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

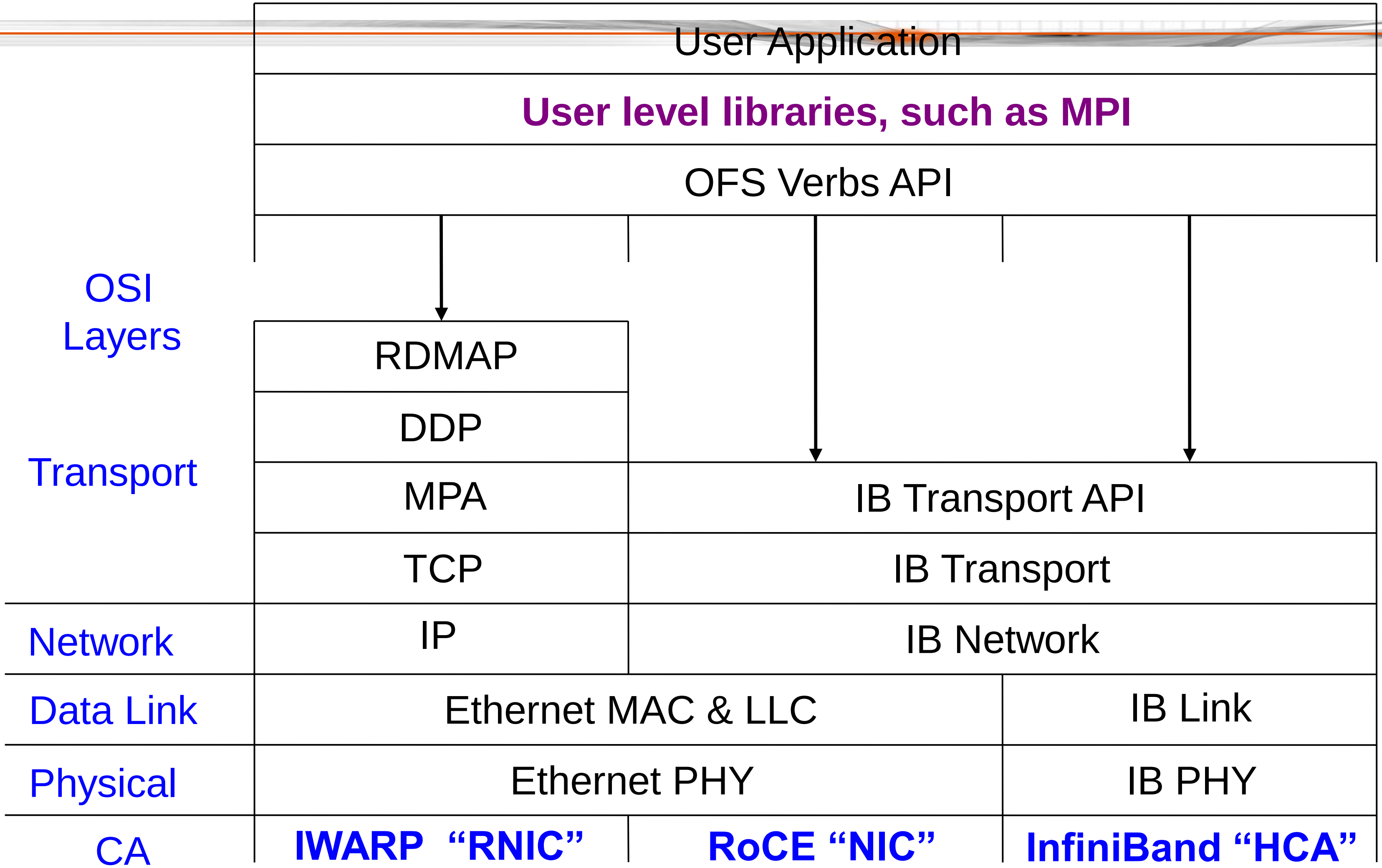
RDMA Technologies

- 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

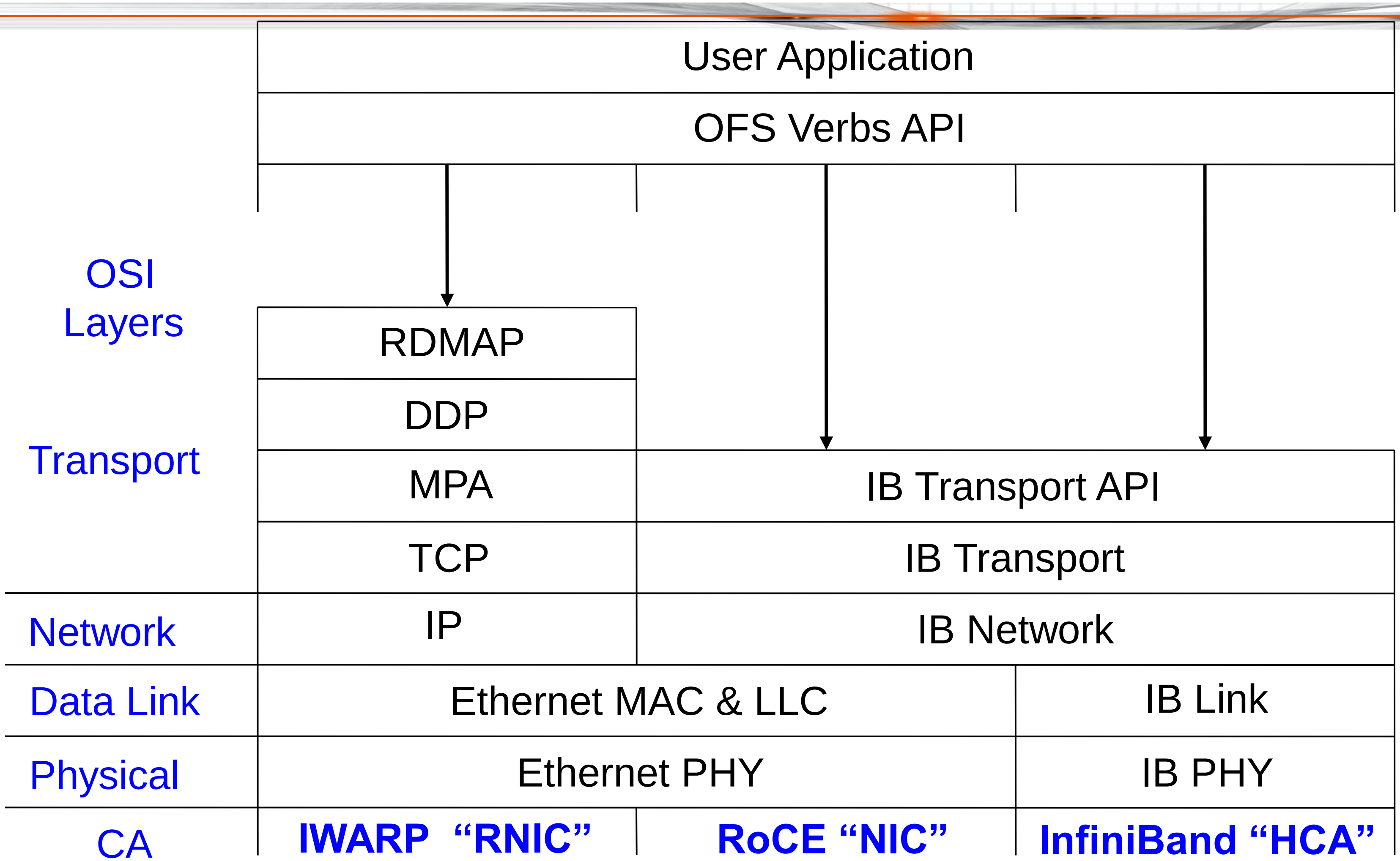
- Via application library – MPI, Lustre, NFS_RDMA
  - embed RDMA into library, so user-level API is unchanged
- Via API like "normal" socket I/O – SDP, rsockets
  - socket(), connect(), send(), recv(), poll(), close()
- Via API like "normal" I/O – GridFTP-XIO
  - open(), read(), write(), poll(), close()
- 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	RDMAP		
	DDP		
Transport	MPA	IB Transport API	
	TCP	IB Transport	
Network	IP	IB Network	
Data Link	Ethernet MAC & LLC		IB Link
Physical	Ethernet PHY		IB PHY
CA	**IWARP “RNIC”**	**RoCE “NIC”**	**InfiniBand “HCA”**

Layering directly to OFS verbs



OSI Layers			
	User Application		
	OFS Verbs API		
	RDMAP		
	DDP		
Transport	MPA	IB Transport API	
	TCP	IB Transport	
Network	IP	IB Network	
Data Link	Ethernet MAC & LLC		IB Link
Physical	Ethernet PHY		IB PHY
CA	**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**

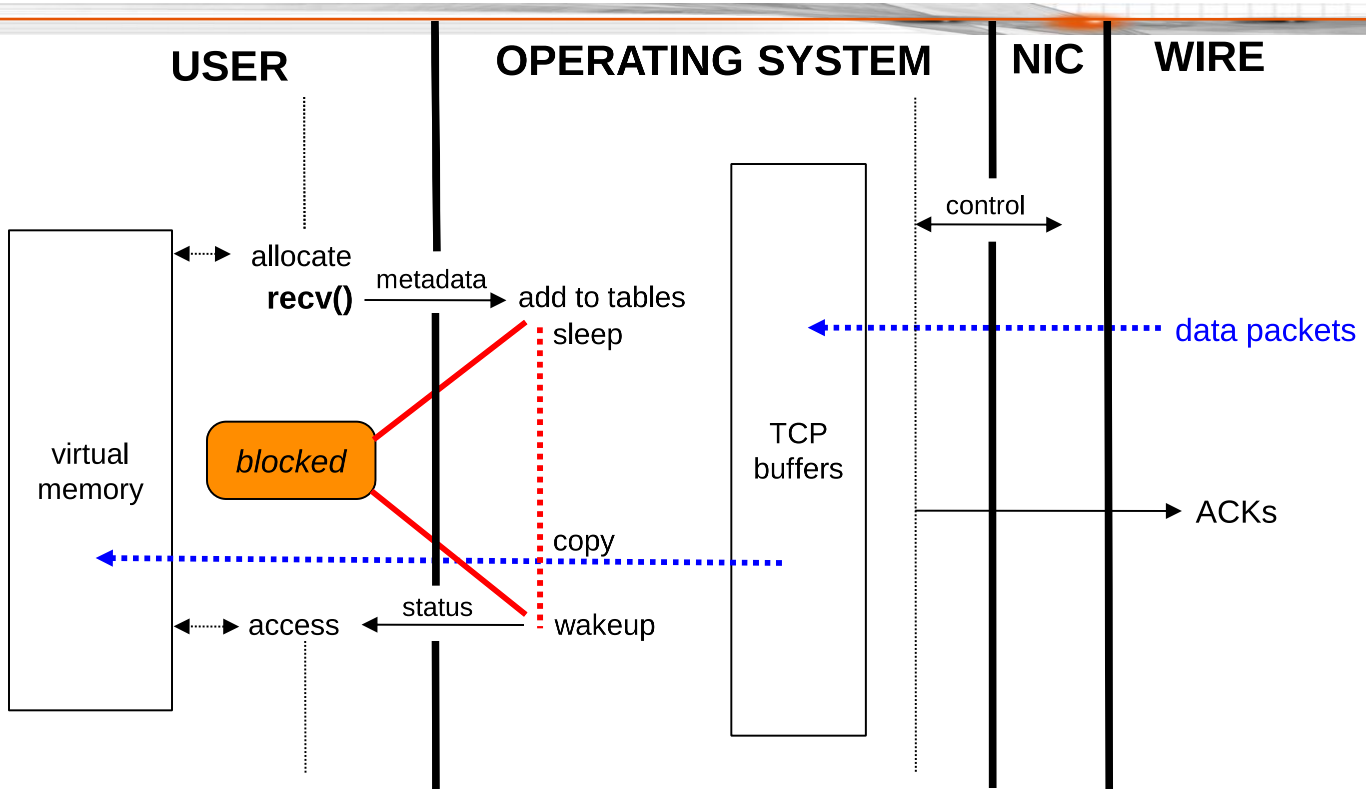
RDMA and TCP differences

- “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

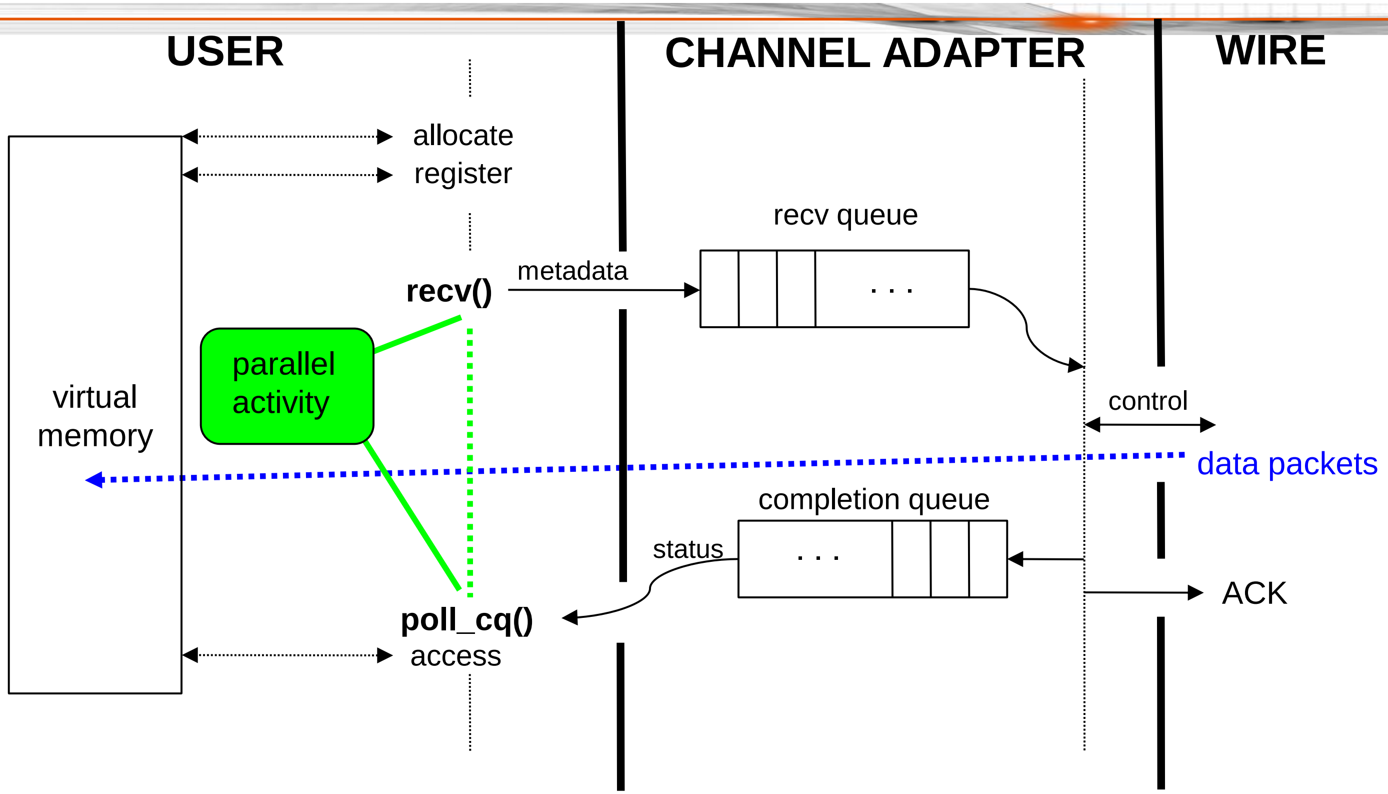
“Normal” TCP/IP socket access

- 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()


USER
OPERATING SYSTEM
NIC
WIRE
control
allocate
metadata
recv()
add to tables
sleep
data packets
virtual memory
blocked
TCP buffers
ACKs
copy
status
access
wakeup

RDMA RECV()


USER
CHANNEL ADAPTER
WIRE
allocate
register
virtual memory
recv()
metadata
recv queue
. . .
parallel activity
control
data packets
completion queue
status
. . .
ACK
poll_cq()
access

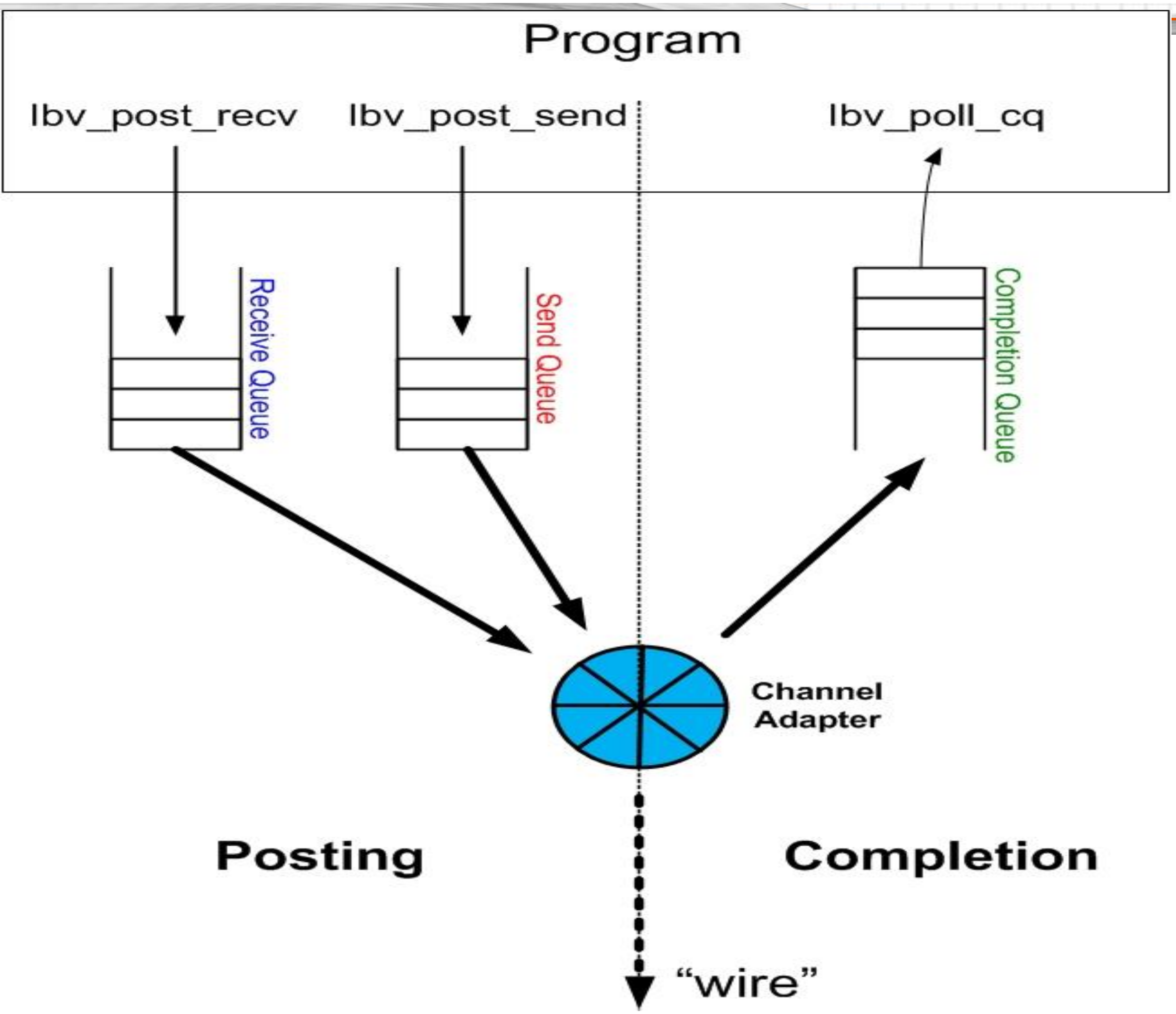
RDMA access model

- ❖ 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

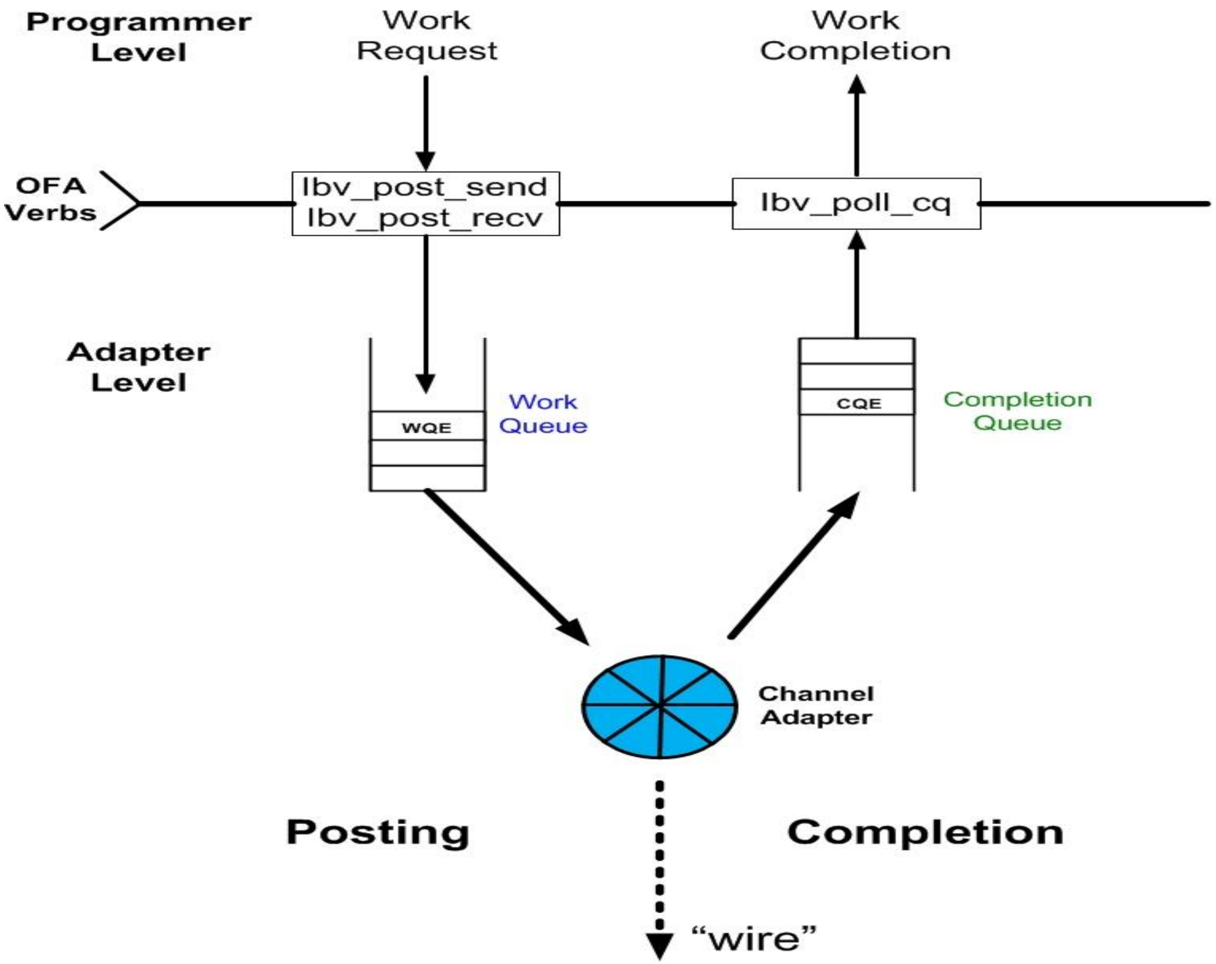
Kernel bypass

- 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



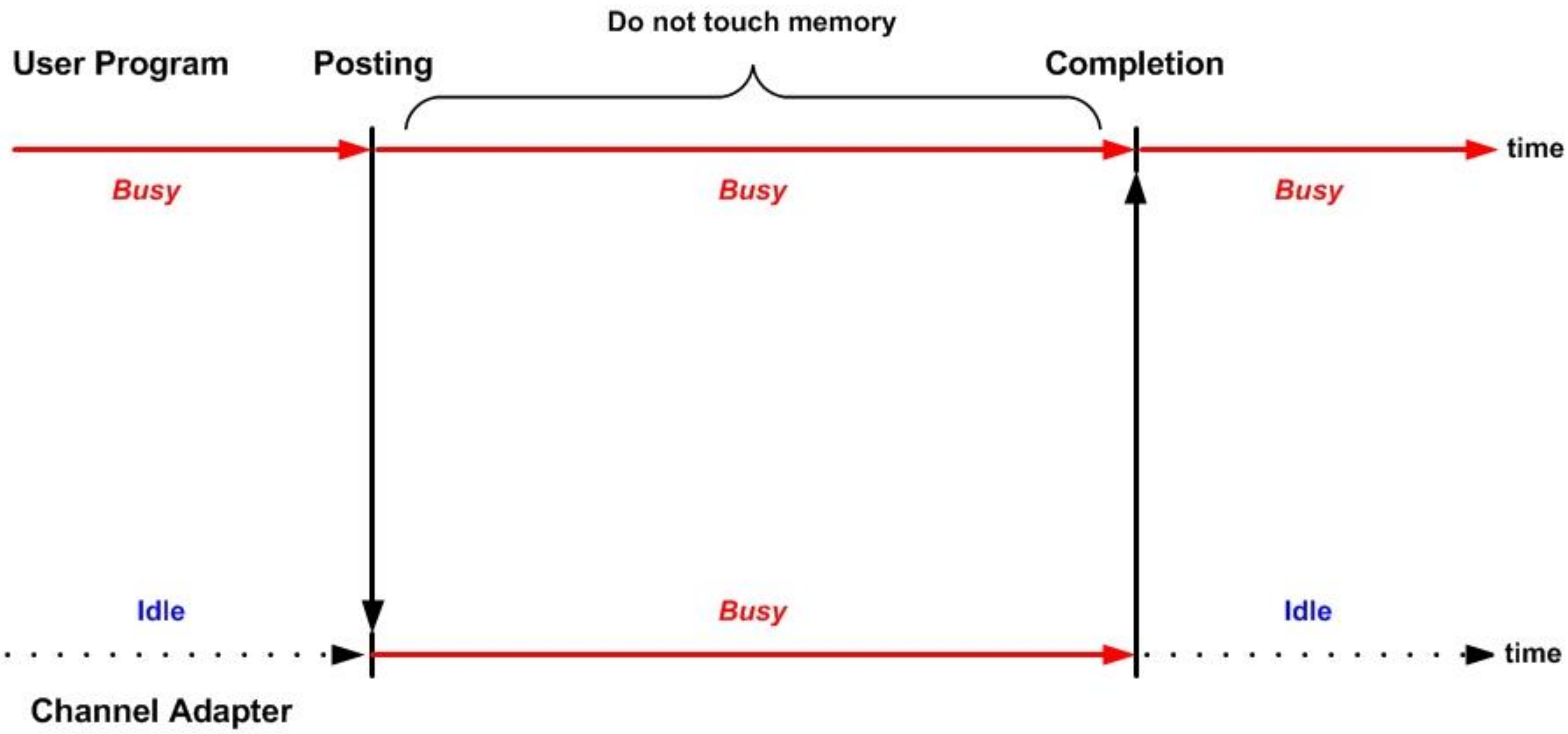
Verbs interface to queues



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

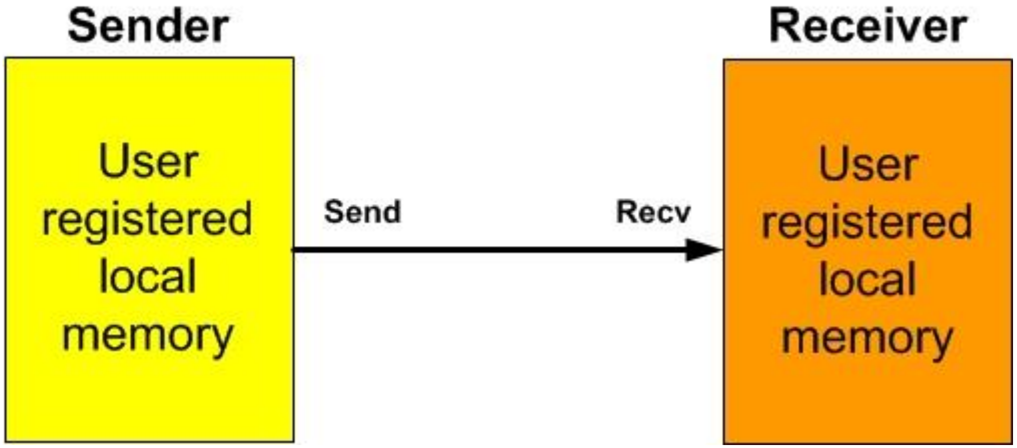
Posting – Completion



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-recv 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

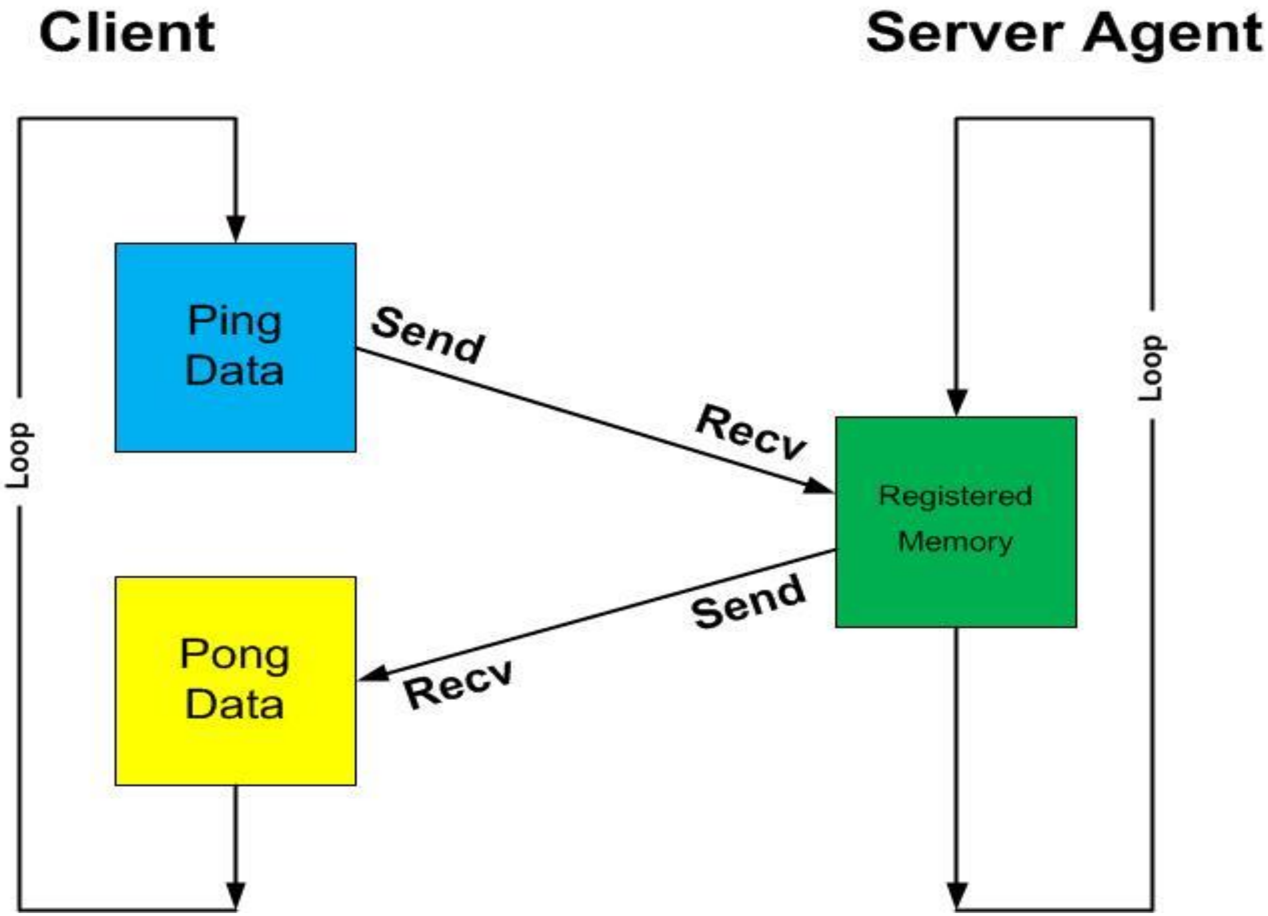
send-recv differences with sockets

- “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

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

Client use phase

TCP	RDMA
9. mark start time for statistics	mark start time for statistics
10. start of transfer loop	start of transfer loop
11.	**post receive** to catch agent's pong data rdma_post_recv()
12. transfer ping data to agent send()	**post send** to start transfer of ping data to agent rdma_post_send()
13. receive pong data from agent recv()	**wait for both send and receive to complete** ibv_poll_cq()
14. optionally verify pong data ok memcmp()	optionally verify pong data ok memcmp()
15. end of transfer loop	end of transfer loop
16. mark stop time and print statistics	mark stop time and print statistics

Client breakdown phase

TCP	**RDMA**
17. break connection with server close()	break connection with server rdma_disconnect()
18.	**deregister user virtual memory** rdma_dereg_mr()
19. free user virtual memory free()	free user virtual memory free()
20.	**destroy queue pair** rdma_destory_qp()
21.	**destroy completeion queue** ibv_destroy_cq()
22.	**destroy local end point** rdma_destroy_ep()
23. free getaddrinfo resources freeaddrinfo()	free rdma_getaddrinfo resources rdma_freeaddrinfo()
24. “unprocess” command-line options	“unprocess” command-line options

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

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

Listener breakdown phase

TCP	RDMA
10. destroy local endpoint close()	destroy local endpoint rdma_destroy_ep()
11. free getaddrinfo resources freegetaddrinfo()	free getaddrinfo resources rdma_freegetaddrinfo()
12. “unprocess” command-line options	“unprocess” command-line options

Agent setup phase

	TCP	RDMA
1.	make copy of listener's options	make copy of listener's options
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

TCP	RDMA
8. mark start time for statistics	mark start time for statistics
9. start of transfer loop	start of transfer loop
10. wait to receive ping data from client recv()	wait to receive ping data from client ibv_poll_cq()
11.	**post next receive** for ping data from client rdma_post_recv()
12. transfer pong data to client send()	**post send** to start transfer of pong data to client rdma_post_send()
13.	**wait for send to complete** ibv_poll_cq()
14. end of transfer loop	end of transfer loop
15. mark stop time and print statistics	mark stop time and print statistics

Agent breakdown phase

TCP	RDMA
16. break connection with client close()	break connection with client rdma_disconnect()
17.	**deregister user virtual memory** rdma_dereg_mr()
18. free user virtual memory free()	free user virtual memory free()
19.	**destroy queue pair** rdma_destory_qp()
20.	**destroy completeion queue** ibv_destroy_cq()
21.	**destroy local end point** rdma_destroy_ep()
22. free copy of listener's options	free copy of listener's options

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)

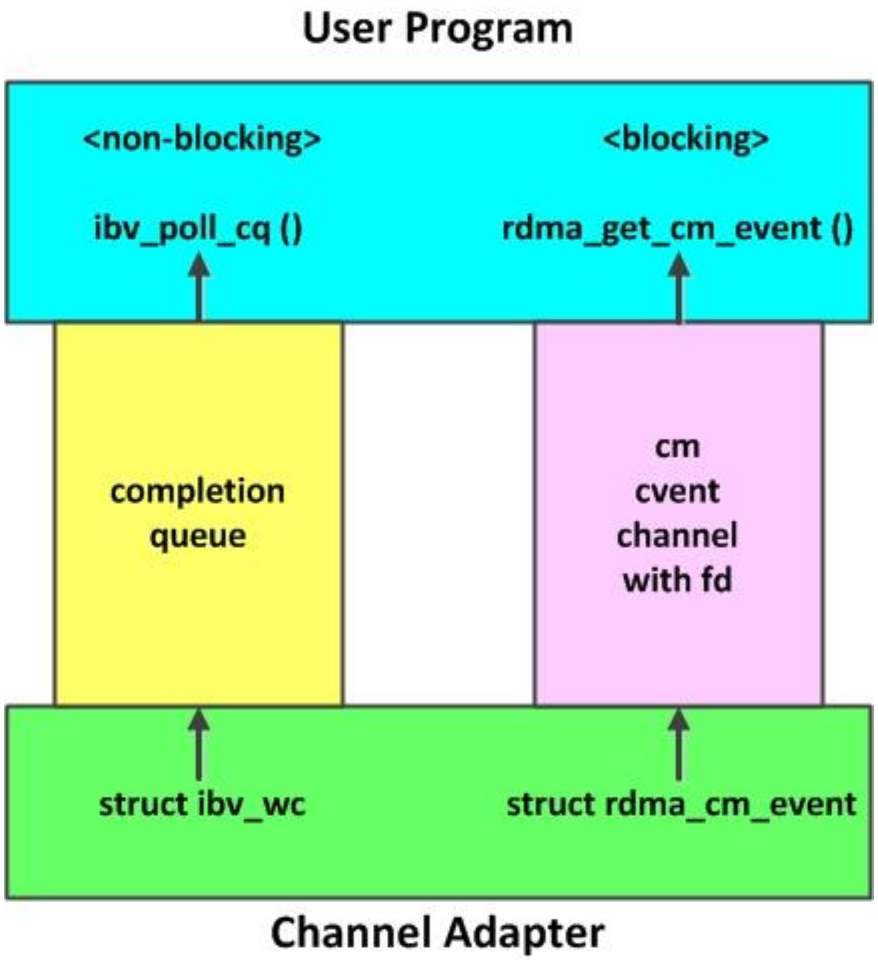
”busy polling” to get completions

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



User Program

<non-blocking>

ibv_poll_cq ()

<blocking>

rdma_get_cm_event ()

completion queue

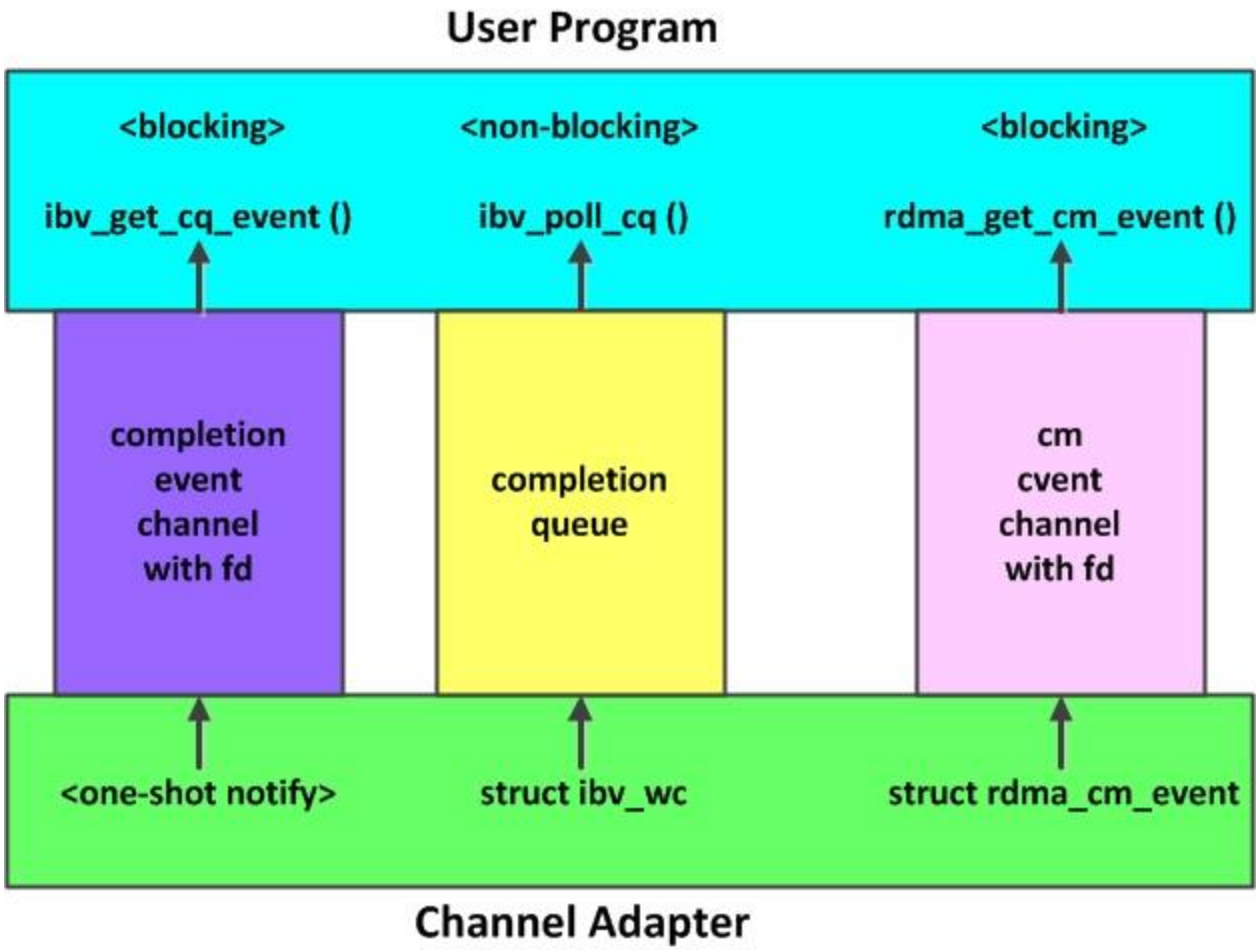
cm cvent channel with fd

struct ibv_wc

struct rdma_cm_event

Channel Adapter

Backflows with completion channel



”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

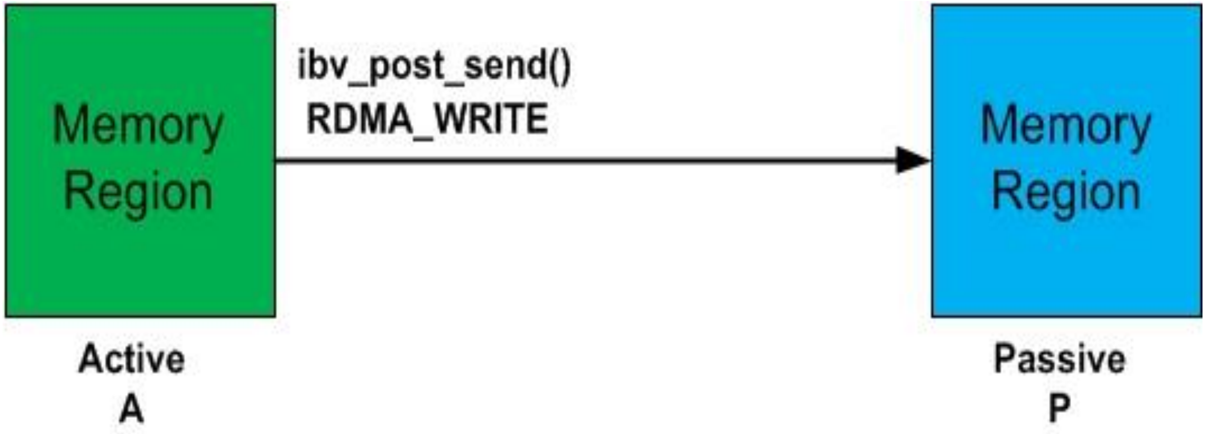
- TCP/UDP transfer operations
  - send()/recv() (and related forms)

- 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



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

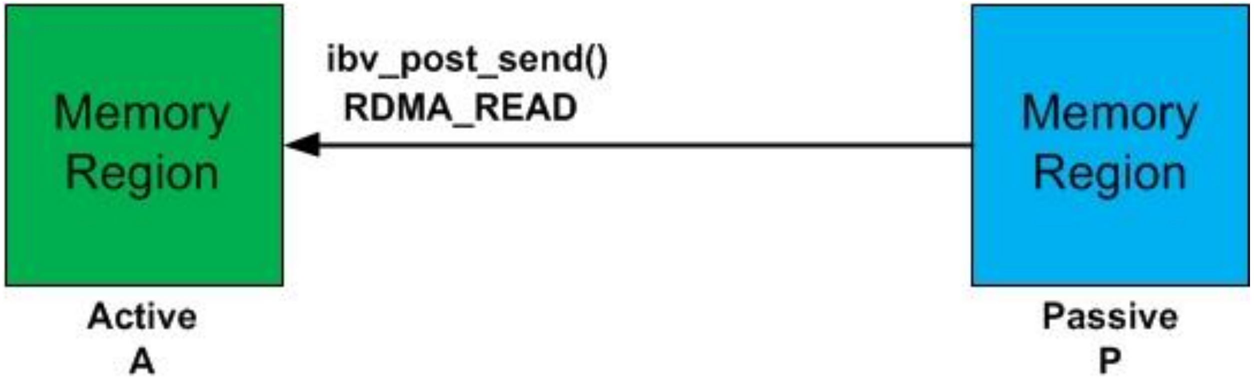
Similarities with RDMA Send

- ❖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

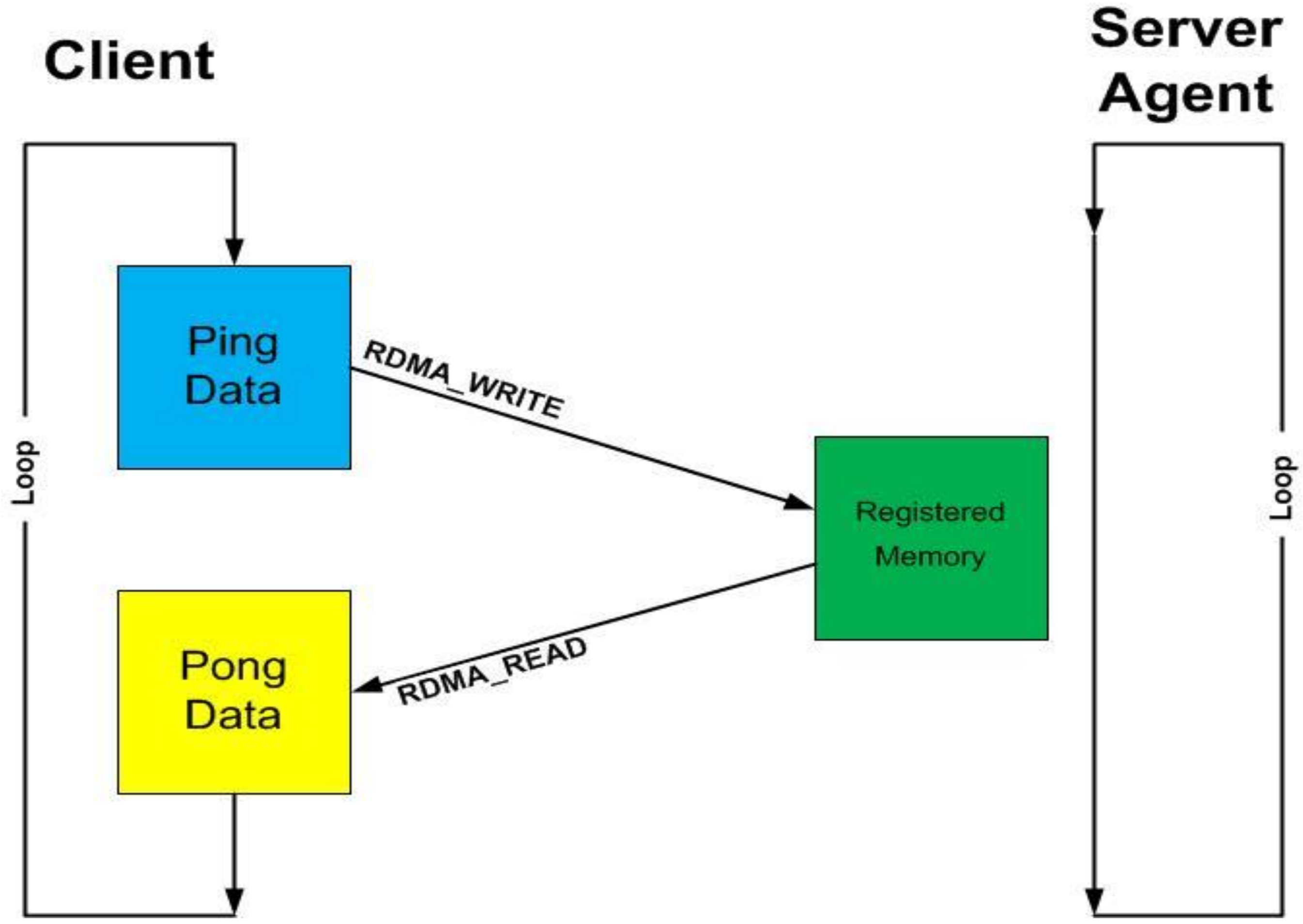
RDMA_READ data flow



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_SIGNALED** 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

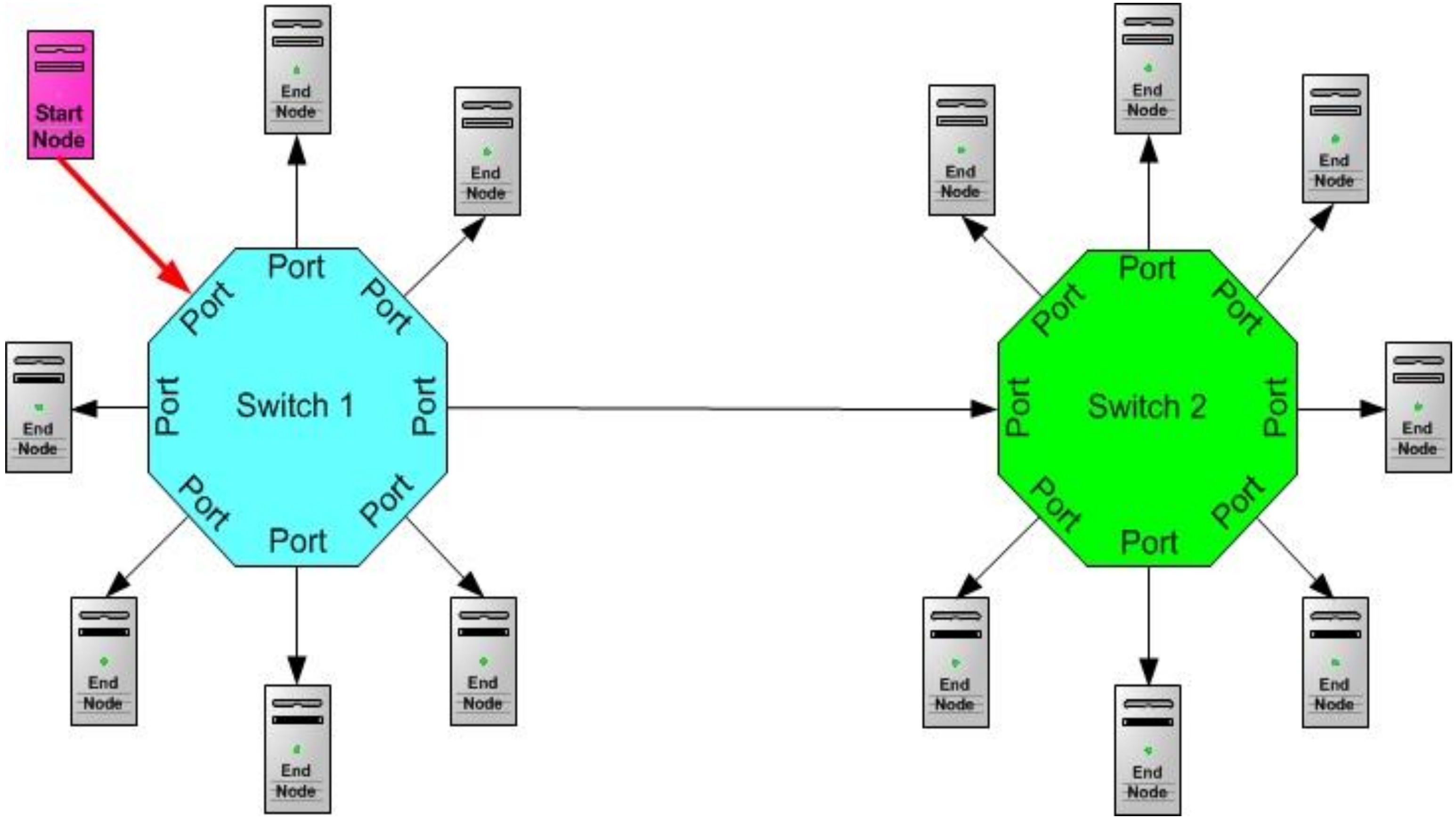
ping-pong RDMA_WRITE/READ performance with unsignaled, notify

- 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 unsignaled 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% unsignaled 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!