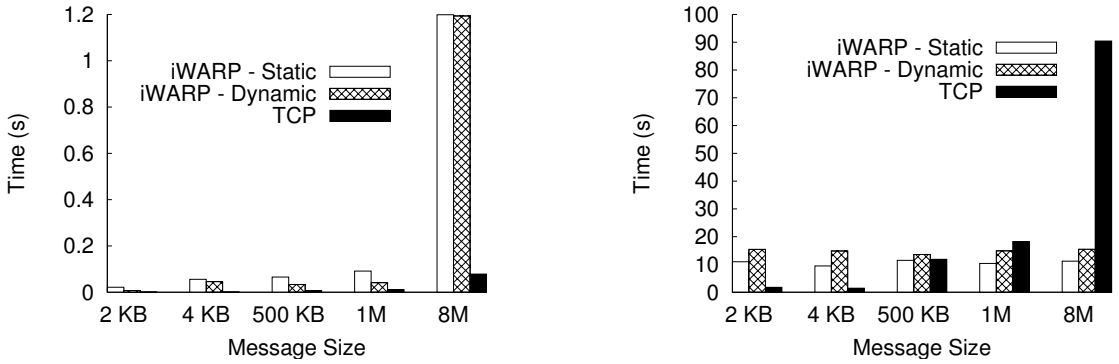

Figure 3. Single file retrieval: a) no CPU load, b) full CPU load.

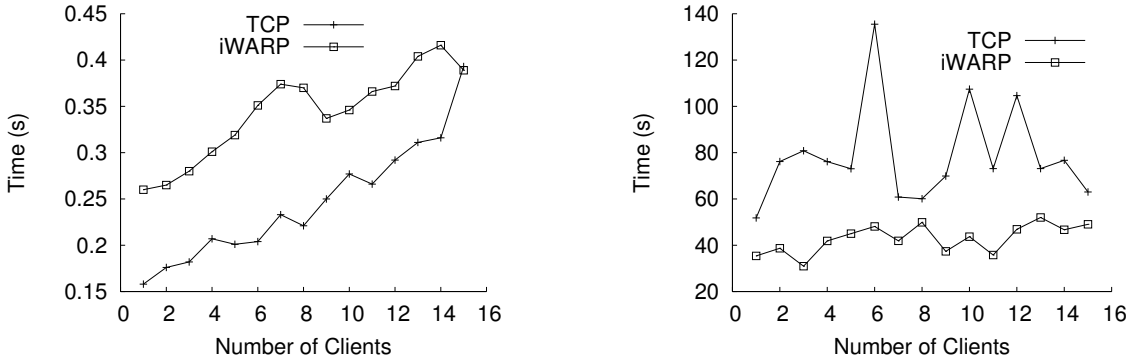

Figure 4. Multiple clients: a) no CPU load, b) full CPU load.

fer data, whereas iWARP, which does not involve the host processor, is relatively unaffected. Another striking issue is that with load, the static case outperforms the dynamic case, unlike in Figure 3a. Since the CPU is heavily utilized, the cost for registering memory is more than the cost to copy memory because registering memory is a CPU bound task, whereas copying memory is bound by the memory bus, in terms of performance.

5.3 Multi-client Performance

Next we look at the performance characteristics of dealing with multiple connections. Again we compare two cases, one when the server is not loaded, and the other when the server is heavily loaded. The case when the server is not loaded is meant to simulate a few clients accessing files, nothing demanding. In the fully loaded case, even though we test with 15 actual clients, the load that we add is meant to simulate a much greater number of connections and a more demanding request load.

For these experiments we have decided to go with the case where we have a static buffer, since it was found to perform a bit better when the server is experiencing a heavy load. In the tests that follow, each client downloads five 1 MB files by using the "recursive" option to wget, without breaking the initial TCP connection.

In Figure 4a we see the effect of increasing the number of clients. Clearly TCP outperforms iWARP, but as the number of clients increaes, and thereby the load, TCP starts to degrade in performance, as is evidenced by the relative slope of the line, in that it is noticeably steeper than that of iWARP. It is very likely that increasing the number of clients beyond 15 would show that iWARP outperforms TCP with a large number of clients.

To test this hypothesis we look at Figure 4b, where we show the performance under heavy CPU load. In this configuration, iWARP outperforms TCP. In fact there are a few cases where TCP is far more than double the time for iWARP. The performance of iWARP also varies little during the test, where as TCP varies greatly at various points in the graph.

The message is clear, when the server load is increased, TCP performance will degrade due its reliance on the CPU. iWARP on the other hand, does not require the CPU to conduct network transfers, and thus shows a large improvement over TCP.

6 Conclusion and Future Work

In this paper we have demonstrated the feasibility of adopting emerging high-speed communication protocols in an evolutionary way on existing TCP/IP and Ethernet networks. Our Apache module offloads much of the work of data serving to a hardware iWARP card, reducing host CPU load and thus allowing the server to accommodate more simultaneous clients, in less time. This approach is completely backward-compatible with

existing servers and clients. No modification to the Apache server core code was required; however, clients must be modified to take advantage of the iWARP protocol, though this could easily be in the form of a browser plug-in.

In the future we plan to address the client compatibility issue by enhancing an existing freely available software iWARP implementation with a sockets interface that can be linked into unmodified clients such as Firefox and wget. We also hope to broaden the scope of our work to encompass PUT-like requests such as are used with WebDAV and other protocols, and to study the interaction of proxy or server caching and RDMA. We also would like to engage some application communities to deploy the RDMA web server and clients in a production setting.

In other related work, we plan to investigate and implement the necessary kernel modifications to enable sending data with RDMA in ways that would remove the burden of memory registration or copies from the critical path.

References

[1] Apache Software Foundation. Apache HTTP Server project. http://httpd.apache.org/, 2005.

[2] R. Armstrong, D. Gannon, A. Geist, et al. Toward a common component architecture for high-performance scientific computing. In *Proceedings of High-Performance Distributed Computing*, 1999.

[3] P. Balaji, H.-W. Jin, K. Vaidyanathan, and D. K. Panda. Supporting iWARP compatibility and features for regular network adapters. In *Proceedings of the IEEE Cluster 2005 Conference, RAIT Workshop*, Burlington, MA, Sept. 2005.

[4] Compaq, Intel, and Microsoft Corporations. Virtual Interface Architecture specification, Dec. 1997.

[5] P. Culley, U. Elzur, R. Recio, S. Bailey, and J. Carrier. Marker PDU aligned framing for TCP specification. http://www.ietf.org/internet-drafts/draft-ietf-rddp-mpa-02.txt.

[6] D. Dalessandro. RDMA over TCP/IP: The next step in ethernet technology. In *Proceedings of the 2005 Commodity Cluster Symposium: On the Use of Commodity Clusters for Large-Scale Scientific Applications*, Greenbelt, MD, July 2005.

[7] D. Dalessandro and P. Wyckoff. A performance analysis of the Ammasso RDMA enabled Ethernet adapter and its iWARP API. In *Proceedings of the IEEE Cluster 2005 Conference, RAIT Workshop*, Boston, MA, Sept. 2005.

[8] D. Dalessandro and P. Wyckoff. Initial Performance Evaluation of the NetEffect 10 Gigabit iWARP Adapter. In *Proceedings of the IEEE Cluster 2006 Conference, RAIT Workshop*, Barcelona, Spain, Sept. 2006.

[9] D. Dalessandro, P. Wyckoff, and A. Devulapalli. Design and implementation of the iWarp protocol in software. In *Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems*, Phoenix, AZ, Nov. 2005.

[10] D. Dalessandro, P. Wyckoff, and A. Devulapalli. iWarp protocol kernel space software implementation. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06), Communication Architectures for Clusters Workshop*, Rhodes, Greece, Apr. 2006.

[11] R. Dooley, K. Milfeld, C. Guiang, S. Pamidighantam, and G. Allen. From proposal to production: Lessons learned developing the computational chemistry grid cyberinfrastructure. In *Proceedings of the Workshop on Grid Applications at GGF 14*, Chicago, IL, June 2005.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[13] I. Foster, C. Kesselman, et al. Grid services for distributed system integration. *Computer*, 35, 2002.

[14] D. Gannon, G. Fox, M. Pierce, et al. Grid portals: A scientist's access point for grid services. Technical report, Global Grid Forum, Sept. 2003.

[15] J. Hilland, P. Culley, J. Pinkerton, and R. Recio. RDMA protocol verbs specification. http://www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf, Apr. 2003.

[16] iWarp Team. Software implementation and testing of iWarp protocol. http://www.osc.edu/research/network_file/projects/iwarp/iwarp_main.shtml, 2006.

[17] A. Medina, M. Allman, and S. Floyd. Measuring the evolution of transport protocols in the internet. *ACM SIGCOMM Computer Communication Review*, 35(2):37–52, Apr. 2005.

[18] National Cancer Institute. Specimen resource locator. http://pluto3.nci.nih.gov/tissue/default.htm, 2002.

[19] Netcraft. Web server survey. http://news.netcraft.com/archives/2007/01/05/january_2007_web_server_survey.html, Apr. 2006.

[20] J. Padhye and S. Floyd. On inferring TCP behavior. *ACM SIGCOMM Computer Communication Review*, 31(4):287–298, Oct. 2001.

[21] RDMA Consortium. Architectural specifications for RDMA over TCP/IP. http://www.rdmaconsortium.org/.

[22] R. Recio, P. Culley, D. Garcia, J. Hilland, and B. Metzler. An RDMA protocol specification. http://www.ietf.org/internet-drafts/draft-ietf-rddp-rdmap-04.txt, Apr. 2005.

[23] D. Robinson and K. Coar. The Common Gateway Interface (CGI) Version 1.1. RFC 3875, Oct. 2004.

[24] B. Rutt, V. S. Kumar, T. C. Pan, T. M. Kurc, U. V. Catalyurek, and J. H. Saltz. Distributed out-of-core preprocessing of very large microscopy images for efficient querying. In *IEEE Cluster 2005 Conference*, Burlington, MA, Sept. 2005.

[25] H. Shah, J. Pinkerton, R. Recio, and P. Culley. Direct data placement over reliable transports. http://www.ietf.org/internet-drafts/draft-ietf-rddp-ddp-04.txt, Feb. 2005.