Design and Testbed Evaluation of RDMA-based Middleware for High-performance Data Transfer Applications

Yufei Ren, Tan Li, Dantong Yu*, Shudong Jin, Thomas Robertazzi
Email: yufei.ren@stonybrook.edu, tan.li@stonybrook.edu, dtyu@bnl.gov, shujin@notes.cc.sunysb.edu, tom@ece.sunysb.edu

Stony Brook University, Stony Brook, New York, USA
**Brookhaven National Laboratory, Upton, New York, USA*

Design and Testbed Evaluation of RDMA-based Middleware for High-performance Data Transfer Applications

Yufei Ren, Tan Li, Dantong Yu*, Shudong Jin, Thomas Robertazzi
Email: yufei.ren@stonybrook.edu, tan.li@stonybrook.edu, dtyu@bnl.gov, shujin@notes.cc.sunysb.edu, tom@ece.sunysb.edu

Stony Brook University, Stony Brook, New York, USA
**Brookhaven National Laboratory, Upton, New York, USA*

Abstract

Providing high-speed data transfer is vital to various data-intensive applications supported by data center networks. We design a middleware layer of high-speed communication based on Remote Direct Memory Access (RDMA) that serves as the common substrate to accelerate various data transfer tools, such as FTP, HTTP, file copy, sync and remote file I/O. This middleware offers better end-to-end bandwidth performance than the traditional TCP-based alternatives, while it hides the heterogeneity of the underlying high-speed architecture. This paper describes this middleware's function modules, including resource abstraction and task synchronization and scheduling, that maximize the parallelism and performance of RDMA operations. For networks without RDMA hardware acceleration, we integrate Linux kernel optimization techniques to reduce data copy and processing in the middleware. We provide a reference implementation of the popular file-transfer protocol over this RDMA-based middleware layer, called RFTP. Our experimental results show that our RFTP outperforms several TCP-based FTP tools, such as GridFTP, while it maintains very low CPU consumption on a variety of data center platforms. Furthermore, those results confirm that our RFTP tool achieves near line-speed performance in both LAN and WAN, and scales consistently from 10Gbps Ethernet to 40Gbps Ethernet and InfiniBand environments.

Keywords: Distributed systems, Middleware, Network protocols, Remote

1. Introduction

Data-intensive applications, including high energy and nuclear physics, astrophysics, climate modeling, nano-scale materials science, genomics, and financing, are expected to generate exabytes of data over the coming years, which must be transferred, visualized, and analyzed by geographically distributed teams of users. High-performance network capabilities must be available to these users at the application level in a transparent, virtualized manner. Moreover, the application users must have the capability to move large datasets from local and remote locations across network environments to their home institutions.

To support these data-intensive distributed applications, much work has been done to accelerate data transfer over high-speed networks. There are two main approaches. The first is protocol offload and hardware acceleration. They are among the most desired techniques to achieve high data transfer rate with marginal host resource consumption. The TCP/IP Offload Engine (TOE) is one of the early examples of protocol offload to meet above requirements. The idea of TOE is to use a dedicated hardware module on a network adapter card to execute the TCP/IP internal operations, such as segmenting, framing, packet reassembling, timing, and congestion and flow control. Research and implementation works Feng et al. (2005); Jang et al. (2008) have shown that TOE is a cost-effective technique to free the host processors from onerous TCP/IP protocol processing, and therefore to enhance the concurrency between communication and computation. Thereafter, Remote Direct Memory Access (RDMA) was proposed as another hardware-based Protocol Offload Engine (POE) realization to enable high-speed and low-latency data transfer with much less CPU resource involvement, and its use has recently become popular when converged Ethernet and data center bridge technologies were proposed and implemented. RDMA has the capability to move bulk data from the source host memory directly to the remote host memory with kernel-bypass and zero-copy operations. Three different RDMA implementations, InfiniBand (IB), Internet Wide Area Protocol (iWARP), and RDMA over Converged Ethernet (RoCE), are available today to offload RDMA protocol stack to different network architectures. Each of them has different trade-offs between system performance, cost, compatibility, and implementation complexity.

The second approach is software optimization and kernel pass techniques. This approach is based on the observation that data-intensive applications impose a formidable requirement on the CPU usage of hosts. In particular, frequent data copy within the host memory space is expensive, but often redundant and inefficient. For example, in a typical file transfer application, file data is first read from disk drivers to the kernel memory, and then copied to the user space. From then on, the data is sent via the socket interface, and thus it is copied from the user space back to the kernel space, i.e., the socket buffer before actual delivery to the network drivers. Various optimization techniques have been proposed in modern computer systems to minimize data copy. In fact, the same idea has been adopted by Web server performance optimization Suzumura et al. (2009). In the current Linux systems, on the other hand, we have seen more standard kernel primitives to facilitate such performance optimization, for example, via the `sendfile` and `splice` primitives Linux (2012).

In addition to achieving near line-speed data transfer, another challenge is to integrate both the hardware acceleration and software optimization techniques to support data transfer applications. For example, it is important to manage a heterogeneity of underlying RDMA architectures as described above for user applications. File Transfer Protocol (FTP) is one of the most widely-used services to transfer bulk data. Existing data transfer applications built on top of the native TCP/IP may not be able to fully utilize the available bare-metal bandwidth of the next-generation high-speed networks, because of the considerable load on host CPU caused by kernel-based TCP/IP realization Bierbaum (2002); Yeh et al. (2002). Various RDMA implementations, as introduced above, offer opportunities to enhance the performance of data transfer service. However, despite of the emergence of industry standards such as OpenFabrics Enterprise Distribution (OFED) OpenFabrics Alliance (2012), it could be a distraction if the user applications have to directly manage the underlying RDMA devices.

In this paper, we describe the design of a middleware software that provides RDMA capability Ren et al. (2012a). This middleware integrates network access, memory management, and multitasking in its core design. We address a number of issues related to its efficient implementation, for instance, explicit buffer management and memory registration, and parallelization of RDMA operations, that are vital to delivering the benefits of RDMA to applications. Built on top of this middleware, an implementation and experimental evaluation of a RDMA-based FTP software, RFTP, are described. This application has been implemented by our team to exploit the full capabilities of advanced RDMA mechanisms for ultra-high speed bulk data transfer applications on U.S. Department of Energy, Energy Sciences Network (ESnet) ESnet (2012a). Our contributions include the following:

- We design the core of our middleware software that offers data transfer and access primitives. This core is designed to have a multi-threaded architecture and facilitates multi-stream data transfer to exploit parallelism of RDMA operations.

- We explore the most efficient operations among various options in RDMA to implement high-speed data transfer over a variety of networks. We implement buffer management and task synchronization mechanisms to allow maximum buffer reuse, reduce synchronization expense, and minimize RDMA connection setup cost. Thus it serves as a reliable base for the development of data transfer applications in high-speed networks.

- We design an improved data transfer method in the middleware that uses software-based kernel techniques to reduce data copy overhead. This method is based on the `sendfile` and `splice` primitives to directly move data between file systems and network interfaces.

- We propose and implement a RDMA extension for the File Transfer Protocol defined in RFC 959. We provide a comprehensive set of evaluations to this RFTP on top of various RDMA implementations. The experiments are done on actual Linux clusters over a variety of testbed platforms and networks, including the world's pioneering 100Gbps ultra-speed wide-area testbed. The experimental results validate our middleware design.

The unique contributions of our work include the follows. First, we design the first middleware layer to support RDMA-based data transfer. Second, we integrate both hardware acceleration and software kernel optimization to eliminate the host processing overhead for data-intensive applications. Third, we report the first experimental results on the data transfer performance over long-haul 100Gbps networks.

The rest of this paper is organized as follows. Background information and related work are presented in Section II. Section III describes the design of our middleware layer. The design of RFTP will be detailed in Section IV. In Section V, we describe the hardware and software setup of our test platforms and present the experiment results. A conclusion of this paper is given in Section VI.

2. Background and Related Work

2.1. RDMA Architectures

The outstanding performance benefit of RDMA technology for data center networks and high performance computing has attracted a great deal of interests in both academia and industry. The original RDMA architecture, known as InfiniBand IBTA (2006), supports top-down RDMA message service with its own layer-4 to layer-2 (and sometimes even layer-1) implementation of the OSI protocol stack. It supplies the message passing service to applications with all the protocol processing operations offloaded to specialized hardware. Unlike the best-effort frame delivery service in Ethernet, the link layer of InfiniBand provides reliability and strong ordering in packet transfer through its credit-based flow control and virtual lane mechanisms. However, the extension of IB on WAN requires proprietary hardware to encapsulate IB into Ethernet frame, which prevents it from being widely adopted.

Two other implementations, Internet Wide Area RDMA Protocol (iWARP) and RDMA over Converged Ethernet (RoCE), were proposed to extend the advantages of RDMA and InfiniBand to ubiquitous IP/Ethernet-based networks and integrate the traditional network structure with advanced RDMA mechanisms. iWARP offloads the whole TCP/IP stack. The Direct Data Placement (DDP) layer of the iWARP stack implements the zero-copy and kernel-bypass, and transfers data in user-space buffer directly to remote application memory. It enables RDMA to seamlessly run over ubiquitous IP networks, and thus over today's Internet. RoCE techniques allow the IB transport protocol to run over Ethernet and offer the advantages of IB in the ubiquitous Ethernet environment. Compared to iWARP, RoCE is a more natural extension of message based transfer, and therefore, offers better efficiency than iWARP.

One objective of our middleware is to support applications across all aforementioned RDMA architectures. We build our system with the common Verbs Application Programming Interface (API) in OpenFabrics Enterprise Distribution (OFED) OpenFabrics Alliance (2012), a unified, cross-platform, transport-independent software stack for RDMA. The OFED offers

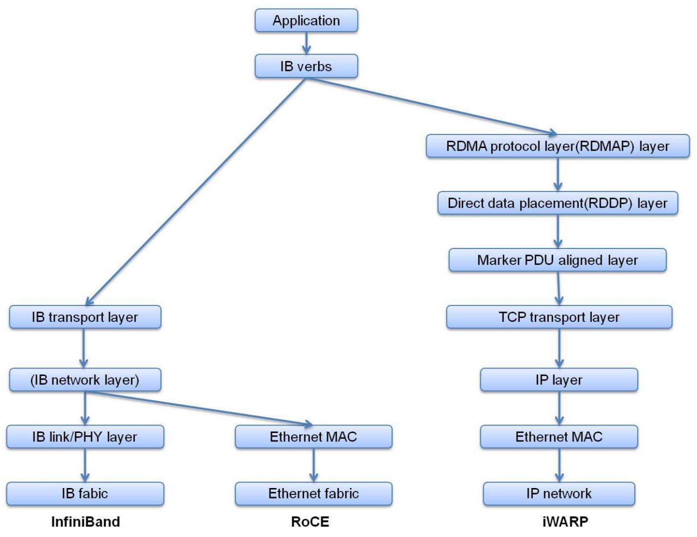


Figure 1: Applications over different RDMA protocols

a uniform application programming interface, which is known as native IB verbs, to access various RDMA architectures. Applications mainly use the *libibverbs* and *librdmacm* libraries. The layered structure, with applications on top, is shown in Figure 1. The OFED software also offers several middleware packages, such as IP over IB The Internet Engineering Task Force (IETF) (2006) and Sockets Direct Protocol (SDP) IBTA (2010), to allow socket based applications running over RDMA devices without rewriting the program. The User Direct Access Programming Library (uDAPL) DAT Collaborative (2002) also provides RDMA capabilities for applications, and has been used in other studies Danalis et al. (2008a,b). Yet, these extensions introduce additional overhead and performance penalties compared to the native RDMA IB verbs interface Lai et al. (2009b).

2.2. RDMA Communication Models

RDMA provides two message transfer semantics: channel semantic and memory semantic. The channel semantic, SEND/RECEIVE, is also referred as two-sided operations in RDMA since the kernels at both the data source and the sink are involved in the data transfer after a connection is established Frey and Alonso (2009). The communication channel between the source and the sink is modeled as a queue pair (QP). Each QP consists of one send queue and one receive queue, and each queue represents one end of the channel. Before an application uses RDMA for data transfer, the receiver posts a work request to the receive queue, and then the sender can post a work request to the send queue. Both the send and receive sides will get a completion event after the data transfer is finished. On the other hand, the memory semantic, or RDMA READ/WRITE, is regarded as one-sided operations. The receiver advertises its available registered memory to the sender, including the key of memory region and the address, so that the sender can directly RDMA WRITE data to the specified memory space at the receiver.

The use of RDMA two-sided operations can be found in Lai et al. (2009b). Their FTP implementation is based on two-sided zero-copy operations in IB networks. However, SEND/RECEIVE operations are originally proposed for delivering control messages. A one-sided semantic, i.e., READ/WRITE is a better choice for high-speed large data transfer due to its ability to decouple data transfer from the host kernel. The authors of Rao et al. (2008); Yu et al. (2008) have shown that even though there are some benefits of using RDMA over LAN and WAN with short latency, there are challenges to achieve good performance in WAN with a long latency due to the low performance issue with RDMA READ operation and the lack of RDMA capability in handling non-contiguous data. Based on prior studies Frey and Alonso (2009); Rao et al. (2008); Yu et al. (2008), our middleware is designed to exploit the full benefit of RDMA by using the RDMA WRITE operation due to its better performance, and lower communication cost for synchronizing senders and receivers.

2.3. OS Kernel Techniques

Another approach to reducing data copy and processing overhead is via software optimization, i.e., techniques supported by modern operating system kernels. The TCP/IP stack involves multiple data copy, which results in high CPU consumption in high speed data transfer environment. One of such examples is the `sendfile` primitive in Linux. It copies data between one file descriptor and another, e.g., between a disk file descriptor and a socket descriptor. Because this copying is done within the kernel, sendfile() is more efficient than the combination of read and write, which would require transferring data to and from the user space.

Another system call, `splice` further extends the functionality of sendfile(). It is a Linux-specific system call that moves data between a file descriptor and a pipe without a round trip to user space. The use of splice() requires the setup of a pipe buffer. A pipe buffer is an in-kernel memory buffer that is transparent to the user space process. A user process can splice the content of a source file, e.g., a socket, into this pipe buffer, then splice the pipe buffer into the destination file descriptor, e.g., a file in the disk systems.

3. Middleware Design

We design a middleware layer between applications and the RDMA network transport layer, with the goal of making this layer a general architecture convenient for developing various applications that can take advantage of RDMA techniques to archive high bandwidth performance and low CPU usage. It provides the necessary data communication and access functions, and maximizes the parallelism of data processing. It considers features such as zero-copy, memory region reuse, multi-stream parallel transfer, and multi-threading. In this section, we first give an overview of the function and components of this RDMA-based middleware layer, and then describe the more specific design of each component. This section also briefly describes the use of Linux kernel primitives (`sendfile` and `splice`) in optimizing non-RDMA conventional TCP-based data transfer, which are integrated in our middleware.

3.1. Overview of the Architecture

The middleware layer, as shown in Figure 2, implements a set of function modules, and provides an abstraction of the computational resources including main memory and network cards. The middleware layer contains two primary sections: one data structure section, which is used to keep track of data structure necessary for data communication and memory access, and one thread pool, which implements all function modules related to data communication, synchronization, and task scheduling. Since registering and deregistering Memory Regions (MRs) increase CPU load Frey and Alonso (2009), our middleware pre-allocates and reuses data structures and threads to avoid the overhead of resource allocation and release during program execution.

The middleware interacts with the host computer adapter (HCA) via an array of queue pairs, which are supported by the OpenFabrics standard. A separate queue, the completion queue (CQ), is also maintained by the same standard. The threads in the middleware layer gain access to the queue pairs and the completion queue via standard programming interface.

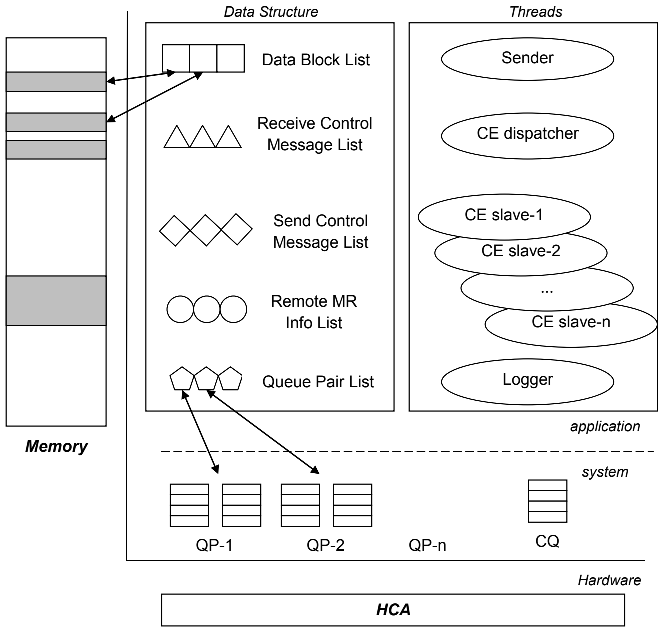


Figure 2: Multi-threaded architecture and data structure of RDMA-based middleware

3.2. Resources Abstraction and Management

The middleware accesses the local and remote memory by maintaining several data block/message lists. This data structure is maintained and updated by the threads, according to certain communication semantics to be described later. This data structure contains:

- A data block is used to contain user payload data, and it is temporarily kept in the data block list before the actual delivery. A user can copy data into the block or extract data from it.
- The send and receive control message lists are used to keep outgoing and incoming control messages, respectively.
- The remote MR information is used to store the memory region information such as key, logical buffer address, and maximum length of acceptable payload at the remote side of the communication.
- A queue pair list is maintained to keep track of the status of the actual queue pairs implemented in the underlying OpenFabrics standard software.

These resource abstractions are managed and maintained by a pool of threads that implement the communication semantics. There is one sender thread, which posts control message tasks or payload tasks into the send queue of the queue pairs. Once a RDMA WRITE operation is completed, a completion event (CE) is generated. A master-slave thread pool handles various types of completion events as follows. The master, i.e., the CE dispatcher thread, on the detection of a completion event, encapsulates this event into a task structure and dispatch it to a CE slave. This CE slave then parses the task to determine the appropriate action, for example, updating the status of a data block. Finally, a logger thread checks the status of ongoing data transfers, for example, whether the transfer is finished or aborted for any reason. The logger also monitors and reports the performance of data transfer.

3.3. Communication Semantics

As described earlier, our middleware layer uses the channel semantic to exchange control messages and the memory semantic to deliver user data payload for low latency and high performance. A dedicated queue pair is used to exchange control messages between two communication parties, and a channel semantic is adopted for this communication. For data payload transfer, we allocate, possibly multiple, additional queue pairs, the number of which is user-configurable. Data packets are exchanged using these queue pairs according to the memory semantic.

There are four categories of control messages to support multi-task, parallel reliable data transfer. We describe each of them as follows.

- **Session identifier negotiation:** Before the data transfer, the middleware at the client side and server side will choose a unique session identifier for each transfer task. For example, when there are multiple files to be transferred simultaneously in one application, each file should be assigned a session identifier for both sides to label all data packets for transferring this particular file.

- **Number of data connections negotiation:** The middleware is designed to support multiple parallel streams for a single data transfer. The client and server will exchange messages to agree on and establish a number of parallel connections.

- **Memory region block request and response:** The memory semantic requires one-sided operations, with which the active side acquires the memory region information of the passive remote side prior to actual data transportation. Before the client issues a RDMA WRITE, for example, it has to compose the sending task information such as the key of the remote memory region, the address of remote memory block, and the maximum allowed length of the remote block. When the server receives this memory region request, it searches for an available memory region for the client and sends back its information.

  For performance consideration, pre-request and batch-mode are used for the control message channel. "Pre-request" means the data source requests free memory regions before an actual data transfer task, and thus the data source could send out the data immediately without the time-consuming memory region information exchange. "Batch-mode' means the data sink feedbacks a list of free memory regions available at the time with a single response, and this reduces the communication overhead due to request control messages.

- **RDMA completion notification:** Since the middleware transfers user payload using one-sided RDMA WRITE, the data sink is not aware of the completion of one data block transfer. Therefore, the data source should issue a notification to the data sink. This notification includes the block address, with which the data sink can locate the data block with user payload.

3.4. Parallel and Pipelined Data Transfer

The middleware layer implements an end-to-end, connection-oriented data transfer. It attempts to maximize the data transfer performance by exploiting parallelism of RDMA operations. We achieve this goal via two ways. First, we allow multiple active data streams and many data blocks simultaneously posted to queue pairs. Second, each single data streams uses execution pipeline, i.e., multiple data blocks of the stream can be posted before any of them is acknowledged.

In particular, the implementation of multiple active streams facilitates parallel data transfer between a single pair of data source and sink. For instance, a single file can be partitioned (or striped), and each partition can be delivered using a single data channel, i.e., a stream. This implementation will potentially increase the data transfer rate for data-intensive applications. With this multi-stream transfer, many data blocks could be transferred through queue pairs simultaneously. So it is possible that some of them arrive at the data sink out-of-order. There is one minor problem related to this multi-stream data transfer. For example, magnetic disk write performance would deteriorate during random access. To solve this problem, our middleware contains a plug-in module which gathers out-of-order blocks temporarily, until they can be delivered to the applications (for example, the disk writer here) in a sequential order.

To guarantee the orderly execution of multiple parallel RDMA operations, we need to carefully maintain the data block list. For this purpose, we associate each block with a status field, and this field is updated according to different operations as shown in Figure 3.

To illustrate this diagram, we consider the following example of disk-to-disk file copy. Initially, blocks at both the data source and sink are in the "free" state.

At the data source, one consumer of free blocks, the disk reader, takes one "free" data block from the data block list. The disk reader loads payload into the data block, then marks this block as in the "sending" state. The

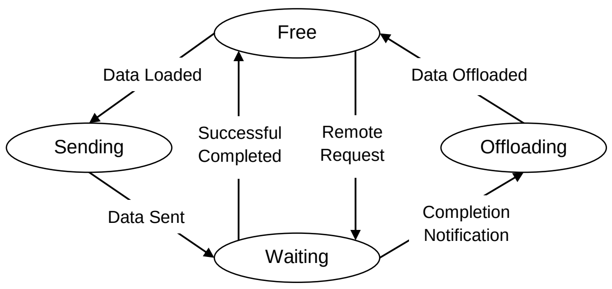


Figure 3: The data block state transition diagram

sender thread sends out this block using a RDMA WRITE operation. As we described, the RDMA operation is asynchronous. After the RDMA WRITE task is posted to the sending queue, the state of this block becomes “waiting”. When the completion event (CE) corresponding to this task is generated, the CE dispatcher captures this event, composes this as a task, and forwards it to a CE slave thread. This CE slave thread checks this block and marks it as “free” again if the send task is finished successfully.

At the data sink, once the application receives a free block request, it searches for one data block in “free” state. It finds out the attributes of this block, such as the key of the memory region, its virtual address, and its maximum payload length. Then it packs this information into a free block response and sends it to the data source. This block is marked as “waiting” state. At this point, the data sink is waiting for the remote side to fill this block using RDMA one-sided operation. After the data sink receives the “finish notification” event, it marks this block as “offloading” state. The content of the data block is passed to the data consumer, i.e., a disk writer. Once the writer offloads all the content in the data block, it marks this block as “free” again.

3.5. Integration of Software Kernel-bypass Techniques

Our middleware is designed to also support the conventional TCP/IP-based data transfer, for applications without RDMA network configurations. This is also an important feature of our middleware, as it provides good compatibility. However, the conventional TCP/IP protocol stack may not be efficient, especially in applications that require ultra-high speed data transfer. For this reason, we integrate some Linux based kernel optimization techniques to reduce the overhead due to excessive data copy.

Various versions of Linux systems implement sendfile() and splice(), two system calls that provide the capability to reduce data copy between file descriptors. The sendfile() copies data between one file descriptor and another. Because this copy is done within the kernel, sendfile() is more efficient than the combination of read() and write(), which would require transferring data to and from user space. The splice() moves data between two file descriptors without copying between kernel space and user space. Once the application calls splice to move data from a file descriptor which reference a disk file to a pipe, the kernel loads data into the pages associated with the pipe directly without copying data into the user space.

In our middleware design, we use these two system calls to reduce the data copy between a disk file descriptor and a socket descriptor. We use the sendfile() for the sender, who wishes to deliver data from a disk file to a socket. The use of sendfile() is rather straightforward. In addition to that, we consider the TCP_CORK option in the sendfile() method. TCP_CORK is a parameter for throughput optimization. If set, kernel wont send out partial frames before a time ceiling (200 millisecond). The reason we use this option is the follows. The sendfile() takes the higher CPU consumption without TCP_CORK than with it. Kernel generates many partial frames in the sender side, and this results in more system calls, and the CPU consumption is high. In our middleware design, we also optimize the data copy at the receiver side of a data transfer. The splice() system is used for this purpose. For this system call, we need to explicitly create a pipe between a socket descriptor and a disk file descriptor.

4. Implementation of RFTP

The design of our middleware layer facilitates the development of various distributed applications that rely on RDMA techniques for optimal data transfer performance. In this section, we show one example, the design and implementation of our RDMA-based FTP service, RFTP. The section starts with the overall layout of the RFTP application supported by the middleware layer. We then discuss our RDMA extensions to the standard FTP protocol, and the workflow in RFTP.

4.1. RFTP Modules

The RFTP has a layered architecture as shown in Figure 4. This RFTP application is extended from the traditional FTP protocol. We support two

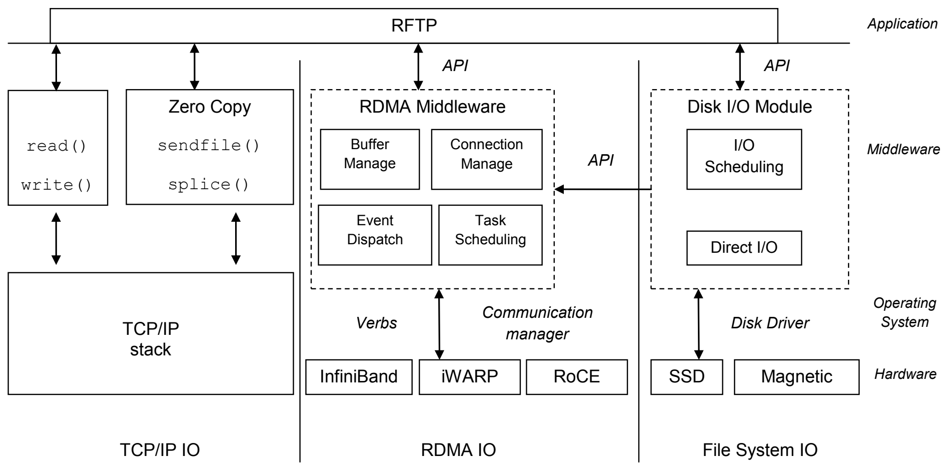


Figure 4: Modules in RFTP

modes: one is solely based on the conventional TCP/IP stack with optimized socket operations, i.e., sendfile() and splice() as we described, and the other is based on our RDMA middleware layer as our extension to the standard FTP protocol. The middleware layer itself consists of several modules, such as buffer management and connection management.

This RDMA middleware is in turn supported by low layer protocols. It uses IB verbs through the InfiniBand, RoCE, and iWARP provided by InfiniBand Verbs library (libibverbs) and RDMA communication manager (librdmacm). Thus, the middleware layer hides all the specifics related to the hardware to provide the desirable transparency, while the applications, for example, FTP, do not have to be aware of those specifics.

During the development of RFTP, we also notice that disk access is critical to the performance of the applications. Thus in our RFTP implementation, we design an additional disk I/O module based on direct I/O operations. This module includes an I/O scheduler, which is responsible for supporting disk multiple readers/writers. The direct I/O access is particularly efficient for solid-state disks (SSD).

4.2. RDMA extension to standard FTP protocol

The standard FTP protocol implements a set of commands that are exchanged between the client and server entities. A control channel is used for transferring these commands, while actually file content is transmitted over another data channel.

Table 1: RDMA-extension to standard FTP protocol

Command	Function and Procedure
RADR	RDMA address information exchange
RSTR	1) Establish RDMA data transfer channel; 2) Send data from client to server using RDMA operations, and store file at server side; 3) Tear down RDMA data transfer channel.
RRTR	1) Establish RDMA data transfer channel; 2) Get data from server to client using RDMA operations, and store the file at client side; 3) Tear down the RDMA data transfer channel

To support data transfer over our RDMA middleware, we extend the set of commands in FTP protocol. Table 1 lists the RDMA-extension FTP commands, RADR, RSTR, and RRTR, to support RDMA-enabled data transfer. They correspond to the PORT, STOR, and RETR commands in RFC 959 - File Transfer Protocol. These extensions explicitly request the remote side to adopt RDMA-specific steps for data transfer and therefore negotiate particular RDMA capabilities and ensure the compatibility between sender and receiver. If, however, when the conventional commands of FTP are used, we resort to the TCP/IP based mode with kernel bypass techniques.

We also provide user commands, *rget* and *rput*, which are extensions to the original FTP commands. We provide a command line parameter to allow users to choose between two modes, i.e., the traditional FTP service that is built on top of the socket interface and the RFTP service that is built on top of our middleware. The advantages of this extension instead of rewriting the entire FTP service include, 1) it is convenient to compare different data transfer mechanisms, 2) it is relatively easier to make FTP service available on these RDMA architectures, and 3) user can easily choose to use the one that is fit with their data transfer environment.

4.3. Workflow in RFTP

Figure 5 describes the workflow of RFTP, which is similar to that of standard FTP service. We add three RDMA-extension commands for the control message carried in the FTP communication channel. When these extended commands are exchanged, a RDMA control message channel and multiple RDMA data channels are initiated for RDMA data movement between the

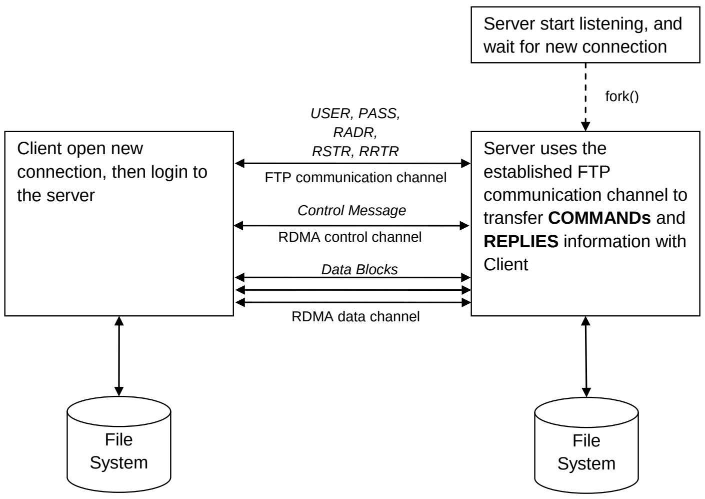

Server start listening, and wait for new connection
fork()
USER, PASS, RADR, RSTR, RRTR
FTP communication channel
Client open new connection, then login to the server
Server uses the established FTP communication channel to transfer **COMMANDs** and **REPLIES** information with Client
Control Message
RDMA control channel
Data Blocks
RDMA data channel
File System
File System

Figure 5: Workflow chart of RFTP

sender and receiver. For each of such data channels, both the sender and receiver will allocate and preregister data blocks. These data blocks are initially organized in a data block list.

We use an example of *rput* in Figure 6 to illustrate the implementation of user-level commands. The figure shows the interaction between the client and server when uploading a file to the server. Note we take advantage of one-sided RDMA operations for fast data transfer and two-sided RDMA operations for related control message exchange. The client first sends a request to the server to get a session identifier for the file to be transferred. The server, upon receiving the request, assigns a global session identifier for this file and sends it back to the client. All the data packages contain this session identifier and sequence numbers. The memory region addresses information and RDMA keys are sent back to the client in the reply message by the server side per each remote memory region request. Afterward, the client starts a set of RDMA WRITE operations to deliver data to the registered memory space. The data delivery, once completed, is followed by a control message to signal its completion for each RDMA WRITE operation. The server, upon receiving the completion control message, then moves the data to the file system. This memory region will be available after the disk write is completed, and the memory region information is sent back to the client for the next round of data transfer. This client/server interaction will repeat

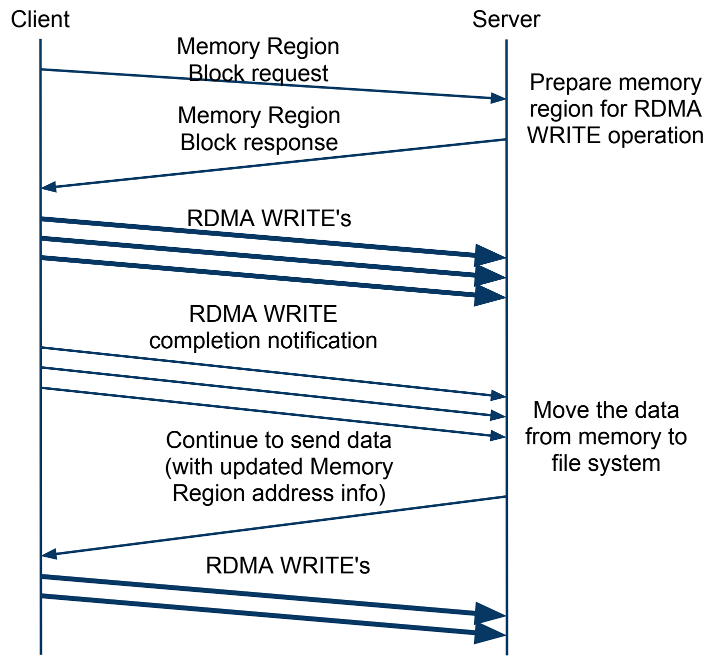

Client
Server
Memory Region Block request
Prepare memory region for RDMA WRITE operation
Memory Region Block response
RDMA WRITE's
RDMA WRITE completion notification
Move the data from memory to file system
Continue to send data (with updated Memory Region address info)
RDMA WRITE's

Figure 6: Implementation of *rput* with RDMA WRITE

until the entire file is delivered.

5. Experimental Results

To validate our middleware and RFTP application built on top of this middleware, we conduct a comprehensive experimental study on real test platforms. We first describe the test setup with different network configurations, including RDMA architectures and IP networks. Then we compare the performance of RFTP with Netkit FTP Holland (1998), a standard FTP implementation over socket, and GridFTP, a high performance data transfer protocol widely used in data-intensive science applications. We use our results and a demo at the 2011 Supercomputing conference, based on the IP-based 100Gbps Advanced Networking Initiative (ANI) testbed, to conclude this section.

In our experiments, we use long data sessions, since we target data-intensive applications and bulk data transfer scenarios. Sophisticated statistical performance analysis based on many experimental runs are valuable, but very difficult due to time constraint. Fortunately, as we will show, for large data transfers, the performance are relatively stable. We should acknowledge that, for short data sessions which although are not our focus, it is more important to conduct comprehensive statistical analysis.

5.1. RDMA Networks Setup

We consider both memory-to-memory and memory-to-disk data transfer between local and remote hosts. For memory-to-memory data transfer, we generate and transfer data between client memory and server memory. In this configuration, our focus is to evaluate the network bandwidth performance and protocol offload efficacy, but not the file system performance. For modern data center applications, as suggested in Lai et al. (2009b), it is a reasonable simplification to avoid the disk I/O bottleneck. Many of these applications usually employ solid state disk arrays or distributed file systems, such as the Lustre file systems Oracle (2011), to achieve high speed I/O performance. In our experiments, the bandwidth performance only counts the pure user payload without all the control messages. For the memory-to-disk data transfer experiments, we set up a disk system at the receiving server side using Fusion-io's solid state disk arrays.

For host connectivity, we consider a variety of networks. For example, LAN connectivity is frequently used in data center applications, while WAN connectivity is required in the ANI testbed where our RFTP tool will be eventually deployed for use. The details of our three configurations are as follows.

5.1.1. High-bandwidth low-latency LANs

In order to provide an application-level performance test over different RDMA architectures, we setup two local-area test platforms. The first test platform is described in Figure 7. We have the client and server hosts set in the Brookhaven National Laboratory, and the propagation delay between them is less than 0.1ms. The detailed configurations for the nodes and switches are listed in Table 2. Each host is equipped with a 40Gbps IB HCA and a 10Gbps iWARP HCA. The IB Mellanox RNIC we use can support both IB and RoCE.

The second test platform contains a 10Gbps link, between two hosts located in University of Michigan, Ann Arbor. We use this platform to evaluate and compare the performance of our RFTP and GridFTP. In order to maximize the performance of GridFTP over 10Gbps links, we tried several TCP

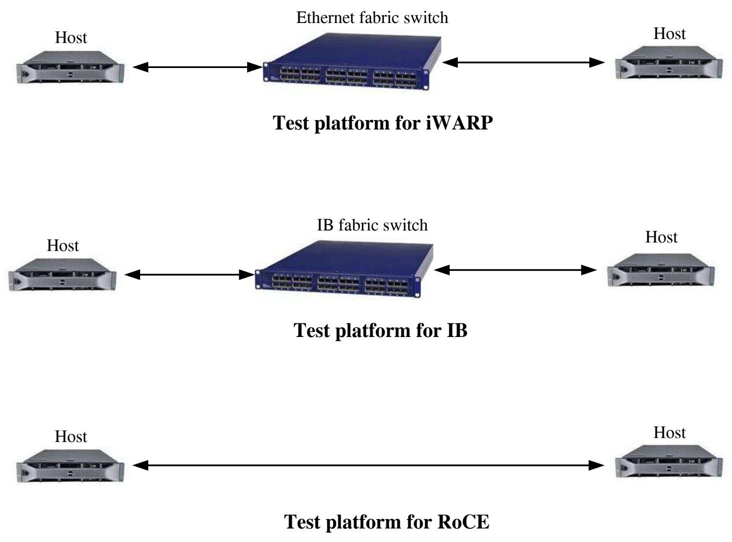


Figure 7: Test platform with different RDMA implementations in LAN

Table 2: LAN cluster configuration

Hardware	24 Intel(R) Xeon(R) CPU X5660 @ 2.80GHz cores
	64GB memory
	12288KB L3 cache
	iWARP HCA: NetEffect NE020 10Gb Adapter
	IB HCA: Mellanox MT26428 ConnectX VPI PCIe IB QDR
Software	RHEL 5 with kernel-2.6.18
	OFED Version 1.5.2
	Netkit FTP version 0.17
Network	Mellanox MTS 3600 InfiniBand switch
	Juniper EX2500 Ethernet fabric switch

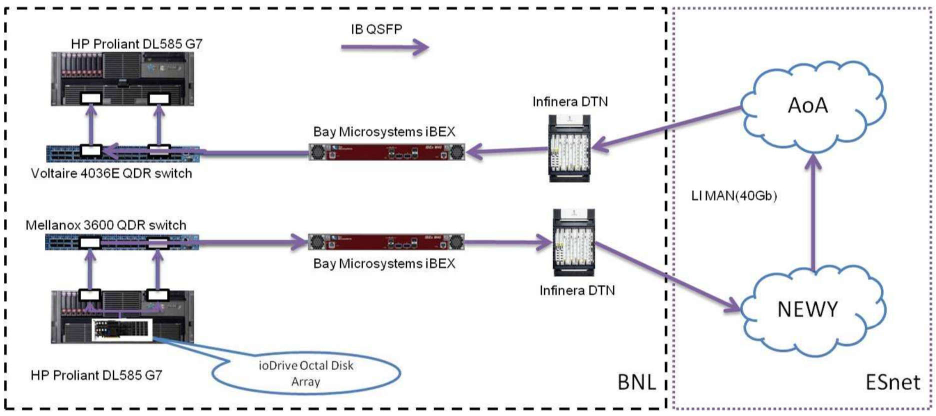


Figure 8: Host connectivity with long haul-link, for MAN performance test

parameter tunings, including setting the MTU at 9000, i.e., jumbo frames, according to ESnet (2012b). The extended block mode (MODE E) of GridFTP Globus (2012) is enabled throughout the test for high performance.

5.1.2. High-bandwidth medium-latency MAN

We set up a communication path in the New York metropolitan area. We have two hosts located inside the Brookhaven National Laboratory, Long Island. These two hosts are connected by a long distance SONET link that goes through New York city, as shown in Figure 8. This infrastructure is also part of the Energy Science Network (ESnet) ESnet (2012a). The capacity of this link is 40Gbps in each direction, with a minimum round trip time (RTT) of 3.6ms.

5.1.3. High-bandwidth high-latency WAN

For the long-haul link for WAN test, we utilize the path between University of Michigan, Ann Arbor, and Brookhaven National Lab, which traverses through the ESnet and UltraLight networks UltraLight (2012), as shown in Figure 9. The capacity of this link is 10Gbps, with a minimum RTT of 31ms.

5.2. Experimental Results over LAN

In this set of experiments, we use memory-to-memory data transfer for the baseline results. We compare the performance between RFTP and Netkit FTP, and between RFTP and GridFTP.

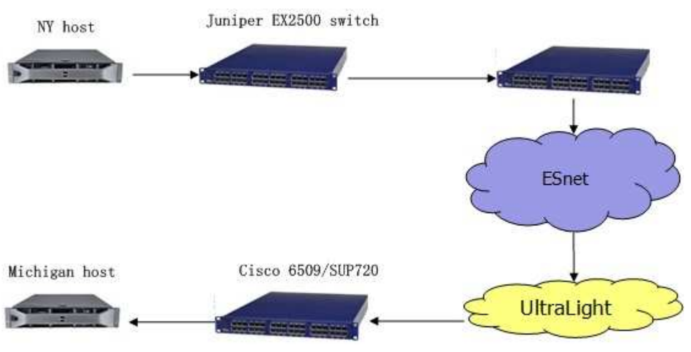


Figure 9: Host connectivity with long haul-link, for WAN performance test

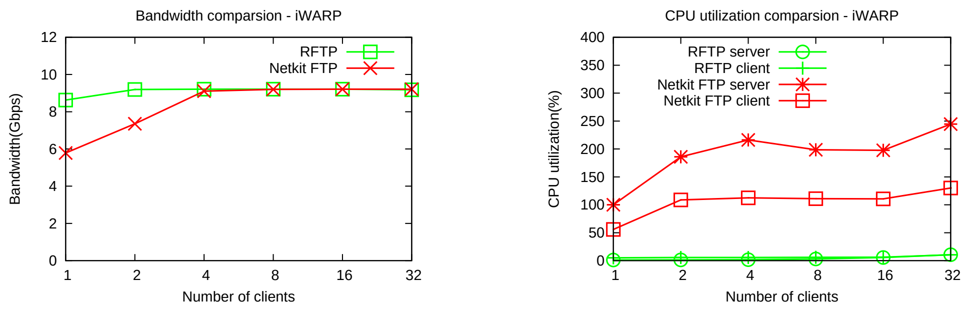


Figure 10: Bandwidth and CPU utilization comparison between RFTP and Netkit FTP over iWARP in LAN

5.2.1. Bandwidth and CPU usage of RFTP and Netkit FTP

We consider the aggregate bandwidth and CPU utilization as the main performance metric, and the performance numbers are obtained as follows. For both Netkit FTP and RFTP, multiple clients are initiated and connected to the server concurrently, and then each client process transfers 100GB of data to the server memory using the *put/rput* command. The server always listens to potential incoming client connection requests, and on each connection request, forks a child process to handle data transfer. The aggregate bandwidth is obtained by monitoring the entire transfer period of all connections, and then we take the average bandwidth during the time period. We use “nmon” tool Griffiths (2006) to obtain the CPU utilization of the application. Note, we have 24 cores in our hosts, and therefore the total CPU utilization can be up to $24 \times 100\%$.

Based on performance metrics described above, we try to evaluate the performance improvement of RFTP over Netkit FTP. We define the following

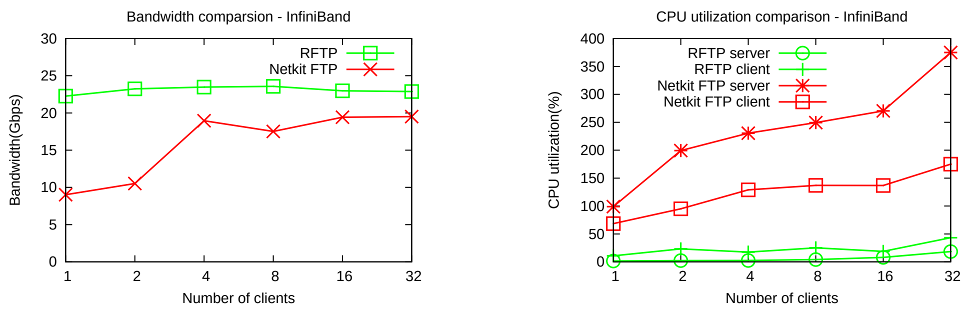


Figure 11: Bandwidth and CPU utilization comparison between RFTP and Netkit FTP over InfiniBand in LAN

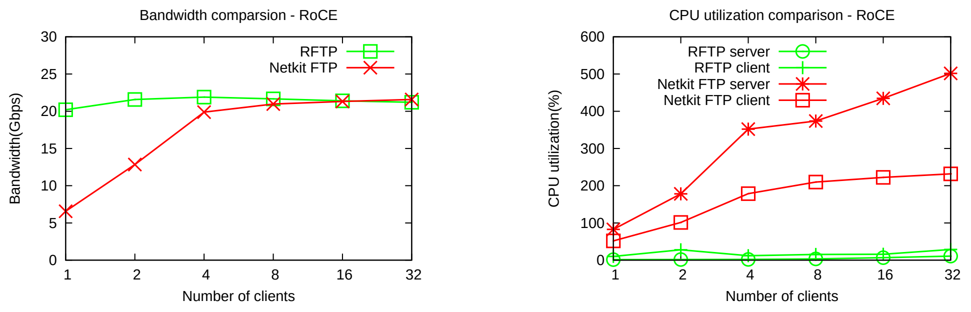


Figure 12: Bandwidth and CPU utilization comparison between RFTP and Netkit FTP over RoCE in LAN

Table 3: Bandwidth and CPU utilization ratio with single client

	iWARP	InfiniBand	RoCE
α	1.492	2.468	3.074
β_{server}	1.168%	1.435%	1.774%
β_{client}	8.906%	15.802%	19.251%

performance ratios:

$$Bandwidth\ ratio\ of\ single\ client\ \alpha = \frac{Bandwidth\ of\ RFTP\ with\ single\ client}{Bandwidth\ of\ Netkit\,FTP\ with\ single\ client} \tag{1}$$

$$CPU\ utilization\ ratio\ of\ single\ host\ \beta = \frac{CPU\ utilization\ of\ RFTP\ with\ single\ host}{CPU\ utilization\ of\ Netkit\ FTP\ with\ single\ host} \tag{2}$$

In addition, when calculating CPU utilization ratio, we consider client and server hosts separately. For instance, β_{client} is calculated by comparing the CPU utilization of a single client running with RFTP, and a single client running with Netkit FTP.

Figures 10–12 show the aggregate bandwidth and CPU utilization performance of RFTP and Netkit FTP over iWARP, InfiniBand, and RoCE in LAN, respectively, with different numbers of concurrent clients. We have the following observations:

- RFTP saturates the bare metal bandwidth with only two concurrent clients, compared with 4-8 clients needed by Netkit FTP. To avoid the raw bandwidth limit of the physical channel, we calculate the ratios of aggregate bandwidth and CPU utilization performance when there is only one client in Table 3. We can see that the bandwidth increases by 49.2% on iWARP, 146.8% on InfiniBand, and 207.4% on RoCE, respectively. With RoCE, the advantages of RFTP is more obvious than with IB, because the RoCE test case uses back-to-back connection while the IB HCAs are connected via an IB switch.

- All three RDMA architectures greatly improve the bandwidth performance with extremely low CPU consumption. All the protocol-intensive operations are offloaded to network adapters. Our results show the advantage of the POE technique clearly.

- By taking advantage of low-latency RDMA operations and an efficient middleware design, RFTP significantly improves the bandwidth performance. On the other hand, TCP/IP stack is known as high overhead protocol for high-speed data transfer applications. At the sender side, the application data must be copied into the socket buffer in the kernel space before sending, and vice versa at the receiver side. The unnecessary intermediate buffering and associated context switching dramatically increase CPU processing overhead, and slow down the data transfer process. The figures above clearly shows that the high CPU load is due to complicated protocol processing. In addition, due to the overhead from flow and congestion control of TCP, Netkit FTP cannot fill the available bandwidth when there is only a couple of clients.

- The bandwidth and CPU load performance do not strictly increase with the number of clients. The reason is that, during our experiments, there were other active processes in the hosts, competing against our test processes. Those processes introduced noise to our CPU load performance. Nevertheless, we notice that the CPU consumption of Netkit FTP increases faster than that of RFTP with the increase of the number of clients. This underscores again the advantage of POE in lowering CPU load.

- As shown in Table III, iWARP uses the least CPU resources since it runs over a 10Gbps link while both IB and RoCE run over 40Gbps network. The raw bandwidth of iWARP is much lower than that of IB and RoCE. This fact leads to the relatively lower operation cost of the middleware layer over iWARP.

- Since we implement one-sided RDMA in the middleware, much of the overhead and complexity is left to the client side in our test. The client uses RDMA WRITE for the *rput* (RDMA upload) operation, and therefore, the CPU usage of the server side is lower than that of the client side.

Table 4: Statistical analysis of performance

		min	25th	mean	median	75th	max
RFTP over iWARP	Bandwidth	9.430	9.534	9.531	9.534	9.535	9.535
	Client CPU	8.953	10.159	11.748	11.112	12.759	18.341
	Server CPU	4.671	6.059	7.845	7.718	9.506	11.812
Netkit FTP over iWARP	Bandwidth	8.925	8.948	8.961	8.957	8.977	9.012
	Client CPU	27.100	27.306	27.515	27.406	27.661	28.350
	Server CPU	99.944	99.983	99.991	99.994	100.000	100.006
RFTP over InfiniBand	Bandwidth	24.500	24.538	24.536	24.5398	24.540	24.541
	Client CPU	95.983	99.800	106.199	102.967	113.983	121.233
	Server CPU	37.333	37.833	38.267	38.317	38.600	40
Netkit FTP over InfiniBand	Bandwidth	13.178	13.244	13.282	13.280	13.340	13.406
	Client CPU	99.8	99.867	99.878	99.875	99.891	99.909
	Server CPU	33.77	34.09	34.550	34.32	35	36.42

One question regarding our test results is whether they are statistically representative of the applications under evaluation. For that purpose, we conducted a large set of independent tests as before. Since the original testbed was no longer available, and we had to run the tests on a new testbed with more powerful PCI Gen 2 hosts, and both 10Gbps iWARP and 40Gbps InfiniBand links. We performed the test 35 times to get the characteristic performance numbers, such as the mean values, the median values, the 25th and 75th percentile values, and the minimum and maximum values. Each time we let one process transfer 100GB data from the local /dev/zero to the remote /dev/null at another host. We show the statistical results in Table 4. We can see that the performance numbers, including both bandwidth and client/server CPU times, are all within tight ranges. Thus, we conclude that the performance advantages of RFTP is consistent.

5.2.2. Bandwidth and CPU usage of RFTP and GridFTP

GridFTP is a popular tool for data transfer in high-performance computing environment. As RFTP, GridFTP allows the use of multiple parallel connections to maximize bandwidth utilization. In this experiment, we compare the performance of GridFTP and RFTP, in terms of both aggregate bandwidth and CPU consumption.

Figure 13 shows the performance comparison between GridFTP and RFTP in the LAN environment with 10Gbps iWARP connection. We run RFTP

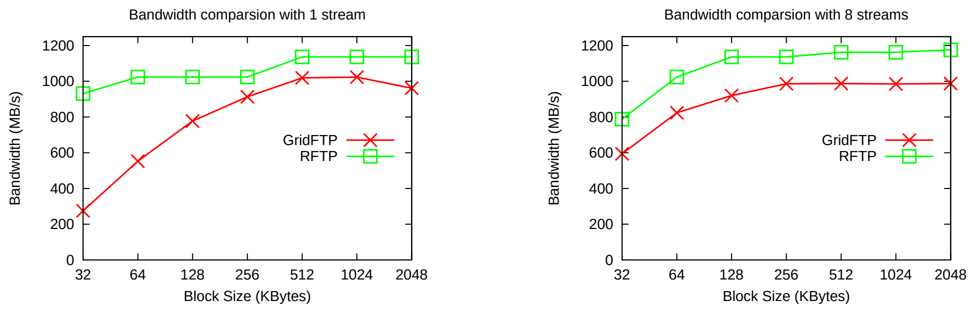


Figure 13: Bandwidth comparison between GridFTP and RFTP over iWARP in LAN

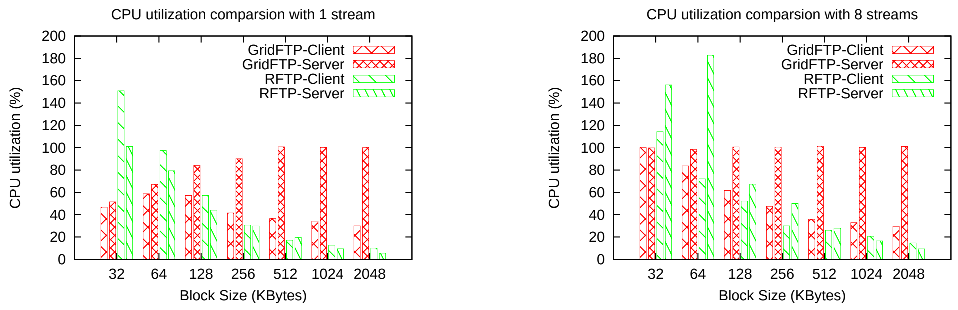


Figure 14: CPU utilization comparison between GridFTP and RFTP over iWARP in LAN

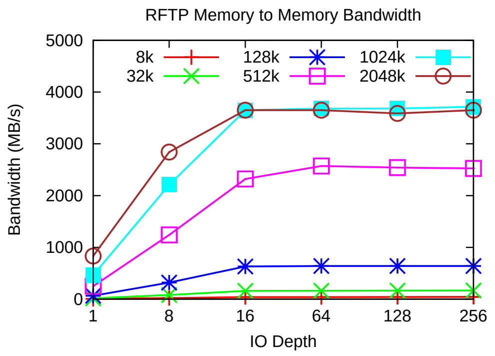


Figure 15: RFTP memory to memory bandwidth in MAN

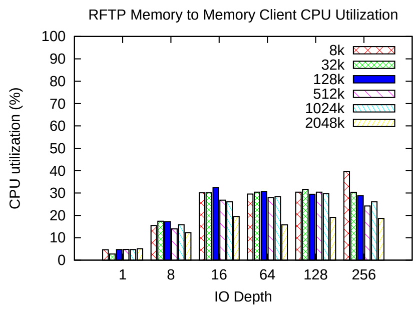
with one stream and eight streams. We also test GridFTP with a single TCP connection and eight parallel connections. In terms of bandwidth performance, RFTP consistently outperforms GridFTP in this setting. Notice that with RFTP, the bandwidth is almost fully utilized when block size is large enough, for example, 128K bytes. For CPU utilization, Figure 14 shows, with a small block size, both RFTP client and server run at higher CPU utilization, due to the higher overhead for handling more control messages and completion events. Because our middleware has a multi-thread design, RFTP takes advantage of multi-core CPU resources efficiently. When the block size is large enough, we observe that the CPU consumption of RFTP drops drastically, as much of the protocol processing and data copy has been offloaded. On the other hand, GridFTP, in particular its server, requires almost a constant CPU consumption even when block size is large.

We note that performance evaluation of GridFTP alone with InfiniBand was done in another study Subramoni et al. (2010). Our work is different in that we consider another RDMA architecture, and we compare the performance of GridFTP to our application.

5.3. Experiment Results over MAN

In this set of experiments, we use both memory-to-memory and memory-to-disk data transfer to show the effectiveness of our design.

The first experiment evaluates the network performance of our middleware layer and RFTP without bandwidth limitation from the disks. At the data source, the disk I/O module loads data from the special file /dev/zero, and simply copies data into /dev/null at the data sink. The application's



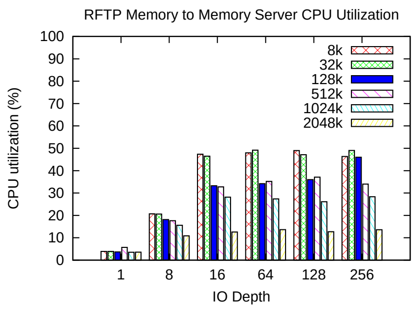


Figure 16: CPU utilization comparison of RFTP with different block size and I/O depth in MAN

throughput performance is calculated by the logging thread in the middleware layer.

We configure with only one data stream in this experiment to check the impact of I/O depth and data block size of each trunk. Figure 15 shows the bandwidth of the RFTP payload. The throughput of the middleware increases as the size of each data block increases. However, the throughput hits the hardware limit offered by the network at certain block size, and further increasing the size does not improve the bandwidth performance. On our testbed, we have found that 2048 kilobytes block size results in the same throughput as 1024 KB's when the RDMA I/O queue depth is greater than 16.

Our middleware design supports multiple packets in flight in each transfer pipeline. It means the RFTP application can post many RDMA tasks into the send queue simultaneously. With the same block size, the larger I/O depth usually means a better performance.

As shown in Figure 16, when the I/O depth is low and the block size is small, .e.g., less than 1024 kilobytes, the CPU utilization remains mostly a constant. This is because the network bandwidth is under-utilized, as shown in the bandwidth figure. However, when the I/O depth is larger than 16 and the block size is greater than 1024 kilobytes, the bandwidth reaches its maximum. At this point, the use of large block size leads to lower CPU consumption. For example, the CPU usage at 2048-kilobytes block size is half that at 1024-kilobytes block size.

In our second experiment, data will be sent from /dev/zero at the data source to a Fusion-io disk set, which contains eight physical solid state disks at the data sink. The experiment launches two concurrent processes, each

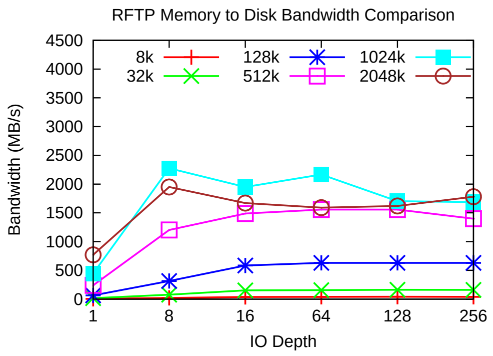


Figure 17: RFTP Memory to Disk Bandwidth comparison in MAN

responsible for delivering four files into four independent disks. As shown in Figure 17, the performance of RFTP in this case is similar to the performance in the previous memory-to-memory experiment when the block size is smaller than 128 kilobytes. When the block size is small, the disks are not the bottleneck. When the block size is larger, the disk bandwidth becomes the bottleneck. We find the RFTP's performance is close to the peak disk rate that is measured with local disk performance tools on the installed Fusion-io disks.

5.4. Experimental Results over WAN

Finally, we run RFTP and Netkit FTP over the long-haul WAN link. iWARP is utilized in this test due to its capability to run RDMA over IP networks. Due to the space limitation, we do not include the full experimental results, but only summarize our findings here. We observed that CPU utilization of RFTP still stay at a relatively low level. However, the bandwidth of RFTP does not show much improvement over Netkit FTP, particularly when the number of concurrent streams is small. Recent studies Cohen et al. (2009); Lai et al. (2009a); Rashti et al. (2009); Subramoni et al. (2009) also reported the shortcomings of iWARP.

To elaborate this, we note the implementation of the TCP/IP stack in the iWARP RNIC is limited and has much fewer parameters to tune than one in the operating systems. For long-haul high-speed network, the TCP buffer size must be set large enough to fully utilize the link capacity regardless whether the data transfer uses a TCP in OS or in POE. RNIC must be configured to allow a very large TCP sliding window to accommodate the long and fat links. Once off-loaded hardware is implemented, it is very difficult to make customized changes for various environments. Therefore, we argue that this is a major disadvantage of POE techniques. We also conjecture that, despite the POE techniques, the iWARP RNIC still suffers from high overhead of TCP processing and the complicated layered structure of iWARP, and thus achieves low data transfer efficiency over high speed WAN links.

5.5. Experiment Results and Demo on ANI testbed

We have also tested the performance our RFTP software on the ANI test. To illustrate these experiments, we should first briefly describe the testbed. The Advanced Networking Initiative (ANI) testbed is part of a larger Advanced Networking Initiative (ANI) program to build a nationwide 100Gbps prototype network which linking several DOE supercomputer sites and the international network exchange, MANLAN, in the New York City. Our middleware system and the RFTP are developed to support data-intensive computing over this network, for example, climate data transfer and various applications. At the time of our experiments and during our demo at the 2011 Supercomputing conference, the RDMA links were not ready for use. Rather, the testbed team built an IP network with an aggregate bandwidth of 100Gbps among three main sites, National Energy Research Scientific Computing Center (NERSC), Argonne National Laboratory (ANL), and Oak Ridge National Laboratory (ORNL).

In our test and demo, we run parallel data transfer sessions, from NERSC to ANL and from NERSC to ORNL. We utilized 15 hosts at NERSC, 14 hosts at ANL, and 10 hosts at ORNL. From NERSC to ANL, we started 14 parallel data transfer session, between 14 pairs of hosts. They are all for disk-to-memory data transfer. The NERSC hosts have General Parallel File System (GPFS), and each host continuously sent a set of files of variable sizes. The ANL hosts did not provide disk systems, and therefore the received data will be discarded. From NERSC to ORNL, we started 10 parallel data transfer session, between 10 pairs of hosts. They are all for disk-to-disk data transfer. The receiving hosts had ext4-ssd disk systems. The GPFS disk systems were able to offer more than 80Gbps aggregate bandwidth.

Our RFTP test was done in the TCP/IP-based mode, but with the software kernel bypass optimization. For all the data transfer sessions, we tuned the parameters such as the number of streams in each session and buffer size. During our test, the ANI testbed was not used by other activities. To ensure

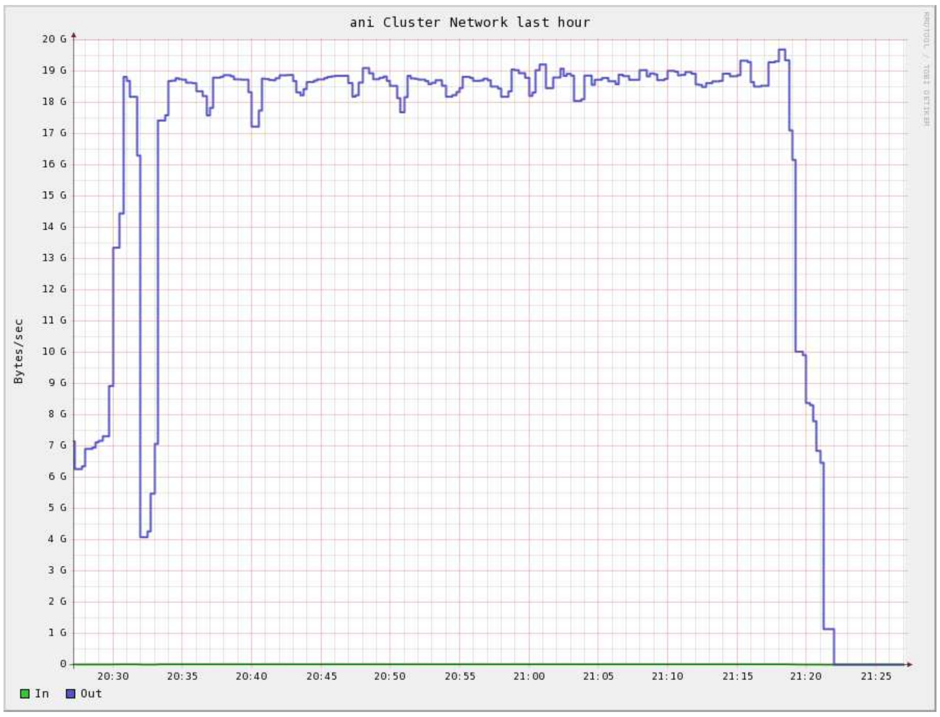


Figure 18: Throughput achieved by RFTP in the ANI testbed 100Gbps network.

this, we made reservations for the time slots of the entire testbed, including the links and the host resources.

Figure 18 shows the results of our test during our demo on November 17, 2011. During this 30-minute time slot, we monitored the network activities at the gateway using the `Ganglia` tool. This tool counts the traffic volume in and out of the gateway in NERSC, and displays the bandwidth throughput. The figure shows that our RFTP software was able to achieve about 80Gbps aggregate throughput. Note that the disk systems were able to deliver only 80Gbps bandwidth. Thus, RFTP already achieved the maximum possible throughput.

6. Conclusions

RDMA is known as a promising high-performance POE technique that supports zero-copy and kernel bypass mechanisms. However, it is difficult to program with RDMA. Therefore, appropriate middleware support are important for the development of efficient applications and underlying hardware transparency. In this paper, we have designed and implemented a RDMA-based middleware layer that provides resource abstraction and management, task scheduling, and parallel data transfer. Our middleware utilizes the most favorable interfaces of the OFED verbs, buffer reuse and task synchronization mechanisms that are tightly coupled with RDMA architecture. We implemented our RDMA-based FTP application based on this middleware layer. To accommodate network environments without hardware RDMA supports, our middleware supports an optimized TCP based data transfer mode, and in this mode, Linux kernel zero-copy techniques are used for performance improvement.

To demonstrate the efficiency of our middleware design, we developed a FTP application based on this middleware layer. In order to obtain the practical test data under different scenarios, we setup a platform with three different RDMA technologies, and we also tested the performance of our system over long-haul MAN links. In particular, we also demonstrated the performance of RFTP on the ANI testbed, which has a 100Gbps IP network. Our application protocol design also shows its efficiency in long-haul high latency RoCE links Ren et al. (2012b). The experiments show that our middleware achieves remarkable bandwidth performance with marginal CPU resources, and it can be a common substrate to accelerate various data transfer applications.

Acknowledgment

The authors are grateful to the facility donation of Mellanox Technologies, Inc. and Fusion-io, Inc. The authors have benefited from the numerous technical discussions with Todd Wilde from Mellanox, David McMillen from System Fabric Works, Inc., and David Strohmeyer from Intel. This work is supported by United States Department of Energy, Grant No. DE-SC0003361.

This research used resources of the ESnet Advanced Network Initiative (ANI) Testbed, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231, funded through the The American Recovery and Reinvestment Act of 2009.

References

Bierbaum, N., 2002. MPI and embedded TCP/IP Gigabit Ethernet cluster computing, in: Proceedings of 27th Annual IEEE Conference on Local

Computer Networks, Tampa, Florida, USA. pp. 733–734.

Cohen, D., Talpey, T., Kanevsky, A., Cummings, U., Krause, M., Recio, R., Crupnicoff, D., Dickman, L., Grun, P., 2009. Remote Direct Memory Access over the Converged Enhanced Ethernet fabric: Evaluating the options, in: 2009 17th IEEE Symposium on High Performance Interconnects (HOTI), pp. 123–130.

Danalis, A., Brown, A., Pollock, L., Swany, M., 2008a. Introducing gravel: An MPI companion library, in: Proceedings of IEEE International Symposium of Parallel and Distributed Processing (IPDPS), Miami, Florida USA.

Danalis, A., Brown, A., Pollock, L., Swany, M., Cavazos, J., 2008b. Gravel: A communication library to fast path MPI, in: Euro PVM/MPI 2008.

DAT Collaborative, 2002. uDAPL: User Direct Access Programming Library. http://www.datcollaborative.org/udapl_doc_062102.pdf.

ESnet, 2012a. Energy Sciences Network: http://www.es.net/.

ESnet, 2012b. Linux TCP Tuning: http://fasterdata.es.net/fasterdata/host-tuning/linux/.

Feng, W., Balaji, P., Baron, C., Bhuyan, L.N., Panda, D.K., 2005. Performance characterization of a 10-Gigabit ethernet TOE, in: Proceedings of 13th Symposium on High Performance Interconnects (HOTI).

Frey, P.W., Alonso, G., 2009. Minimizing the hidden cost of RDMA, in: Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS).

Globus, 2012. GT 4.0 GridFTP Glossary: http://www.globus.org/toolkit/docs/4.0/data/gridftp/gridftp_glossary.html.

Griffiths, N., 2006. nmon performance: A free tool to analyze AIX and Linux performance.

Holland, D.A., 1998. Linux NetKit: http://www.hcs.harvard.edu/ dholland/computers/old-netkit.html.

IBTA, 2006. InfiniBand Architecture Specification. Release 1.2.1 .

IBTA, 2010. Infiniband Trade Association. http://www.infinibandta.org/.

Jang, H., Chung, S.H., Yoo, D.H., 2008. Implementation of an efficient RDMA mechanism tightly coupled with a TCP/IP offload engine, in: Proceedings of International Symposium on Industrial Embedded Systems (SIES).

Lai, P., Balaji, P., Thakur, R., Panda, D.K., 2009a. ProOnE: A general-purpose protocol onload engine for multi- and many-core architectures. Computer Science - Research and Development 23, 133–142.

Lai, P., Subramoni, H., Narravula, S., Mamidala, A., Panda, D.K., 2009b. Designing efficient FTP mechanisms for high performance data-transfer over InfiniBand, in: Proceedings of International Conference on Parallel Processing (ICPP).

Linux, 2012. sendfile manpage: http://www.kernel.org/doc/man-pages/online/pages/man2/sendfile.2.html.

OpenFabrics Alliance, 2012. OpenFabrics Alliance: http://www.openfabrics.org/.

Oracle, 2011. Lustre file system: http://wiki.lustre.org/.

Rao, N.S.V., Yu, W., Wing, W.R., Poole, S.W., Vetter, J.S., 2008. Wide-area performance profiling of 10GigE and InfiniBand technologies, in: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC).

Rashti, M.J., Grant, R.E., Afsahi, A., Balaji, P., 2009. iWARP redefined: Scalable connectionless communication over high-speed Ethernet, in: Proceedings of 16th International Conference on High Performance Computing (HiPC).

Ren, Y., Li, T., Yu, D., Jin, S., Robertazzi, T., 2012a. Middleware support for rdma-based data transfer in cloud computing, in: Proceedings of High-Performance Grid and Cloud Computing Workshop.

Ren, Y., Li, T., Yu, D., Jin, S., Robertazzi, T., Tierney, B., Pouyoul, E., 2012b. Protocols for wide-area data-intensive applications: Design and performance issues, in: Proceedings of the IEEE/ACM Conference on Supercomputing SC'12.

Subramoni, H., Lai, P., Kettimuthu, R., Panda, D.K., 2010. High performance data transfer in grid environment using gridftp over infiniband, in: Int'l Symposium on Cluster Computing and the Grid (CCGrid).

Subramoni, H., Lai, P., Luo, M., Panda, D.K., 2009. RDMA over Ethernet: A preliminary study, in: Proceedings of Cluster Computing Workshops, CLUSTER'09.

Suzumura, T., Tatsubori, M., Trent, S., Tozawa, A., Onodera, T., 2009. Highly scalable web applications with zero-copy data transfer.

The Internet Engineering Task Force (IETF), 2006. RFC 4392 - IP over InfiniBand (IPoIB) Architecture.

UltraLight, 2012. Ultralight: An ultrascale information system for data intensive research: http://ultralight.caltech.edu/website/ultralight/html/index.html.

Yeh, E., Chao, H., Mannem, V., Gervais, J., Booth, B., 2002. Introduction to TCP/IP Offload Engine (TOE). 10 Gigabit Ethernet Alliance (10GEA) .

Yu, W., Rao, N.S., Wyckoff, P., Vette, J.S., 2008. Performance of RDMA-capable storage protocols on wide-area network, in: Proceedings of Petascale Data Storage Workshop.