Supporting RDMA capable network compatibility and features for regular network adapters

P. BALAJI, H. -W. JIN, K. VAIDYANATHAN AND D. K. PANDA

Technical Report
Ohio State University (OSU-CISRC-6/05-TR37)

Supporting RDMA capable network compatibility and features for regular network adapters*

P. Balaji H. -W. Jin K. Vaidyanathan D. K. Panda

Department of Computer Science and Engineering
The Ohio State University
Columbus, Ohio 43210
{balaji, jinhy, vaidyana, panda}@cse.ohio-state.edu

Abstract

With aggressive initiatives in the offloaded technology present on network adapters, the user market is now distributed amongst various technology levels including regular Ethernet network adapters, TCP Offload Engines (TOEs) and the recently introduced Remote Direct Data Placement (RDDP) capable networks. While RDDP networks provide all the features provided by its predecessors (TOEs and regular Ethernet network adapters) and a new richer programming interface, in order to achieve a wide-spread acceptance they have to provide backward compatibility. In this aspect, two important issues need to be considered. First, not all network adapters support RDDP; thus, software compatibility for regular network adapters (which have no offloaded protocol stack) with RDDP aware network adapters needs to be achieved. Second, rewriting existing applications using the new RDDP interface is cumbersome and impractical; thus it is desirable to have an extended sockets interface which allows existing applications to run directly without any modifications and at the same time exposes the richer feature set of RDDP to the applications. In this paper, we design and implement a software stack to handle both these issues. Specifically, (i) the software stack provides applications with a sockets interface that has been extended with the rich RDDP features and (ii) it is capable of emulating the functionality of the RDDP stack in software to provide compatibility for regular Ethernet adapters with RDDP offloaded networks.

Keywords: Remote Direct Data Placement, RNIC, iWARP, and RDMA

1 Introduction

While TCP/IP [14] is considered the most ubiquitous standard for transport and network protocols, the host-based implementation of TCP/IP has not been able to scale very well with the sky-rocketing network speeds. In high-speed networks, the CPU has to dedicate more processing to handle the network traffic than to the applications it is running. Partial and complete Protocol Offload Engines (POEs) such as the TCP/IP Offload Engines (TOEs) [23] have provided a mechanism by which the host computational requirements of the TCP/IP stack can be curbed. Most TOEs retain the standard sockets interface while replacing the host-based TCP/IP stack with the hardware offloaded TCP/IP stack [10]; this allows transparent compatibility for existing applications to be directly deployed on to TOEs.

Though TOEs have been able to handle most of the inefficiencies of the host-based TCP/IP stack, they are still plagued with some of the limitations in order to maintain backward compatibility with the existing infrastructure and applications. For example, the traditional sockets interface is several times not the best interface to allow high performance communication [4]. Several techniques used with the sockets interface (e.g., pick-and-post, where the receiver first posts a small buffer to read the header information and then decides the length of the actual data buffer to be posted) make it difficult to efficiently perform zero-copy data transfers with such an interface.

A new initiative by IETF called Remote Direct Data Placement (RDDP) [22] was started to tackle such limitations with basic TOEs and other POEs. The RDDP standard, when offloaded on to the network adapter, provides two primary extensions to the TOE stack: (i) it exposes a rich interface including zero-copy and asynchronous communication providing capabilities for one-sided communication as well and (ii) it extends the TCP/IP implementation on the TOE to allow such communication while maintaining compatibility with the existing TCP/IP implementations.

With such aggressive initiatives in the offloaded technology present on network adapters, the user market is now distributed amongst these various technology levels. Several users still use regular Ethernet network adapters (35.2% of the Top500 supercomputers use Ethernet with most, if not all, of them relying on regular Gigabit Ethernet adapters [1]) which do not perform any kind of protocol offload; then we have users who utilize the offloaded protocol stack provided with TOEs; finally with the advent of RDDP offloaded network adapters, a part of the user group is also moving towards such RDDP aware networks.

TOEs and regular Ethernet network adapters have been compatible with respect to both the data format sent out on the hardware (Ethernet + TCP + IP + data payload) as well as with the interface they exposed to the applications (both using the sockets interface). With RDDP offloaded network adapters, such compatibility is disturbed to some extent. For example, currently an RDDP capable network adapter can only communicate with another RDDP capable network adapter[1].

*This project is supported in part by the DOE grant #DE-FG02-04ER86204 through Seafire Micros Inc.

[1]The intermediate switches, routers, etc., need not, however, support RDDP.

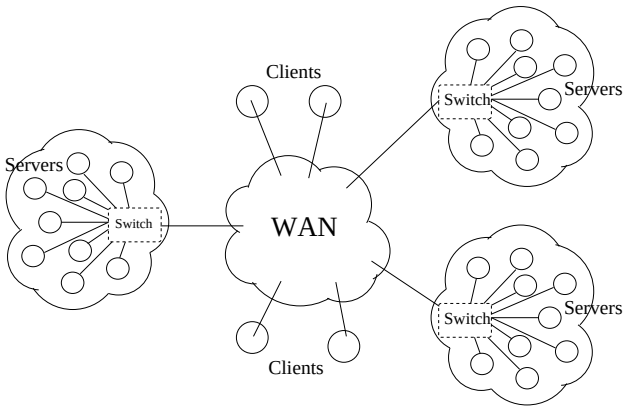

Figure 1. Multiple clients with regular network adapters communicating with servers using RDDP aware network adapters.

Also, the interface exposed by the RDDP network is no longer sockets; it is a much richer and newer interface. As several network vendors have learnt it the hard way, a network architecture can not survive unless it maintains compatibility with the existing and widely used network infrastructure.

Thus, for a wide-spread acceptance of the RDDP stack, two important extensions seem to be quite necessary.

1. Let us consider a scenario where a server handles requests from various client nodes (Figure 1). In this scenario, for performance reasons, it is desirable for the server to use the RDDP interface for all communication and might use an RDDP offloaded network adapter. The client on the other hand might *NOT* be equipped with an RDDP aware network card (e.g., it might use a regular Fast Ethernet or Gigabit Ethernet adapter or even a TOE). For such and various other scenarios, it becomes quite necessary to have a software implementation of the RDDP stack on such networks in order to maintain compatibility with the hardware offloaded RDDP implementations.

2. Though the RDDP interface provides a richer feature-set as compared to the sockets interface, it requires applications to be rewritten with this interface. While this is not a concern for new applications, it is quite cumbersome and impractical to port existing applications to use this new interface. Thus, it is desirable to have an extended sockets interface which allows existing applications to run directly without any modifications and at the same time allows a richer feature set including zero-copy, asynchronous and one-sided communication.

In general, we would like to have a software stack which would provide both the above mentioned extensions for regular Ethernet network adapters as well as TOEs. In this paper, however, we focus only on regular Ethernet adapters and design and implement a software stack to provide both these extensions. Specifically, (i) the software stack provides applications with a sockets interface that has been extended with the rich RDDP features and (ii) it is capable of emulating the functionality of the RDDP stack in software to provide compatibility for regular Ethernet adapters with RDDP offloaded networks.

The rest part of the paper is organized as follows: In Section 2, we provide a brief background about TOEs and the RDDP protocol stack standard. In Section 3 we go into details about the design and implementation of our software RDDP-aware extended sockets interface. In addition, we suggest design alternatives for software implementation of a DDP. We present the experimental evaluation of our stacks in Section 4, some related work in Section 5 and conclude the paper in Section 6.

2 Background

In this section, we provide a brief background about TOEs and the RDDP protocol stack standard.

2.1 TCP Offload Engines

The processing of traditional protocols such as TCP/IP and UDP/IP is accomplished by software running on the central processor, CPU or microprocessor, of the server. As network connections scale beyond Gbps speeds, the CPU becomes burdened with the large amount of protocol processing required. Resource-intensive memory copies, checksum computation, interrupts, and reassembling of out-of-order packets put a tremendous amount of load on the host CPU. In high-speed networks, the CPU has to dedicate more processing to handle the network traffic than to the applications it is running. TCP Offload Engines (TOEs) [23] are emerging as a solution to limit the processing required by CPUs for networking.

The basic idea of a TOE is to offload the processing of protocols from the host processor to the hardware on the adapter or in the system. A TOE can be implemented with a network processor and firmware, specialized ASICs, or a combination of both. Most TOE implementations available in the market concentrate on offloading the TCP and IP processing, while a few of them focus on other protocols such as UDP/IP.

As a precursor to complete protocol offloading, some operating systems started incorporating support for features to offload some compute-intensive features from the host to the underlying adapter, e.g., checksum computation. But as Ethernet speeds increased beyond Gbps, the need for further protocol processing offload became a clear requirement. Some GigE adapters complemented this requirement by offloading TCP/IP and UDP/IP segmentation onto the network adapter [13, 8]. With the advent of multi-gigabit networks, the host-processing requirements became so burdensome that they ultimately led to adapter solutions with *complete* protocol offload.

2.2 RDDP Specification Overview

The RDDP protocol stack usually comprises of three protocol layers other than the TCP/IP protocol stack: (i) RDMA interface, (ii) Direct Data Placement (DDP) layer and (iii) Marker PDU Aligned (MPA) layer.

The RDMA layer is a thin interface which allows applications to interact with the DDP layer. The DDP layer uses an

IP based reliable protocol stack such as TCP/IP to perform the actual data transmission. The MPA stack is an extension to the TCP/IP stack in order to maintain backward compatibility with the existing infrastructure. Details about the DDP and MPA layers are provided in Sections 2.2.1 and 2.2.2 respectively.

2.2.1 Direct Data Placement (DDP)

DDP standard was developed to serve two purposes. First, the protocol should be able to provide high performance in SAN and other controlled environments by utilizing an offloaded protocol stack and zero-copy data transfer between host memories. Second, the protocol should maintain compatibility with the existing IP infrastructure using an implementation over an IP based reliable transport layer stack. Maintaining these two, sometimes contradicting, features involves novel designs for several aspects. We describe some of these in this section.

In-Order Delivery and Out-of-Order Placement: DDP relies on de-coupling of placement and delivery of messages, i.e., placing the data in the user buffer is performed in a decoupled manner with informing the application that the data has been placed in its buffer. In this approach, the sender breaks the message into multiple segments of MTU size; the receiver places each segment directly into the user buffer, performs book-keeping to keep track of the data that has already been placed and once all the data has been placed, informs the user about the arrival of the data. This approach has two benefits: (i) there are no copies involved in this approach and (ii) suppose a segment is dropped, the future segments do not need to be buffered till this segment arrives; they can directly be placed into the user buffer as and when they arrive. The approach used, however, involves two important features to be satisfied by each segment: Self-Describing and Self-Contained segments.

The Self-Describing property of segments involves adding enough information in the segment header so that each segment can individually be placed at the appropriate location without any information from the other segments. The information contained in the segment includes the Message Sequence Number (MSN), the Offset in the message buffer to which the segment has to be placed (MO) and others. Self-Containment of segments involves making sure that each segment contains either a part of a single message, or the whole of a number of messages, but not parts of more than one message.

Middle Box Fragmentation: DDP is an end-to-end protocol. The intermediate nodes do not have to support DDP. This means that the nodes which forward the segments between two DDP nodes, do not have to follow the DDP specifications. In other words, DDP is transparent to switches with IP forwarding and routing. However, this might lead to a problem known as "Middle Box Fragmentation" for Layer IV or above switches.

Layer III Switches: Layer III switches can be typically thought of as the routers. These switches use more intelligent forwarding algorithms depending on the IP address of the destination node(s). This requires the forwarding to take place at the IP layer. Thus, there might be a possibility of fragmentation of the segment at the IP layer (since this switch might not be aware of DDP). In this case we rely on the re-assembly mechanism of IP to deliver the complete segment. Though this requires buffering of data, there is no loss of functionality with these kind of switches.

Layer IV (or above) Switches: Layer IV switches are protocol specific and capable of making more intelligent decisions regarding the forwarding of the arriving message segments. The forwarding in these switches takes place at the TCP layer. The modern load-balancers (which fall under this category of switches) allow a hardware based forwarding of the incoming segments. They support optimization techniques such as TCP Splicing [7] in their implementation. The problem with such an implementation is that, there need not be a one-to-one correspondence between the segments coming in and the segments going out. This means that the segments coming in might be re-fragmented and/or re-assembled at the switch.

It is to be noted that the segments coming in-order would not be effected by this. However, for segments arriving out of order, this might require buffering at the receiver node, since the receiver cannot recognize the DDP headers for each segments. This mandates that the protocol not assume the self-containment property at the receiver end, and add additional information in each segment to help recognize the DDP header.

2.2.2 Marker PDU Aligned (MPA)

In case of "Middle Box Fragmentation", the self-containment property of the segments might not hold true. The solution for this problem needs to have the following properties:

- It must be independent of the segmentation algorithm used by TCP or any layer below it.
- A deterministic way of determining the segment boundaries are preferred.
- It should enable out-of-order placement of segments. In the sense, the placement of a segment must not require information from any other segment.
- It should contain a stronger data integrity check like the Cyclic Redundancy Check (CRC).

The solution to this problem involves the development of the Marker PDU[2] Aligned (MPA) protocol [9]. Figure 2 illustrates the new segment format with MPA. This new segment is known as the FPDU or the Framing Protocol Data Unit. The FPDU format has three essential changes:

- Markers: Strips of data to point to the DDP header in case of middle box fragmentation

[2]PDU stands for a Protocol Data Unit; essentially a unit of data or a segment of data given by the layer above it, the DDP layer in this case.

- Cyclic Redundancy Check (CRC): A Stronger Data Integrity Check
- Segment Pad Bytes

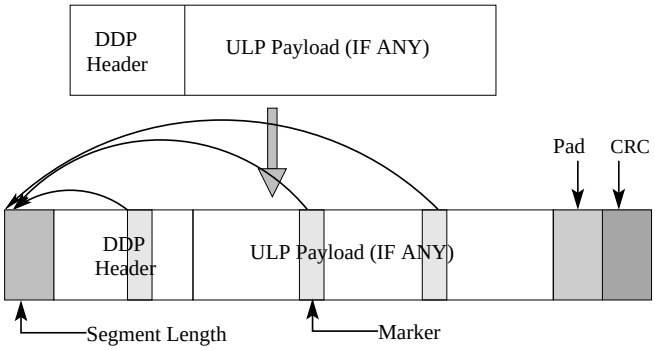

Figure 2. Marker PDU Aligned (MPA) protocol Segment format

The markers placed as a part of the MPA protocol are strips of data pointing to the MPA header and spaced uniformly based on the TCP sequence number. This provides the receiver with a deterministic way to find the markers in the received segments and eventually find the right header for the segment.

3 Designing Issues and Implementation Details

To provide compatibility for regular Ethernet network adapters with hardware offloaded RDDP implementations, we propose a software stack to be used on the various nodes. We break down the stack into two layers, namely, the *Extended sockets interface* and the *software RDDP layer*. Amongst these two layers, the *Extended sockets interface* is generic for all kinds of RDDP implementations; for example it can be used over the *software RDDP layer* for regular Ethernet networks presented in this paper, over a *software RDDP layer* for TOEs, or over hardware offloaded RDDP implementations. Further, for the *software RDDP layer* for regular Ethernet networks, we propose two kinds of implementations: user-level DDP and kernel-level DDP. Applications, however, only interact with the extended sockets interface which uses the appropriate RDDP stack available on the system. The different implementations of the stack are illustrated in Figure 3. In this paper, we only concentrate on the design and implementation of the stack on regular Ethernet network adapters (Figures 3a and 3b).

3.1 Extended Sockets Interface

The extended sockets interface is designed two serve two purposes. First, it provides a transparent compatibility for existing sockets applications to run. Second, it exposes the richer interface provided by RDDP to the applications to utilize as and when required. For existing sockets applications (which do not use the richer DDP interface), the extended sockets interface just passes on the control to the underlying sockets layer. This underlying sockets layer could be the traditional host-based TCP/IP sockets for regular Ethernet networks or a High Performance Sockets layer on top of TOEs [10] or other POEs. For applications which *DO* use the richer DDP interface, the extended sockets interface maps the calls to appropriate calls in the interface provided by the underlying DDP layer. Again, the underlying DDP layer could be a software implementation of DDP (for regular Ethernet network adapters or TOEs) or a hardware DDP implementation.

In order to extend the sockets interface to support the richer interface provided by DDP, certain sockets based calls need to be aware of the existence of DDP. The setsockopt() system call, for example, is a standard sockets call. But, it can be used to set a given socket to *DDPMODE*. All future communication using this socket will be transferred using the DDP transport layer. Further, `read()`, `write()` and several other socket calls need to check if the socket mode is set to *DDPMODE* before carrying out any communication. This requires modifications to these calls, while making sure that existing sockets applications (which do not use the extended sockets interface) are not hampered.

In our implementation of the extended sockets interface, we carried this out by overloading the standard *libc* library using our own extended sockets interface. This library first checks whether a given socket is currently in *DDPMODE*. If it is, it carries out the standard DDP procedures to transmit the data. If it is not, the extended sockets interface dynamically loads the *libc* library to pass on the control to the traditional sockets interface for the particular call.

3.2 User-Level DDP

In this approach, we designed and implemented the entire RDDP stack in user space above the sockets layer (Figure 3a). Being implemented in user-space and above the sockets layer, this implementation is very portable across various hardware and software platforms[3]. However, the performance it can deliver might not be optimal. Extracting the maximum possible performance for this implementation requires efficient solutions for several issues including (i) supporting gather operations, (ii) supporting non-blocking operations, (iii) asynchronous communication, (iv) handling shared queues during asynchronous communication and several others. In this section, we discuss some of these issues and propose various solutions to handle these issues.

Gather operations supported by the DDP specifications: The DDP specification defines gather operations for a list of data segments to be transmitted. Since, the user-level DDP implementation uses TCP as the underlying mode of communication, there are interesting challenges to support this without any additional copy operations. Some of the approaches we considered are as follows:

1. The simplest approach would be to copy data into a stan-

[3]Though the user-level DDP implementation is mostly in the user-space, it requires a small patch in the kernel to extend the MPA CRC to include the TCP header too and to provide information about the TCP sequence numbers used in the connection in order to place the markers at appropriate places (this cannot be done from user-space).

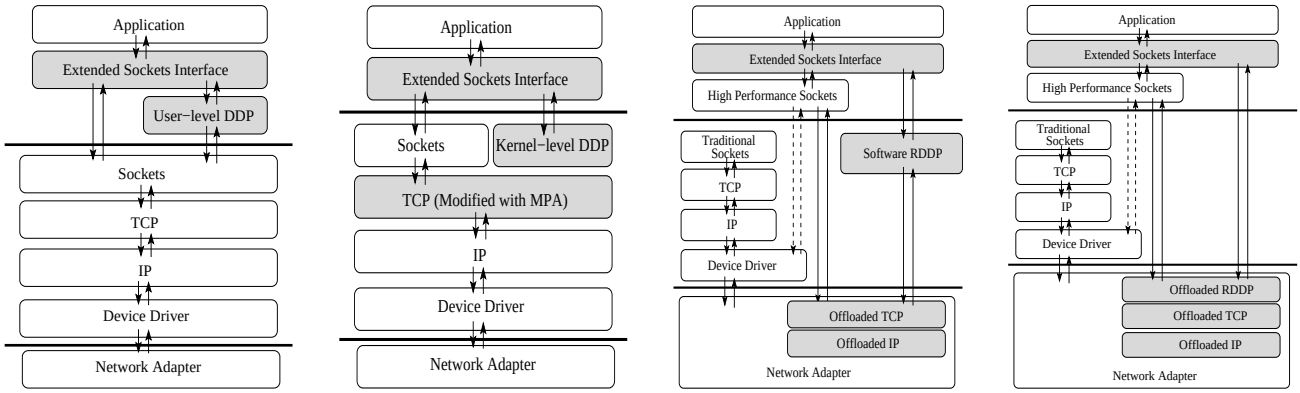

Figure 3. Extended sockets interface with different Implementations of the RDDP protocol stack: (a) User-Level DDP (for regular Ethernet networks), (b) Kernel-Level DDP (for regular Ethernet networks), (c) Software DDP (for TOEs) and (d) Hardware offloaded DDP (for RDDP capable network adapters).

dard buffer and send the data out from this buffer. This approach is very simple but would require an extra copy of the data.

2. The second approach is to use the scatter-gather readv() and writev() calls provided by the traditional sockets interface. Though in theory traditional sockets supports scatter/gather of data using readv() and writev() calls, the actual implementation of these calls is specific to the kernel. It is possible (as is currently implemented in the 2.4.x linux kernels) that the data in these list of buffers be sent out as different messages and not aggregated into a single message. While this is perfectly fine with TCP, it creates a lot of fragmentation for DDP, forcing it to have additional buffering to take care of this.

3. The third approach is to use the TCP_CORK mechanism provided by TCP/IP. The TCP_CORK socket option allows data to be pushed into the socket buffer. However, until the entire socket buffer is full, data is not sent onto the network. This allows us to copy all the data from the list of the application buffers directly into the TCP socket buffers before sending them out on to the network, thus saving an additional copy and at the same time guaranteeing that all the segments are sent out as a single message.

Non-blocking communication operations support: As with RDDP, the extended sockets also supports non-blocking communication operations. This means that the application layer can just post a send descriptor; once this is done, it can carry out with its computation and check for completion at a later time. In our approach, we use a multi-threaded design for user-level DDP to allow non-blocking communication operations (Figure 4). As shown in the figure, the application thread posts a send and a receive to the asynchronous threads and returns control to the application; these asynchronous threads take care of the actual data transmission for send and receive, respectively. To allow the data movement between the threads, we use pthreads() rather than fork(). This approach

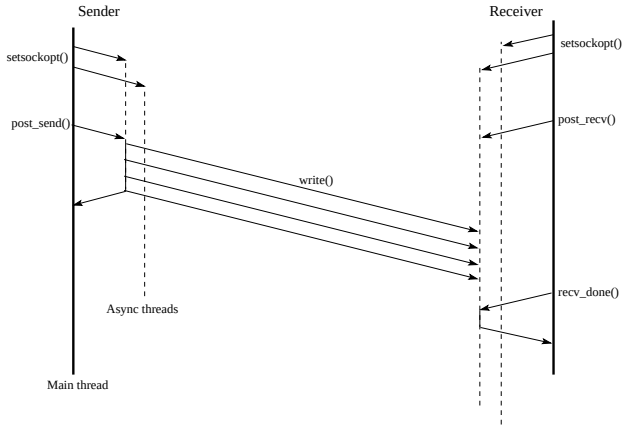

Figure 4. Asynchronous Threads Based Non-Blocking Operations

gives the flexibility of a shared physical address space for the application and the asynchronous threads. The pthreads() specification states that all pthreads should share the same process ID (pid). Operating Systems such as Solaris follow this specification. However, due to the flat architecture of Linux, this specification was not followed in the Linux implementation. This means that all pthreads() have a different PID in Linux. We use this to carry out inter-thread communication using inter-process communication (IPC) primitives.

Asynchronous communication supporting non-blocking operations: In the previous issue (non-blocking communication operations support), we chose to use pthreads to allow cloning of virtual address space between the processes. Communication between the threads was intended to be carried out using IPC calls. The DDP specification does not allow a shared queue for the multiple sockets in an application. Each socket has separate send and receive work queues where descriptors posted for that socket are placed. We use UNIX socket connections between the main thread and the asynchronous threads. The first socket set to *DDPMODE* opens a connection with the asynchronous threads and all subsequent sockets use this connection in a persistent manner. This option

allows the main thread to post descriptors in a non-blocking manner (since the descriptor is copied to the socket buffer) and at the same time allows the asynchronous thread to use a `select()` call to make progress on all the *DDPMODE* sockets as well as the inter-process communication. It is to be noted that though the descriptor involves an additional copy by using this approach, the size of a descriptor is typically very small (around 60 bytes in the current implementation), so this copy does not affect the performance too much.

3.3 Kernel-Level DDP

The kernel-level DDP is built directly over the TCP/IP stack bypassing the traditional sockets layer as shown in Figure 3b. This implementation requires modifications to the kernel and hence is not as portable as the user-level implementation. However, it can deliver a better performance as compared to the user-level DDP. The kernel-level design of DDP has several issues and design challenges. Some of these issues and the solutions chosen for them are presented in this section.

Though most part of the DDP implementation can be done completely above the TCP stack by just inserting modules (with appropriate symbols exported from the TCP stack), there are a number of changes that are required for the TCP stack itself. For example, ignoring the remote socket buffer size, efficiently handling out-of-order segments, etc. require direct changes in the core kernel. This forced us to recompile the linux kernel as a patched kernel. We have modified the base kernel.org kernel version 2.4.18 to the patched kernel to facilitate these changes.

Immediate copy to user buffers: Since DDP provides non-blocking communication, copying the received data to the user buffers is a tricky issue. One simple solution is to copy the message to the user buffer when the application calls a completion function, i.e., when the data is received the kernel just keeps it with itself and when the application checks with the kernel if the data has arrived, the actual copy to the user buffer is performed. This approach, however, loses out on the advantages of non-blocking operations as the application has to block waiting for the data to be copied while checking for the completion of the data transfer. Further, this approach requires another kernel trap to perform the copy operation.

The approach we used in our implementation is to immediately copy the received message to the user buffer as soon as the kernel gets the message. An important issue to be noted in this approach is that since multiple processes can be running on the system at the same time, the current process scheduled can be different with the owner of the user buffer for the message; thus we need a mechanism to access the user buffer even when the process is not currently scheduled. To do this, we pin the user buffer (prevent it from being swapped out) and map it to a kernel memory area. This ensures that the kernel memory area and the user buffer point to the same physical address space. Thus, when the data arrives, it is immediately copied to the kernel memory area and is automatically reflected into the user buffer.

User buffer registration: The DDP specification defines an API for the buffer registration, which performs pre-communication processes such as buffer pinning, address translation between virtual and physical addresses, etc. These operations are required mainly to achieve a zero-copy data transmission on RDDP offloaded network adapters. Though this is not critical for the kernel-level DDP implementation as it anyway performs a copy, this can protect the buffer from being swapped out and avoid the additional overhead for page fetching. Hence, in our approach, we do pin the user-buffer.

Efficiently handling out-of-order segments: DDP allows out-of-order placement of data. This means that out-of-order segments can be directly placed into the user-buffer without waiting for the intermediate data to be received. In our design, this is handled by placing the data directly and maintaining a queue of received segment sequence numbers. At this point, technically, the received data segments present in the kernel can be freed once they are copied into the user buffer. However, the actual sequence numbers of the received segments are used by TCP for acknowledgments, re-transmissions, etc. Hence, to allow TCP to proceed with these without any hindrance, we defer the actual freeing of these segments till their sequence numbers cross TCP's unacknowledged window.

4 Experimental Evaluation

In this section, we perform experimental evaluations for the extended sockets interface using the user- and kernel-level DDP implementations. Due to time constraints, we have performed evaluations over a 1Gbps network currently; we hope to present results for a 10-Gigabit Ethernet network too during the camera-ready version of the paper.

The experimental test-bed used is as follows: Two Pentium III 700MHz Quad machines, each with an L2-cache size of 1 MB and 1 GB of main memory. The interconnect was a Gigabit Ethernet network with Alteon NICs on each machine connected using a Packet Engine switch. We used the RedHat 9.0 linux distribution installed with the kernel.org kernel version 2.4.18.

4.1 Micro-benchmark Evaluation

In this section, we present the ping-pong latency and uni-directional bandwidth achieved by two kinds of tests. In the first set of tests, we measure the performance achieved for standard sockets based applications; for such applications, the extended sockets interface does basic processing to ensure that the applications do not want to utilize the extended interface (by checking if the *DDPMODE* is set) and passes on the control to the traditional sockets layer. In the second kind of tests, we use applications which utilize the extensions provided by the sockets interface based on the RDDP interface; for such applications, the extended sockets interface utilizes the software RDDP implementations to carry out the communication.

The latency test is carried out in a standard ping-pong fashion. The sender sends a message and waits for a reply from receiver. The time for this is recorded by the sender and it is divided by two to get the one-way latency. For measuring the bandwidth, a simple window based approach was followed. The sender sends $WindowSize$ number of messages and wait

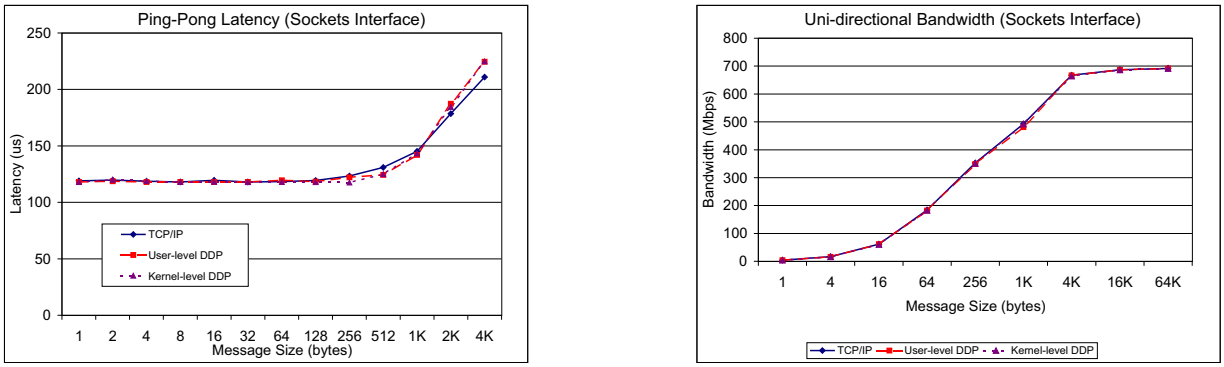

Figure 5. Micro-Benchmark Evaluation for applications using the standard sockets interface: (a) Ping-pong latency and (b) Uni-directional bandwidth

for a message from the receiver for every $WindowSize$ messages.

The results for the applications with the standard unmodified sockets interface are presented in Figure 5. As shown in the figure, the extended sockets interface adds very minimal overhead to existing sockets applications for both the latency and the bandwidth tests.

For the applications using the extended interface, the results are shown in Figure 6. We can see that the user-level DDP and kernel-level DDP incur an overhead of about 100μs and 5μs comparing with TCP/IP, respectively. There are several reasons for this overhead. First, the user- and kernel-level DDP implementations are built over the sockets and TCP/IP respectively; so they are not expected to give a better performance than TCP/IP itself. Second, the user-level DDP has additional threads for non-blocking operations and requires IPC between threads. Also, the user-level DDP performs locking for shared queues between threads. However, it is to be noted that the basic purpose of these implementations is to allow compatibility for regular network adapters with RDDP aware network adapters and the performance is not the primary goal of these implementation. We can observe that both user- and kernel-level DDP can achieve about 550Mbps in the peak bandwidth. An interesting result in the figure is that the bandwidth of user- and kernel-level DDP for small and medium message sizes is lesser compared to TCP/IP. This is mainly because they disable Nagle's algorithm in order to try to maintain message boundaries. For large messages, we see a degradation compared to TCP/IP due to the additional overhead of CRC data integrity performed by the DDP implementations. We see a slightly better performance for user-level DDP as compared to kernel-level DDP for large messages. We are not entirely sure about the reason for this; we are currently to analyzing the stacks in more detail to understand this behavior.

4.2 Impact of Packet Loss

In this section we compare the behavior of user-level DDP with that of kernel-level DDP when there are packet losses in the system. In TCP/IP sockets, if the n^{th} packet is lost and the m^{th} ($m > n$) packet arrived, the receiver has to wait for the n^{th} packet before the subsequent packets are copied into the user buffer losing out on any opportunity to pipeline the copy. Since the user-level DDP implementation is on top of the sockets interface, it too suffers from this issue. However, the kernel-level DDP can directly place the m^{th} packets without waiting the n^{th} packet as we have described in Section 3.3. This allows a better pipelining of the copies for the kernel-level DDP implementation as compared to TCP/IP as well as the user-level DDP implementation. It is to be noted that for small amounts of packet drops, TCP/IP goes to fast retransmit mode and such pipelining would be critical for its performance. However, as the packet loss rate becomes very high, TCP/IP goes to congestion mode and drastically reduces the congestion window; in this stage, we do not expect the kernel-level DDP implementation to give a significant improvement in the performance.

Due to time restrictions, we have not been able to get the relevant data for this experiment and hope to present this in the final version of the paper.

5 Related Work

Several researchers, including Feng et. al. and ourselves, have performed a significant amount of researchers on the performance of RDDP-unaware network adapters including regular Ethernet-based network adapters [12, 11, 4] as well as TCP Offload Engines [10, 2]. Also, there has been a lot of research for implementing high performance sockets over various protocol offload engines including TOEs [19, 18, 5, 6, 3, 16, 17]. However, all this literature focuses on the improving the performance of the sockets interface for host-based or offloaded protocol stacks and does not deal with any kind of extensions to it.

Shivam et. al. had implemented a new protocol stack, *EMP* [21, 20], on top of Gigabit Ethernet which provides RDDP like features to the applications. However, this protocol has a completely different interface and cannot support sockets based applications directly. Further, this protocol is not IP compatible and thus cannot be used in a WAN environment unlike TOEs or RDDP aware network adapters. We had previously implemented a high performance sockets imple-

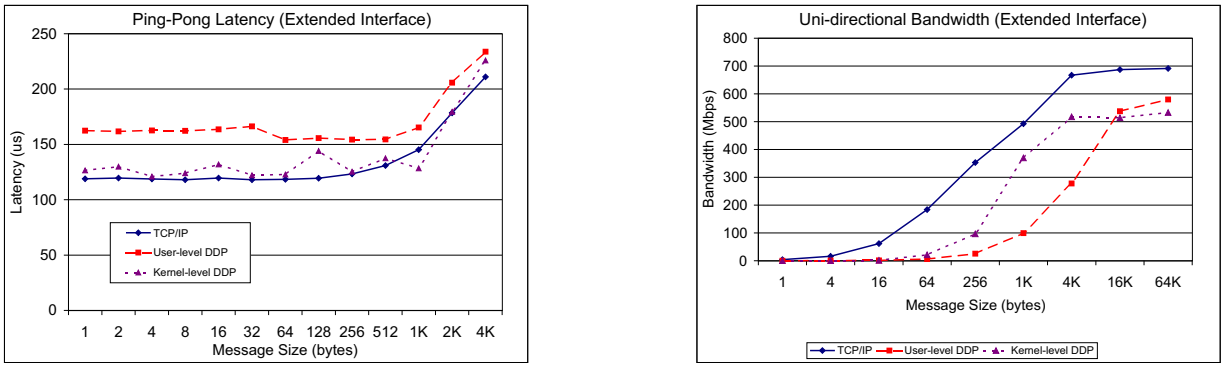

Figure 6. Micro-Benchmark Evaluation for applications using the extended RDDP interface: (a) Ping-pong latency and (b) Uni-directional bandwidth

mentation over *EMP* [5]; while this allows compatibility for existing sockets applications, it still does not allow IP compatibility. Further, this layer only provides the basic sockets interface with no RDDP based extensions as such.

Jagana et. al. have developed a software system to provide kernel support for RDDP and other RDMA aware networks [15]. This work can be considered a complementary development towards RDDP aware networks while our work deals with RDDP capabilities for regular Ethernet networks. We hope to unify our solution with this software system in order to avoid further fragmentation in the software stacks provided to end users.

6 Concluding Remarks

A new initiative by IETF called Remote Direct Data Placement (RDDP) was started to tackle several limitations with TOEs while providing a completely new and feature rich interface for applications to utilize. For a wide-spread acceptance of the RDDP stack, however, two important issues need to be considered. First, software compatibility needs to be provided for regular network adapters (which have no offloaded protocol stack) with RDDP aware network adapters. Second, the predecessors of RDDP aware network adapters such as TOEs and host-based TCP/IP stacks used the sockets interface for applications to utilize them while the RDDP networks provide a completely new and richer interface. Rewriting existing applications using the new RDDP interface is cumbersome and impractical; thus its desirable to have an extended sockets interface which allows existing applications to run directly without any modifications and at the same time exposes the richer interface of RDDP including zero-copy, asynchronous and one-sided communication. In this paper, we design and implement a software stack to provide both these extensions.

As continuing work, we are currently working in two broad directions. First, we are providing the extended sockets interface for hardware offloaded RDDP stacks such as the network adapters provided by Ammasso as well as the TCP Offload Engines (TOEs). This will allow a common interface for all applications whether they are utilizing regular NICs (software RDDP), TCP Offload Engines (software RDDP) and RDDP offloaded network adapters (hardware RDDP). Second, we are developing a simulator which can provide details about the actual architectural requirements for different designs of the offloaded RDDP stack.

References

[1] Top500 supercomputer list. http://www.top500.org, November 2004.

[2] P. Balaji, W. Feng, Q. Gao, R. Noronha, W. Yu, and D. K. Panda. Head-to-TOE Evaluation of High-Performance Sockets over Protocol Offload Engines. Technical Report LA-UR-05-2635, Los Alamos National Laboratory, June 2005.

[3] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda. Sockets Direct Protocol over InfiniBand in Clusters: Is it Beneficial? In *the Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, Austin, Texas, March 10-12 2004.

[4] P. Balaji, H. V. Shah, and D. K. Panda. Sockets vs RDMA Interface over 10-Gigabit Networks: An In-depth analysis of the Memory Traffic Bottleneck. In *Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations, and Technologies (RAIT)*, San Diego, CA, Sep 20 2004.

[5] P. Balaji, P. Shivam, P. Wyckoff, and D.K. Panda. High Performance User Level Sockets over Gigabit Ethernet. In *the Proceedings of Cluster Computing*, Chicago, IL, Sept 23-26 2002.

[6] P. Balaji, J. Wu, T. Kurc, U. Catalyurek, D. K. Panda, and J. Saltz. Impact of High Performance Sockets on Data Intensive Applications. In *the Proceedings of the IEEE International Conference on High Performance Distributed Computing (HPDC 2003)*, June 2003.

[7] Ariel Cohen, Sampath Rangarajan, and Hamilton Slye. On the Performance of TCP Splicing for URL-aware Redirection. In *the Proceedings of the USENIX Symposium on Internet Technologies and Systems*, October 1999.

[8] Chelsio Communications. http://www.chelsio.com/.

[9] P. Culley, U. Elzur, R. Recio, and S. Bailey. Marker PDU Aligned Framing for TCP Specification, November 2002.

[10] W. Feng, P. Balaji, C. Baron, L. N. Bhuyan, and D. K. Panda. Performance Characterization of 10-Gigabit Ethernet: From Head to TOE. Technical Report LA-UR-05-2635, Los Alamos National Laboratory, April 2005.

[11] W. Feng, J. Hurwitz, H. Newman, S. Ravot, L. Cottrell, O. Martin, F. Coccetti, C. Jin, D. Wei, and S. Low. Optimizing 10-Gigabit Ethernet for Networks of Workstations, Clusters and Grids: A Case Study. In *SC '03*.

[12] J. Hurwitz and W. Feng. End-to-End Performance of 10-Gigabit Ethernet on Commodity Systems. *IEEE Micro '04*.

[13] Ammasso Incorporation. http://www.ammasso.com/.

[14] University of Southern California Information Sciences Institute. TRANSMISSION CONTROL PROTOCOL (TCP), RFC 793, 1981.

[15] V. Jagana, B. Metzler, and F. Neeser. Open RDMA Project: Building an RDMA Ecosystem for Linux. In *the workshop on Remote Direct Memory Access (RDMA): Applications, Implementations, and Technologies (RAIT)*, 2004.

[16] H. W. Jin, P. Palaji, C. Yoo, J. Y. Choi, and D. K. Panda. Exploiting NIC Architectural Support for Enhancing IP Based Protocols on High Performance Networks. *Journal of Parallel and Distributed Computing(JPDC)*. in press.

[17] H. W. Jin, C. Yoo, and S. K. Park. Stepwise Optimizations of UDP/IP on a Gigabit Network. In *Euro-Par 2002*, April 2002.

[18] J. S. Kim, K. Kim, and S. I. Jung. SOVIA: A User-level Sockets Layer over Virtual Interface Architecture. In *Proceedings of Cluster Computing*, 2001.

[19] H. V. Shah, C. Pu, and R. S. Madukkarumukumana. High Performance Sockets and RPC over Virtual Interface (VI) Architecture. In *Proceedings of CANPC workshop*, 1999.

[20] P. Shivam, P. Wyckoff, and D. K. Panda. Can user Level Protocols Take Advantage of Multi-CPU NIC? In *IPDPS '02*. accepted to be presented.

[21] P. Shivam, P. Wyckoff, and D. K. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *Int'l Conference on Supercomputing (SC '01)*, November 2001.

[22] Thomas Talpey Stephen Bailey. Remote Direct Data Placement (RDDP), April 2005.

[23] Eric Yeh, Herman Chao, Venu Mannem, Joe Gervais, and Bradley Booth. Introduction to tcp/ip offload engines (toe). White Paper, May 2002.