Verbs programming tutorial

Dotan Barak

OpenSHMEM, 2014
A Senior Software Manager at Mellanox Technologies. I have more than 10 years experience in various roles such as:

- a manager
- a developer

Was involved in several documentation projects in verbs programming

- Man pages of libibverbs
- Wrote the “RDMA Aware Networks Programming User Manual”

Wrote tens of applications over verbs

- Over several verb(s) generations
- In different OS’s

Author of “RDMAmojo” - a blog on the RDMA technology
InfiniBand Advantages

- **Features:**
  - Remote Direct Memory Access (RDMA) – zero copy
  - Kernel bypass
  - Highly scalable (10K's of nodes)
  - Message based transactions
  - Credit based flow control

- **Performance**
  - High BW (56 Gb/sec)
  - Low latency (700 nsec)
  - High message rate (137 Mpps)
  - The lower 4 OSI layers implemented in HW

- **Other**
  - Industry Standard
  - Developed as Open Source (in *NIX)
  - Inbox in several OS’s
InfiniBand Hardware Components
Channel Adapters

- Channel Adapters (CAs) are the end nodes in the subnet.
  - They create/consume packets

- There are two types of CAs:
  - HCA – CA which supports the verbs interface
  - TCA – “weak” CA, does not have to support many features

- xCAs are located in hosts/systems and they have a DMA engine that allows to initiate local and remote DMA operations.

- xCA has at least one port
Switches allow routing of a packet in a subnet according to the packet’s local destination addressing
- Very fast routing (140-200 nsec)
- Only Unicast packets are routed
- Multicast packets may be duplicated across some/all ports

A switch has at least three ports
Router and Gateway

- Routers allow routing of packets between different subnets according to the packet’s destination Global address

- Gateways allow encapsulation/de-capsulation of InfiniBand packets in other protocol packets

- Routers and Gateways have at least one port per subnet
Links connect xCAs, switches, routers, gateways together to create a subnet
• Every link is full duplex
  - Lines(s) for TX
  - Lines(s) for RX
• Link can be:
  - Copper cable
  - Optical cable
  - Printed circuit

Repeaters are transparent physical entities that extend the range of a link
InfiniBand subnet
InfiniBand connected subnets
RDMA Data Model
InfiniBand Addressing

- Every node in InfiniBand has a Global Unique Identifier (GUID) – Node GUID
  - Persistent
  - World-wide unique 64 bits value (node GUID)
- Every port in a node has a port GUID
  - Ports can be identified using the port GUID
  - Every port can be configured with multiple additional GID (Global IDentifier) addresses in the port’s GID table
- A system, which contains several nodes, may have a System GUID configured in each node
A subnet is configured and managed by a Subnet Manager (SM)

- One centralized entity which configures and manages the subnet
  - Fills various tables in the node’s ports
  - Fills the forwarding tables and more switch tables
- The SM assigns each port in an end-node a Local Identifier (LID)
  - 16 bits value
  - Unique in a specific subnet
  - Port may have one or range of LIDs
  - All switch ports have the same LID
- One is active and the other if exists is in standby state
  - One of them becomes active if the active one cannot function
The Classic Data Transfer Model (Sockets)

- The receiver prepares the receive buffer(s)
- The sender prepares the send buffer(s)

Points:
- Network software stack involved
  - Usually part of the OS’s kernel
  - Consumes CPU
  - If needed, provides reliability
  - Caches the data
- Order of operations does matter
  - Receive buffer(s) **should** be available before data is received
RDMA Data Transfer Model

- **Send**
  - Just like the classic model
  - Data is read in local side
    - Can be gathered from multiple buffers
  - Sent over the wire as a message
  - Remote side specify where the message will be stored
    - Can be scattered to multiple buffers

- **RDMA**
  - Local side can write data directly to remote side memory
    - Can be gathered locally from multiple buffers
  - Local side can read data directly from remote side memory
    - Can be scattered locally to multiple buffers
  - Remote side isn’t aware to any activity
    - No CPU involvement at remote side
RDMA Data Transfer Model – Work Queues

- **Work Request (WR)**
  - Work items that the HW should perform

- **Work Completion (completion)**
  - When a WR is completed, it may create a Work Completion which provides information about the ended WR
    - Type, opcode, amount of data send/received, more

- **Work Queue (WQ)**
  - A queue which contains WRs
  - Scheduled by the HW
    - WR execution ordering is guaranteed within the same WQ – FIFO
    - There is no guarantee about the order between different Work Queues
  - Can be either Send or Receive Queue
  - Adding a WR to a WQ is called “posting a WR”
  - Every WR that was posted is considered “outstanding” until it ends with Work completion. While outstanding:
    - One cannot know if it was scheduled by the HW or not
    - Send buffer(s) cannot be (re)used/freed
    - Receive buffer(s) content is undetermined
RDMA data transfer model – Work Queues (cont.)
RDMA Data Transfer Model – Work Queues (cont.)

- **Send Queue (SQ)**
  - A Work Queue that handles sending messages
  - Every entry is called a Send Request (SR). It specifies:
    - How data is used
    - What memory buffers to use
      - To send or receive data – depends on the opcode
    - How much data is sent
    - More attributes
  - Adding a SR to an SQ is called “posting a Send Request”
  - SR may end with a Work Completion

- **Receive Queue (RQ)**
  - A Work Queue that handles incoming messages
  - Every entry is called a Receive Request (RR). It specifies:
    - Memory buffers to be used (If RR is consumed – depends on opcode)
  - Adding a RR is to the RQ called “posting a Receive Request”
  - An RR always ends with a Work Completion
  - This queue may send data as a response – depends on opcode

- **Queue Pair (QP)**
  - An object which unifies both Send and Receive Queues
  - Every Queue is independent
  - Every QP is associated with a Partition Key (P_Key)
RDMA Data Transfer Model - Roles

- **Requester**
  - The active side
    - It initiates the data transfer
  - Post SR(s)

- **Responder**
  - The passive side
    - It sends or receives data – depend on the used opcode
  - May Post RR(s)
  - Send ACK/NACK for reliable transport types
RDMA opcodes: Send

- The responder Post Receive Requests (before data is received)
- The requester Post Send Request
  - Only data is sent over the wire
- ACK is sent only in reliable transport types
RDMA opcodes: RDMA Write

- The requester Post Send Request
  - Data and remote memory attributes are sent
  - Responder is passive
  - Immediate data can be used to consume RRs at the responder side
- ACK is sent only in reliable transport types
RDMA opcodes: RDMA Read

- The requester Post Send Request
  - Data and remote memory attributes are sent
  - Responder is passive

- Data is sent from the responder
  - Available only in reliable transport types

Diagram:
- Requester
  - Post SR
  - addr + rkey
  - Poll CQ
- Responder
  - data

© 2014 Mellanox Technologies
RDMA opcodes: Atomic

- **The requester Post Send Request**
  - Data and remote memory attributes are sent
  - Responder is passive
- **Original data is sent from the responder**
  - Read-modify-write is performed in responder’s memory
  - Available only in reliable transport types

![Diagram](image-url)
Fetch and Add
- The following is done in atomic way for 64-bits numbers at responder’s memory:
  - Fetch data from memory
  - Add a value
  - Store the new number
- Send the original value to the requester

Compare and Swap
- The following is done in atomic way for 64-bits numbers at responder’s memory:
  - Fetch data from memory
  - Compare the data with number1
    - If they are equal - store number2 in memory
- Send the original value to the requester
### RDMA opcodes summary

<table>
<thead>
<tr>
<th></th>
<th>Send</th>
<th>RDMA Write</th>
<th>RDMA Read</th>
<th>Atomic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data direction</strong></td>
<td>requester -&gt; responder</td>
<td>requester -&gt; responder</td>
<td>requester &lt;-&gt; responder</td>
<td>requester &lt;-&gt; responder</td>
</tr>
<tr>
<td><strong>Require RR at responder</strong></td>
<td>Yes</td>
<td>Only in case of RDMA Write with immediate</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Requester memory operations</strong></td>
<td>Gather data (multiple buffers)</td>
<td>Gather data (multiple buffers)</td>
<td>Scatter data (multiple buffers)</td>
<td>Scatter data (multiple buffers)</td>
</tr>
<tr>
<td><strong>Responder memory operation</strong></td>
<td>Scatter data (multiple buffers)</td>
<td>Write data (contiguous block)</td>
<td>Write data (contiguous block)</td>
<td>Write data (contiguous block)</td>
</tr>
<tr>
<td><strong>Atomicity</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Verbs
Prerequisites

- **Software**
  - Linux distribution
  - RDMA stack installed
    - MLNX-OFED
    - Community's OFED
    - Native RDMA stack within the Linux distribution
    - Download manually the needed libraries and compile the Linux kernel

- **Hardware**
  - RDMA device
    - InfiniBand/RoCE is preferred
    - Connected in loopback or back-to-back/switch
What is the “verbs”?

- Verbs is an abstract description of the functionality that is provided for applications for using RDMA.
  - Verbs is not an API
  - There are several implementations for it
- Verbs can be divided into two major groups
  - Control path – manage the resources and usually requires context switch
    - Create
    - Destroy
    - Modify
    - Query
    - Work with events
  - Data path – Use the resources to send/receive data and doesn’t require context switch
    - Post Send
    - Post Receive
    - Poll CQ
    - Request for completion event
Why use verbs?

- Verbs is a low level description for RDMA programming
  - Verbs are close to the “bear-metal” and provide best performance
    - Latency
    - BW
    - Message rate
  - Verbs can be used as building blocks for many applications
    - Sockets
    - Storage
    - Parallel computing

- Any other level of abstraction over verbs may harm the performance

- Once you know it, verbs are not so mysterious …
libibverbs, developed and maintained by Roland Dreier since 2006, are de-facto the verbs API standard in *nix

- Developed as an Open source
- The kernel part of the verbs is integrated in the Linux kernel since 2005 – Kernel 2.6.11
- Inbox in several *nix distributions
- There are level low-level libraries from several HW vendors

Same API for all RDMA-enabled transport protocols

- **InfiniBand** – Networking architecture which supports RDMA
  - requires both NICs and switches that supports it.
- **RDMA Over Converged Ethernet (RoCE)** – encapsulation of RDMA packets over Ethernet/IP frames
  - requires NICs which supports it and standard Ethernet switches
- **Internet Wide Area RDMA Protocol (iWARP)** – provides RDMA over Stream Control Transmission Protocol (SCTP) and Transmission Control Protocol (TCP)
  - requires NICs which supports it and standard Ethernet switches
libibverbs: Thread Support

- libibverbs is completely thread safe
  - libibverbs itself is thread safe
  - The userspace low-level driver libraries are thread safe as well
  - Application can use RDMA resources in multiple threads

- Warning: libibverbs will not prevent you from getting into troubles
  - Destroying a resource in one thread and using it in another thread will end up with segmentation fault
    - This happens in non-thread code as well
libibverbs: fork() support

- **When possible, avoid using fork()**
  - fork() or other system calls that call fork(), for example: system(), popen(), etc.

- **If fork() must be used, the next rules should be followed:**
  - Prepare libibverbs to work with fork():
    - Call the verb `ibv_fork_init()`
    - or
      - Setting the environment variables `RDMAV_FORK_SAFE` or `IBV_FORK_SAFE`
    - This will allocate internal structures in way which is more “fork()”-friendly
  - RDMA should be used only in the parent process
  - Child process should call immediately exec*() or exit()
  - If huge pages are used, set the environment variable `RDMAV_HUGEPAGES_SAFE` as well

- **Warning:** Not following those rules may lead to data corruption or segmentation fault!
General tips on libibverbs

- Source code that uses libibverbs should include the header:
  - `#include <infiniband/verbs.h>`
- Executables/libraries that work with libibverbs should be linked with:
  - `-libverbs`
- All input structures should be zeroed
  - Using `memset()` or structure initialization
  - If the structure will be extended in the future, the value zero will keep the legacy behavior
- Most resource handles are pointers, so using bad handles may cause segmentation fault
- Verbs that return a pointer – return a valid value in case of a success and NULL incase of a failure
- Verbs that return an integer – return zero in case of a success and -1 or `errno` incase of a failure
  - For more information, read the documentation of each verb
RDMA resource creation hierarchy

Mandatory

Optional

struct ibv_device
   struct context
      struct ibv_pd
         struct ibv_mr
         struct ibv_ah
         struct ibv_srq
      struct ibv_comp_channel
      struct ibv_cq
      struct ibv_qp
In every machine, there can be one or more RDMA devices
  • Each RDMA device is a PCI device
  • Multiple devices can be:
    - From the same or different vendors
    - From the same or different models

One should check which devices are available, and open the requested device(s)
  • Every device has a unique name (at specific point in time) and a node GUID which is unique and persistent
  • GUID should be used to identify the device

Every RDMA device is independent
  • Every resource exists in the scope of the RDMA device it was created in
  • Resources cannot be shared between different devices
Working with RDMA devices: API

- struct ibv_device **ibv_get_device_list(int *num_devices);
  - Return a NULL-terminated list of RDMA device that exist in the local host
  - num_devices is optional. NULL can be provided instead of a valid pointer.
  - Notice the following fields in struct ibv_device:
    - node_type - The node type of this device (xCA, switch, etc.)
    - transport_type – The transport type used by this device (IB, iWARP, etc.)

- void ibv_free_device_list(struct ibv_device **list);
  - Free the list of RDMA devices that was returned from ibv_get_device_list()
  - This verb should be called after the required RDMA device was opened

- const char *ibv_get_device_name(struct ibv_device *device);
  - Return a string that describe the name of the RDMA device

- uint64_t ibv_get_device_guid(struct ibv_device *device);
  - Return the GUID of the RDMA device
  - This verb should be used to identify a device since the GUID is both unique and persistent
struct ibv_context *ibv_open_device(struct ibv_device *device);
  • Return a context from an RDMA device
    - This context will be used to create resources in this device
  • Notice the following fields in struct ibv_context:
    - num_comp_vectors – Number of completions vectors that this device supports
    - async_fd – File descriptor that will be used to report about asynchronous events from kernel

int ibv_close_device(struct ibv_context *context);
  • Close the context of the RDMA device
    - This verb should be called after destroying all the resources that were created using this context
      ▪ Not doing so will cause a memory leak
struct ibv_device **device_list;
int num_devices;
int i;

device_list = ibv_get_device_list(&num_devices);
if (!device_list) {
    fprintf(stderr, "Error, ibv_get_device_list() failed\n");
    exit(1);
}

for (i = 0; i < num_devices; ++ i)
    printf("RDMA device[%d]: name=%s\n", i, ibv_get_device_name(device_list[i]));

ibv_free_device_list(device_list);
Exercise 1:

Write a program that go over all the RDMA devices and print for every device its node type.

Tip: use `ibv_node_type_str()` to get a string from a node type enumerated value.
Context queries

- Every context supports several queries
  - Queries about the device attributes
  - Queries about a specific port properties
    - Or about tables within a port, such as GID and P_Key tables

- Some attributes are constant
  - Most of the device attributes are constant

- Some attributes are dynamic
  - Most of the port attributes are dynamic (LID, MTU, etc.)
Context queries: API

- **int ibv_query_device(struct ibv_context *context, struct ibv_device_attr *device_attr);**
  - Query for the device attributes
    - For number of objects, the mentioned values are the higher limit
    - Important properties: supported number of objects, number of physical ports

- **int ibv_query_port(struct ibv_context *context, uint8_t port_num, struct ibv_port_attr *port_attr);**
  - Query for specific port properties
    - First port number is: 1
    - Important properties: LID, MTU, supported VL, number of P_Keys, number of GIDs

- **int ibv_query_pkey(struct ibv_context *context, uint8_t port_num, int index, uint16_t *pkey);**
  - Query for the value in a specific index in a port’s P_Key table
    - First index is: 0
    - Index 0 contains the default P_Key value (0xffff)

- **int ibv_query_gid(struct ibv_context *context, uint8_t port_num, int index, union ibv_gid *gid);**
  - Query for the value in a specific index in a port’s GID table
    - First index is: 0
    - Index 0 is a special (constant) value – it contains the port GUID
struct ibv_port_attr port_attr;
int port_num = 1;

rc = ibv_query_port(ctx, port_num, &port_attr);
if (rc) {
    fprintf(stderr, "Error, failed to query port %d attributes in device '%s'\n",
            port_num, ibv_get_device_name(ctx->device));
    return -1;
}
Exercise 2:

Write a program that go over all the RDMA devices and print for every port in each device the GID in entry 0 (i.e. Port GUID)
Protection Domain (PD)

- Protection Domain is a mechanism for associating Queue Pairs with other RDMA resources
  - Such as Memory Regions and Address Handles
- Not all resources have a PD
  - For example: Completion Queues
- Protection Domain as its name state is a mean of protection
  - Mixing resources that were associated with different PDs will result a Work Completion with error
- Every Protection Domain can be imagined as a different color
  - Resources that are associated with a Protection Domain get its color
  - Resources from different colors can’t work together
Protection Domain (PD): API

- `struct ibv_pd *ibv_alloc_pd(struct ibv_context *context);`
  - Create a Protection Domain

- `int ibv_dealloc_pd(struct ibv_pd *pd);`
  - Destroy a Protection Domain

  - This verb should be called after destroying all the resources that are associated with it
struct ibv_context *context;
struct ibv_pd *pd;

pd = ibv_alloc_pd(context);
if (!pd) {
    fprintf(stderr, "Error, ibv_alloc_pd() failed\n");
    return -1;
}

...

if (ibv_dealloc_pd(pd)) {
    fprintf(stderr, "Error, ibv_dealloc_pd() failed\n");
    return -1;
}
Memory Region (MR)

- Memory Region is a virtually contiguous memory block that was registered, i.e. prepared for work with RDMA.
  - Any memory buffer in the process’ virtual space can be registered
  - Available permissions. One or more of the following permissions (Or’ed):
    - Local operations (Local Read is always supported)
      - IBV_ACCESS_LOCAL_WRITE
      - IBV_ACCESS_MW_BIND
    - Remote operations
      - IBV_ACCESS_REMOTE_WRITE
      - IBV_ACCESS_REMOTE_READ
      - IBV_ACCESS_REMOTE_ATOMIC
  - If Remote Write or Remote Atomic is enabled, local Write should be enabled too
  - The same memory buffer can be registered multiple times
    - even with different permissions
  - After a successful memory registration, two keys are being generated:
    - Local Key (lkey)
    - Remote Key (rkey)
  Those keys are used when referring to this MR in a Work Request
Memory pages pinning has advantages
- Memory pages are always present in RAM
  - Never swapped out
  - Low latency
- Address translation can be cached

But Memory pages pinning decreases the available memory for the kernel
- Only the process that registered it can use this memory
- This may cause high use of the swap file
- When too much memory is pinned, the kernel may start killing processes
  - Or machine may work very slow

Creating the various queues register memory as well..
Memory Region (MR): API

- `struct ibv_mr *ibv_reg_mr(struct ibv_pd *pd, void *addr, size_t length, enum ibv_access_flags access);`
  - Register a memory buffer with specific permissions

  - Notice the following fields in struct ibv_mr:
    - lkey - The local key of this MR
    - rkey - The remote key of this MR
    - addr – The start address of the memory buffer that this MR registered
    - length – The size of the memory buffer that was registered

- `int ibv_dereg_mr(struct ibv_mr *mr);`
  - Deregister a Memory Region

  - This verb should be called if there is no outstanding Send Request or Receive Request that points to it
struct ibv_pd *pd;
struct ibv_mr *mr;

mr = ibv_reg_mr(pd, buf, buf_size, IBV_ACCESS_LOCAL_WRITE);
if (!mr) {
    fprintf(stderr, "Error, ibv_reg_mr() failed\n");
    return -1;
}

...
Exercise 3:

Write a program that open a device, allocate a PD and register 3 MRs:
1) One that allow local read
2) One that allow local write
3) One that allow remote write
Completion Event Channel

- Completion Event channel is a mechanism for delivering notification about the creation of Work Completions in CQs that is attached to it.
  - Useful to reduce the CPU consumption
- This object will be used when creating new CQs
- One Completion Event channel can be used with multiple CQs
Completion Event channel: API

- struct ibv_comp_channel *ibv_create_comp_channel(struct ibv_context *context);
  - Create a new Completion Event channel
  - Notice the following fields in struct ibv_comp_channel:
    - fd – File descriptor that will be used to report about completion events from kernel

- int ibv_destroy_comp_channel(struct ibv_comp_channel *channel);
  - Destroy a Completion Event channel
  - This verb should be called after destroying all the CQs that are associated with it
struct ibv_context *context;
struct ibv_comp_channel *channel;

channel = ibv_create_comp_channel(context);
if (!channel) {
    fprintf(stderr, "Error, ibv_create_comp_channel() failed\n");
    return -1;
}

...

if (ibv_destroy_comp_channel(channel)) {
    fprintf(stderr, "Error, ibv_destroy_comp_channel() failed\n");
    return -1;
}
Completion Queue (CQ)

- Completion Queue is a Queue that holds information about completed Work Requests
  - Every Work Completion contains information about the corresponding completed Work Request

- A Completion Queue size is limited
  - If more Work Completions than its size are added, the CQ is overruled and all associated Work Queues are moved to the Error state
    - It is up to the user to make sure that the CQ size is enough
    - It is up to the user to empty the CQ in order to prevent CQ overrun

- One CQ can be shared with multiple queues
  - Several Queue Paris
  - Only Send Queues
  - Only Receive Queues
  - Mix of the above
Completion Queue (CQ): API

- `struct ibv_cq *ibv_create_cq(struct ibv_context *context, int cqe, void *cq_context, struct ibv_comp_channel *channel, int comp_vector);`
  - Create a new Completion Queue

  - Notice the following fields in `struct ibv_cq`:
    - `cqe` – The actual number of Work Completions that the CQ can hold
    - `cq_context` – The private context that the CQ is associated with

- `int ibv_destroy_cq(struct ibv_cq *cq);`
  - Destroy a Completion Queue

  - This verb should be called after destroying all the QPs that are associated with it

- `int ibv_resize_cq(struct ibv_cq *cq, int cqe);`
  - Resize an existing Completion Queue

  - The new size should be able to contain the Work Completions that currently populate the CQ
struct ibv_context *context;
struct ibv_comp_channel *channel;
struct ibv_cq *cq;

cq = ibv_create_cq(context, cq_size, NULL, channel, 0);
    or

cq = ibv_create_cq(context, cq_size, NULL, NULL, 0);
if (!cq) {
    fprintf(stderr, "Error, ibv_create_cq() failed\n");
    return -1;
}

...

if (ibv_destroy_cq(cq)) {
    fprintf(stderr, "Error, ibv_destroy_cq() failed\n");
    return -1;
}
Write a program that open a device, create a Completion Event channel and create 2 CQs:

1) One without any associated Completion Event channel
2) One with associated Completion Event channel
Queue Pair (QP)

- **Queue Pair is the actual object that transfers data**
  - It encapsulates both Send and Receive Queue
    - Each of them is completely independent
    - Send Queue can generate Work Completion for every Send Request or for specific Send Requests
    - Receive queue generates Work Completion for every completed Receive Request
  - Full duplex
  - A QP represent a real HW resource

- **There are three major transport types**
  - **Reliable Connected (RC)**
    - An RC QP is connected to a single RC QP
    - Reliability is guaranteed (ordering, integrity and arrival of all packets)
    - Supports operations that need ACK
  - **Unreliable Connected (UC)**
    - An UC QP connected to a single UC QP
    - Reliability is not guaranteed
  - **Unreliable Datagram (UD)**
    - An UD QP can send/receive messages to/from any UD QP
    - Reliability is not guaranteed
    - Multicast is supported
    - Each message is limited to one packet
<table>
<thead>
<tr>
<th>Metric</th>
<th>UD</th>
<th>UC</th>
<th>RC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliability</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Send (with immediate)</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>RDMA Write (with immediate)</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>RDMA Read</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Atomic operations</td>
<td></td>
<td></td>
<td>☺</td>
</tr>
<tr>
<td>Multicast</td>
<td>☺</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max message size</td>
<td>MTU</td>
<td>2GB</td>
<td>2GB</td>
</tr>
<tr>
<td>CRC</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
</tbody>
</table>
QP state machine

Create QP/EE → Reset → Modify QP/EE → Initialized → Modify QP/EE → Ready to Receive → Modify QP/EE → Ready to Send → SQ Error → SQ Drain → Modify QP/EE → Modified QP/EE

Notes:
- Destroy QP/EE Context can be called from any state and exits the state diagram.
- It is possible to transition from any state to either the Error state or the Reset state with the Modify QP/EE Verb, with the exception of Reset to Error.
- An error can be forced from any state, except Reset, with the Modify QP/EE Verb.

QP: Can Post & process Receive WRs & Send ACKs. EE: EE can be used to process incoming messages on RDs.

QP: Can Post & process Send WRs & Send ACKs. EE: EE can be used for outgoing WRs on RDs & process incoming messages on RDs.
<table>
<thead>
<tr>
<th>Reset</th>
<th>Init</th>
<th>RTR</th>
<th>RTS</th>
<th>SQD</th>
<th>SQE</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Post RR</td>
<td>Disallowed</td>
<td>Allowed</td>
<td>Allowed</td>
<td>Allowed</td>
<td>Allowed</td>
<td>Allowed</td>
</tr>
<tr>
<td>Post SR</td>
<td>Disallowed</td>
<td>Disallowed</td>
<td>Disallowed</td>
<td>Allowed</td>
<td>Allowed</td>
<td>Allowed</td>
</tr>
<tr>
<td>RR processing</td>
<td>Not processed</td>
<td>Not processed</td>
<td>Processed</td>
<td>Processed</td>
<td>Processed</td>
<td>Flushed with error</td>
</tr>
<tr>
<td>SR processing</td>
<td>Not processed</td>
<td>Not processed</td>
<td>Not processed</td>
<td>Processed</td>
<td>New WRs aren’t processed</td>
<td>Flushed with error</td>
</tr>
<tr>
<td>Incoming packets</td>
<td>Silently dropped</td>
<td>Silently dropped</td>
<td>Handled</td>
<td>Handled</td>
<td>Handled</td>
<td>Silently dropped</td>
</tr>
<tr>
<td>Outgoing packets</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>Initiated</td>
<td>Initiated</td>
<td>None</td>
</tr>
</tbody>
</table>
Connecting QPs

- Communication should be established between the connected QPs
  - Each side needs to know who is the other side
  - Each side needs to have information about the other side and the path to it
  - Each side needs to configure attributes that describe the send attributes

- Problem: How to connect QP X with QP Y?
  - We cannot transfer the needed information to establish the connection until the connection has already been established between them …

- Solutions:
  1. Exchange information Out Of Band
     - For example: over sockets
  2. Use Communication Manager (CM) ← this is the right way to connect QPs
Connecting QPs (cont.)

- The following information needs to be exchanged when connecting QPs
  - QP number
  - LID number
  - RQ Packet Serial Number (PSN)
  - GID (if GRH is used)

- Path MTU must be equal on both sides

- If RDMA opcodes are used, the permissions of QP and MR should be configured to support them

- In each QP state transition, the relevant attributes to enable the state functionality needs to be configured
  - There are different attributes for every transport type
    - For RC QPs: retransmission count and timers
    - For RC/UC QPs: Primary path and alternate path (optional)
Queue Pair (QP): API

- **struct ibv_qp *ibv_create_qp(struct ibv_pd *pd,**
  **struct ibv_qp_init_attr *qp_init_attr);**
  - Create a new Queue Pair
  - Notice the following fields in struct ibv_qp:
    - qp_num – The physical QP number
    - qp_context – The private context that the QP is associated with

- **int ibv_destroy_qp(struct ibv_qp *qp);**
  - Destroy a Queue Pair
  - This verb should be called after detach is from all multicast groups

- **int ibv_modify_qp(struct ibv_qp *qp, struct ibv_qp_attr *attr,**
  **enum ibv_qp_attr_mask attr_mask);**
  - Modify the QP attributes

- **int ibv_query_qp(struct ibv_qp *qp, struct ibv_qp_attr *attr,**
  **enum ibv_qp_attr_mask attr_mask,**
  **struct ibv_qp_init_attr *init_attr);**
  - Query the attributes of a QP
Queue Pair (QP): API (cont.)

```c
struct ibv_qp_cap {
    uint32_t max_send_wr;   - The number of Send Requests that can be outstanding in the QP
    uint32_t max_recv_wr;   - The number of Receive Requests that can be outstanding in the QP
    uint32_t max_send_sge;  - The number of S/G entries that each Send Request may hold
    uint32_t max_recv_sge;  - The number of S/G entries that each Receive Request may hold
    uint32_t max_inline_data; - The requested inline data (in bytes) to be sent
};

struct ibv_qp_init_attr {
    void *qp_context;                  - A private context that the QP will be associated with
    struct ibv_cq *send_cq;           - The CQ to be associated with the QP’s Send Queue
    struct ibv_cq *recv_cq;           - The CQ to be associated with the QP’s Receive Queue
    struct ibv_srq *srq;               - Optional: if not NULL, the SRQ to be associated with
    struct ibv_qp_cap cap;            - The QP attributes to be created
    enum ibv_qp_type qp_type;         - The QP transport type
    int sq_sig_all;                    - Indication if every completed Send Request will generate a Work Completion
};
```
Queue Pair (QP): example

```c
struct ibv_pd *pd;
struct ibv_cq *cq;
struct ibv_qp *qp;
struct ibv_qp_init_attr init_attr = {
    send_cq = cq,
    .recv_cq = cq,
    .cap = {
        .max_send_wr = 1,
        .max_recv_wr = rx_depth,
        .max_send_sge = 1,
        .max_recv_sge = 1
    },
    .qp_type = IBV_QPT_RC
};

qp = ibv_create_qp(pd, &init_attr);
if (!qp) {
    fprintf(stderr, "Error, ibv_create_qp() failed\n");
    return -1;
}

...

if (ibv_destroy_qp(qp)) {
    fprintf(stderr, "Error, ibv_destroy_qp() failed\n");
    return -1;
}
```
Data Transfer verbs
Every Work Request contains usually one or more S/G entries
- Every S/G entry refers to a Memory Region or part of it
- No S/G entries means zero-byte message
- Gather – when local data is read and sent over the wire
- Scatter – when data is received and written locally

```c
struct ibv_sge {
    uint64_t addr;  // Start address of the memory buffer (usually registered memory)
    uint32_t length; // Size (in bytes) of the memory buffer
    uint32_t lkey; // Lkey of Memory Region that is associated with this memory buffer
};
```

Warning: The value zero in ‘length’ is special – it means 2 GB
Post Send Request

- **Add a Send Request to the Send Queue**
  - No context switch will occur
  - The HW will process it according to its scheduling algorithm

- **Specify the attributes of the data transfer**
  - How data will be sent (opcode, attributes)
  - How much data will be sent
  - Which local memory buffer(s) to read/write to
    - Depends on the opcode
  - If RDMA: the remote memory buffer attributes
  - If atomic: the remote memory buffer attributes and needed operands
  - If UD QP: information on how to reach to remote side

- **Every Send Request is considered outstanding until a work Completion was generated for it or for other Send Request that followed it**
  - While a Send Request is outstanding, the resources that this Send Request use must not be destroyed/(re)used
    - The content of memory buffers that their content will be filled is undefined
    - The memory buffers that their content is sent must be available
    - For UD QPs: Address Handles must be available
Post Send Request: API

- \texttt{int ibv\_post\_send(struct ibv\_qp *qp, struct ibv\_send\_wr *wr, \newline \hspace{1cm} \textit{struct ibv\_send\_wr **bad\_wr);}}

- Add a linked list of Send Requests to the Send Queue

- Warning: \texttt{bad\_wr} is mandatory; It will be assigned with the address of the Send Request that its posting failed

\texttt{struct ibv\_send\_wr \{}\
\hspace{1cm} \text{\begin{itemize}
\item \texttt{uint64\_t wr\_id; } - Private context that will be available in the corresponding Work Completion
\item \texttt{struct ibv\_send\_wr *next; } - Address of the next Send Request. Should be NULL in the last Send Request
\item \texttt{struct ibv\_sge *sg\_list; } - Array of scatter/gather elements
\item \texttt{int num\_sge; } - Number of elements in \texttt{sg\_list}
\item \texttt{enum ibv\_wr\_opcode opcode; } - The opcode to be used
\item \texttt{int send\_flags; } - Send flags. Or of the following flags:
\begin{itemize}
\item \texttt{IBV\_SEND\_FENCE – Prevent process this Send Request until the processing of previous RDMA Read and Atomic operations were completed.}
\item \texttt{IBV\_SEND\_SIGNALED – Generate a Work Completion after processing of this Send Request ends side}
\item \texttt{IBV\_SEND\_SOLICITED – Generate Solicited event for this message in remote side}
\item \texttt{IBV\_SEND\_INLINE } - allow the low-level driver to read the gather buffers
\end{itemize}
\item \texttt{uint32\_t imm\_data; \newline \hspace{1cm}} bits, in network - Send message with immediate data (for supported opcodes); extra 32 order, that will be available in remote’s Work Completion
\end{itemize}}
union {
    struct {
        uint64_t remote_addr;        - Remote start address (the message size is
        uint32_t rkey;                      - rkey of Memory Region that is associated with
    } rdma;
    struct {
        uint64_t remote_addr;        - Remote start address (the message size is
        uint64_t compare_add;       - Value to compare/add (depends on opcode)
        uint64_t swap;                     - Value to swap if the comparison passed
        uint32_t rkey;                       - rkey of Memory Region that is associated with
    } atomic;
    struct {
        struct ibv Ah *ah;                 - Address Handle to get to remote side
        uint32_t remote_qpn;           - Remote QP number (of 0xffffffff for multicast
        uint32_t remote_qkey;          - Remote Q_Key value
    } ud;
} wr;
struct ibv_qp *qp;

struct ibv_sge sg_list = {
    .addr    = (uintptr_t)buf,
    .length  = buf_size,
    .lkey    = mr->lkey
};

struct ibv_send_wr wr = {
    .next         = NULL,
    .sg_list      = &sg_list,
    .num_sge      = 1,
    .opcode       = IBV_WR_SEND
};

struct ibv_send_wr *bad_wr;

If (ibv_post_send(qp, &wr, &bad_wr)) {
    fprintf(stderr, "Error, ibv_post_send() failed\n");
    return -1;
}
Post Receive Request

- **Add a Receive Request to the Receive Queue**
  - No context switch will occur
  - The HW will process it according to its scheduling algorithm

- **Specify where incoming message that needs Receive Request will be saved**
  - The local memory buffer(s) to write to
  - Each incoming message will consume one Receive Request
  - The S/G list must be able to hold the incoming message
  - If the message was received on a UD QP
    - Extra 40 bytes should be added to the scatter list (for the Global Routing Header (GRH))
    - The message data will start at offset 40

- **Every Receive Request is considered outstanding until a work Completion was generated to it**
  - While a Receive Request is outstanding, the resources that this Receive Request use mustn't be destroyed/(re)used
    - The content of memory buffers that their content will be filled is undefined
int ibv_post_recv(struct ibv_qp *qp, struct ibv_recv_wr *wr, struct ibv_recv_wr **bad_wr);

- Add a linked list of Receive Requests to the Receive Queue

- Warning: bad_wr is mandatory; It will be assigned with the address of the Receive Request that its posting failed

struct ibv_recv_wr {
    uint64_t wr_id;               // Private context that will be available in the corresponding Work Completion
    struct ibv_recv_wr *next;    // Address of the next Receive Request. Should be NULL in the last Receive Request
    struct ibv_sge *sg_list;     // Array of scatter elements
    int num_sge;                 // Number of elements in sg_list
};
Post Receive Request: example

```c
struct ibv_qp *qp;

struct ibv_sge sg_list = {
  .addr    = (uintptr_t)buf,
  .length  = buf_size,
  .lkey    = mr->lkey
};
struct ibv_recv_wr wr = {
  .next     = NULL,
  .sg_list  = &sg_list,
  .num_sge  = 1
};
struct ibv_recv_wr *bad_wr;

if (ibv_post_recv(qp, &wr, &bad_wr)) {
  fprintf(stderr, "Error, ibv_post_recv() failed\n");
  return -1;
}
```
Polling for Work Completion checks if the processing of a Work Request has ended

A Work Completion holds information about a completed Work Request

- Every Work Completion contains information about the corresponding completed Work Request

Every Work Completion contain several attributes

- The following fields are always valid (even if the Work Completion was ended with error)
  - `wr_id`
  - `status`
  - `qp_num`
  - `vendor_err`

- The rest of the fields depend on the QP’s transport type, opcode and status

Work Completion of Send Requests:

- Mark that a Send Request was performed and its memory buffers can be (re)used
  - For reliable transport QP: this means that the message was written in the buffers (if status is successful)
  - For unreliable transport QP: this means that the message was sent from the local port

Work Completion of Receive Requests:

- Mark that an incoming message was completed and its memory buffers can be (re)used
- Contains some attributes about the incoming message, such as size, origin, etc.
Polling for Work Completion: API

- int ibv_poll_cq(struct ibv_cq *cq, int num_entries, struct ibv_wc *wc);
  - Read one or more Work Completions from a CQ and remove them from the CQ
  - If the return value is non-negative – this is the number of polled Work Completions
  - If the return value is negative – error occurred

struct ibv_wc {
  uint64_t wr_id;     - Private context that was posted in the corresponding Work Request
  enum ibv_wc_status status;   - The status of the Work Completion
  enum ibv_wc_opcode opcode;  - The opcode of the Work Completion
  uint32_t vendor_err;      - Vendor specific error syndrome
  uint32_t byte_len;       - Number of bytes that were received
  uint32_t imm_data;      - Immediate data, in network order, if the flags indicate that such exists
  uint32_t qp_num;        - The local QP number that this Work Completion ended in
  uint32_t src_qp;        - The remote QP number
  int wc_flags;          - Work Completion flags. Or of the following flags:
                           IBV_WC_GRH – Indicator that the first 40 bytes of the receive buffer(s) contain a valid GRH
                           IBV_WC_WITH_IMM – Indicator that the received message contains immediate data
  uint16_t pkey_index;  - For UD QP: the source LID
  uint16_t slid;        - For UD QP: the source Service Level
  uint8_t sl;           - For UD QP: the destination LID path bits
  uint8_t dlid_path_bits;  - For UD QP: the destination LID path bits
};
Typical Work Completion status:

- **IBV_WC_SUCCESS** – Operation completed successfully
- **IBV_WC_LOC_LEN_ERR** – Local length error when processing SR or RR
- **IBV_WC_LOC_PROT_ERR** – Local Protection error; S/G entries doesn’t point to a valid MR
- **IBV_WC_WR_FLUSH_ERR** – Work Request flush error; it was processed when the QP was in Error state
- **IBV_WC_RETRY_EXC_ERR** – Retry exceeded; the remote QP didn’t send any ACK/NACK, even after message retransmission
- **IBV_WC_RNR_RETRY_EXC_ERR** – Receiver Not Ready; a message that requires a Receive Request was sent, but isn’t any RR in the remote QP, even after message retransmission
Polling for Work Completion: example

```c
struct ibv_cq *cq;
struct ibv_wc wc;
int num_comp;

do {
    num_comp = ibv_poll_cq(cq, 1, &wc);
} while (num_comp == 0);

if (num_comp < 0) {
    fprintf(stderr, "ibv_poll_cq() failed\n");
    return -1;
}

if (wc.status != IBV_WC_SUCCESS) {
    fprintf(stderr, "Failed status %s (%d) for wr_id %d\n", ibv_wc_status_str(wc.status),
            wc.status, (int)wc.wr_id);
    return -1;
}
```
Full data flow

Requester
- Open device
- Allocate PD
- Register MR
- Create CQ
- Create QP
- Connect QPs
- Post SR
- Poll CQ
- Check completion status

Responder
- Open device
- Allocate PD
- Register MR
- Create CQ
- Create QP
- Connect QPs + post RR
- Post Send Request
- Poll CQ
- Check completion status

CM
Exercise 5:

Write a program that will open all the needed resources and transfer data for RC QP for every Send opcode.

Optional:
1. Extend it to support RDMA Write operation.
2. Change the QP transport type to UC.
Additional Control Operations
Asynchronous events

- Asynchronous event is a mechanism to report about an event that occurred to a specific process’ resource (CQ, QP, SRQ) or for global resource (port, device)
  - Sometimes there is no other way to update about this scenario
- It is advised to create a dedicated threat that will handle the asynchronous events
Asynchronous events: API

- **int ibv_get_async_event(struct ibv_context *context, struct ibv_async_event *event);**
  - Read an asynchronous event for this context
  - Default behavior: Blocking. Can be changed to be non-blocking

- **void ibv_ack_async_event(struct ibv_async_event *event);**
  - Acknowledge an incoming asynchronous event
  - Every asynchronous event must be acknowledged. Not doing this may cause destruction of RDMA resources to be blocked forever.

```
struct ibv_async_event {
    union {
        depends on the event type
        struct ibv_cq *cq;
        struct ibv_qp *qp;
        struct ibv_srq *srq;
        int port_num;
    } element;
    enum ibv_event_type event_type;   - The asynchronous event type
};
```
Asynchronous events: API (cont.)

- **CQ events**
  - IBV_EVENT_CQ_ERR – Error occurred to the CQ

- **QP events**
  - IBV_EVENT_COMM_EST – incoming message received while the QP in RTR state
  - IBV_EVENT_SQ_DRAINED – The processing of all Send Requests was ended
  - IBV_EVENT_PATH_MIG – The alternative path of the QP was loaded (for connected QPs)
  - IBV_EVENT_QP_LAST_WQE_REACHED – Receive Request won’t be read from SRQ anymore
  - IBV_EVENT_QP_FATAL - Error occurred to the CQ
  - IBV_EVENT_QP_REQ_ERR – Transport errors detected in responder side
  - IBV_EVENT_QP_ACCESS_ERR – Violation detected in responder side
  - IBV_EVENT_PATH_MIG_ERR – Failed to load the alternative path of the QP (for connected QPs)

- **SRQ events**
  - IBV_EVENT_SRQ_LIMIT_REACHED – SRQ limit was reached
  - IBV_EVENT_SRQ_ERR – Error occurred to the SRQ
Asynchronous events: API (cont.)

- **Port events**
  - `IBV_EVENT_PORT_ACTIVE` – Port’s logical state become active
  - `IBV_EVENT_LIDCHANGE` – Port’s LID changed
  - `IBV_EVENT_PKEY_CHANGE` – Port’s P_Key table was changed
  - `IBV_EVENT_GID_CHANGE` - Port’s GID table was changed
  - `IBV_EVENT_SM_CHANGE` – New SM started to manage the subnet
  - `IBV_EVENT_CLIENT_REREGISTER` - New SM started to manage the subnet
  - `IBV_EVENT_PORT_ERR` – Port’s logical state is not active anymore

- **Device events**
  - `IBV_EVENTDEVICE_FATAL` – Something really bad happened to the device
Asynchronous events: example

```c
struct ibv_context *context;
struct ibv_async_event event;

if (ibv_get_async_event(context, &event)) {
    fprintf(stderr, "Error, ibv_get_async_event() failed\n");
    return -1;
}

...

ibv_ack_async_event(&event);
```
Exercise 6:

Write a program that open a device, and listen for asynchronous events in a loop and print them.
Completion Events

- Working with completion events help reducing the CPU usage
- Once the application requests to get a notification on a specific CQ, it can be block until the next Work Completion be enqueued to this CQ

- The following pseudo-code example demonstrates one possible way to work with completion events. It performs the following steps:

  Stage I: Preparation
  1. Creates a CQ
  2. Request for notification upon a new (first) completion event

  Stage II: Completion Handling Routine
  3. Wait for the completion event and ack it
  4. Request for notification upon the next completion event
  5. Empty the CQ

- Note that an extra event may be triggered without having a corresponding completion entry in the CQ. This occurs if a completion entry is added to the CQ between Step 4 and Step 5, and the CQ is then emptied (polled) in Step 5.
Completion Events: API

- **int ibv_req_notify_cq(struct ibv_cq *cq, int solicited_only);**
  - Request for a notification about the next Work Completion to be added to the Completion Queue event for this context
  - Request for any Work Completion or only for Work Completion of completed Receive Requests that their requester send them with the solicited event indicator on

- **int ibv_get_cq_event(struct ibv_comp_channel *channel,**
  struct ibv_cq **cq, void **cq_context);**
  - Read a Completion event
  - Default behavior: Blocking. Can be changed to be non-blocking

- **void ibv_ack_cq_events(struct ibv_cq *cq, unsigned int nevents);**
  - Acknowledge Completion event(s)
  - Multiple completion events can be acknowledge in one time
Completion Events: example

```c
struct ibv_cq *cq, *en_cq;
ibv_comp_channel *channel;
void *ev_ctx;

if (ibv_req_notify_cq(cq, 0)) {
    fprintf(stderr, "Error, ibv_req_notify_cq() failed\n");
    return -1;
}

... if (ibv_get_cq_event(channel, &ev_cq, &ev_ctx)) {
    fprintf(stderr, "Error, ibv_get_cq_event() failed\n");
    return -1;
}

ibv_ack_cq_events(ev_cq, 1);

if (ibv_req_notify_cq(ev_cq, 0)) {
    fprintf(stderr, "Error, ibv_req_notify_cq() failed\n");
    return -1;
}

TODO - Need to empty the CQ here ...
```
Shared Receive Queue (SRQ)

- Shared Receive Queue is an object that provides better scalability
  - One Receive Queue to multiple QPs
    - When there is a need for a Receive Request, it is being fetched, in atomic way, from the SRQ
  - If there are N QPs, each of them may get M incoming messages at any random time
    - Without SRQ: there is a need to post N * M RRIs
    - With SRQ: we can post K * M RRIs (where K << N)

- SRQ provides mechanism for the application to be notified when a number of RRIs is dropped below a limit
  - Using the SRQ limit event

- RRs will be posted to the SRQ
  - And not to the QP
Shared Receive Queue (SRQ): API

- **struct ibv_srq** *ibv_create_srq(struct ibv_pd *pd, struct ibv_srq_init_attr *srq_init_attr);
  
  - Create a new Shared Receive Queue

- **int ibv_destroy_srq(struct ibv_srq *srq);**
  
  - Destroy a Shared Receive Queue
  
  - This verb should be called after destroying all the QPs that are associated with it

- **int ibv_modify_srq(struct ibv_srq *srq, struct ibv_srq_attr *srq_attr, enum ibv_srq_attr_mask srq_attr_mask);**
  
  - Resize or modify the Shared Receive Queue attributes

- **int ibv_query_srq(struct ibv_srq *srq, struct ibv_srq_attr *srq_attr);**
  
  - Query the attributes of a Shared Receive Queue
  
  - The limit value may change

- **int ibv_post_srq_recv(struct ibv_srq *srq, struct ibv_recv_wr *recv_wr, struct ibv_recv_wr **bad_recv_wr);**
  
  - Add a linked list of Receive Requests to a Shared Receive Queue
struct ibv_srq_attr {
    uint32_t max_wr;  // The number of Receive Requests that can be outstanding in the SRQ
    uint32_t max_sge; // The number of scatter entries that each Receive Request may hold
    uint32_t srq_limit; // The SRQ watermark value (only relevant in modify_srq)
};

struct ibv_srq_init_attr {
    void *srq_context; // A private context that the SRQ will be associated with
    struct ibv_srq_attr attr; // The SRQ attributes to be created
};
struct ibv_pd *pd;
struct ibv_srq *srq;
struct ibv_srq_init_attr attr = {
    .attr = {
        .max_wr = rx_depth,
        .max_sge = 1
    }
};

srq = ibv_create_srq(pd, &attr);
if (!srq) {
    fprintf(stderr, "Error, ibv_create_srq() failed\n");
    return -1;
}

...

if (ibv_destroy_srq(srq)) {
    fprintf(stderr, "Error, ibv_destroy_srq() failed\n");
    return -1;
}
struct ibv_srq *srq;
struct ibv_srq_attr srq_attr;

memset(&srq_attr, 0, sizeof(srq_attr));

srq_attr.srq_limit = 10;

if (ibv_modify_srq(srq, &srq_attr, IBV_SRQ_LIMIT)) {
    fprintf(stderr, "Error, ibv_modify_srq() failed when arming an SRQ\n");
    return -1;
}
Exercise 7:

Use the program from exercise 5 and add an SRQ support.

Tips:
- Add the SRQ handle when creating the QP
- Post the RR to the SRQ and not to the QP
Address Handle (AH)

- Every UD QP can send message to any other UD QP
- Address Handle describes the path from local to remote ports
  - Same AH can be used by multiple QPs
- The Address Handle will be used when posting a Send Request to an UD QP
Address Handle (AH): API

- `struct ibv_ah *ibv_create_ah(struct ibv_pd *pd, struct ibv_ah_attr *attr);`
  - Create a new Address Handle

- `int ibv_init_ah_from_wc(struct ibv_context *context, uint8_t port_num, struct ibv_wc *wc, struct ibv_grh *grh, struct ibv_ah_attr *ah_attr);`
  - Initialize an `ibv_ah_attr` structure according to a Work Completion and a GRH buffer

- `struct ibv_ah *ibv_create_ah_from_wc(struct ibv_pd *pd, struct ibv_wc *wc, struct ibv_grh *grh, uint8_t port_num);`
  - Create an Address Handle according to a Work Completion and a GRH buffer

- `int ibv_destroy_ah(struct ibv_ah *ah);`
  - Destroy an Address Handle

  - This verb should be called if there isn’t any outstanding Send Request that points to it
struct ibv_global_route {
  union ibv_gid dgid;               - Destination GID address
  uint32_t flow_label;              - Flow label which is a hint for switches and routers which path to take
  uint8_t sgid_index;               - The index in the port’s GID table of the source GID
  uint8_t hop_limit;                - The number of hops to take before dropping the message
  uint8_t traffic_class;            - Traffic class of the message (priority)
};

struct ibv_ah_attr {
  struct ibv_global_route grh;      - Description of the Global Routing Header
  uint16_t dlid;                     - The Destination LID (can be unicast or multicast)
  uint8_t sl;                        - The Service Level value of the message
  uint8_t src_path_bits;            - The source path bits used when the port has a range of LIDs
  uint8_t static_rate;              - The static rate between local and remote port speeds
  uint8_t is_global;                - Indication that the message will be sent with GRH
  uint8_t port_num;                 - The local port number to send the message from
};
Address Handle (AH): example

```c
struct ibv_pd *pd;
struct ibv_ah *ah;
struct ibv_ah_attr ah_attr = {
    .is_global = 0,
    .dlid = dlid,
    .sl = sl,
    .src_path_bits = 0,
    .port_num = port
};

ah = ibv_create_ah(pd, &ah_attr);
if (!ah) {
    fprintf(stderr, "Error, ibv_create_ah() failed\n");
    return -1;
}
...

if (ibv_destory_ah(ah)) {
    fprintf(stderr, "Error, ibv_destory_ah() failed\n");
    return -1;
}
```
Exercise 8:

Use the program from exercise 5 and add an UD support.

Tips:

- AH should be used when posting in the SR
- Remote side attributes should be added to the SR as well
- The data in the receive buffer starts at address 40
Tips and Tricks
Performance tips

- **General tips**
  - Avoid using control operations in data path
    - They will perform context switch
    - They may allocate/free dynamic resources
  - Set affinity for process/task
  - Work with local NUMA node
  - Use MTU which provide best performance
  - Register physical contiguous memory
  - UD is more scalable than RC

- **Posting**
  - Post multiple Work Request in one call
  - Avoid using many scatter/gather elements
  - Atomic operations are performance killers
  - Work with big messages
  - Use selective signaling to reduce number of Work Completions
  - Inline data will provide better latency

- **Polling**
  - Read multiple Work Completion in one call
  - Use polling to get low latency and Completion events to get lower CPU usage
  - When working with events: acknowledge multiple events at once
When packets are “lost” – check the counters

- `/sys/class/infiniband/<device name>/diag_counters`

- `rq_num_lle` - Responder - number of local length errors
- `rq_num_lqpoe` - Responder - number local QP operation error
- `rq_num_leeeoe` - Responder - number local EE operation error
- `rq_num_lpe` - Responder - number of local protection errors
- `rq_num_wrfe` - Responder - number of WR flushed errors
- `rq_num_lae` - Responder - number of local access errors
- `rq_num_rire` - Responder - number of remote invalid request errors
- `rq_num_rae` - Responder - number of remote access errors
- `rq_num_roe` - Responder - number of remote operation errors
- `rq_num_rnr` - Responder - number of RNR Naks sends
- `rq_num_oos` - Responder - number of out of sequence requests received
- `rq_num_dup` - Responder - number of duplicate requests received
- `rq_num_rirdre` - Responder - number of remote invalid RD request errors
- `rq_num_mce` - Responder - number of bad multicast packets received
- `rq_num_rsync` - Responder - number of RESYNC operations
- `num_cqovf` - Number of CQ overflows
- `num_eqovf` - Number of EQ overflows
When packets are “lost” – check the counters (cont.)

- `sq_num_lle` - Requester - number of local length errors
- `sq_num_lqpoe` - Requester - number local QP operation error
- `sq_num_leeoe` - Requester - number local EE operation error
- `sq_num_lpe` - Requester - number of local protection errors
- `sq_num_wrfe` - Requester - number of WR flushed errors
- `sq_num_mwbe` - Requester - number of memory window bind errors
- `sq_num_bre` - Requester - number of bad response errors
- `sq_num_rire` - Requester - number of remote invalid request errors
- `sq_num_rae` - Requester - number of remote access errors
- `sq_num_roe` - Requester - number of remote operation errors
- `sq_num_rnr` - Responder - number of RNR Naks received
- `sq_num_oos` - Requester - number of out of sequence Naks received
- `sq_num_to` - Requester - number of time out received
- `sq_num_tree` - Requester - number of transport retries exceeded errors
- `sq_num_rree` - Requester - number of RNR Nak retries exceeded errors
- `sq_num_ldrve` - Requester - number of local RDD violation errors
- `sq_num_rabrte` - Requester - number of remote aborted errors
- `sq_num_ieecne` - Requester - number of invalid EE context number errors
- `sq_num_ieecse` - Requester - invalid EE context state errors
- `sq_num_rsync` - Requester - number of RESYNC operations
More Information is Available

- InfiniBand specifications
- RDMAmojo (my blog) 😊
- The document “RDMA Aware Networks Programming User Manual”
- The man pages
- Code samples that comes with libibverbs and librdmacm
Advanced Features and Enhancements
eXtended Reliable Connection (XRC)

- Each two nodes can use the same XRC QP
  - No matter how many cores they have
- Provide reliable transport type
  - Like RC
- Increase the scalability
  - Reduce the number of QPs between each two nodes
  - Reduce the memory usage
Dynamically Connected Transport (DCT)

- Allow dynamically (re)connect QPs when needed
- Provide reliable transport type
  - Like RC
- Provide very high scalability
  - Reduce the number of opened QPs in each node
    - One QP per core
  - Reduce the memory usage
GPUDirect RDMA

- Allow registration of GPU memory
- Data can be send and received using RDMA devices directly to the GPU memory
  - Avoiding unnecessary memory copy
- Working with NVIDIA’s CUDA toolkit
CORE-direct (Collective Offload Resource Engine)

- Provide offload for collective operations
  - Perform collective operations in wire speed
  - Floating point operation
  - Support blocking and non-blocking operations
- Increase the scalability
- Decrease the CPU usage
Thank You