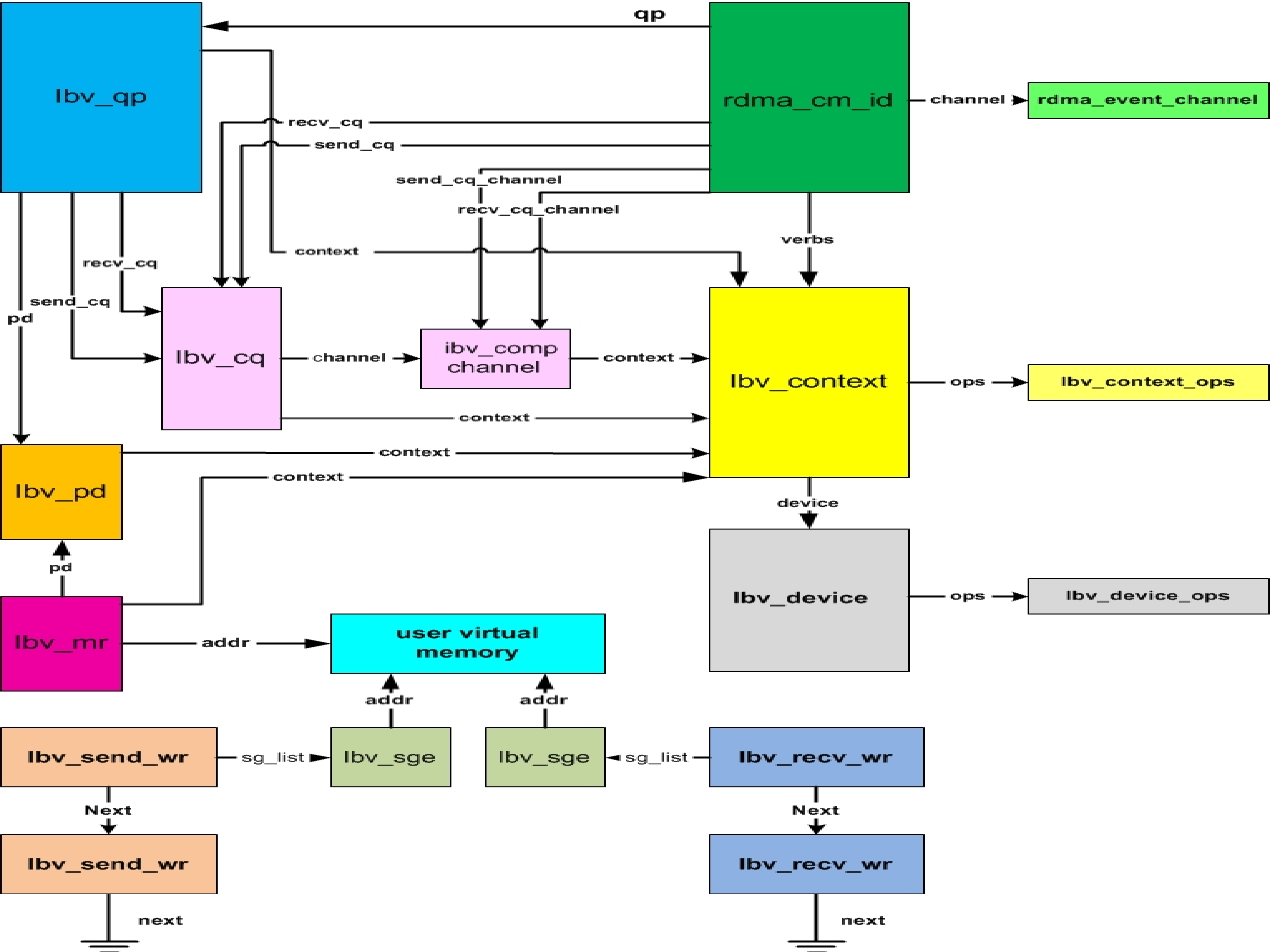
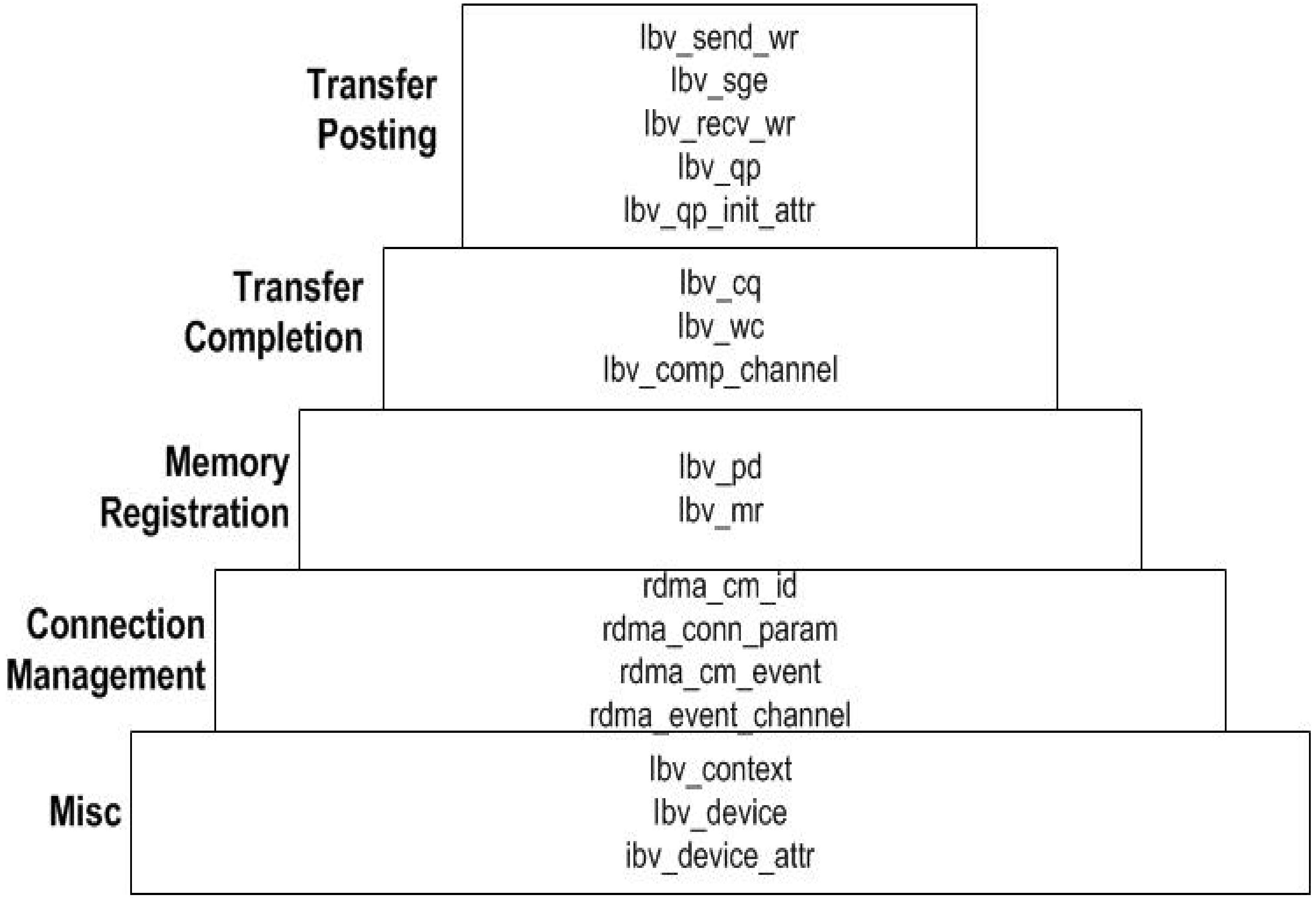
Writing Application Programs for RDMA using OFA Software Part 3

Open Fabrics Alliance

Copyright Statement


Ibv_qp
qp
rdma_cm_id
channel
rdma_event_channel
recv_cq
send_cq
send_cq_channel
recv_cq_channel
context
verbs
recv_cq
send_cq
pd
Ibv_cq
channel
ibv_comp channel
context
Ibv_context
ops
Ibv_context_ops
context
context
Ibv_pd
context
device
pd
Ibv_device
ops
Ibv_device_ops
Ibv_mr
addr
user virtual memory
addr
addr
Ibv_send_wr
sg_list
Ibv_sge
Ibv_sge
sg_list
Ibv_recv_wr
Next
Next
Ibv_send_wr
Ibv_recv_wr
next
next

Data Structures Pyramid



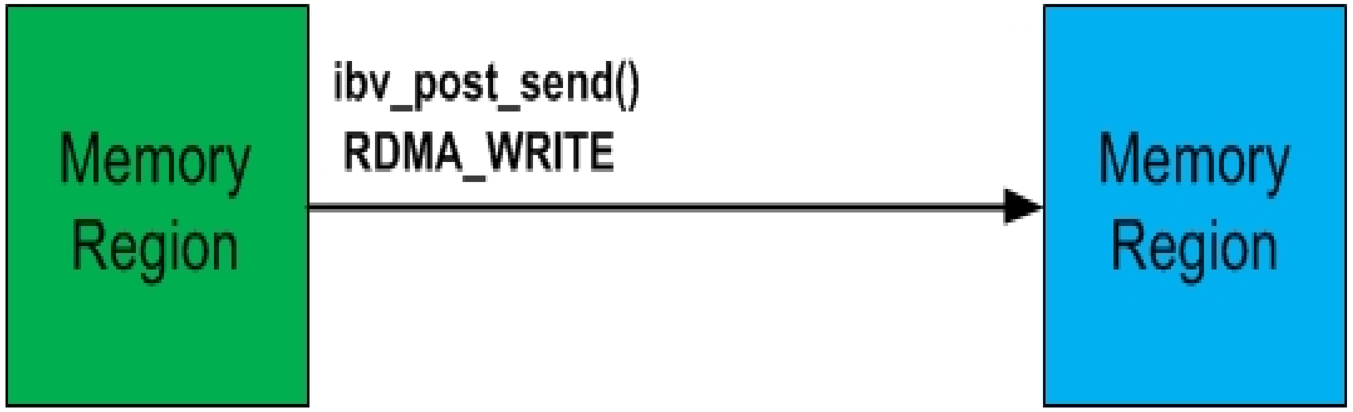
Transfer Posting	Ibv_send_wr Ibv_sge Ibv_recv_wr Ibv_qp Ibv_qp_init_attr
Transfer Completion	Ibv_cq Ibv_wc Ibv_comp_channel
Memory Registration	Ibv_pd Ibv_mr
Connection Management	rdma_cm_id rdma_conn_param rdma_cm_event rdma_event_channel
Misc	Ibv_context Ibv_device ibv_device_attr

	Setup	Use	Break-Down
Transfer Posting	rdma_create_qp	Ibv_post_send Ibv_post_recv	rdma_destroy_qp
Transfer Completion	ibv_create_cq ibv_create_comp_channel	Ibv_poll_cq Ibv_wc_status_str Ibv_req_notify_cq Ibv_get_cq_event Ibv_ack_cq_events	ibv_destroy_cp ibv_destroy_comp_channel
Memory Registration	Ibv_alloc_pd Ibv_reg_mr		Ibv_dealloc_pd Ibv_dereg_mr
Connection Management	rdma_create_id rdma_create_event_channel	rdma_resolve_addr rdma_resolve_route rdma_connect rdma_disconnect rdma_bind_addr rdma_listen rdma_get_cm_event rdma_ack_cm_event rdma_event_str rdma_accept rdma_reject rdma_migrate_id rdma_get_local_addr rdma_get_peer_addr	rdma_destroy_id rdma_destroy_event_channel
Misc		rdma_get_devices rdma_free_devices rdma_query_devices	

RDMA_WRITE operation

- Very different from “normal” socket operations
- Very different from Send/Recv
- Only one side is active, other side is passive
- Active side A calls **ibv_post_send()**
- Passive side P does NOTHING!! – completely passive

RDMA_WRITE data flow



RDMA_WRITE operation

•Active side A calls **ibv_post_send()**

•Passive side P does NOTHING!! – completely passive

–Channel Adapters move data directly from active side A's virtual memory into passive side P's virtual memory

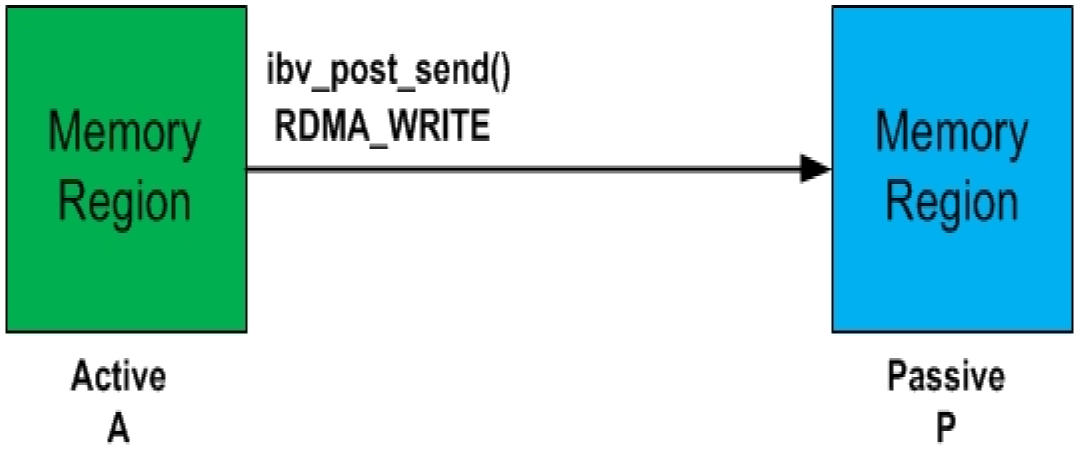
–P does nothing – no **ibv_post_recv()**, **ibv_post_send()**

–P sees nothing – no CPU cycles expended

–P receives no feedback – no events, no completions

•Transmits messages only, no streams

RDMA_WRITE data flow



Differences with Send/Recv

•Local active side A that calls **ibv_post_send()** MUST know virtual memory location on remote passive side P

•Passive side P knows nothing about virtual memory location on active side A

•Passive side P does NOT call **ibv_post_recv()** to match active side A's **ibv_post_send()**

•Prior to the transfer, side P must inform side A of its virtual memory location and its **rkey**

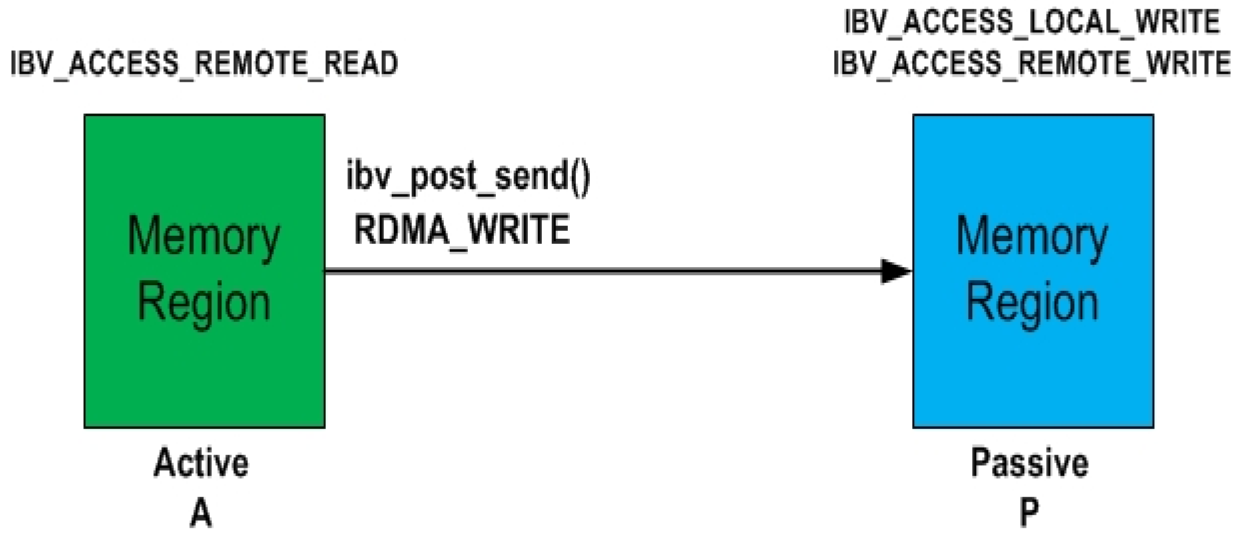
Similarities with Send/Recv

- Both types of transfer are unbuffered
- Both types of transfer require virtual memory on each side to be registered by that side
- Both types of transfer operate asynchronously
- Both types of transfer use same:
  - work request list and scatter-gather list structures
  - completion queues and completion events
  - connection management operations and events
  - verbs and data structures
- Both types of transfer move messages, not streams

Prior to RDMA_WRITE

- Before active side A issues **ibv_post_send()** for RDMA_WRITE, side P MUST inform side A of side P's virtual memory location and **rkey**
- Passive side P must register its virtual memory for both IBV_ACCESS_LOCAL_WRITE and IBV_ACCESS_REMOTE_WRITE
- Active side A must register its virtual memory for IBV_ACCESS_REMOTE_READ

RDMA_WRITE access rights



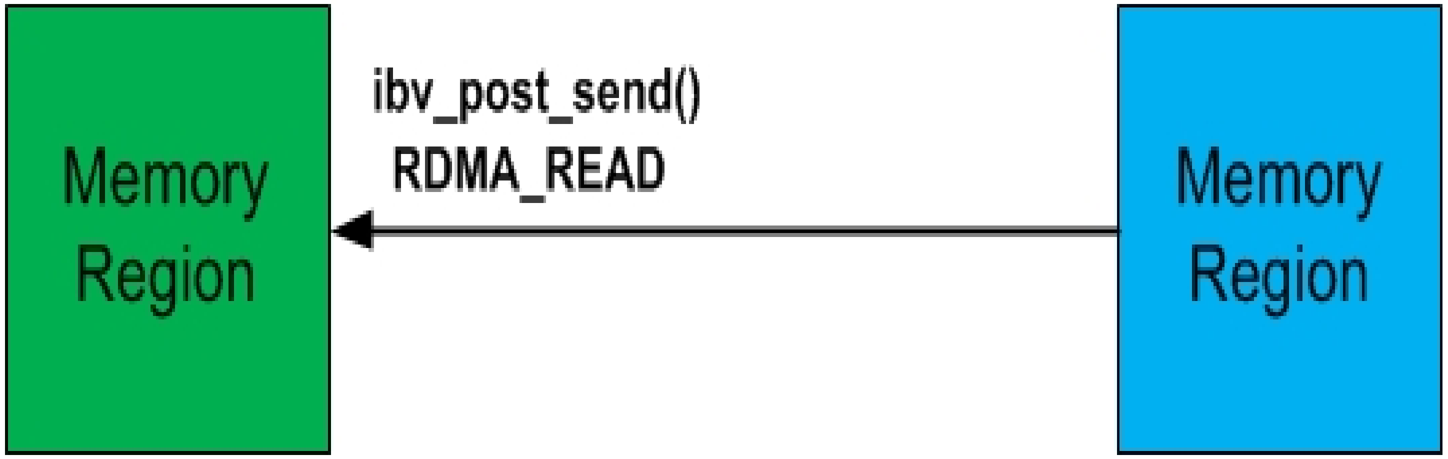
SWR for RDMA_WRITE

- Differences from **struct ibv_send_wr** for SEND
  - **opcode** is IBV_WR_RDMA_WRITE, not IBV_WR_SEND
  - **lkey** in SGE must have been created with access IBV_ACCESS_REMOTE_READ
  - Two more fields in **struct ibv_send_wr** must be filled
- **wr.rdma.remote_addr** – remote side P's virtual memory address
- **wr.rdma.rkey** – remote side P's **rkey**
- Remote memory on side P must be one

RDMA_READ operation

- Very different from “normal” socket operations
- Very different from Send/Recv
- Only one side is active, other side is passive
- Active side A calls **ibv_post_send()** – YES, SEND!!!
- Passive side B does NOTHING!! – completely passive

RDMA_READ data flow



RDMA_READ operation

•Active side A calls **ibv_post_send()** – YES, SEND!!!

•Passive side P does NOTHING!!!– completely passive

–Channel Adapters move data directly from passive side P's virtual memory into active side A's virtual memory

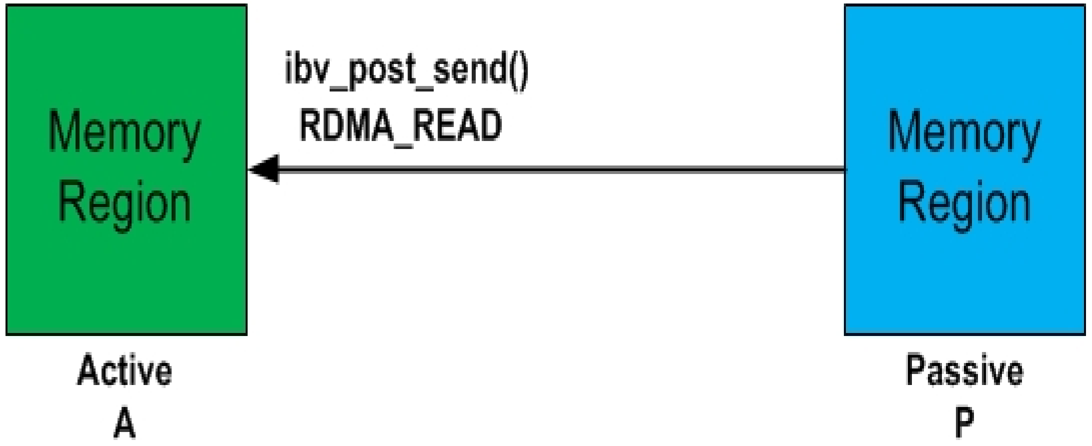
–P does nothing – no **ibv_post_recv()** or **ibv_post_send()**

–P sees nothing – no CPU cycles expended

–P receives no feedback – no events, no completions

•Transmits messages only, no streams

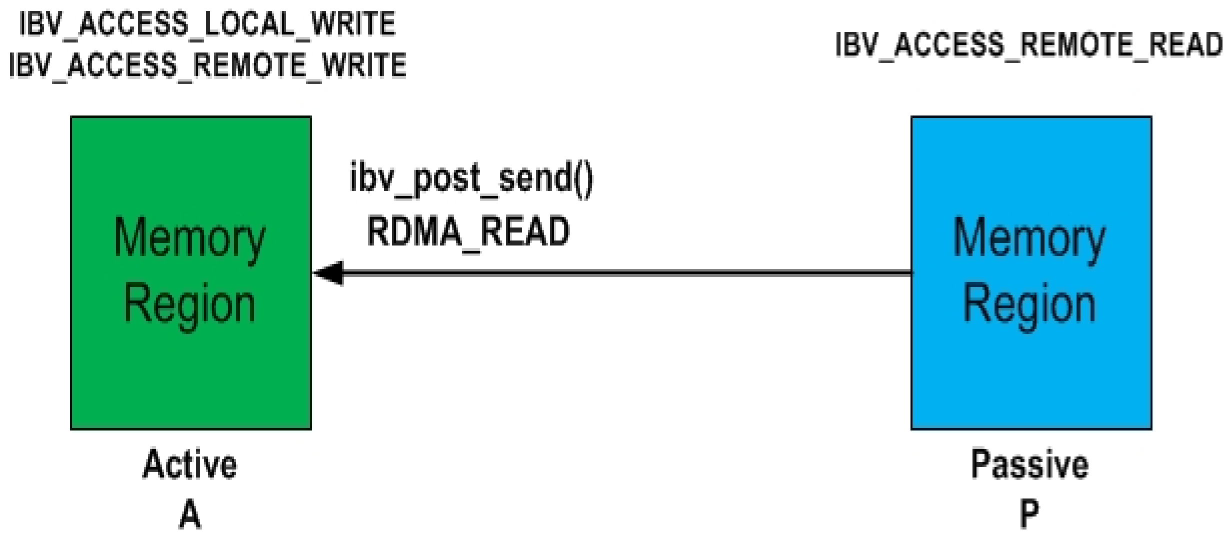
RDMA_READ data flow



Prior to RDMA_READ

- Before active side A issues **ibv_post_send()** for RDMA_READ, passive side P MUST inform side A of side P's virtual memory location and **rkey**
- Passive side P must register its virtual memory for IBV_ACCESS_REMOTE_READ
- Active side A must register its virtual memory for IBV_ACCESS_LOCAL_WRITE and IBV_ACCESS_REMOTE_WRITE

RDMA_READ access rights



SWR for RDMA_READ

•Differences from **struct ibv_send_wr** for RECV

–Do NOT use **struct ibv_recv_wr**

–**opcode** is IBV_WR_RDMA_READ

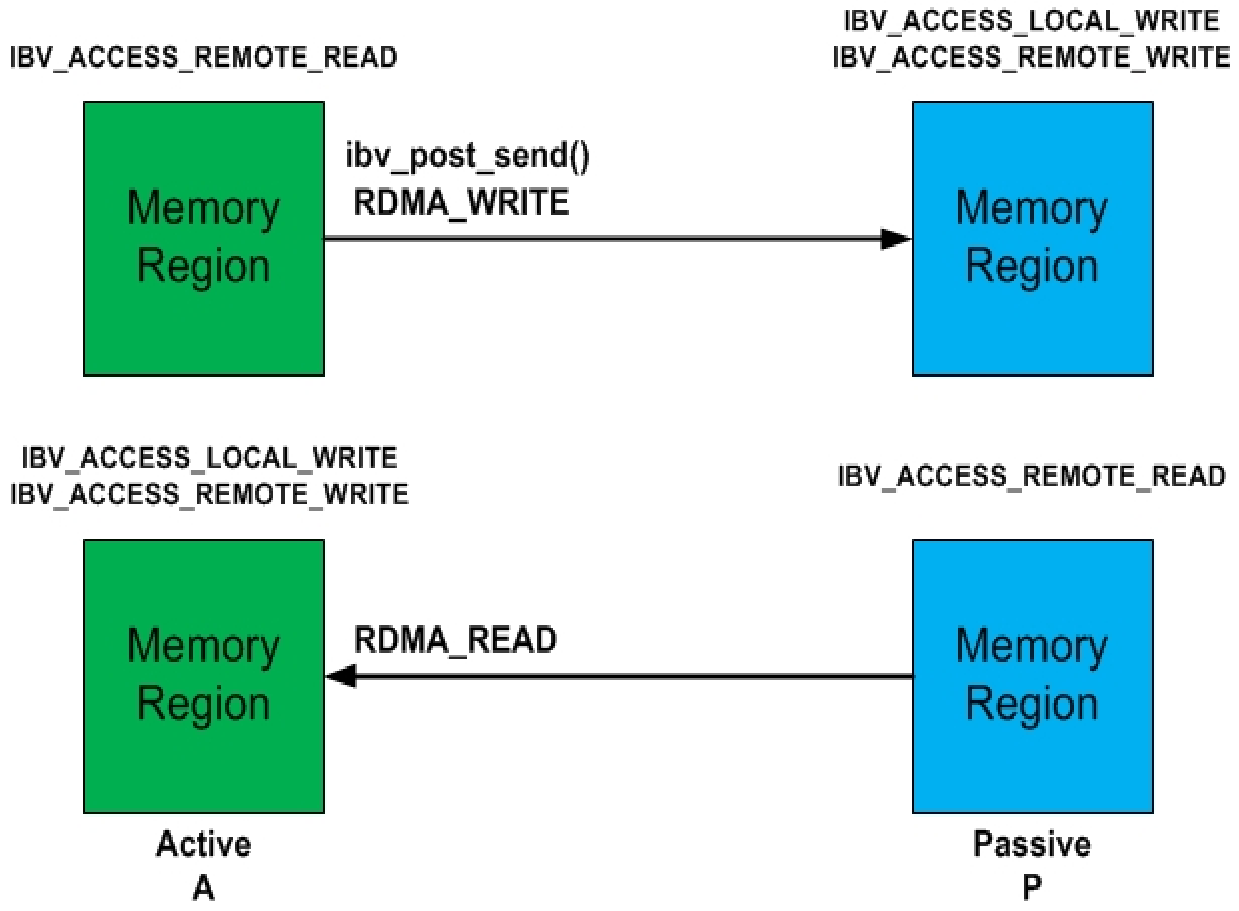
•(RECV did not need an opcode in **struct ibv_recv_wr**)

–**lkey** in SGE must have been created with access IBV_ACCESS_LOCAL_WRITE and IBV_ACCESS_REMOTE_WRITE

–Two more fields in **struct ibv_send_wr** must be filled

•**wr.rdma.remote_addr** – remote side P's virtual memory address

Summary of RDMA access rights



Ping-Pong example rdma-1

•Client does all the work in the “ping-pong” loop

–Client does rdma_write on ping data

–Client does rdma_read on pong data

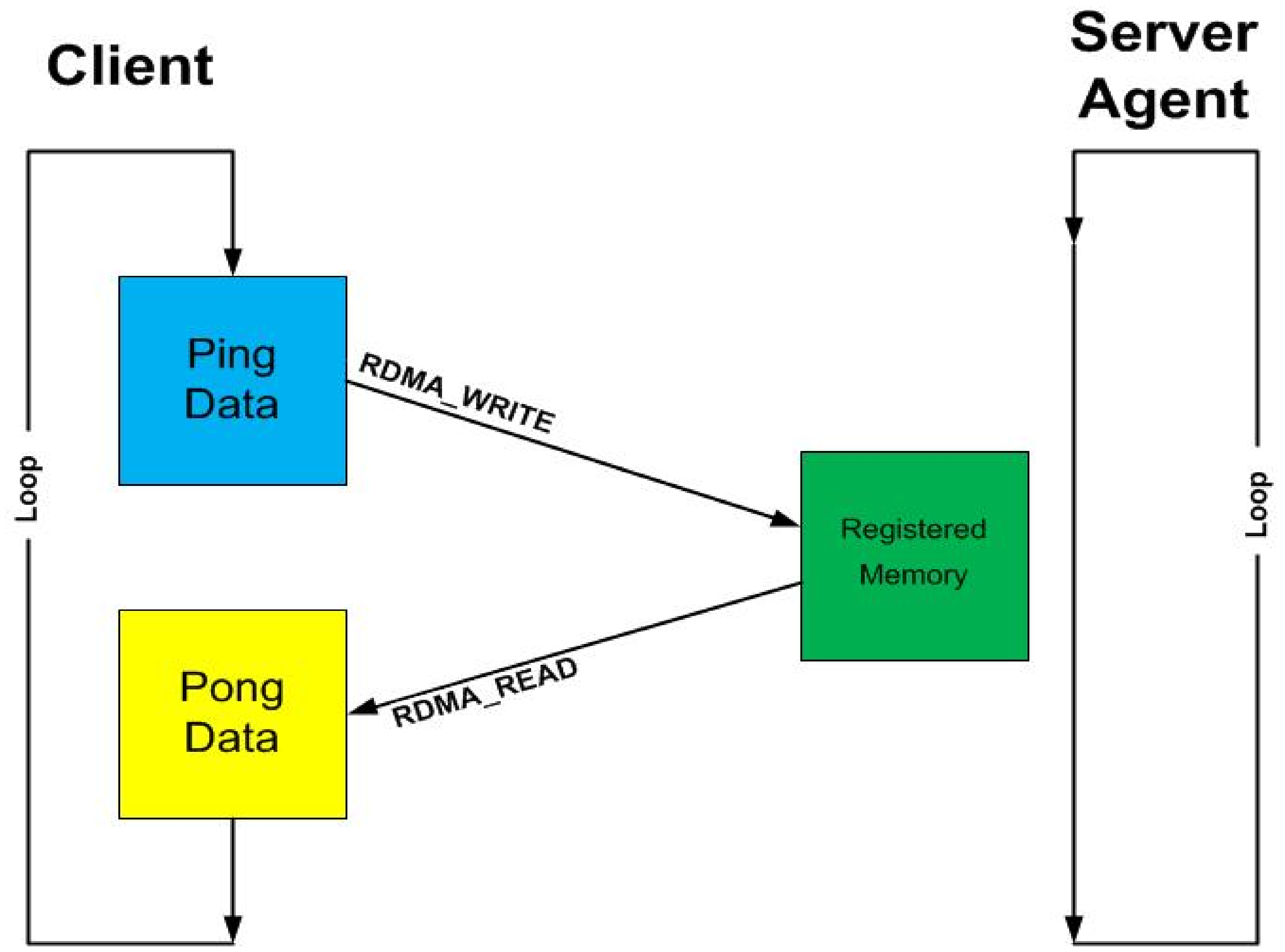
•Agent “sleeps” during “ping-pong” loop

–Agent's CPU utilization is 0

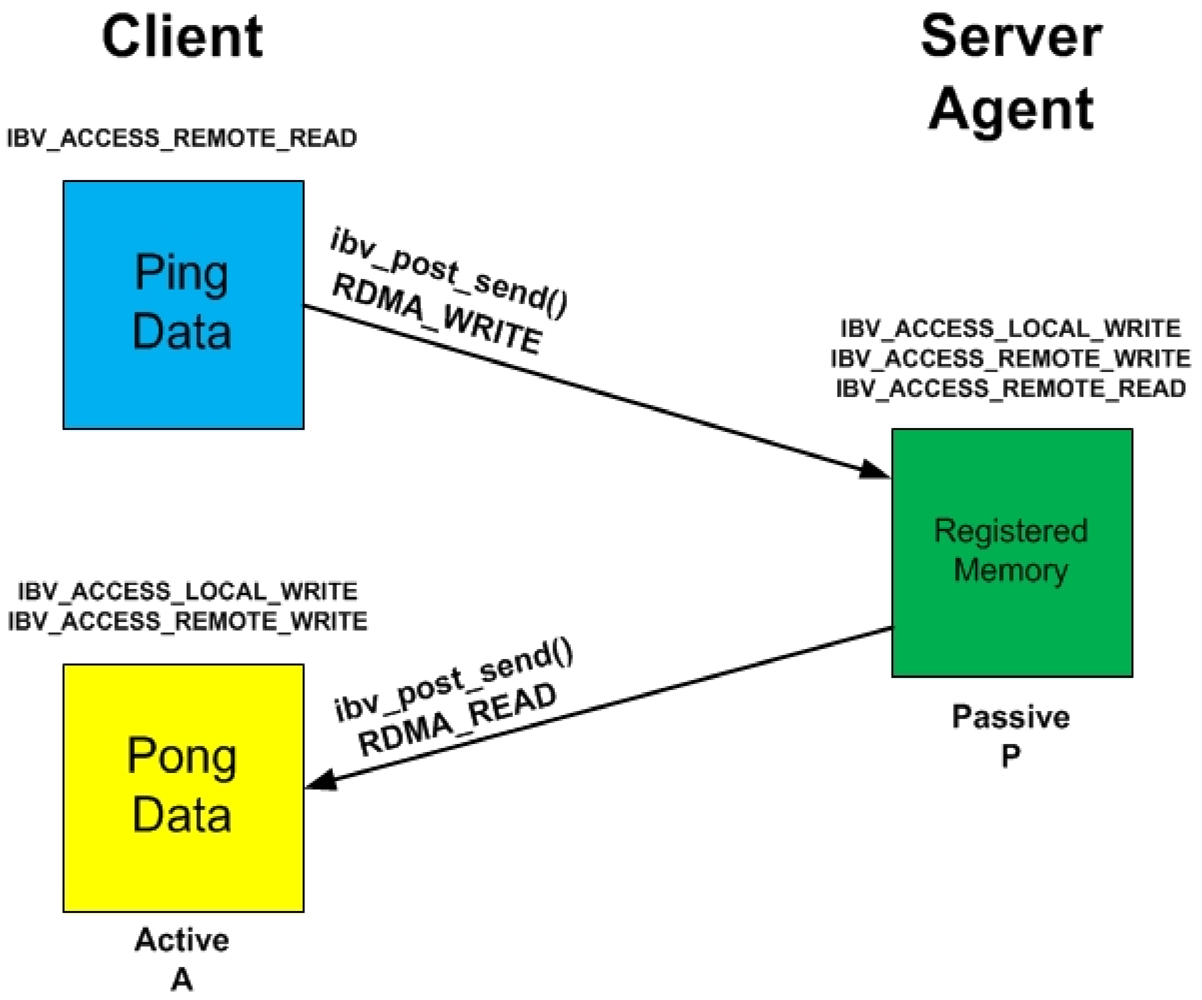
•Both need extra step before loop to exchange buffer info

•Both need extra step after loop to synchronize end-of-run

Ping-pong using RDMA



Ping buffers with access rights



Set up client data buffers

•"ping-pong" client needs 2 user data buffers

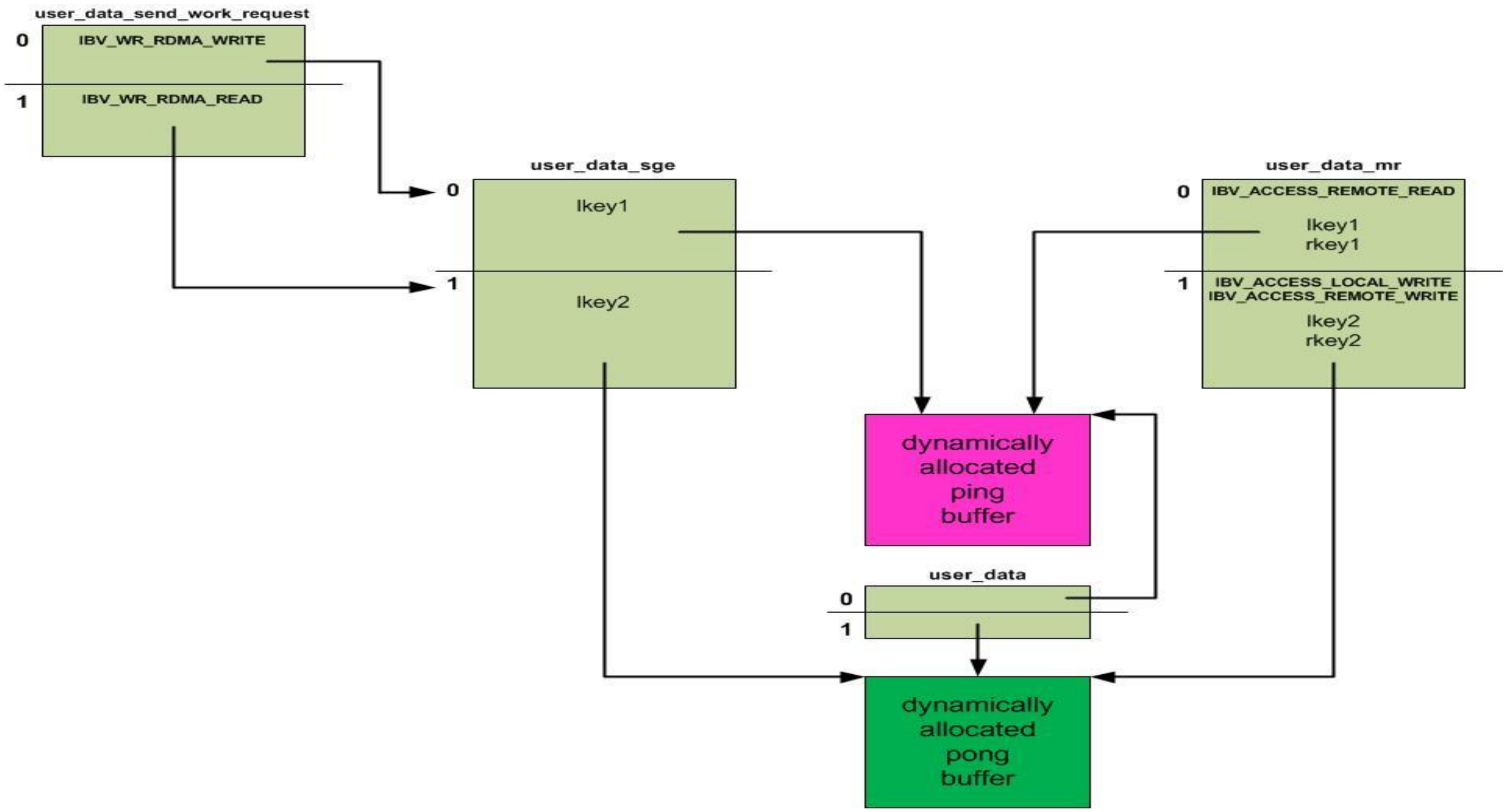
–First contains original "ping" data

•Access rights IBV_ACCESS_REMOTE_READ allow client to post RDMA_WRITE to agent

–Second gets reflected "pong" data

•Access rights both IBV_ACCESS_LOCAL_WRITE and IBV_ACCESS_REMOTE_WRITE allow client to post RDMA_READ from agent

Client's data buffers,WRs and MRs



Problems

•How does client get the agent's buffer location and access rights key (so client can write-into and read-from agent's buffer)?

•How does agent know when data transfer is finished (during transfer the agent is completely passive)?

Solutions

- Agent must convey its buffer location and access rights key to client prior to start of data transfer
  - Accomplished via an exchange of messages using send/recv called buffer-info exchange
- Client must convey an end-of-run indication to agent after end of data transfer
  - Accomplished via an exchange of messages using send/recv called end-of-run ack exchange

Structure of Client Use Phase

- buffer-info exchange
  - performed using send/recv
- ping-pong data transfer loop
  - performed using rdma_write and rdma_read
- end-of-run ack exchange
  - performed using send/recv

Set up client buffer_info exchange

•Client needs 2 new buffer_info buffers

–one to hold information on local data buffers

•In this demo, agent will not use this info

–other to hold information on remote data buffers

•In this demo, agent has only 1 buffer

•Client needs 2 new work requests

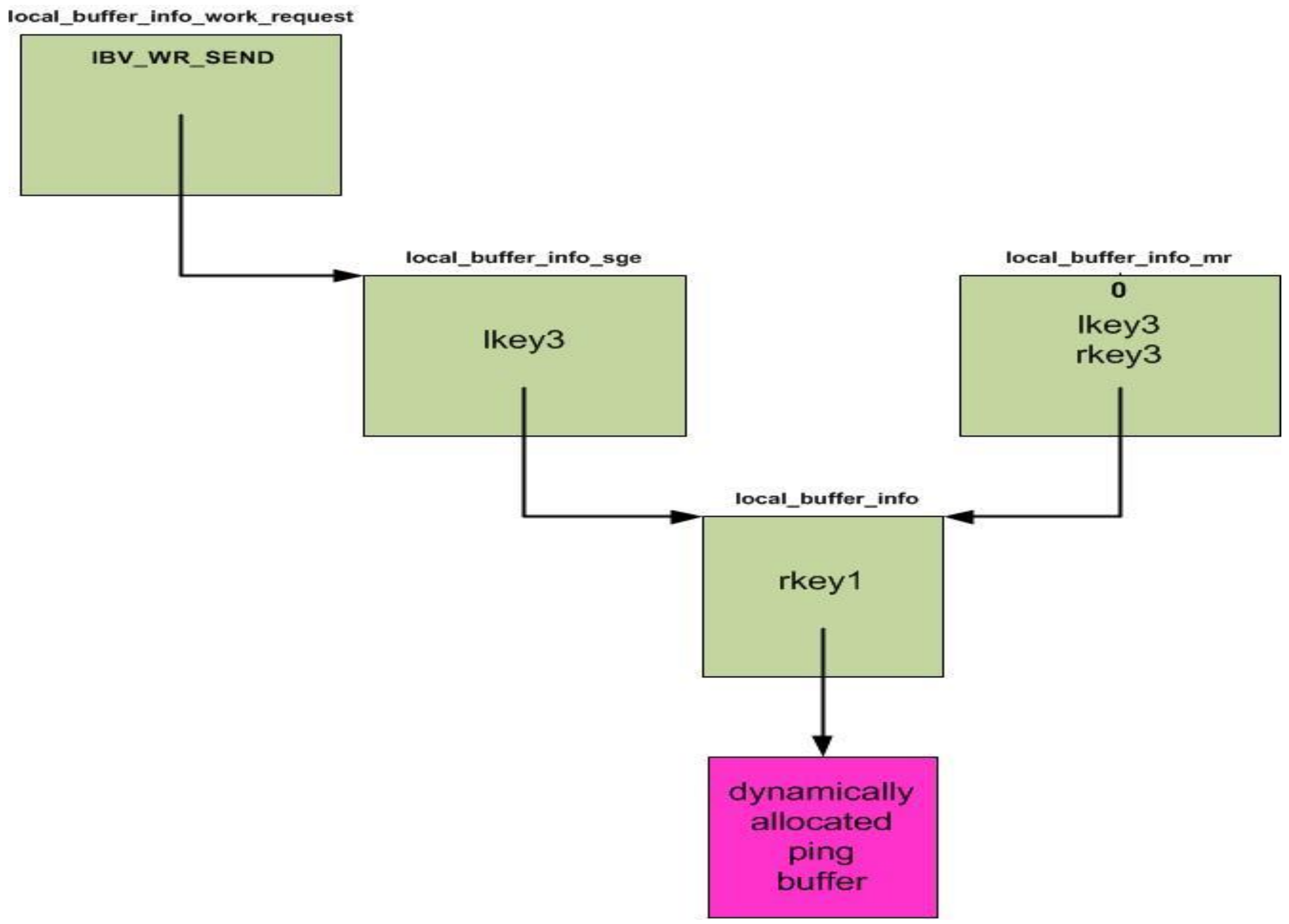
–one to send local buffer_info to agent

•In this demo, agent uses this message only for synchronization

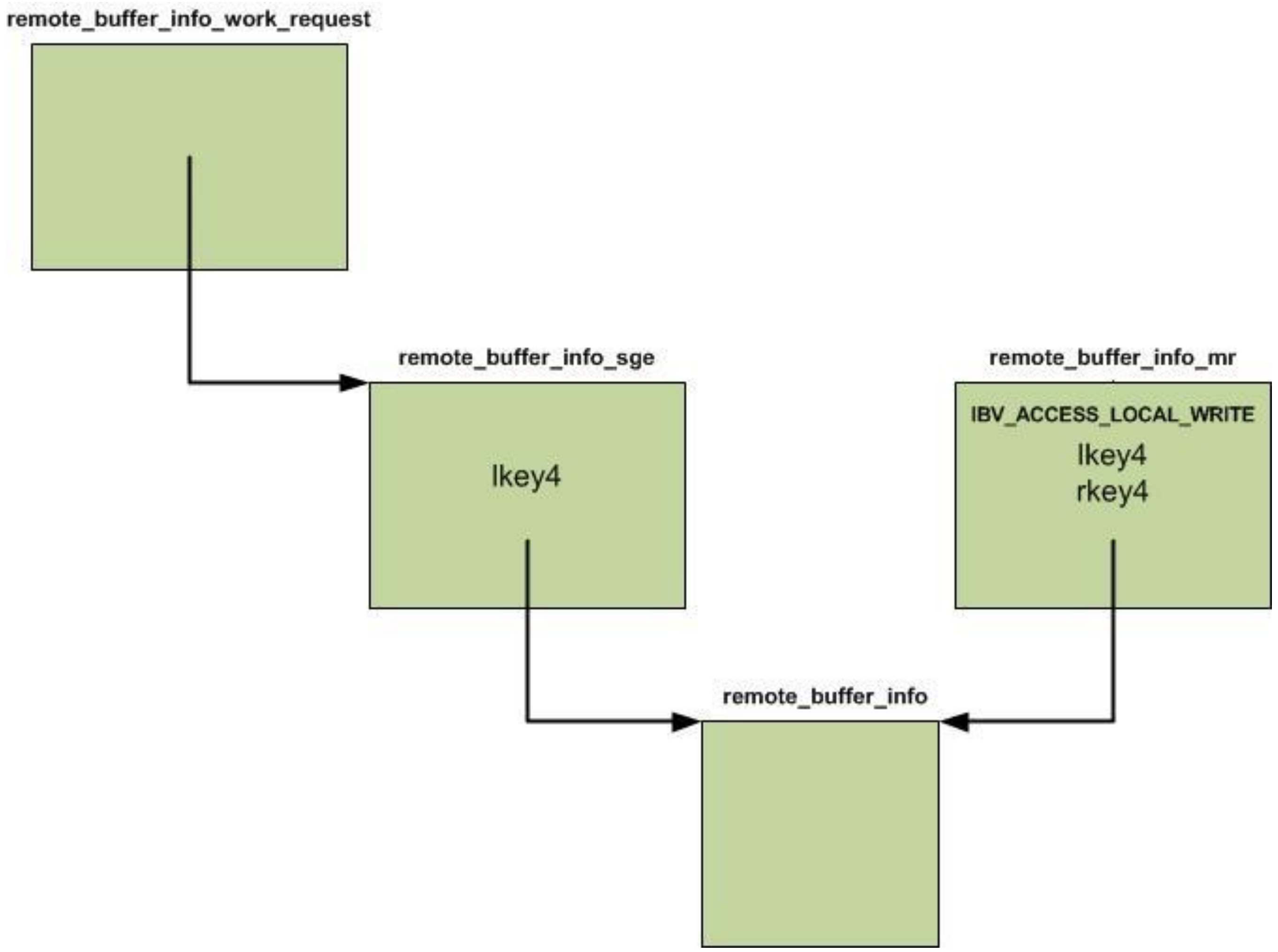
–other to recv remote buffer_info from agent

•In this demo, client must use this info in both the

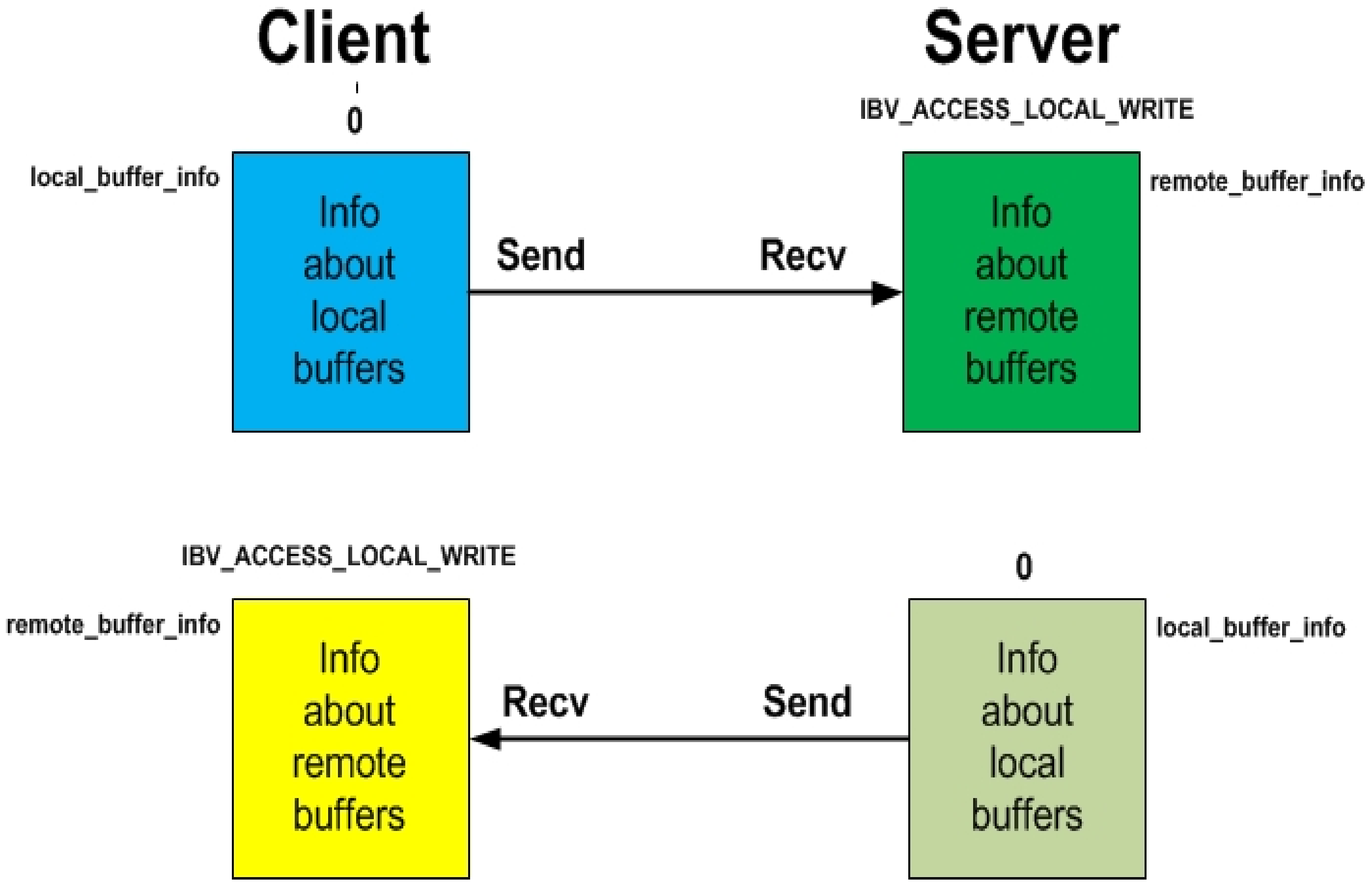
Client's local buffer_info send WR



Client's remote buffer_info recv WR



Exchange of info about buffers



Set up end-of-run acks

•Needed because agent “sleeps” during data transfer

–Client sends ack containing number of transfers finished

–Agent “wakes up” and sends reply ack to client

–Agent ends timings and starts break-down phase

•Client needs 2 new ack buffers

–one to hold final transfer count for sending to agent

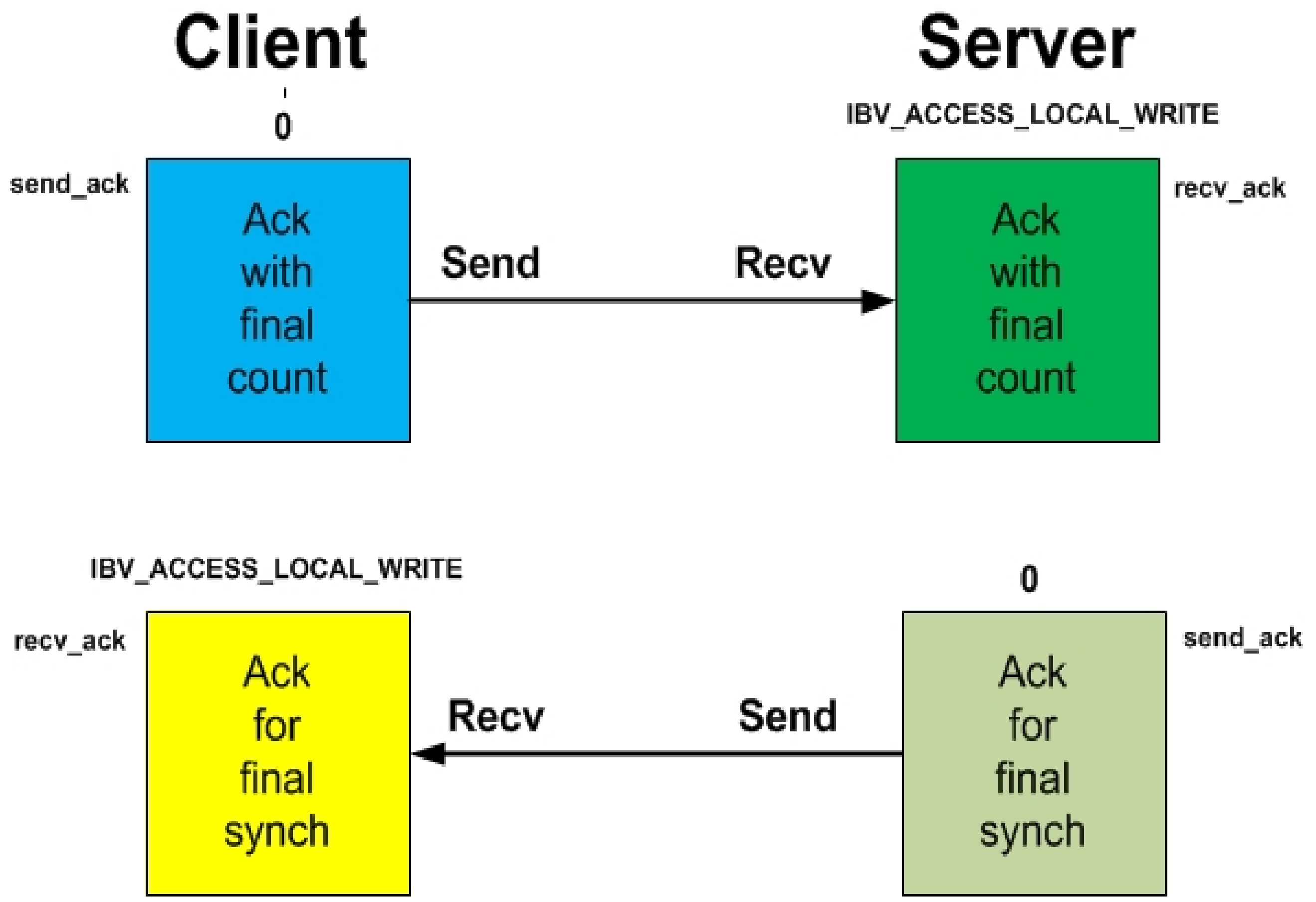
–one to hold final ack for receiving from agent

•Client needs 2 new work requests

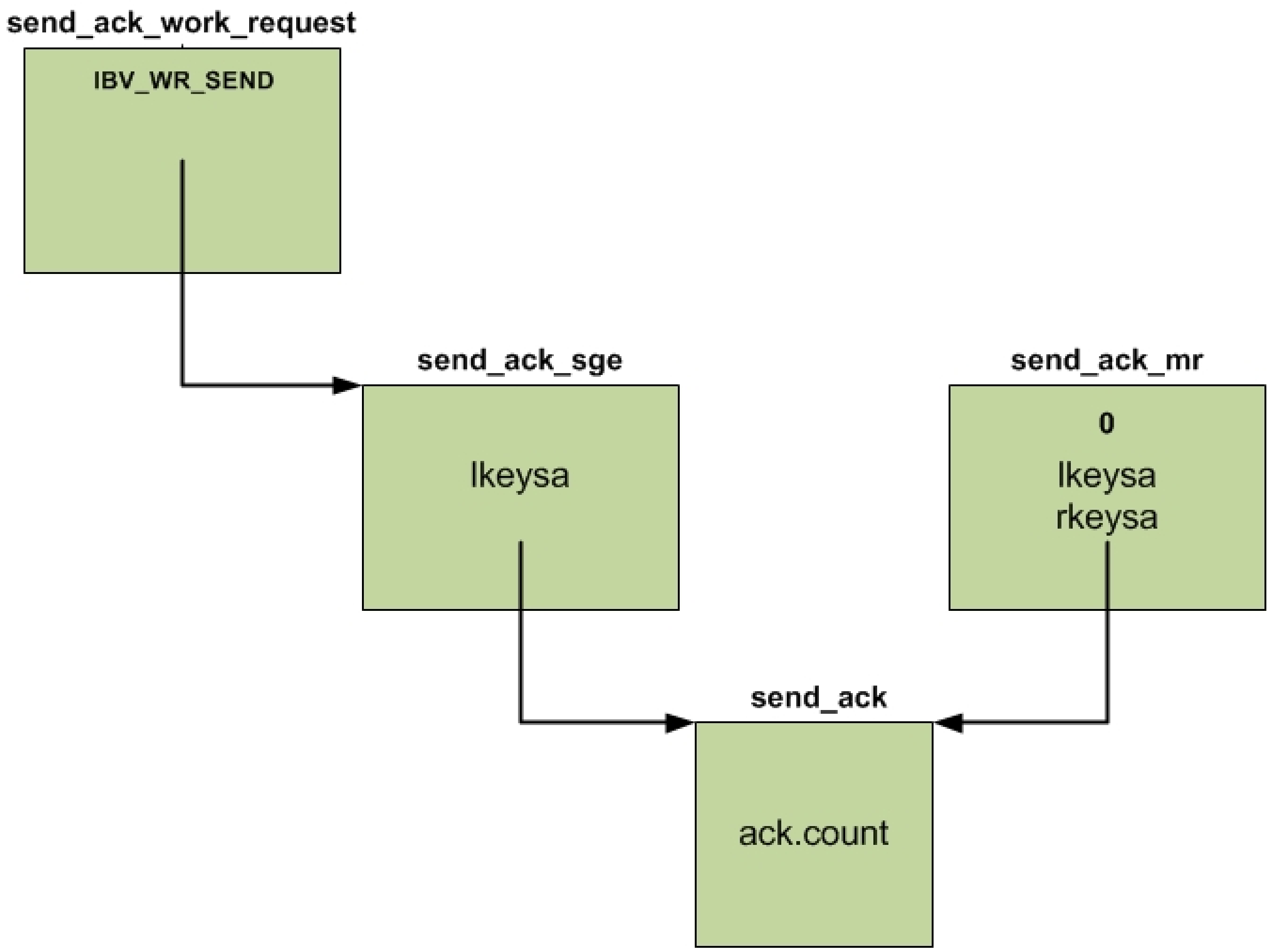
–one to send ack to agent

–other to recv ack back from agent

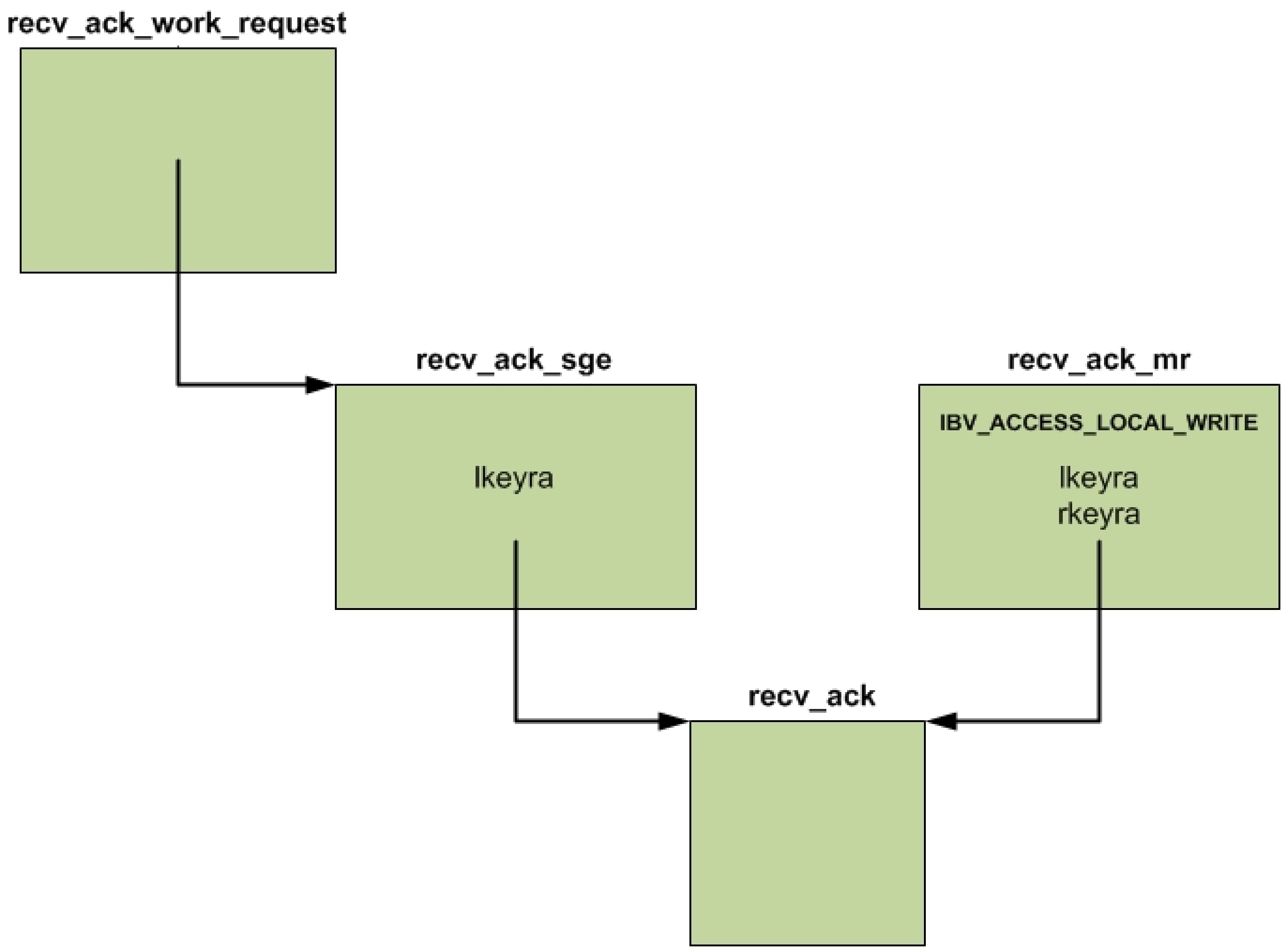
Exchange of end-of-run acks



Client's send ack work request



Client's receive ack work request



Structure of Client Use Phase

1. buffer-info exchange

   performed using send/recv

1. ping-pong data transfer loop

   performed using rdma_write and rdma_read

1. end-of-run ack exchange

   performed using send/recv

1. Client buffer_info exchange

- our_post_recv() to get agent's buffer_info
- our_post_send() to send client's buffer_info
  - buffer_info must be sent in network byte order
- our_await_completion() for send
- our_await_completion() for recv
- Use agent's buffer_info to fill in client's work requests
  - first WR for RDMA_WRITE “ping” data to agent
  - second WR for RDMA_READ “pong” data from agent
  - buffer_info received in network byte order

Client buffer-info exchange code 1

```
/* post a receive to catch the remote agent's buffer info */
ret = our_post_recv(client_conn,
            &client_conn->remote_buffer_info_work_request, options);
if (ret != 0) {
    goto out0;
}

/* now we send our local buffer info to the remote agent */
ret = our_post_send(client_conn,
               &client_conn->local_buffer_info_work_request, options);
if (ret != 0) {
    goto out0;
}
```

Client buffer-info exchange code 2

```
/* wait for the send local buffer info to complete */
ret = our_await_completion(client_conn, &work_completion, options);
if (ret != 0) {
    goto out0;
}

/* wait for the recv remote buffer info to complete */
ret = our_await_completion(client_conn, &work_completion, options);
if (ret != 0) {
    goto out0;
}
```

Client buffer-info exchange code 3

```
/* use remote agent's buffer info to fill in
 * rdma part of our RDMA_WRITE and RDMA_READ work requests
 * to both point to the remote agent's single buffer
 */
client_conn->user_data_send_work_request[0].wr.rdma.remote_addr
                    = ntohll(client_conn->remote_buffer_info[0].addr);
client_conn->user_data_send_work_request[0].wr.rdma.rkey
                    = ntohl(client_conn->remote_buffer_info[0].rkey);
client_conn->user_data_send_work_request[1].wr.rdma.remote_addr
                    = ntohll(client_conn->remote_buffer_info[0].addr);
client_conn->user_data_send_work_request[1].wr.rdma.rkey
                    = ntohl(client_conn->remote_buffer_info[0].rkey);
```

2. Synopsis of client's ping-pong loop

```
client_conn->wc_rdma_both = 0;
while (client_conn->wc_rdma_both < options->limit) {
    call our_post_send() for RDMA_WRITE to agent
    call our_await_completion()
    call our_post_send() for RDMA_READ from agent
    call our_await_completion()
    client_conn->wc_rdma_both++
}
```

3. Client's ack exchange

- •our_post_recv() to catch agent's final ack
- •fill in local ack with final count in network byte order
- •our_post_send() to send ack with count to agent
- •our_await_completion() for send
- •our_await_completion() for recv
  - –Used for synchronization, contains no useful info

Client's ack exchange code 1

```
/* post a receive to catch the remote agent's only ACK */
ret = our_post_recv(client_conn,
                    &client_conn->recv_ack_work_request, options);
if (ret != 0) {
    goto out1;
}

/* tell the agent the number of iterations we finished */
client_conn->send_ack.ack_count = htonl(client_conn->wc_rdma_both);

/* now we send our only ACK to the remote agent */
ret = our_post_send(client_conn,
                    &client_conn->send_ack_work_request, options);
if (ret != 0) {
    goto out1;
}
```

Client's ack exchange code 2

```
/* wait for the send ACK to complete */
ret = our_await_completion(client_conn,&work_completion,options);
if (ret != 0) {
    goto out1;
}

/* wait for agent's only ACK to complete */
ret = our_await_completion(client_conn,&work_completion,options);
if (ret != 0) {
    /* hit error or FLUSH_ERR, in either case leave now */
    goto out1;
}
```

Example ping-rdma-1

•Run ping-rdma-1

•Client drives all data transfers

•Agent completely passive during data transfer

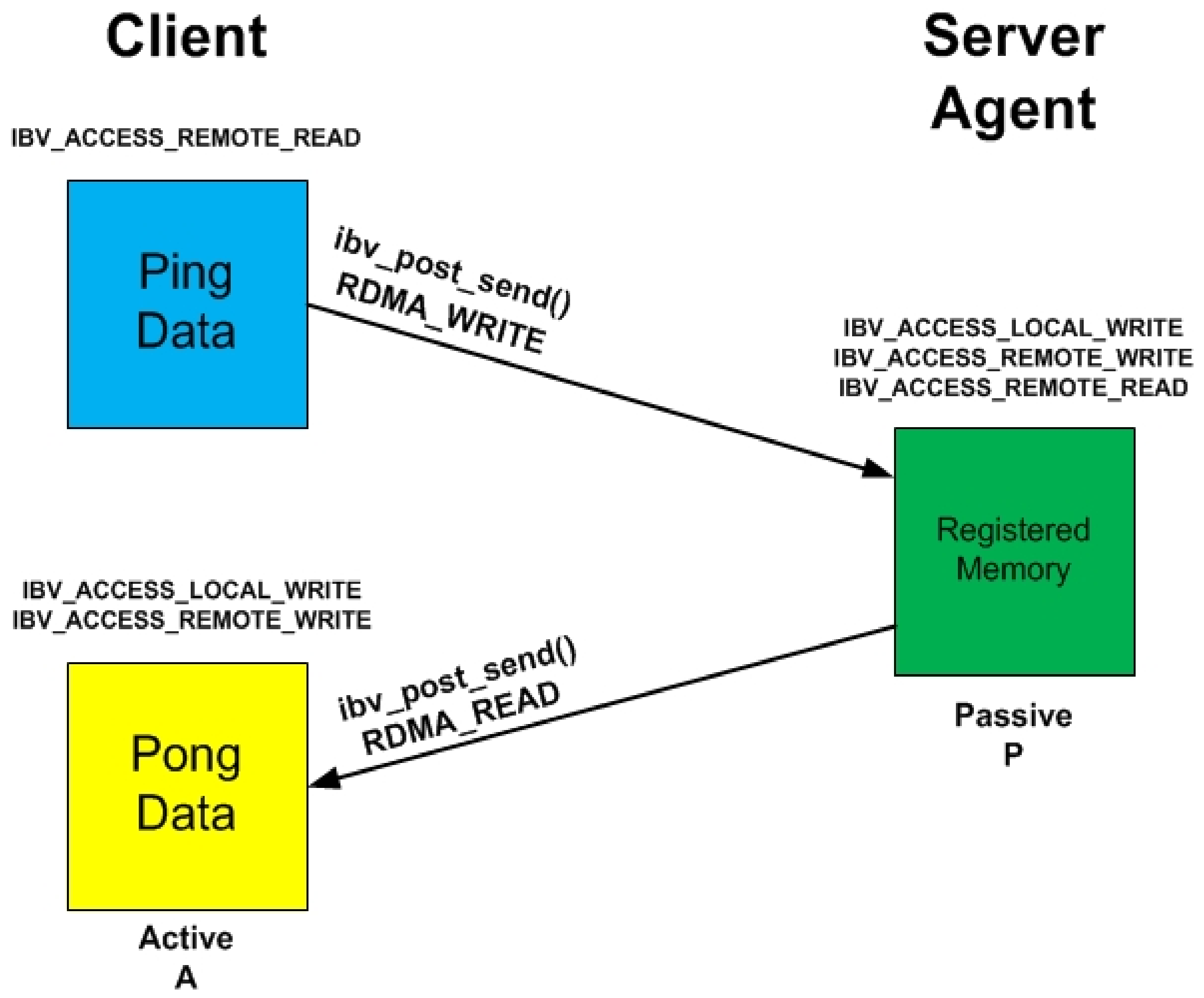
–CPU usage is 0

•Walk through client code as necessary

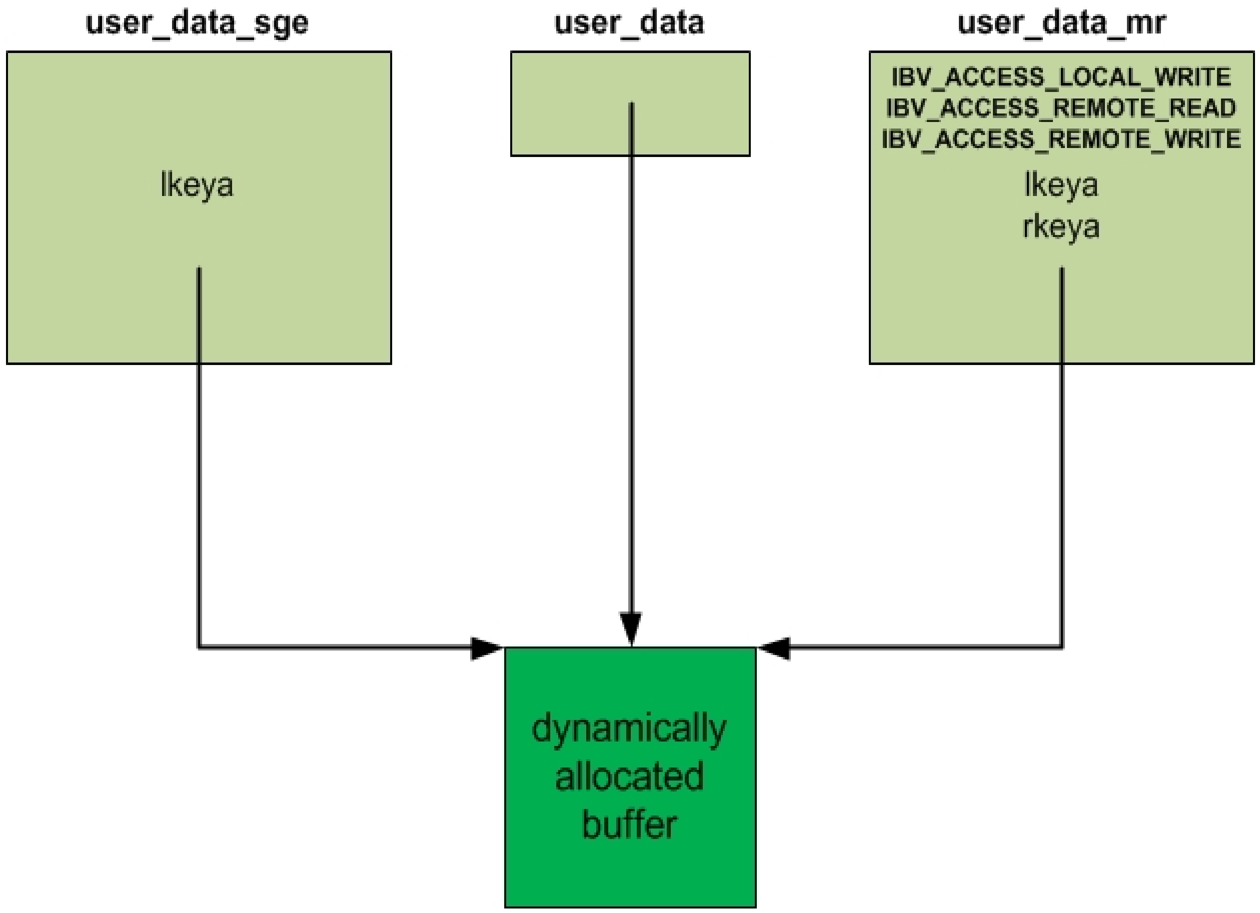
Set up agent data buffers

- “ping-pong” agent needs 1 user data buffer
  - Agent issues no operations on it
  - Client issues RDMA_WRITE and RDMA_READ on it
  - Access rights IBV_ACCESS_LOCAL_WRITE, IBV_ACCESS_REMOTE_WRITE and IBV_ACCESS_REMOTE_READ

Ping buffers with access rights



Agent's data buffer setup



Structure of Agent Use Phase

- buffer-info exchange
  - performed using recv/send
- ping-pong data transfer loop
  - do nothing! - client does all the work
- end-of-run ack exchange
  - performed using recv/send

Set up agent buffer_info exchange

- Agent needs 2 new buffer_info buffers
  - One to hold information on all local data buffers
    - In this demo, agent has only 1 data buffer
  - One to hold information on all remote data buffers
    - In this demo, agent will not use this info
- Agent needs 2 new work requests
  - One to send local buffer_info to client
  - One to recv remote buffer_info from client
    - In this demo, agent will not use this info

Set up end-of-run acks

•Agent needs 2 new ack buffers

–one to hold final transfer count to receive from client

–other to hold final ack to send to client

•Agent needs 2 new work requests

–one to recv ack from client

–other to send ack back to client

Agent buffer_info exchange

•our_post_recv() to get client's buffer_info

–Needed for synchronization, info not used by agent

•our_post_recv() to get client's end-of-run ack

•our_await_completion() for first recv

•our_post_send() to send agent's buffer_info

•our_await_completion() for send

Synopsis of agent's ping-pong loop

Nothing to do!!! Completely passive

Agent's ack exchange

- our_await_completion() for client's ack with count
- convert final count from network byte order
- our_post_send() to send final ack to client
- our_await_completion() for send

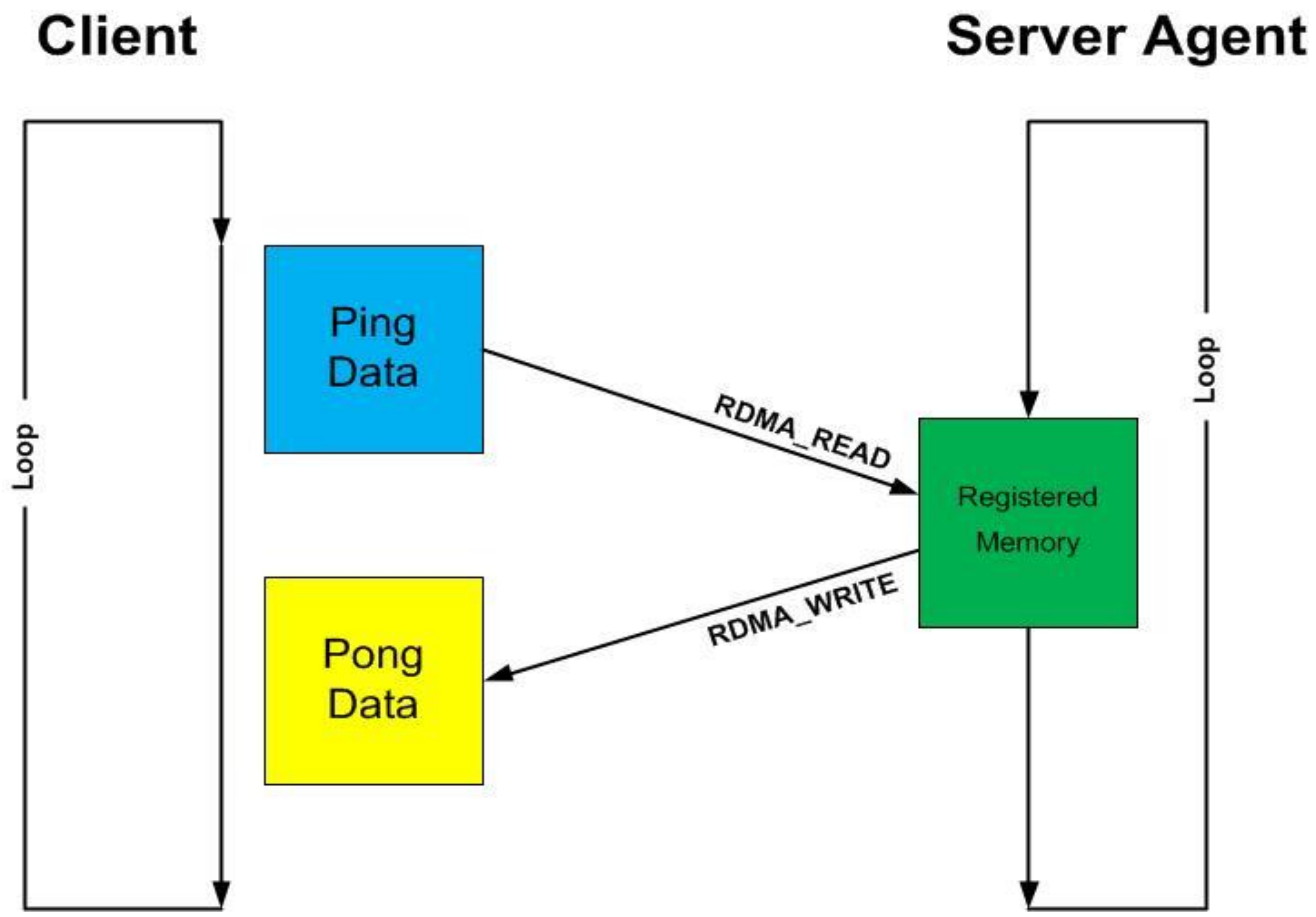
Example ping-rdma-1

- Walk through agent code as necessary

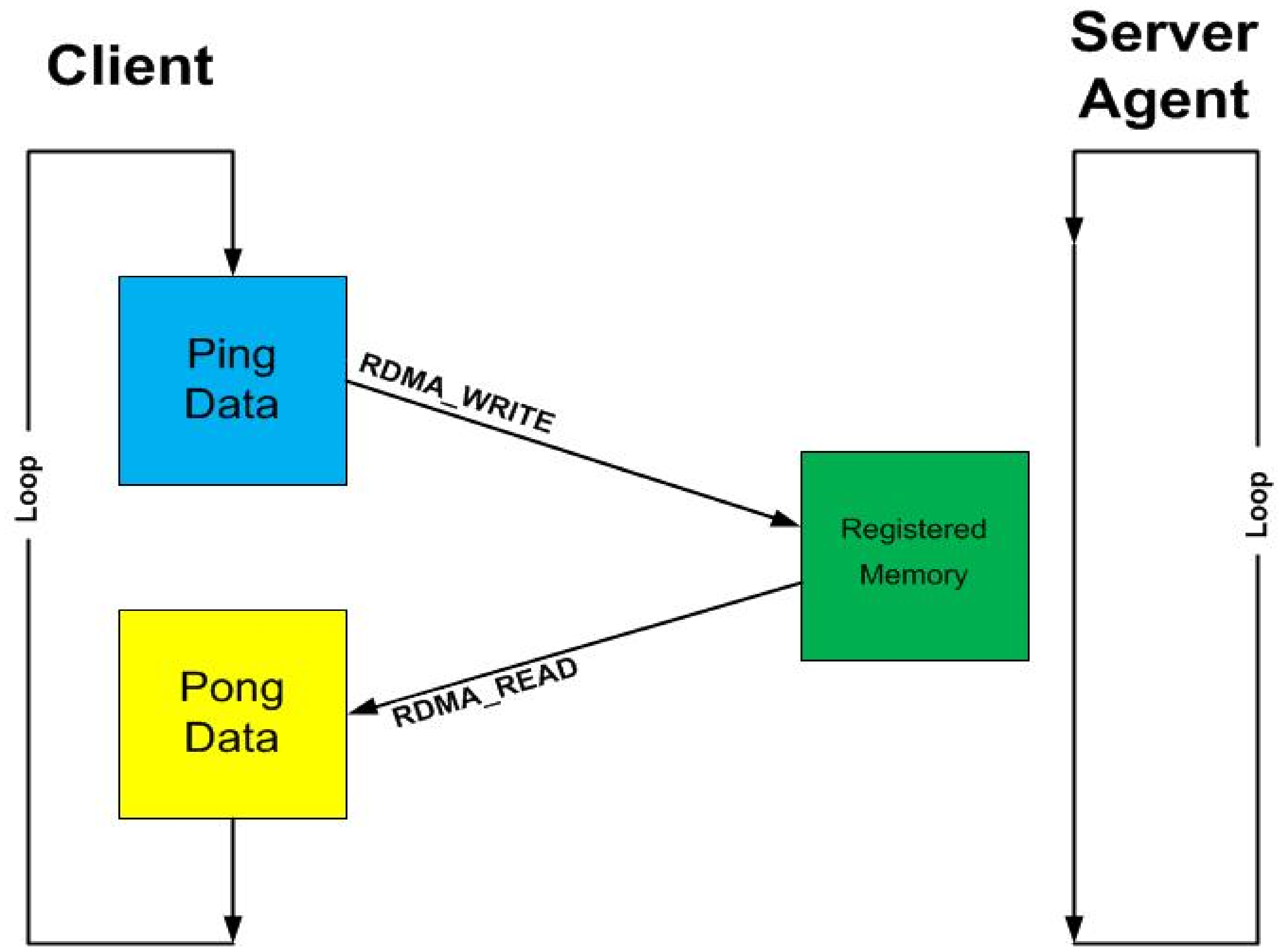
Ping-Pong exercise ping-rdma-1e

- Opposite of previous example
- Server now does all the work
  - Server does rdma_read on ping data
  - Server does rdma_write on pong data
- Client does nothing – CPU utilization is 0
- Exercise for reader during labs

Server drives all data transfers



Another look at ping-rdma-1



How to improve performance

- In everything done so far, all work requests (WRs) submitted to a send or receive queue have resulted in a work completion (WC) being generated in a completion queue (CQ)
- RDMA specifications require this for all receive work requests (RWRs), but not for all send work requests (SWRs) – the user can control this
- We have been requesting this for all our SWRs in **our_setup_send_wr()** by setting the value of the **send_flag** field in **struct ibv_send_wr** to IBV_SEND_SIGNALED

Unsignaled send work requests

- If we set the **send_flag** field in **struct ibv_send_wr** to 0, no WC is generated in the corresponding CQ if that SWR completes successfully (a WC is always generated on any error processing a posted WR)
- A SWR with **send_flag** value of 0 is called an unsignaled work request
- So how does a user know when an unsignaled work request completes successfully?

Unsignaled work request completion

- An unsignaled WR completes successfully when:
  - A WC for a subsequent WR is retrieved from the CQ associated with the SQ where the unsignaled WR was posted
  - And that subsequent WR was posted on the same SQ as the unsignaled WR
  - And that subsequent WR is ordered after the unsignaled WR
- Only then can resources associated with the unsignaled WR be reused

Ordering rules

•The rules for processing WRs and WCs allow for flexibility in actual implementations

•WRs submitted to a single SQ must be initiated, sent and completed in the order they are submitted

•However, processing of the data transfers from multiple WRs submitted to the same SQ can be done in parallel – in particular, data may be placed into target memory in any order

Ordering rules continued

- If different messages, or parts of the same message, being processed in parallel refer to the same or overlapping buffers, then it is possible that the last incoming write will not be the last outgoing data sent
- For an RDMA_WRITE, the contents of the target buffer are indeterminate until a subsequent Send message is completed by consuming a WC at the target (e.g., an ACK is sent back and completes at the target)

Special ordering rules

•When a user submits to the same SQ a SWR for RDMA_WRITE followed by a SWR for RDMA_READ targeting the same remote buffer (which is what we are doing in this demo), the RDMA_READ must return the data as modified by the RDMA_WRITE

•This is always true, whether or not the RDMA_WRITE SWR is signaled or unsignaled

Benefiting from these rules

- In this demo we submit to the same SQ a SWR for RDMA_WRITE followed by a SWR for RDMA_READ both targeting the same buffer
- Therefore, the RDMA_WRITE SWR can be unsignaled (i.e., it generates no WC) because the WC generated by the SWR for RDMA_READ guarantees that the SWR for RDMA_WRITE completed successfully prior to the RDMA_READ SWR completion

Revised client ping-pong loop

```
/* turn off SIGNALED flag on RDMA_WRITE */
client_conn->user_data_send_work_request.send_flags
                                        &= ~IBV_SEND_SIGNALED;
client_conn->wc_rdma_both = 0;
while (client_conn->wc_rdma_both < options->limit) {
    call our_post_send() for RDMA_WRITE to agent
    /***** do NOT call our_await_completion() *****/
    call our_post_send() for RDMA_READ from agent
    call our_await_completion()
    client_conn->wc_rdma_both++;
}
```

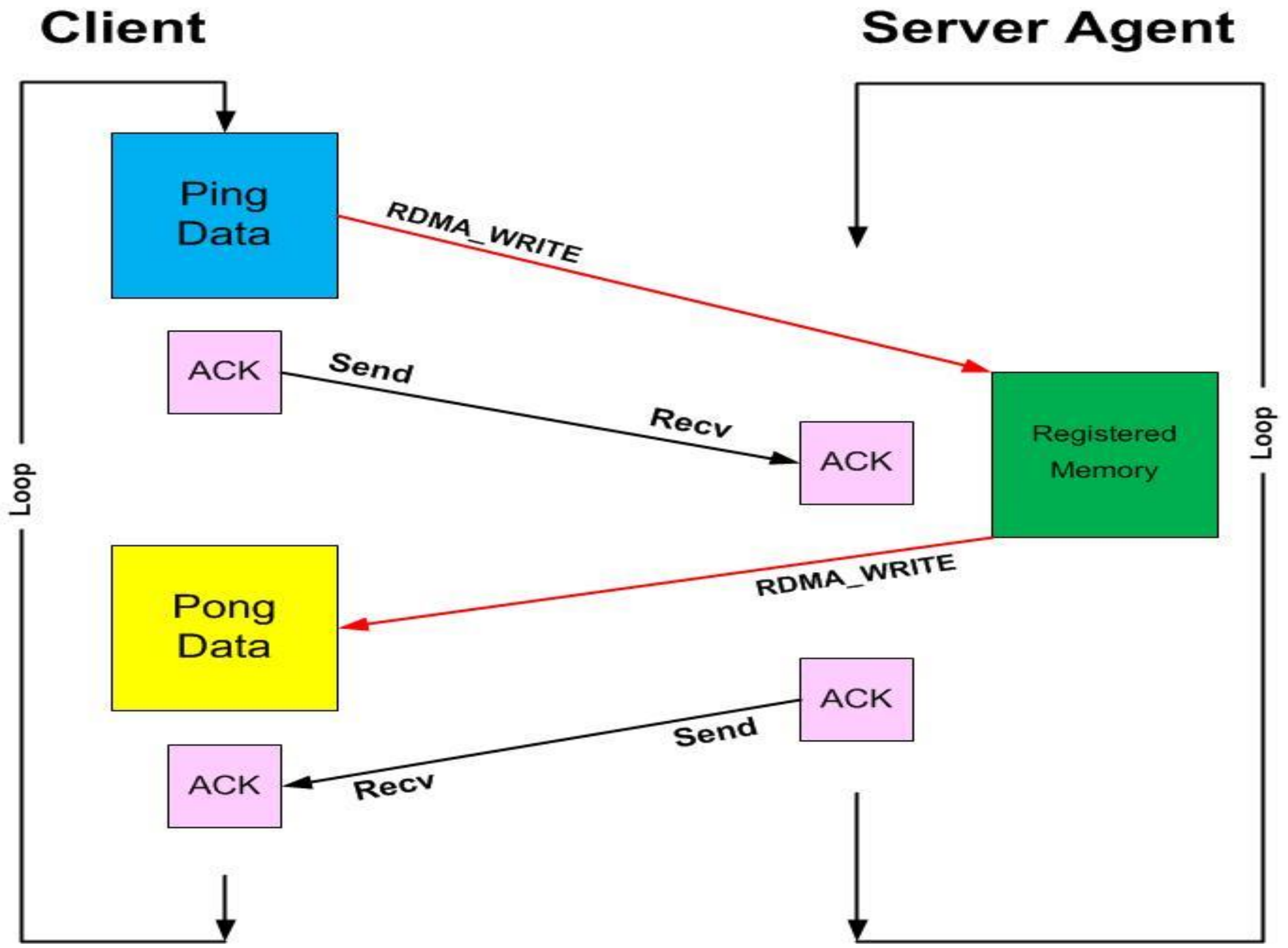
Ping-pong example ping-rdma-2

- Run it and compare round-trip time with previous demo
- Diff the client.c files to show how minor the differences are
- Walk the code as necessary

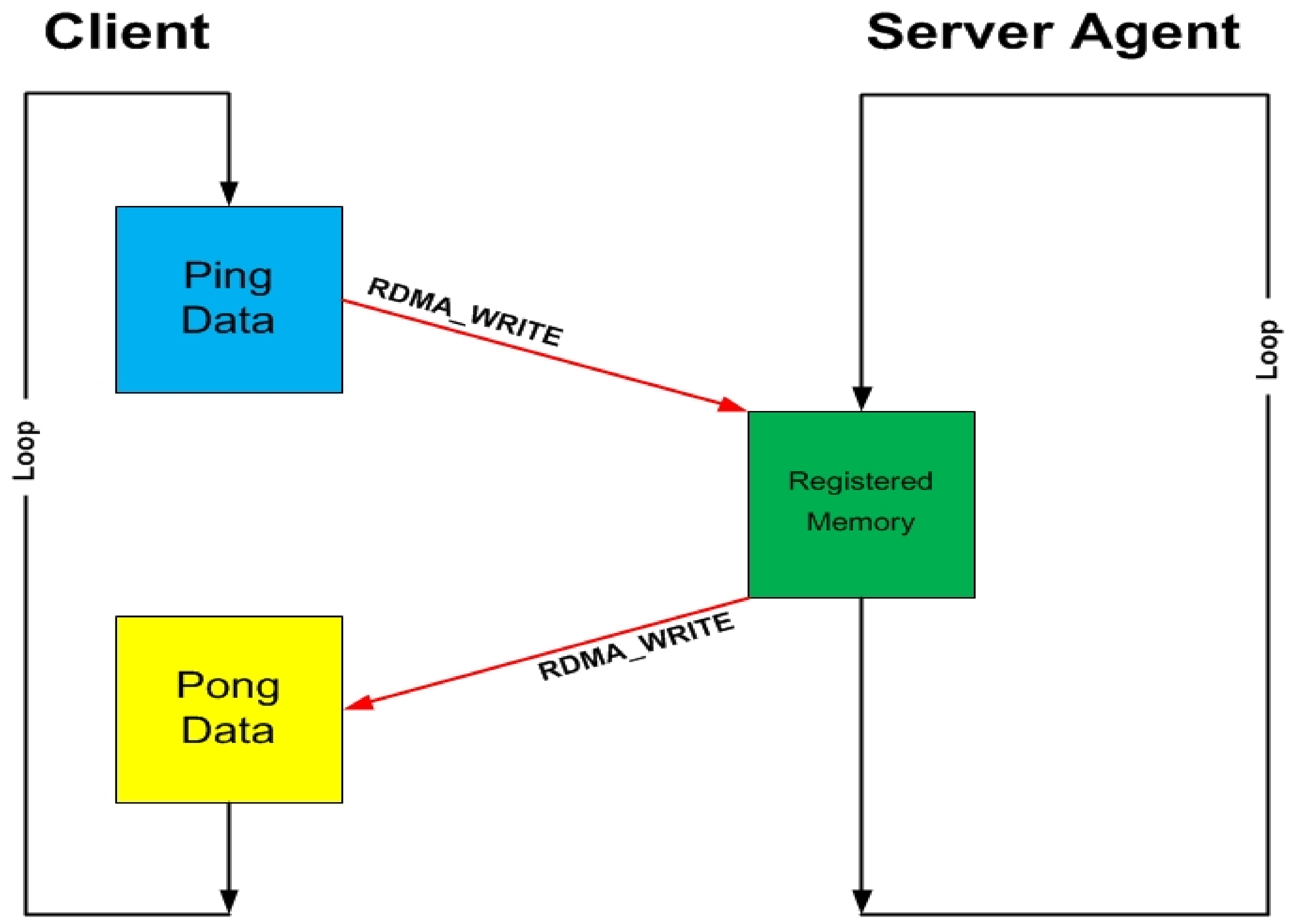
Ping-Pong using only RDMA_WRITE

- Use RDMA_WRITE on both client and agent
  - Client does rdma_write of ping data
  - Agent does rdma_write of pong data
- Problem – how does each side know when it is its turn to act?
- Solutions
  - Exchange acks using send/recv
    - Client sends ack after it completes its rdma_write
    - Agent sends ack after it completes its rdma_write
  - Use memory patterns with no acks exchanged

Using RDMA_WRITE with ack



Using RDMA_WRITE without ack



Ping using RDMA_WRITE with ack

- Use RDMA_WRITE followed by an ack
  - Client does rdma_write to deliver ping data to agent
  - Client sends ack to agent
  - Agent receives ack from client
  - Agent does rdma_write to deliver pong data to client
  - Agent sends ack to client
  - Client receives ack from agent

Performance issues

- Following each RDMA_WRITE with a send of an ack effectively doubles the time to do a ping, and doubles the time to do a pong
- Might as well just do a send of the ping data and a send of the pong data (as we did in earlier demos)
- How can we avoid the “extra” ack sends?

Token Passing

•These acks effectively “pass the token” from current writer to next writer

1. Client starts with token
2. Client does rdma_write to agent
3. Client passes token to agent (sends ack)
4. Agent gets the token (receives ack)
5. Agent does rdma_write to client
6. Agent passes token to client (sends ack)
7. Client gets token from agent (receives ack)
8. Loop to step 2

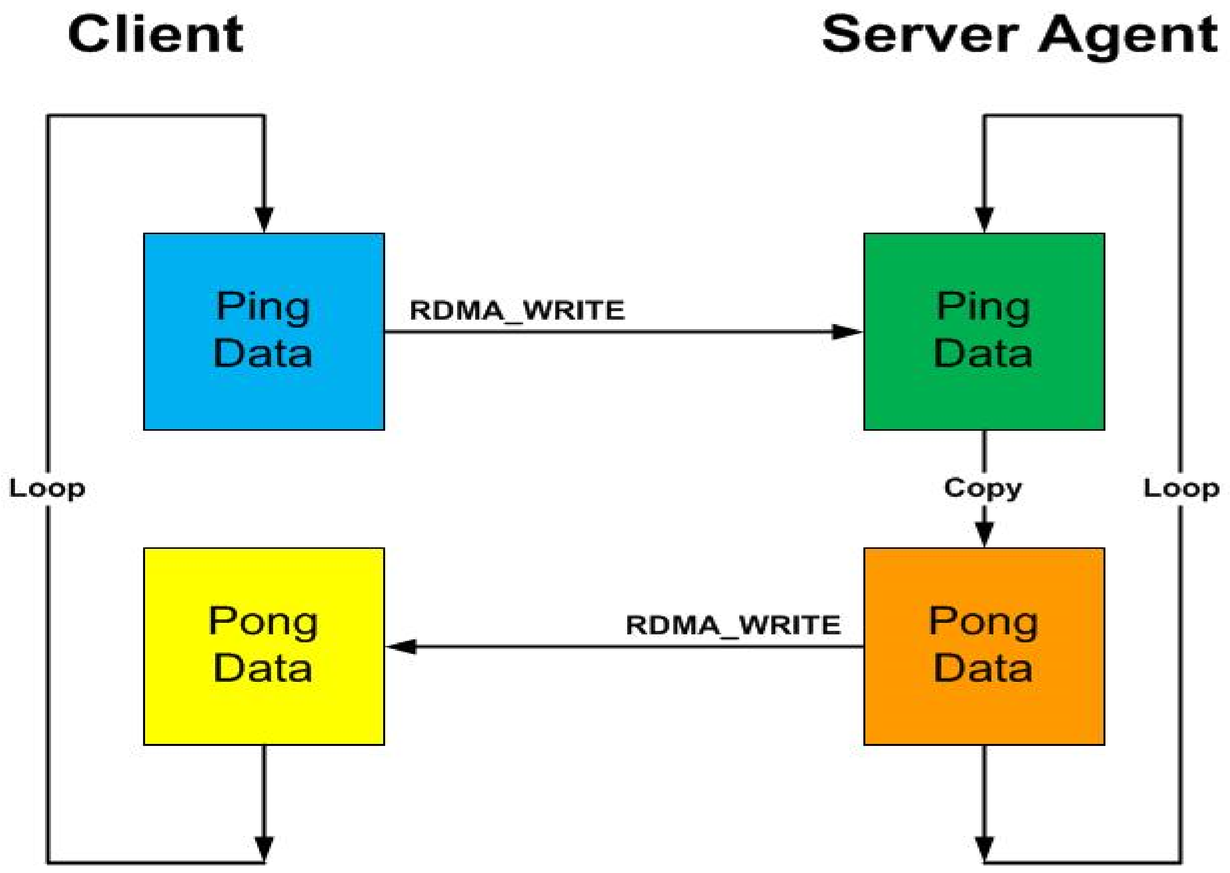
Using the Data as the Token

- Instead of passing the token separately from data, use the data as the token
  - That's how it worked with Send/Recv!!
- Requires the side receiving the data to somehow recognize that it has ALL arrived
- Simple way – detect a change in buffer contents
  - Requires busy waiting on the data buffer
  - Must be sure ENTIRE buffer has been changed
- One possible technique
  - Fill receive buffer with character not in data
  - Busy wait until character is nowhere in buffer

Ping-Pong example ping-rdma-3

- Use RDMA_WRITE without ack
- Client needs 2 buffers (as before)
- Agent also needs 2 buffers (before had only 1)
  - Client does rdma_write into agent's ping buffer
  - Agent copies data from ping buffer to pong buffer
  - Agent does rdma_write from pong buffer to client
- Each side needs to detect arrival of ALL data from other side
- Done by busy waiting for change in contents of ENTIRE receive buffer

Data flow in ping-rdma-3



Client buffers and pattern use

1. Client fills ping buffer with data pattern containing only printable characters
2. Client fills pong buffer with different pattern containing all binary zeroes ('\0' character)
3. Client does RDMA_WRITE of ping data to agent
4. Client waits for RDMA_WRITE to complete
5. Client busy waits until all binary zeroes in pong buffer have been completely replaced
6. Client optionally verifies data in pong buffer
7. Client loops to step 2

Agent buffers and pattern use

1. Agent fills ping buffer with pattern containing all binary zeroes ('\0' character)
2. Agent busy waits until all binary zeros in ping buffer have been completely replaced
3. Agent copies ping buffer into pong buffer
4. Agent fills ping buffer with pattern containing all binary zeroes ('\0' character)
5. Agent does RDMA_WRITE of pong data to client
6. Agent waits for RDMA_WRITE to complete
7. Agent loops to step 2

Example ping-rdma-3

- Both sides take turns doing RDMA_WRITE
- User data is the token
- Each side busy waits on data buffer to detect token
- No extra ack messages are necessary

- Run demo
- Walk code as necessary

Ping-Pong using only RDMA_READ

•Use RDMA_READ on both client and server

–Server does rdma_read to get ping data from client

–Client does rdma_read to get pong data from server

•Problem – same as before – how to know when it's “your turn” to ack

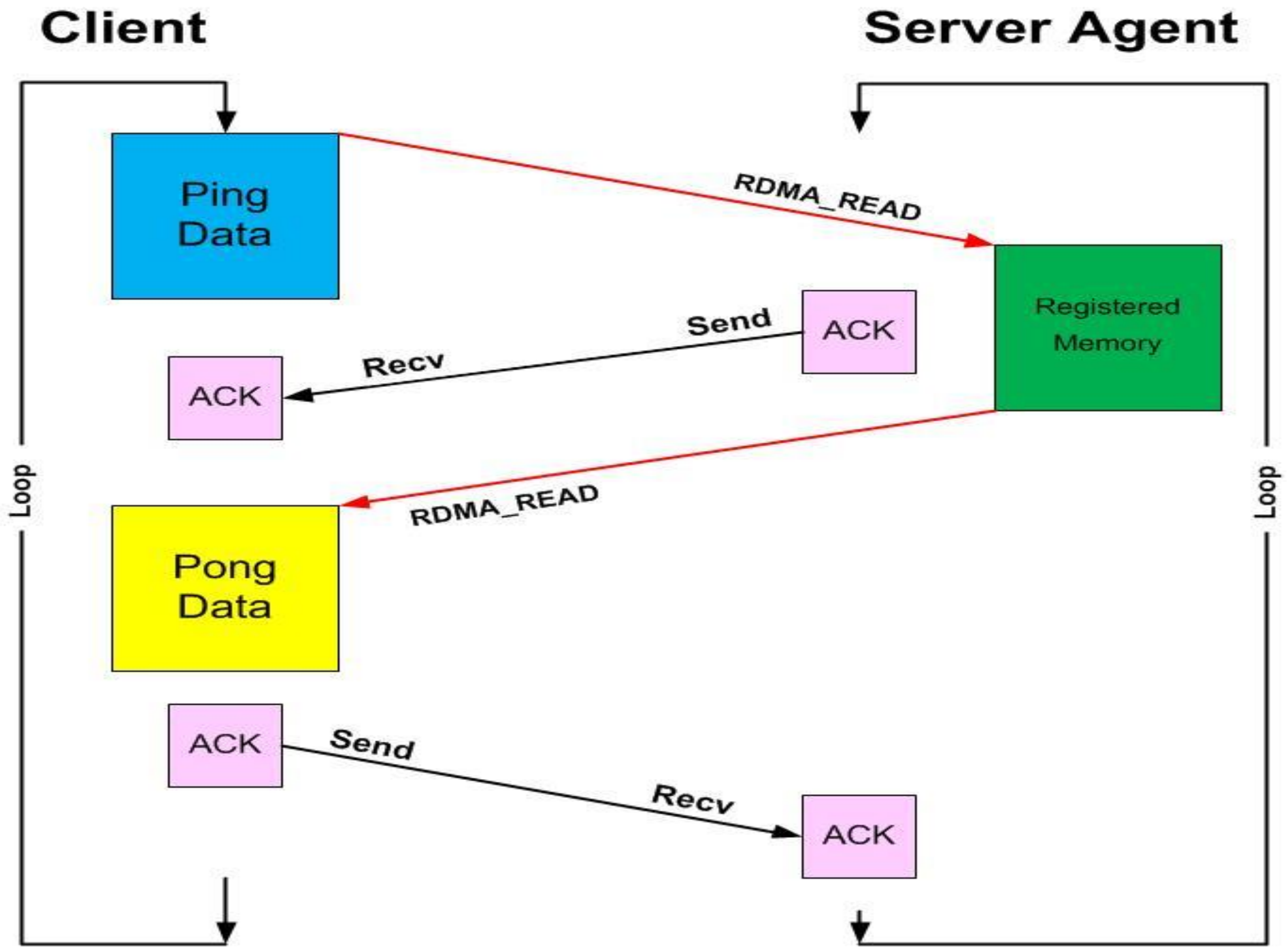
•Solution with acks to pass token

–Exercise for reader

•Solution without acks – is it possible?

–Challenge for reader to prove or disprove this

Using only RDMA_READ with ack



Ping using RDMA_READ with ack

•Use RDMA_READ with acks

–Agent does rdma_read to get ping data from client

–Agent sends ack to client

–Client receives ack from agent

–Client does rdma_read to get pong data from agent

–Client sends ack to agent

–Agent receives ack from client

Blast using only RDMA_WRITE

•Change our attention from ping-pong to blast

•Client uses chained work requests to blast as much data as possible to agent

•Agent does nothing

–CPU utilization is 0

•Client does everything

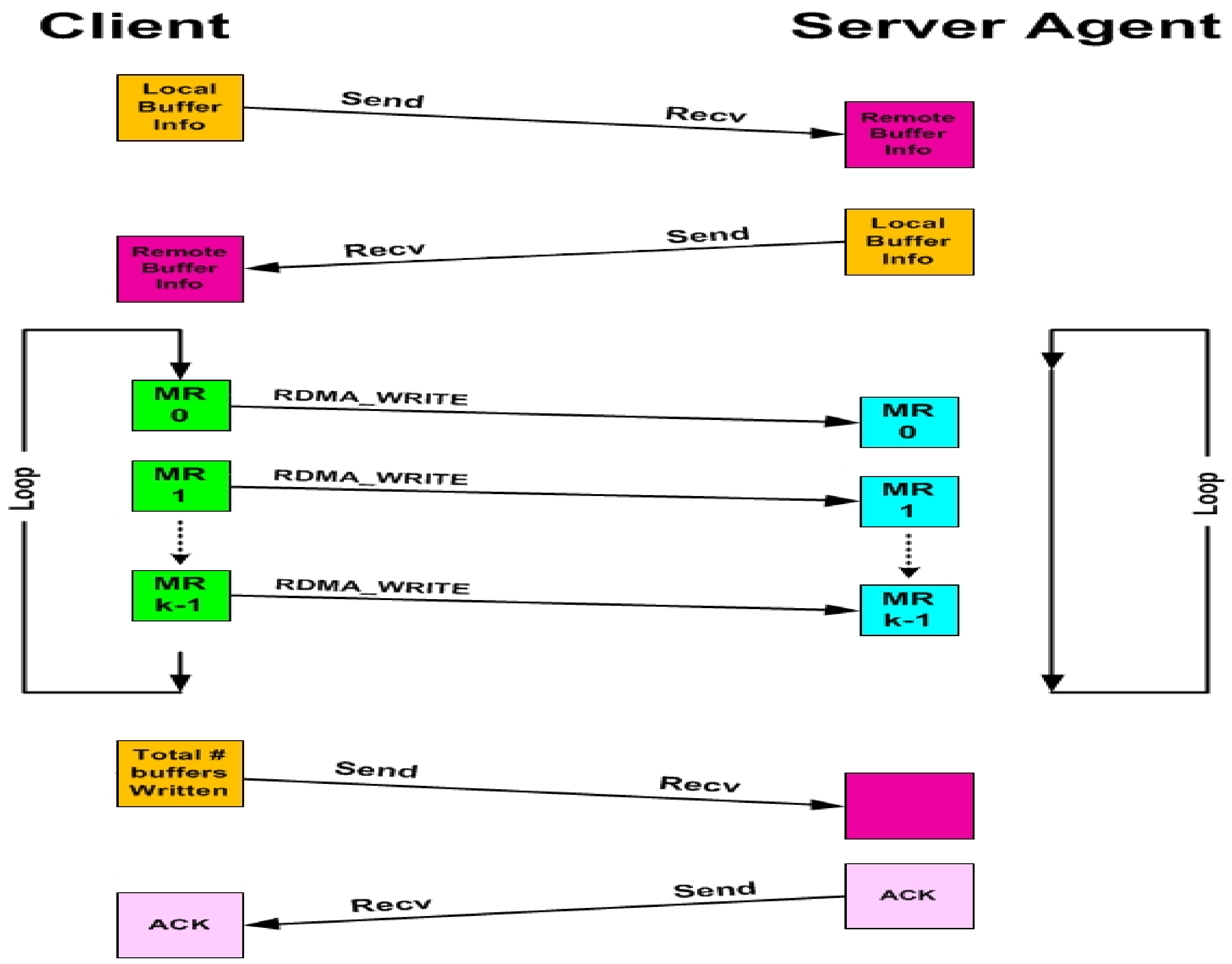
–Uses rdma_write with chain of all available SWRs

–For large enough message sizes, throughput is near maximum bandwidth available to user payload (i.e., after accounting for protocol headers, etc.)

Blast Buffers

- Arbitrary number of buffers in both client and agent
- Buffers are written by client to agent using chained work requests
- Buffers are reused on both sides
- Client does not synchronize buffer use by agent
- Need same organization of Use Phase
  - Perform buffer_info exchange using send/recv
  - Client blasts data using RDMA_WRITE
  - Perform end-of-run ack exchange using send/recv

Client drives blasting



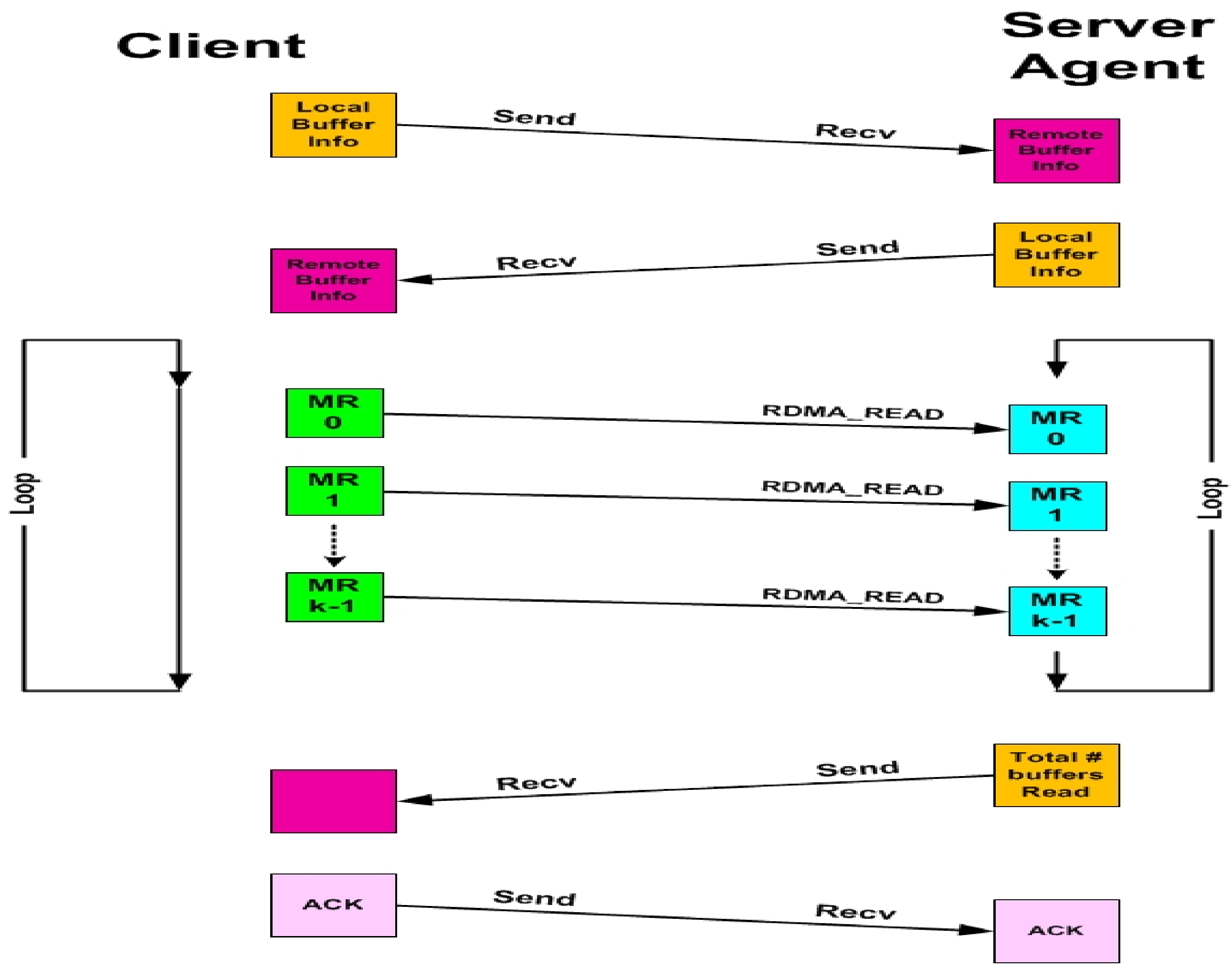
Blast example blast-rdma-1

- Client uses chained work requests to blast as much data as possible to agent with one **ibv_post_send()**
- Agent does nothing
  - CPU utilization is 0
- Client does everything
  - Uses rdma_write with chain of all available SWRs
- Run demo
- Walk code as necessary

Blast with agent doing all work

- Agent uses chained work requests to blast as much data as possible from client
- Client does nothing
  - CPU utilization is 0
- Agent does everything
  - Uses rdma_read with chain of all available SWRs
- In Lab, exercise for reader

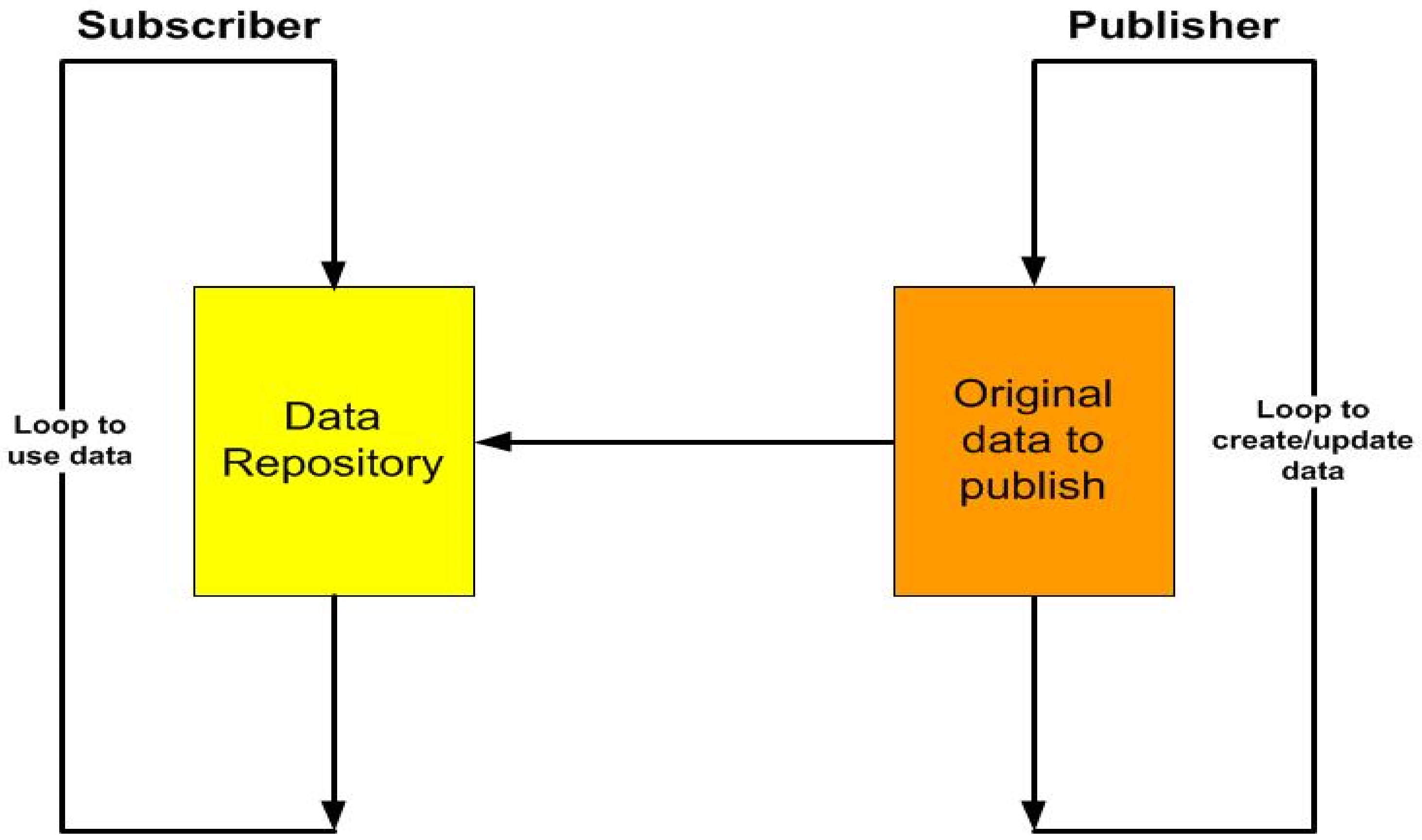
Agent drives blasting



Persistent access

- •RDMA_WRITE and RDMA_READ are both "one-sided"
  - –Side issuing the RDMA_WRITE or RDMA_READ is active – it does all the work
  - –Other side is passive – it does nothing
- •Leads to idea of persistent access
  - –Passive side gives access permission to active side only once
  - –Active side can issue RDMA_WRITE or RDMA_READ repeatedly to same memory region on passive side

Publish-Subscribe data flow

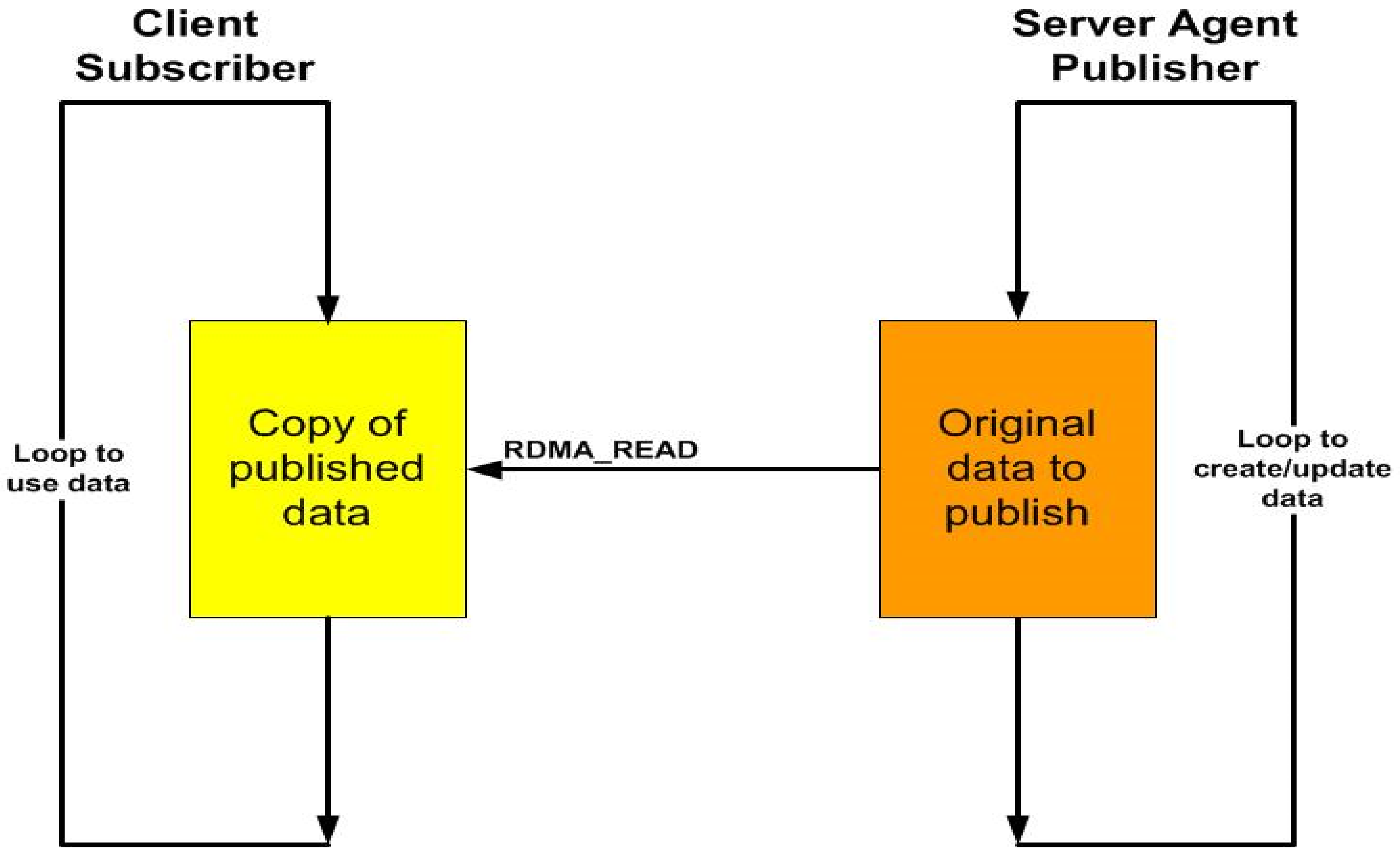


Publish-subscribe pub-sub-rdma-1

- Publisher (server) is passive side (issues no I/O)
  - Periodically updates its local data repository
- Subscriber (client) is active side
  - Periodically does RDMA_READ from server into local buffer

- Question – how does subscriber know ALL data in its local buffer is valid?

Data flow in pub-sub-rdma-1



Data Integrity in persistent access

- Problem caused by changes to block of virtual memory while it is being transferred onto wire
  - data transmitted may be only partially changed
- Solutions
  - For single “words”, memory hardware solves it
  - For small blocks, could transmit them twice
- Receiving side compares the blocks
- Uses them only if both equal
- Rereads if both not equal
- Depends on speed of update and speed of transfer
  - Safer to compute and include CRC in the transfer

Data Integrity using CRC

•Include CRC as part of data repository

•To update local data repository, publisher must

1.Change data in repository (thereby invalidating CRC)

2.Compute new CRC on updated repository into a temp

3.Store temp into CRC slot in local data repository

•To verify integrity of its local buffer, subscriber must

1.RDMA_READ repository from server into local buffer

2.Compute CRC on local buffer

3.If CRC checks ok, use this buffer

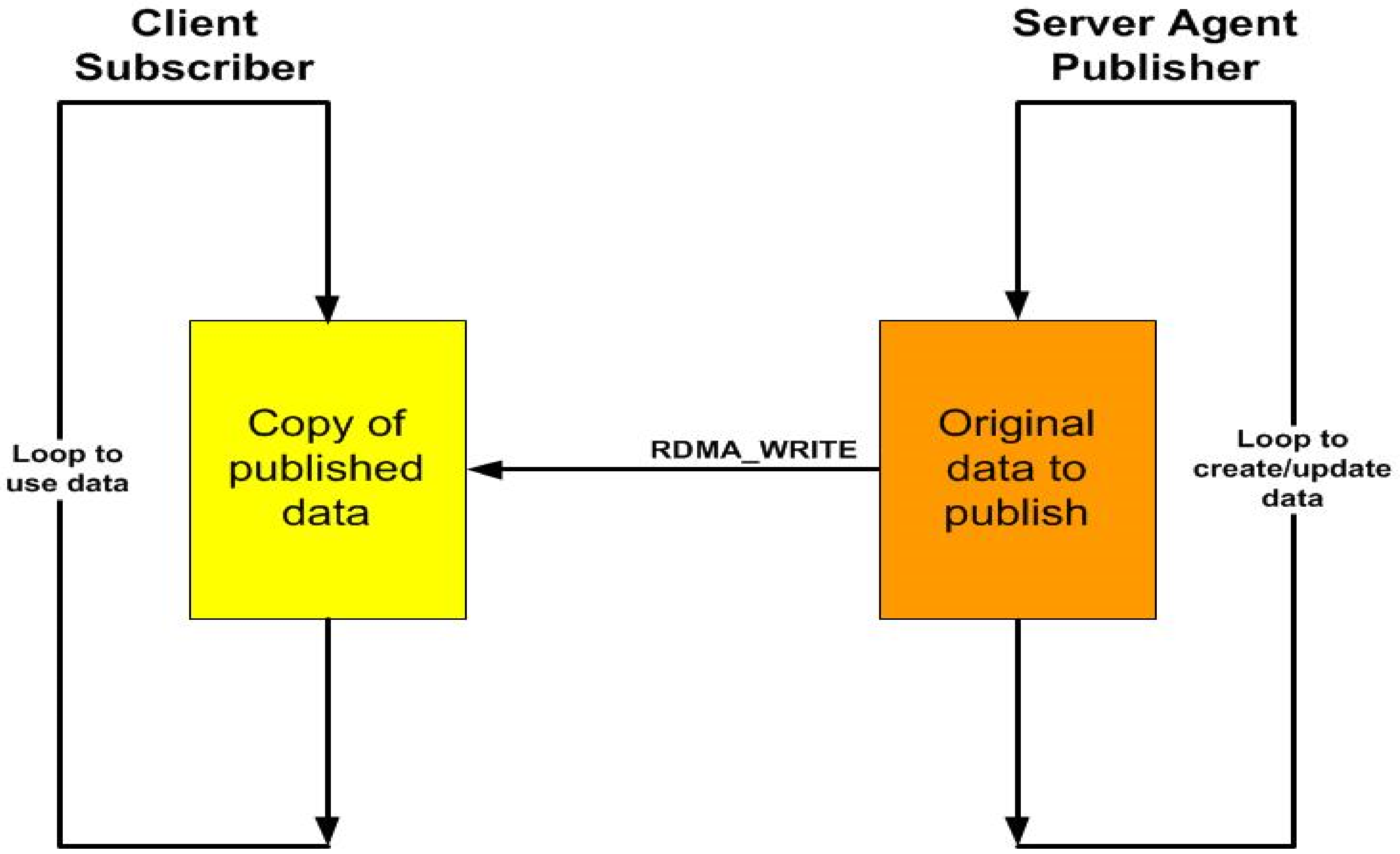
4.If CRC is bad, repeat at step 1 (because RDMA_READ was done while server was updating its

Publish-Subscribe pub-sub-rdma-2

- Server (publisher) is active side
  - Periodically updates its local data repository
  - After each update, server does RDMA_WRITE to remote buffer in client
- Client (subscriber) is passive side (issues no I/O)
  - Periodically looks at its local buffer

- Question – as before – how does subscriber know ALL the data in its local buffer is valid?

Data flow in pub-sub-rdma-2



Adding CRC to pub-sub-rdma-2

•Before doing RDMA_WRITE of its data to client, server must compute CRC of all data and include that in repository data written to client

•To verify integrity of its local repository, client must do:

1.Copy area written by server into another buffer

2.Compute CRC on that buffer

3.If CRC checks ok, use that buffer

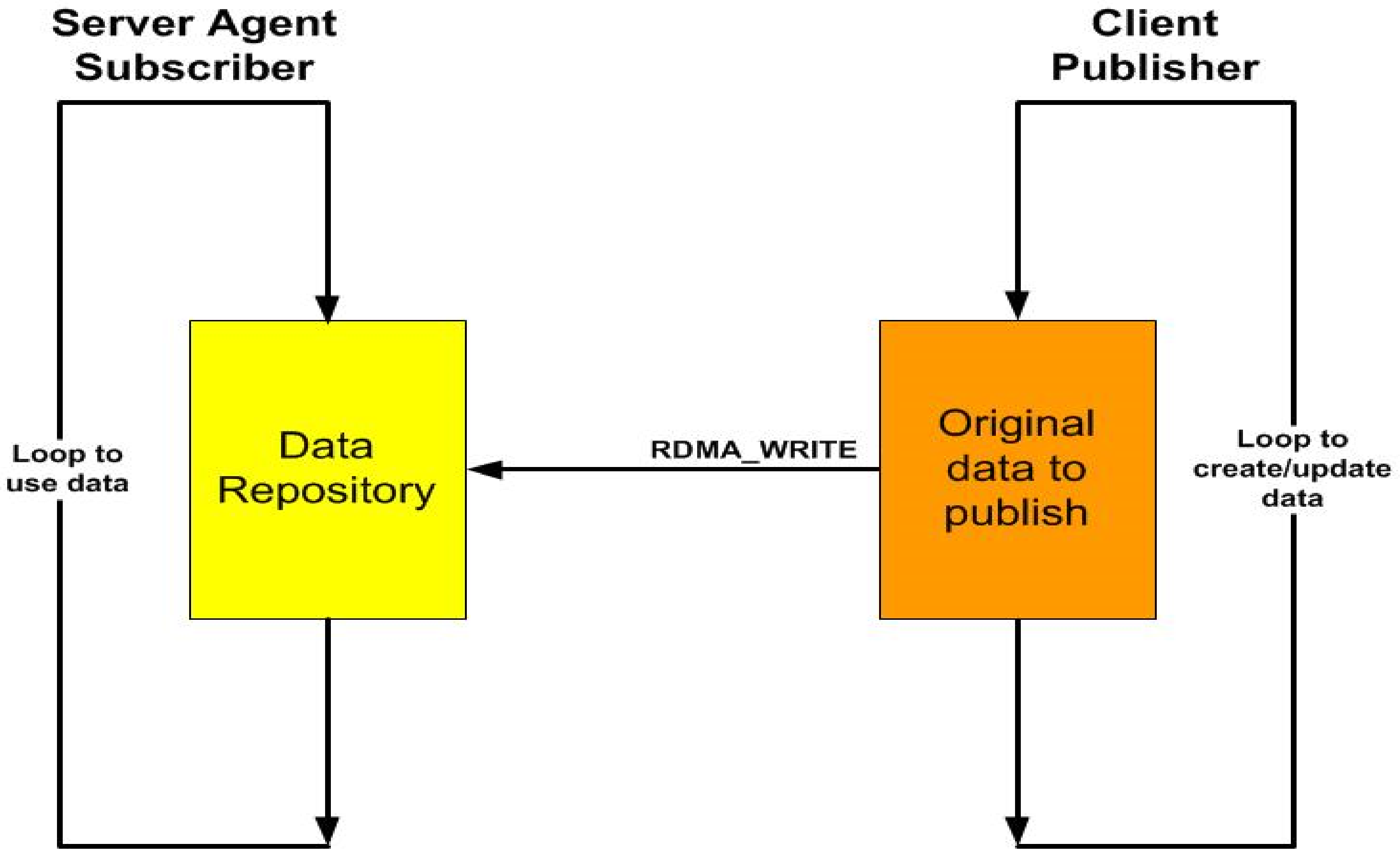
4.If CRC is bad, repeat at step 1 (because copy was made while server was transferring new data)

Exercise pub-sub-rdma-2e

- Client (publisher) is active side
  - Periodically updates its local data repository
  - After each update, client does RDMA_WRITE to remote buffer in agent
- Agent (subscriber) is passive side (issues no I/O)
  - Periodically looks at its local buffer
- Question – as before – how does subscriber know ALL the data in its local buffer is valid?

- Exercise for the reader in lab

Data flow in pub-sub-rdma-2e



Unreliable Datagram (UD) Mode

•Implemented only on IB, not iWARP

•Port space parameter to rdma_create_id has value RDMA_PS_UDP

•Can only use Send/Recv, **not** RDMA_WRITE or RDMA_READ

–Both sides must actively participate in transfer

–One side does ibv_post_recv(), other ibv_post_send()

•Transmission is still in messages, not streams

•Messages can be lost, since route not resolved in advance

•Message size limited to link MTU of underlying technology

Multicast

- Only possible with IB, not iWARP
- Is optional to implement in IB
- Only possible in Unreliable Datagram (UD) mode
  - Only Send/Recv operations allowed
  - Both sides must actively participate in data transfers
- Allows single SEND to be delivered to RECV in multiple destinations
- Receiver must have RECV posted for next Send
- Receiver must process each RECV completion

Multicast-Publish-Subscribe

- Active processing of updates on both sides
- Server establishes multicast group
- Server maintains data repository, periodically updates and sends it to multicast group
- Clients join multicast group
- Clients need 2 outstanding ibv_post_recv()
- When client ibv_post_recv() completes
  - client posts another to receive in second buffer
  - then uses the data from first buffer
- Clients can leave multicast group at any time

Ping-Pong example ping-sr-4

- Set both cm event channel and completion channel into O_NONBLOCK mode
- Use POSIX poll() to wait for both cm events and completion events
- Closest in style to “normal” sockets synchronous programming
- No need for extra thread
- Run
- Walk the code

End of Part 2