Proceedings of the 20th IASTED International Conference
Parallel and Distributed Computing and Systems (PDCS 2008)
November 16 - 18, 2008 Orlando, Florida, USA

THE EXTENDED SOCKETS INTERFACE FOR ACCESSING RDMA HARDWARE

Robert Russell
University of New Hampshire InterOperability Laboratory
121 Technology Drive, Suite 2
Durham, NH 03824-4716
rdr@iol.unh.edu

ABSTRACT
The Extended Sockets API (ES-API) is a specification published by the OpenGroup that defines extensions to the traditional socket API which include two major new features: asynchronous I/O, and memory registration. These features enable programmers to take advantage of today's multi-core processors and Remote Direct Memory Access (RDMA) network hardware, such as iWARP and InfiniBand interfaces, in a convenient yet efficient manner.

This paper describes the UNH EXS interface, an implementation of the ES-API that provides additional API facilities which enable a programmer to utilize RDMA network hardware while selectively choosing those features of this interface that are most germane to the particular application. In addition, the UNH EXS interface is implemented entirely in user space on the Linux operating system. This provides easy porting, modification and adoption of UNH-EXS, since it requires no changes to existing Linux kernels. Preliminary results demonstrate that applications based on EXS can achieve high bandwidth utilization and low CPU overhead.

KEY WORDS
Remote Direct Memory Access (RDMA), Extended Sockets (EXS), iWARP, 10GigEthernet.

1 Introduction

1.1 The ES-API

The Extended Sockets API (ES-API) [1] is a specification published by the OpenGroup [2] that defines extensions to the traditional socket API in order to provide improved efficiency in network programming. It contains two major new features: asynchronous I/O, and memory registration. These extensions can be a useful method for efficient, high-level access to Remote Direct Memory Access (RDMA) over IP [3]. The UNH EXS interface is an implementation of the ES-API that also provides some additional API facilities. In addition, the UNH EXS interface is implemented entirely in user space on the Linux operating system. This provides easy porting, modification and adoption of UNH-EXS, since it requires no changes to existing Linux kernels. The current implementation of the EXS interface runs on top of free software provided by the OpenFabrics Alliance (OFA) [4], called the OpenFabrics Enterprise Distribution (OFED) [5] (the OFED stack for short). This software runs in both user and kernel space, and provides efficient, asynchronous access to various types of RDMA networking hardware from different vendors. See Figure 1.

1.2 Remote Direct Memory Access

There are two major types of Remote Direct Memory Access (RDMA) interfaces on the market today: InfiniBand and iWarp. The OFED stack is designed to run on both of them. The UNH-EXS was initially implemented and tested only on iWarp. In principal, however, UNH-EXS should run on both interfaces, although that remains to be tried.

The new 10 Gigabit/second Ethernet (10GigEthernet) standard allows IP networks to reach competitive speeds, but without offloading onto specialized *Network Interface Cards* (NICs) standard, TCP/IP is not sufficient to make use of the entire 10GigEthernet bandwidth. This is due to data copying, packet processing and interrupt handling on the CPUs at each end of a TCP/IP connection. In a traditional TCP/IP network stack, an interrupt occurs for every packet sent and received, data is copied at least once in each host computer's memory (from user space into the kernel's TCP/IP buffers), and the CPU is responsible for processing multiple nested packet headers for all protocol levels in all incoming and outgoing packets. To eliminate these inefficiencies, specialized *Remote Direct Memory Access* (RDMA) hardware can be used. One type of RDMA hardware uses the *Internet Wide Area Remote Protocol* (iWARP) protocol suite [6, 7, 8] to move data directly from the memory of one computer to the memory of a remote computer without extra copying or CPU intervention at either end. The entire iWARP protocol suite is offloaded from the CPU onto the *RDMA Network Interface Card* (RNIC). The RNIC reduces the number of data copies that are required by the TCP/IP network to one per host (between main memory and the Ethernet wire), and offloads the bulk of the network processing from the CPU. This means applications utilizing RNICs have lower CPU usage than those utilizing standard NICs, and can achieve throughput close to the full capacity of 10GigEthernet links.

1.3 OFED Stack

The OpenFabrics Alliance (OFA) [4] provides a free software stack that provides a socket-like abstraction layer called the Communication Manager Abstraction (CMA), along with a verbs layer that is used to perform data transfers over RDMA hardware (together these pieces are referred to as the OpenFabrics Enterprise Distribution (*OFED*) stack). The OFED stack has been growing in popularity due to its active development community and inclusion in the Linux kernel. In addition, the OFED stack also provides a user-space library that allows user-space applications to use its API to interface directly with RDMA hardware. We therefore decided to implement UNH-EXS entirely in user-space on top of the OFED user-space API, as shown in Figure 1. The major reasons for this were the ease of implementing and debugging software in user-space as compared to kernel-space, the portability of the resulting code, the ready availability and applicability of the OFED user stack, and the near impossibility of getting changes added to the Linux kernel that would be necessary to implement EXS in the kernel.

2 UNH-EXS

The principal difference between the UNH EXS interface and the OpenGroup ES-API [1] is that the UNH EXS interface runs entirely in user space, whereas the ES-API was intended to be integrated into the operating system kernel. Thus the ES-API depends on internal modifications to the existing standard I/O interface to UNIX in order to reuse a large number of existing OS functions for RDMA access. The ES-API does this whenever the number and types of parameters to a socket function are unchanged for use by RDMA. Examples are many of the conventional socket functions – socket, bind, listen, close, getsockname, and getpeername.

Of course, the ES-API also supplies many new functions to be used in place of or in addition to existing socket functions. This is necessary when the number or types of parameters of existing functions have to be changed to accommodate the expanded requirements of asynchronous operation and registered memory in RDMA. Some examples are exs_connect, exs_accept, exs_send, and exs_recv.

Most ES-API functions new and old are based on the use of UNIX *file descriptors* or fds to identify network connections. These fds are used in UNIX as an index into a process-specific table in the kernel that points to all the control information about the file. User-level code has only limited access to this information, and cannot add the types of new information necessary to implement the EXS functionality. The ES-API expects this to be dealt with by modifications to the operating system kernel. But because UNH-EXS is implemented entirely in user space, with no changes to the Linux kernel, it must provide, in addition to all new EXS-API functions, its own equivalent type of file descriptor and its own versions of all standard functions that can be applied to RDMA sockets: exs_socket, exs_bind, exs_listen, exs_close, exs_getsockname, and exs_getpeername.

In addition, UNH-EXS added some new functions to provide additional capabilities, as discussed in section 5.

user space	user application program UNH EXS library OFED user library
kernel space	OFED kernel modules iWARP driver
hardware	iWARP RNIC 10Gig Ethernet

Figure 1. Layering of EXS, OFED, and iWARP

3 Programming in EXS

There are two major differences between a program that uses normal sockets and a program that fully uses the features provided by EXS:

- The need to create and use event queues to control asynchronous operation;
- The need to register and unregister memory used in exs_send and exs_receive operations.

3.1 Asynchronous Operation

The general paradigm for programming in EXS is to use threads and events. All I/O operations are partitioned into two distinct phases: the ”start” phase, and the ” completion” phase. Between these phases the I/O operation proceeds in parallel with thread operation. A start phase begins with one of the new EXS functions, such as exs_connect, exs_accept, exs_send, and exs_recv. One of the parameters to these functions is an event queue which must be created prior to the call by the new exs_qcreate function. When a parallel I/O operation completes, the EXS interface ”posts” an event on the queue given as a parameter in the start phase. A user thread waits on the event queue to receive this event, which will contain the status of the I/O operation.

3.2 Memory Registration

In order to utilize the direct memory-to-memory transfer feature of the RDMA interface hardware, the memory on both ends of a transfer must be ”registered”. This accomplishes several necessary functions:

- It establishes the location and size of a memory area that can be utilized in RDMA transfers;
- It establishes the access permissions allowed to each side in the transfers;

- It ”pins” the virtual memory into real memory so that the RDMA hardware can access the memory without involving the CPU’s paging hardware.

An EXS user can choose to either explicitly or implicitly register the memory utilized in an exs_send or exs_recv.

Explicit registration is accomplished by the use of the exs_mregister function, which returns a ”handle” for the registered area of memory. This handle is then used as a parameter in the exs_send and exs_recv functions. Implicit registration is accomplished by simply passing a special EXS_MHANDLE_UNREGISTERED value as the parameter to exs_send and exs_recv functions. The EXS interface handles implicit registration by dynamically registering the memory at the start of an operation, and unregistering it when the operation completes.

4 Mapping EXS functions onto RDMA operations

All socket communication requires a sender and a receiver, and EXS provides the two corresponding functions exs_send and exs_recv. The underlying RDMA transfer operations are not so simple, due to the requirements of memory registration on both ends of the connection and the memory information needed by RDMA reads and writes. Therefore, one side of the connection has to ”advertise” to the other side the memory it wishes to use in the transfer before the actual transfer can occur, and the EXS interface must match up advertisements from the remote side with requests on the local side before it can initiate an actual RDMA transfer.

The UNH EXS implementation chose to always have the recipient of a data transfer control the RDMA transfer, a decision that was based on our prior experience with iSCSI [9][10][11] and with the use of iSER [12] in conjunction with iSCSI over RDMA [13]. This means that when an application calls the exs_send function, the EXS implementation translates that into an RDMA ”send” operation that sends just a short ”unsolicited” advertisement containing three pieces of descriptive information: the size of the data block to be sent, the starting address of the data block to be sent, and a memory registration handle that gives the receiving side permission to read the block of data directly from the sender’s memory.

When an application calls the exs_recv function, the EXS implementation tries to match it with an already received advertisement from the sending side, and waits (asynchronously) if no such advertisement has yet been received. When an match is found, the receiving side’s EXS implementation issues (asynchronously) an RDMA ”read_request” operation that includes all the information from the advertisement plus the corresponding information from the exs_recv. This allows the RDMA NICs on both sides to cooperate in transferring the data from the memory of the sending machine directly into the memory of the receiving machine without any CPU intervention.

Alternative designs, in which, for example, the sender always controlled the data transfer, were rejected because the chosen scheme maps better onto the asymmetric RDMA instructions provided by iWARP RDMA hardware.

4.1 Flow Control

The advertisements sent by exs_send and the matching done by exs_recv are not seen by the user of EXS, and neither is the internal EXS ”flow control” mechanism described next.

Data transferred from user memory specified by an exs_send directly into user memory specified by an exs_recv clearly does not require any additional buffer memory to be provided by the EXS implementation, the OFED stack or the operating system on either end.

However, an advertisement constitutes additional ”metadata” about a subsequent data transfer, and as such must be stored in small buffers within the sending side EXS implementation, and received into small buffers within the receiving side EXS. Furthermore, advertisements can be sent at any time without prior warning – they are ”unsolicited” – and the receiver must have previously provided buffers to receive these unsolicited messages or the RDMA hardware will cause a fatal error. Therefore the EXS implementation must implement some form of flow control to at least ensure that a sender does not try to send an advertisement unless it knows in advance that a receiver has a buffer ready to accept it.

A credit mechanism is used by the UNH EXS implementation to accomplish this. Each interface maintains two internal credit values: ”send_credits” is the number of advertisements it is allowed to send to the other side, and ”recv_credits” is the number of advertisements is is prepared to receive from the other side. Clearly one side’s send_credits should equal the other side’s recv_credits, and when an EXS connection is first established, the EXS interfaces on both sides negotiate these numbers so they are initialized correctly. Prior to connection establishment, the user application can use the exs_fcntl function to set the values it wishes to use in a negotiation. If not explicitly set by a user, the EXS interface defines some default values to use in the negotiation.

Sending an advertisement requires that the sender have an unused send_credit – if not, it must wait until one becomes available before the actual send can occur. Receiving an advertisement reduces the receiver’s unused recv_credits (as well as its available buffers). Both numbers need to be increased once the actual data transfer finishes, and this is indicated to the receiver’s EXS interface by the OFED stack when the RDMA read_request operation completes. However, the only notification given by either the RDMA hardware or the OFED stack on the sending side is one indicating that the RDMA send for the advertisement was completed. There is no notification on the sender side when the actual RDMA data transfer starts nor when it completes. Therefore, the receiver’s EXS interface must

also send a short unsolicited "acknowledgment' message back to the sender at the completion of a data transfer. This acknowledgment conveys to the sending EXS interface the completion status of the transfer, allowing it to increment its send_credits for this connection and to post the completion event to the sending application.

5 Additional Features of UNH EXS

A number of additional functions have been added to the UNH implementation of EXS to give the user greater control over network communications. These functions also allow us to experiment with parameters of the interface to tune it for various application scenarios.

5.1 Small Packets

As described in section 4 above, data transfers in EXS actually require several RDMA operations: one to send an unsolicited advertisement, one to start an RDMA read, and one to send an unsolicited acknowledgment. (There is also a fourth, hidden RDMA "read_response" operation performed entirely by the RNIC on the sending side in response to the RDMA "read_request" operation performed by the receiver's RNIC.)

When the amount of data in a transfer is small, the total transfer time will be dominated by the overhead time needed to execute all these operations and to transfer the extra metadata. Therefore, the UNH EXS interface allows users to utilize the exs_fcntl function to set a new "small_packet_max_size" parameter on a socket prior to establishing a connection on that socket. Then when the user sends data on that socket using exs_send, the UNH EXS interface will actually send the user's data as an "immediate" part of the unsolicited advertisement itself. Since this data is registered on the user side, the RDMA hardware will still copy it directly from the sending user's memory without any additional copying or buffering.

When the receiving side EXS matches this advertisement with a local exs_recv, it just copies this immediate data into the memory area provided by the user in the exs_recv, avoiding the RDMA read_request and the hidden RDMA read_response operations, but introducing a previously unneeded data copy operation on the receiving side (only). Part of the tuning that can be done by a user is to determine where the small_packet_max_size break-even point is for his or her application and environment. Note that once the small_packet_max_size parameter is set, the user does not have to make any changes to the exs_send or exs_recv calls – the EXS interface deals with sending the immediate data and copying it on the receive side (as would be done for all data when using normal sockets). We stress that this is an optional feature controlled by the user, and demonstrate its effect in section 6.

5.2 In-place Receiving

To avoid even the copy of immediate data just discussed for small packets, the UNH EXS interface provides a new flag value that a user can pass as a parameter to the exs_recv to indicate that the user is prepared to receive small packet data "in-place" rather than in the buffer supplied by that user in the exs_recv call. If this particular exs_recv matches an advertisement that does not contain any immediate data, this new flag value is ignored and the exs_recv and its completion perform as already described. However, if the matched advertisement contains immediate data, then the completion event for this exs_recv will return a pointer to the buffer within the EXS interface that contains the immediate data rather that a pointer to the buffer supplied by the user in the exs_recv because the data is not copied at all by the EXS receiver.

Again use of this optional feature is entirely controlled by the user. It is most appropriate when the user application has set a small_packet_max_size and when his or her application is written to use the data pointer returned in the completion event to access the received data.

Although in-place receiving avoids copying small packet data, it introduces another complexity into a user's program. Because a user is given access to an internal EXS buffer, the EXS interface itself can no longer use that buffer, and can no longer send an acknowledgment to the remote side allowing that buffer to be reused, until the user tells the EXS interface that it is ok to do so. This is done when the user calls the new EXS function exs_recycle_recv_buffer. It is clearly incumbent upon a user to call this function as soon as he or she has finished accessing the data contained in the in-place buffer.

5.3 Utilization Choices in UNH-EXS

In order take full advantage of the features offered by asynchronous operation and RDMA hardware, programs need to be written to utilize cooperating asynchronous threads and memory registration. This necessitates a somewhat different style of programming that that used with "normal" sockets programming. However, to ease the transition, the UNH EXS implementation provides a number of functions that "hide" some of the new details with a corresponding loss of some performance. These functions look very much like the "normal" socket functions, and the idea is to provide a path by which a programmer can modify an existing program written in terms of normal socket functions, get it running over RDAM hardware with very little change, and then slowly modify parts of it to introduce more and more of the EXS features with a corresponding gain in performance. Once a programmer has done this with one program, he or she should henceforth be able to write code directly to utilize the advanced EXS features.

- The first step is to modify a program that uses normal socket functions by simply changing the func-

tion names. The change involves adding the string "exs_synch_" in front of every normal socket function name, so that, for example, "connect" becomes exs_sync_connect". The goal is to allow a direct translation of existing socket-based programs into UNH-EXS, thereby providing a convenient transition path for existing sockets-based software to start utilizing RDMA hardware. With these functions, the user's program sees all I/O as synchronous and all user memory as unregistered, even though RDMA hardware will be used.

- The next step is to modify that first program by adding explicit memory registration. The goal is to gain the performance advantages of using registered memory in RDMA operations, but without the need to engage in asynchronous (i.e., threads) programming. The user will need to utilize the EXS memory registration functions, and a different set of EXS socket functions with extra parameters to handle the registered memory. All I/O will still be seen as synchronous from the user's program, but the RDMA hardware will transfer the user's data directly from one memory to the other without the need for dynamic registration and unregistration.

- The next step is to modify the second program by adding threads programming. The goal is to gain the performance advantages of both registered memory and asynchronous operation. At this point the programmer is utilizing the full functionality defined in the Open Group's ES-API, and sees I/O as synchronous and user memory as registered.

- The next step, which is optional, is to tune the third version of the program by employing small packets and in-place asynchronous access. The goal is to gain a small amount of additional performance by avoiding the extra overhead when transferring small packets and the in-memory copy that accompanies receipt of these transfers.

6 Results

Preliminary results show that programs written in EXS are capable of utilizing a very high proportion of the bandwidth available in a 10GigEthernet link. The graph in Figure 2 shows the results of "blasting" data from a user on one workstation to another user on another workstation. Each workstation contains four 64-bit Intel 2.66 GHz processors with 4 Gbytes of memory and a NetEffect 10 Gigabit/second RNIC. They are connected back-to-back with 10GigEthernet copper cables. User-level throughput in Megabits per second is on the y-axis. The x-axis is the payload size in bytes sent by the application on one machine using one exs_send and received by the application on the other machine using one exs_recv. Both axes are plotted using a logarithmic scale. The upper line in the graph shows the throughput when small packet "immediate" data is in use, the bottom line when it is not. This shows that packets containing upto around 100,000 bytes produce slightly better throughput when sent as "small" packets. Note that the maximum user-level throughput actually achieved is 9.329 Gbps for very large packets. This was achieved using 1500 byte Ethernet frames that allow for a maximum theoretical user payload of 9.363 Gbps once the required headers and CRCs are accounted for.

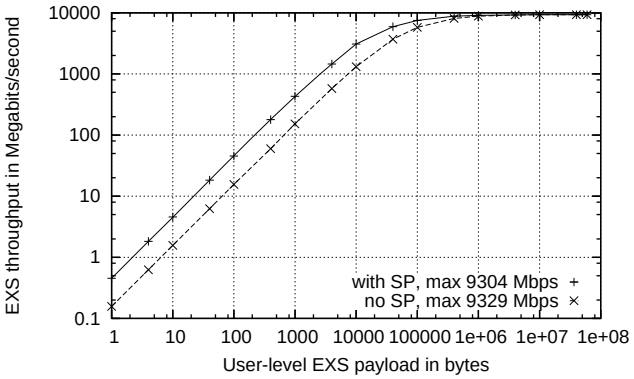

Figure 2. EXS throughput in Mbps

The graph in Figure 3 shows the percentage of CPU time used by the two runs just shown in Figure 2. The differences in CPU utilization are startling. When using the hardware RDMA read_request operation to transfer user-level data, the percent CPU utilization never exceeds 30%. Furthermore, when the size of a transfer reaches 100,000 bytes, the CPU utilization drops to 10%, and after 1,000,000 bytes it is close to 0%. But when the user-level data for a small packet is transferred as part of the advertisement, the percent CPU utilization never drops below 40%, and after the transfer size reaches 1,000,000 bytes, it rapidly shoots up to 90% or more.

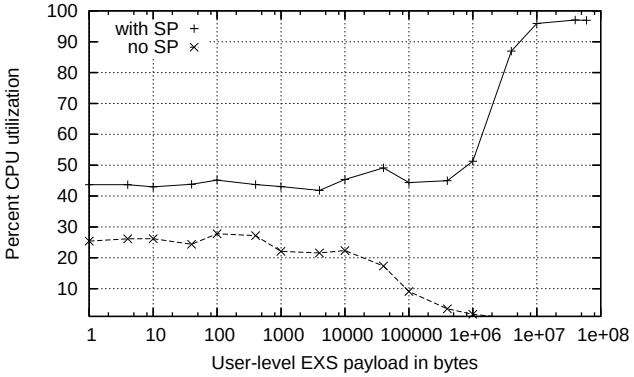

Figure 3. Percentage CPU Utilization

Thus the slight gain in bandwidth utilization for short packets shown in Figure 2 is more than compensated for

by the large cost in terms of extra CPU utilization needed by short packets shown in Figure 3. With EXS, the user chooses which performance he or she prefers by setting the small packet maximum size parameter.

7 Conclusion and Future Work

Due to its similarity with the conventional sockets API, we believe the EXS interface is ideally suited for convenient, high-level access to the benefits of RDMA network hardware. Furthermore, preliminary results shown in this paper demonstrate that application programs using the UNH EXS interface can attain the high bandwidth utilization and low CPU overhead promised by RDMA network hardware.

However, we are just starting to get measurements for application programs of different types run under varying conditions. We are also just starting to tune these applications and to compare the different performance given when different EXS features, such as small packets, are used. Much work needs to be done to gather these measurements, study their implications, and use their results to tune the applications to obtain better performance.

In addition, the OFED stack has been undergoing some recent changes that have yet to be incorporated into our code. For example, one recent change is designed to improve the latency of dynamic memory registration.

Finally there are performance measurements and tuning that need to be done to compare the UNH EXS software running over RNICs from different vendors with both 1500-byte and 9000-byte (jumbo) Ethernet frames. We also need to try running this software over InfiniBand, and comparing the results with iWARP.

References

[1] The Interconnect Software Consortium in association with The Open Group. Extended Sockets API (ES-API) Issue 1.0, January 2005.

[2] The Open Group. http://www.opengroup.org.

[3] A. Romanow, J. Mogul, T. Talpey, and S. Bailey. Remote Direct Memory Access (RDMA) over IP Problem Statement. RFC 4297 (Informational), December 2005.

[4] Open Fabrics Alliance. http://www.openfabrics.org.

[5] OpenFabrics Enterprise Distribution. www.mellanox.com/pdf/products/software/OFED_PB_.1 pdf, 2008.

[6] P. Culley, U. Elzur, R. Recio, S. Bailey, and J. Carrier. Marker PDU Aligned Framing for TCP Specification. RFC 5044 (Standards Track), October 2007.

[7] H. Shah, J. Pinkerton, R. Recio, and P. Culley. Direct Data Placement over Reliable Transports. RFC 5041 (Standards Track), October 2007.

[8] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A Remote Direct Memory Access Protocol Specification. RFC 5040 (Standards Track), October 2007.

[9] R. Russell. iSCSI: Past, Present, Future. In *Proceedings of the 2nd JST CREST Workshop on Advanced Storage Systems*, pages 121–148, December 2005.

[10] Y. Shastry, S. Klotz, and R. Russell. Evaluating the Effect of iSCSI Protocol Parameters on Performance. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN 2005)*, pages 159–166, February 2005.

[11] A. Chadda, A. Palekar, R. Russell, and N. Ganapathy. Design, Implementation, and Performance Analysis of the iSCSI Protocol for SCSI over TCP/IP. In *Internetworking 2003 International Conference*, pages 22–24, June 2003.

[12] M. Ko, M. Chadalapaka, J. Hufferd, U. Elzur, H. Shah, and P. Thaler. Internet Small Computer System Interface (iSCSI) Extensions for Remote Direct Memory Access (RDMA). RFC 5046 (Standards Track), October 2007.

[13] E. Burns and R. Russell. Implementation and Evaluation of iSCSI over RDMA. In *Proceedings of the 5th International Workshop on Storage Network Architecture and Parallel I/O (SNAPI'08)*, September 2008.