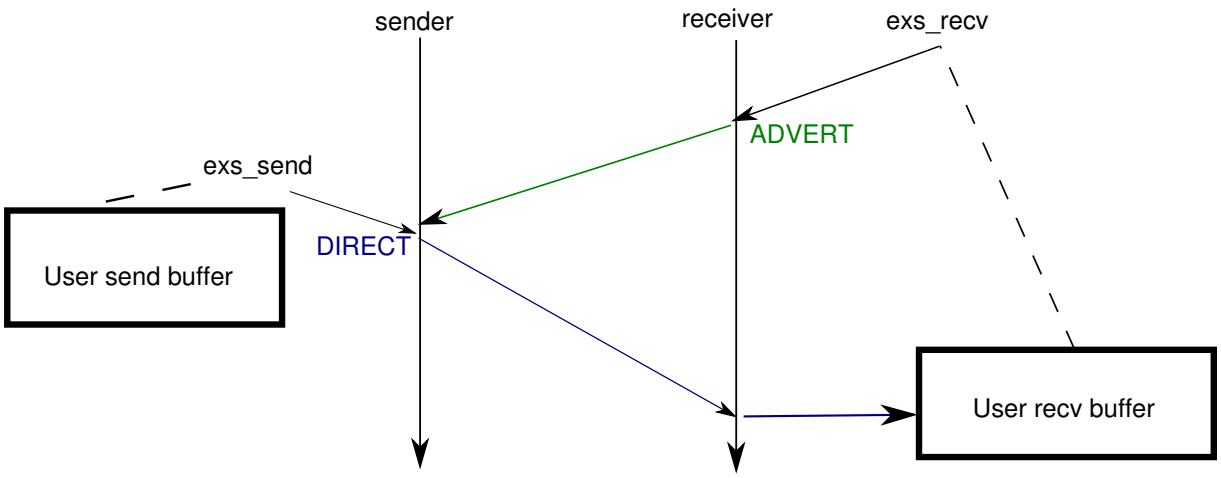

Figure 1: An illustration of a direct transfer.

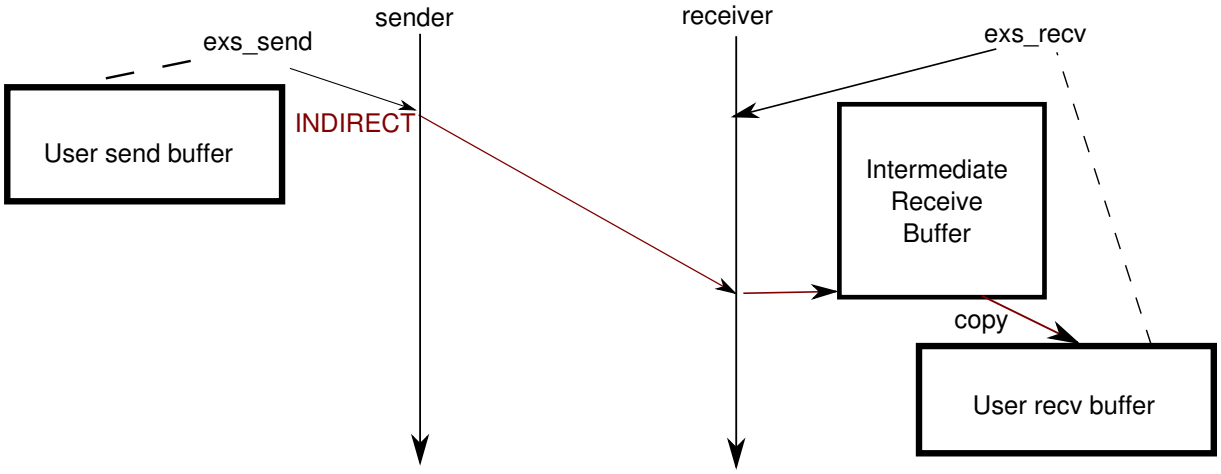

Figure 2: An illustration of an indirect transfer.

This document describes the dynamic stream protocol and algorithms required to implement it in UNH EXS. If you just want a specification of the wire messages sent in the current version of UNH EXS, read `protocol.txt` in this directory.

The dynamic stream protocol [1] is intended to optimize sending stream data by sending zero-copy direct transfers when advertisements are available and sending indirect transfers when they are not.

During a direct transfer, the sender waits for an advertisement of a registered user data buffer given in a call to exs_recv(). The sender then performs a WWI operation which places the send data directly into the user's memory. The advertisement is consumed by this transfer *unless* the MSG_WAITALL flag was supplied, in which case the advertisement is only partially consumed and future exs_send() operations can continue to use the advertisement until the buffer is full. A direct transfer is illustrated in Figure 1.

During an indirect transfer, the sender performs a WWI operation to send as much data as it can from the user's exs_send() buffer to the intermediate buffer at the receiver. There

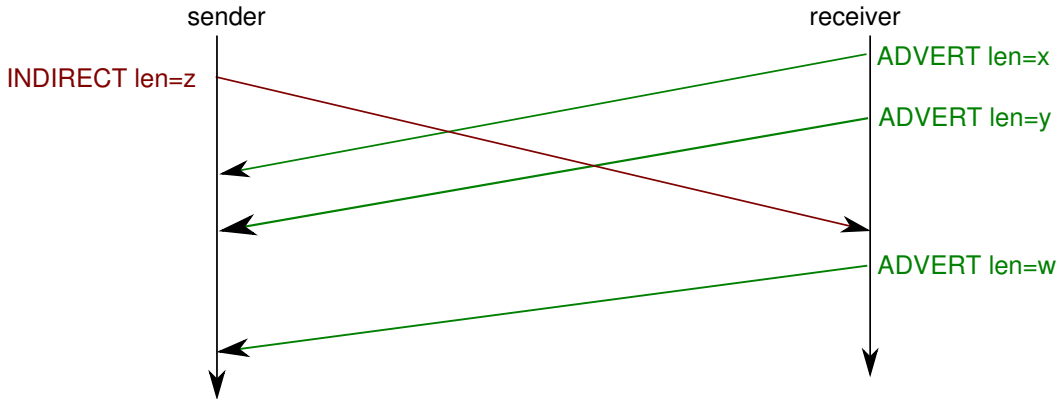

Figure 3: This figure illustrates that the sender cannot be sure exactly which advertisements were consumed by the indirect transfer. This problem is resolved by the algorithm used by UNH EXS.

is currently *no* intermediate buffer at the sender. The receiver then copies data from the intermediate buffer to the user's exs_recv() buffer. An exs_recv() operation is completely satisfied if any data gets copied unless the MSG_WAITALL flag was supplied, in which case the exs_recv() operation remains pending until more data arrives into the intermediate buffer. An indirect transfer is illustrated in Figure 2.

The dynamic protocol allows a mixture of both direct and indirect transfers. This document describes some of the issues that must be handled with this protocol.

The key issue is that although the user requests n bytes in an exs_recv() call, UNH EXS will eventually place m bytes into the user's buffer, where $1 \leq m \leq n$. Unless the MSG_WAITALL flag is supplied by the user to request exactly $m = n$ bytes, the exact value of m is not known at the time that UNH EXS sends the advertisement. However, the exs_recv() call is asynchronous, so the user can have many simultaneously outstanding exs_recv() transactions. This presents a problem when the sender tries to reconcile multiple received advertisements with indirect transfers, as illustrated in Figure 3. Simply placing a sequence number into the advertisement does not work, since we don't know the exact sequence number to use. We also cannot simply force all exs_recv() operations to use MSG_WAITALL, since this would not allow the user to request more data than will actually be sent, which is a behavior that is allowed by TCP.

Thus, in every advertisement, the receiver places an *estimated* next-expected sequence number and a phase number. This value is stored in the `source_seqno` field of the transaction block. The estimate is incremented by 1 after each advertisement is sent. When any exs_recv() operation completes, if an advertisement was sent for the operation— whether or not the transfer was actually direct— we add the difference to `source_seqno` in order to correct future estimates. That is, if 4 bytes were actually received, we would add 3 to `source_seqno`.

The phase number simply refers to a sequence of consecutive direct or indirect transfers. The sender and receiver at each side start at phase 0. During an even numbered phase, direct transfers are always allowed, since the sender and receiver are guaranteed to be in sync in the absence of indirect transfers. When the sender sends an indirect transfer, it increments

its phase to an odd number. Similarly, when the receiver receives an indirect transfer, it also increments its phase if it was previously even. During an odd phase, the receiver will remain passive until it has flushed all previous advertisements (by receiving indirect transfers), at which point it will increment its phase and send a new advertisement. During an odd phase, an advertisement can only be accepted by the sender if the advertisement has a higher phase number than the current phase number at the sender, and the sequence number of the advertisement matches the sender's current sequence number. This indicates that the advertisement actually refers to the advertisement at the head of the receiver's exs_recv() queue. Once the sender accepts an advertisement, the sender increments its phase back to even. However, if an advertisement with a higher phase number is rejected, the sender increments its phase to the first odd number beyond the advertisement's phase. This logic ensures that when a direct transfer responding to advertisement A arrives at the receiver, the exs_recv() operation corresponding to advertisement A is at the head of the exs_recv() queue.

We discuss specific scenarios that motivated the design in Appendix A of this document. If any changes are made to the implementation, these scenarios should be re-tested to ensure that the changes do not cause "stale" advertisements to be affected.

References

[1] P. MacArthur and R. Russell, "An Efficient Method for Stream Semantics over RDMA," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2014)*, May 2014.

A Case Study

The case studies are presented in Figures 4, 5, 6, and 7. The interpretation of the labels is provided by the legend in Table 1. In all figures, the client is on the left and the server is on the right. Phase numbers are presented as "R*n*" or "S*n*", where "R*n*" is the phase number for data going in the left-to-right direction, and "S*n*" is the phase number for data going in the right-to-left direction.

All scenarios in this case study were discovered by using the `pumpexs` utility included in UNH EXS. The `pumpexs` tool blasts data as quickly as it can from client to server. However, there is an initialization phase consisting of three steps. First, the client sends the message "go" to the server. Second, the server receives the message and posts exs_recv() transactions for all of its buffers. Finally, the server sends the message "ok" to the client, which allows the client to start sending data.

Event	**Format**
Send ADVERT	A*id_number*;*seqno*;*length*; *phase*
Send Direct WWI	D;*matching_advert*;*seqno*;*length*; *phase*
Send Indirect WWI	I;(*invalidated_adverts_if_any*);*seqno*; *length*;*phase*
Recv Any	*phase*

Table 1: Legend

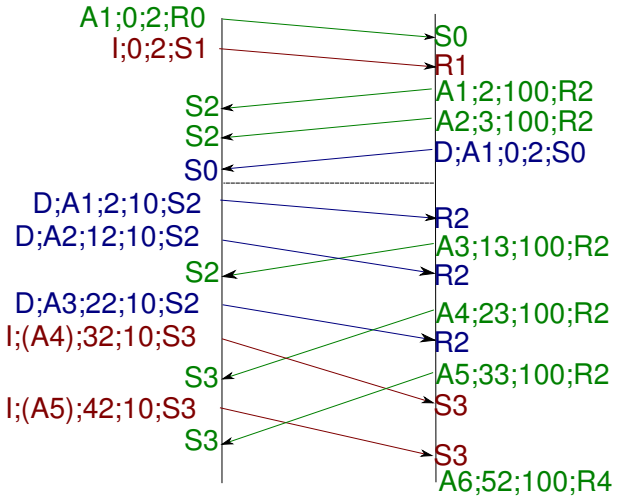

Figure 4: Here the receiver is forbidden to send an advertisement A6 after receiving indirect transfer I3 until all outstanding advertisements have been satisfied, whether by direct or indirect transfer from the sender. In this case, the receiver must wait for advertisement A5 to be satisfied via an indirect transfer. The sender increments its phase after sending an indirect transfer and does not accept advertisements with a phase less than its current phase. Without these restrictions, the sender has no way of knowing how many advertisements are consumed at the receiver by indirect transfer, creating a possibility of a "break" in the exs_recv() sequence at the receiver.

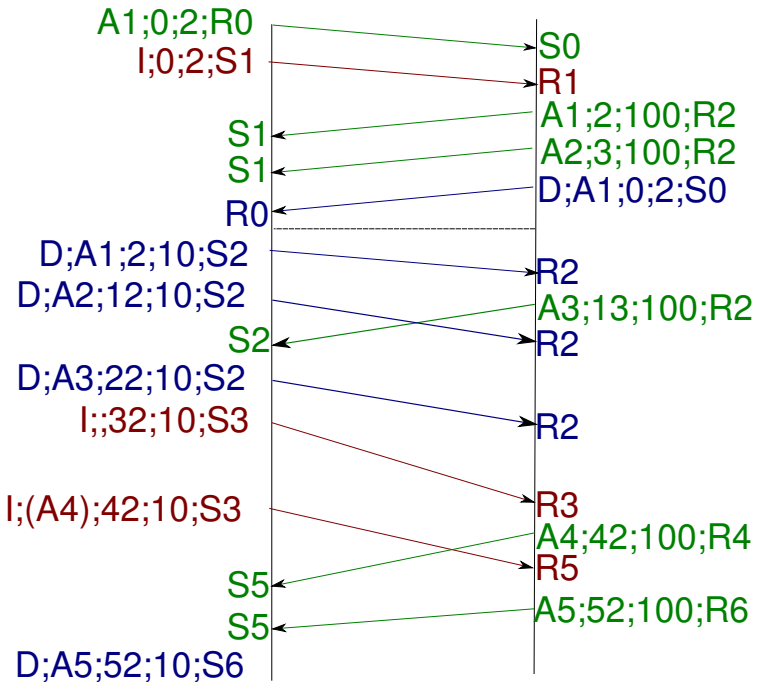

Figure 5: Here, advertisement A4 should be rejected since it was satisfied by indirect transfer I3. Advertisement A5 should be accepted since at that point the sender has caught up with the receiver. To do this, we examine the sequence number of the advertisement when we receive advertisement A5 with a phase 6 which is greater than the current phase at the sender 5. We only accept the advertisement if the sequence numbers match.

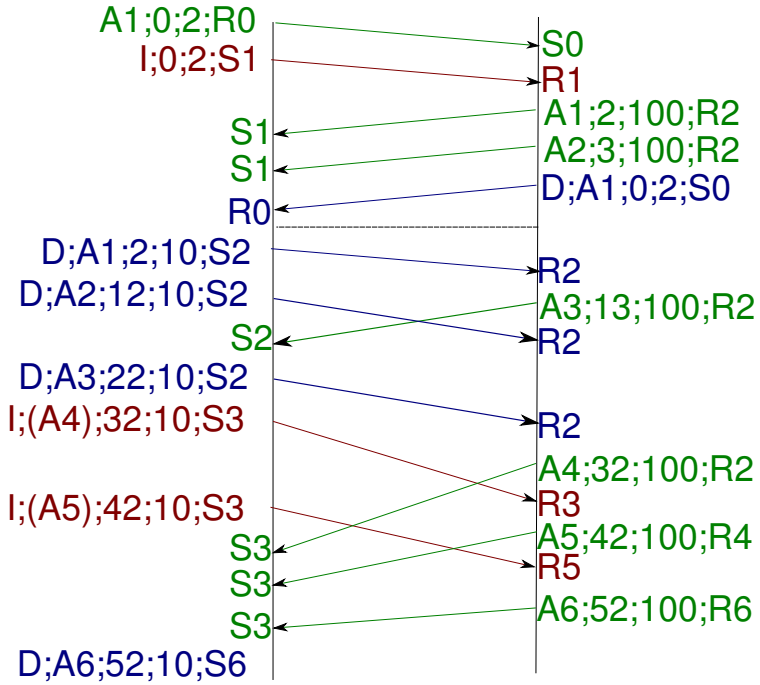

Figure 6: This is another tricky case similar to Figure 5, in which multiple advertisements are matched by indirect WWI messages. Here, we simply follow the same process and we wind up matching the correct advertisement.

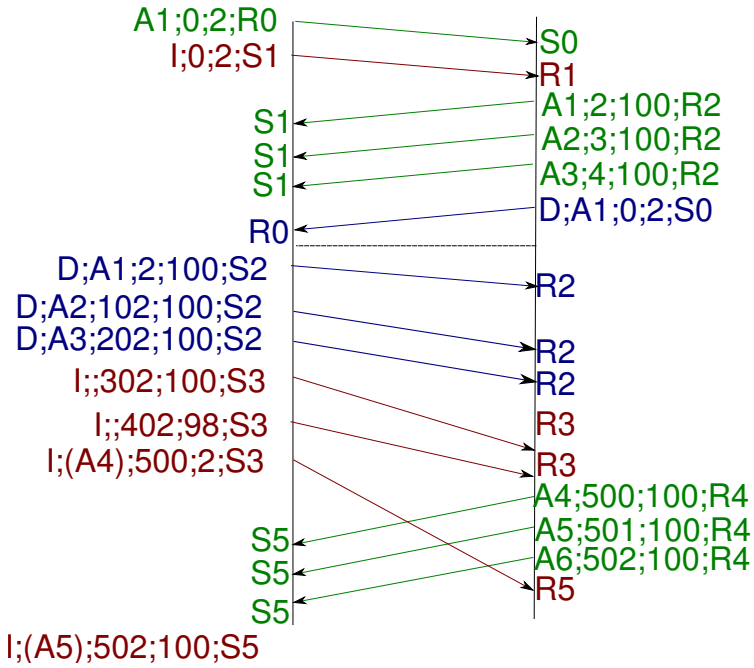

Figure 7: This illustrates a very tricky interaction between the stream buffer and the advertisement sequence. Here, we use a very small stream buffer of 200 bytes, but send and receive 100 byte messages. This means that when the sender gets ahead after sending D202, the sender can send a 100 byte indirect transfer (I302), a 98 byte indirect transfer (I402), and a 2 byte indirect transfer (I500) in sequence before filling the stream buffer and being unable to send. This means that the sequence number at the sender is now 502. This is due to the initial 2 byte message that starts the `pumpexs` application. If the receiver then catches up and starts sending advertisements after receiving the first two RDMA WRITE message, the sequence number in the third advertisement A6 will appear to match, but the correct match would be the *previous* advertisement A5. To avoid this scenario, we instead increment the phase as soon as we reject an advertisement to be larger than that in the advertisement, so that any future advertisements in that sequence will be rejected. In this case, the sender's phase is incremented from 3 to 5 since it rejected advertisement A4 with phase number 4.