An Efficient Method for Stream Semantics over RDMA

Patrick MacArthur, Robert D. Russell
Department of Computer Science
University of New Hampshire
Durham, New Hampshire 03824-3591, USA
{pio3,rdr}@cs.unh.edu

***Abstract*—Most network applications today are written to use TCP/IP via sockets. Remote Direct Memory Access (RDMA) is gaining popularity because its zero-copy, kernel-bypass features provide a high throughput, low latency reliable transport. Unlike TCP, which is a stream-oriented protocol, RDMA is a message-oriented protocol, and the OFA verbs library for writing RDMA application programs is more complex than the TCP sockets interface. UNH EXS is one of several libraries designed to give applications more convenient, high-level access to RDMA features. Recent work has shown that RDMA is viable both in the data center and over distance.**

One potential bottleneck in libraries that use RDMA is the requirement to wait for message advertisements in order to send large zero-copy messages. By sending messages first to an internal, hidden buffer and copying the message later, latency can be reduced at the expense of higher CPU usage at the receiver. This paper presents a communication algorithm that has been implemented in the UNH EXS stream-oriented mode to allow dynamic switching between sending transfers directly to user memory and sending transfers indirectly via an internal, hidden buffer depending on the state of the sender and receiver. Based on preliminary results, we see that this algorithm performs well under a variety of application requirements.

***Keywords*-RDMA; Reliable stream transport; Communication algorithm;**

I. Introduction

Traditionally, most network applications use TCP/IP. However, on high-speed networks, TCP/IP is inefficient due to buffering and operating system involvement in data transfers. Remote Direct Memory Access (RDMA) allows an application developer to transfer data between user-space applications running on different systems while bypassing the kernel as well as the intermediate buffering involved in TCP/IP. RDMA also directly exposes the queues used to request transfer operations, allowing multiple transfers to be simultaneously outstanding from the perspective of the application. The OpenFabrics Alliance (OFA) [1] produces the OpenFabrics Enterprise Distribution (OFED) [2], which consists of a low-level transport- and vendor-independent API called "verbs" for writing RDMA applications.

Due to the complexity of programming for verbs, most application writers write their RDMA programs for an upper-layer protocol such as MPI [3] or rsockets [4]. These protocols attempt to hide the complexity of verbs, but in doing so, must give up some of the benefits of RDMA. In particular, these APIs are missing explicit registration of user I/O memory or asynchronous I/O, both of which are necessary for efficient zero-copy transfers.

The UNH EXS [5], [6], [7] library is an implementation of the Extended Sockets API (ES-API) [8], a specification written by the Open Group [9] that offers a sockets-like interface that explicitly allows the user access to RDMA features such as memory registration and asynchronous socket I/O. UNH EXS offers both stream-oriented (SOCK_STREAM) and message-oriented (SOCK_SEQPACKET) connected sockets, and was not created with the goal of running unmodified sockets applications, which simplifies the design considerably.

RDMA is gaining popularity among a wider audience of application developers as they need higher levels of performance than TCP/IP sockets can offer. However, due to the ubiquity of the byte stream-oriented TCP/IP protocol, most applications expect byte stream semantics. There are subtle differences between stream-oriented and message-oriented transports that make it difficult to simply port a program expecting byte stream semantics to a message-oriented transport. In particular, if a sender tries to send more bytes than the receiver is expecting in a stream-oriented protocol, the sockets library will split the message into chunks so that the receiver will still get the entire message. However, a message-oriented protocol such as UDP or RDMA will only send the part of the message that fits into the receiver's memory area. Thus, a naïve attempt to port a program may result in data loss.

Additionally, recent work has focused on using RDMA over distance, which presents significant challenges [10]. Most importantly, increasing distance means that propagation delay becomes orders of magnitude larger than the sum of the other delay components. Direct data transfers using the RDMA_READ and RDMA_WRITE operations require that the application know the virtual address and remote key of the remote user memory area. This is usually communicated through small message "advertisement" messages sent before big data transfers. This works well when an application reuses I/O memory frequently or when the propagation delay is insignificant. However, over distance, having to wait for an advertisement in order to send a large message is impractical due to the high latency. In this case, it is actually faster for the receiver to copy from a static intermediate buffer than to wait for the advertisements to be satisfied.

The problem we are attempting to solve is to design a thread-safe algorithm that combines the zero-copy benefit of RDMA with the fast send response benefit of TCP-style buffering in a dynamic manner.

In this paper, we develop a byte stream protocol for UNH EXS that combines copies from an intermediate buffer at the receiver with direct data transfers using user advertisements. This is intended to handle two cases: (i) when the sender is ahead of the receiver, the sender will send directly into the receiver's intermediate buffer and the receiver will copy from its intermediate buffer to user memory; (ii) otherwise, the receiver is ahead of the sender, and the sender can match its user data with user memory advertisements sent by the receiver and send the data directly into the receiver's advertised user memory, bypassing the intermediate buffer. We expect that case (i) is most frequent in current sockets applications due to the synchronous nature of the sockets API, and case (ii) will become more prevalent in future applications which will be written with asynchronous and zero-copy I/O in mind. Our goal is to adaptively handle both cases efficiently. Additionally, this will also allow adapting to the current state of the network.

The contributions of this paper are: (i) design of an algorithm to dynamically switch between buffered and zero-copy transfers over RDMA, depending on current conditions; (ii) proof of the correctness and safety of this algorithm; (iii) implementation of this algorithm as part of the UNH EXS library; (iv) demonstration and evaluation of the performance of this implementation.

II. Background

A. Related Work

We first examine other protocols that implement stream semantics over RDMA. The Sockets Direct Protocol (SDP) was included as an annex of the InfiniBand specification [11], and represents the first attempt to implement a TCP-like stream protocol over RDMA. The initial implementation of SDP, referred to as BCopy mode, used buffer copies, similar to BSD sockets. However, other implementers wrote a zero-copy mode for SDP [12], [13]. The first of these, ZCopy mode [12], did not allow multiple simultaneous send requests—the send() call would block until the data was received. This was to prevent the user from modifying the user memory involved in a data transfer while the data transfer was in progress. A later implementation, Asynchronous Zero-Copy SDP (AZ-SDP) mode [13], allowed for multiple simultaneous send requests. AZ-SDP uses the mprotect() call to force a segmentation fault if the user modifies the contents of memory that is part of an ongoing transfer, and either blocks the user application or copies the data if the user changes the contents of the memory area during a transfer operation. However, this protection mechanism introduces an extra kernel call for every data transfer operation, as well as the complexity of correctly resuming the user application after the segmentation fault handler completes. The added complexity is necessary because SDP was intended to run unmodified sockets applications, and these applications assume that they may reuse memory as soon as the sockets library returns control to the application.

The newer rsockets protocol [4] attempts to solve the same problem as SDP, but uses a different protocol. The current goal of rsockets is parity with standard TCP-based sockets, so that the rsend() and rrecv() calls are blocking and perform buffer copies on both the send and receive side on all transfers. However, the roadmap includes adding an asynchronous API and zero-copy functionality as a set of extra functions on top of rsockets.

Another approach for handling streams in RDMA is uStream, described in [14]. The uStream protocol uses threads to allow asynchronous send requests. However, it also uses internal preregistered send and receive buffers, which means that uStream is not truly zero-copy. It also requires two communication channels—a data channel and a control channel. This simplifies the data path but requires extra resources to manage the connections.

In our previous work [15], we studied the performance impact of various RDMA verbs programming techniques. Many of the results from the study are applicable to UNH EXS. In particular, using many simultaneous outstanding operations is essential to achieving good performance for an RDMA connection. Thus, any high-performance sockets replacement for RDMA must be asynchronous and capable of queueing many simultaneous transfer operations. UNH EXS also makes use of the inline functionality of modern RDMA hardware for small messages as well as the RDMA_WRITE_WITH_IMM operation, as recommended in the study.

B. Terminology

The RDMA verbs library provides two sets of transfer operations via asynchronous verbs that post an operation onto a send or receive queue. The SEND and RECV operations provide familiar channel semantics, in which each SEND matches a single RECV. However, a RECV operation must be pending at the receiver before the sender may initiate a SEND operation. This is because RDMA will not perform an intermediate copy, so the HCA at the receiver must know the location in memory to place the data at the time that the data arrives. To satisfy this requirement, each side of an RDMA connection will post n RECV transactions at startup, prior to connection establishment. Each side then gives the other n *send credits*. A sender consumes a credit whenever it performs an action, such as SEND, that would consume a RECV at the receiver. The receiver returns credits by periodic *acknowledgment* (ACK) messages, which indicate that new RECV transactions have been posted.

The other set of transfer operations provide memory semantics. The RDMA_WRITE operation allows a sender to place data directly into a specified location in the receiver's virtual memory space. The receiver application is completely passive and receives no notification that the operation has started or completed. A similar RDMA_READ operation works in the opposite direction, but is not used in our solution.

UNH EXS uses the RDMA_WRITE_WITH_IMM transfer operation, which pushes the contents of a memory area directly from user virtual memory at the sender to user virtual memory at the receiver and then notifies the receiver (by consuming a previously posted RECV transaction). This operation exists in InfiniBand, RoCE, and newer versions of iWARP. The operation can be simulated on older iWARP hardware by following an RDMA_WRITE with a small SEND. Hereafter, we will refer to this RDMA operation simply as *WWI*.

UNH EXS provides an API in which almost all calls are truly asynchronous. When the application requests an EXS operation, the EXS library places a request onto a queue and control immediately returns to the caller. When the operation completes, the EXS library places an *event* onto an event queue previously created by the user. The user then polls the event queue for completions, and retrieves the status of the operation.

We consider two different types of WWI operations in the context of the UNH EXS API. Specifically, this deals with the ownership of the receiving destination memory area in a WWI operation. A *direct transfer* transfers data directly to memory registered and owned by the receiving application. An *indirect transfer* transfers data to an intermediate buffer owned by the intermediate receiving API, which is later copied to the memory owned by the receiving application.

Also, in this paper, the term *stream* always refers to the byte stream used by a stream-oriented network protocol such as TCP. In a stream-oriented protocol, each transfer has a *sequence number* marking its position in the byte stream. That is, the sequence number of transfer x is the number of data bytes sent on the connection prior to the start of transfer x.

C. Modes of Operation

UNH EXS connections currently operate in one of two modes, message-oriented and stream-oriented. The application requests message-oriented mode using the SOCK_SEQPACKET socket type in the exs_socket() call. Although not the focus of this paper, a summary of the implementation of this mode will help the reader understand the stream-oriented mode. The RDMA protocol for message-oriented connections is simple. When the application calls exs_recv(), the EXS library at the receiver sends an advertisement (ADVERT) to the EXS library at the sender with the virtual memory address, length, and RDMA remote key of the receiver's memory area. When the user at the other end of the connection calls exs_send() and an ADVERT has reached the EXS library at that end, the sender then posts an WWI request with the data. The sender's host channel adapter (HCA) then transfers the data directly into the receiving user's memory with no intermediate copies. In low-latency networks, this allows for very high throughput and little additional latency.

An application requests stream-oriented mode by using the SOCK_STREAM socket type in the exs_socket() call. In the initial release of UNH EXS, this mode only used direct transfers. When using direct transfers, the difference between this mode and the message-oriented mode is that if an exs_send() request is larger than the advertised exs_recv() memory area x, the EXS library at the sender will split the message between memory area x and subsequent exs_recv() calls. The stream-oriented mode also supports the MSG_WAITALL flag at the receiver, indicating that the receiver should wait until the user memory is full before signaling completion to the user. UNH EXS implements this by adding a flag to the ADVERT, such that if the length y of the exs_send() request is smaller than the advertised exs_recv() memory area x, the sender will keep the ADVERT at the head of its queue until the sender has transferred all x bytes to the receiver via subsequent exs_send() calls.

III. Proposed Solution

Waiting for ADVERTs can be inefficient, especially if the sender is ready to send before any ADVERTs have arrived. Thus, having a way to quickly send messages without waiting for ADVERTs is essential for good performance. Therefore, we added a "hidden" (in the EXS library) intermediate receive-side buffer for stream-oriented connections. The ideal solution will send messages "indirectly" to this intermediate buffer only when absolutely necessary, in a manner that is transparent to the user. In this paper, we describe the methods used to dynamically choose between direct and indirect transfers in the same connection, based on whether the sender or receiver is currently "ahead."

During an indirect transfer, the EXS library at the sender sends data to the intermediate receive buffer, and the library at the receiver then copies from this intermediate buffer to user memory. The intermediate buffer is circular, and the sender keeps a pointer to the next position in the intermediate buffer to place data, while the receiver keeps a pointer to the next position in the intermediate buffer to remove data. Both sides keep track of the number of bytes currently stored in the intermediate buffer, and the receiver periodically sends acknowledgment packets as it removes data from the intermediate buffer. The intermediate buffer reduces sender-side latency at the cost of higher CPU usage on the receiver due to the extra copies.

Because using only indirect transfers throws away one of the primary benefits of using RDMA, which is zero-copy semantics, we designed an algorithm to dynamically choose between using the stream buffer or sending direct transfers if an ADVERT reaches the sender before it is ready to send.

The motivations behind our approach are as follows. We require that the EXS library deliver user data from the sender to the receiver in order and with no errors. We also wish to maintain as little extra state as possible. In particular, we would like the sender and receiver to be as independent as possible, and to make as few assumptions about the other side as possible. This makes it easier to modify implementation details at a later time. Finally, we would like the sender to use direct transfers as frequently as possible; otherwise, the added complexity produces no performance benefit.

Handling both direct and indirect transfers in the same connection for a stream-oriented protocol that allows multiple simultaneous asynchronous requests is nontrivial. The biggest

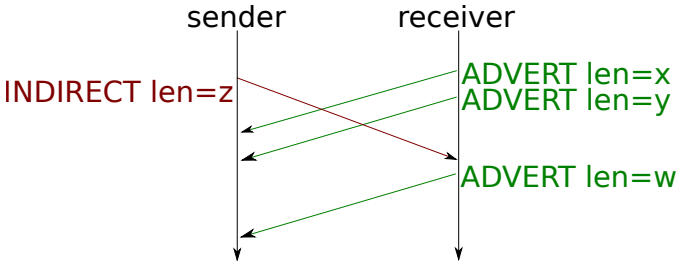


Fig. 1. An indirect transfer crosses with multiple ADVERTs from the receiver. The next WWI in the sequence could match the second or third ADVERT, depending on how the indirect transfer gets matched at the receiver. This illustrates the challenge of figuring out which ADVERT comes next in sequence after an indirect transfer.

TABLE I
VARIABLES USED IN THIS PAPER AND THEIR ASSOCIATED MEANINGS.

Variable	Meaning
b_r	Full byte count of the intermediate buffer at the receiver
b_s	Intermediate buffer free byte count at the sender
k_a	Count of outstanding adverts from prior phase at receiver
k_b	Count of outstanding exs_recv() with no advert at receiver
l_c	Length copied out of the intermediate buffer
l_r	Length given in a user's exs_recv() call
l_w	Length of a WWI operation
P_A	Phase number contained in ADVERT
P_r	Phase number at receiver
P_s	Phase number at sender
q_A	Queue of received ADVERTs at sender
S_A	Sequence number contained in ADVERT
S_r	Sequence number at receiver
S'_r	Next-expected sequence number to use in next advert
S_s	Sequence number at sender

reason for this is the nature of byte streams. If the user calls exs_recv(*fd*, *buf*, *len*, ...) and there is data to receive, UNH EXS will eventually place n bytes into *buf*, where $1 \leq n \leq$ *len*. That is, the receiver does not know the exact number of bytes that will be received when the user requests an exs_recv() operation or when the EXS library sends an ADVERT. Fig. 1 illustrates an issue when the sender uses indirect transfers via the remote receive buffer.

Both the sender and the receiver keep track of their current position in the stream, hereafter referred to as the *sequence number*. Each ADVERT includes the expected sequence number of the corresponding exs_recv(). For the first ADVERT in a sequence, this should match the actual expected sequence number. However, in subsequent ADVERTs in the sequence, this expected sequence number will be an estimate, which is 1 more than the previous ADVERT. As the receiver receives data, the receiver updates the estimated next-expected sequence number used for future ADVERTs to reflect the actual amount of data transferred.

The sender and receiver also keep track of a *phase number*, where each phase represents a sequence of consecutive direct or indirect transfers. This phase mechanism provides a "logical time" similar to that provided by Lamport's logical clocks [16], and orders ADVERTs with respect to sequences of indirect transfers. Initially, the sender and receiver start in

```
1:  while ¬EMPTY(q_A) do
2:      A ← HEAD(q_A)
3:      if PHASE_IS_INDIRECT(P_s) ∧ (P_A < P_s ∨ S_A < S_s)
    then
4:          if P_s < P_A then
5:              P_s ← NEXT_PHASE(P_A)
6:          end if
7:          throw away ADVERT A
8:      else
9:          if PHASE_IS_INDIRECT(P_s) then
10:             P_s ← P_A                    ▷ P_s is now direct
11:         end if
12:         S_s ← S_s + l_w
13:         send direct transfer
14:         return
15:     end if
16: end while
17: if ¬FULL(b_s) then
18:     if PHASE_IS_DIRECT(P_s) then
19:         P_s ← NEXT_PHASE(P_s)        ▷ P_s is now indirect
20:     end if
21:     S_s ← S_s + l_w
22:     b_s ← b_s − l_w
23:     send indirect transfer
24:     return
25: end if
```

Fig. 2. This is the algorithm to match an exs_send() request to an ADVERT A or the intermediate stream buffer b. Variable definitions are in Table I.

```
1:  if b_r > 0 ∨ k_a > 0 ∨ k_b > 0 then
2:      do not send ADVERT
3:      return
4:  end if
5:  if PHASE_IS_INDIRECT(P_r) then
6:      P_r ← NEXT_PHASE(P_r)            ▷ P_r is now direct
7:  end if
8:  P_A ← P_r
9:  S_A ← S'_r
10: if MSG_WAITALL is set then
11:     S'_r ← S'_r + l_r
12: else
13:     S'_r ← S'_r + 1
14: end if
15: send ADVERT
```

Fig. 3. This is the algorithm used by the receiver when sending an ADVERT for a user exs_recv() buffer. Variable definitions are in Table I.

1: **if** incoming transfer is direct **then**
2: $S_r \leftarrow S_r + l_w$
3: **if** MSG_WAITALL was *not* set **then**
4: $S'_r \leftarrow S'_r + l_w - 1$
5: **end if**
6: do normal processing
7: **else** ▷ incoming transfer is indirect
8: **if** PHASE_IS_DIRECT(P_r) **then**
9: $P_r \leftarrow$ NEXT_PHASE(P_r)
10: **end if**
11: do normal processing
12: **end if**

Fig. 4. This is the algorithm used by the receiver when a transfer arrives. Variable definitions are in Table I.

1: copy data from stream buffer
2: send ACK to sender notifying of freed space
3: $b_r \leftarrow b_r - l_c$
4: $S_r \leftarrow S_r + l_c$
5: **if** advert sent and MSG_WAITALL was *not* set **then**
6: $S'_r \leftarrow S'_r + l_c - 1$
7: **end if**

Fig. 5. This is the algorithm used by the receiver to copy data out of the intermediate buffer to user memory. The precondition for this algorithm is that $l_c \leq b_r$. Variable definitions are in Table I.

phase 0. If the sender is able to use a direct transfer, then the phase remains unchanged. However, if the sender uses an indirect transfer, then it increments its phase to 1. The receiver increments its phase to 1 when it receives the indirect transfer. Once the receiver empties its intermediate buffer, it will attempt to send ADVERTs again, and increment its phase to 2. Advertisements from the receiver include the current sequence number and phase. If both the sequence number and phase in the ADVERT are at least as large as those at the sender, then the sender knows that the receiver has caught up and it can safely use the ADVERT. Otherwise, the sender knows that the ADVERT is "stale" and throws it away. The phase number will be even during a sequence of direct transfers, and odd during a sequence of indirect transfers.

We present the algorithms in more detail in Fig. 2, 3, 4, and 5. The PHASE_IS_DIRECT function returns true if the phase number is even; the PHASE_IS_INDIRECT function returns true if the phase number is odd; and the NEXT_PHASE function returns $p + 1$ for input p.

When the receiver receives an indirect transfer, the receiver knows that any unsatisfied ADVERTs in the current phase will not be satisfied. The receiver will also not send any new ADVERTs until it completely empties its intermediate buffer. This means that the sender will continue to send indirect transfers until it fills the buffer or runs out of data to send in its current burst of data. Once the receiver empties the buffer completely, it will again start to send ADVERTs for direct transfers, but only after resynchronizing with the sender. In order to do this, the receiver must ensure that the sequence number of the next ADVERT matches what the sender expects. This will not be the case if there are any outstanding ADVERTs from a previous phase, since the sequence numbers of both those ADVERTs and future ADVERTs were estimates, as shown

TABLE II
LEGEND FOR THE DIAGRAMS IN FIG. 6, 7, AND 8.

Event	Format
Send ADVERT	A*id_number*;*seqno*;*length*;*phase*
Send direct transfer	D;*matching_advert*;*seqno*;*length*;*phase*
Send indirect transfer	I;(*invalidated_adverts*);*seqno*;*length*;*phase*
Recv Any	*phase*

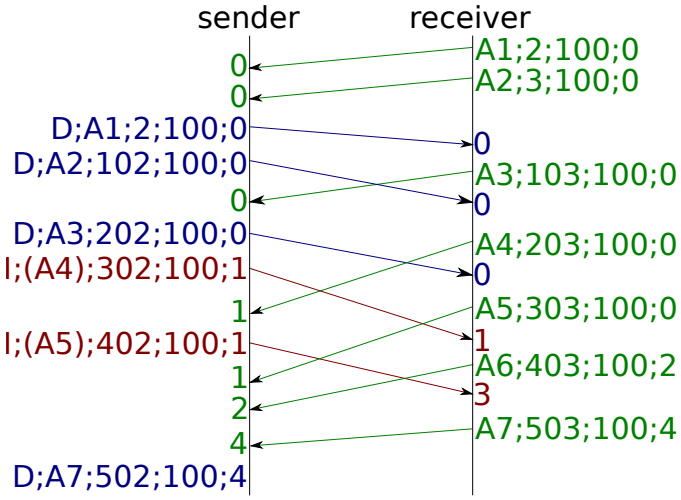


Fig. 6. This illustrates a potential sequence number gap if the receiver continues to send ADVERTs after it receives an indirect transfer. In this case, ADVERT A7 will be incorrectly matched (since it contains a higher phase number than that at the sender and the next-expected sequence number), causing the sender to perform a direct transfer into the wrong memory location. The notation in this figure may be interpreted using Table II.

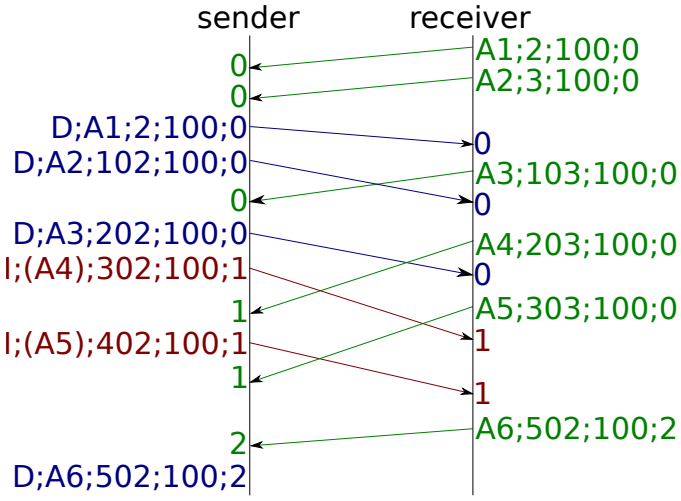


Fig. 7. This illustrates the fix for the issue shown in Fig. 6. The receiver must hold off on sending ADVERTs until all prior ADVERTs have been consumed with data from indirect transfers. The notation in this figure may be interpreted using Table II.

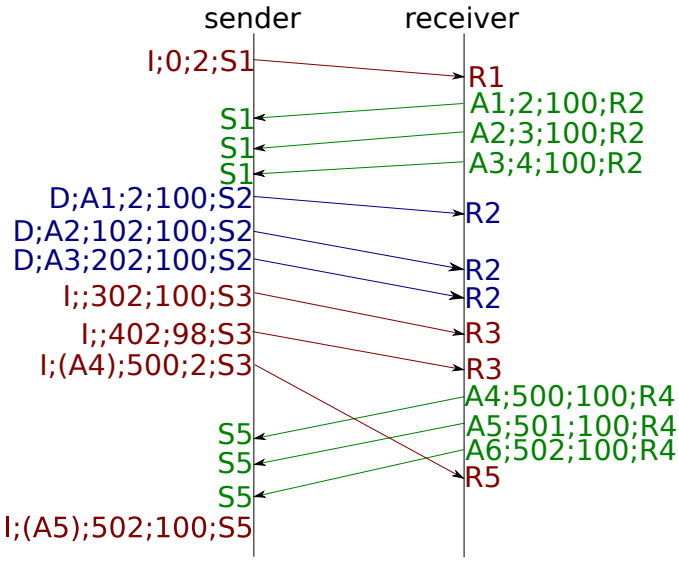


Fig. 8. This illustrates the reason that the sender must increment its phase number whenever an ADVERT arrives with a higher phase number. If the phase number is not incremented, then ADVERT A6 will be incorrectly matched (since its phase and sequence numbers would be higher than that at the sender), causing the sender to perform a direct transfer into the wrong memory location. The notation in this figure may be interpreted using Table II.

in Fig. 6. To prevent an incorrect match, the receiver will not send any more ADVERTs until all exs_recv()s from the previous phase have been satisfied. The result is shown in Fig. 7. Consequently, the receiver will not send an ADVERT if there is a pending receive with no corresponding ADVERT, a situation possible only if we had stopped sending ADVERTs because we received an indirect transfer.

At the sender, a pending exs_send() will only match an ADVERT if its sequence number S_A matches exactly with the current send sequence number, and the phase P_A is greater than or equal to the sender's current phase. If either do not match, then the sender discards the received ADVERT. The sender then increments the phase to at least $P_A + 1$ if $P_A > P_s$. This avoids a scenario illustrated in Fig. 8, in which a future ADVERT appears to match because its sequence number happens to match S_s, by effectively dropping the entire sequence of ADVERTs. As a result, when an ADVERT is dropped or no ADVERTs are available, the send data will be written indirectly into the remote intermediate buffer. The corresponding exs_recv() will thus be satisfied from the intermediate receive buffer. Once the receiver empties its buffer, it makes a new attempt to synchronize with the sender.

IV. Evaluation

A. Correctness

We present a proof of the correctness of the stream ADVERT handling algorithms.

Lemma 1. *Every ADVERT from the receiver to the sender will contain a direct phase number. (See Fig. 3)*

Proof: We know that the receiver's phase P_{r0} prior to starting the algorithm must be either direct or indirect.

Consider the case where P_{r0} is direct. Due to the check in line 5, phase P_{r0} is placed unchanged into the ADVERT at line 8. Thus the ADVERT contains direct phase number P_A.

Otherwise, P_{r0} is indirect, and it is advanced at line 6 before being placed into the ADVERT at line 8. We know that the next-phase for an indirect phase is direct, so the phase number P_{A0} placed into the ADVERT is again direct. ■

Lemma 2. *If the receiver sends an ADVERT with phase number P_{A0}, all future ADVERTs will contain phase number P_{A0} until the receiver receives an indirect transfer.*

Proof: From Lemma 1 we know that P_{A0} is direct. Since for all ADVERTs P_A is set in line 8 in Fig. 3, we know that $P_r = P_{A0}$ and that P_r is direct. We also know that the only way for the receiver to advance P_r when P_r is direct is to receive an indirect transfer, due to the check at line 8 in Fig. 4. Thus, P_r cannot have advanced since ADVERT A_0 in the absence of an indirect transfer. ■

Lemma 3. *At the sender side, if the sender's phase is direct, then the most recent message sent by the sender (if any) was a direct WWI.*

Proof: If the sender has not yet sent a message, then the sender's phase is 0 and trivially direct.

Otherwise, the last transfer from the sender was either direct or indirect. Suppose, for purposes of contradiction, that the most recent message sent by the sender was an indirect transfer, but that the sender's current phase P_s is direct. Let P_{s0} be the sender's phase prior to sending the indirect transfer (at line 17 of Fig. 2). If P_{s0} were direct, then the phase P_s would be advanced at line 19, due to the check at line 18, and would thus be indirect. Otherwise, if P_{s0} were indirect, then the phase P_s would be unchanged after the check at line 18. In either case, the new (and current) phase P_s after the indirect transfer in line 23 would be indirect, but that contradicts our previous assumption that P_s was direct. Thus, the most recent message sent by the sender must have been a direct transfer. ■

Lemma 4. *If the sender's current phase P_s is direct and the sender receives an ADVERT A with phase P_A, then $P_s = P_A$.*

Proof: By Lemma 1 we know that the ADVERT contains a direct phase number. By Lemma 3 we know that when the current phase P_s is direct, the last message sent by the sender (if any) was a direct transfer.

We now examine the receiver's phase P_r prior to sending the ADVERT. Let us assume that P_r is indirect at line 5 in Fig. 3. If P_r were indirect, then the last transfer that the receiver received would have been indirect, and thus the sender's phase would have to be indirect, but we have assumed that the sender's phase P_s is direct. Contradiction.

Thus, P_r must be direct, and the check in line 5 prevents the phase number from being advanced in line 6 prior to sending the ADVERT. Consider first the case where P_r is 0. Thus, P_s

must be 0, since if it were non-zero, the sender would have sent at least one indirect transfer, and we know that the sender did not send an indirect transfer. Thus, trivially $P_s = P_A$.

Next consider the case where P_r is not zero. We know that the sender's phase P_s is not zero because the only way for the receiver's phase to become nonzero is to receive an indirect transfer (thus increasing the receiver's phase to one, which is indirect), which would have caused the sender's phase to increase. Thus the sender's phase at some previous time must have been indirect. Since the P_s is now direct by our initial assumption, the sender must have sent at least one direct send, because the only way to transition from an indirect phase to a direct phase is via line 10 of Fig. 2. Let A_0 be the ADVERT that caused this transition. When the sender sent a direct transfer in line 13 in response to ADVERT A_0, it first advanced its initial phase P_{s0} to P_{A_0}. Due to Lemma 2, P_r cannot have advanced since ADVERT A_0 in the absence of an indirect transfer. Thus, all ADVERTs sent since A_0 will also have the same phase $P_A = P_{A_0}$. We also know that P_s cannot have advanced since the previous direct transfer, because the only way for the sender to advance P_s when it is direct is via the check for direct phase in line 18 in Fig. 2. Thus, $P_s = P_A$. ■

Theorem. ***(Safety Property.)*** *If the receiver receives a direct transfer, then the direct transfer matches the sequence number S_D and the phase number P_D of the ADVERT D sent by the receive transaction block at the head of the receiver's exs_recv() queue. That is, the sender will never accept a stale ADVERT, and all data will arrive in the order that the sender sent it with no data loss.*

Proof: The state of the sender at line 8 of Fig. 2 can be either direct or indirect.

(a) If the sender's phase is direct, then by Lemma 4, the phase number of the ADVERT matches the current phase at the sender. Thus, the sender is in the middle of a sequence of direct sends, so the advert A at the sender is the same as the advert D at the head of the receiver's exs_recv() queue.

(b) Consider the case when P_s is indirect. Thus, we know that the most recent transfer I sent by the sender was indirect, since the only way that the sender's phase can become indirect is via line 19 of Fig. 2, due to the check at line 18. We also know that $P_A \geq P_s$ and $S_A \geq S_s$, since we are in the "else" branch of the check in line 3. Let us assume, for purposes of contradiction, that ADVERT A at line 8 of Fig. 2 is not the same as ADVERT D at the head of the exs_recv() queue at the receiver. This implies that at least one of the following must be true: $P_A < P_D$, $P_A > P_D$, $S_A < S_D$, or $S_A > S_D$. We will establish the contradiction by showing that none of these can be true.

(b1) First consider the case where $P_A < P_D$. The difference between P_D and P_A must have been caused by first advancing P_r from direct to indirect in line 9 in Fig. 4 when the indirect transfer I was received, and then from indirect to direct in line 6 in Fig. 3 when advert D was sent. Thus, the receiver received an indirect transfer after the receiver sent ADVERT A, when P_r was direct. Then, the corresponding exs_recv() operation for ADVERT A was consumed by the algorithm in Fig. 5, so $S_s > S_A$ since it was advanced by line 21 in Fig. 2. But this contradicts our earlier knowledge that $S_A \geq S_s$. Therefore $P_A \not< P_D$.

(b2) Next consider the case where $P_A > P_D$. Because the phase number at the sender and receiver is monotonically nondecreasing, we know that ADVERT D comes in sequence before ADVERT A. We also know that at the time ADVERT A was sent, ADVERT D was not in the exs_recv() queue because $P_D < P_A$, due to the check in line 1 in Fig. 3. This contradicts our earlier assumption that D was at the head of the receive queue. Thus $P_A \not> P_D$.

Since we have shown that both $P_A \not< P_D$ and $P_A \not> P_D$ are true, we have $P_A = P_D$.

(b3) Next consider the case where $P_A = P_D$ but $S_A > S_D$. Because the sequence numbers in each sequence of ADVERTs are monotonically increasing due to the conditional starting at line 10 of Fig. 3, $S_A > S_D$ implies that ADVERT D comes in sequence before ADVERT A. Therefore, the sender received ADVERT D before ADVERT A. Thus, since the sender just removed A from its ADVERT queue q_A, ADVERT D must have been previously removed from its ADVERT queue. Since the last transfer sent by the sender was indirect, the sender must have rejected ADVERT D at line 7 of Fig. 2. Therefore, $S_s > S_A$ due to lines 4-5. But this contradicts our earlier knowledge that $S_A \geq S_s$. Thus $S_A \not> S_D$.

(b4) Finally consider the case where $P_A = P_D$ but $S_A < S_D$. Because the sequence numbers in each sequence of ADVERTs are monotonically increasing due to the conditional starting at line 10 of Fig. 3, we know that ADVERT D comes in sequence after ADVERT A. Therefore, the sender received ADVERT D after ADVERT A. But $S_A < S_D$ implies that ADVERT A is no longer in the receiver's exs_recv() queue, since ADVERT D is at the head of the receiver's exs_recv() queue. Thus the exs_recv() operation corresponding to ADVERT A must have been consumed by an indirect transfer, since the sender's phase is indirect and the only way that the sender's phase could become indirect is via line 19 of Fig. 2. Thus, the number of bytes copied l_c was large enough to advance S_r beyond S_A in line 4 of Fig. 5. We know that $l_c \leq l_w$ because the data could only arrive into the stream buffer via a transfer from the sender. Thus, the length l_w of the transfer must also have been large enough such that the sender's current sequence number S_s was advanced to greater than S_A by line 21 in Fig. 2. But this contradicts our earlier knowledge that $S_A \geq S_s$. Thus $S_A \not< S_D$.

Since we have derived a contradiction from all of the possible cases b1 through b4, we must conclude that our initial assumption that the ADVERT A at line 8 of Fig. 2 is not the same as the ADVERT D at the head of the exs_recv() queue at the receiver is false. ■

B. Performance Study

We present a performance study of the dynamic stream ADVERT handling algorithm as implemented in the UNH

EXS library. This study uses a blast tool, written to utilize UNH EXS, which sends messages as quickly as possible from the client to the server. This is meant to model the traffic generated by a large file transfer. The tool outputs the average throughput, time per message, and CPU usage on each side. Additionally, UNH EXS itself keeps statistics on the number of indirect vs. direct transfers. Unless otherwise specified, we ran each test 10 times and took the average and 95% confidence interval for each measurement. All tests use event notification for retrieving RDMA completion events, as most messages in this study are large enough that there is little advantage to busy polling [15]. We define throughput as the total number of user bytes sent divided by the time elapsed between the start of the first transfer and the end of the last transfer, as shown in Equation 1.

$$throughput = \frac{total\_user\_bytes\_sent}{end\_time - start\_time} \quad (1)$$

In our experiments, we compare against two baseline protocols. The *direct-only* protocol forces the sender to always wait for an ADVERT from the receiver before sending, so that it will never send to the intermediate buffer. In the *indirect-only* protocol, the receiver does not send ADVERTs at all, forcing the sender to send all messages indirectly. Both of these protocols will correctly transfer all data; they just force a particular mode of operation for the purposes of performance comparison. Our blast tool activates these protocols by passing specific flags to the UNH EXS library.

There were two parts to the study. The first part tests our algorithm over two FDR InfiniBand channel adapters connected through an FDR switch. The second part tests the effect of our algorithm over distance. For this we used an Anue network emulator to introduce a fixed delay on a 10Gb Ethernet connection between two RoCE NICs. We used 10Gb Ethernet because our network emulator did not support InfiniBand or higher Ethernet speeds.

1) FDR InfiniBand: We performed our first series of tests on two identical nodes with Mellanox Connect-X 3 FDR InfiniBand HCAs, connected through a Mellanox SX6036 FDR InfiniBand switch. The nodes contain Intel Xeon E5-2609 CPUs running at 2.40GHz, 64 gigabytes of RAM, and a PCIe Generation 3 bus. The nodes run Scientific Linux 6.3 with OFED 3.5. The average one-way latency between the two nodes, as measured by the `ib_write_lat` tool for 64-byte messages is 1.76 microseconds.

In Fig. 9a and 9b we can see the effect of the number of simultaneously outstanding operations on throughput for each stream protocol with messages of random size chosen from an exponential distribution. For the indirect-only protocol, we see that the throughput is between 20 and 27 Gbps, and for the direct-only protocol, the throughput is between 35 and 44 Gbps. The performance for the indirect protocol is always substantially lower due to the required buffer copies, which are far slower than the network. In tests on QDR InfiniBand, the indirect protocol compares much more favorably in terms of throughput, since the maximum possible throughput of QDR InfiniBand is not dramatically higher than the memory copy throughput.

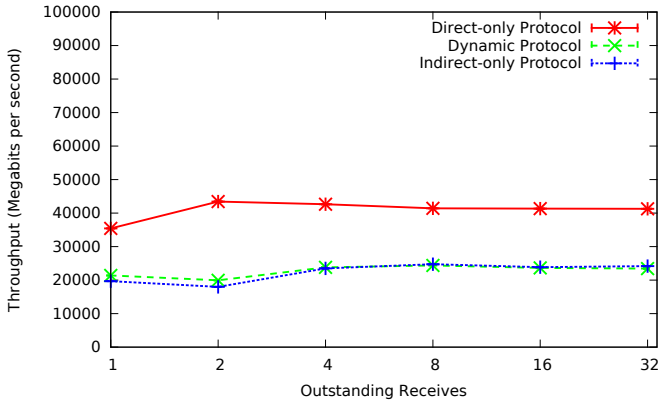
TABLE III
AVERAGE NUMBER OF MODE SWITCHES FOR GIVEN NUMBER OF OUTSTANDING OPERATIONS AT SENDER AND RECEIVER.

Outstanding Operations		Mode Switch	Direct to Total
Receiver	Sender	Count	Transfer Ratio
1	1	93 ± 86	0.0011 ± 0.001
2	2	6529 ± 346	0.0632 ± 0.04
4	4	1 in all cases	< 0.001
8	8	1 in all cases	< 0.001
16	16	1 in all cases	< 0.001
32	32	1 in all cases	< 0.001
2	1	20 ± 6	0.99949 ± 0.0002
4	2	1 in all cases	0.191 ± 0.1143
8	4	0.1 ± 0.12	0.914 ± 0.169
16	8	0 in all cases	1 in all cases
32	16	0 in all cases	1 in all cases

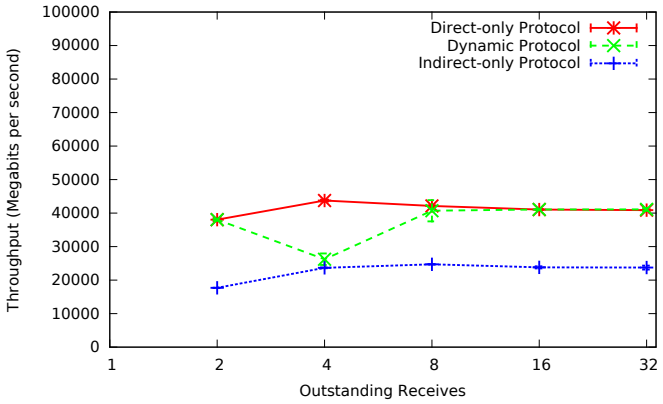
The dynamic protocol follows the behavior of the indirect or direct protocol depending on the number of outstanding operations. Fig. 9b shows that the throughput is approximately the same as the direct-only protocol if the number of outstanding receive operations is twice as large as the number of outstanding send operations. In these cases, the receiver is always able to send ADVERTs before the sender is ready to send, so the sender always sends direct transfers. Fig. 9a shows that when the number of outstanding send and receive operations are equal for the dynamic protocol, the throughput drops to the level of the indirect-only protocol. This is because in this case, the sender is always able to send indirect transfers before the receiver gets a chance to send any ADVERTs.

There is a single anomaly in Fig. 9b when the number of outstanding operations is 4 at the receiver and 2 at the sender. To explain this anomaly, we look at Table III, which shows for each case in Fig. 9a and Fig. 9b the average number of times that the dynamic protocol switches between direct and indirect transfer modes and the average ratio of direct transfers to total transfers along with the 95% confidence interval. In the case with the anomaly, there is exactly one mode switch. Since UNH EXS starts out in a direct phase, we can conclude that the sender sends about 20% of its messages as direct transfers and then sends the remaining transfers as indirect. In all other cases in Fig. 9b, the receiver is able to keep up with the sender for the entire duration of the run. This indicates that 2 "extra" outstanding operations at the receiver is insufficient to ensure that there is always an ADVERT ready at the sender, since the PCIe generation 3 bus goes through the CPU caches when accessing or writing memory [17], making the data transfers faster than the receiver can send more ADVERTs.

We also show the CPU usage on the receiving side in Fig. 10a and 10b. For the indirect-only protocol, CPU usage approaches 100% as the number of simultaneously outstanding operations increases because the copies from the intermediate buffer take a significant amount of CPU time and increasing the number of outstanding operations makes more efficient use of the fabric, increasing the frequency at which data is added to the stream buffer. For the direct-only protocol, the CPU

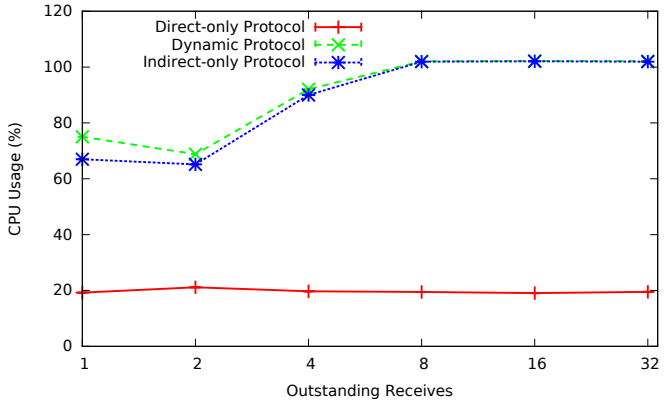


(a) Number of outstanding operations at the sender and receiver are equal.

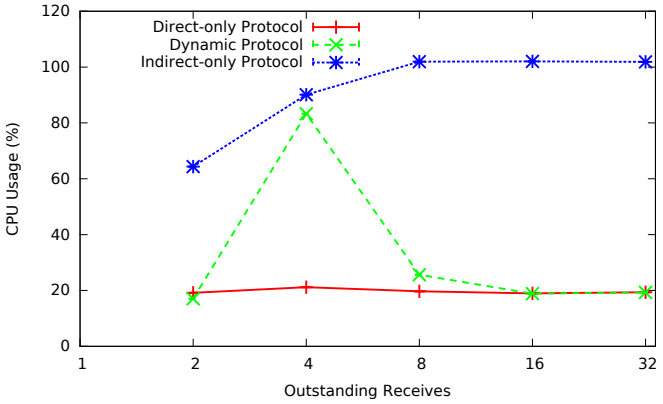


(b) Number of outstanding operations at the sender is half that at the receiver.

Fig. 9. Throughput vs. number of simultaneous outstanding operations at the sender and receiver. Message sizes were selected at random from an exponential distribution with $\lambda = 1048576$ and a maximum message size of 4 MiB.



(a) Number of outstanding operations at the sender and receiver are equal.



(b) Number of outstanding operations at the sender is half that at the receiver.

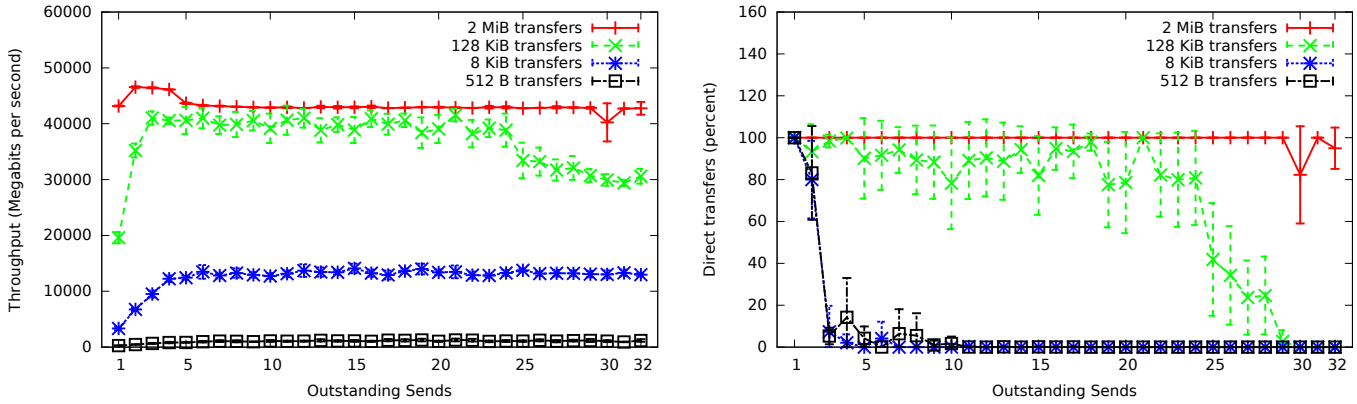
Fig. 10. CPU usage at receiver vs. number of simultaneous outstanding operations at the sender and receiver. Message sizes were selected at random from an exponential distribution with $\lambda = 1048576$ and a maximum message size of 4 MiB.

usage is always much lower because of the zero-copy nature of RDMA. From this plot, we can see that in cases where the dynamic protocol is able to use direct transfers, the dynamic protocol adds little CPU overhead.

We next hold the number of outstanding receive operations constant at 32 and vary the number of outstanding send operations from 1 to 32. The throughput is shown in Fig. 11a. Here, we can see that the throughput increases with message size, as expected. We also see that the throughput has little variation as the number of outstanding send operations increases above 5, except when the message size is 128 KiB. To explain this variation, we also show the ratio of direct transfers to total transfers in Fig. 11b. Here, we see that although the variation is low for most message sizes, the variation in the number of direct transfers is high when the message size is 128 KiB. Thus, the number of direct transfers has a significant effect on throughput for the dynamic protocol, again due to the expense of copying large messages.

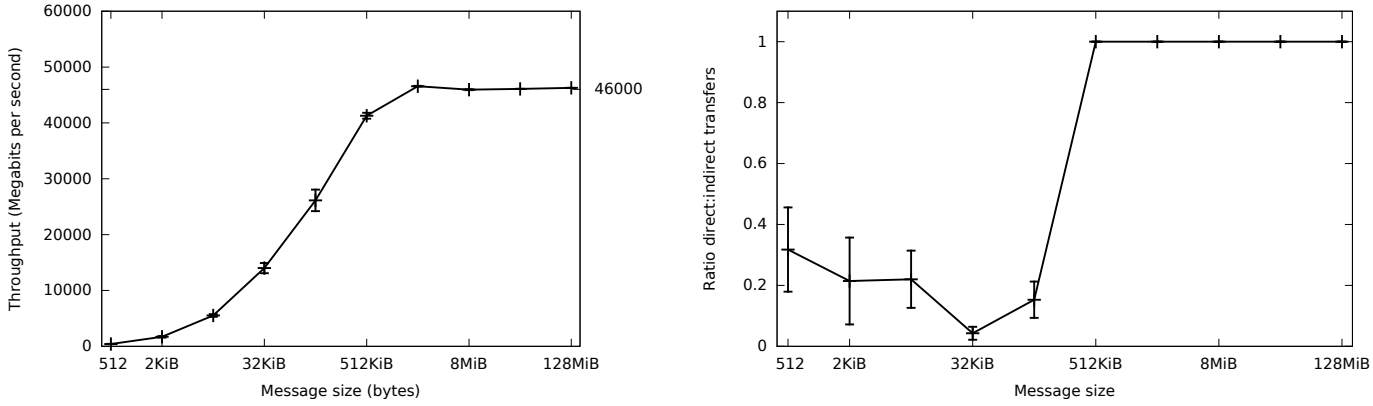
We next examined the effect of message size on the throughput and ratio of direct transfers. Fig. 12a shows that throughput generally increases with message size. However, there is a 46.5 Gbps peak at the 2 mebibyte message size, with slightly lower throughput for higher message sizes. This may be due to the effects of caching on the InfiniBand hardware. The effect on the ratio of direct transfers is shown in Fig. 12b. The ratio of direct sends to total sends decreases with message size until the message size reaches about 32 kibibytes, at which point the ratio begins to increase again. With 512KiB or higher message sizes, the sender is able to use all direct sends. This is due to the transmission delay for each message, which increases as the message size increases. Once the amount of data in transit becomes large enough, the transmission delay for the data transfers becomes larger than the total delay for the ADVERTs flowing in the opposite direction. Therefore, the receiver's ADVERT for the next exs_recv() operation always arrives before the sender is ready to perform its next send operation.

2) 10Gbps RoCE with Anue Network Emulator: We performed testing using an Anue network emulator to introduce delay between nodes. The two nodes under test used Mellanox



(a) Throughput (b) Ratio of direct to total transfers

Fig. 11. Effect of changing the number of simultaneously outstanding operations at the sender for the dynamic protocol. The number of simultaneously outstanding operations at the receiver was held constant at 32.



(a) Throughput (b) Ratio of direct to total messages

Fig. 12. Effect of message size on the dynamic protocol. The number of simultaneously outstanding operations at the receiver was 4 and at the sender 2.

Connect-X 2 HCAs with one port configured for 10Gbps RoCE, as this was the highest speed supported by this network emulator. The nodes were connected through the Anue but there were no other switches between the two systems. The nodes contain Intel Xeon X5670 CPUs running at 2.93GHz, 64 gigabytes of RAM, and a PCIe Generation 2 bus. The nodes run Scientific Linux 6.3 with OFED 3.5. We emulated the effect of distance by using the Anue network emulator to set a fixed round-trip delay of 48 ms. The tests themselves used the same blast tool as in the previous experiments.

We again vary the number of simultaneously outstanding operations at both the sender and receiver for the dynamic protocol. The results are shown in Fig. 13. Interestingly, over distance, all three algorithms had similar performance. However, when 4-32 buffers are used, the indirect protocol gives slightly higher throughput than the direct protocol, with a difference of about 100-400 Mbps. The dynamic protocol is shown to adapt and give higher throughput than the direct protocol in this case.

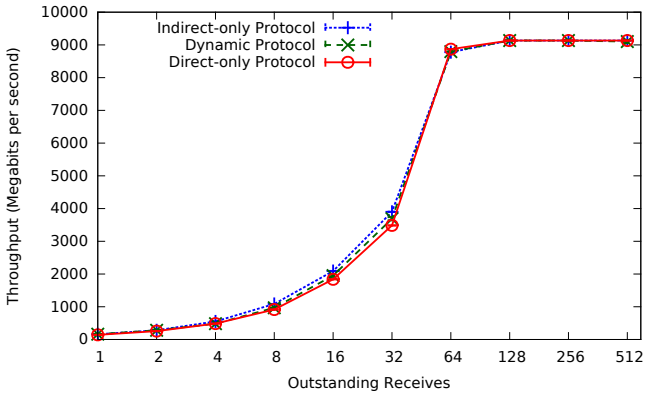


Fig. 13. Throughput vs. number of simultaneously outstanding operations at the sender and receiver. The number of simultaneous outstanding operations at the sender and receiver were equal. Message sizes were selected at random from an exponential distribution with $\lambda = 1048576$ and a maximum message size of 4 MiB. The delay was 48 ms.

C. Performance conclusions

From these results, we conclude that EXS stream throughput with dynamic adaptation is much higher if there are more outstanding receives than outstanding sends. RDMA assumes that the receiver is ready to receive before the sender attempts to send data. By using more outstanding receive operations, the receiver is ready to receive more often and this increases throughput and decreases CPU usage at the receiver. However, using fewer outstanding operations increases the adaptability of the protocol. Thus, this protocol is well-suited for legacy applications which may not be written to handle multiple outstanding network operations on a single connection.

There is a small difference in throughput between direct vs. indirect transfers over distance. It should be noted that our test programs involved repeated transfers in one direction with many outstanding receive operations. Thus, since ADVERT messages are very small, when the receiver sends ADVERTs, all of the ADVERTs will arrive at the sender quickly in succession. When the receiver receives a direct transfer and the application immediately posts a new receive operation, the next ADVERT in sequence will be sent out. If this ADVERT arrives at the sender before the sender is finished sending messages corresponding to the first burst of ADVERTs, then the dynamic protocol will be able to use direct transfers with no additional cost. However, in applications in which many receive operations cannot be pre-posted, ADVERTs will have a much larger effect on throughput. In this case, the dynamic protocol will cause most of the transfers to be indirect, thus avoiding suboptimal latency or throughput.

This algorithm is dynamic and adapts to the current network conditions, but if the network and application reach a steady state, then the algorithm will remain in its current transfer mode. This ability to adapt lasts throughout the entire life of the socket connection, so a sudden, large change in network state will cause the protocol to switch transfer modes appropriately.

V. Conclusions

This paper presented: (i) the design of an algorithm to dynamically switch between buffered and zero-copy transfers over RDMA, depending on current conditions; (ii) a proof of the correctness and safety of this algorithm; (iii) an implementation of this algorithm as part of the UNH EXS library; (iv) the results and evaluation of the performance of this implementation.

VI. Future Work

We plan to develop more test applications in order to further determine the performance profile of the dynamic algorithm, such as dynamically changing send and receive message sizes and burstiness during a connection. We also plan on performing latency studies.

In addition, we wish to do more extensive testing over distance. We plan to use our network emulator to set a jitter function in order to vary the delay to see the effect of jitter on our implementation. We would also like to do more testing on a real long-distance network, such as the ESnet 100G testbed.

Acknowledgments

The authors would like to thank the University of New Hampshire InterOperability Laboratory (UNH-IOL) for the use of their RDMA cluster for the development, maintenance, and testing of UNH EXS. We also would like to thank the UNH-IOL and Ixia for the use of the Anue network emulator for performance testing.

This material is based upon work supported by the National Science Foundation under Grant No. OCI-1127228 and under the National Science Foundation Graduate Research Fellowship under Grant. No. DGE-0913620.

References

[1] OpenFabrics Alliance. http://www.openfabrics.org.

[2] OpenFabrics Enterprise Distribution. https://www.openfabrics.org/resources/ofed-for-linux-ofed-for-windows/ofed-overview.html.

[3] The Message Passing Interface (MPI) standard. [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/

[4] Rsockets. [Online]. Available: https://www.openfabrics.org/ofa-documents/doc_download/495-rsockets.html

[5] Unh exs. [Online]. Available: http://www.iol.unh.edu/services/research/unh-exs

[6] R. Russell, "The Extended Sockets Interface for Accessing RDMA Hardware," in *Proceedings of the 20th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2008)*, T. Gonzalez, Ed., Nov. 2008, pp. 279–284.

[7] ——, "A General-purpose API for iWARP and InfiniBand," in *the First Workshop on Data Center Converged and Virtual Ethernet Switching (DC-CAVES)*, Sep. 2009.

[8] Interconnect Software Consortium in association with the Open Group, "Extended Sockets API (ES-API) Issue 1.0," Jan. 2005.

[9] Open Group, "http://www.opengroup.org," 2009.

[10] T. Carlin, "Implementation of an RDMA verbs driver for GridFTP," Master's thesis, University of New Hampshire, New Hampshire, USA, 2012.

[11] Infiniband Trade Association, "Supplement to Infiniband Architecture Specification Volume 1, Release 1.2.1: Annex A4: Sockets Direct Protocol (SDP)," Oct. 2011.

[12] D. Goldenberg, M. Kagan, R. Ravid, and M. S. Tsirkin, "Zero copy sockets direct protocol over Infiniband—preliminary implementation and performance analysis," in *High Performance Interconnects, 2005. Proceedings. 13th Symposium on*. IEEE, 2005, pp. 128–137.

[13] P. Balaji, S. Bhagvat, H.-W. Jin, and D. K. Panda, "Asynchronous zero-copy communication for synchronous sockets in the sockets direct protocol (SDP) over InfiniBand," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006, pp. 8–pp.

[14] Y. Lin, J. Han, J. Gao, and X. He, "uStream: a user-level stream protocol over InfiniBand," in *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*. IEEE, 2009, pp. 65–71.

[15] P. MacArthur and R. D. Russell, "A performance study to guide RDMA programming decisions," in *2012 IEEE 14th International Conference on High Performance Computing and Communication*. IEEE, 2012, pp. 778–785.

[16] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[17] J. Ajanovic, "PCI Express 3.0 overview," in *Proceedings of Hot Chip: A Symposium on High Performance Chips*, 2009.