CHAPTER 13

InfiniBand

This chapter was written by Dotan Barak, an InfiniBand Expert. Dotan is a Senior Software Manager at Mellanox Technologies working on RDMA Technologies. Dotan has been working at Mellanox for more than 10 years in various roles, both as a developer and a manager. Additionally, Dotan maintains a blog about the RDMA technology: `http://www.rdmamojo.com`.

Chapter 12 dealt with the wireless subsystem and its implementation in Linux. In this chapter, I will discuss the InfiniBand subsystem and its implementation in Linux. Though the InfiniBand technology might be perceived as a very complex technology for those who are unfamiliar with it, the concepts behind it are surprisingly straightforward, as you will see in this chapter. I will start our discussion with Remote Direct Memory Access (RDMA), and discuss its main data structures and its API. I will give some examples illustrating how to work with RDMA, and conclude this chapter with a short discussion about using RDMA API from the kernel level and userspace.

RDMA and InfiniBand—General

Remote Direct Memory Access (RDMA) is the ability for one machine to access—that is, to read or write to—memory on a remote machine. There are several main network protocols that support RDMA: InfiniBand, RDMA over Converged Ethernet (RoCE) and internet Wide Area RDMA Protocol (iWARP), and all of them share the same API. InfiniBand is a completely new networking protocol, and its specifications can be found in the document "InfiniBand Architecture specifications," which is maintained by the InfiniBand Trade Association (IBTA). RoCE allows you to have RDMA over an Ethernet network, and its specification can be found as an Annex to the InfiniBand specifications. iWARP is a protocol that allows using RDMA over TCP/IP, and its specifications can be found in the document, "An RDMA Protocol Specification," which is being maintained by the RDMA Consortium. **Verbs** is the description of the API to use RDMA from a client code. The RDMA API implementation was introduced to the Linux kernel in version 2.6.11. At the beginning, it supported only InfiniBand, and after several kernel versions, iWARP and RoCE support were added to it as well. When describing the API, I mention only one of them, but the following text refers to all. All of the definitions to this API can be found in `include/rdma/ib_verbs.h`. Here are some notes about the API and the implementation of the RDMA stack:

- Some of the functions are inline functions, and some of them aren't. Future implementation might change this behavior.
- Most of the APIs have the prefix "ib"; however, this API supports InfiniBand, iWARP and RoCE.
- The header `ib_verbs.h` contains functions and structures to be used by:
  - The RDMA stack itself
  - Low-level drivers for RDMA devices
  - Kernel modules that use the stack as consumers

I will concentrate on functions and structures that are relevant only for kernel modules that use the stack as consumers (the third case). The following section discusses the RDMA stack organization in the kernel tree.

The RDMA Stack Organization

Almost all of the kernel RDMA stack code is under `drivers/infiniband` in the kernel tree. The following are some of its important modules (this is not an exhaustive list, as I do not cover the entire RDMA stack in this chapter):

- **CM:** Communication manager (`drivers/infiniband/core/cm.c`)
- **IPoIB:** IP over InfiniBand (`drivers/infiniband/ulp/ipoib/`)
- **iSER:** iSCSI extension for RDMA (`drivers/infiniband/ulp/iser/`)
- **RDS:** Reliable Datagram Socket (`net/rds/`)
- **SRP:** SCSI RDMA protocol (`drivers/infiniband/ulp/srp/`)
- Hardware low-level drivers of different vendors (`drivers/infiniband/hw`)
- **verbs:** Kernel verbs (`drivers/infiniband/core/verbs.c`)
- **uverbs:** User verbs (`drivers/infiniband/core/uverbs_*.c`)
- **MAD:** Management datagram (`drivers/infiniband/core/mad.c`)

Figure 13-1 shows the Linux InfiniBand stack architecture.

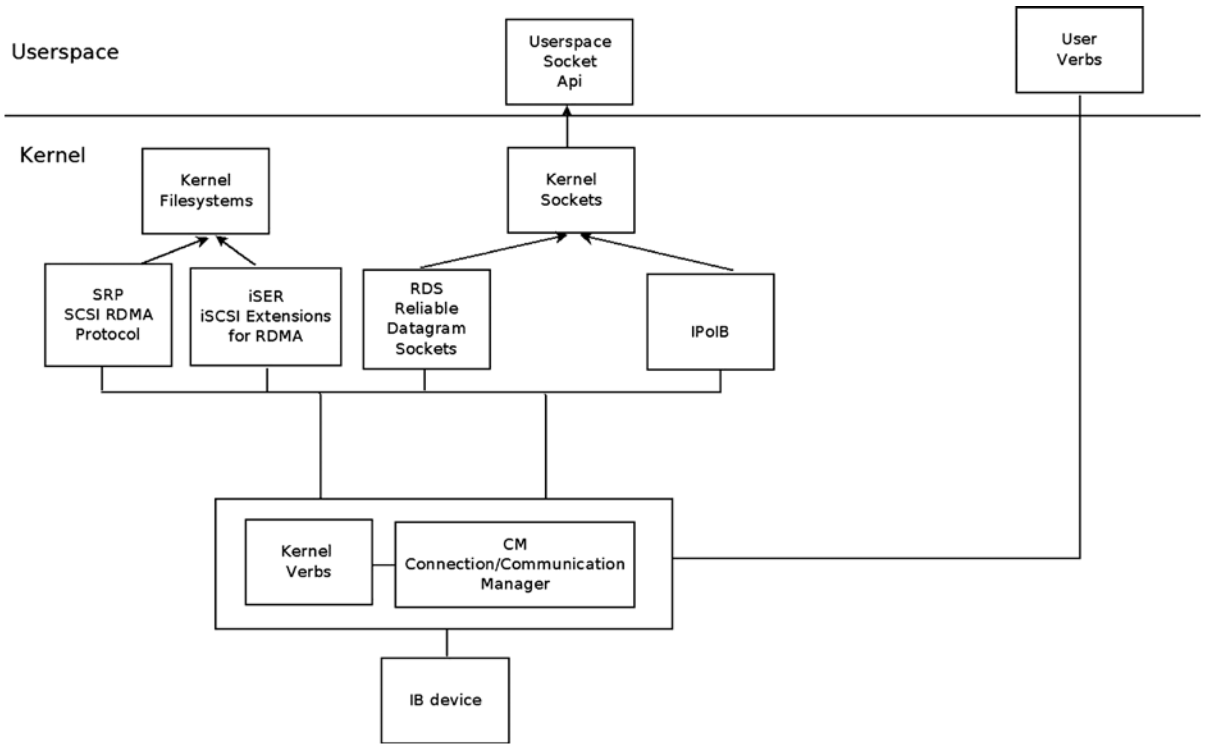

Figure 13-1. *Linux Infiniband stack architecture*

In this section, I covered the RDMA stack organization and the kernel modules that are part of it in the Linux kernel.

RDMA Technology Advantages

Here I will cover the advantages of the RDMA technology and explain the features that make it popular in many markets:

- **Zero copy:** The ability to directly write data to and read data from remote memory allows you to access remote buffers directly without the need to copy it between different software layers.
- **Kernel bypass:** Sending and receiving data from the same context of the code (that is, userspace or kernel level) saves the context switches time.
- **CPU offload:** The ability to send or receive data using dedicated hardware without any CPU intervention allows for decreasing the usage of the CPU on the remote side, because it doesn't perform any active operations.
- **Low latency:** RDMA technologies allow you to reach a very low latency for short messages. (In current hardware and on current servers, the latency for sending up to tens of bytes can be a couple of hundred nanoseconds.)
- **High Bandwidth:** In an Ethernet device, the maximum bandwidth is limited by the technology (that is, 10 or 40 Gbits/sec). In InfiniBand, the same protocol and equipment can be used from 2.5 Gbits/sec up to 120 Gbits/sec. (In current hardware and on current servers, the BW can be upto 56 Gbits/sec.)

InfiniBand Hardware Components

Like in any other interconnect technologies, in InfiniBand several hardware components are described in the spec, some of them are endpoints to the packets (generating packets and the target of the packet), and some of them forward packets in the same subnet or between different subnets. Here I will cover the most common ones:

- **Host Channel Adapter (HCA):** The network adapter that can be placed at a host or at any other system (for example, storage device). This component initiates or is the target of packets.
- **Switch:** A component that knows how to receive a packet from one port and send it to another port. If needed, it can duplicate multicast messages. (Broadcast isn't supported in InfiniBand.) Unlike other technologies, every switch is a very simple device with forwarding tables that are configured by the Subnet Manager (SM), which is an entity that configures and manages the subnet (later on in this section, I will discuss its role in more detail). The switch doesn't learn anything by itself or parse and anlyze packets; it forwards packets only within the same subnet.
- **Router:** A component that connects several different InfiniBand subnets.

A subnet is a set of HCAs, switches, and router ports that are connected together. In this section, I described the various hardware components in InfiniBand, and now I will discuss the addressing of the devices, system, and ports in InfiniBand.

Addressing in InfiniBand

Here are some rules about InfiniBand addressing and an example:

- In InfiniBand, the unique identifier of components is the Globally Unique Identifier (GUID), which is a 64-bit value that is unique in the world.
- Every node in the subnet has a Node GUID. This is the identifier of the node and a constant attribute of it.
- Every port in the subnet, including in HCAs and in switches, has a port GUID. This is the identifier of the port and a constant attribute of it.
- In systems that are made from several components, there can be a system GUID. All of the components in that system have the same system GUID.

Here is an example that demonstrates all the aforementioned GUIDs: a big switch system that is combined from several switch chips. Every switch chip has a unique Node GUID. Every port in every switch has a unique port GUID. All of the chips in that system have the same system GUID.

- Global IDentifier (GID) is used to identify an end port or a multicast group. Every port has at least one valid GID at the GID table in index 0. It is based on the port GUID plus the subnet identifier that this port is part of.
- Local IDentifier (LID) is a 16-bit value that is assigned to every subnet port by the Subnet Manager. A switch is an exception, and the switch management port has the LID assignment, and not all of its ports. Every port can be assigned only one LID, or a contiguous range of LIDs, in order to have several paths to this port. Each LID is unique at a specific point of time in the same subnet, and it is used by the switches when forwarding the packets to know which egress port to use. The unicast LID's range is 0x001 to 0xbfff. The multicast LIDs range is 0xc000 to 0xfffe.

InfiniBand Features

Here we will cover some of the InfiniBand protocol features:

- InfiniBand allows you to configure partitions of ports of HCAs, switches, and routers and allows you to provide virtual isolation over the same physical subnet. Every Partition Key (P_Key) is a 16-bit value that is combined from the following: 15 lsbs are the key value, and the msb is the membership level; 0 is limited membership; and 1 is full membership. Every port has a P_Key table that is being configured by the SM, and every Queue Pair (QP), the actual object in InfiniBand that sends and receives data, is associated with one P_Key index in this table. One QP can send or receive packets from a remote QP only if, in the P_Keys that each of them is associated with, the following is true:
  - The key value is equal.
  - At least one of them has full membership.
- **Queue Key (Q_Key):** An Unreliable Datagram (UD) QP will get unicast or multicast messages from a remote UD QP only if the Q_Key of the message is equal to the Q_Key value of this UD QP.
- **Virtual Lanes (VL):** This is a mechanism for creating multiple virtual links over a single physical link. Every virtual lane represents an autonomic set of buffers for send and receive packets in each port. The number of supported VLs is an attribute of a port.
- **Service Level (SL):** InfiniBand supports up to 16 service levels. The protocol doesn't specify the policy of each level. In InfiniBand, the QoS is implemented using the SL-to-VL mapping and the resources for each VL.
- **Failover:** Connected QPs are QPs that can send packets to or receive packets from only one remote QP. InfiniBand allows defining a primary path and an alternate path for connected QPs. If there is a problem with the primary path, instead of reporting an error, the alternate path will be used automatically.

In the next section, we will look at what packets in InfiniBand look like. This is very useful when you debug problems in InfiniBand.

InfiniBand Packets

Every packet in InfiniBand is a combination of several headers and, in many cases, a payload, which is the data of the messages that the clients want to send. Messages that contain only an ACK or messages with zero bytes (for example, if only immediate data is being sent) won't contain a payload. Those headers describe from where the packet was sent, what the target of the packet is, the used operation, the information needed to separate the packets into messages, and enough information to detect packet loss errors.

Figure 13-2 presents the InfiniBand packet headers.

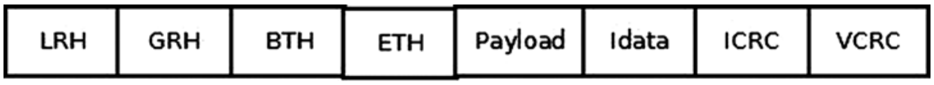

Figure 13-2. *InfiniBand packet headers*

Here are the headers in InfiniBand:

- **Local Routing Header (LRH):** 8 bytes. Always present. It identifies the local source and destination ports of the packet. It also specifies the requested QoS attributes (SL and VL) of the message.
- **Global Routing Header (GRH):** 40 bytes. Optional. Present for multicast packets or packets that travel in multiple subnets. It describes the source and destination ports using GIDs. Its format is identical to the IPv6 header.
- **Base Transport Header (BTH):** 12 bytes. Always present. It specifies the source and destination QPs, the operation, packet sequence number, and partition.
- **Extended Transport Header (ETH):** from 4 to 28 bytes. Optional. Extra family of headers that might be present, depending on the class of the service and the operation used.
- **Payload:** Optional. The data that the client wants to send.
- **Immediate data:** 4 bytes. Optional. Out-of-band, 32-bit value that can be added to Send and RDMA Write operations.
- **Invariant CRC (ICRC):** 4 bytes. Always present. It covers all fields that should not be changed as the packet travels in the subnet.
- **Variant CRC (VCRC):** 2 bytes. Always present. It covers all of the fields of the packet.

Management Entities

The SM is the entity in the subnet that is responsible for analyzing the subnet and configuring it. These are some of its missions:

- Discover the physical topology of the subnet.
- Assign the LIDs and other attributes—such as active MTU, active speeds, and more—to each port in the subnet.
- Configure the forwarding table in the subnet switches.
- Detect any changes in the topology (for example, if new nodes were added or removed from the subnet).
- Handle various errors in the subnet.

Subnet Manager is usually a software entity that can be running in a switch (which is called a *managed switch*) or in any node in the subnet.

Several SMs can be running in a subnet, but only one of them will be active and the rest of them will be in standby mode. There is an internal protocol that performs master election and decides which SM will be active. If the active SM is going down, one of the standby SMs will become the active SM. Every port in the subnet has a Subnet Management Agent (SMA), which is an agent that knows how to receive management messages sent by the SM, handle them, and return a response. Subnet Administrator (SA) is a service that is part of the SM. These are some of its missions:

- Provide information about the subnet—for example, information about how to get from one port to another (that is, a path query).
- Allow you to register to get notifications about events.
- Provide services for management of the subnet, such as joining or leaving a multicast. Those services might cause the SM to (re)configure the subnet.

Communication Manager (CM) is an entity that is capable of running on each port, if the port supports it, to establish, maintain, and tear down QP connections.

RDMA Resources

In the RDMA API, a lot of resources need to be created and handled before any data can be sent or received. All of the resources are in the scope of a specific RDMA device, those resources cannot be shared or used across more than one local device, even if there are multiple devices in the same machine. Figure 13-3 presents the RDMA resource creation hierarchy.

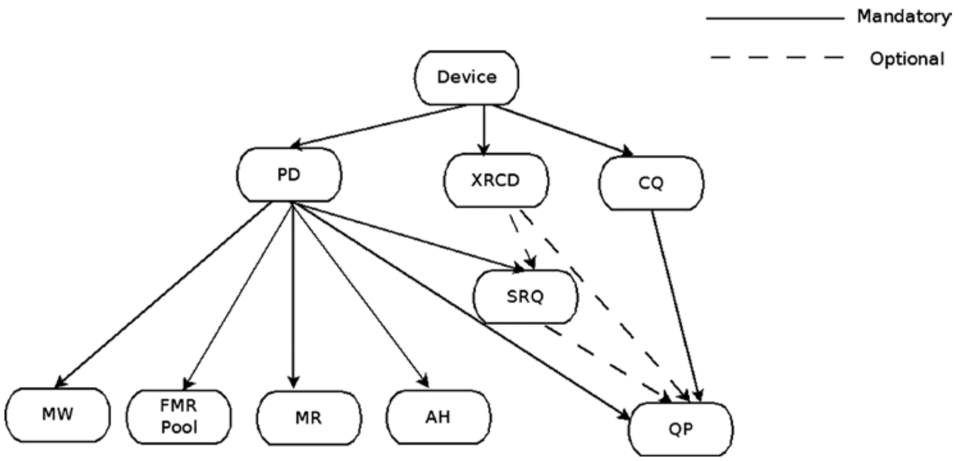

Figure 13-3. *RDMA resource creation hierarchy*

RDMA Device

The client needs to register with the RDMA stack in order to be notified about any RDMA device that is being added to the system or removed from it. After the initial registration, the client is notified for all existing RDMA devices. A callback will be invoked for every RDMA device, and the client can start working with these devices in the following ways:

- Query the device for various attributes
- Modify the device attributes
- Create, work with and destroy resources

The ib_register_client() method registers a kernel client that wants to use the RDMA stack. The specified callbacks will be invoked for every new InfiniBand device that currently exists in the system and that will be added to or removed from (using hot-plug functionality) the system. The ib_unregister_client() method unregisters a kernel client that wants to stop using the RDMA stack. Usually, it is called when the driver is being unloaded. Here is an sample code that shows how to register the RDMA stack in a kernel client:

```
static void my_add_one(struct ib_device *device)
{
...
}

static void my_remove_one(struct ib_device *device)
{
...
}

static struct ib_client my_client = {
    .name   = "my RDMA module",
    .add    = my_add_one,
    .remove = my_remove_one
};

static int __init my_init_module(void)
{
    int ret;

    ret = ib_register_client(&my_client);
    if (ret) {
        printk(KERN_ERR "Failed to register IB client\n");
        return ret;
    }

    return 0;
}

static void __exit my_cleanup_module(void)
{
    ib_unregister_client(&my_client);
}

module_init(my_init_module);
module_exit(my_cleanup_module);
```

Following here is a description of several more methods for handling an InfiniBand device.

- The ib_set_client_data() method sets a client context to be associated with an InfiniBand device.
- The ib_get_client_data() method returns the client context that was associated with an InfiniBand device using the ib_set_client_data() method.

- The `ib_register_event_handler()` method registers a callback to be called for every asynchronous event that will occur to the InfiniBand device. The callback structure must be initialized with the INIT_IB_EVENT_HANDLER macro.
- The `ib_unregister_event_handler()` method unregisters the event handler.
- The `ib_query_device()` method queries the InfiniBand device for its attributes. Those attributes are constant and won't be changed in subsequent calls of this method.
- The `ib_query_port()` method queries the InfiniBand device port for its attributes. Some of those attributes are constant, and some of them might be changed in subsequent calls of this method—for example, the port LID, state, and some other attributes.
- The `rdma_port_get_link_layer()` method returns the link layer of the device port.
- The `ib_query_gid()` method queries the InfiniBand device port's GID table in a specific index. The `ib_find_gid()` method returns the index of a specific GID value in a port's GID table.
- The `ib_query_pkey()` method queries the InfiniBand device port's P_Key table in a specific index. The `ib_find_pkey()` method returns the index of a specific P_Key value in a port's P_Key table.

Protection Domain (PD)

A PD allows associating itself with several other RDMA resources—such as SRQ, QP, AH, or MR—in order to provide a means of protection among them. RDMA resources that are associated with PDx cannot work with RDMA resources that were associated with PDy. Trying to mix those resources will end with an error. Typically, every module will have one PD. However, if a specific module wants to increase its security, it will use one PD for each remote QP or service that it uses. Allocation and deallocation of a PD is done like this:

- The `ib_alloc_pd()` method allocates a PD. It takes as an argument the pointer of the device object that was returned when the driver callback was called after its registration.
- The `ib_dealloc_pd()` method deallocates a PD. It is usually called when the driver is being unloaded or when the resources that are associated with the PD are being destroyed.

Address Handle (AH)

An AH is used in the Send Request of a UD QP to describe the path of the message from the local port to the remote port. The same AH can be used for several QPs if all of them send messages to the same remote port using the same attributes. Following is a description of four methods related to the AH:

- The `ib_create_ah()` method creates an AH. It takes as an argument a PD and attributes for the AH. The AH attributes of the AH can be filled directly or by calling the `ib_init_ah_from_wc()` method, which gets as a parameter a received Work Completion (`ib_wc` object) that includes the attributes of a successfully completed incoming message, and the port it was received from. Instead of calling the `ib_init_ah_from_wc()` method and then the `ib_create_ah()` method, one can call the `ib_create_ah_from_wc()` method.
- The `ib_modify_ah()` method modifies the attributes of an existing AH.
- The `ib_query_ah()` method queries for the attributes of an existing AH.
- The `ib_destroy_ah()` method destroys an AH. It is called when there isn't a need to send any further messages to the node that the AH describes the path to.

Memory Region (MR)

Every memory buffer that is accessed by the RDMA device needs to be registered. During the registration process, the following tasks are performed on the memory buffer:

- Separate the contiguous memory buffer to memory pages.
- The mapping of the virtual-to-physical translation will be done.
- The memory pages permission is checked to ensure that the requested permissions for the MR is supported by them.
- The memory pages are pinned, to prevent them from being swapped out. This keeps the virtual-to-physical mapping unchanged.

After a successful memory registration is completed, it has two keys:

- **Local key (lkey):** A key for accessing this memory by local Work Requests.
- **Remote key (rkey):** A key for accessing this memory by a remote machine using RDMA operations.

Those keys will be used in Work Requests when referring to those memory buffers. The same memory buffers can be registered several times, even with different permissions. The following is a description of some methods related to the MR:

- The `ib_get_dma_mr()` method returns a Memory Region for system memory that is usable for DMA. It takes a PD and the requested access permission for the MR as arguments.
- The `ib_dma_map_single()` method maps a kernel virtual address, that was allocated by the `kmalloc()` method family, to a DMA address. This DMA address will be used to access local and remote memory. The `ib_dma_mapping_error()` method should be used to check whether the mapping was successful.
- The `ib_dma_unmap_single()` method unmaps a DMA mapping that was done using `ib_dma_map_single()`. It should be called when this memory isn't needed anymore.

■ **Note** There are some more flavors of `ib_dma_map_single()` that allow the mapping of pages, mapping according to DMA attributes, mapping using a scatter/gather list, or mapping using a scatter/gather list with DMA attributes: `ib_dma_map_page()`, `ib_dma_map_single_attrs()`, `ib_dma_map_sg()`, and `ib_dma_map_sg_attrs()`. All of them have corresponding unmap functions.

Before accessing a DMA mapped memory, the following methods should be called:

- `ib_dma_sync_single_for_cpu()` if the DMA region is going to be accessed by the CPU, or `ib_dma_sync_single_for_device()` if the DMA region is going to be accessed by the InfiniBand device.
- The `ib_dma_alloc_coherent()` method allocates a memory block that can be accessed by the CPU and maps it for DMA.
- The `ib_dma_free_coherent()` method frees a memory block that was allocated using `ib_dma_alloc_coherent()`.

- The `ib_reg_phys_mr()` method takes a set of physical pages, registers them, and prepares a virtual address that can be accessed by an RDMA device. If you want to change it after it was created, you should call the `ib_rereg_phys_mr()` method.
- The `ib_query_mr()` method retrieves the attributes of a specific MR. Note that most low-level drivers do not implement this method.
- The `ib_dereg_mr()` method deregisters an MR.

Fast Memory Region (FMR) Pool

Registration of a Memory Region is a "heavy" procedure that might take some time to complete, and the context that performs it even might sleep if required resources aren't available when it is called. This behavior might be problematic when performed in certain contexts—for example, in the interrupt handler. Working with an FMR pool allows you to work with FMRs with registrations that are "lightweight" and can be registered in any context. The API of the FMR pool can be found in `include/rdma/ib_fmr_pool.h`.

Memory Window (MW)

Enabling a remote access to a memory can be done in two ways:

- Register a memory buffer with remote permissions enabled.
- Register a Memory Region and then bind a Memory Window to it.

Both of those ways will create a remote key (`rkey`) that can be used to access this memory with the specified permissions. However, if you wish to invalidate the `rkey` to prevent remote access to this memory, performing Memory Region deregistration might be a heavy procedure. Working with Memory Window on this Memory Region and binding or unbinding it when needed might provide a "lightweight" procedure for enabling and disabling remote access to memory. Following is a description of three methods related to the MW:

- The `ib_alloc_mw()` method allocates a Memory Window. It takes a PD and the MW type as arguments.
- The `ib_bind_mw()` method binds a Memory Window to a specified Memory Region with a specific address, size, and remote permissions by posting a special Work Request to a QP. It is called when you want to allow temporary remote access to its memory. A Work Completion in the Send Queue of the QP will be generated to describe the status of this operation. If `ib_bind_mw()` was called to a Memory Windows that is already bounded, to the same Memory Region or a different one, the previous binding will be invalidated.
- The `ib_dealloc_mw()` method deallocates the specified MW object.

Completion Queue (CQ)

Every posted Work Request, to either Send or Receive Queue, is considered outstanding until there is a corresponding Work Completion for it or for any Work Request that was posted after it. While the Work Request is outstanding, the content of the memory buffers that it points to is undetermined:

- If the RDMA device reads this memory and sends its content over the wire, the client cannot know if this buffer can be (re)used or released. If this is a reliable QP, a successful Work Completion means that the message was received by the remote side. If this is an unreliable QP, a successful Work Completion means that the message was sent.
- If the RDMA device writes a message to this memory, the client cannot know if this buffer contains the incoming message.

A Work Completion specifies that the corresponding Work Request was completed and provides some information about it: its status, the used opcode, its size, and so on. A CQ is an object that contains the Work Completions. The client needs to poll the CQ in order to read the Work Completions that it has. The CQ works on a first-in, first-out (FIFO) basis: the order of Work Completions that will be de-queued from it by the client will be according to the order that they were enqueued to the CQ by the RDMA device. The client can read the Work Completions in polling mode or request to get a notification when a new Work Completion is added to the CQ. A CQ cannot hold more Work Completions than its size. If more Work Completions than its capacity are added to it, a Work Completion with an error will be added, a CQ error asynchronous event will be generated, and all the Work Queues associated with it will get an error. Here are some methods related to the CQ:

- The `ib_create_cq()` method creates a CQ. It takes the following as its arguments: the pointer of the device object that was returned when the driver callback was called after its registration and the attributes for the CQ, including its size and the callbacks that will be called when there is an asynchronous event on this CQ or a Work Completion is added to it.
- The `ib_resize_cq()` method changes the size of a CQ. The new number of entries cannot be less than the number of the Work Completions that currently populate the CQ.
- The `ib_modify_cq()` method changes the moderation parameter for a CQ. A Completion event will be generated if at least a specific number of Work Completions enter the CQ or a timeout will expire. Using it might help reduce the number of interrupts that happen in an RDMA device.
- The `ib_peek_cq()` method returns the number of available Work Completions in a CQ.
- The `ib_req_notify_cq()` method requests that a Completion event notification be generated when the next Work Completion, or Work Completion that includes a solicited event indication, is added to the CQ. If no Work Completion is added to the CQ after the `ib_req_notify_cq()` method was called, no Completion event notification will occur.
- The `ib_req_ncomp_notif()` method requests that a Completion event notification be created when a specific number of Work Completions exists in the CQ. Unlike the `ib_req_notify_cq()` method, when calling the `ib_req_ncomp_notif()` method, a Completion event notification will be generated even if the CQ currently holds this number of Work Completions.
- The `ib_poll_cq()` method polls for Work Completions from a CQ. It reads the Work Completions from the CQ in the order they were added to it and removes them from it.

Here is an example of a code that empties a CQ—that is, reads all the Work Completions from a CQ, and checks their status:

```
struct ib_wc wc;
int num_comp = 0;

while (ib_poll_cq(cq, 1, &wc) > 0) {
    if (wc.status != IB_WC_SUCCESS) {
        printk(KERN_ERR "The Work Completion[%d] has a bad status %d\n",
                         num_comp, wc.status);
        return -EINVAL;
    }
    num_comp ++;
}
```

eXtended Reliable Connected (XRC) Domain

An XRC Domain is an object that is used to limit the XRC SRQs an incoming message can target. That XRC domain can be associated with several other RDMA resources that work with XRC, such as SRQ and QP.

Shared Receive Queue (SRQ)

An SRQ is a way for the RDMA architecture to be more scalable on the receive side. Instead of having a separate Receive Queue for every Queue Pair, there is a shared Receive Queue that all of the QPs are connected to. When they need to consume a Receive Request, they fetch it from the SRQ. Figure 13-4 presents QPs that are associated with an SRQ.

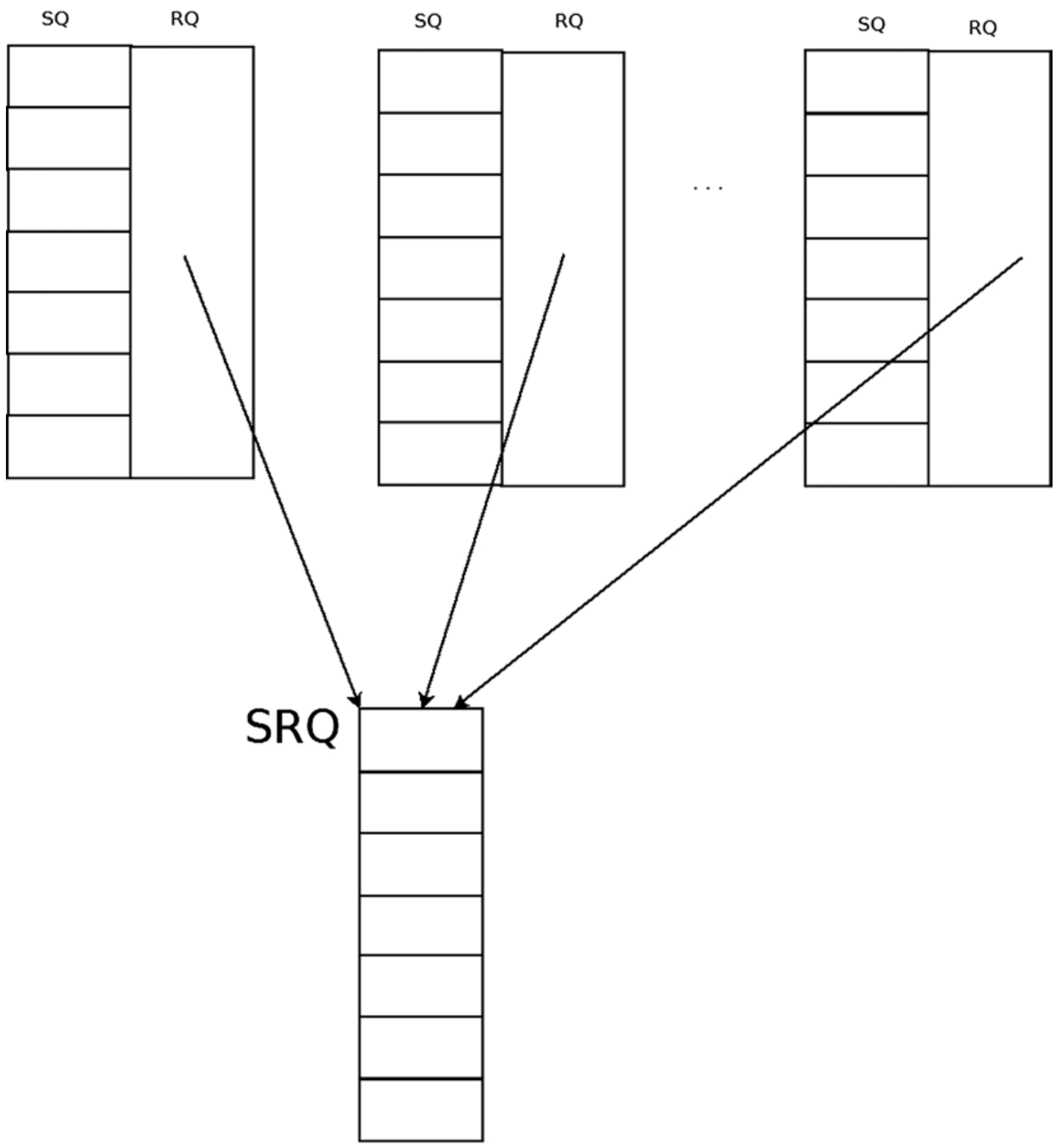

Figure 13-4. *QPs that are associated with an SRQ*

Here's what you do if you have *N* QPs, and each of them might receive a burst of *M* messages at a random time:

- Without using an SRQ, you post N*M Receive Requests.
- With SRQs, you post K*M (where K << N) Receive Requests.

Unlike a QP, which doesn't have any mechanism to determine the number of outstanding Work Requests in it, with an SRQ you can set a watermark limit. When the number of Receive Requests drops below this limit, an SRQ limit asynchronous event will be created for this SRQ. The downside of using an SRQ is that you cannot predict which QP will consume every posted Receive Request from the SRQ, so the message size that each posted Receive Request will be able to hold must be the maximum incoming message size that any of the QPs might get. This limitation can be handled by creating several SRQs, one for each different maximum message size, and associating them with the relevant QPs according to their expected message sizes.

Here is a description of some methods related to the SRQ and an example:

- The `ib_create_srq()` method creates an SRQ. It takes a PD and attributes for the SRQ.
- The `ib_modify_srq()` method modifies the attributes of the SRQ. It is used to set a new watermark value for the SRQ's limit event or to resize the SRQ for devices that support it.

Here is an example for setting the value of the watermark to get an asynchronous event when the number of RRs in the SRQ drops below 5:

```
struct ib_srq_attr srq_attr;
int ret;

memset(&srq_attr, 0, sizeof(srq_attr));
srq_attr.srq_limit = 5;

ret = ib_modify_srq(srq, &srq_attr, IB_SRQ_LIMIT);
if (ret) {
    printk(KERN_ERR "Failed to set the SRQ's limit value\n");
    return ret;
}
```

Following here is a description of several more methods for handling an SRQ.

- The `ib_query_srq()` method queries for the current SRQ attributes. This method is usually used to check the content of the SRQ's limit value. The value 0 in the `srq_limit` member in the `ib_srq_attr` object means that there isn't any SRQ limit watermark set.
- The `ib_destroy_srq()` method destroys an SRQ.
- The `ib_post_srq_recv()` method takes a linked list of Receive Requests as an argument and adds them to a specified Shared Receive Queue for future processing.

Here is an example for posting a single Receive Request to an SRQ. It saves an incoming message in a memory buffer, using its registered DMA address in a single gather entry:

```
struct ib_recv_wr wr, *bad_wr;
struct ib_sge sg;
int ret;

memset(&sg, 0, sizeof(sg));
sg.addr   = dma_addr;
sg.length = len;
sg.lkey   = mr->lkey;
```

```
memset(&wr, 0, sizeof(wr));
wr.next     = NULL;
wr.wr_id    = (uintptr_t)dma_addr;
wr.sg_list  = &sg;
wr.num_sge  = 1;

ret = ib_post_srq_recv(srq, &wr, &bad_wr);
if (ret) {
    printk(KERN_ERR "Failed to post Receive Request to an SRQ\n");
    return ret;
}
```

Queue Pair (QP)

Queue Pair is the actual object used to send and receive data in InfiniBand. It has two separate Work Queues: Send and Receive Queues. Every Work Queue has a specific number of Work Requests (WR) that can be posted to it, a number of scatter/gather elements that are supported for each WR, and a CQ to which the Work Requests whose processing has ended will add Work Completion. Those Work Queues can be created with similar or different attributes—for example, the number of WRs that can be posted to each Work Queue. The order in each Work Queue is guaranteed—that is, the processing of a Work Request in the Send Queue will start according to the order of the Send Requests submission. And the same behavior applies to the Receive Queue. However, there isn't any relation between them—that is, an outstanding Send Request can be processed even if it was posted after posting a Receive Request to the Receive Queue. Figure 13-5 presents a QP.

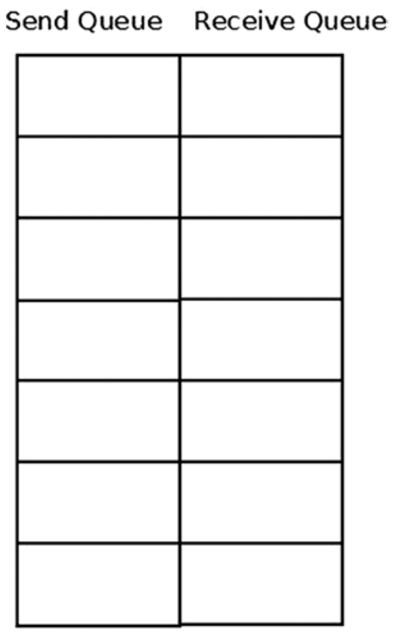

Figure 13-5. *QP (Queue Pair)*

Upon creation, every QP has a unique number across the RDMA device at a specific point in time.

QP Transport Types

There are several QP transport types supported in InfiniBand:

- **Reliable Connected (RC):** One RC QP is connected to a single remote RC QP, and reliability is guaranteed—that is, the arrival of all packets according to their order with the same content that they were sent with is guaranteed. Every message is fragmented to packets with the size of the path MTU at the sender side and defragmented at the receiver side. This QP supports Send, RDMA Write, RDMA Read, and Atomic operations.
- **Unreliable Connected (UC):** One UC QP is connected to a single remote UC QP, and reliability isn't guaranteed. Also, if a packet in a message is lost, the whole message is lost. Every message is fragmented to packets with the size of the path MTU at the sender side and defragmented at the receiver side. This QP supports Send and RDMA Write operations.
- **Unreliable Datagram (UD):** One UD QP can send a unicast message to any UD QP in the subnet. Multicast messages are supported. Reliability isn't guaranteed. Every message is limited to one packet message, with its size limited to the path MTU size. This QP supports only Send operations.
- **eXtended Reliable Connected (XRC):** Several QPs from the same node can send messages to a remote SRQ in a specific node. This is useful for decreasing the number of QPs between two nodes from the order of the number of CPU cores—that is, QP in a process per core, to one QP. This QP supports all operations that are supported by RC QP. This type is relevant only for userspace applications.
- **Raw packet:** Allows the client to build a complete packet, including the L2 headers, and send it as is. At the receiver side, no header will be stripped by the RDMA device.
- **Raw IPv6/Raw Ethertype:** QPs that allow sending raw packets that aren't interpreted by the IB device. Currently, both of these types aren't supported by any RDMA device.

There are special QP transport types that are used for subnet management and special services:

- **SMI/QP0:** QP used for subnet managements packets.
- **GSI/QP1:** QP used for general services packets.

The `ib_create_qp()` method creates a QP. It takes a PD and the requested attributes that this QP will be created with as arguments. Here is an example for creating an RC QP using a PD that was created, with two different CQs: one for the Send Queue and one for the Receive Queue.

```
struct ib_qp_init_attr init_attr;
struct ib_qp *qp;

memset(&init_attr, 0, sizeof(init_attr));
init_attr.event_handler       = my_qp_event;
init_attr.cap.max_send_wr     = 2;
init_attr.cap.max_recv_wr     = 2;
init_attr.cap.max_recv_sge    = 1;
init_attr.cap.max_send_sge    = 1;
init_attr.sq_sig_type         = IB_SIGNAL_ALL_WR;
init_attr.qp_type             = IB_QPT_RC;
init_attr.send_cq             = send_cq;
init_attr.recv_cq             = recv_cq;
```

```
    qp = ib_create_qp(pd, &init_attr);
    if (IS_ERR(qp)) {
        printk(KERN_ERR "Failed to create a QP\n");
        return PTR_ERR(qp);
    }
```

QP State Machine

A QP has a state machine that defines what the QP is capable of doing at each state:

- **Reset state:** Each QP is generated at this state. At this state, no Send Requests or Receive Requests can be posted to it. All incoming messages are silently dropped.
- **Initialized state:** At this state, no Send Requests can be posted to it. However, Receive Requests can be posted, but they won't be processed. All incoming messages are silently dropped. It is a good practice to post a Receive Request to a QP at this state before moving it to RTR (Ready To Receive). Doing this prevents a case where remote QP sends messages need to consume a Receive Request but such were not posted yet.
- **Ready To Receive (RTR) state:** At this state, no Send Requests can be posted to it, but Receive Requests can be posted and processed. All incoming messages will be handled. The first incoming message that is received at this state will generate the communication-established asynchronous event. A QP that only receives messages can stay at this state.
- **Ready To Send (RTS) state**: At this state, both Send Requests and Receive Requests can be posted and processed. All incoming messages will be handled. This is the common state for QPs.
- **Send Queue Drained (SQD) state:** At this state, the QP completes the processing of all the Send Requests that their processing has started. Only when there aren't any messages that can be sent, you can change some of the QP attributes. This state is separated into two internal states:
  - **Draining:** Messages are still being sent.
  - **Drained:** The sending of the messages was completed.
- **Send Queue Error (SQE) state:** The RDMA device automatically moves a QP to this state when there is an error in the Send Queue for unreliable transport types. The Send Request that caused the error will be completed with the error reason, and all of the consecutive Send Requests will be flushed. The Receive Queue will still work—that is, Receive Requests can be posted, and incoming messages will be handled. The client can recover from this state and modify the QP state back to RTS.
- **Error state:** At this state, all of the outstanding Work Requests will be flushed. The RDMA device can move the QP to this state if this is a reliable transport type and there was an error with a Send Request, or if there was an error in the Receive Queue regardless of which transport type was used. All incoming messages are silently dropped.

A QP can be transitioned by `ib_modify_qp()` from any state to the Reset state and to the Error state. Moving the QP to the Error state will flush all of the outstanding Work Requests. Moving the QP to the Reset state will clear all previously configured attributes and remove all of the outstanding Work Request and Work Completions that were ended on this QP in the Completion Queues that this QP is working with. Figure 13-6 presents a QP state machine diagram.

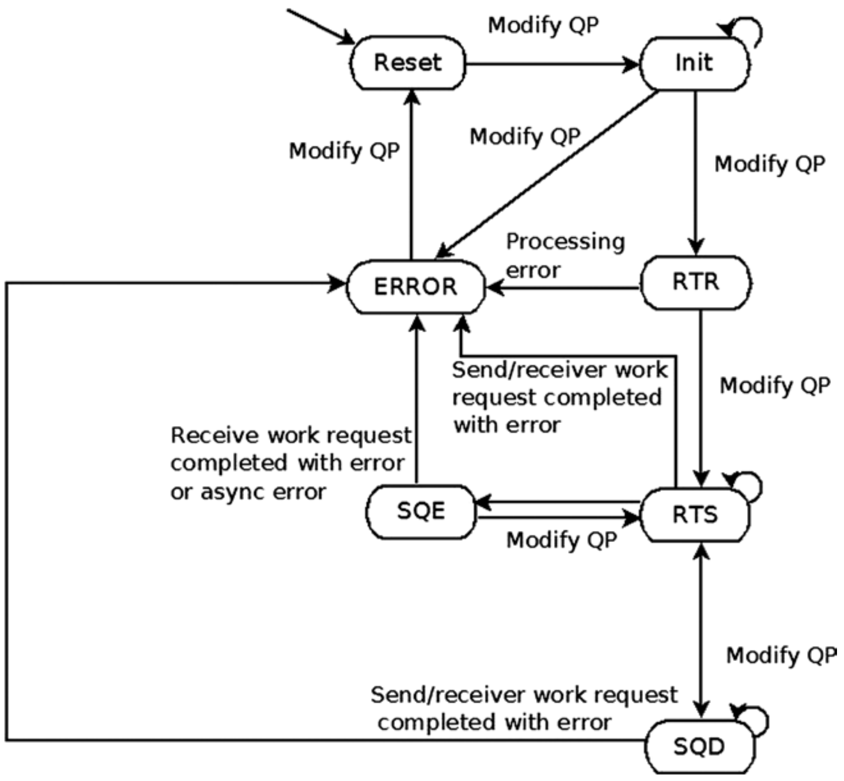
Modify QP
Reset
Init
Modify QP
Modify QP
Modify QP
Processing error
ERROR
RTR
Send/receiver work request completed with error
Modify QP
Receive work request completed with error or async error
SQE
RTS
Modify QP
Modify QP
Send/receiver work request completed with error
SQD

Figure 13-6. *QP state machine*

The `ib_modify_qp()` method modifies the attributes of a QP. It takes as an argument the QP to modify and the attributes of the QP that will be modified. The state machine of the QP can be changed according to the diagram shown in Figure 13-6. Every QP transport type requires different attributes to be set in each QP state transition.

Here is an example for modifying a newly created RC QP to the RTS state, in which it can send and receive packets. The local attributes are the outgoing port, the used SL, and the starting Packet Serial Number for the Send Queue. The remote attributes needed are the Receive PSN, the QP number, and the LID of the port that it uses.

```
struct ib_qp_attr attr = {
    .qp_state        = IB_QPS_INIT,
    .pkey_index      = 0,
    .port_num        = port,
    .qp_access_flags = 0
};

ret = ib_modify_qp(qp, &attr,
          IB_QP_STATE          |
          IB_QP_PKEY_INDEX     |
          IB_QP_PORT           |
          IB_QP_ACCESS_FLAGS);
```

```
if (ret) {
          printk(KERN_ERR "Failed to modify QP to INIT state\n");
          return ret;
}

attr.qp_state              = IB_QPS_RTR;
attr.path_mtu              = mtu;
attr.dest_qp_num           = remote->qpn;
attr.rq_psn                = remote->psn;
attr.max_dest_rd_atomic    = 1;
attr.min_rnr_timer         = 12;
attr.ah_attr.is_global     = 0;
attr.ah_attr.dlid          = remote->lid;
attr.ah_attr.sl            = sl;
attr.ah_attr.src_path_bits = 0,
attr.ah_attr.port_num      = port

ret = ib_modify_qp(ctx->qp, &attr,
          IB_QP_STATE              |
          IB_QP_AV                 |
          IB_QP_PATH_MTU           |
          IB_QP_DEST_QPN           |
          IB_QP_RQ_PSN             |
          IB_QP_MAX_DEST_RD_ATOMIC |
          IB_QP_MIN_RNR_TIMER);
if (ret) {
  printk(KERN_ERR "Failed to modify QP to RTR state\n");
  return ret;
}

attr.qp_state       = IB_QPS_RTS;
attr.timeout        = 14;
attr.retry_cnt      = 7;
attr.rnr_retry      = 6;
attr.sq_psn         = my_psn;
attr.max_rd_atomic  = 1;
ret = ib_modify_qp(ctx->qp, &attr,
          IB_QP_STATE            |
          IB_QP_TIMEOUT          |
          IB_QP_RETRY_CNT        |
          IB_QP_RNR_RETRY        |
          IB_QP_SQ_PSN           |
          IB_QP_MAX_QP_RD_ATOMIC);
if (ret) {
  printk(KERN_ERR "Failed to modify QP to RTS state\n");
  return ret;
}
```

Following here is a description of several more methods for handling a QP:

- The `ib_query_qp()` method queries for the current QP attributes. Some of the attributes are constant (the values that the client specifies), and some of them can be changed (for example, the state).
- The `ib_destroy_qp()` method destroys a QP. It is called when the QP isn't needed anymore.

Work Request Processing

Every posted Work Request, to either the Send or Receive Queue, is considered outstanding until there is a Work Completion, which was polled from the CQ which is associated with this Work Queue for this Work Request or for Work Requests in the same Work Queue that were posted after it. Every outstanding Work Request in the Receive Queue will end with a Work Completion. A Work Request processing flow in a Work Queue is according to the diagram shown in Figure 13-7.

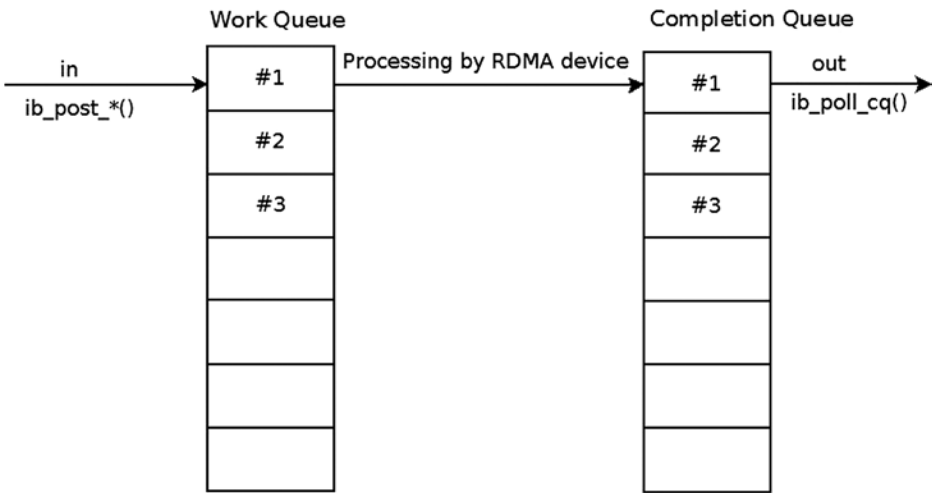

Figure 13-7. *Work Request processing flow*

In the Send Queue, you can choose (when creating a QP) whether you want every Send Request to end with a Work Completion or whether you want to select the Send Requests that will end with Work Completions—that is, selective signaling. You might encounter an error for an unsignaled Send Request; nevertheless, a Work Completion with bad status will be generated for it.

When a Work Request is outstanding one cannot (re)use or free the resources that were specified in it when posting this Work Request. For example:

- When posting a Send Request for a UD QP, the AH cannot be freed.
- When posting a Receive Request, the memory buffers that were referred to in a scatter/gather (s/g) list cannot be read, because it is unknown if the RDMA device already wrote the data in them.

"Fencing" is the ability to prevent the processing of a specific Send Request until the processing of the previous RDMA Read and Atomic operations ends. Adding the Fence indication to a Send Request can be useful, for example, when using RDMA Read from a remote address and sending the data, or part of it, in the same Send Queue. Without fencing, the send operation might start before the data is retrieved and available in local memory. When posting a Send Request to a UC or RC QP, the path to the target is known, because it was provided when moving the QP to the RTR state. However, when posting a Send Request to a UD QP, you need to add an AH to describe the path to the target(s) of this message. If there is an error related to the Send Queue, and if this is an Unreliable transport type, the Send Queue will move to the Error state (that is, the SQE state) but the Receive Queue will still be fully functional. The client can recover from this state and change the QP state back to RTS. If there is an error related to the Receive Queue, the QP will be moved to the Error state because this is an unrecoverable error. When a Work Queue is moved to the Error state, the Work Request that caused the error is ended with a status that indicates the nature of the error and the rest of the Work Requests in this Queue are flushed with error.

Supported Operations in the RDMA Architecture

There are several operation types supported in InfiniBand:

- **Send:** Send a message over the wire. The remote side needs to have a Receive Request available, and the message will be written in its buffers.
- **Send with Immediate:** Send a message over the wire with an extra 32 bits of out-of-band data. The remote side needs to have a Receive Request available, and the message will be written in its buffers. This immediate data will be available in the Work Completion of the receiver.
- **RDMA Write:** Send a message over the wire to a remote address.
- **RDMA Write with Immediate:** Send a message over the wire, and write it to a remote address. The remote side needs to have a Receive Request available. This immediate data will be available in the Work Completion of the receiver. This operation can be seen as RDMA Write + Send with immediate with a zero-byte message.
- **RDMA Read:** Read a remote address, and fill the local buffer with its content.
- **Compare and Swap:** Compare the content of a remote address with valueX; if they are equal, replace its content with the valueY. All of this is performed in an atomic way. The original remote memory content is sent and saved locally.
- **Fetch and Add:** Add a value to the content of a remote address in an atomic way. The original remote memory content is sent and saved locally.
- **Masked Compare and Swap:** Compare the part of the content using maskX of a remote address with valueX; if they are equal, replace part of its content using the bits in maskY with valueY. All of this is performed in an atomic way. The original remote memory content is sent and saved locally.
- **Masked Fetch and Add:** Add a value to the content of a remote address in an atomic way, and change only the bits that are specified in the mask. The original remote memory content is sent and saved locally.
- **Bind Memory Window:** Binds a Memory Windows to a specific Memory Region.
- **Fast registration:** Registers a Fast Memory Region using a Work Request.
- **Local invalidate:** Invalidates a Fast Memory Region using a Work Request. If someone uses its old `lkey`/`rkey`, it will be considered an error. It can be combined with send/RDMA read; in such a case, first the send/read will be performed, and only then this Fast Memory Region will be invalidated.

The Receive Request specifies where the incoming message will be saved for operations that consume a Receive Request. The total size of the memory buffers specified in the scatter list must be equal to or greater than the size of the incoming message.

For UD QP, because the origin of the message is unknown in advance (same subnet or another subnet, unicast or multicast message), an extra 40 bytes, which is the GRH header size, must be added to the Receive Request buffers. The first 40 bytes will be filled with the GRH of the message, if such is available. This GRH information describes how to send a message back to the sender. The message itself will start at offset 40 in the memory buffers that were described in the scatter list.

The `ib_post_recv()` method takes a linked list of Receive Requests and adds them to the Receive Queue of a specific QP for future processing. Here is an example for posting a single Receive Request for a QP. It saves an incoming message in a memory buffer using its registered DMA address in a single gather entry. qp is a pointer to a QP that was created using ib_create_qp(). The memory buffer is a block that was allocated using kmalloc() and mapped for DMA using ib_dma_map_single(). The used lkey is from the MR that was registered using ib_get_dma_mr().

```
struct ib_recv_wr wr, *bad_wr;
struct ib_sge sg;
int ret;

memset(&sg, 0, sizeof(sg));
sg.addr   = dma_addr;
sg.length = len;
sg.lkey   = mr->lkey;

memset(&wr, 0, sizeof(wr));
wr.next     = NULL;
wr.wr_id    = (uintptr_t)dma_addr;
wr.sg_list  = &sg;
wr.num_sge  = 1;

ret = ib_post_recv(qp, &wr, &bad_wr);

if (ret) {
    printk(KERN_ERR "Failed to post Receive Request to a QP\n");
    return ret;
}
```

The ib_post_send() method takes as an argument a linked list of Send Requests and adds them to the Send Queue of a specific QP for future processing. Here is an example for posting a single Send Request of a Send operation for a QP. It sends the content of a memory buffer using its registered DMA address in a single gather entry.

```
struct ib_sge sg;
struct ib_send_wr wr, *bad_wr;
int ret;

memset(&sg, 0, sizeof(sg));
sg.addr   = dma_addr;
sg.length = len;
sg.lkey   = mr->lkey;

memset(&wr, 0, sizeof(wr));
wr.next       = NULL;
wr.wr_id      = (uintptr_t)dma_addr;
wr.sg_list    = &sg;
wr.num_sge    = 1;
wr.opcode     = IB_WR_SEND;
wr.send_flags = IB_SEND_SIGNALED;

ret = ib_post_send(qp, &wr, &bad_wr);
```

```
if (ret) {
    printk(KERN_ERR "Failed to post Send Request to a QP\n");
    return ret;
}
```

Work Completion Status

Every Work Completion can be ended successfully or with an error. If it ends successfully, the operation was finished and the data was sent according to the transport type reliability level. If this Work Completion contains an error, the content of the memory buffers is unknown. There can be many reasons that the Work Request status indicates that there is an error: protection violation, bad address, and so on. The violation errors won't perform any retransmission. However, there are two special retry flows that are worth mentioning. Both of them are done automatically by the RDMA device, which retransmit packets, until the problem is solved or it exceeds the number of retransmissions. If the issue was solved, the client code won't be aware that this even happened, besides a temporary performance hiccup. This is relevant only for Reliable transport types.

Retry Flow

If the receiver side didn't return any ACK or NACK to the sender side within the expected timeout, the sender might send the message again, according to the timeout and the retry count attributes that were configured in the QP attributes. There might be several reasons for having such a problem:

- The attributes of the remote QP or the path to it aren't correct.
- The remote QP state didn't get to (at least) the RTR state.
- The remote QP state moved to the Error state.
- The message itself was dropped on the way from the sender to the receiver (for example, a CRC error).
- The ACK or NACK of messages was dropped on the way from the receiver to the sender (for example, a CRC error).

Figure 13-8 presents the retry flow becasue of a packet loss that overcame a packet drop.

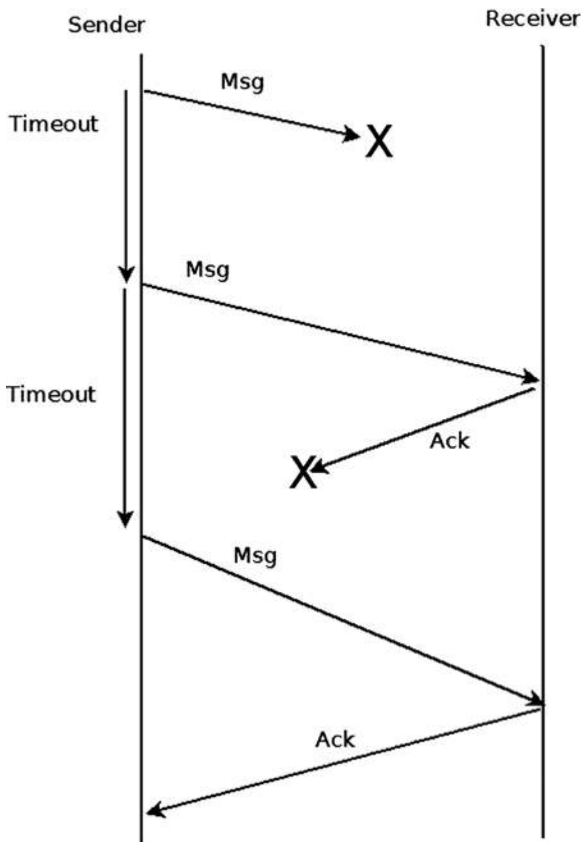

Figure 13-8. *A retry flow (on reliable transport types)*

If eventually the ACK/NACK is received by the sender QP successfully, it will continue to send the rest of the messages. If any message in the future has this problem too, the retry flow will be done again for this message as well, without any history that this was done before. If even after retrying several times the receiver side still doesn't respond, there will be a Work Completion with Retry Error on the sender side.

Receiver Not Ready (RNR) Flow

If the receiver side got a message that needs to consume a Receive Request from the Receiver Queue, but there isn't any outstanding Receive Request, the receiver will send back to the sender an RNR NACK. After a while, according to the time that was specified in the RNR NACK, the sender will try to send the message again.

If eventually the receiver side posts a Receiver Request in time, and the incoming message consumes it, an ACK will be sent to the sender side to indicate that the message was saved successfully. If any message in the future has this problem too, the RNR retry flow will be done again for this message as well, without any history that this was done before. If even after retrying several times the receiver side still didn't post a Receiver Request and an RNR NACK was sent to the sender for each sent message, a Work Completion with RNR Retry Error will be generated on the sender side. Figure 13-9 presents the RNR retry flow of retry that overcome a missing Receive Request in he receiver side.

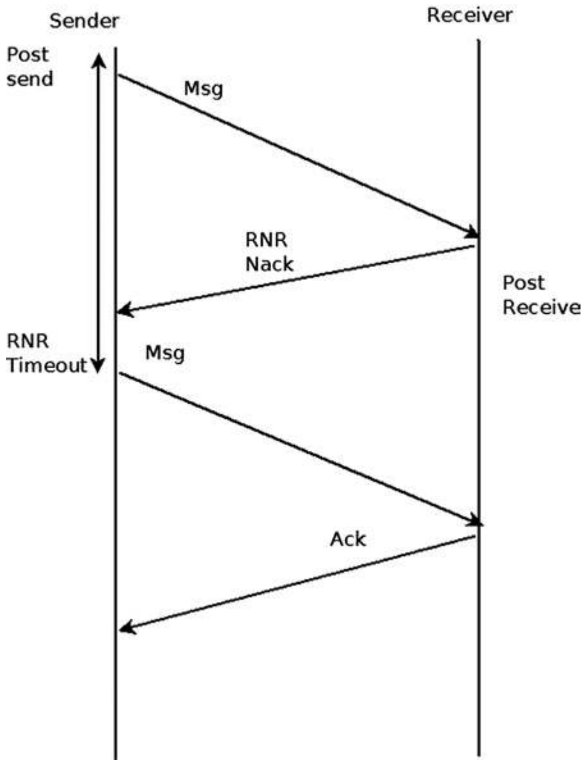

Figure 13-9. *RNR retry flow (on reliable transport types)*

In this section, I covered the Work Request status and some of the bad flows that can happen to a message. In the next section, I will discuss the multicast groups.

Multicast Groups

Multicast groups are a means to send a message from one UD QP to many UD QPs. Every UD QP that wants to get this message needs to be attached to the multicast group. When a device gets a multicast packet, it duplicates it to all of the QPs that are attached to that group. Following is a description of two methods related to multicast groups:

- The `ib_attach_mcast()` method attaches a UD QP to a multicast group within an InfiniBand device. It accepts the QP to be attached and the multicast group attributes.
- The `ib_detach_mcast()` method detaches a UD QP from a multicast group.

Difference Between the Userspace and the Kernel-Level RDMA API

The userspace and the kernel level of the RDMA stack API are quite similar, because they cover the same technology and need to be able to provide the same functionality. When the userspace is calling a method of the control path from the RDMA API, it performs a context switch to the kernel level to protect privileged resources and to synchronize objects that need to be synchronized (for example, the same QP number cannot be assigned to more than one QP at the same time).

However, there are some differences between the userspace and the kernel-level RDMA API and functionality:

- The prefix of all the APIs in the kernel level is "ib_", while in the userspace the prefix is "ibv_".
- There are enumerations and macros that exist only in the RDMA API in the kernel level.
- There are QP types that are available only in the kernel (for example, the SMI and GSI QPs).
- There are privileged operations that can be performed only in the kernel level—for example, registration of a physical memory, registration of an MR using a WR, and FMRs.
- Some functionality isn't available in the RDMA API in the userspace—for example, Request for N notification.
- The kernel API is asynchronous. There are callbacks that are called when there is an asynchronous event or Completion event. In the userspace, everything is synchronous and the user needs to explicitly check if there is an asynchronous event or Completion event in its running context (that is, thread).
- XRC isn't relevant for kernel-level clients.
- There are new features that were introduced to the kernel level, but they are not available (yet) in the userspace.

The userspace API is supplied by the userspace library "libibverbs." And although some of the RDMA functionality in the user level is less than the kernel-level one, it is enough to enjoy the benefits of the InfiniBand technology.

Summary

You have learned in this chapter about the advantages of the InfiniBand technology. I reviewed the RDMA stack organization. I discussed the resource-creation hierarchy and all of the important objects and their API, which is needed in order to write client code that uses InfiniBand. You also saw some examples that use this API. The next chapter will deal with advanced topics like network namespaces and the Bluetooth subsystem.

Quick Reference

I will conclude this chapter with a short list of important methods of the RDMA API. Some of them were mentioned in this chapter.

Methods

Here are the methods.

int ib_register_client(struct ib_client *client);

Register a kernel client that wants to use the RDMA stack.

void ib_unregister_client(struct ib_client *client);

Unregister a kernel client that wants to stop using the RDMA stack.

void ib_set_client_data(struct ib_device *device, struct ib_client *client, void *data);

Set a client context to be associated with an InfiniBand device.

void *ib_get_client_data(struct ib_device *device, struct ib_client *client);

Read the client context that was associated with an InfiniBand device.

int ib_register_event_handler(struct ib_event_handler *event_handler);

Register a callback to be called for every asynchronous event that occurs to the InfiniBand device.

int ib_unregister_event_handler(struct ib_event_handler *event_handler);

Unregister a callback to be called for every asynchronous event that occurs to the InfiniBand device.

int ib_query_device(struct ib_device *device, struct ib_device_attr *device_attr);

Query an InfiniBand device for its attributes.

int ib_query_port(struct ib_device *device, u8 port_num, struct ib_port_attr *port_attr);

Query an InfiniBand device port for its attributes.

enum rdma_link_layer rdma_port_get_link_layer(struct ib_device *device, u8 port_num);

Query for the link layer of the InfiniBand device's port.

int ib_query_gid(struct ib_device *device, u8 port_num, int index, union ib_gid *gid);

Query for the GID in a specific index in the InfiniBand device's port GID table.

int ib_query_pkey(struct ib_device *device, u8 port_num, u16 index, u16 *pkey);

Query for the P_Key-specific index in the InfiniBand device's port P_Key table.

int ib_find_gid(struct ib_device *device, union ib_gid *gid, u8 *port_num, u16 *index);

Find the index of a specific GID value in the InfiniBand device's port GID table.

int ib_find_pkey(struct ib_device *device, u8 port_num, u16 pkey, u16 *index);

Find the index of a specific P_Key value in the InfiniBand device's port P_Key table.

struct ib_pd *ib_alloc_pd(struct ib_device *device);

Allocate a PD to be used later to create other InfiniBand resources.

int ib_dealloc_pd(struct ib_pd *pd);

Deallocate a PD.

struct ib_ah *ib_create_ah(struct ib_pd *pd, struct ib_ah_attr *ah_attr);

Create an AH that will be used when posting a Send Request in a UD QP.

int ib_init_ah_from_wc(struct ib_device *device, u8 port_num, struct ib_wc *wc, struct ib_grh *grh, struct ib_ah_attr *ah_attr);

Initializes an AH attribute from a Work Completion of a received message and a GRH buffer. Those AH attributes can be used when calling the `ib_create_ah()` method.

struct ib_ah *ib_create_ah_from_wc(struct ib_pd *pd, struct ib_wc *wc, struct ib_grh *grh, u8 port_num);

Create an AH from a Work Completion of a received message and a GRH buffer.

int ib_modify_ah(struct ib_ah *ah, struct ib_ah_attr *ah_attr);

Modify the attributes of an existing AH.

int ib_query_ah(struct ib_ah *ah, struct ib_ah_attr *ah_attr);

Query the attributes of an existing AH.

int ib_destroy_ah(struct ib_ah *ah);

Destroy an AH.

struct ib_mr *ib_get_dma_mr(struct ib_pd *pd, int mr_access_flags);

Return an MR system memory that is usable for DMA.

static inline int ib_dma_mapping_error(struct ib_device *dev, u64 dma_addr);

Check if the DMA memory points to an invalid address—that is, check whether the DMA mapping operation failed.

static inline u64 ib_dma_map_single(struct ib_device *dev, void *cpu_addr, size_t size, enum dma_data_direction direction);

Map a kernel virtual address to a DMA address.

static inline void ib_dma_unmap_single(struct ib_device *dev, u64 addr, size_t size, enum dma_data_direction direction);

Unmap a DMA mapping of a virtual address.

static inline u64 ib_dma_map_single_attrs(struct ib_device *dev, void *cpu_addr, size_t size, enum dma_data_direction direction, struct dma_attrs *attrs)

Map a kernel virtual memory to a DMA address according to DMA attributes.

static inline void ib_dma_unmap_single_attrs(struct ib_device *dev, u64 addr, size_t size, enum dma_data_direction direction, struct dma_attrs *attrs);

Unmap a DMA mapping of a virtual address that was mapped according to DMA attributes.

static inline u64 ib_dma_map_page(struct ib_device *dev, struct page *page, unsigned long offset, size_t size, enum dma_data_direction direction);

Maps a physical page to a DMA address.

static inline void ib_dma_unmap_page(struct ib_device *dev, u64 addr, size_t size, enum dma_data_direction direction);

Unmap a DMA mapping of a physical page.

static inline int ib_dma_map_sg(struct ib_device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction);

Map a scatter/gather list to a DMA address.

static inline void ib_dma_unmap_sg(struct ib_device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction);

Unmap a DMA mapping of a scatter/gather list.

static inline int ib_dma_map_sg_attrs(struct ib_device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction, struct dma_attrs *attrs);

Map a scatter/gather list to a DMA address according to DMA attributes.

static inline void ib_dma_unmap_sg_attrs(struct ib_device *dev, struct scatterlist *sg, int nents, enum dma_data_direction direction, struct dma_attrs *attrs);

Unmap a DMA mapping of a scatter/gather list according to DMA attributes.

static inline u64 ib_sg_dma_address(struct ib_device *dev, struct scatterlist *sg);

Return the address attribute of a scatter/gather entry.

static inline unsigned int ib_sg_dma_len(struct ib_device *dev, struct scatterlist *sg);

Return the length attribute of a scatter/gather entry.

static inline void ib_dma_sync_single_for_cpu(struct ib_device *dev, u64 addr, size_t size, enum dma_data_direction dir);

Transfer a DMA region ownership to the CPU. It should be called before the CPU accesses a DMA mapped region whose ownership was previously transferred to the device.

static inline void ib_dma_sync_single_for_device(struct ib_device *dev, u64 addr, size_t size, enum dma_data_direction dir);

Transfer a DMA region ownership to the device. It should be called before the device accesses a DMA mapped region whose ownership was previously transferred to the CPU.

static inline void *ib_dma_alloc_coherent(struct ib_device *dev, size_t size, u64 *dma_handle, gfp_t flag);

Allocate a memory block that can be accessed by the CPU, and map it for DMA.

static inline void ib_dma_free_coherent(struct ib_device *dev, size_t size, void *cpu_addr, u64 dma_handle);

Free a memory block that was allocated using `ib_dma_alloc_coherent()`.

struct ib_mr *ib_reg_phys_mr(struct ib_pd *pd, struct ib_phys_buf *phys_buf_array, int num_phys_buf, int mr_access_flags, u64 *iova_start);

Take a physical page list, and prepare it for being accessed by the InfiniBand device.

int ib_rereg_phys_mr(struct ib_mr *mr, int mr_rereg_mask, struct ib_pd *pd, struct ib_phys_buf *phys_buf_array, int num_phys_buf, int mr_access_flags, u64 *iova_start);

Change the attributes of an MR.

int ib_query_mr(struct ib_mr *mr, struct ib_mr_attr *mr_attr);

Query for the attributes of an MR.

int ib_dereg_mr(struct ib_mr *mr);

Deregister an MR.

struct ib_mw *ib_alloc_mw(struct ib_pd *pd, enum ib_mw_type type);

Allocate an MW. This MW will be used to allow remote access to an MR.

static inline int ib_bind_mw(struct ib_qp *qp, struct ib_mw *mw, struct ib_mw_bind *mw_bind);

Bind an MW to an MR to allow a remote access to local memory with specific permissions.

int ib_dealloc_mw(struct ib_mw *mw);

Deallocates an MW.

struct ib_cq *ib_create_cq(struct ib_device *device, ib_comp_handler comp_handler, void (*event_handler)(struct ib_event *, void *), void *cq_context, int cqe, int comp_vector);

Create a CQ. This CQ will be used to indicate the status of ended Work Requests for Send or Receive Queues.

int ib_resize_cq(struct ib_cq *cq, int cqe);

Change the number of entries in a CQ.

int ib_modify_cq(structib_cq *cq, u16 cq_count, u16 cq_period);

Modify the moderation attributes of a CQ. This method is used to decrease the number of interrupts of an InfiniBand device.

int ib_peek_cq(structib_cq *cq, intwc_cnt);

Return the number of available Work Completions in a CQ.

static inline int ib_req_notify_cq(struct ib_cq *cq, enum ib_cq_notify_flags flags);

Request a Completion notification event to be generated when the next Work Completion is added to the CQ.

static inline int ib_req_ncomp_notif(struct ib_cq *cq, int wc_cnt);

Request a Completion notification event to be generated when there is a specific number of Work Completions in a CQ.

static inline int ib_poll_cq(struct ib_cq *cq, int num_entries, struct ib_wc *wc);

Read and remove one or more Work Completions from a CQ. They are read in the order that they were added to the CQ.

struct ib_srq *ib_create_srq(struct ib_pd *pd, struct ib_srq_init_attr *srq_init_attr);

Create an SRQ that will be used as a shared Receive Queue for several QPs.

int ib_modify_srq(struct ib_srq *srq, struct ib_srq_attr *srq_attr, enum ib_srq_attr_mask srq_attr_mask);

Modify the attributes of an SRQ.

int ib_query_srq(struct ib_srq *srq, struct ib_srq_attr *srq_attr);

Query for the attributes of an SRQ. The SRQ limit value might be changed in subsequent calls to this method.

int ib_destroy_srq(struct ib_srq *srq);

Destroy an SRQ.

struct ib_qp *ib_create_qp(struct ib_pd *pd, struct ib_qp_init_attr *qp_init_attr);

Create a QP. Every new QP is assigned with a QP number that isn’t in use by other QPs at the same time.

int ib_modify_qp(struct ib_qp *qp, struct ib_qp_attr *qp_attr, int qp_attr_mask);

Modify the attributes of a QP, which includes Send and Receive Queue attributes and the QP state.

int ib_query_qp(struct ib_qp *qp, struct ib_qp_attr *qp_attr, int qp_attr_mask, struct ib_qp_init_attr *qp_init_attr);

Query for the attributes of a QP. Some of the attributes might be changed in subsequent calls to this method.

int ib_destroy_qp(struct ib_qp *qp);

Destroy a QP.

static inline int ib_post_srq_recv(struct ib_srq *srq, struct ib_recv_wr *recv_wr, struct ib_recv_wr **bad_recv_wr);

Adds a linked list of Receive Requests to an SRQ.

static inline int ib_post_recv(struct ib_qp *qp, struct ib_recv_wr *recv_wr, struct ib_recv_wr **bad_recv_wr);

Adds a linked list of Receive Requests to the Receive Queue of a QP.

static inline int ib_post_send(struct ib_qp *qp, struct ib_send_wr *send_wr, struct ib_send_wr **bad_send_wr);

Adds a linked list of Send Requests to the Send Queue of a QP.

int ib_attach_mcast(struct ib_qp *qp, union ib_gid *gid, u16 lid);

Attaches a UD QP to a multicast group.

int ib_detach_mcast(struct ib_qp *qp, union ib_gid *gid, u16 lid);

Detaches a UD QP from a multicast group.