Introduction to RDMA Programming

Robert D. Russell <rdr@unh.edu>

InterOperability Laboratory &
Computer Science Department
University of New Hampshire
Durham, New Hampshire 03824-3591, USA

RDMA – what is it?

❖ A (relatively) new method for interconnecting platforms in high-speed networks that overcomes many of the difficulties encountered with traditional networks such as TCP/IP over Ethernet.

- new standards
- new protocols
- new hardware interface cards and switches
- new software

Remote Direct Memory Access

- **R**emote
  - –data transfers between nodes in a network
- **D**irect
  - –no Operating System Kernel involvement in transfers
  - –everything about a transfer offloaded onto Interface Card
- **M**emory
  - –transfers between user space application virtual memory
  - –no extra copying or buffering
- **A**ccess
  - –send, receive, read, write, atomic operations

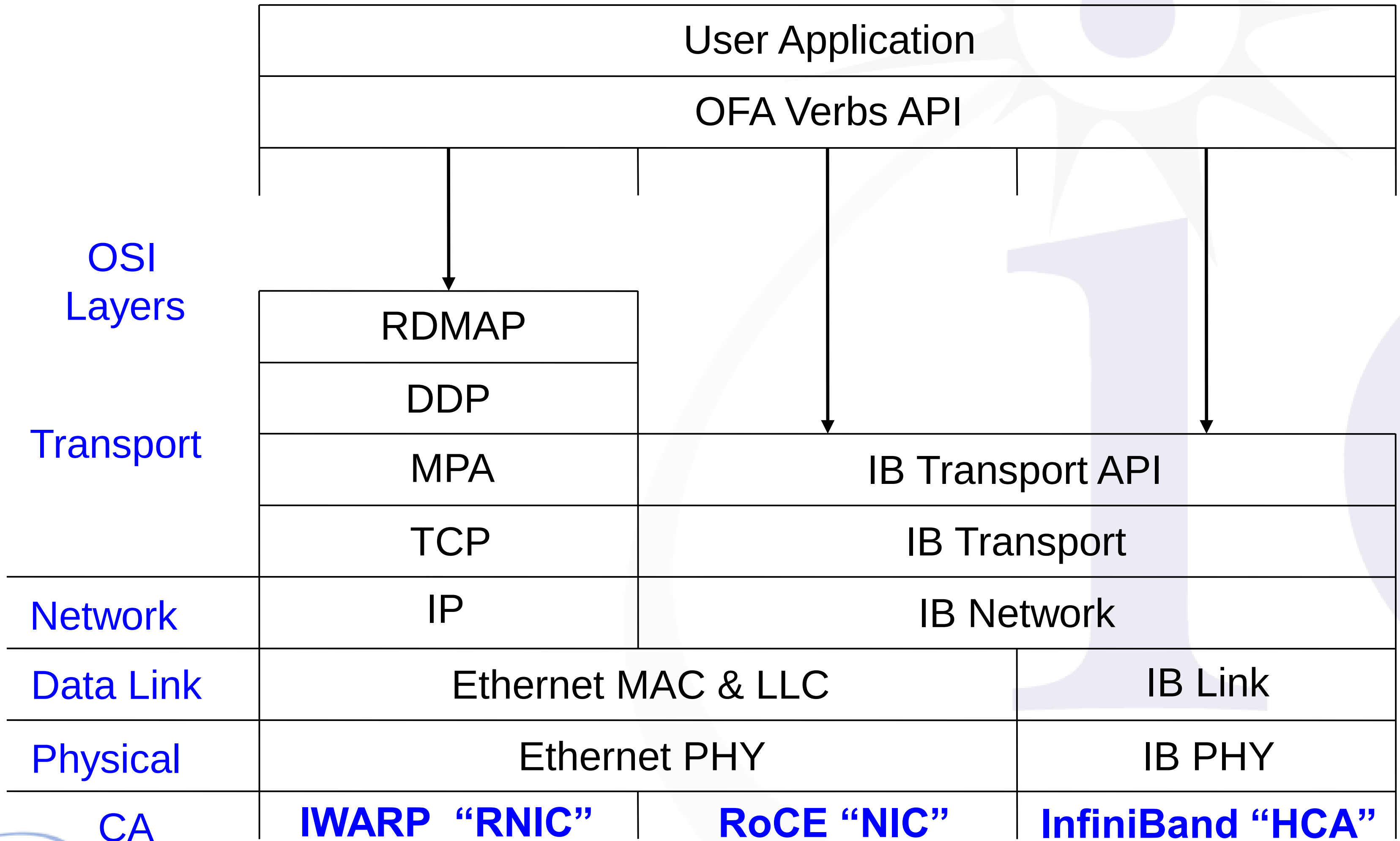
RDMA Benefits

- ❖High throughput
- ❖Low latency
- ❖High messaging rate
- ❖Low CPU utilization
- ❖Low memory bus contention
- ❖Message boundaries preserved
- ❖Asynchronous operation

RDMA Technologies

- InfiniBand – (41.8% 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
- RoCE – RDMA over Converged Ethernet
  - 10 Gbps
  - 40 Gbps

RDMA Architecture Layering



Software RDMA Drivers

- ❖ Softiwarp
  - – www.zurich.ibm.com/sys/rdma
  - – open source kernel module that implements iWARP protocols on top of ordinary kernel TCP sockets
  - – interoperates with hardware iWARP at other end of wire
- ❖ Soft RoCE
  - – www.systemfabricworks.com/downloads/roce
  - – open source IB transport and network layers in software over ordinary Ethernet
  - – interoperates with hardware RoCE at other end of wire

Verbs

- ❖ InfiniBand specification written in terms of verbs
  - – semantic description of required behavior
  - – no syntactic or operating system specific details
  - – implementations free to define their own API
    - • syntax for functions, structures, types, etc.
- ❖ OpenFabrics Alliance (OFA) Verbs API
  - – one possible syntactic definition of an API
  - – in syntax, each “verb” becomes an equivalent “function”
  - – done to prevent proliferation of incompatible definitions
  - – was an OFA strategy to unify InfiniBand market

OFA Verbs API

- ❖Implementations of OFA Verbs for Linux, FreeBSD, Windows
- ❖Software interface for applications
  - – data structures, function prototypes, etc. that enable C/C++ programs to access RDMA
- ❖User-space and kernel-space variants
  - – most applications and libraries are in user-space
- ❖Client-Server programming model
  - – some obvious analogies to TCP/IP sockets
  - – many differences because RDMA differs from TCP/IP

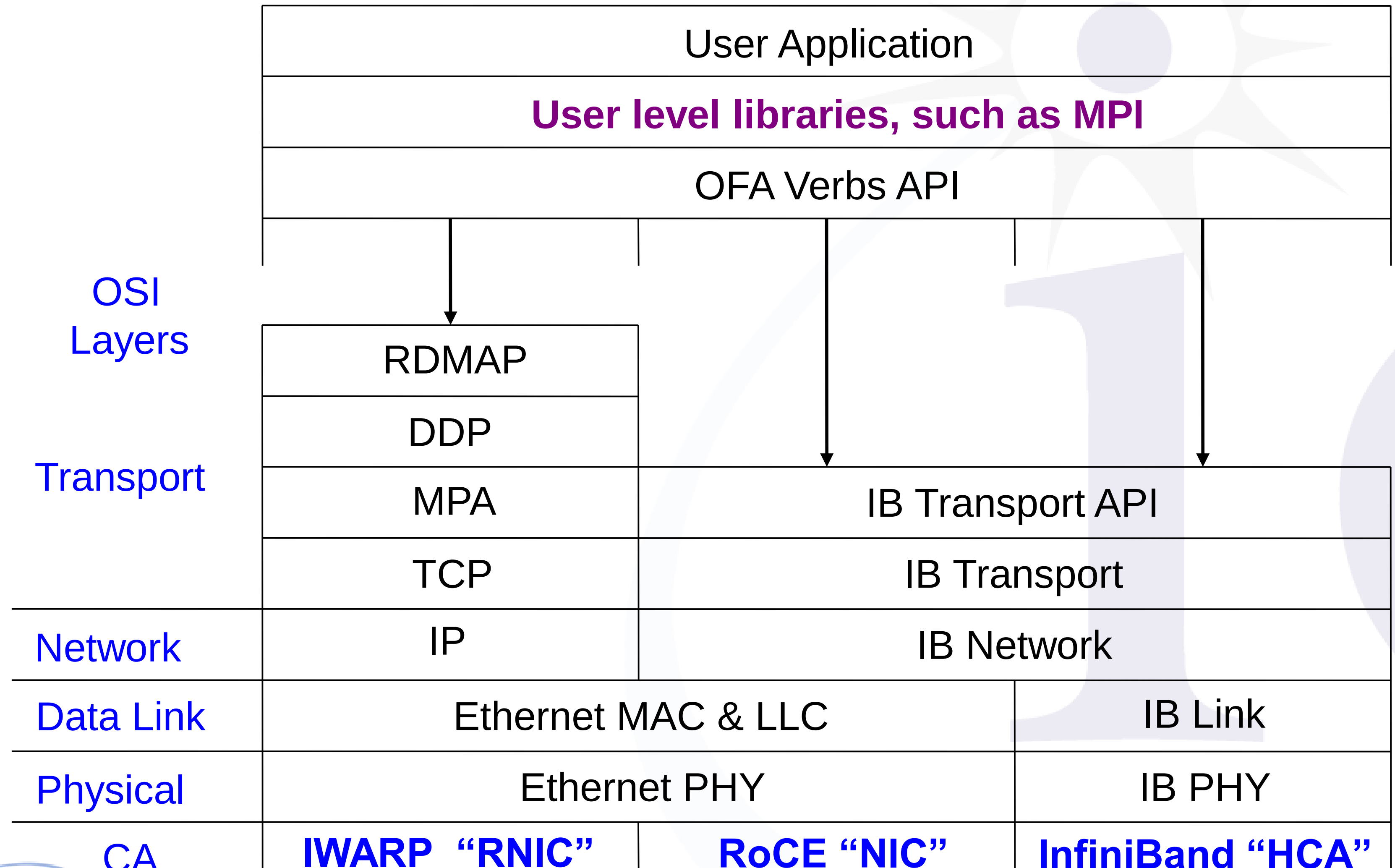
Users of OFA Verbs API

- Applications
- Libraries
- File Systems
- Storage Systems
- Other protocols

Libraries that access RDMA

- ❖ MPI – Message Passing Interface
  - –Main tool for High Performance Computing (HPC)
    - –Physics, fluid dynamics, modeling and simulations
  - –Many versions available
    - •OpenMPI
    - •MVAPICH
    - •Intel MPI

Layering with user level libraries



Additional ways to access RDMA

File systems

- Lustre – parallel distributed file system for Linux
- NFS_RDMA – Network File System over RDMA

Storage appliances by DDN and NetApp

- SRP – SCSI RDMA (Remote) Protocol – Linux kernel
- iSER – iSCSI Extensions for RDMA – Linux kernel

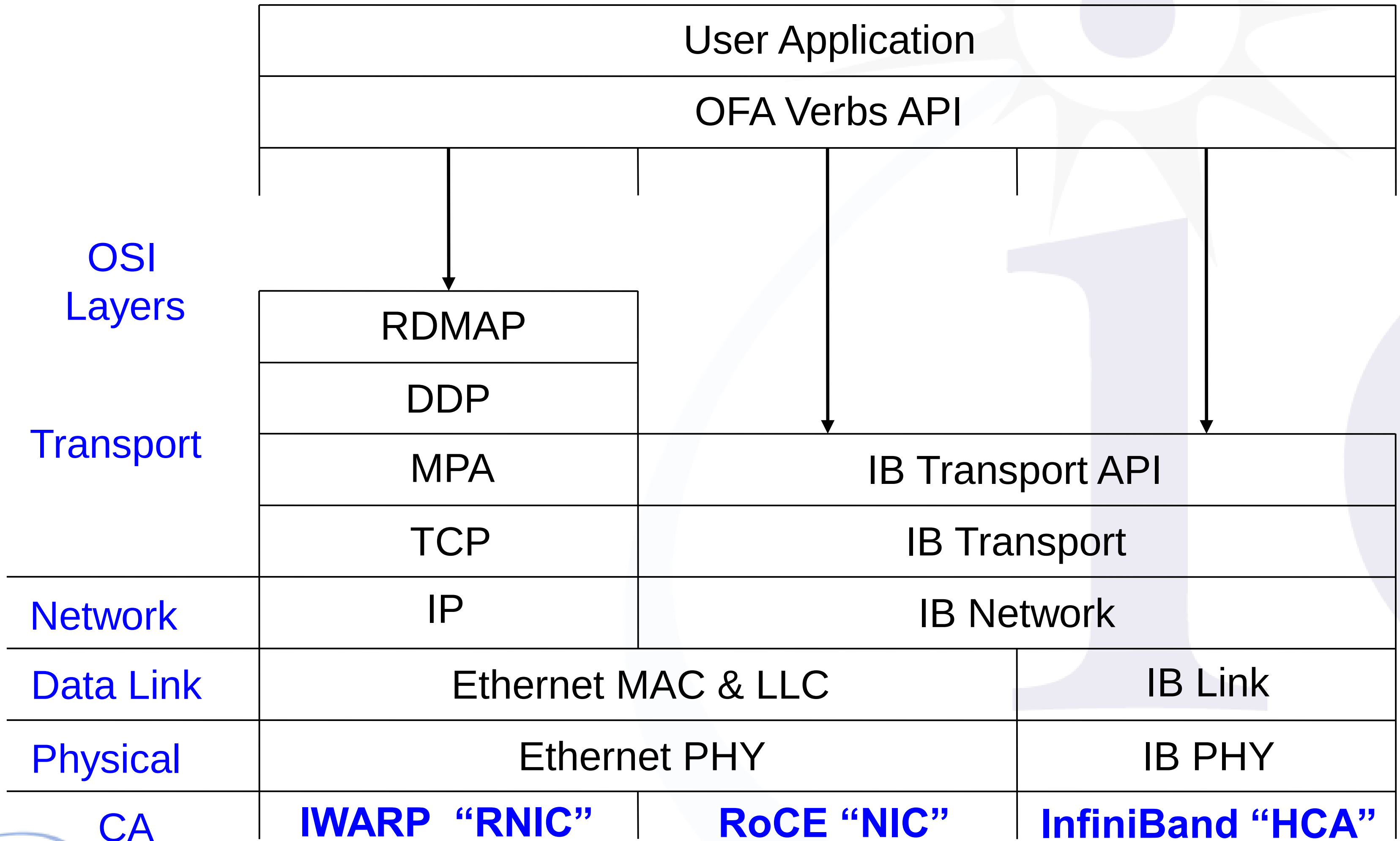
Additional ways to access RDMA

Pseudo sockets libraries

- SDP – Sockets Direct Protocol – supported by Oracle
- rsockets – RDMA Sockets – supported by Intel
- mva – Mellanox Messaging Accelerator
- SMC-R – proposed by IBM

All these access methods written on top of OFA verbs

RDMA Architecture Layering



Similarities between TCP and RDMA

- ❖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**

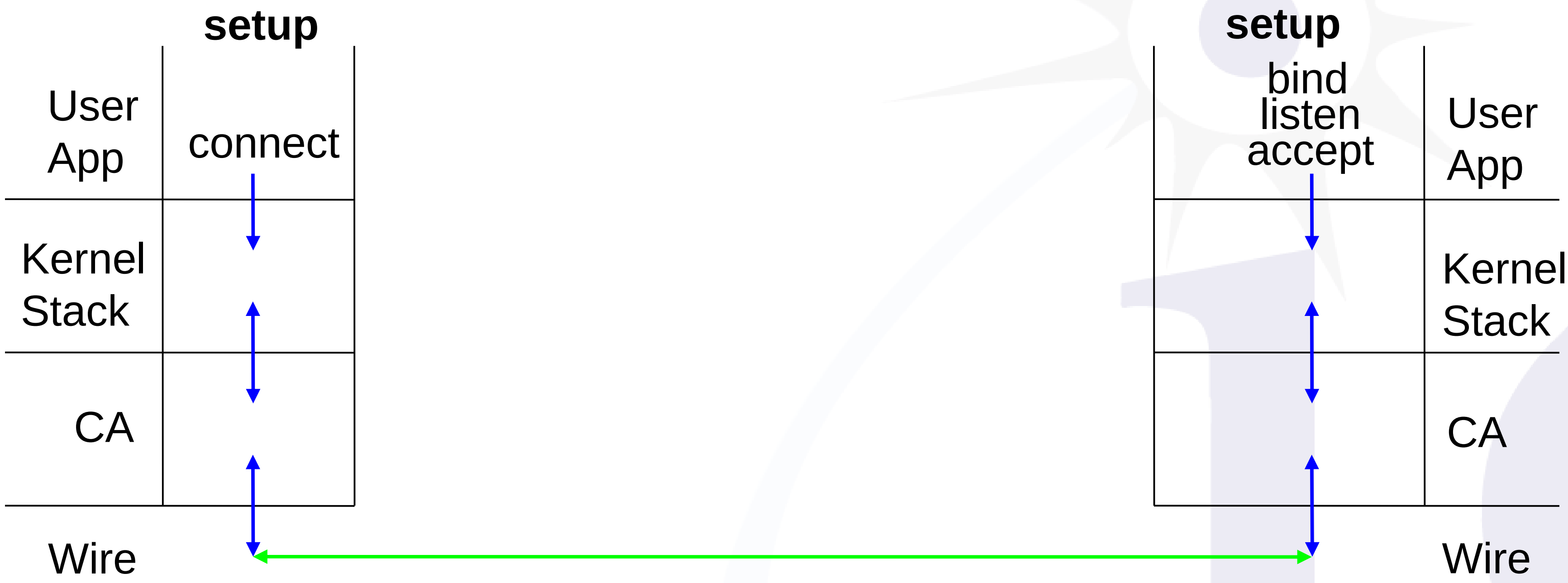
How RDMA differs from TCP/IP

- “zero copy” – data transferred directly from virtual memory on one node to virtual memory on another node
- “kernel bypass” – no operating system involvement during data transfers
- asynchronous operation – threads not blocked during I/O transfers

TCP/IP setup

client

server

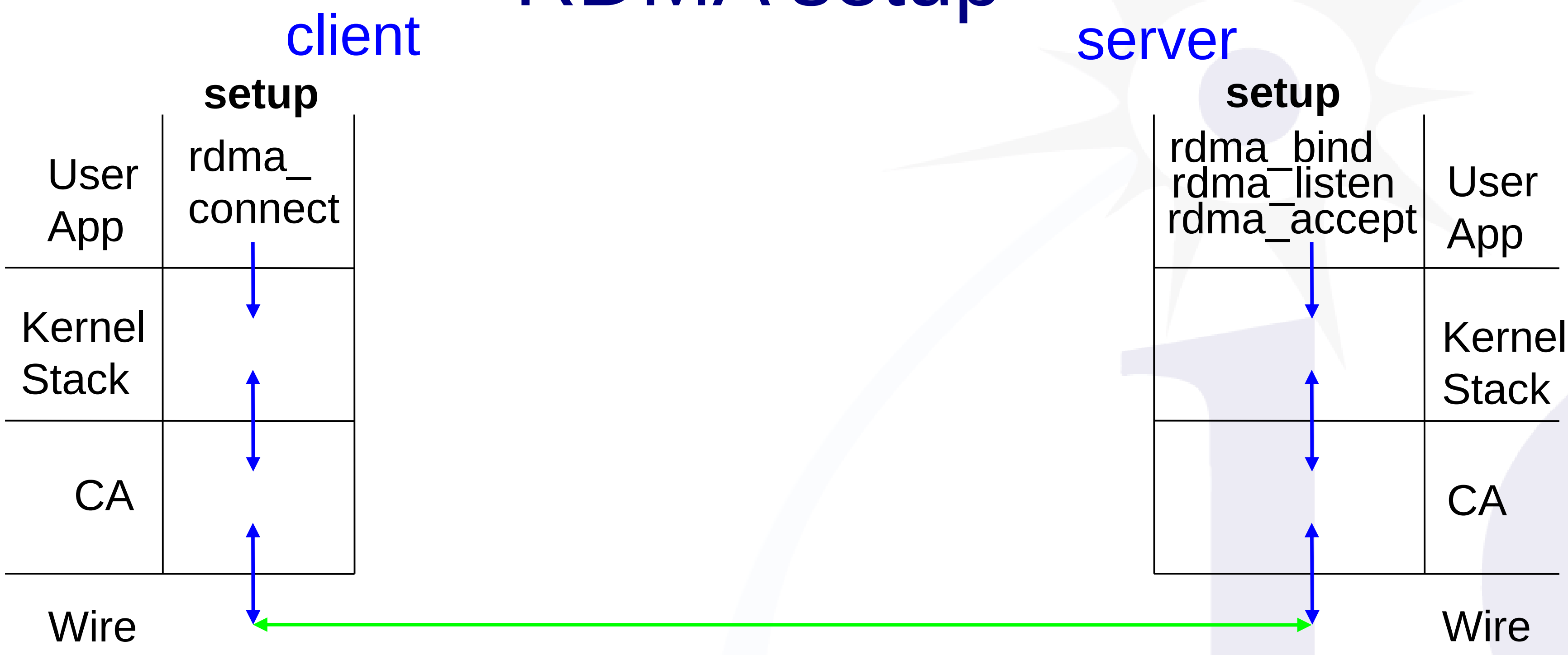


blue lines: control information

red lines: user data

green lines: control and data

RDMA setup

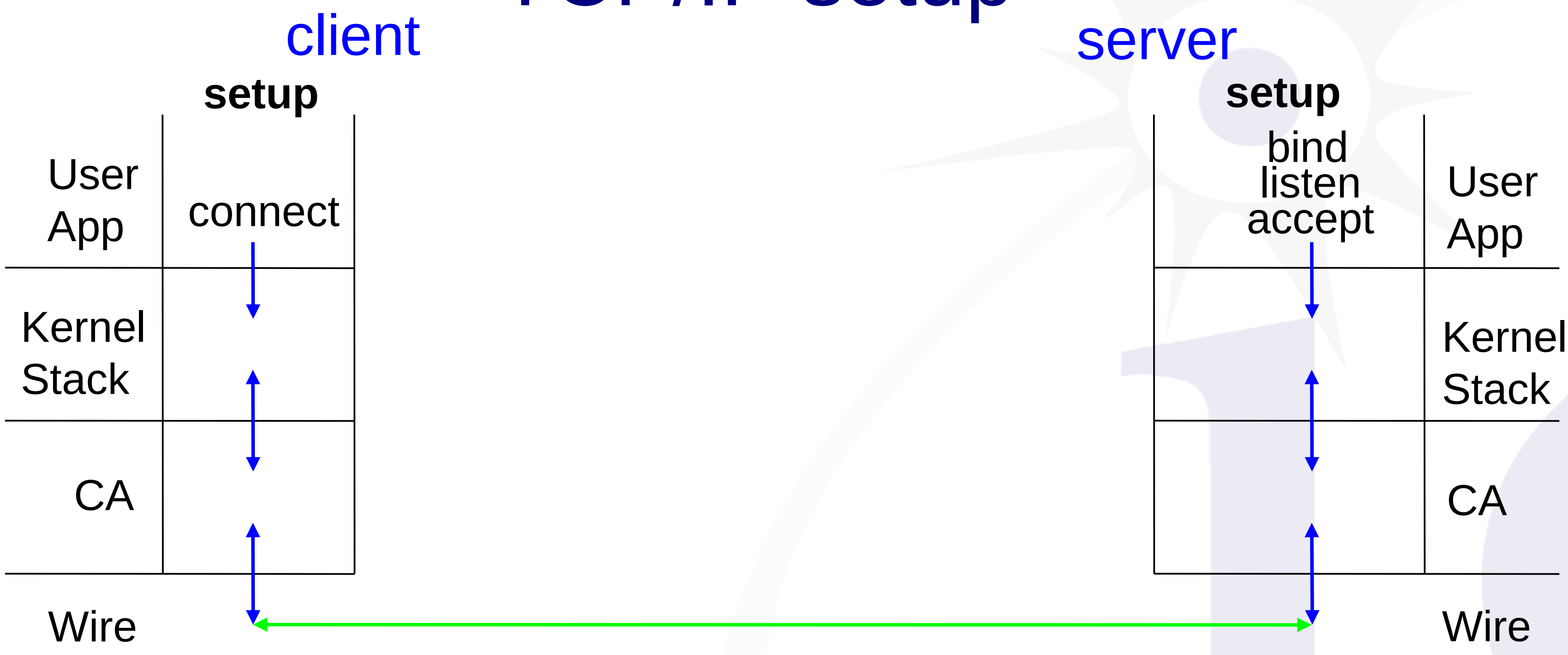


blue lines: control information

red lines: user data

green lines: control and data

TCP/IP setup

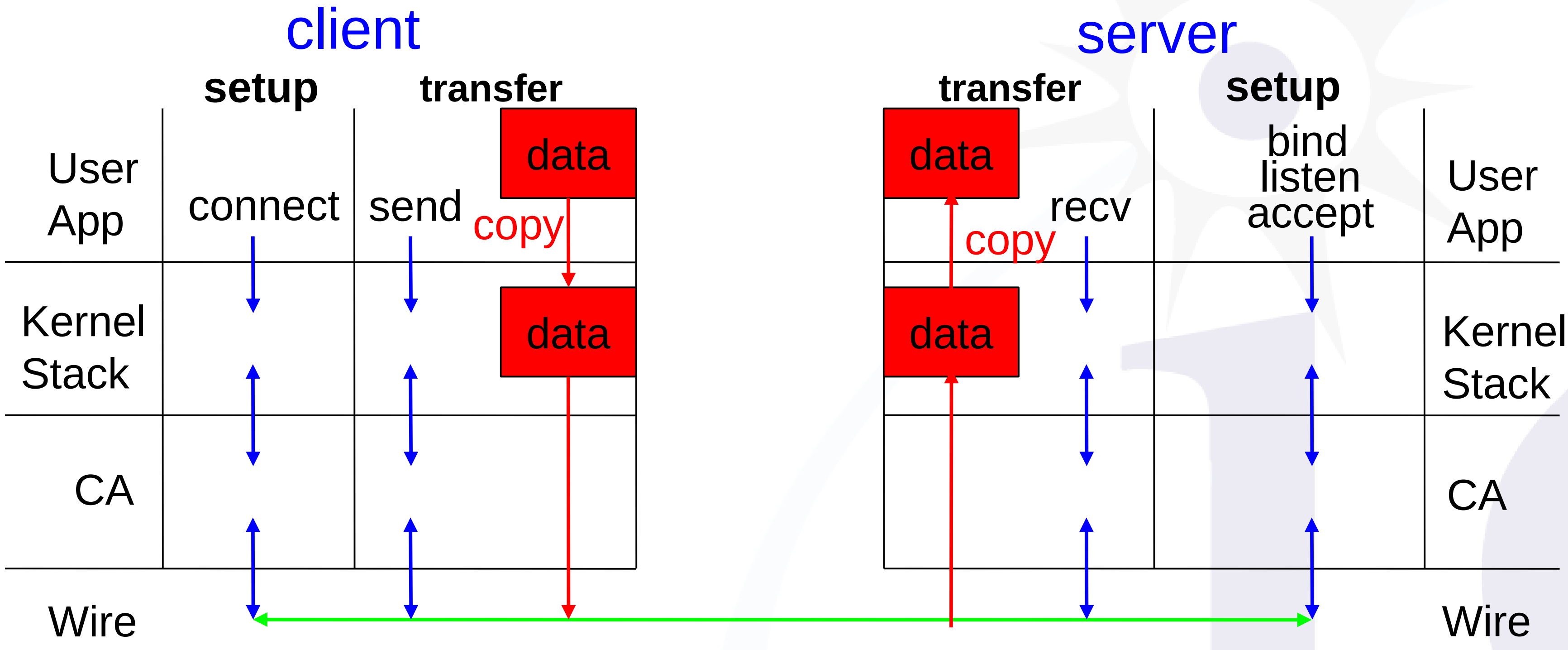


blue lines: control information

red lines: user data

green lines: control and data

TCP/IP transfer

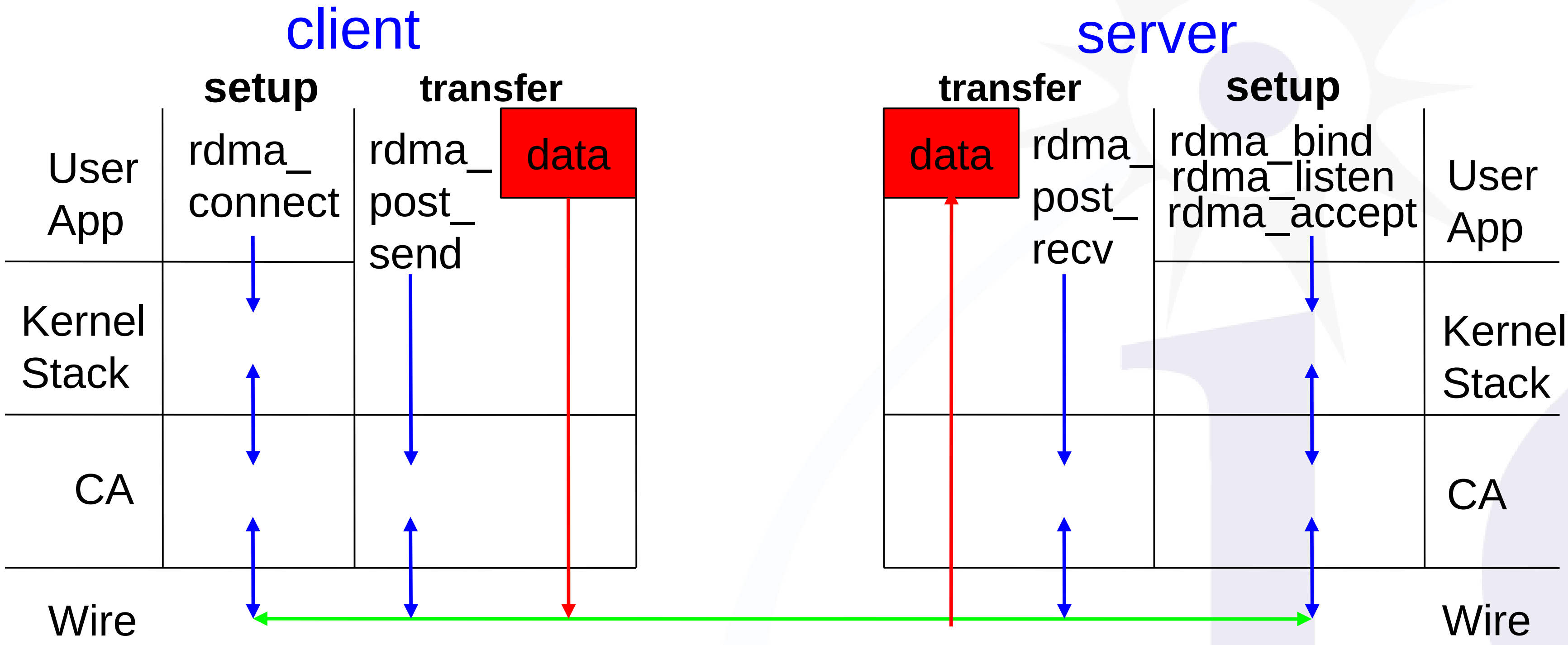


blue lines: control information

red lines: user data

green lines: control and data

RDMA transfer



blue lines: control information

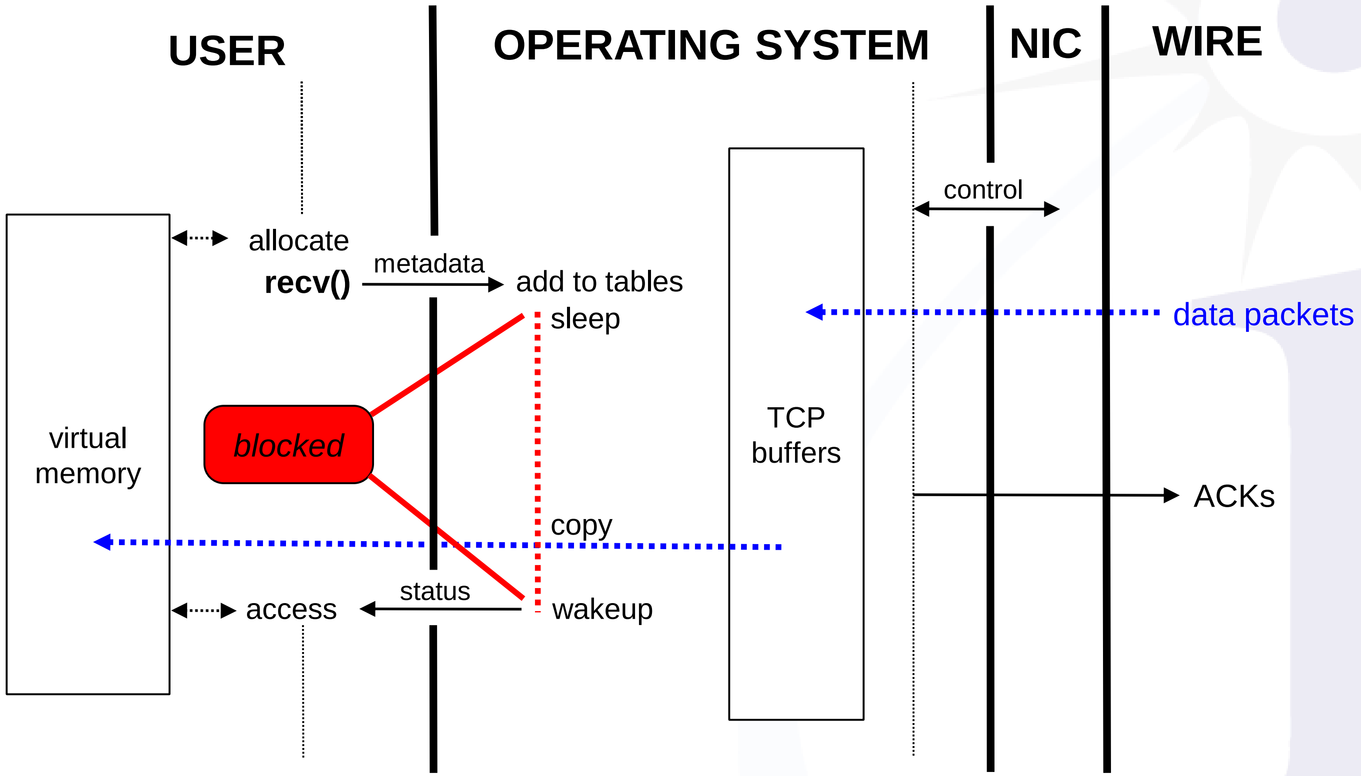
red lines: user data

green lines: control and data

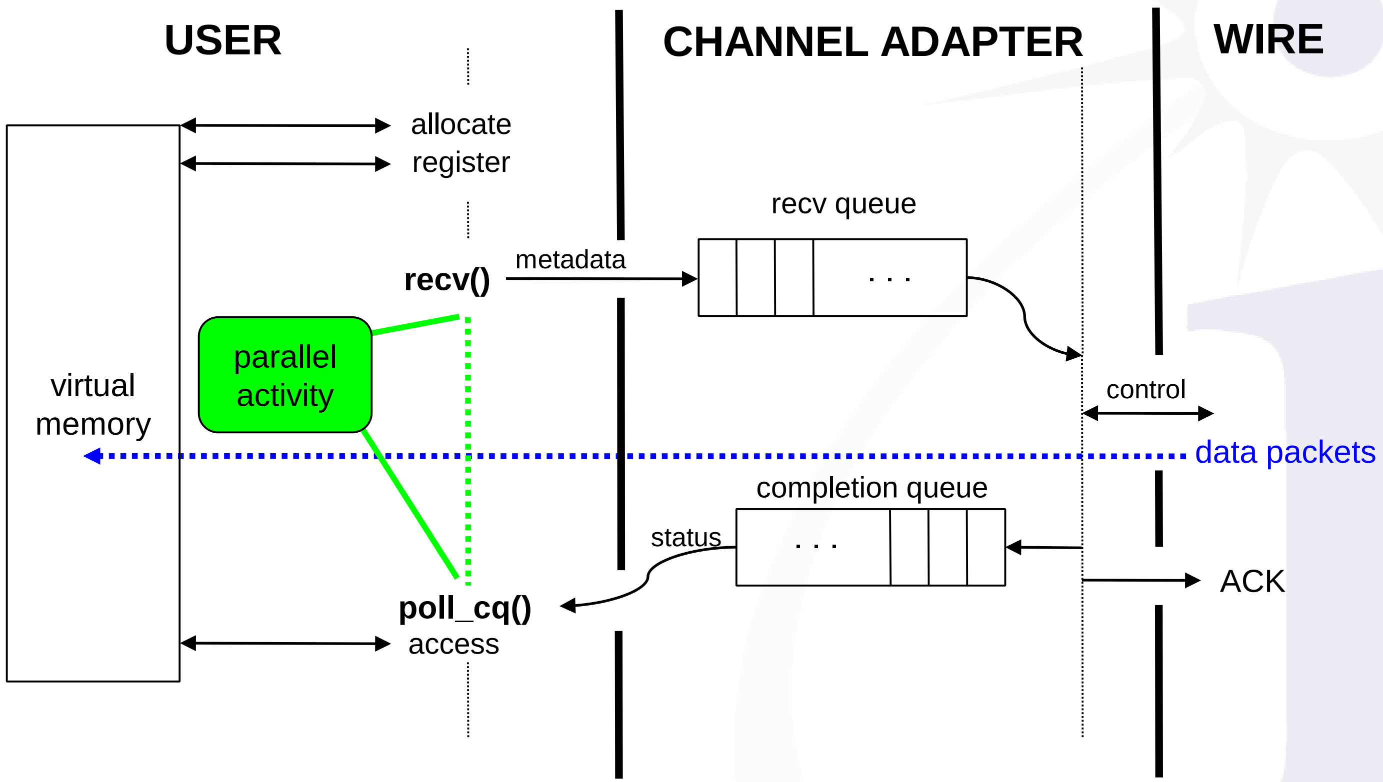
“Normal” TCP/IP socket access model

- Byte streams – requires application to delimit / recover message boundaries
- Synchronous – blocks until data is sent/received
  - O_NONBLOCK, MSG_DONTWAIT are **not** asynchronous, are “try” and “try again”
- send() and recv() are paired
  - both sides must participate in the transfer
- Requires data copy into system buffers
  - order and 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
virtual memory
allocate
recv()
metadata
add to tables
sleep
blocked
copy
access
status
wakeup
TCP buffers
control
data packets
ACKs

RDMA RECV()


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

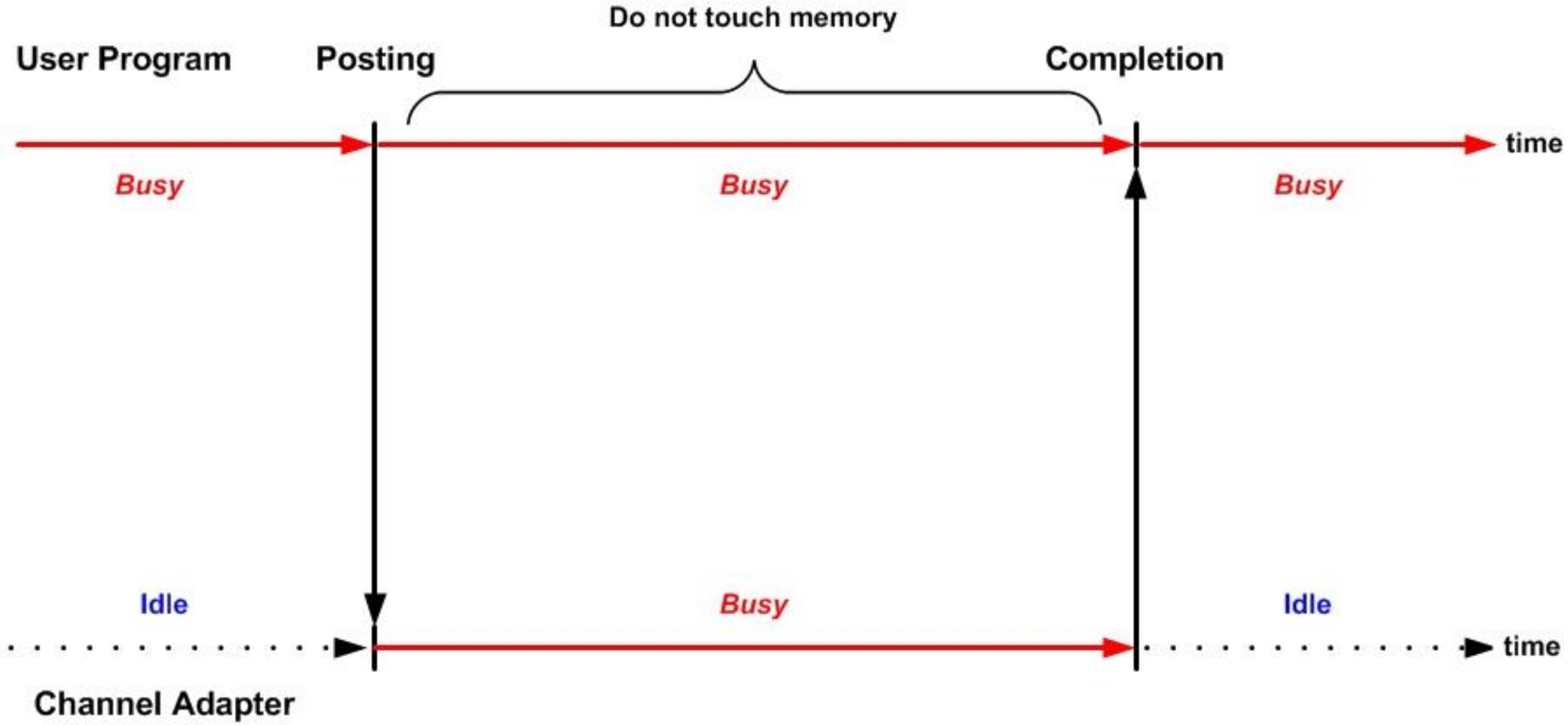
RDMA access model

- ❖Messages – preserves user'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 and timing of send() and recv() are **relevant**
    - •recv() must be waiting before issuing send()
  - –memory involved in transfer is **untouchable** between start and completion of transfer

Asynchronous Data Transfer

- ❖ Posting
  - term used to mark the initiation of a data transfer
  - done by adding a work request to a work queue
- ❖ Completion
  - term used to mark the end of a data transfer
  - done by removing a work completion from completion queue
- ❖ Important note:
  - between posting and completion the state of user memory involved in the transfer is undefined and should NOT be changed by the user program

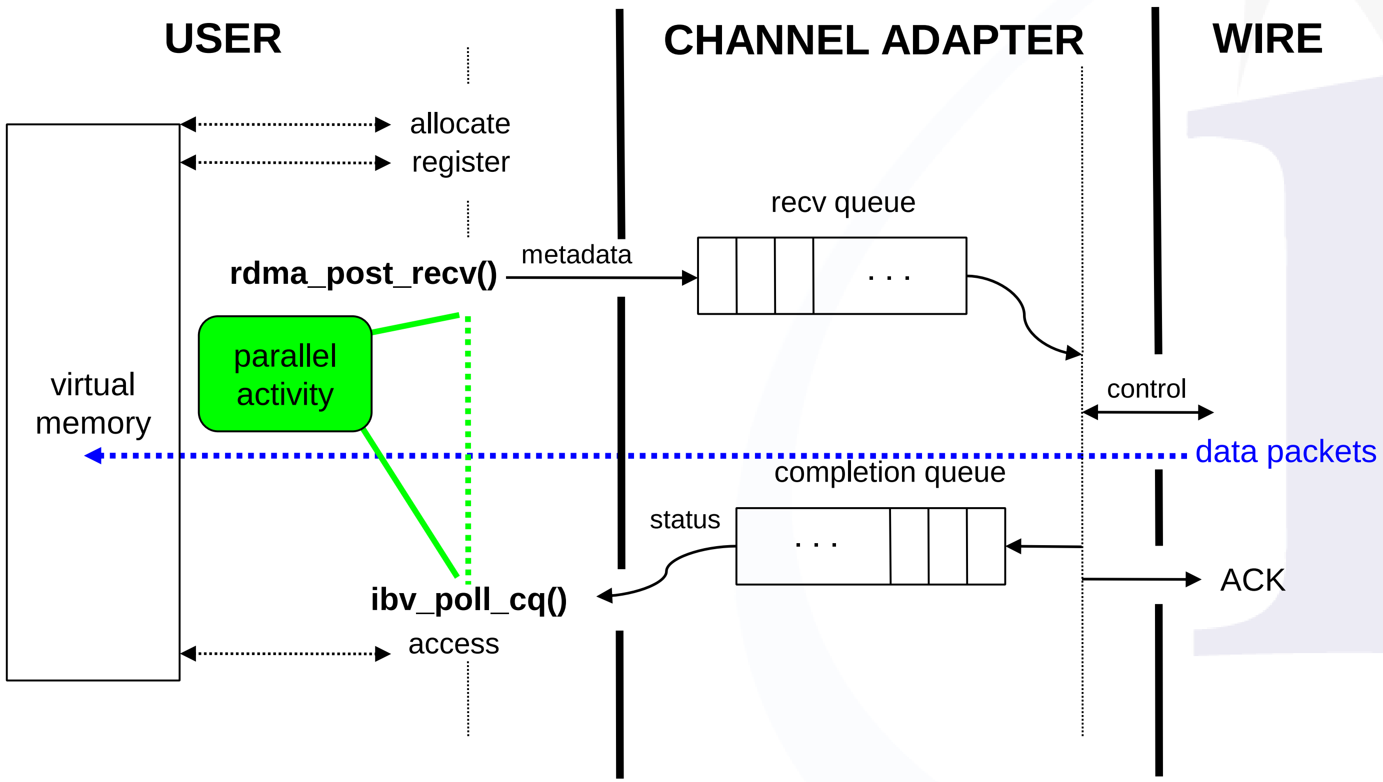
Posting – Completion



Kernel Bypass

- ❖User interacts directly with CA queues
- ❖Queue Pair from program to CA
  - – work request – data structure describing data transfer
  - – send queue – post work requests to CA that send data
  - – secv queue – post work requests to CA that receive data
- ❖Completion queues from CA to program
  - – work completion – data structure describing transfer status
  - – Can have separate send and receive completion queues
  - – Can have one queue for both send and receive completions

RDMA recv and completion queues



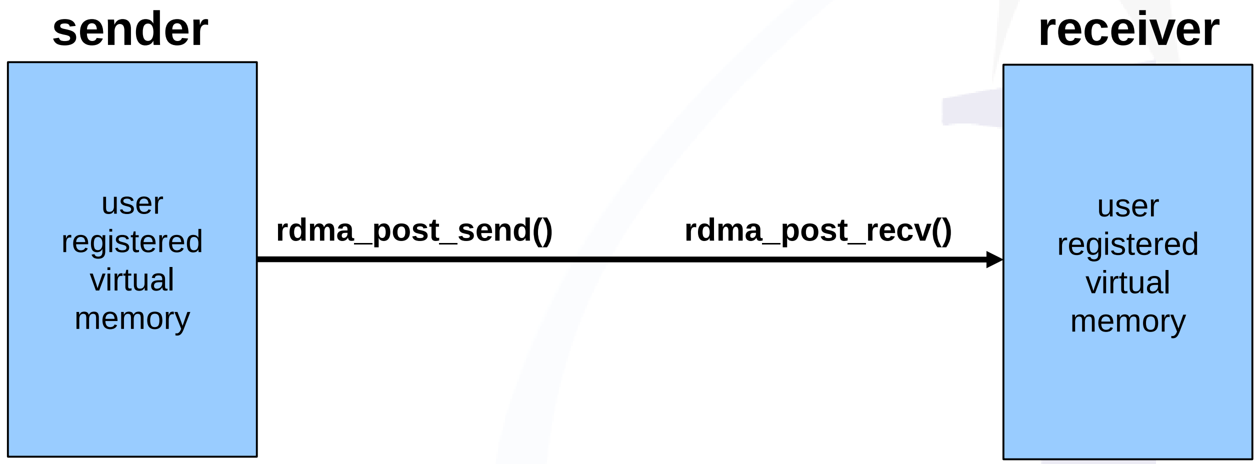
RDMA memory **must** be registered

- To "pin" it into physical memory
  - so it 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
- To create "keys" linking memory, process, and CA
  - supplied by user as part of every transfer
  - allows user to control access rights of a transfer
  - allows CA to find correct mapping in a transfer
  - allows CA to verify access rights in a transfer

RDMA transfer types

- SEND/RECV – similar to “normal” TCP sockets
  – each send on one side must match a recv on other side
- WRITE – only in RDMA
  – “pushes” data into remote virtual memory
- READ – only in RDMA
  – “pulls” data out of remote virtual memory
- Atomics – only in InfiniBand and RoCE
  – updates cell in remote virtual memory
- Same verbs and data structures used by all

RDMA SEND/RECV data transfer



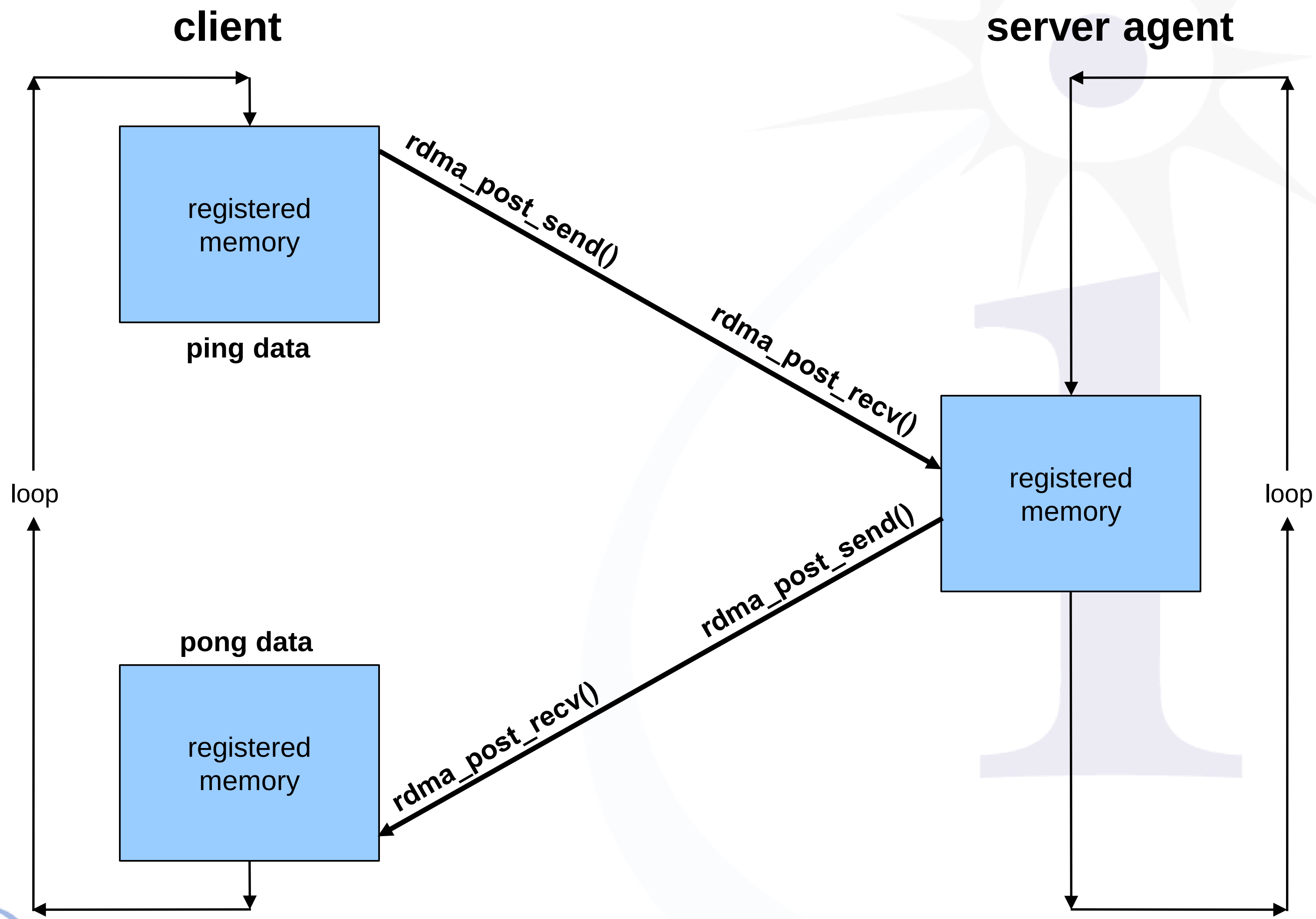
SEND/RECV similarities with sockets

- 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 sockets are buffered
  - time order of send() and recv() on each side is irrelevant
- RDMA sockets are not buffered
  - recv() must be posted by receiver before send() can be posted by sender
  - not doing this results in a fatal error
- “normal” TCP/IP sockets have no notion of “memory registration”
- RDMA sockets require that memory participating be “registered”

ping-pong using RDMA SEND/RECV



3 phases in using reliable connections

- Setup Phase
  - obtain and convert addressing information
  - create and configure local endpoints for communication
  - setup local memory to be used in transfer
  - establish the connection with the remote side
- Use Phase
  - actually transfer data to/from the remote side
- Break-down Phase
  - basically “undo” the setup phase
  - close connection, free memory and communication 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.		define properties of new queue pair struct ibv_qp_init_attr
4.	create local end point socket()	create local end point rdma_create_ep()
5.	allocate user virtual memory malloc()	allocate user virtual memory malloc()
6.		register user virtual memory with CA rdma_reg_msgs()
7.		define properties of new connection struct rdma_conn_param
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.		wait for send to complete ibv_poll_cq()
14.	receive pong data from agent recv()	wait for receive to complete ibv_poll_cq()
15.	optionally verify pong data is ok memcmp()	optionally verify pong data is ok memcmp()
16.	end of transfer loop	end of transfer loop
17.	mark stop time and print statistics	mark stop time and print statistics

Client breakdown phase

	TCP	RDMA
18.	break connection with server close()	break connection with server rdma_disconnect()
19.		deregister user virtual memory rdma_dereg_mr()
20.	free user virtual memory free()	free user virtual memory free()
21.		destroy local end point rdma_destroy_ep()
22.	free getaddrinfo resources freeaddrinfo()	free rdma_getaddrinfo resources rdma_freeaddrinfo()
23.	“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()	define properties of new queue pair struct ibv_qp_init_attr
4.	bind to address and port bind()	create and bind local end point rdma_create_ep()
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

Listen 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 and new cm_id for client
2.	allocate user virtual memory malloc()	allocate user virtual memory malloc()
3.		register user virtual memory with CA rdma_reg_msgs()
4.		post first receive of ping data from client rdma_post_recv()
5.		define properties of new connection struct rdma_conn_param
6.		finalize connection with client rdma_accept()

Agent use phase

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

Agent breakdown phase

	TCP	RDMA
15.	break connection with client close()	break connection with client rdma_disconnect()
16.		deregister user virtual memory rdma_dereg_mr()
17.	free user virtual memory free()	free user virtual memory free()
18.	free copy of listener's options	free copy of listener's options
19.		destroy local end point rdma_destroy_ep()

ping-pong measurements

- 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
- InfiniBand QDR 4x through a switch

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 a completion
  - (i.e., gives low latency)

”busy polling” to get completions

1. start loop

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

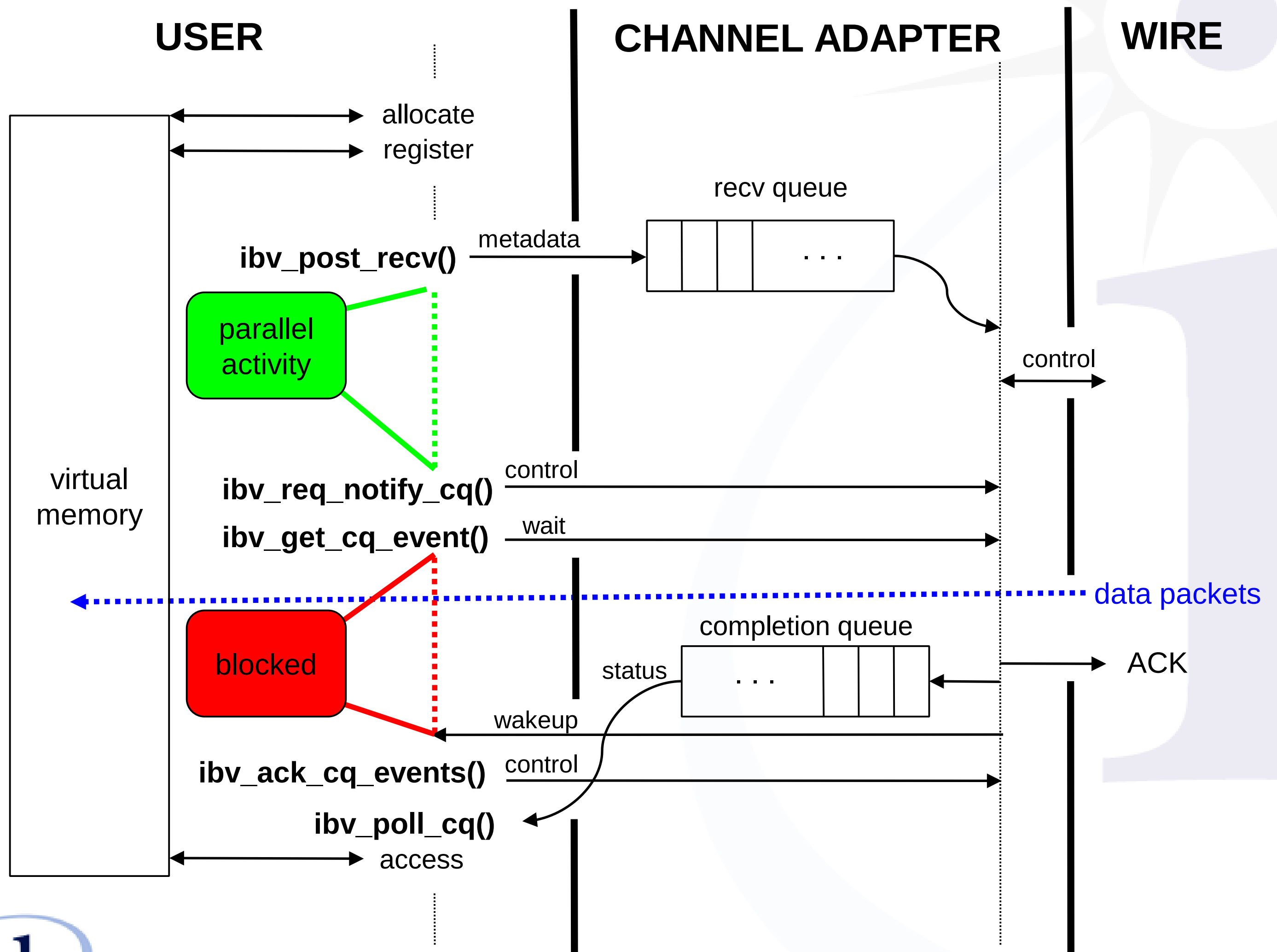
3. exit loop if a completion is found

4. end loop

How to eliminate “busy polling”

- Cannot make **ibv_poll_cq()** block
  - no flag parameter
  - no timeout parameter
- Must replace busy loop with “wait – wakeup”
- Solution is a “wait-for-event” mechanism
  - **ibv_req_notify_cq()** - tell CA to send an “event” when next WC enters CQ
  - **ibv_get_cq_event()** - blocks until gets “event”
  - **ibv_ack_cq_event()** - acknowledges “event”

OFA verbs API for recv wait-wakeup



”wait-for-event” to get completions

1. start loop
2. **ibv_poll_cq()** to get any completion in CQ
3. exit loop if a completion is found
4. **ibv_req_notify_cq()** to arm CA to send event on next completion added to CQ
5. **ibv_poll_cq()** to get new completion between 2&4
6. exit loop if a completion is found
7. **ibv_get_cq_event()** to wait until CA sends event
8. **ibv_ack_cq_events()** to acknowledge event
9. end loop

ping-pong measurements with wait

- Client
  - round-trip-time 21.1 microseconds – up 34%
  - user CPU time 9.0% of elapsed time
  - kernel CPU time 9.1% of elapsed time
  - 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
  - kernel CPU time 6.5% of elapsed time
  - total CPU time 21% of elapsed time – down 79%

rdma_xxxx "wrappers" around ibv_xxxx

- **rdma_get_recv_comp()** - wrapper for wait-wakeup loop on receive completion queue
- **rdma_get_send_comp()** - wrapper for wait-wakeup 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, etc.
- /usr/include/rdma/rdma_cm.h
  - contains rdma_yyyy verbs and all rdma data structures, etc. for connection management

Transfer choices

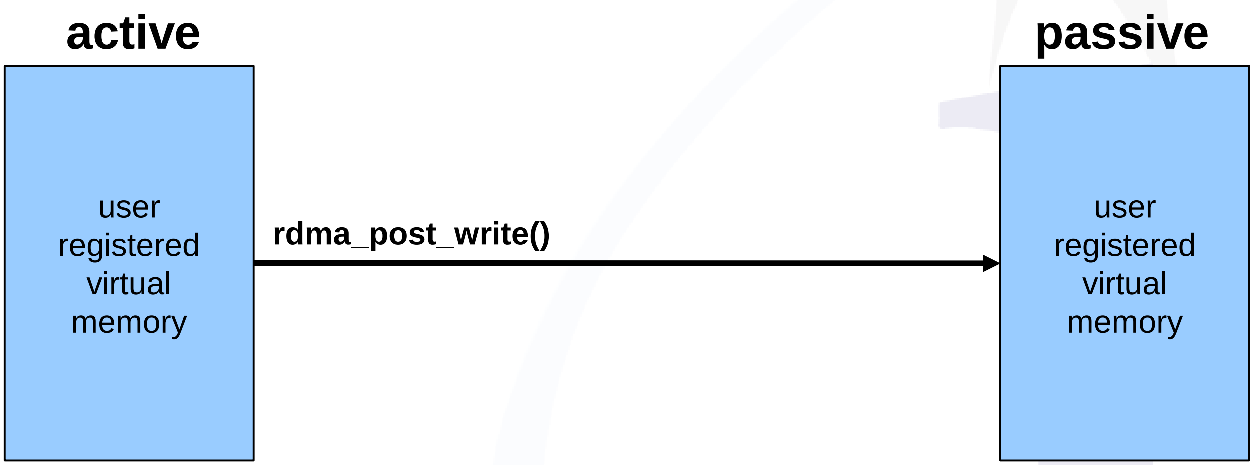
- TCP/UDP transfer operations
  - –send()/recv() (and related forms)

- 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 passive side 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 sides' virtual memory
  - passive side issues no operation, uses no CPU cycles, gets no indication “push” started or completed

RDMA WRITE data transfer



Differences with RDMA SEND

- Active side calls **rdma_post_write()**
  - 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
- ❖Both transfer types require registered virtual memory on both sides of the 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

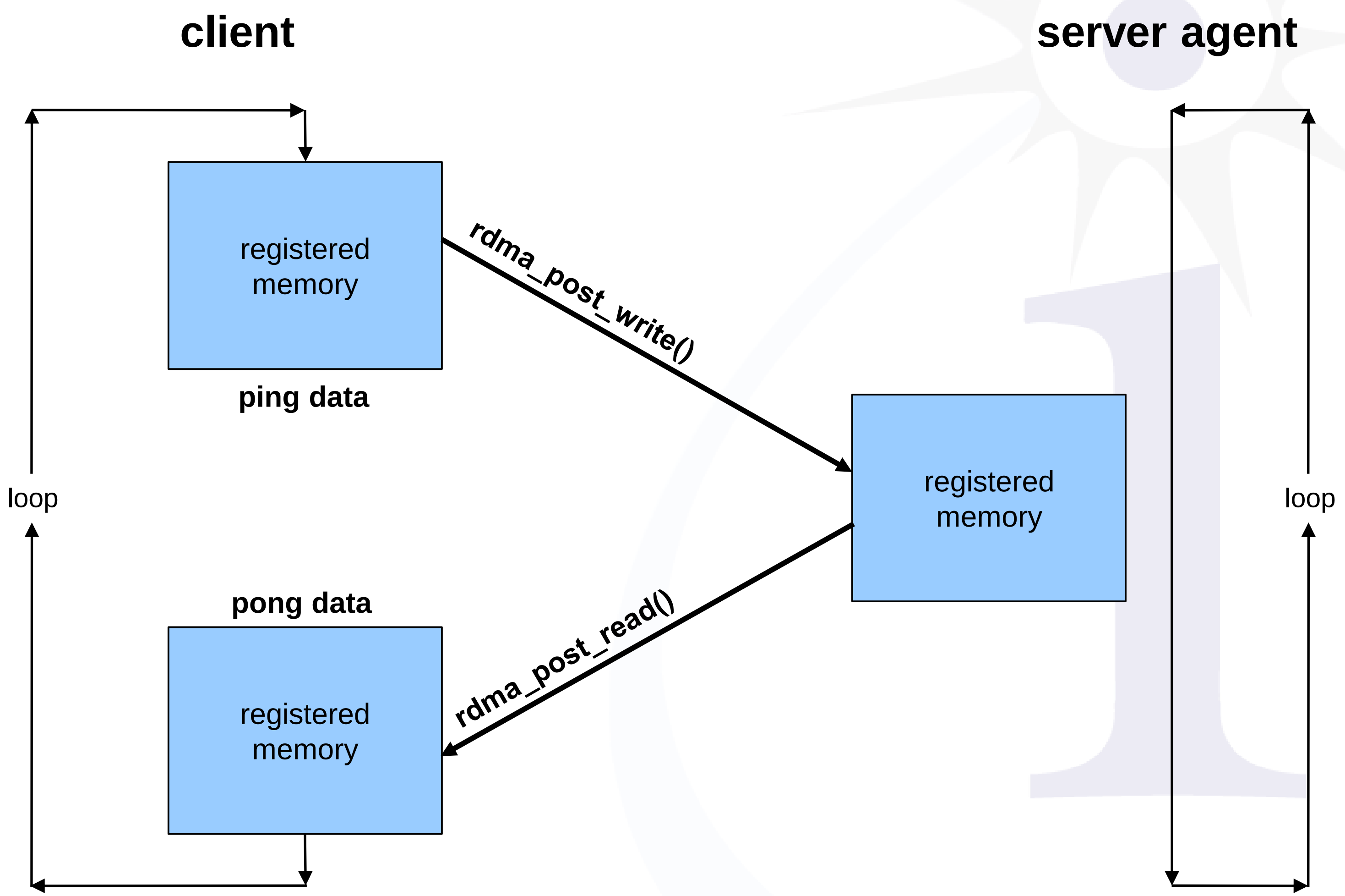
- ❖Very different from normal TCP/IP
- ❖Very different from RDMA SEND/RECV
- ❖Only one side is active, other is passive
- ❖Active side (requester) issues RDMA READ
- ❖Passive side (responder) does NOTHING!
- ❖A better name would be “RDMA PULL”
  - – data is “pulled” into active side's virtual memory from passive sides' virtual memory
  - – passive side issues no operation, uses no CPU cycles, gets no indication “pull” started or completed

RDMA READ data transfer

Ping-pong with RDMA WRITE/READ

- Client is active side in ping-pong loop
  - client posts RDMA WRITE from 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's address and registration key to client before loop
- Client must send total number of transfers to agent after the loop
  - otherwise agent has no way of knowing this number
  - agent needs to receive something to tell it “loop finished”

ping-pong using RDMA WRITE/READ



Client ping-pong transfer loop

- start of transfer loop
- **rdma_post_write()** of RDMA WRITE ping data
- **ibv_poll_cq()** to wait for RDMA WRITE completion
- **rdma_post_read()** 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

- **ibv_poll_cq()** to wait for “finished” from client

ping-pong RDMA WRITE/READ measurements with wait

❖ Client
  - round-trip-time 14.3 microseconds
  - user CPU time 26.4% of elapsed time
  - kernel CPU time 3.0% of elapsed time
  - total CPU time 29.4% of elapsed time

❖ Server
  - round-trip time 14.3 microseconds
  - user CPU time 0% of elapsed time
  - kernel CPU time 0% of elapsed time
  - total CPU time 0% of elapsed time

Improving performance further

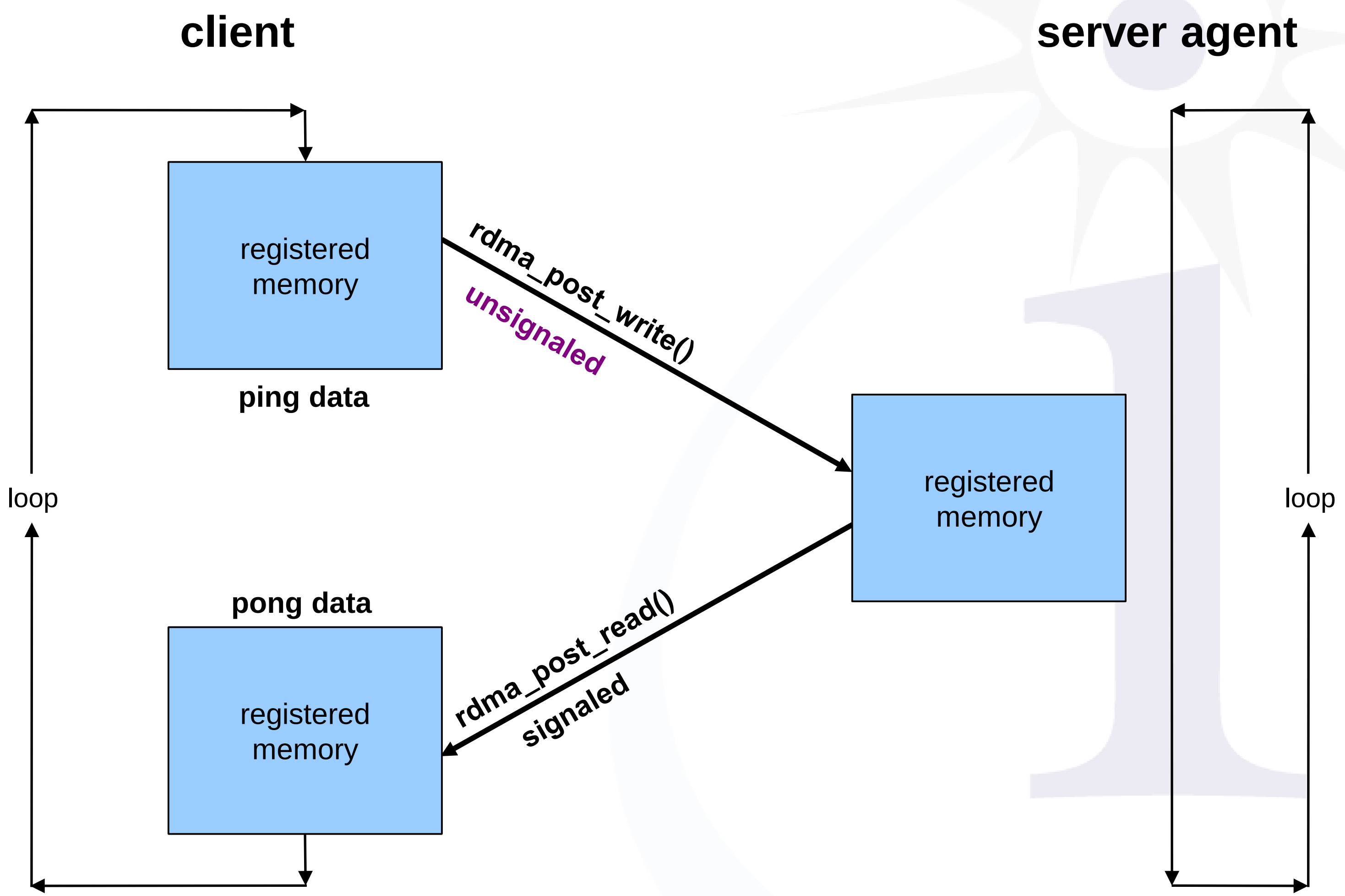
- ❖ All postings discussed so far generate completions
  - required for all **rdma_post_recv()** postings
  - optional for all other **prdma_post_xxxx()** postings

- ❖ User controls completion generation with **IBV_SEND_SIGNALED** flag in **rdma_post_write()**
  - 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

How client can benefit from this feature

- ❖RDMA READ posting follows RDMA WRITE
- ❖RDMA READ must finish after RDMA WRITE
  - – due to strict ordering rules in standards
- ❖Therefore we 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

ping-pong using unsignaled WRITE



Client unsignaled transfer loop

- start of transfer loop
- **rdma_post_write()** of **unsignaled** RDMA WRITE
  - generates no completion (except on error)
- **do not wait** for RDMA WRITE completion
- **rdma_post_read()** of **signaled** RDMA READ
- **ibv_poll_cq()** to wait for RDMA READ completion
- optionally verify pong data equals ping data
- end of transfer loop

ping-pong RDMA WRITE/READ measurements with unsignaled wait

- Client
  - round-trip-time 8.3 microseconds – down 42%
  - user CPU time 28.0% of elapsed time – up 6.1%
  - kernel CPU time 2.8% of elapsed time – down 6.7%
  - total CPU time 30.8% of elapsed time – up 4.8%
- Server
  - round-trip time 8.3 microseconds – down 42%
  - user CPU time 0% of elapsed time
  - kernel CPU time 0% of elapsed time
  - total CPU time 0% of elapsed time

Ping-pong performance summary

- ❖ Rankings for Round-Trip Time (RTT)
  - 8.3 usec unsignaled RDMA_WRITE/READ with wait
  - 14.3 usec signaled RDMA_WRITE/READ with wait
  - 15.7 usec signaled SEND/RECV with busy polling
  - 21.1 used signaled SEND/RECV with wait
- ❖ Rankings for client CPU usage
  - 18.0% signaled SEND/RECV with wait
  - 29.4% signaled RDMA_WRITE/READ with wait
  - 30.8% unsignaled RDMA_WRITE/READ with wait
  - 100% signaled SEND/RECV with busy polling

QUESTIONS?

Acknowledgments

❖This material is based upon work supported by the National Science Foundation under Grant No. OCI-1127228.

❖Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

THANK YOU!