RDMA programming concepts

Robert D. Russell <rdr@iol.unh.edu>

InterOperability Laboratory &
Computer Science Department
University of New Hampshire
Durham, New Hampshire 03824, USA
RDMA benefits for user applications

- High throughput
- Low latency
- High messaging rate
- Low CPU utilization
- Low memory bus contention
RDMA Technologies

- **InfiniBand** – (44.8% of top 500 supercomputers)
  - SDR 4x – 8 Gbps
  - DDR 4x – 16 Gbps
  - QDR 4x – 32 Gbps
  - FDR 4x – 54 Gbps

- **iWarp** – internet Wide Area RDMA Protocol
  - 10 Gbps
  - 40 Gbps

- **RoCE** – RDMA over Converged Ethernet
  - 10 Gbps
  - 40 Gbps
How users can access RDMA

- Via application library – MPI, Lustre, NFS_RDMA
  - embed RDMA into library, so user-level API is unchanged
- Via API like “normal” socket I/O – SDP, rsockets
  - socket(), connect(), send(), recv(), poll(), close()
- Via API like “normal” I/O – GridFTP-XIO
  - open(), read(), write(), poll(), close()
- Explicitly program with OpenFabrics Software (verbs)
  - ibv_post_recv(), ibv_post_send(), ibv_poll_cq()
Layering with user level libraries

User Application

User level libraries, such as MPI

OFS Verbs API

**OSI Layers**

Transport

- RDMAP
- DDP
- MPA
- TCP

Network

- IP

Data Link

- Ethernet MAC & LLC

Physical

- Ethernet PHY

CA

- IWARP “RNIC”
- RoCE “NIC”
- InfiniBand “HCA”
# Layering directly to OFS verbs

## OSI Layers

<table>
<thead>
<tr>
<th>Layer</th>
<th>User Application</th>
<th>OFS Verbs API</th>
<th>IB Transport API</th>
<th>IB Network</th>
<th>IB Link</th>
<th>IB PHY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport</td>
<td>RDMAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DDP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TCP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network</td>
<td>IP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Link</td>
<td>Ethernet MAC &amp; LLC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Physical</td>
<td>Ethernet PHY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td>IWARP “RNIC”</td>
<td>RoCE “NIC”</td>
<td>InfiniBand “HCA”</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- IWARP “RNIC”
- RoCE “NIC”
- InfiniBand “HCA”
RDMA and TCP similarities

- Both utilize the *client-server* model

- Both require a *connection* for reliable transport

- Both provide a *reliable transport* mode
  - TCP provides a reliable in-order sequence of *bytes*
  - RDMA provides a reliable in-order sequence of *messages*
RDMA and TCP differences

- **“zero copy”** – RDMA transfers data directly from user virtual memory on one node to user virtual memory on another node, TCP copies into/out of system buffers on both nodes.
- **“kernel bypass”** – RDMA involves no kernel intervention during data transfers, TCP does.
- **asynchronous operation** – RDMA does not block threads during I/O operations, TCP does.
- **message oriented** – RDMA transfer preserves message boundaries, TCP does not.
“Normal” TCP/IP socket access

- Byte streams – require App to delimit/recover message boundaries (if desired)
- Synchronous – send(), recv() block until data copied
  - O_NONBLOCK, MSG_DONTWAIT are not asynchronous, they are “try” and get error
- send() and recv() are paired
  - both sides must participate in the transfer
- System copies data into “hidden” system buffers
  - order, timing of send() and recv() are irrelevant
  - user memory accessible immediately before and immediately after each send() and recv() call
TCP RECV()
RDMA RECV()
RDMA access model

 Messages – preserve App's message boundaries
 Asynchronous – no blocking during a transfer, which
  – starts when metadata added to “work queue”
  – finishes when status available in “completion queue”
 1-sided (unpaired) and 2-sided (paired) transfers
 No data copying into system buffers
  – order, timing of send() and recv() are relevant
    • recv() must be waiting before issuing send()
  – memory involved in transfer should not be touched by program between start and completion of transfer
Kernel bypass

- Program uses **queues** to interact directly with CA
- Queue Pair – user enqueues work for CA
  - work request – data structure from user describing transfer
  - send queue – holds work requests to CA that send data
  - recv queue – holds work requests to CA that receive data
- Completion Queue – CA enqueues status to user
  - work completion – data structure from CA containing transfer result status
  - one **completion queue** can hold work completions for both send and receive transfers
  - can also have separate completion queues for each
Transfer and completion queues
Verbs interface to queues

Programmer Level

OFA Verbs

Work Request

lbv_post_send
lbv_post_recv

Work Completion

lbv_poll_cq

Adapter Level

Work Queue

WQE

Completion Queue

CQE

Channel Adapter

Posting

Completion

“wire”

© 2013 OpenFabrics Alliance, Inc.
Asynchronous Data Transfer

- **Posting**
  - term used to mark the initiation of a data transfer
  - user adds a “work request” to a “work queue”

- **Completion**
  - term used to mark the end of a data transfer
  - user removes a “work completion” from a “completion queue”

- **Important note:**
  - between posting and completion the state of user memory involved in the transfer is **undefined** and should NOT be changed or used by the user program
Posting – Completion
RDMA transfer types

- **send/recv** – similar to “normal” TCP sockets
  - each send on one side must match a recv on other side
- **RDMA_WRITE** – only in RDMA
  - “pushes” data into remote virtual memory
- **RDMA_READ** – only in RDMA
  - “pulls” data out of remote virtual memory
- Same verbs and data structures used by all types
  - parameter values and field values depend on type
RDMA send-receive data transfer
send-recv similarities with sockets

- Sender **must** issue `listen()` before client issues `connect()`.
- Both sender and receiver **must** actively participate in all data transfers:
  - Sender **must** issue `send()` operations.
  - Receiver **must** issue `recv()` operations.
- Sender does not know remote receiver's virtual memory location.
- Receiver does not know remote sender's virtual memory location.
send-receive differences with sockets

- “normal” TCP/IP transfers are buffered
  - time order of send() and recv() on each side is irrelevant

- RDMA transfers are not buffered
  - recv() must be posted by receiver before send() can be posted by sender
  - not doing this results in a few retries, then fatal error

- “normal” TCP/IP has no notion of “memory registration”

- RDMA requires that all memory participating in a transfer be “registered”
User memory **must** be registered

- So kernel can “pin” or “lock” it into physical memory
  - so user memory can not be paged in/out during transfer
  - so CA can obtain physical to virtual mapping
    - CA, not OS, does mapping during a transfer
    - CA, not OS, checks validity of the transfer

- So CA can create “**keys**” linking memory, process
  - supplied by user as part of every transfer
  - allow user to control access rights of a transfer
  - allow CA to find correct mapping in a transfer
  - allow CA to verify access rights in a transfer
ping-pong using send-recv
3 phases in reliable connections

- **Setup Phase**
  - obtain, convert addressing information
  - create, configure local endpoints for communication
  - setup local memory for use in transfers
  - establish connection with remote side

- **Use Phase**
  - actually transfer data to/from remote side

- **Break-down Phase**
  - basically “undo” setup phase
  - close connection, free memory, free other resources
## Client setup phase

<table>
<thead>
<tr>
<th>TCP</th>
<th>RDMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. process command-line options</td>
<td>process command-line options</td>
</tr>
<tr>
<td>2. convert DNS name and port no. getaddrinfo()</td>
<td>convert DNS name and port no. rdma_getaddrinfo()</td>
</tr>
<tr>
<td>3. create local end point socket()</td>
<td>create local end point rdma_create_ep()</td>
</tr>
<tr>
<td>4.</td>
<td>create completion queue to get CA status ibv_create_cq()</td>
</tr>
<tr>
<td>5.</td>
<td>create queue pair to give CA xfer metadata rdma_create_qp()</td>
</tr>
<tr>
<td>6. allocate user virtual memory malloc()</td>
<td>allocate user virtual memory malloc()</td>
</tr>
<tr>
<td>7.</td>
<td>register user virtual memory with CA rdma_reg_msgs()</td>
</tr>
<tr>
<td>8. create connection with server connect()</td>
<td>create connection with server rdma_connect()</td>
</tr>
</tbody>
</table>
### Client use phase

<table>
<thead>
<tr>
<th>TCP</th>
<th>RDMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>9. mark start time for statistics</td>
<td>mark start time for statistics</td>
</tr>
<tr>
<td>10. start of transfer loop</td>
<td>start of transfer loop</td>
</tr>
<tr>
<td>11. post receive to catch agent’s pong data</td>
<td>rdma_post_recv()</td>
</tr>
<tr>
<td>12. transfer ping data to agent</td>
<td>post send to start transfer of ping data to agent</td>
</tr>
<tr>
<td>send()</td>
<td>rdma_post_send()</td>
</tr>
<tr>
<td>13. receive pong data from agent</td>
<td>wait for both send and receive to complete</td>
</tr>
<tr>
<td>recv()</td>
<td>ibv_poll_cq()</td>
</tr>
<tr>
<td>14. optionally verify pong data ok</td>
<td>optionally verify pong data ok</td>
</tr>
<tr>
<td>memcmp()</td>
<td>memcmp()</td>
</tr>
<tr>
<td>15. end of transfer loop</td>
<td>end of transfer loop</td>
</tr>
<tr>
<td>16. mark stop time and print statistics</td>
<td>mark stop time and print statistics</td>
</tr>
<tr>
<td></td>
<td>TCP</td>
</tr>
<tr>
<td>---</td>
<td>-----</td>
</tr>
<tr>
<td>17.</td>
<td>break connection with server close()</td>
</tr>
<tr>
<td>18.</td>
<td></td>
</tr>
<tr>
<td>19.</td>
<td>free user virtual memory free()</td>
</tr>
<tr>
<td>20.</td>
<td></td>
</tr>
<tr>
<td>21.</td>
<td></td>
</tr>
<tr>
<td>22.</td>
<td></td>
</tr>
<tr>
<td>23.</td>
<td>free getaddrinfo resources freeaddrinfo()</td>
</tr>
<tr>
<td>24.</td>
<td>“unprocess” command-line options</td>
</tr>
</tbody>
</table>
Server participants

- **Listener**
  - waits for connection requests from client
  - gets new system-provided connection to client
  - hands-off new connection to agent
  - never transfers any data to/from client

- **Agent**
  - creates control structures to deal with one client
  - allocates memory to deal with one client
  - performs all data transfers with one client
  - disconnects from client when transfers all finished
## Listener setup and use phases

<table>
<thead>
<tr>
<th>TCP</th>
<th>RDMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. process command-line options</td>
<td>process command-line options</td>
</tr>
<tr>
<td>2. convert DNS name and port no.</td>
<td>convert DNS name and port no.</td>
</tr>
<tr>
<td>getaddrinfo()</td>
<td>rdma_getaddrinfo()</td>
</tr>
<tr>
<td>3. create local end point socket()</td>
<td>create local end point</td>
</tr>
<tr>
<td></td>
<td>rdma_create_ep()</td>
</tr>
<tr>
<td>4. bind to address and port bind()</td>
<td></td>
</tr>
<tr>
<td>5. establish socket as listener</td>
<td>establish socket as listener</td>
</tr>
<tr>
<td>listen()</td>
<td>rdma_listen()</td>
</tr>
<tr>
<td>6. start loop</td>
<td>start loop</td>
</tr>
<tr>
<td>7. get connection request from client</td>
<td>get connection request from client</td>
</tr>
<tr>
<td>accept()</td>
<td>rdma_get_request()</td>
</tr>
<tr>
<td>8. hand connection over to agent</td>
<td>hand connection over to agent</td>
</tr>
<tr>
<td>9. end loop</td>
<td>end loop</td>
</tr>
</tbody>
</table>
# Listener breakdown phase

<table>
<thead>
<tr>
<th>TCP</th>
<th>RDMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>10. destroy local endpoint</td>
<td>destroy local endpoint</td>
</tr>
<tr>
<td>close()</td>
<td>rdma_destroy_ep()</td>
</tr>
<tr>
<td>11. free getaddrinfo resources</td>
<td>free getaddrinfo resources</td>
</tr>
<tr>
<td>freegetaddrinfo()</td>
<td>rdma_freegetaddrinfo()</td>
</tr>
<tr>
<td>12. “unprocess” command-line options</td>
<td>“unprocess” command-line options</td>
</tr>
</tbody>
</table>
## Agent setup phase

<table>
<thead>
<tr>
<th>TCP</th>
<th>RDMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. make copy of listener's options</td>
<td>make copy of listener's options</td>
</tr>
</tbody>
</table>
| 2. | create completion queue to get CA status  
ibv_create_cq() |
| 3. | create queue pair to give CA xfer metadata  
rdma_create_qp() |
| 4. allocate user virtual memory  
malloc() | allocate user virtual memory  
malloc() |
| 5. | register user virtual memory with CA  
rdma_reg_msgs() |
| 6. | post first receive of ping data from client  
rdma_post_recv() |
| 7. | finalize connection with client  
rdma_accept() |
## Agent use phase

<table>
<thead>
<tr>
<th></th>
<th>TCP</th>
<th>RDMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>mark start time for statistics</td>
<td>mark start time for statistics</td>
</tr>
<tr>
<td>9</td>
<td>start of transfer loop</td>
<td>start of transfer loop</td>
</tr>
<tr>
<td>10</td>
<td>wait to receive ping data from client recv()</td>
<td>wait to receive ping data from client ibv_poll_cq()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>post next receive for ping data from client rdma_post_recv()</td>
</tr>
<tr>
<td>12</td>
<td>transfer pong data to client send()</td>
<td>post send to start transfer of pong data to client rdma_post_send()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wait for send to complete ibv_poll_cq()</td>
</tr>
<tr>
<td>14</td>
<td>end of transfer loop</td>
<td>end of transfer loop</td>
</tr>
<tr>
<td>15</td>
<td>mark stop time and print statistics</td>
<td>mark stop time and print statistics</td>
</tr>
<tr>
<td></td>
<td>TCP</td>
<td>RDMA</td>
</tr>
<tr>
<td>---</td>
<td>-------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>16.</td>
<td>break connection with client</td>
<td>break connection with client</td>
</tr>
<tr>
<td></td>
<td>close()</td>
<td>rdma_disconnect()</td>
</tr>
<tr>
<td>17.</td>
<td>deregister user virtual memory</td>
<td>rdma_dereg_mr()</td>
</tr>
<tr>
<td>18.</td>
<td>free user virtual memory</td>
<td>free user virtual memory</td>
</tr>
<tr>
<td></td>
<td>free()</td>
<td>free()</td>
</tr>
<tr>
<td>19.</td>
<td>destroy queue pair</td>
<td>destroy queue pair</td>
</tr>
<tr>
<td></td>
<td>rdma_destory_qp()</td>
<td>rdma_destory_qp()</td>
</tr>
<tr>
<td>20.</td>
<td>destroy completion queue</td>
<td>destroy completion queue</td>
</tr>
<tr>
<td></td>
<td>ibv_destroy_cq()</td>
<td>ibv_destroy_cq()</td>
</tr>
<tr>
<td>21.</td>
<td>destroy local end point</td>
<td>destroy local end point</td>
</tr>
<tr>
<td></td>
<td>rdma_destroy_ep()</td>
<td>rdma_destroy_ep()</td>
</tr>
<tr>
<td>22.</td>
<td>free copy of listener's options</td>
<td>free copy of listener's options</td>
</tr>
</tbody>
</table>
ping-pong SEND/RECV performance

- InfiniBand QDR 4x, through a switch
- Client
  - round-trip-time 15.7 microseconds
  - user CPU time 100% of elapsed time
  - kernel CPU time 0% of elapsed time
- Server
  - round-trip time 15.7 microseconds
  - user CPU time 100% of elapsed time
  - kernel CPU time 0% of elapsed time
How to reduce 100% CPU usage

- **Cause is “busy polling” to wait for completions**
  - in tight loop on `ibv_poll_cq()`
  - burns CPU since most calls find nothing

- **Why is “busy polling” used at all?**
  - simple to write such a loop
  - gives very fast response to completions
  - (i.e., gives low latency)
"busy polling" to get completions

1. start loop

2. `ibv_poll_cq()` to get any completion in queue

3. exit loop if a completion is found

4. end loop
How to eliminate “busy polling”

- Cannot make `ibv_poll_cq()` block
  - no flag parameter
  - no timeout parameter

- How to eliminate busy polling loop and just wait?

- Solution is “wait-for-notify” event mechanism
  - `ibv_req_notify_cq()` - tell CA to send a notify “event” when next WC enters CQ
  - `ibv_get_cq_event()` - blocks until gets notify “event”
  - `ibv_ack_cq_event()` - acknowledges notify “event”
Backflows from CA to user
Backflows with completion channel

User Program

- **<blocking>**
  - ibv_get_cq_event ()

- **<non-blocking>**
  - ibv_poll_cq ()

- **<blocking>**
  - rdma_get_cm_event ()

Completion event channel with fd

Completion queue

Completion event channel with fd

One-shot notify

struct ibv_wc

struct rdma_cm_event
"wait-for-notify" to get completions

1. start loop
2. `ibv_poll_cq()` gets any completion in CQ
3. exit loop if completion is found
4. `ibv_req_notify()` enables CA to send event on next completion added to CQ
5. `ibv_poll_cq()` gets any completion between 2&4
6. exit loop if completion is found
7. `ibv_get_cq_event()` blocks until CA sends event
8. `ibv_ack_cq_events()` acknowledges event
9. end loop
ping-pong SEND/RECV performance with “wait-for-notify”

- **Client**
  - round-trip-time 21.1 microseconds – up 34%
  - user CPU time 9.0% of elapsed time – was 100%
  - kernel CPU time 9.1% of elapsed time – was 0%
  - total CPU time 18% of elapsed time – down 82%

- **Server**
  - round-trip time 21.1 microseconds – up 34%
  - user CPU time 14.5% of elapsed time – was 100%
  - kernel CPU time 6.5% of elapsed time – was 0%
  - total CPU time 21% of elapsed time – down 79%
rdma_xxxx “wrappers” around ibv_xxxx

- rdma_get_recv_comp() - wrapper for notify loop on receive completion queue
- rdma_get_send_comp() - wrapper for notify loop on send completion queue
- rdma_post_recv() - wrapper for ibv_post_recv()
- rdma_post_send() - wrapper for ibv_post_send()
- rdma_reg_msgs() - wrapper for ibv_reg_mr for SEND/RECV
- rdma_dereg_mr() - wrapper for ibv_dereg_mr()
where to find “wrappers”, prototypes, data structures, etc.

- `/usr/include/rdma/rdma_verbs.h` contains `rdma_xxxx` “wrappers”
- `/usr/include/infiniband/verbs.h` contains `ibv_xxxx` verbs and all ibv data structures, defines, types, and function prototypes
- `/usr/include/rdma/rdma_cm.h` contains `rdma_yyyy` verbs and all rdma data structures, etc. for connection management
Transfer choices

- **TCP/UDP transfer operations**
  - `send()`/`recv()` (and related forms)

- **RDMA transfer operations**
  - `SEND/RECV` similar to TCP/UDP
  - `RDMA_WRITE` “push” to remote virtual memory
  - `RDMA_READ` “pull” from remote virtual memory
  - `RDMA_WRITE_WITH_IMM` “push” with notification
RDMA_WRITE operation

- Very different concept from normal TCP/IP
- Very different concept from RDMA Send/Recv
- Only one side is active, other is passive
- Active side (requester) issues RDMA_WRITE
- Passive side (responder) does NOTHING!
- A better name would be “RDMA_PUSH”
  - data is “pushed” from active side’s virtual memory into passive side’s virtual memory
  - passive side issues no operation, uses no CPU cycles, gets no indication “push” started or completed
RDMA_WRITE data flow

![Diagram of RDMA_WRITE data flow](image)

- **Memory Region**
  - Active A

- **Memory Region**
  - Passive P
Differences with RDMA Send

- Active side calls `ibv_post_send()`
  - opcode is RDMA_WRITE, not SEND
  - work request MUST include passive side's virtual memory address and memory registration key
- Prior to issuing this operation, active side MUST obtain passive side's address and key
  - use send/recv to transfer this “metadata”
  - (could actually use any means to transfer “metadata”)
- Passive side provides “metadata” that enables the data “push”, but does not participate in it
Similarities with RDMA Send

- Both transfer types move **messages**, not streams
- Both transfer types are unbuffered ("zero copy")
- Both transfer types require **registered** virtual memory on both sides of a transfer
- Both transfer types operate **asynchronously**
  - active side posts work request to send queue
  - active side gets work completion from completion queue
- Both transfer types use same verbs and data structures (although values and fields differ)
RDMA_READ operation

- Very different from normal TCP/IP
- Very different from RDMA Send/Recv
- Only one side is active, other is passive
- Active side (requester) issues RDMA_READ
- Passive side (responder) does NOTHING!
- A better name would be “RDMA_PULL”
  - data is “pulled” into active side’s virtual memory from passive side’s virtual memory
  - passive side issues no operation, uses no CPU cycles, gets no indication “pull” started or completed
RDMA_READ data flow
Ping-pong using RDMA_WRITE/READ

- Client is active side in ping-pong loop
  - client posts RDMA_WRITE out of ping buffer
  - client posts RDMA_READ into pong buffer
- Server agent is passive side in ping-pong loop
  - does nothing
- Server agent must send its buffer address and registration key to client before the loop starts
- Client must send agent a message with total number of transfers after the loop finishes
  - otherwise agent has no way of knowing this number
  - agent needs to receive something to know when to finish
Ping-pong using RDMA_WRITE/READ
Client ping-pong transfer loop

- start of transfer loop
- `ibv_post_send()` of RDMA_WRITE ping data
- `ibv_poll_cq()` to wait for RDMA_WRITE completion
- `ibv_post_send()` of RDMA_READ pong data
- `ibv_poll_cq()` to wait for RDMA_READ completion
- optionally verify pong data equals ping data
- end of transfer loop
Agent ping-pong transfer loop

- `ibv_post_recv()` to catch client's "finished" message

- wait for completion of "finished" from client
  - use "busy polling" or "wait-for-notify"
ping-pong RDMA_WRITE/READ performance with “wait-for-notify”

- **Client**
  - round-trip-time 14.3 microseconds – down from 21.1
  - user CPU time 26.4% of elapsed time – up from 9.0%
  - kernel CPU time 3.0% of elapsed time – down from 9.1%
  - total CPU time 29.4% of elapsed time – up from 18%

- **Server**
  - round-trip time 14.3 microseconds – down from 21.1
  - user CPU time 0% of elapsed time – down from 14.5%
  - kernel CPU time 0% of elapsed time – down from 6.5%
  - total CPU time 0% of elapsed time – down from 21.0%
Improving performance further

- All postings discussed so far generate completions
  - required for all `ibv_post_recv()` postings
  - optional for `ibv_post_send()` postings

- User controls completion generation with `IBV_SEND_SIGNALLED` flag in `ibv_post_send()`
  - supplying this flag always generates a completion for that posting
  - not setting this flag generates a completion for that posting only in case of an error – a successful transfer generates no completion
How client benefits from this feature

- RDMA_READ posting follows RDMA_WRITE
- RDMA_READ must finish after RDMA_WRITE
  - due to strict ordering rules in standards
- Therefore, don't need to do anything with RDMA_WRITE completion
  - completion of RDMA_READ guarantees RDMA_WRITE transfer succeeded
  - error on RDMA_WRITE transfer will generate a completion

- Therefore we can send RDMA_WRITE unsignaled and NOT wait for its completion
Client unsignaled transfer loop

- start of transfer loop
- `ibv_post_send()` with **unsignaled** RDMA_WRITE
  - generates no completion (except on error)
- **do not wait** for RDMA_WRITE completion
- `ibv_post_send()` of RDMA_READ pong data
- `ibv_poll_cq()` to wait for RDMA_READ completion
  - will get RDMA_WRITE completion on error
- optionally verify pong data equals ping data
- end of transfer loop
ping-pong RDMA_WRITE/READ performance with unsignaled, notify

- **Client**
  - round-trip-time 8.3 microseconds – down 42%
  - user CPU time 28.0% of elapsed time – up from 26.4%
  - kernel CPU time 2.8% of elapsed time – down from 3.0%
  - total CPU time 30.8% of elapsed time – up from 29.4%

- **Server**
  - round-trip time 8.3 microseconds – down 42%
  - user CPU time 0% of elapsed time – unchanged
  - kernel CPU time 0% of elapsed time – unchanged
  - total CPU time 0% of elapsed time – unchanged
Ping-pong performance summary

- Rankings for Round-Trip Time (RTT)
  - 8.3 usec unsigaled RDMA_WRITE/READ with wait for notify
  - 14.3 usec signaled RDMA_WRITE/READ with wait for notify
  - 15.7 usec signaled SEND/RECV with busy polling
  - 21.1 usec signaled SEND/RECV with wait for notify

- Rankings for client total CPU usage
  - 18.0% signaled SEND/RECV with wait for notify
  - 29.4% signaled RDMA_WRITE/READ with wait for notify
  - 30.8% unsigaled RDMA_WRITE/READ with wait for notify
  - 100% signaled SEND/RECV with busy polling
Multicast concept
RDMA Multicast

- Only possible with IB and RoCE, not iWARP
- Single SEND is delivered to RECV in multiple destinations – switches make copies as necessary carefully avoiding duplicate deliveries
- Only possible in **Unreliable Datagram (UD) mode**
  - only RDMA SEND/RECV operations allowed
  - message size limited to underlying MTU size
- Based on concept of **multicast groups**
  - communication model is peer-to-peer
  - any node can SEND to entire group at any time
  - messages are lost on nodes with no RECV posted
RDMA Multicast Groups

- Created and managed by subnet manager
  - utilizes IPv6 multicast addresses
- Any program can join a group at any time
  - `rdma_join_multicast()`
  - attaches existing queue pair to multicast group
- Once joined, program can leave group at any time
  - `rdma_leave_multicast()`
  - detaches existing queue pair from multicast group
- Only possible in **Unreliable Datagram (UD)** mode
  - only `Send/Recv` operations allowed
  - both sides must actively participate in data transfers
Restriction on Multicast Groups

- Maximum MTU for an active group is decided when first CA joins the group
  - size of 1st CA's active MTU becomes group's MTU

- CAs with smaller active MTU sizes cannot join an active group
  - if 1st MTU is 1024, others can be 1024, 2048, 4096
  - if 1st MTU is 2048, others can be 2048, 4096
  - if 1st MTU is 4096, others can be 4096 only

- Maximum MTU for an active group is unchanged as long as group contains at least 1 member
Multicast Publish-Subscribe

- Publisher maintains data repository, periodically updates and posts it to multicast group using `ibv_post_send()`
- Subscriber posts 2 or more `ibv_post_recv()`
- When an `ibv_post_recv()` completes:
  - post another `ibv_post_recv()` into another buffer
  - use published data from completed buffer
- Subscribers (and publishers) can join or leave multicast group at any time
  - no indication given to other group members
OpenFabrics software training

- 2-day OFA Training Class
  - “Writing Application Programs for RDMA Using OFA Software”
  - www.openfabrics.org/resources
  - taught periodically at the University of New Hampshire InterOperability Laboratory
  - can also be taught at a company site
  - contact Rupert Dance at rsdance@soft-forge.com
QUESTIONS?
THANK YOU!