# RDMA Programming - Mellanox

Saturday, December 23, 2017 12:11 PM

Infiniband - IBTA

Read Chapter - IBTA Specs ( 1 & 4)
MPI Programming
Socket Programming

Infiniband Advantage

1. Easy Scaling
2. Low latency
3. High Throughput

Using RDMA has the following major advantages:

- Zero-copy - applications can perform data transfer without the network software stack involvement and data is being send received directly to the buffers without being copied between the network layers.
- Kernel bypass - applications can perform data transfer directly from userspace without the need to perform context switches.
- No CPU involvement - applications can access remote memory without consuming any CPU in the remote machine. The remote memory machine will be read without any intervention of remote process (or processor). The caches in the remote CPU(s) won't be filled with the accessed memory content.
- Message based transactions - the data is handled as discrete messages and not as a stream, which eliminates the need of the application to separate the stream into different messages/transactions.
- Scatter/gather entries support - RDMA supports natively working with multiple scatter/gather entries i.e. reading multiple memory buffers and sending them as one stream or getting one stream and writing it to multiple memory buffers

From <http://www.rdmamojo.com/2014/03/31/remote-direct-memory-access-rdma/>

A single lane is a serial link operating at one of five data rates:

- Single Data Rate (SDR) - **2.5Gb/s**
- Double Data Rate (DDR) - **5Gb/s**
- Quad Data Rate (QDR) - **10Gb/s**
- Fourteen Data Rate (FDR) - **14Gb/s**
- Enhanced Data Rate (EDR) - **25Gb/s**

IB Widths (combined lanes): **1X, 4X**

RDMA is completely message-oriented, so all application messages are sent and received as units, unlike TCP/IP, which treats network communication as a stream of bytes.

RDMA Technolgoies

1. IB
2. iWarp
3. ROCE

**Quality of Service (QoS)** is the ability to provide different priority to different applications, users, or data flows, or to guarantee a certain level of performance to a data flow.

QoS can be achieved by:

- Defining I/O channels at the adapter level
- Defining Virtual Lanes at the link level

It allows the control of the congestion on the network.

IB enables a scaling of up to

# 48,000

nodes in a single subnet

## CPU Offloads

Mellanox ACADEMY

The IB architecture supports packets transportation in the fabric with minimum CPU intervention.

That is achieved thanks to:

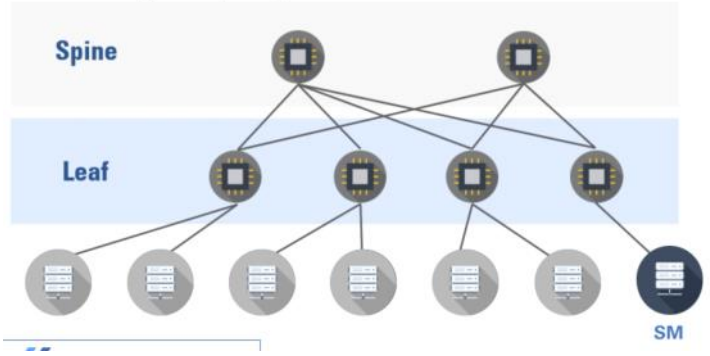
- Hardware based transport protocol
- Kernel Bypass
- Reliable transport - no need for additional CPU cycles for lost packets
- RDMA support - the ability to share and utilize data that is loaded in the memory directly between nodes in the fabric

Subnet Manager

The Subnet Manger makes the fabric management very simple:

- Plug & Play end nodes environment
- Centralized route manager

As part of the IB spec it offers a variety of in-band diagnostics & management tools that can be further enhanced and upgraded by IB equipement manifactures like Mellanox.




EDR = 25* 4 = 100 Gbps

## Transport Layer - Responsibilities

Mellanox ACADEMY

The transport portion of the packet delivers the packet to the proper QP (Queue Pair) and instructs the QP how to process the packet's data.

The transport layer is responsible for segmentation/reassembly:

- **Segmenting** an operation into multiple packets when the message's data payload is greater than the maximum transfer unit (MTU) of the path
- The QP on the receiving end **reassembles** the data into the specified data buffer in its memory.

Responsible for configuring each QP with a certain class of **service type**:

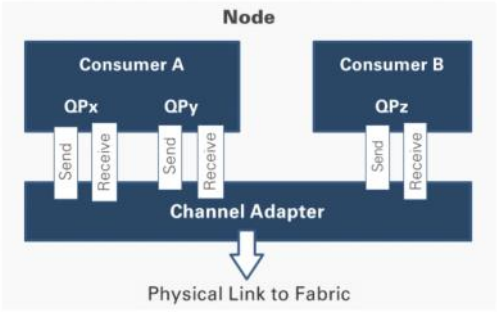
- Connection-oriented vs. Datagram
- Acknowledged vs. Unacknowledged

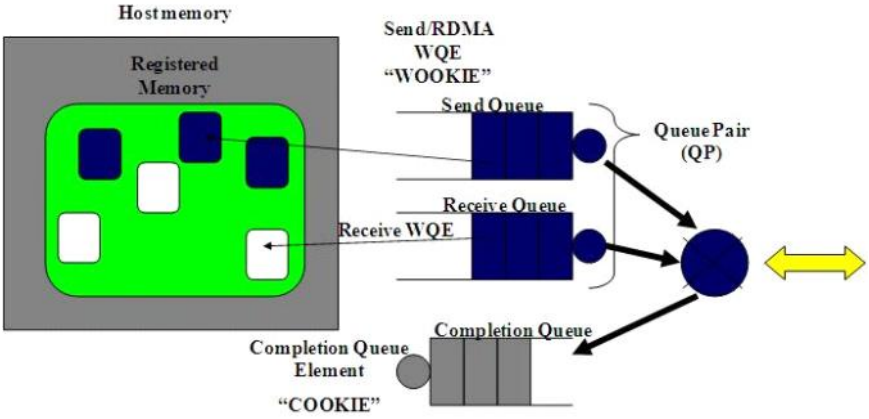

## Queue Pairs (QP)

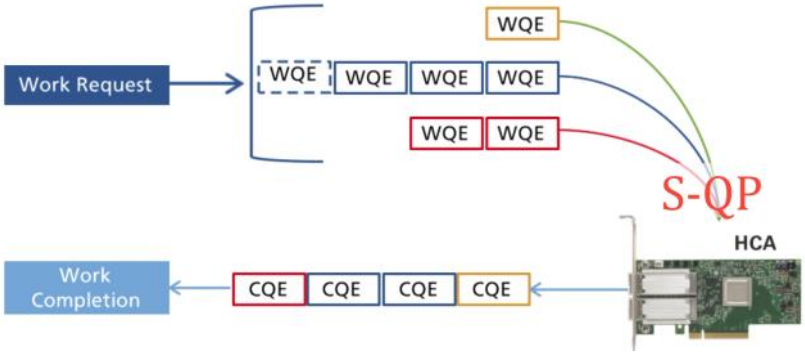
Mellanox ACADEMY

A QP consists of a **Send Queue** and a **Receive Queue**

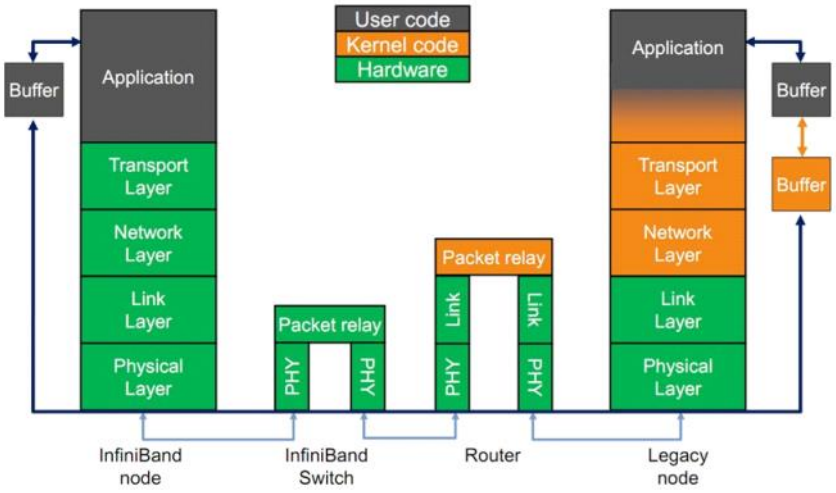
- Each QP represents one end of a channel
- Send and receive queues are created as a pair
- A Queue Pair is identified by its **QP Number**
- In order to avoid OS involvement, applications have direct access to these QPs
- This is accomplished by mapping each application's virtual address space directly into the QPs




Zero Copy
-No copy of data from APP to kernel
-use DMA and MMU (memory mapping)
-Sendfile/sendfile64 API




- IB HCA - Performance Everything in Hardware , So if SM is present it get the SUBNET ID without OS.IB HCA Understand Verbs and executes it.
- The caches in the remote CPU(s) won't be filled with the accessed memory content.

B mainly consists of 5 layers.

| No | Layers | Functions |
|---|---|---|
| 1st | Software Transport Verbs and Upper Layer Protocols | Interface between applications and hardwares<br>Define methodology for management functions |
| 2nd | Transport | Delivers packets to the appropriate QP node<br>Message Assembly/De-assembly<br>Access Right |
| 3rd | Network | Route packets between different partitions/subnets |
| 4th | Data Link(Symbols and framing) | Route packets on the same partition/subnet |
| 5th | Physical | Signal levels/Media/Connections |

From <http://jcf94.com/2016/06/27/2016-06-27-rdma/>

**Memory Binding**

Memory Binding instructs the hardware to alter memory registration relationships.
It associates (binds) a memory Window to a specified range within an existing memory Region.

**Remote Direct Memory Access (RDMA)**

RDMA supports zero-copy networking by enabling the network adapter to transfer data directly to or from application memory, eliminating the need to copy data between application memory and the data buffers in the operating system.
For an RDMA operation, the WQE specifies a block of data in the consumer's memory space for the hardware to send to the destination, letting a receive WQE already queued at the destination specify where to place that data.
In addition, it also specifies the address in the remote consumer's memory.
Thus an RDMA operation does not need to involve the receive work queue of the destination.
There are 3 types of RDMA operations:, RDMA-WRITE, RDMA-READ, and ATOMIC.

**Receive Queue Operation**

A RECEIVE WQE specifies where the hardware should place the data received from another consumer when that consumer executes a SEND operation.
Each time the remote consumer successfully executes a SEND operation, the hardware takes the next entry from the receive queue, places the received data in the memory location specified in that receive WQE, and places a CQE on the completion queue, indicating to the consumer that the receive operation has completed. Thus the execution of a SEND operation causes a receive queue operation at the remote consumer.

RDMA Opcodes

1. RDMA Send
2. RDMA Recv
3. RDMA Write
4. RDMA Read

**RDMA Send/Receive ( 2 Sided Transfer)**

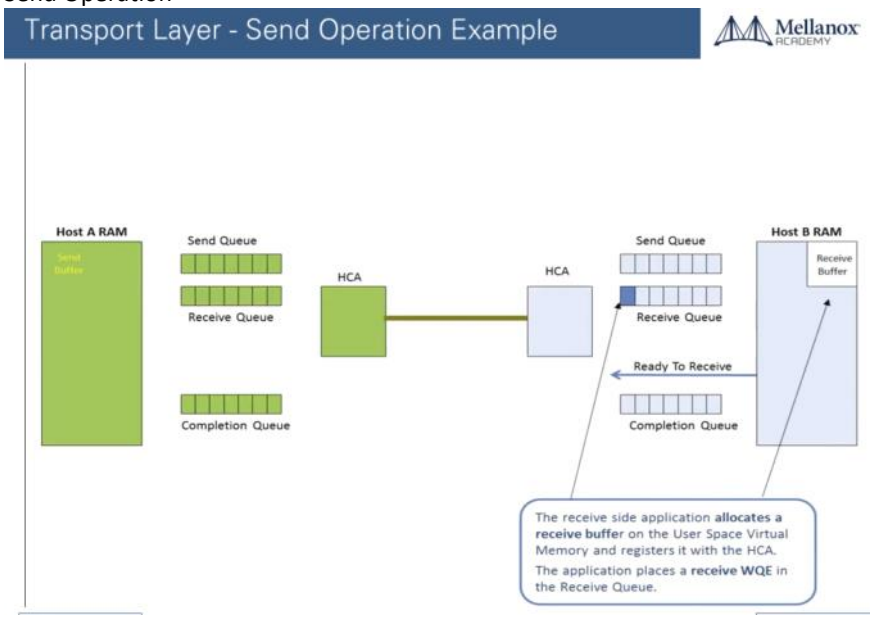
-A receiver first posts a RECV work request that describes a virtual memory area into which the adapter should place a single message.

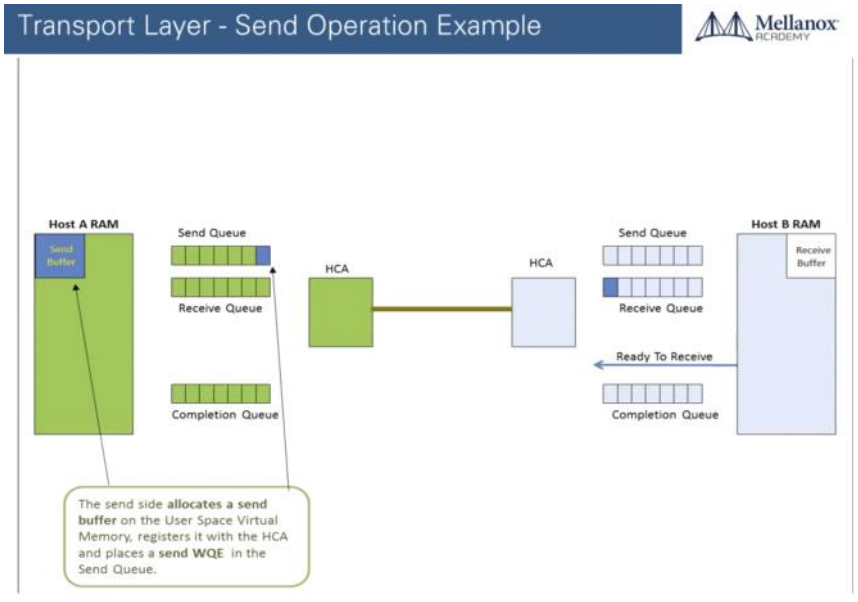
-The sender then posts a SEND work request describing a virtual memory area containing the message to be sent.

-The network adapters transfer data directly from the sender's virtual memory area to the receiver's virtual memory area without any intermediate copies.

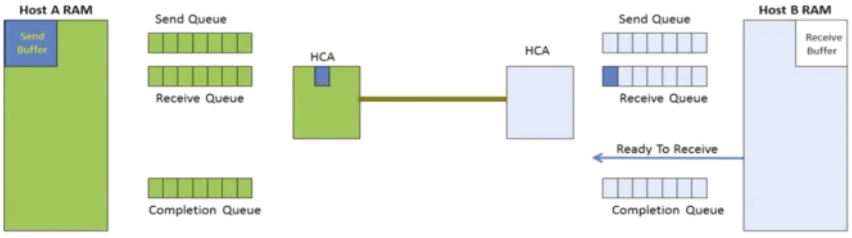
- Since both sides of the transfer are required to post work requests, this is called a "two-sided"

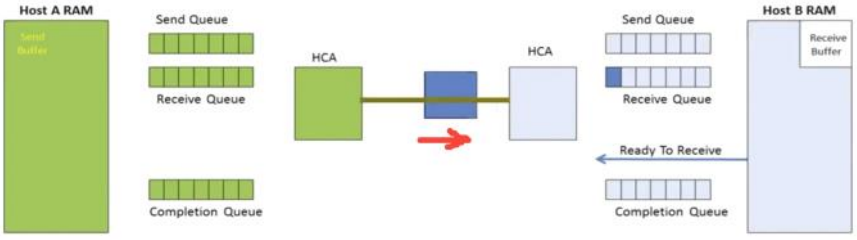
Send Operation



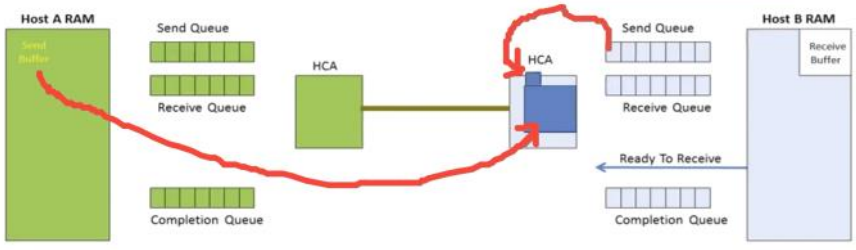



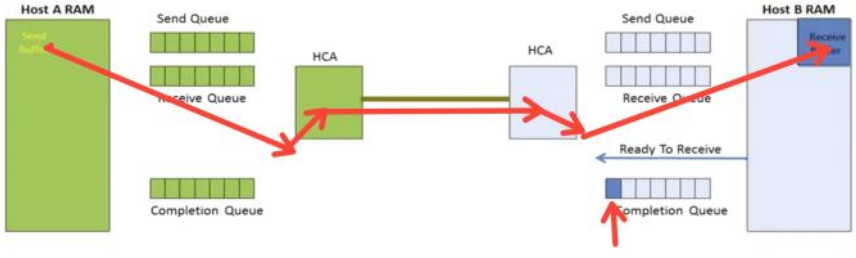
The **HCA executes the send request**, reads the buffer of the Host Ram and **sends to the remote side (HCA).**



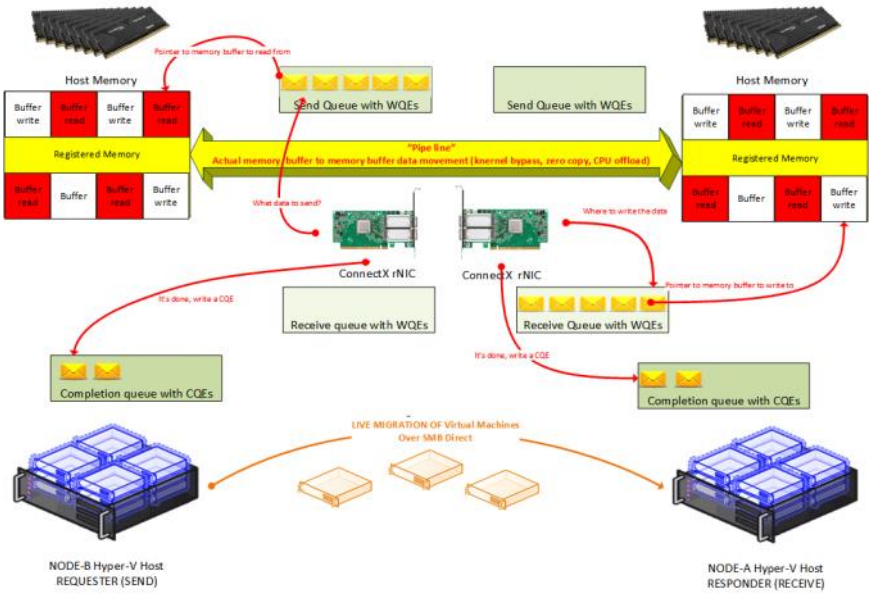




When the packet arrives at the HCA, it **executes the receive WQE Commands, places the buffer CONTENT in the appropriate location** and **generates a Completion Queue.**



In RDMA, actions are specified by verbs which convey requests to the network adapter.



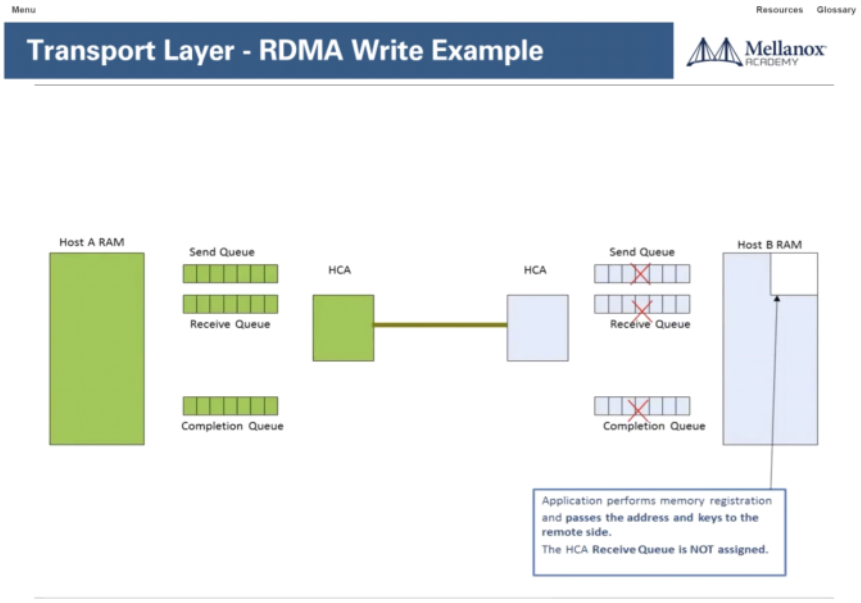

To initiate a transfer, **ibv_post_send** places a work request data structure describing the transfer onto a network adapter queue.

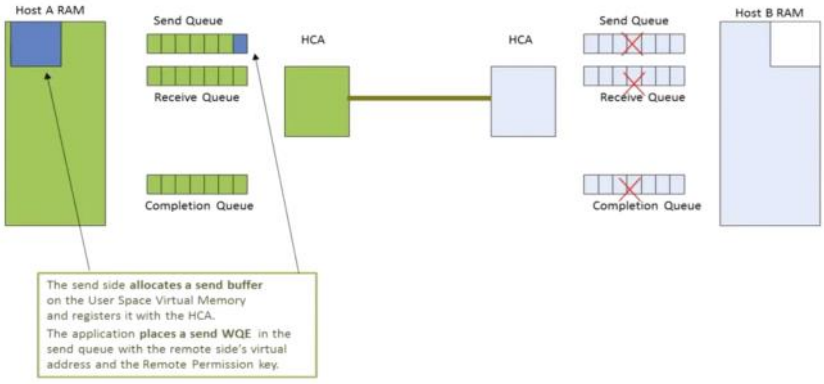
Data transfers are all asynchronous: once a work request has been posted, control returns to the user-space application which must later use the **ibv_poll_cq** function to remove a work completion data structure from a network adapter's completion queue.

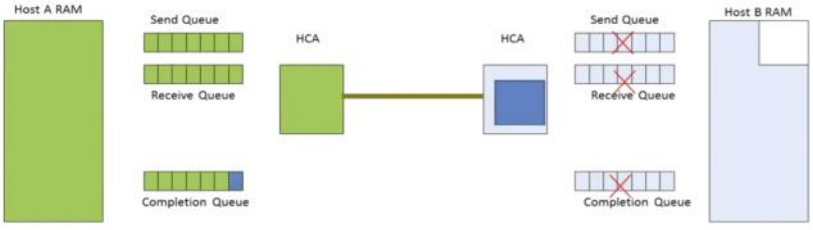
This completion contains the status for the finished transfer and tells the application it can again safely access the virtual memory used in the transfer

**RDMA Write ( 1 -Side Transfer)**
A sender posts a RDMA WRITE request that "pushes" a message directly into a virtual memory area that the receiving side previously described to the sender.

The receiving side's CPU is completely "passive" during the transfer, which is why this is called "one-sided."




**RDMA Read ( 1- Side Transfer)**
Receiver posts a RDMA READ request that "pulls" a message directly from the sending side's virtual memory,
and the sending side's CPU is completely passive.

**RDMA Write with IMM**
in RDMA write operaiton , since Passive do not when does data has been Completed ,

Sender instead of RDMA write posts a RDMA WRITE WITH IMM , which is same as RDMA as write but but the send work request also includes 4 bytes of immediate (out-of-band) data that is delivered to the receiver on completion of the transfer.

The receiving side posts a RECV work request to catch these 4 bytes, and the work completion for the RECV indicates the status and amount of data transferred in the message.

- Inline Data: The API provides an optional “inline” feature that allows an interface adapter to copy the data from small messages into its own memory as part of a posted work request. This immediately frees the buffer for application reuse, and makes the transfer more efficient since the adapter has the data ready to send and does not need to retrieve it over the memory bus during the transfer

**Processing Completion Queue**

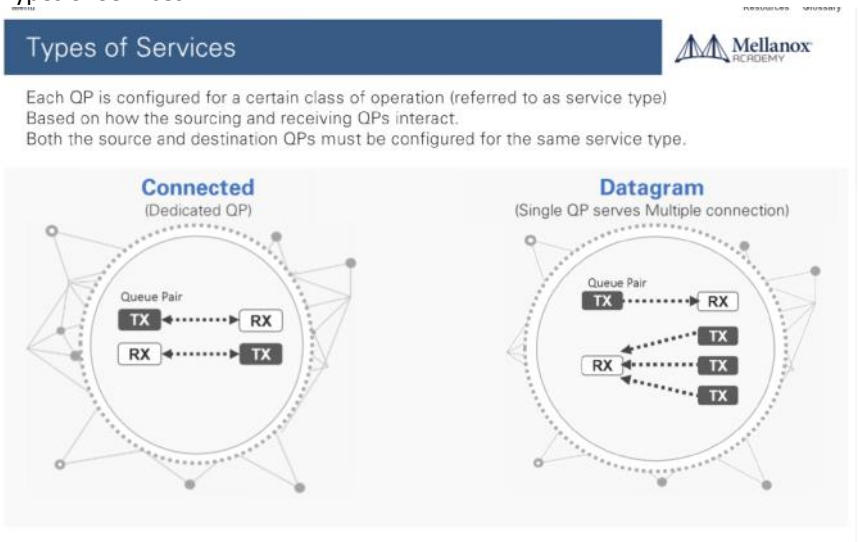
1. Busy Polling

The first completion detection strategy, called “busy polling”, is to repeatedly poll the completion queue until a
completion becomes available. It allows immediate reactionto completions at the cost of very high CPU utilization, but
requires no operating system intervention.

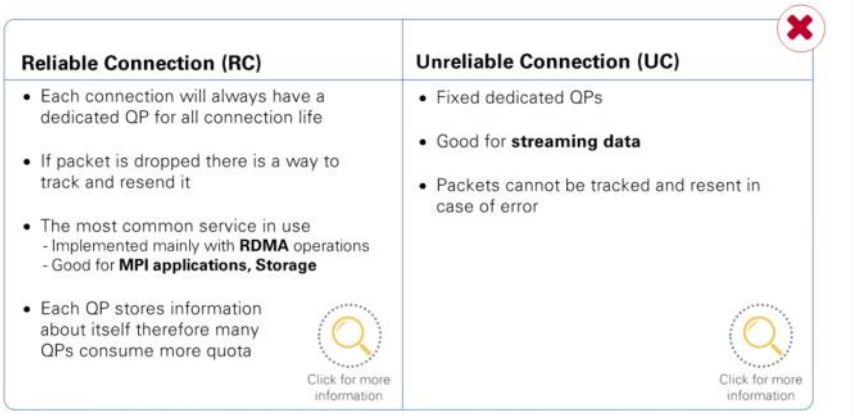
2. Event Notification

The second strategy, called “event notification”, is to set up a completion channel that allows an application to
wait until the interface adapter signals a notification on this channel, at which time the application obtains the work
completion by polling. It requires the application to wait for the notification by transferring to the operating system,
but reduces CPU utilization significantly

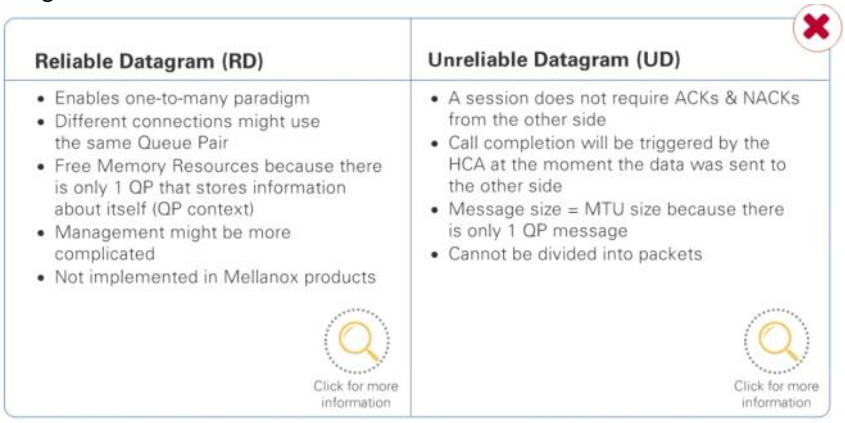
Types of Services



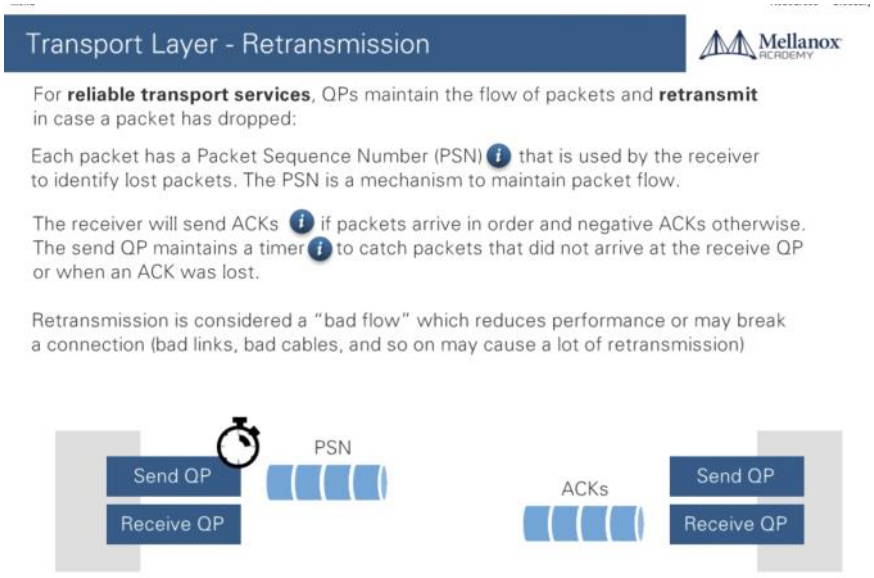

Connection Oriented

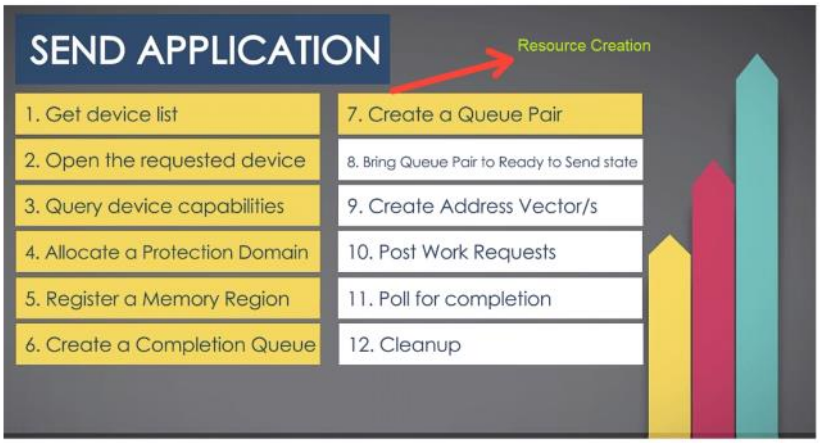


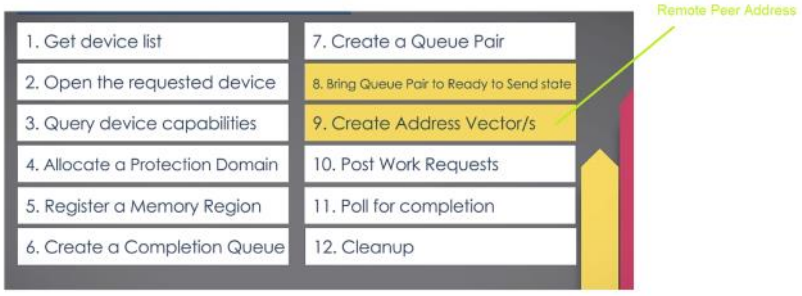

Datagram Oriented




- Most common is Reliable connection & unreliable for real time traffic.
- ACK are only send in reliable type of service ( RC and RD)







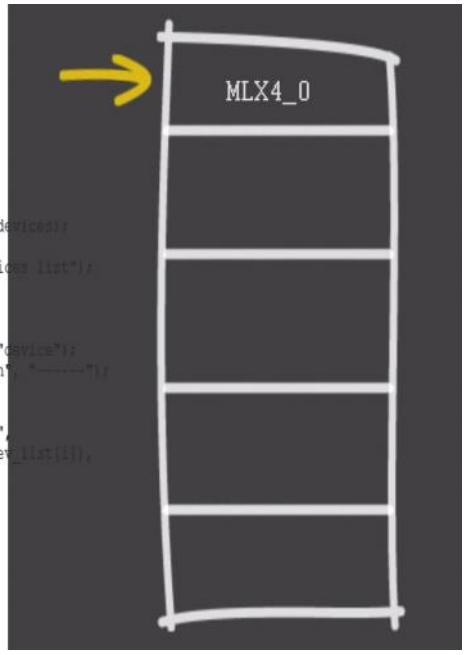
FAQs

## Why do we always see Responder CQE Errors with RoCE RDMA?

*RDMA receivers need to post receive WQEs to handle incoming messages if the application does not know how many messages are expected to be received (e.g. by maintaining high-level message credits) they may post more receive WQEs than will actually be used. On application tear-down, if the application did not use up all of its receive WQEs the device will issue completion with the error for these WQEs to indicate HW does not plan to use them, this is done with a clear syndrome indication of "Flushed with error".*

1 .Get IB devices

```c
#include <stdio.h>
#include <infiniband/verbs.h>
#include <infiniband/arch.h>

int main(int argc, char *argv[])
{
        struct ibv_device **dev_list;
        int num_devices, i;

        dev_list = ibv_get_device_list(&num_devices);
        if (!dev_list) {
                perror("Failed to get IB devices list");
                return 1;
        }

        printf("    %-16s\t                "device");
        printf("    %-16s\t----------------\n", "------");

        for (i = 0; i < num_devices; ++i) {
                printf("    %-16s\t%016llx\n",
                       ibv_get_device_name(dev_list[i]),

        }

        ibv_free_device_list(dev_list);

        return 0;
}
```




Returns Array of dev_list ( which is structure to ibv_device)
Ibv_get_device_name - > string name
Ibv_free_device_list -> to free the dev_list.

```
[root@l-fw01 solution]# ./ ibv_devices
    device
    ------
    mlx4_0
```

```
l-fw01        > ibstat
CA 'mlx4_0'
        CA type: MT4103
        Number of ports: 2
        Firmware version: 2.30.8006
        Hardware version: 0
        Node GUID: 0x0002c90300e95840
        System image GUID: 0x0002c90300e95843
        Port 1:
                State: Active
                Physical state: LinkUp
                Rate: 10
                Base lid: 0
                LMC: 0
                SM lid: 0
                Capability mask: 0x00010000
                Port GUID: 0x0202c9fffee95841
                Link layer: Ethernet
        Port 2:
                State: Active
                Physical state: LinkUp
                Rate: 40 (FDR10)
                Base lid: 3
                LMC: 0
                SM lid: 3
                Capability mask: 0x0251486a
                Port GUID: 0x0002c90300e95842
                Link layer: InfiniBand
```

2 .Open Device and Get the Attribute

```
static int print_hca_cap(struct ibv_device *ib_dev, uint8_t ib_port)
{
        struct ibv_context *ctx;
        struct ibv_device_attr device_attr;
        struct ibv_port_attr port_attr;
        int rc = 0;
        uint8_t port;
        char buf[256];
                                        1. OPEN DEVICE
        ctx = ibv_open_device(ib_dev);
        if (!ctx) {
                fprintf(stderr, "Failed to open device\n");
                rc = 1;
                goto cleanup;
                                        2.Device Context
                                                        3.Device Attribute
        if (ibv_query_device(ctx, &device_attr)) {
                fprintf(stderr, "Failed to query device props");
                rc = 2;
                goto cleanup;
        }
```

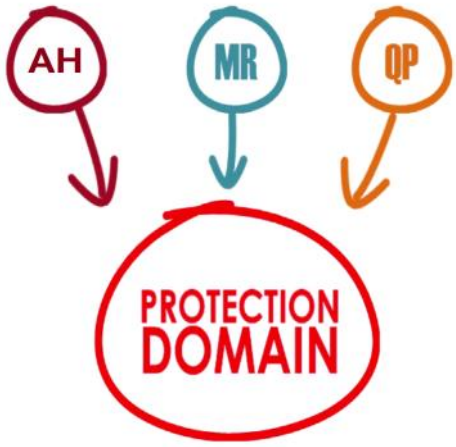
Ibv_open_device - >opens device and return a context
Use this context in ibv_query_device to get the device attribute

```
if (verbose) {
        printf("\tmax_qp:\t\t\t\t%d\n", device_attr.max_qp);          MaX QP device Supports
        printf("\tmax_qp_wr:\t\t\t%d\n", device_attr.max_qp_wr);      Max No of Work request
        printf("\tmax_sge:\t\t\t%d\n", device_attr.max_sge);          Max no of scatter gatther element
        printf("\tmax_cq:\t\t\t\t%d\n", device_attr.max_cq);          Max no of CQ
        printf("\tmax_cqe:\t\t\t%d\n", device_attr.max_cqe);         Max size of each completion queue
        printf("\tmax_mr:\t\t\t\t%d\n", device_attr.max_mr);         Max no of MR
        printf("\tmax_mr_size:\t\t\t0x%llx\n",
                        (unsigned long long) device_attr.max_mr_size);   Largest Memory size a device can create
        printf("\tmax_pd:\t\t\t\t%d\n", device_attr.max_pd);
}
```

Output

```
hca_id: mlx4_0
        transport:                      InfiniBand (0)
        fw_ver:                         2.30.8006
        node_guid:                      0002:c903:00e9:5840
        sys_image_guid:                 0002:c903:00e9:5843
        vendor_id:                      0x02c9
        vendor_part_id:                 4103
        hw_ver:                         0x0
        board_id:                       MT_1090111019
        phys_port_cnt:                  2
        max_qp:                         393080
        max_qp_wr:                      16351
        max_sge:                        32
        max_cq:                         65408
        max_cqe:                        4194303
        max_mr:                         524032
        max_mr_size:                    0xffffffffffffffff
        max_pd:                         32764
```

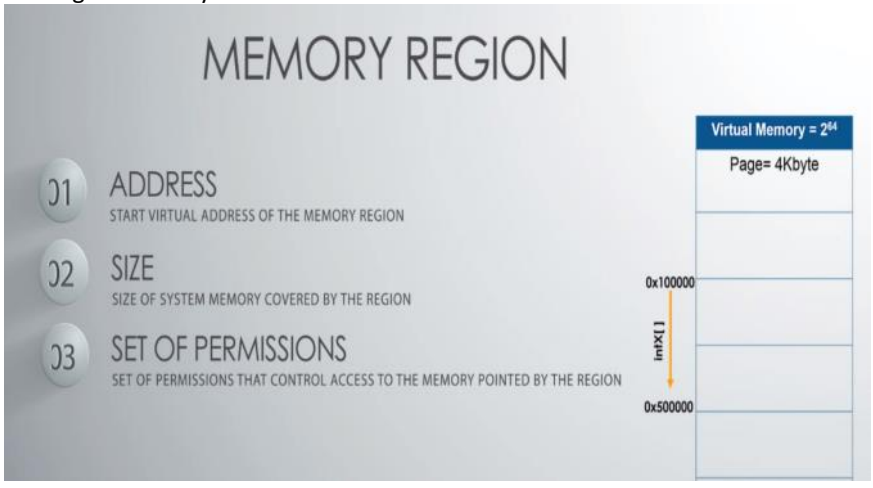
3. Allocation of PD
   Protection Domain - Grouping of resources, Resources of same protection domain
   Can access each other.




AH- Address Vector or Address Handler ( remote Information)
MR - Local Memory Information
QP - SQ + RQ

struct ibv_context *context = ibv_open_device(device);--->returns context
struct ibv_pd *pd = ibv_alloc_pd(context);--> for device context return pd pointer

4. Allocation of MR
   Pinning of Memory for RMDA transfer





Permissions by default is Local Read.

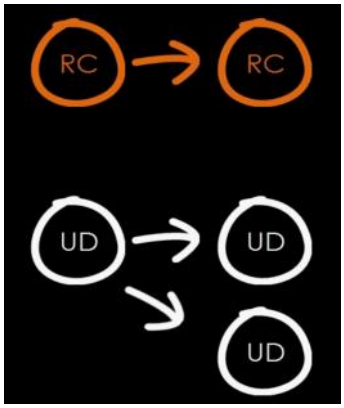
5. Setting up Completion Queue ( CQ)

Used by Adapter to Notify application of status of transfer or completed work request, so that MR can be freed.
Each entry in CQ is called CQE ( CQ element)
Status of one or more completed WR.
There can be single CQ for All the QP.

```
#define CQ_SIZE 0x100

        struct ibv_cq *cq = ibv_create_cq(context, CQ_SIZE, NULL,
                                          0, 0);
        if (cq) {
                fprintf(stderr, "Couldn't create CQ\n");
                goto free_mr;
        }
```

CQ_SIZE is minimum size request, It can be more than that .
#define MAX_NUM_SENDS 0x10
#define MAX_GATHER_ENTRIES 2
#define MAX_SCATTER_ENTRIES 2

```
struct ibv_qp_init_attr attr = {
        .send_cq = cq,
        .recv_cq = cq,
        .cap     = {
                .max_send_wr  = MAX_NUM_SENDS,
                .max_recv_wr  = 0,
                .max_send_sge = MAX_GATHER_ENTRIES,
                .max_recv_sge = MAX_SCATTER_ENTRIES,
        },
        .qp_type = IBV_QPT_UD,---> Unreliable Datagram
};
```




6. Creating a QP
QP = SQ + RQ
Application post WR to SQ contains memory Information to be Transferred, if it wants to Send Data
Application post WR to RQ containing Memory information where Data has to Be written , once Data is received from Sender.

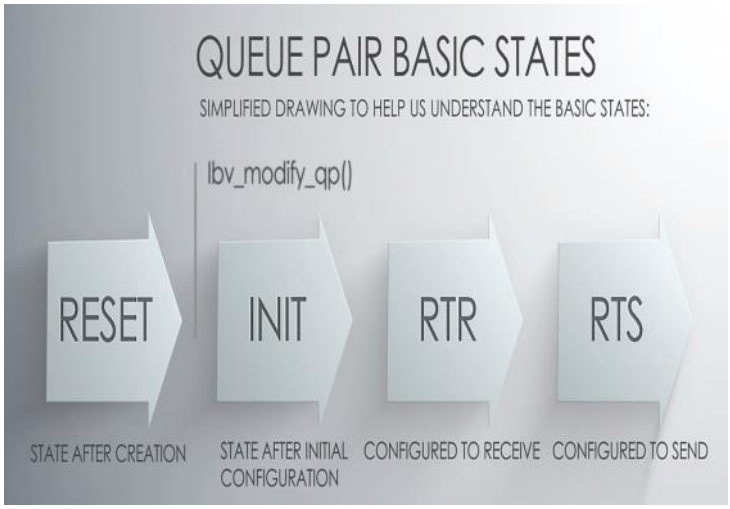
A ) Queue Pair Attributes & QP Creation

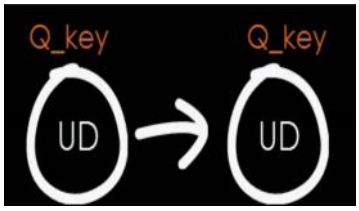
define MAX_NUM_SENDS 0x10
#define MAX_GATHER_ENTRIES 2
#define MAX_SCATTER_ENTRIES 2

```
        struct ibv_qp_init_attr qp_attr = {
                .send_cq = cq,                                          ----------> defines CQ for receive and send QP/
                .recv_cq = cq,
                .cap     = {
                        .max_send_wr  = MAX_NUM_SENDS,    ---------->Max no of WR for TX
                        .max_recv_wr  = 0,
                        .max_send_sge = MAX_GATHER_ENTRIES,
                        .max_recv_sge = MAX_SCATTER_ENTRIES,
                },
                .qp_type = IBV_QPT_UD,---------->UD type of QP
        };

    struct ibv_qp *qp = ibv_create_qp(pd, &qp_attr);
        if (!qp) {
                fprintf(stderr, "Couldn't create QP\n");
                goto free_cq;
        }
```

B ) QP Change state

```
#define WELL_KNOWN_QKEY 0x11111111

        qp_modify_attr.qp_state       = IBV_QPS_INIT;---> QP State it should later RTS ( for TX) or RTR ( for RX)
        qp_modify_attr.pkey_index     = 0; --->
        qp_modify_attr.port_num       = dev_port; --->( Port no in HCA)
        qp_modify_attr.qkey           = WELL_KNOWN_QKEY;---> only Used in UD and is same both side (RX and TX)
        if (ibv_modify_qp(qp, &qp_modify_attr,
                          IBV_QP_STATE              |
                          IBV_QP_PKEY_INDEX         |
                          IBV_QP_PORT               |
                          IBV_QP_QKEY)) {
                fprintf(stderr, "Failed to modify QP to INIT\n");
                goto free_qp;
        }


        qp_modify_attr.qp_state                   = IBV_QPS_RTR;

        if (ibv_modify_qp(qp, &qp_modify_attr, IBV_QP_STATE)) {
                fprintf(stderr, "Failed to modify QP to RTR\n");
                goto free_qp;
        }
```




RESET
During creation

RTR state
In this state, the QP handles incoming packets. If the QP don't be used as a requester (i.e. Work Requests won't be posted to its send queue), the QP may stay in the RTR state.

RTS state
In most of the applications, the QPs will be transitioned to the RTS state. In this state, the QP can send packets as a reque ster and handle incoming packets.
Even if your QP won't act as a requester, it can be transitioned to this state.