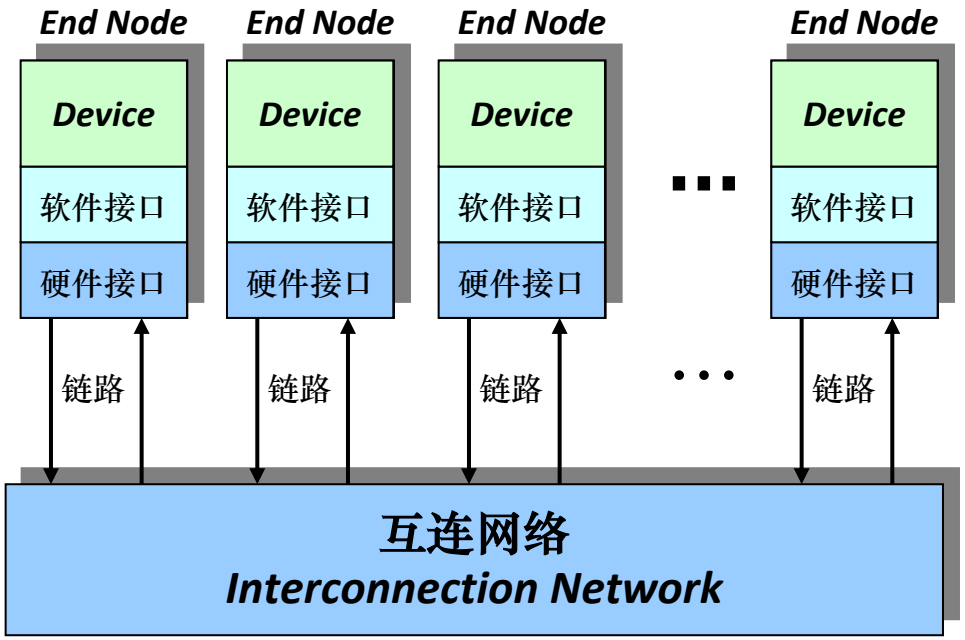
系统域互连网络

——Infiniband体系结构和Verbs编程

王展
wangzhan@ncic.ac.cn

课程回顾-I

- 互连网络将不同的装置连接到一起，并允许装置之间相互通信
  - 装置（*Device*）
    - 单个计算机内的组件
    - 单个计算机
    - 多机系统
  - 关联单元
    - 终端节点
      （*End Node = Device + Interface*）
    - 链路（*Link*）
    - 互连网络
      （*Interconnection Network*）



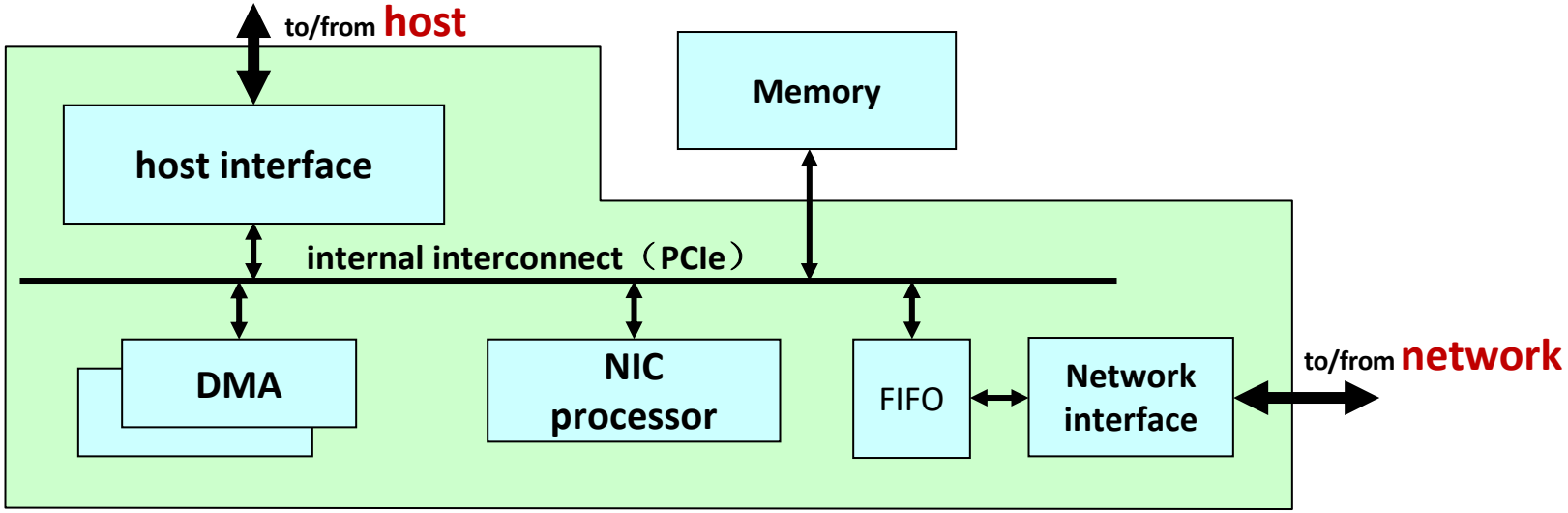
- 用尽可能少的消耗（时间、成本、功耗）传输尽可能多的信息

课程回顾-II

- **系统域互连网络影响并行系统的性能和扩展性**
  - 系统互连N个节点的代价是什么：延迟、带宽、功耗、成本等
- **上节课我们主要研究网络侧部分**
  - 拓扑、路由、交换、流控、链路
- **互连网络设计的第一步：网络仿真**

系统域网络接口

- 网络接口顾名思义，是主机节点和互连网络的桥接接口
  - 主机端
    - 与节点内其他硬件（CPU、内存）交互，一般通过PCIe总线；
    - 与节点上部署的软件（OS、应用）交互，Register、Buffer等；
  - 网络端
    - 网络包接收与发送，如组包、解包功能，流控功能等；
    - 网络连接状态管理，可靠网络流Request/Response，不可靠数据报等；



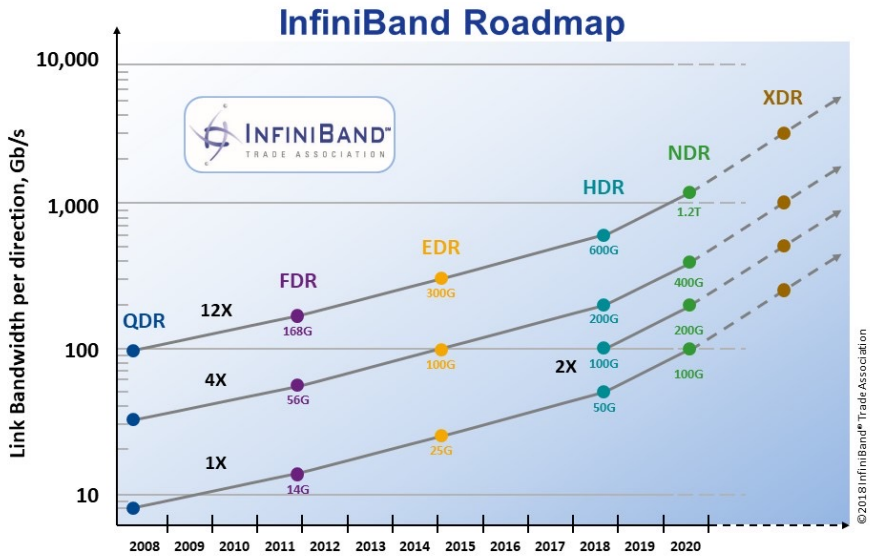
Infiniband体系结构

InfiniBand简介

- *InfiniBand*是由*IBTA(InfiniBand Trade Association)*制定的工业互连标准
  - 发起于1999年
- *InfiniBand*标准在制定之初试图涵盖服务器、通信设备、存储系统及嵌入式系统等多种互连体系结构（*SAN*和*LAN*）
- *InfiniBand*的典型特征是低延迟、高带宽，处理开销低、高可扩展、 同时支持在单个连接上传输多种流量类型(计算、存储、管理)
- 目前，*InfiniBand*主要应用于高性能计算集群和对性能要求较高的企业数据中心里

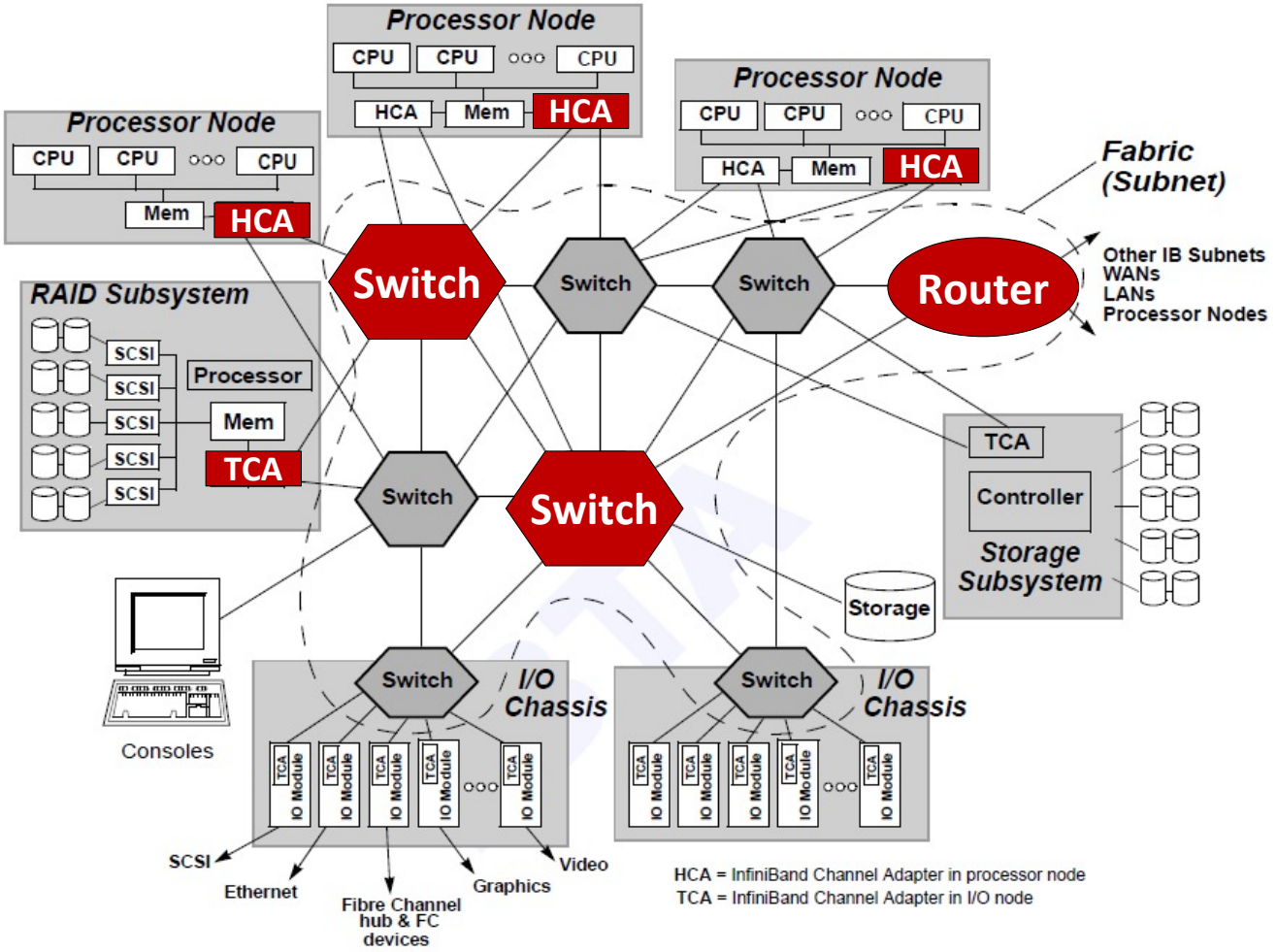
InfiniBand典型特征概述

- **高带宽**
  - 最高*12*条*NDR*串行链路并行
- **低延时**
  - 协议源生RDMA操作支持
  - 点对点应用通信延时*<1μs*
- **低开销**
  - 传输层以下全部硬件卸载
  - OS-Bypass用户态的通信调用
- **可扩展**
  - 基于交换的网络协议，节点间可并行路由转发
  - 每个子网最多支持*48k*节点，全网（多子网）最多支持2^{128}节点
  - 多样化的的网络拓扑支持：*Fat-tree*、*Dragonfly*、*Torus*等

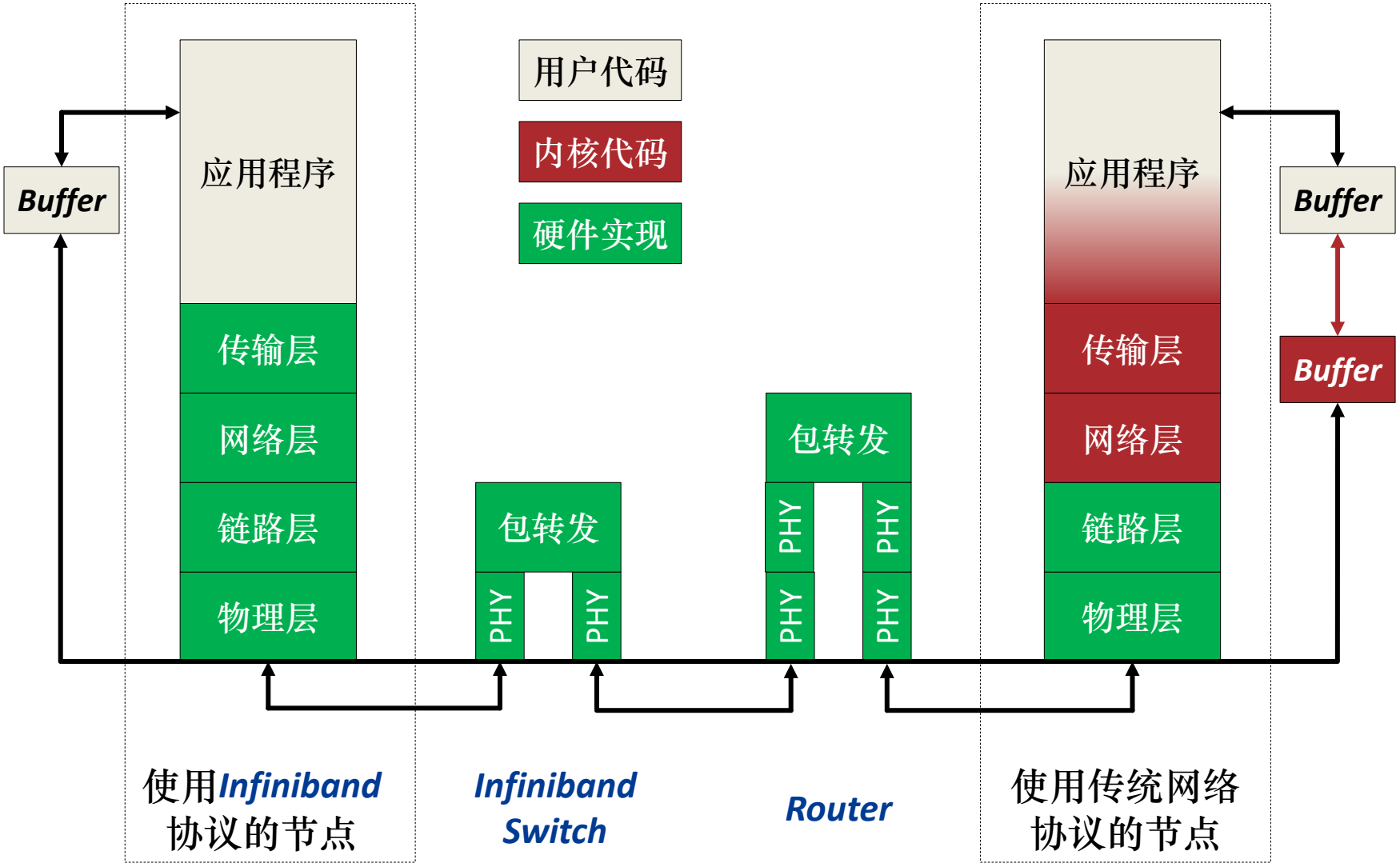


InfiniBand网络组件

- *HCA/TCA*
  - 终端接入
- *Switch*
  - 子网内交换
- *Router*
  - 跨子网交换



InfiniBand网络协议分层



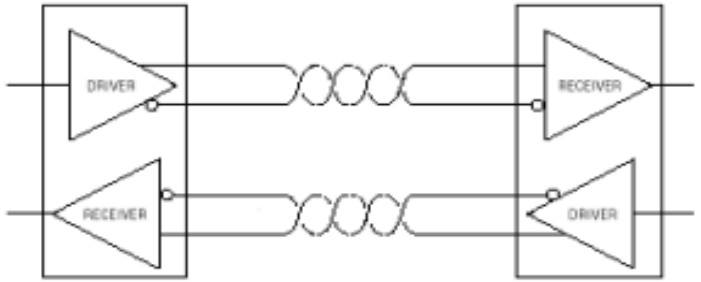
物理层简介

负责数据比特的发送和接受

- ***bit***
  - 定义***bit***流如何放在传输线上，如何组成***symbol***，以什么速率传输。（例如将链路层***8-bit***数据编码成***10-bit***数据流传输）
- ***symbol***
  - 经过信道编码和调制后的数据称为***symbol***“符号”
  - 定义***symbol***格式（如包头帧***symbol***格式、包尾帧***symbol***格式）、***symbol***对齐方式
- 错误处理
  - 检查编码错误、传输***skew***错误
- 连接处理
  - 物理链路训练，通知链路层物理链路状态

物理层信号

- ***InfiniBand*使用串行的比特流进行数据传输**
- ***Link Width***
  - ***1x*** - 每个***TX/RX 1***组差分对
  - ***4x*** - 每个***TX/RX 4***组差分对
  - ***12x*** – 每个***TX/RX 12***组差分对



- ***Link Speed***
  - *Single Data Rate (SDR) - 2.5Gb/s per lane (10Gb/s for 4x)*
  - *Double Data Rate (DDR) - 5Gb/s per lane (20Gb/s for 4x)*
  - *Quad Data Rate (QDR) - 10Gb/s per lane (40Gb/s for 4x)*
  - *Fourteen Data Rate (FDR) - 14Gb/s per lane (56Gb/s for 4x)*
  - *Enhanced Data Rate (EDR) - 25Gb/s per lane (100Gb/s for 4x)*
  - *High Data Rate (HDR) – 50Gb/s per lane (200Gb/s for 4x)*
  - *Next Data Rate (NDR) – 100Gb/s per lane (400Gb/s for 4x)*

物理层线缆和编码

- 连接媒介类型
  - *PCB*: 几英寸
  - 无源铜缆（*Passive copper*）：*20m SDR, 10m DDR, 7m QDR, 3m FDR, 3m EDR, 3m HDR*
  - 光纤（*Fiber*）：*300m SDR, 150m DDR, 100/300m QDR*
- 链路编码
  - *SDR, DDR, QDR*： *8* 到*10 bit* 编码
  - *FDR, EDR, HDR*: *64* 到 *66 bit* 编码
- 工业标准组件
  - 铜线-连接器
  - 光缆-光模块
  - 背板-连接器

PCB

copper

fiber

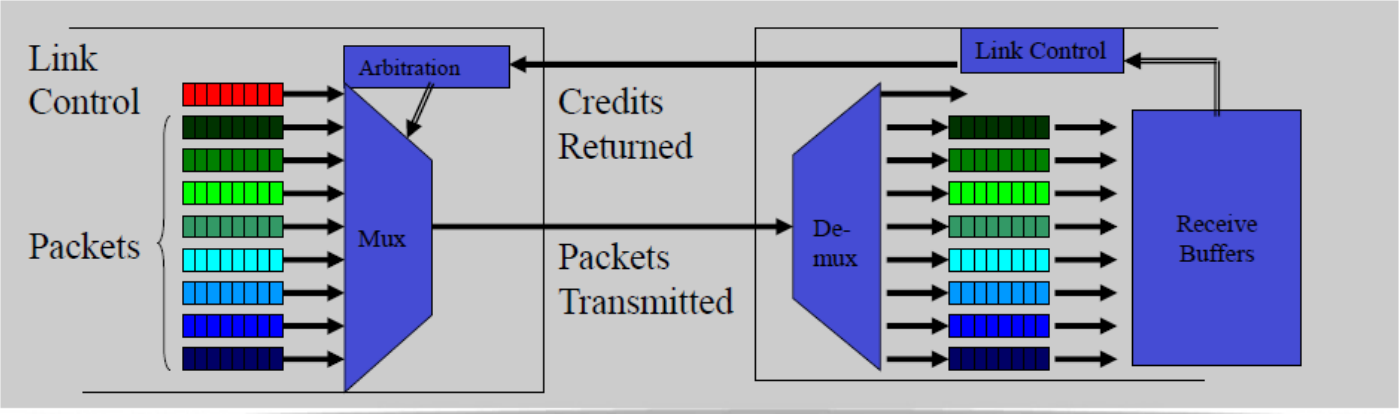
链路层简介

负责处理*packet*级的数据发送和接收

- 寻址
  - 发送：添加*SLID（Source LID）*信息
  - 接收：对比*DLID（Destination LID）*信息
- 缓冲和*QoS*
  - 端口的发送、接收缓冲区（与虚通道*VL*对应，每个端口至少2个*VL*）
  - 每个包携带*Service Level（SL）*，将*SL*与*VL*映射
- 流控：对虚通道接收的数据进行链路层的流控
- 错误检查和校验
- 包交换
  - 在交换机中，链路层根据*DLID*域查找转发表，决定包交换到那个端口发出

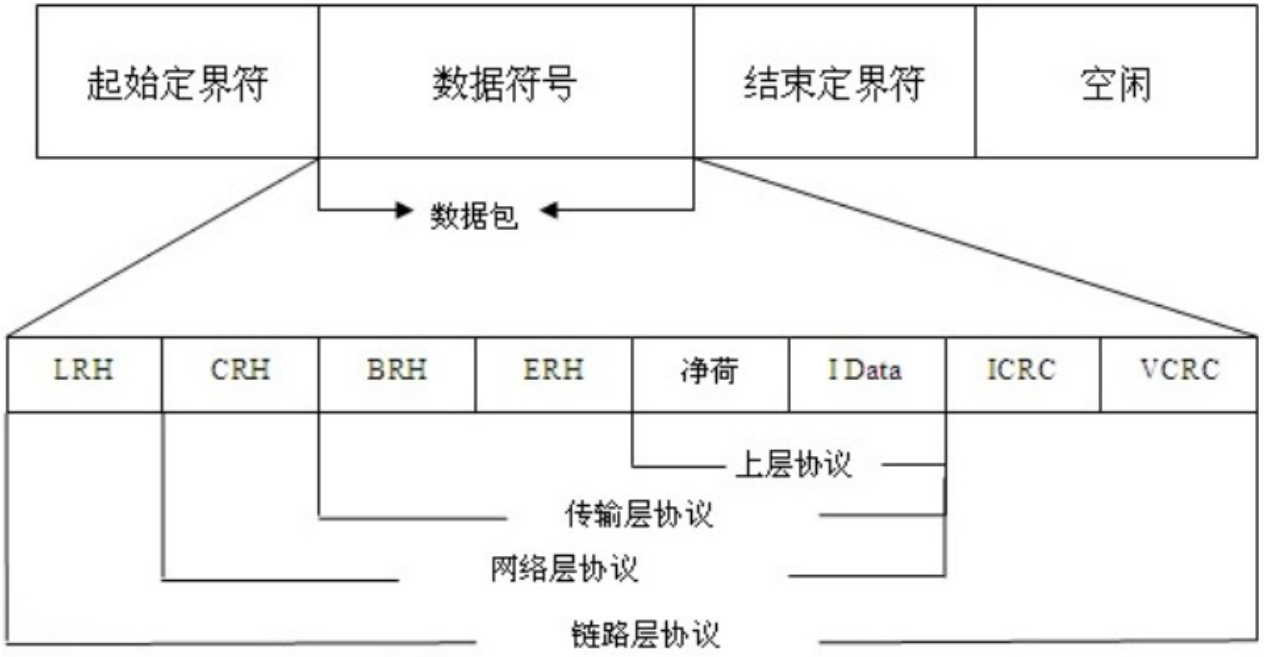
链路层QoS和流控

- 每个虚拟通道分开进行流量控制
  - 缓解排头阻塞（*Head-of-Line blocking*）
  - 虚通道：一个虚通道（*VL*）上的拥塞和延迟不会影响另一个虚通道的*QoS*保证，即使这两个虚通道共享同一个物理链路
- 基于信用的链路级流量控制
  - 确保在网络拥塞的状况下依然不会有丢包
  - 链路接收器为每个虚拟通道（*Virtual Lane*）授予接收缓冲空间的信用（*credit*）
  - 流量控制的信用以*64bit*为单元进行分发



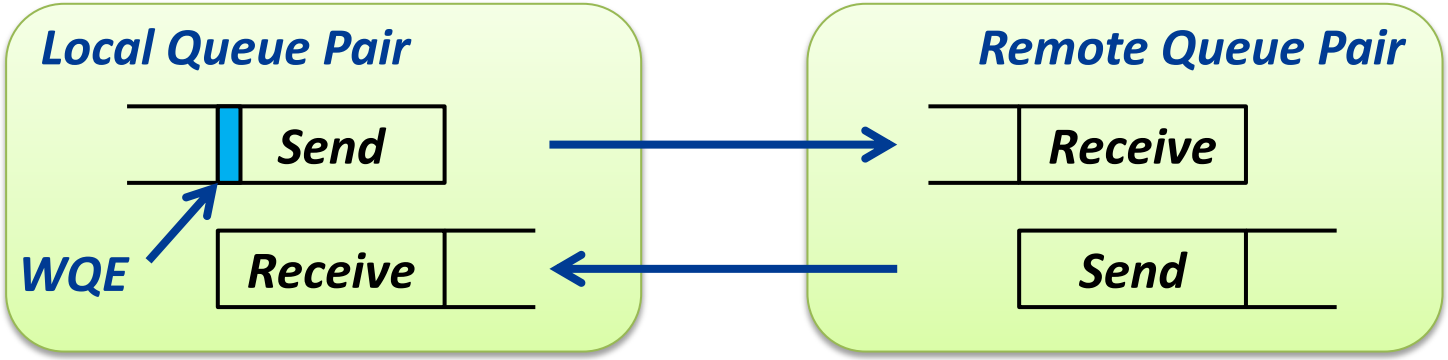
网络层和传输层简介

- 网络层提供子网间转发数据包的功能，类似于IP网络中的网络层。实现子网间的数据路由，数据在子网内传输时不需网络层的参与
- 传输层负责应用对接，提供基本传输服务，将应用消息切割成合理大小分段发送、接收和重组。

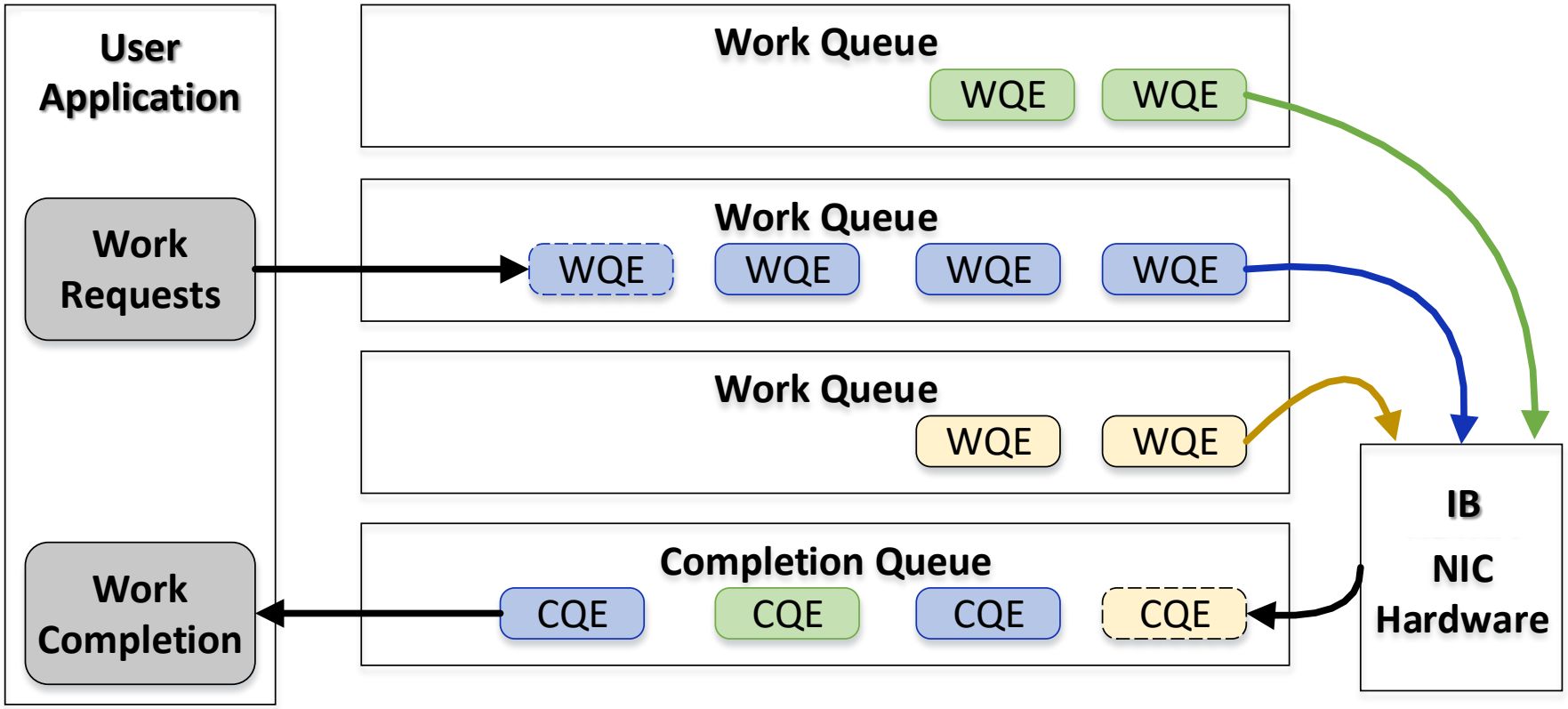


传输层用户接口

- ***WQ (Work Queue)***：工作队列，应用提交请求的接口
- ***QP (Queue Pair)***：工作队列通常成对使用 ***(Send/Receive)***
- ***WQE (Work Queue Element)***：封装用户请求并写入工作队列
- ***CQ (Completion Queue)***：请求执行完后的通知写回队列
- ***CQE (Completion Queue Element)***：封装完成通知



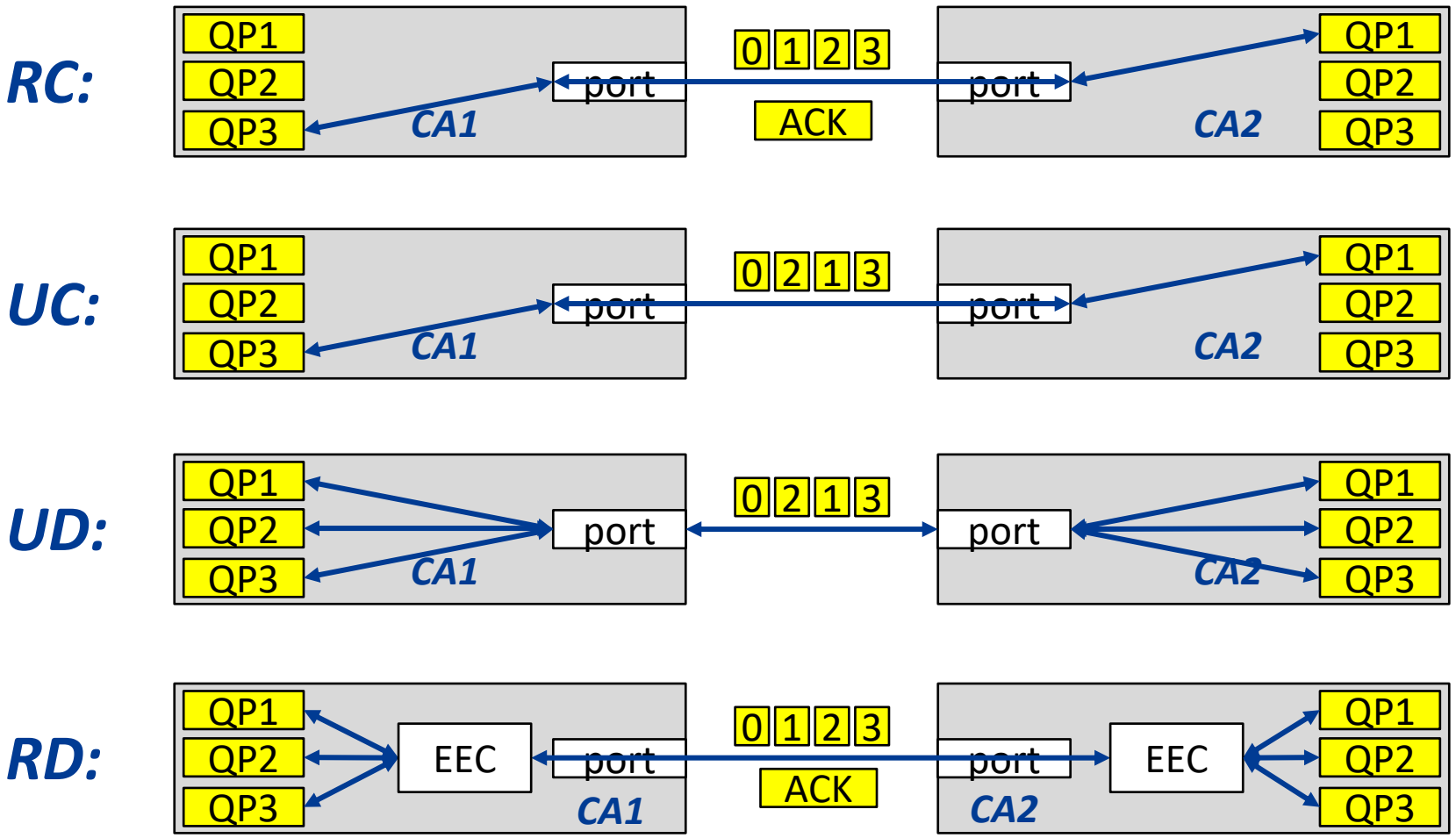
用户接口工作流程


User Application
Work Requests
Work Completion
Work Queue
WQE
WQE
Work Queue
WQE
WQE
WQE
WQE
Work Queue
WQE
WQE
Completion Queue
CQE
CQE
CQE
CQE
IB
NIC
Hardware

传输层支持的传输模式

- *RC*（*Reliable Connection*）
  - *QP*之间一对一建立连接，需要*ACK/NAK*响应和重传，类比*TCP*
  - 每个消息最大为*2GB*
- *UC*（*Unreliable Connection*）
  - *QP*之间一对一建立连接，无需响应和重传
  - 每个消息最大为*2GB*
- *UD*（*Unreliable Datagram*）
  - *QP*之间可以一对多通信，无需响应和重传，类比*UDP*
  - 每个消息长度不超过一个包的负载（适用于小消息，如管理信息）
- *RD*（*Reliable Datagram*）
  - *QP*之间可以一对多通信，需要 *ACK/NAK*响应和重传
  - 每个消息最大为*2GB*

传输模式图解



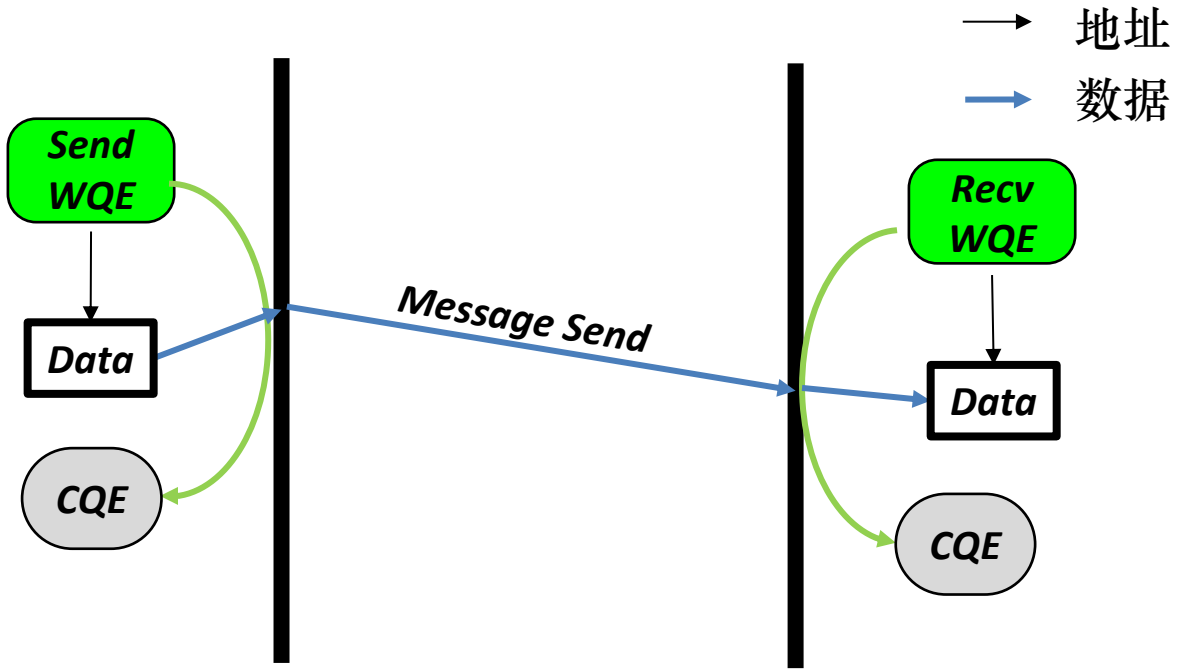
EEC: End-to-End Context

传输层支持的操作模式/通信语义

- 双边操作
  - *Send/Receive*
- 单边操作
  - *Read (RDMA)*
  - *Write (RDMA)*
  - *Atomic*

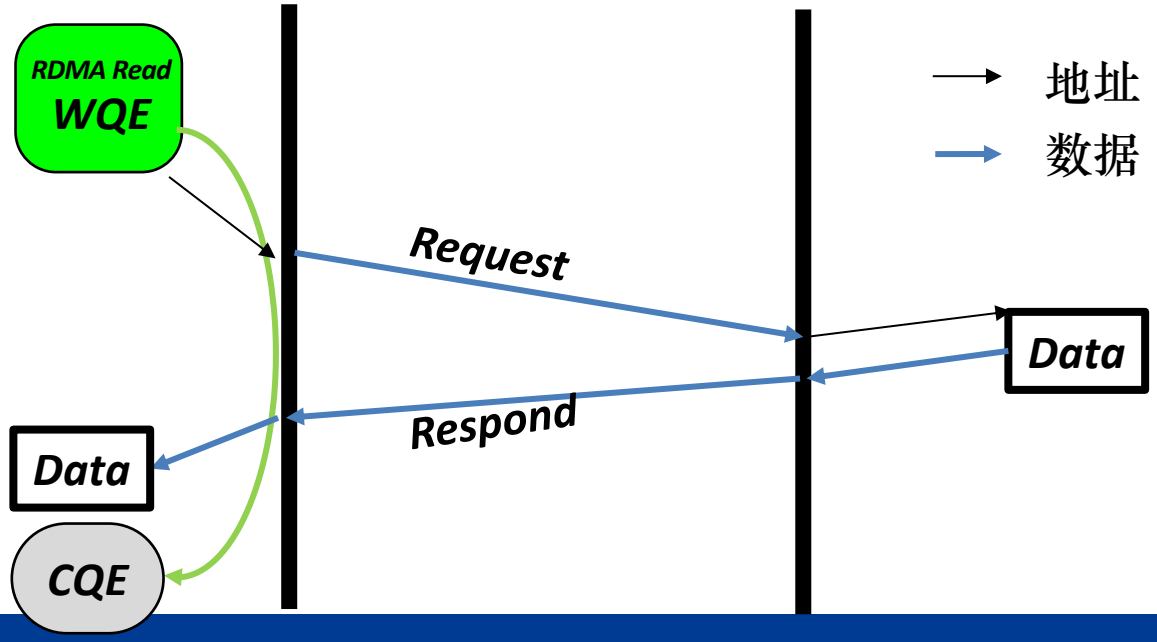
双边语义—*Send/Receive*

- 通信流程
  - 接收端*Post Receive Request (RR)*
  - 发送端*Post Send Request (SR)*
  - 接收端、发送端*Poll CQ*，轮询完成队列
  - 可靠连接需要*ACK/NAK*响应



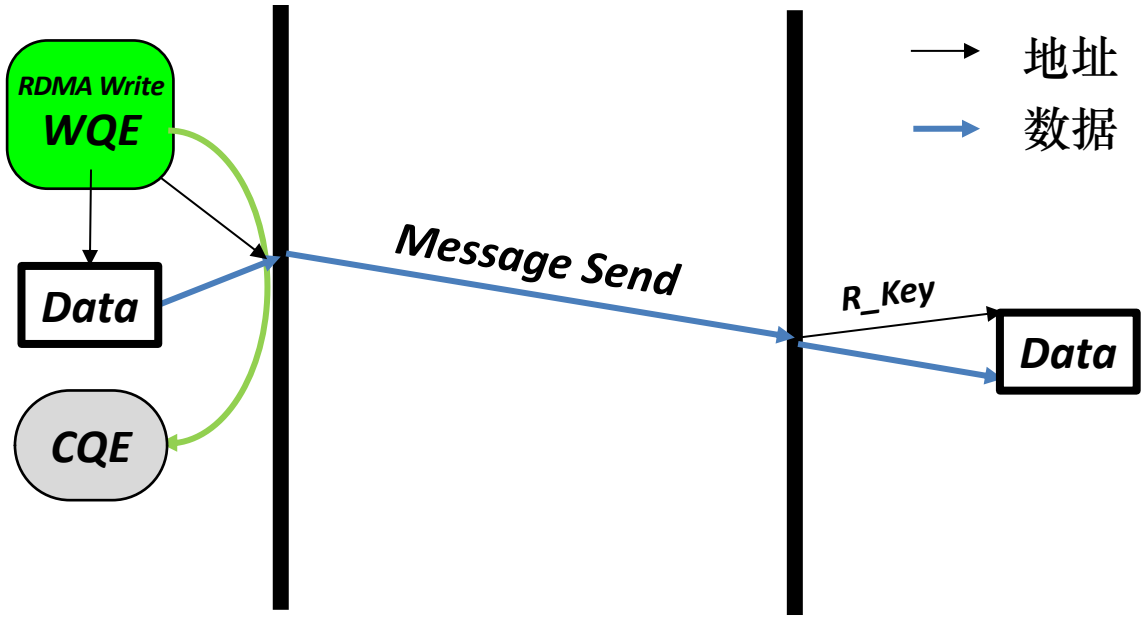
单边语义—*Read*

- 通信流程
  - 发起端*Post Send Request (SR)*
  - *SR*包含要读数据的远端地址信息、远端内存访问权限
  - 响应端返回数据
  - 发起端*Poll CQ*，轮询完成队列
  - *Read*操作仅存在于可靠连接模式



单边语义—Write

- 通信流程
  - 发起端*Post Send Request (SR)*
  - *SR*包含要写的数据、远端地址信息、远端内存访问权限
  - 发起端*Poll CQ*，轮询完成队列
  - 可靠连接需要*ACK/NAK*响应



单边语义—*Atomic*

- 通信流程
  - 发起端*Post Send Request (SR)*
  - *SR*包含原子操作需要的数据、远端地址信息、远端内存访问权限
  - 响应端执行操作，并返回原始数据
  - 发起端*Poll CQ*，轮询完成队列
  - *ATOMIC*操作仅存在于可靠连接模式

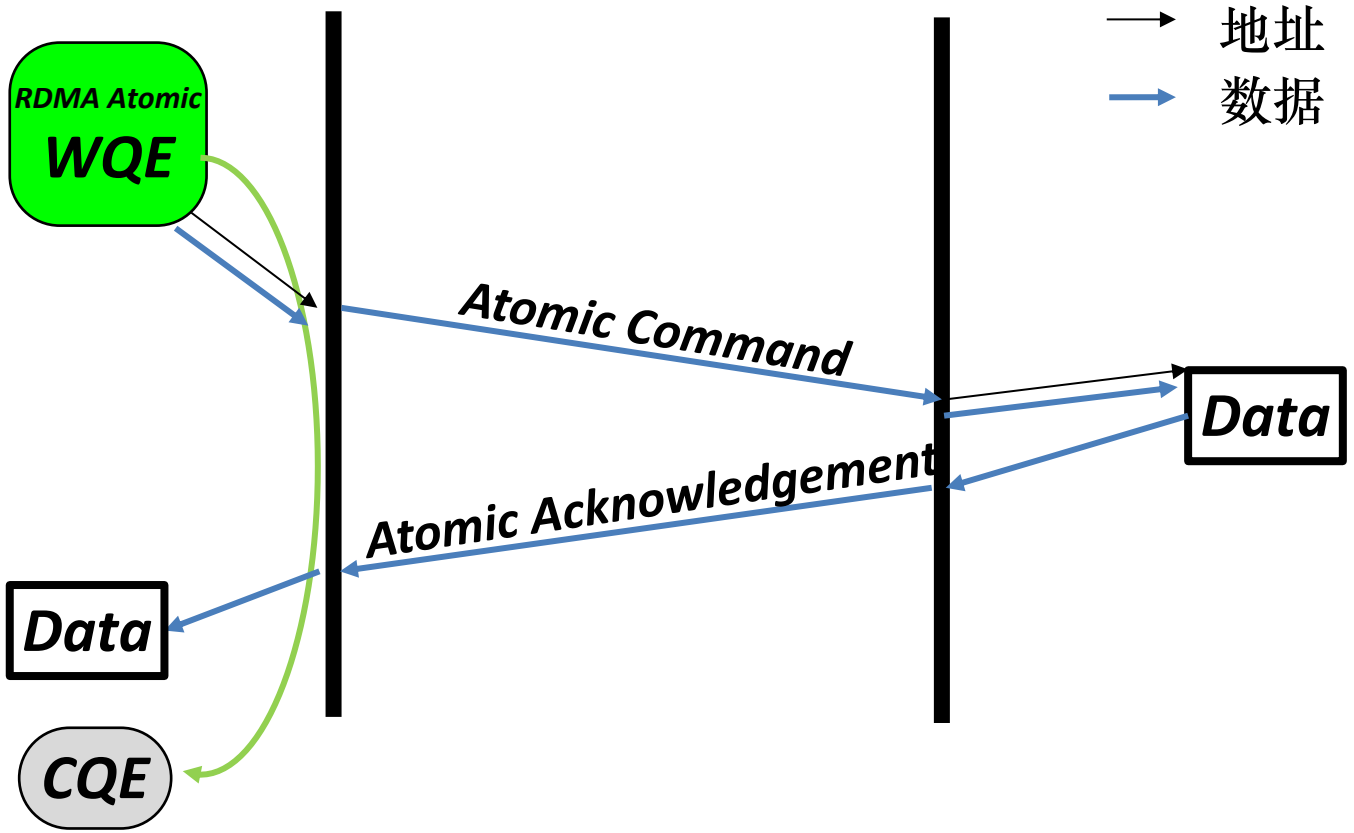
Fetch and Add

- 响应端返回原始数据
- 将原数据加上发送的数据更新响应端数据

Compare and Swap

- 响应端返回原始数据
- 将原数据与发送的数据比较，若相等，将发送数据写入，否则不写入

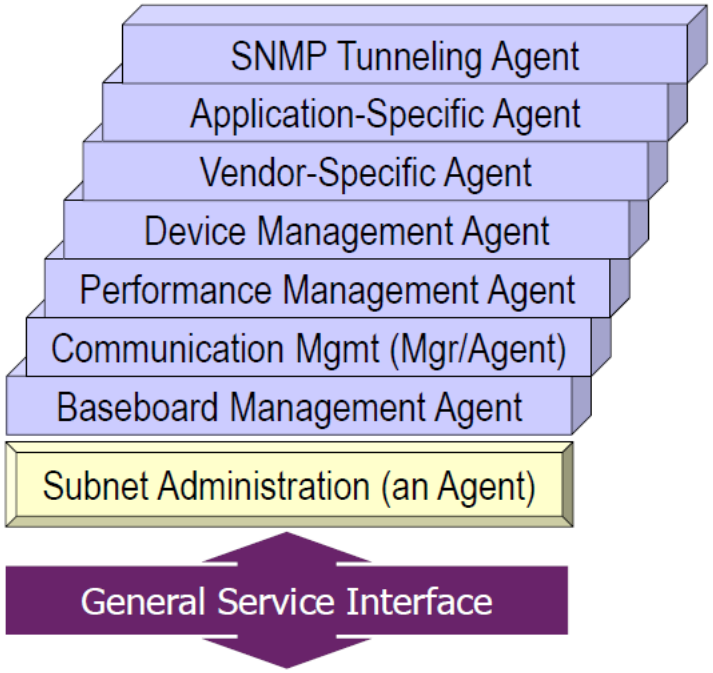
Atomic语义图解


地址
数据
RDMA Atomic
WQE
Atomic Command
Data
Atomic Acknowledgement
Data
CQE

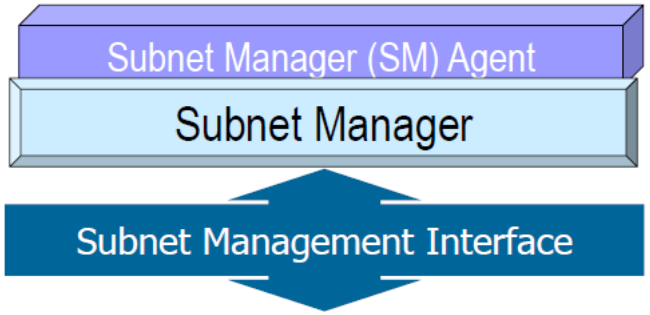
*Infiniband*网络管理

- IBA Management 定义了一套通用的管理架构
- 子网管理（Subnet Management，SM）
  - 为子网管理器提供发现和配置设备的方法
  - 网络管理
- 一般管理服务
  - Subnet Administration – 为节点提供子网管理（SM）收集的信息
  - 为节点提供注册器，以注册它们提供的一般服务
  - 在终端节点间建立通信和连接管理
  - 性能管理
    - 监视和报告明确定义的性能计数器
  - 其它。。。

管理模型



- 使用QP1进行数据传输
- 不能使用VL15
- MAD被称为GMP，使用LID路由
- 服从流量控制策略



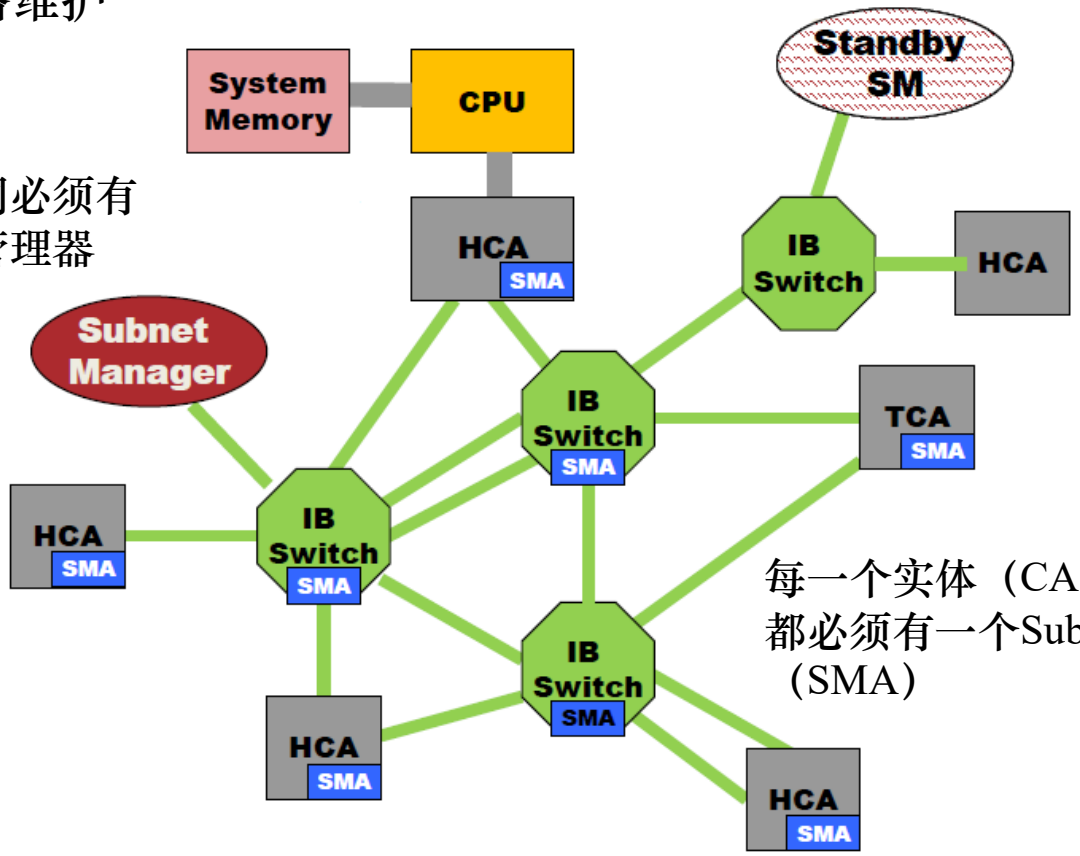
- 使用QP0进行数据传输
- 只能使用VL15
- MAD被称为SMP，使用LID路由或直接路由
- 没有流量控制

注：MAD为Management Datagram；GMP为General Management Packet；SMP为Subnet Management Packet

子网管理

- 拓扑发现
- 网络维护

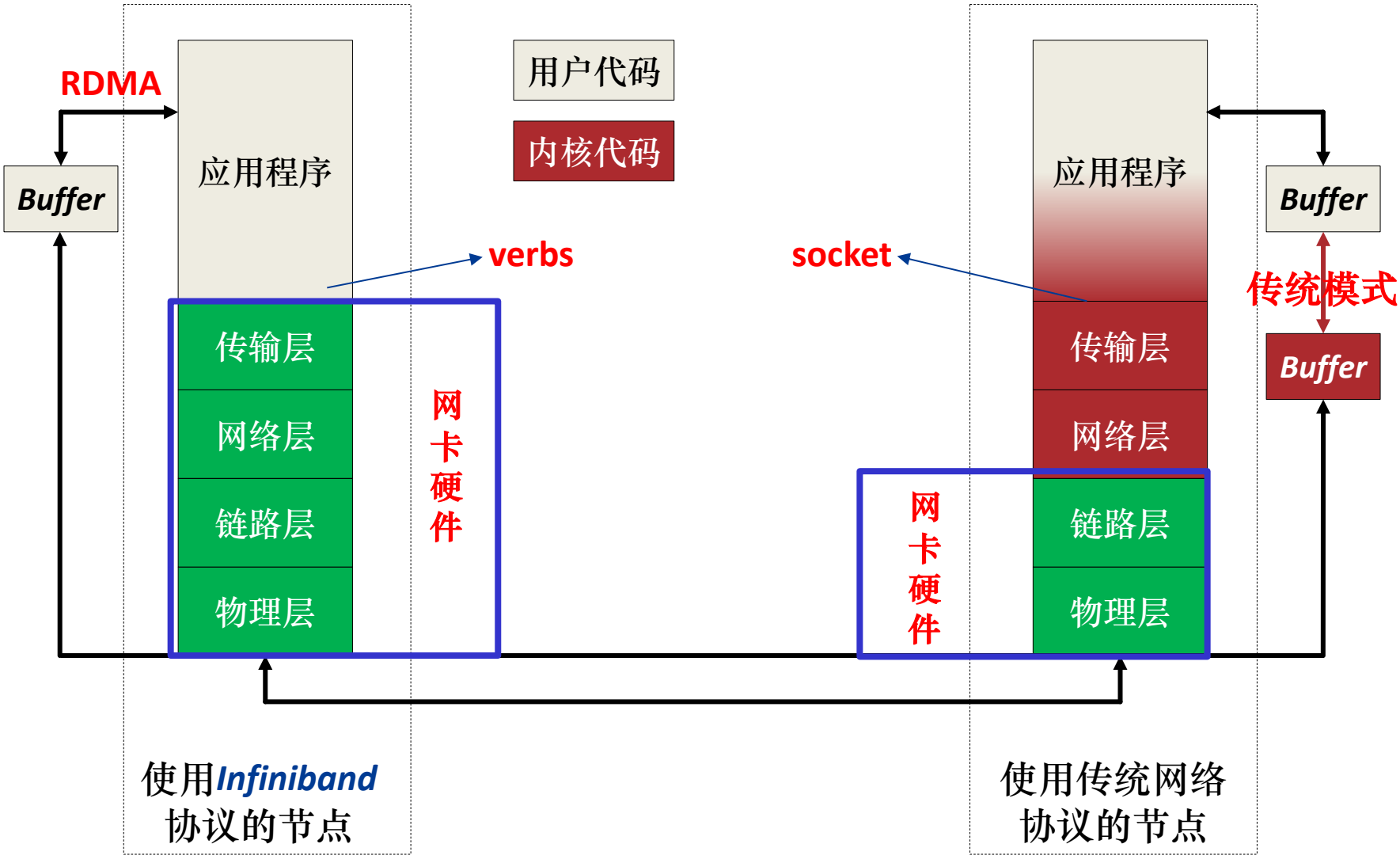
每一个子网必须有
一个子网管理器



每一个实体（CA，路由器，交换机）
都必须有一个Subnet Management Agent
（SMA）

Verbs编程

InfiniBand协议使用verbs接口为用户提供RDMA编程接口



什么是Verbs（1/2）

- Verbs最初是为高性能RDMA通信提供的抽象描述

- 控制路径（***Control Path***）：用于各种资源管理
  - 创建（Create）
  - 释放（Destroy）
  - 修改（Modify）
  - 请求（Query）
  - 事件处理（Events）
- 数据路径（***Data Path***）：用于完成数据的发送和接收
  - 发送请求（Post Send）
  - 接收请求（Post Receive）
  - 轮训完成队列（Poll CQ）
  - 查看完成事件（Request for Completion event）

什么是Verbs（2/2）

- Verbs软硬件要求
  - 软件
    - Linux OS
    - RDMA协议栈（如MLNX-OFED）
  - 硬件
    - 支持RDMA（Infiniband/RoCE）的网络设备
    - Loopback或者交换机连接
- Verbs特性
  - 性能：数据路径旁路OS，减少数据拷贝，性能更高
    - 低时延、高带宽
  - 应用
    - 存储（如NVMoF）
    - 并行计算

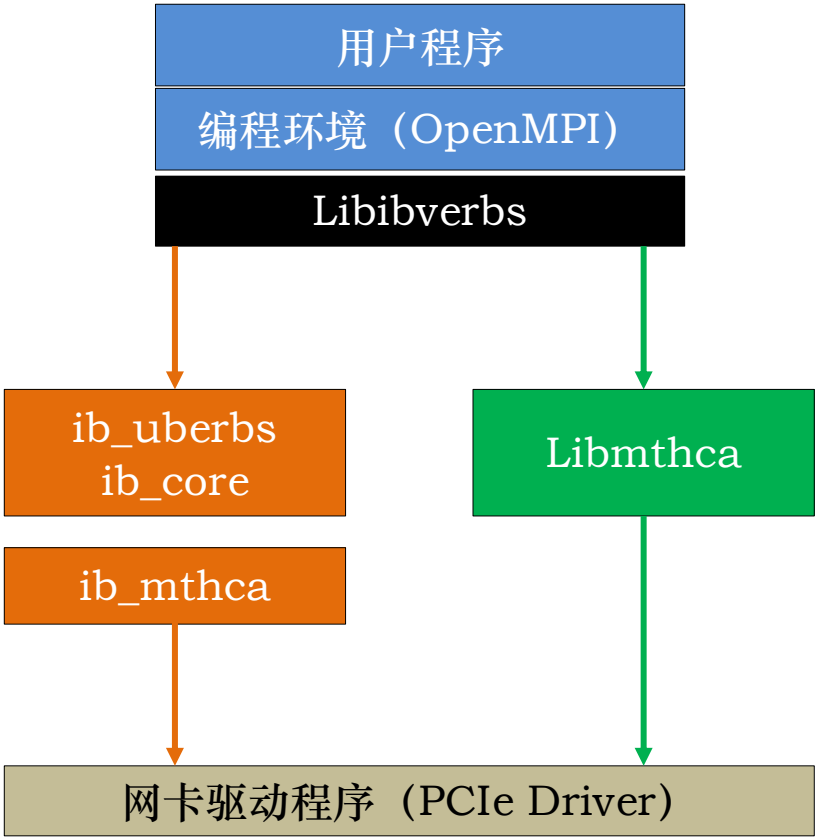
什么是libibverbs

- libibverbs是标准化的verbs API

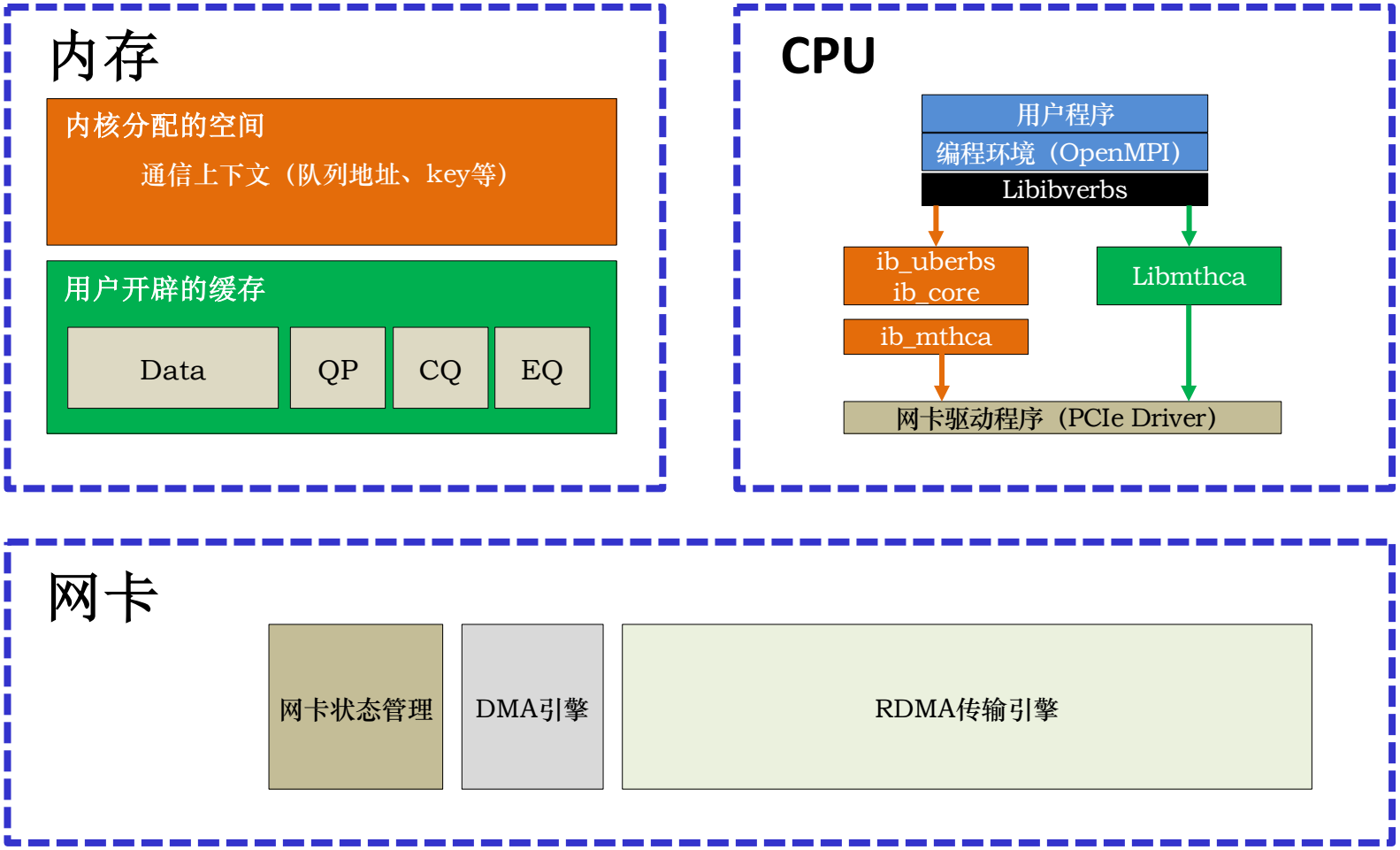
- 生态：业内广泛认可的标准RDMA通信编程接口
  - libibverbs开源
  - 自2005年集成在Linux 内核中（v2.6.11及更新版本）
  - 众多硬件厂家提供兼容的底层硬件相关的驱动

- 协议支持：支持所有的RDMA及变种
  - Infiniband，纯RDMA通信，硬件上，支持Infiniband的NIC和switch。
  - RoCE、RoCEv2(RDMA over Converged Ethernet)，通过以太网执行RDMA通信，硬件上，支持RoCE的NIC和标准Ethernet switch。
  - iWARP(Internet Wide Area RDMA Protocol)，通过TCP执行RDMA通信，硬件上，支持iWARP的NIC和标准Ethernet switch。

Libibverbs在软件栈中所处的位置



Verbs接口与QP以及硬件之间的关系



为什么要了解verbs编程

- **理解高性能互连网络工作的细节**
  - 例子：曙光7000的大规模并行程序优化，Infiniband HDR 200G网络
- **编写更适合应用的通信中间件**
  - 例子：阿里、AWS等使用verbs编写RPC库
- **设计更适合应用的通信原语（新的verbs）和互连网络**
  - 例子：AWS自研的互连网络接口卡

libibverbs API

- 建议、说明

- 源文件include verbs头文件
  - #include <infiniband/verbs.h>
- 输入的结构体初始化为0
  - 建议使用memset()初始化结构体
- 许多资源通过指针管理
  - 建议注意指针的使用，避免使用错误指针导致错误
- API返回值说明
  - 返回值为指针的API
    - 非空表示成功；NULL表示失败
  - 返回值为整型的API
    - 0表示成功；-1或者其他数值表示失败

参考文献：Mellanox Tec，RDMA Aware Networks Programming User Manual Rev 1.7

Libibverbs编程流程总述（1/2）

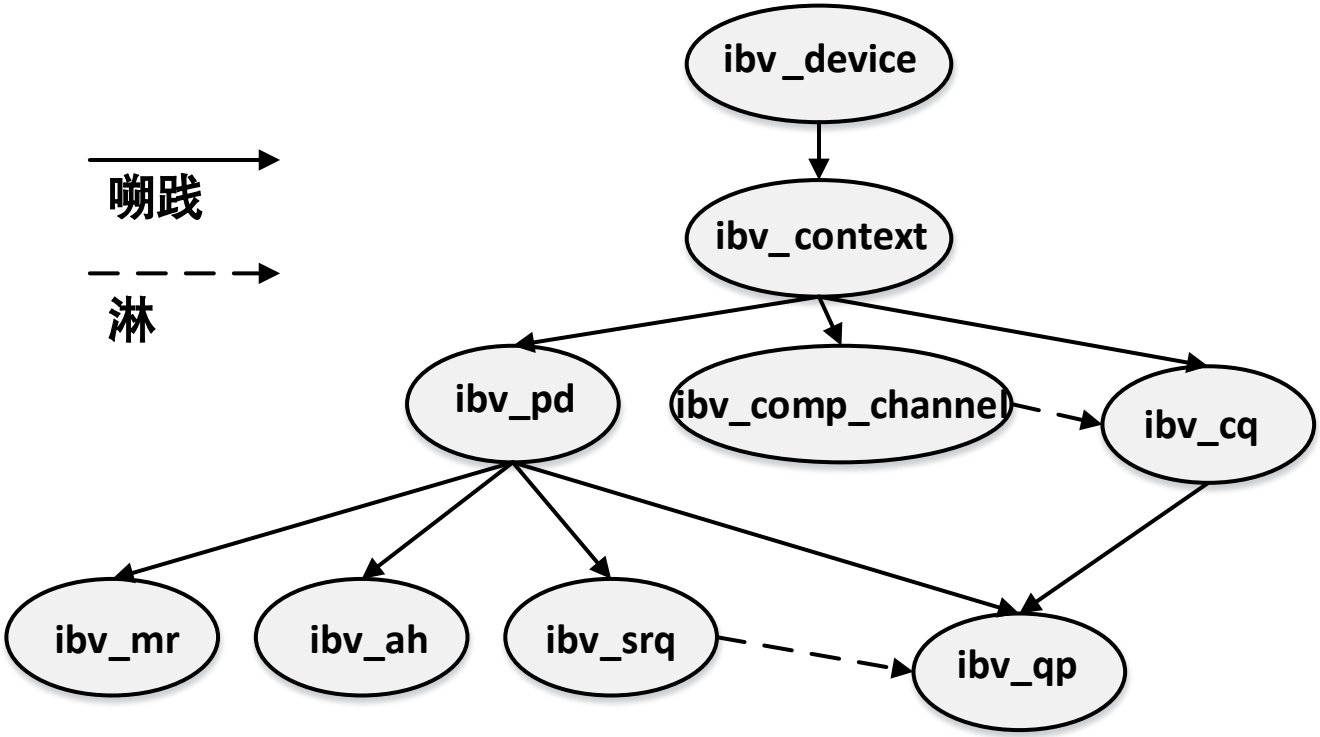
- 资源初始化
  - 调用API获取/创建以下资源：ibv_device、ibv_context、ibv_comp_channel、ibv_qp、ibv_pd、ibv_mr、ibv_ah、ibv_cq
- 交换连接信息
  - 通信两端交互连接信息，包括：QP号（qpn）、内存访问键（rkey，单边操作时需要）、ID信息（LID即Local ID、GID即Global ID）、序列号（psn）
- 建立连接
  - 调用ibv_modify_qp，连接状态更改顺序为INIT（初始化）>RTR（Ready-To-Receive）>RTS（Ready-To-Send）
- 数据传输
  - 调用ibv_post_send、ibv_post_recv完成数据传输

Libibverbs编程流程总述（2/2）

- 查询传输结果
  - ibv_poll_cq，查询对应CQ，获得数据传输结果
  - 用户可以通过ibv_get_cq_event、 ibv_ack_cq_event查询完成事件
- 释放资源
  - 通信完成时，调用含有destroy、dealloc等字样的API释放资源

资源初始化

- 调用API获取/创建以下资源：ibv_device、ibv_context、ibv_comp_channel、ibv_qp、 ibv_srq、 ibv_pd、ibv_mr、ibv_ah、ibv_cq



创建资源结构示意图

资源初始化——ibv_device

- 获取ibv_device
  - API：struct ibv_device ** ibv_get_device_list(int *num_devices)、const char *ibv_get_device_name (struct ibv_device *device)
  - 示例

```
//ibv_device声明
struct ibv_device **dev_list;
struct ibv_device *ib_dev;
……
dev_list = ibv_get_device_list(NULL); //获取ibv_device设备列表
if (!dev_list)
   ……//错误处理

//若用户传入设备名称，则从设备列表中查询到对应的设备返回
int i;
for (i = 0; dev_list[i]; ++i)
   if (!strcmp(ibv_get_device_name(dev_list[i]), ib_devname))
         break;
ib_dev = dev_list[i];
if (!ib_dev)
   ……//错误处理
```

资源初始化——ibv_context

- 获取ibv_context
  - API: struct ibv_context *ibv_open_device(struct ibv_device *device)
  - 用于获取设备的上下文信息
  - 示例

```
//ibv_context声明
struct ibv_context *context;
...

//获取ibv_context
context = ibv_open_device(ib_dev);
        if (! context) {
                ·····//错误处理
        }
```

资源初始化——ibv_comp_channel

- 创建ibv_comp_channel
  - API: struct ibv_comp_channel *ibv_create_comp_channel(struct ibv_context *context)
  - 当有CQE到达CQ时，Completion Channel（CC）用于提醒用户
  - 示例

```
// ibv_comp_channel声明
struct ibv_comp_channel *channel;
...

//创建ibv_comp_channel
channel = ibv_create_comp_channel(context);
        if (! channel) {
                ·····//错误处理
        }
```

注：当使用轮询机制处理CQ时，并不需要CC，用户仅需在一定时间内轮询CQ。当需要中断机制时，就需要CC，CC用于当CQ中有新的CQE到来时，提醒用户。同一个CC可以用在多个CQ上。

资源初始化——ibv_pd

- 创建ibv_pd
  - API: struct ibv_pd *ibv_alloc_pd(struct ibv_context *context)
  - 创建一个Protection Domain（PD）。PD用于保护QP可以访问的Memory Region（MR）。用户必须创建至少一个PD。
  - 示例

```
//ibv_pd声明
struct ibv_pd *pd;
...

//创建ibv_pd
pd = ibv_alloc_pd(context);
        if (! pd) {
                ·····//错误处理
        }
```

资源初始化——ibv_cq

- 创建ibv_cq
  - API: struct ibv_cq *ibv_create_cq(struct ibv_context *context, int cqe, void *cq_context, struct ibv_comp_channel *channel, int comp_vector)
  - 创建一个CQ，用于存放WR完成时产生的CQE
  - 参数说明：cqe 定义了CQ最小的容纳的CQE数；cq_context 是一个用户定义的值，若该值在CQ创建期间指定，那么在使用CC时，该值将作为ibv_get_cq_event接口中的参数返回；channel 用于指定一个CC；comp_vector 用于指定完成向量，该完成向量用于通知完成事件。
  - 示例

```
//ibv_cq声明
struct ibv_cq *cq;
...
//创建ibv_cq,深度为depth，与CC channel关联
cq = ibv_create_cq(context, depth, NULL, channel, 0);
        if (! cq){
                ·····//错误处理
        }
```

资源初始化——ibv_mr

- 创建ibv_mr
  - API: struct ibv_mr *ibv_reg_mr(struct ibv_pd *pd, void *addr, size_t length, enum ibv_access_flags access)
  - 创建一个与PD关联的Memory Region（MR），根据ibv_access_flags分配lkey、rkey。进行数据通信需要使用内存都需要通过该接口进行注册。
  - 参数说明：addr为内存基地址；ength为字节单位的MR长度；访问权限包含如下几种，常用的为本地写，及远端的读、写、原子操作

```
enum ibv_access_flags {
IBV_ACCESS_LOCAL_WRITE = 1,
IBV_ACCESS_REMOTE_WRITE = (1<<1),
IBV_ACCESS_REMOTE_READ = (1<<2),
IBV_ACCESS_REMOTE_ATOMIC = (1<<3),
IBV_ACCESS_MW_BIND = (1<<4),
IBV_ACCESS_ZERO_BASED = (1<<5),
IBV_ACCESS_ON_DEMAND = (1<<6};}
```

  - 示例

```
//ibv_mr声明
struct ibv_mr *mr;
...
//创建ibv_mr
mr = ibv_reg_mr(pd, buf, size, ibv_access_flags);
        if (! mr){·····//错误处理
        }
```

资源初始化——ibv_ah

- 创建ibv_ah
  - API: struct ibv_ah *ibv_create_ah(struct ibv_pd *pd, struct ibv_ah_attr *attr)
  - 创建一个与PD关联的Address Handler（AH），AH包含到达远程目的地所需的所有数据。
  - 参数说明：

```
struct ibv_ah_attr {
    struct ibv_global_route grh;/* 全局路由信息*/
    uint16_t dlid;/* 目的Local ID lid */
    uint8_t sl;/* 服务级别*/
    uint8_t src_path_bits;/* source path bits */
    uint8_t static_rate;
    uint8_t is_global;/* 全局地址，需要使用上面提到的grh*/
    uint8_t port_num;/* 目的端口号*/ };
```

  - 示例

```
//ibv_ah声明
struct ibv_ah *ah;
...
//创建ibv_ah
ah = ibv_create_ah(pd, &attr);
        if (! ah){·····//错误处理
        }
```

资源初始化——ibv_qp (1/2)

- 创建ibv_qp
  - API: struct ibv_qp *ibv_create_qp(struct ibv_pd *pd, struct ibv_qp_init_attr *qp_init_attr)
  - 根据qp_init_attr创建一个QP，当QP刚被创建时，它处于Reset 状态。
  - 参数说明：

```
struct ibv_qp_init_attr {
    void *qp_context;
    struct ibv_cq *send_cq;/* Send CQ */
    struct ibv_cq *recv_cq;/* Receive CQ. */
    struct ibv_srq *srq; /* (optional) 仅用于SRQ QP*/
    struct ibv_qp_cap cap;/*QP的WR容量、sg_list容量、in_line数据容量*/
    enum ibv_qp_type qp_type;
    /*QP类型，包括：IBV_QPT_RC,
                   IBV_QPT_UC,
                   IBV_QPT_UD,
                   IBV_QPT_XRC,
                   IBV_QPT_RAW_PACKET,
                   IBV_QPT_RAW_ETH*/
    int sq_sig_all;/*是否产生CQE标志位 */
    struct ibv_xrc_domain *xrc_domain;/*(Optional) 仅用于XRC*/
};
```

资源初始化——ibv_qp (2/2)

- 创建ibv_qp
  - 示例：创建RC类型的QP，发送、接收使用同一CQ

```
struct ibv_qp_init_attr init_attr = {
        .send_cq = cq,                  //发送、接收使用同一CQ
        .recv_cq = cq,
        .cap     = {
                .max_send_wr  = 1, //最多1个SR
                .max_recv_wr  = 1, //最多1个RR
                .max_send_sge = 1, //最多1个发送scatter gather entry
                .max_recv_sge = 1 //最多1个接收scatter gather entry
        },                          //sge用于描述数据分布，见下文
        .qp_type = IBV_QPT_RC       //RC类型QP
};
qp = ibv_create_qp(pd, &init_attr);
        if (! qp)  { ·····          //错误处理
        }
```

```
struct ibv_sge {
    uint64_t addr;/* buffer基地址 */
    uint32_t length;/* buffer长度 */
    uint32_t lkey;/* 对应的MR中的local key (lkey)*/
};
```

资源初始化——ibv_srq

- 创建ibv_srq
  - API: struct ibv_srq *ibv_create_srq(struct ibv_pd *pd, struct ibv_srq_init_attr *srq_init_attr)
  - 根据srq_init_attr创建一个SRQ（Shared Receive Queue），RQ可供多个QP共享
  - 参数说明:

```
struct ibv_srq_attr {
     uint32_t max_wr;//WR最大数量
     uint32_t max_sge;//sge最大数量
     uint32_t srq_limit; //创建时无需使用最多1个SR
     };
```

  - 示例

```
struct ibv_srq_init_attr attr = {
          .attr = {.max_wr  = depth,
                   .max_sge = 1 }
          };
srq = ibv_create_srq(pd, &attr);
          if (! srq)   { ·····        //错误处理
          }
```

交换连接信息

- 通信两端交互连接信息，包括：QP号（qpn）、内存访问键（rkey，单边操作时需要）、ID信息（LID即Local ID、GID即Global ID）、序列号（psn）
  - 两端要建立RDMA连接，需要知道以上信息，因此用户在建立RDMA连接之前，需要交换以上信息。

  - 在实际应用中可以使用多种方式完成信息交换：
    - 使用普通的socket连接交换信息
      集群中多数情况下含有Ethernet网，可供交换连接信息

    - 使用RDMA_CM（Connect Manager）交换信息
      关于CM的使用是以“rdma_” 为前缀的一系列API，在此不做介绍

建立连接(1/4)

- 调用ibv_modify_qp，连接状态更改顺序为RESET > INIT（初始化）> RTR（Ready-To-Receive）> RTS（Ready-To-Send）
  - API: int ibv_modify_qp(struct ibv_qp *qp, struct ibv_qp_attr *attr, enum ibv_qp_attr_mask attr_mask)
  - 根据attr和指定的attr_mask修改QP
  - 参数说明：

```
struct ibv_qp_attr {
    enum ibv_qp_state qp_state;
    enum ibv_qp_state cur_qp_state;
    enum ibv_mtu path_mtu;
    enum ibv_mig_state path_mig_state;
    uint32_t qkey;
    uint32_t rq_psn;
    uint32_t sq_psn;
    uint32_t dest_qp_num;
    int qp_access_flags;
    struct ibv_qp_cap cap;
    struct ibv_ah_attr ah_attr;
    struct ibv_ah_attr alt_ah_attr;
    uint16_t pkey_index;
```

接右边

```
    uint16_t alt_pkey_index;
    uint8_t en_sqd_async_notify;
    uint8_t sq_draining;
    uint8_t max_rd_atomic;
    uint8_t max_dest_rd_atomic;
    uint8_t min_rnr_timer;
    uint8_t port_num;
    uint8_t timeout;
    uint8_t retry_cnt;
    uint8_t rnr_retry;
    uint8_t alt_port_num;
    uint8_t alt_timeout;
};
```

建立连接(2/4)

- RESET > INIT
  - 由创建时的RESET转换为INIT 状态，此时，用户可以通过ibv_post_recv接口将接收缓冲区发送到RQ，RQ可以接收WR（但不能处理）。
  - 示例

```
struct ibv_qp_attr attr = {
         .qp_state        = IBV_QPS_INIT,
         .pkey_index      = 0,
         .port_num        = port,
         .qp_access_flags = 0
         };

if (ibv_modify_qp(ctx->qp, &attr, IBV_QP_STATE | IBV_QP_PKEY_INDEX |
IBV_QP_PORT | IBV_QP_ACCESS_FLAGS)) {
          .....   //错误处理
         }
```

建立连接(3/4)

- INIT > RTR
  - QP由INIT 状态转换到RTR 状态，此时，QP可以进行RQ WR的处理。
  - 示例

```
struct ibv_qp_attr attr = {
        .qp_state                 = IBV_QPS_RTR,
        .path_mtu                 = mtu,
        .dest_qp_num              = dest->qpn, /*qpn连接信息*/
        .rq_psn                   = dest->psn, /*psn连接信息*/
        .max_dest_rd_atomic       = 1, /*允许接收的最大请求数*/
        .min_rnr_timer            = 12,    /*最小的RNR Nak时间*/
        .ah_attr          = { .is_global    = 0,
                .dlid             = dest->lid, /*LID连接信息*/
                .sl               = sl,
                .src_path_bits    = 0,
                .port_num         = port /*对端端口号*/
                }
        };
if (ibv_modify_qp( qp, &attr, IBV_QP_STATE | IBV_QP_AV
| IBV_QP_PATH_MTU | IBV_QP_DEST_QPN | IBV_QP_RQ_PSN |
IBV_QP_MAX_DEST_RD_ATOMIC | IBV_QP_MIN_RNR_TIMER)) {
          .....   //错误处理
        }
```

建立连接(4/4)

- RTR > RTS
  - QP由RTR状态转换到RTS 状态，此时用户可以通过ibv_post_send接口提交发送请求。QP可以进行SQ WR的处理。
  - 示例

```
attr.qp_state         = IBV_QPS_RTS;
attr.timeout          = 14; /*超时限制*/
attr.retry_cnt        = 7; /*发送失败后重新尝试的次数*/
attr.rnr_retry        = 7; /*收到RNR响应时，重新尝试的次数*/
attr.sq_psn           = my_psn; /*psn连接信息，匹配远端的psn*/
attr.max_rd_atomic  = 1; /*允许的最大atomic操作请求数*/
if (ibv_modify_qp( qp, &attr,
                   IBV_QP_STATE              |
                   IBV_QP_TIMEOUT            |
                   IBV_QP_RETRY_CNT          |
                   IBV_QP_RNR_RETRY          |
                   IBV_QP_SQ_PSN             |
                   IBV_QP_MAX_QP_RD_ATOMIC)) {
                  .....   //错误处理
}
```

数据传输——接收（1/2）

- 接收请求处理
  - API: int ibv_post_recv(struct ibv_qp *qp, struct ibv_recv_wr *wr, struct ibv_recv_wr **bad_wr)
  - 在RQ上提交一个WR的链表，在第一个错误出现时，将停止对WR列表的处理，并在bad_wr中返回一个指向违规的WR的指针。
  - 参数说明：

```
struct ibv_recv_wr {
    uint64_t wr_id;/* 用户分配的WR ID */
    struct ibv_recv_wr *next;/* 指向下一个WR, 最后一个WR的该字段为NULL*/
    struct ibv_sge *sg_list;/* 该WR的sge链表 */
    int num_sge;/* sg_list中ibv_sge的数量*/
};
```

ibv_sge用于描述数据分布

```
struct ibv_sge {
    uint64_t addr;/* buffer基地址 */
    uint32_t length;/* buffer长度 */
    uint32_t lkey;/* 对应的MR中的local key (lkey)*/
};
```

注：该API仅在对端执行Send、Send with Immediate、RDMA Write with Immediate时使用

数据传输——接收（2/2）

- 接收请求处理
  - 示例：该示例仅提交了一个WR，如需要提交多个链表，则需设置next指针

```
struct ibv_sge list = {
        .addr    = (uintptr_t) buf,
        .length = size,
        .lkey    = mr->lkey
};
struct ibv_recv_wr wr = {
        .wr_id       = PINGPONG_RECV_WRID,
        .next       = NULL,
        .sg_list     = &list,
        .num_sge     = 1,
};
struct ibv_recv_wr *bad_wr;
if (ibv_post_recv(qp, &wr, &bad_wr)){
        .....    //错误处理
}
```

数据传输——发送（1/4）

- 发送请求处理
  - API: int ibv_post_send(struct ibv_qp *qp, struct ibv_send_wr *wr, struct ibv_send_wr **bad_wr)　注：该API用于所有通信操作的发起
  - 在SQ上提交一个WR的链表，在第一个错误出现时，将停止对WR列表的处理，并在bad_wr中返回一个指向违规的WR的指针。
  - 参数说明：

```
struct ibv_send_wr {
     uint64_t wr_id;/* 用户分配的WR ID */
     struct ibv_send_wr *next;/* 指向下一个WR，最后一个WR的该字段为NULL*/
     struct ibv_sge *sg_list;/* 该WR的sge链表 */
     int num_sge;/* sg_list中ibv_sge的数量*/
     enum ibv_wr_opcode opcode;
     int send_flags;
     uint32_t imm_data; /* 立即数*/
     接下页
     .......
```

Opcode:

```
IBV_WR_RDMA_WRITE
IBV_WR_RDMA_WRITE_WITH_IMM
IBV_WR_SEND
IBV_WR_SEND_WITH_IMM
IBV_WR_RDMA_READ
IBV_WR_ATOMIC_CMP_AND_SWP
IBV_WR_ATOMIC_FETCH_AND_ADD
```

send_flags:

```
IBV_SEND_FENCE //保证顺序标志
IBV_SEND_SIGNALED //产生完成事件标志
IBV_SEND_SEND_SOLICITED //事件中断显示通知标志
IBV_SEND_INLINE //将sg_list中的数据直接放在WR中发出
```

数据传输——发送（2/4）

- 发送请求处理
  - 参数说明（续）：

```
struct ibv_send_wr {
     ....
     union {
          struct {
               uint64_t remote_addr;/*RDMA操作的远端地址*/
               uint32_t rkey;/* RDMA操作的远端内存key*/
          } rdma;
          struct {
               uint64_t remote_addr; /*atomic操作的远端地址*/
               uint64_t compare_add;/* 用于比较的立即数*/
               uint64_t swap;/* 用于替换的 立即数*/
               uint32_t rkey; /*atomic操作的远端内存key*/
          } atomic;
          struct {
          struct ibv_ah *ah;/* 用于数据报操作的AH */
          uint32_t remote_qpn;/*用于数据报操作的远端qp号*/
          uint32_t remote_qkey;/*用于数据报操作的远端qp键*/
          } ud;
     } wr;
     接下页·····
```

数据传输——发送（3/4）

- 发送请求处理
  - 参数说明（续）：

```
    ....
    union {/* 用于srq操作的远端srq信息 */
        union {
            struct {
                uint32_t remote_srqn;
            } xrc; ;} qp_type;
        uint32_t xrc_remote_srq_num;
    };
    struct {/* 用于绑定memory window（mw）的mw地址、键、绑定信息*/
        struct ibv_mw *mw;
        uint32_t rkey;
        struct ibv_mw_bind_info bind_info;
    } bind_mw;
};
```

数据传输——发送（4/4）

- 发送请求处理
  - 示例：该示例为提交了一个操作为IBV_WR_SEND、产生完成事件IBV_SEND_SIGNALED的WR，如需要提交多个链表，则需设置next指针

```
struct ibv_sge list = {
        .addr    = (uintptr_t) buf,
        .length = size,
        .lkey    = mr->lkey
};
struct ibv_send_wr wr = {
        .wr_id       = PINGPONG_SEND_WRID,
        .sg_list     = &list,
        .num_sge     = 1,
        .opcode      = IBV_WR_SEND,
        .send_flags = IBV_SEND_SIGNALED,
};
struct ibv_send_wr *bad_wr;

if ibv_post_send(qp, &wr, &bad_wr)){
        .....    //错误处理
}
```

查询完成事件（1/2）

- 设置CQ完成事件提醒
  - API: int ibv_req_notify_cq(struct ibv_cq *cq，int solicited_only)
  - 用于请求完成事件提醒
  - 参数说明：solicited_only为0则任何CQE都会触发完成事件；非0，则只有被标记为solicited 的CQE才会触发完成事件。
- 获取CQ完成事件
  - API: int ibv_get_cq_event(struct ibv_comp_channel *channel, struct ibv_cq **cq, void **cq_context)
  - 用于等待CQ将完成事件提醒送到channel，该接口是阻塞操作，若CQ为空，则需一直等待，直到有CQE进入。
- 应答CQ完成事件
  - API: void ibv_ack_cq_events(struct ibv_cq *cq, unsigned int nevents)
  - 用于应答从ibv_get_cq_event得到的CQ完成事件。
  - 参数说明：nevents为需要应答的CQ完成事件数量

查询完成事件（2/2）

- 示例

```
/* 在产生CQE之前，请求CQ完成事件通知*/
if (ibv_req_notify_cq(cq, 0)) {
     fprintf(stderr, "Couldn't request CQ notification\n");
     return 1;
}
......
/* 等待获取CQ完成事件*/
if (ibv_get_cq_event(channel, &ev_cq, &ev_ctx)) {
     fprintf(stderr, "Failed to get cq_event\n");
     return 1;
}
/* 应答CQ完成事件*/
ibv_ack_cq_events(ev_cq, 1);

/*在产生新的CQE之前，重新请求CQ完成事件通知*/
if (ibv_req_notify_cq(ev_cq, 0)) {
     fprintf(stderr, "Couldn't request CQ notification\n");
     return 1;
}
```

查询传输结果（1/2）

- 查询对应的CQ，获得完成信息，排空CQ
  - API: int ibv_poll_cq(struct ibv_cq *cq, int num_entries, struct ibv_wc *wc)
  - 最多查询num_entries个CQE，返回查询到的CQE个数，并在wc中返回CQE的详细信息。调用完该API，CQ排空
  - 参数说明：

```
struct ibv_wc {
    uint64_t wr_id;/* 用户定义的WR ID*/
    enum ibv_wc_status status;/* 返回的WC状态，成功则是IBV_WC_SUCCESS*/
    enum ibv_wc_opcode opcode;/* 操作*/
    uint32_t vendor_err;/* 厂家定义的错误*/
    uint32_t byte_len;/* 已发送的字节数*/
    uint32_t imm_data;
    uint32_t qp_num;/* 本地QP号*/
    uint32_t src_qp;/* 远端QP号*/
    int wc_flags;
    uint16_t pkey_index;
    uint16_t slid;/* 源端LID */
    uint8_t sl;/* 服务级别*/
    uint8_t dlid_path_bits;
};
```

查询传输结果（2/2）

- 示例

```
do {
    ne = ibv_poll_cq(cq, 1, &wc);
    if (ne < 0) {          // ibv_poll_cq返回值小于0表示poll_cq失败
        fprintf(stderr, "Failed to poll completions from the CQ\n");
        return 1;
    }

    if (ne == 0) // ibv_poll_cq返回值等于0表示目前没有新的CQE
        continue;

    /* wc状态不是IBV_WC_SUCCESS，则表示对应的WR执行有误 */
    if (wc.status != IBV_WC_SUCCESS) {
        fprintf(stderr, "Completion with status 0x%x was found\n",
        wc.status);
        return 1;
    }
} while (ne);
```

释放资源

- 通信完成时，调用含有free、de**等字样的API释放资源
  - API:

    void ibv_free_device_list(struct 1 ibv_device **list)
    int ibv_close_device(struct ibv_context *context)
    int ibv_destroy_comp_channel(struct ibv_comp_channel *channel)
    int ibv_dealloc_pd(1 struct ibv_pd *pd)
    int ibv_dereg_mr(1 struct ibv_mr *mr)
    int ibv_destroy_cq(struct ibv_cq *cq)
    int ibv_destroy_ah(1 struct ibv_ah *ah)
    int ibv_destroy_qp(1 struct ibv_qp *qp)
    int ibv_destroy_srq(1 struct ibv_srq *srq)

Soft-RoCE

- IBTA RoCEv2规范的软件实现
  - 基于Linux内核网络栈实现IB RDMA传输层
  - 无需硬件RDMA支持，适合开发、测试、非对称部署
  - 兼容OFED IB Verbs软件栈；区别主要是RoCE与IB编址不同
- 性能特点
  - QP由内核分配后映射入用户空间，应用发起Doorbell需要通过系统调用
  - QP由Linux `tasklet` 软中断处理（“软卸载”）
  - 发送端0拷贝、接收端单次拷贝（可类比Linux `sendfile`）
- RXE软件栈起初分为内核驱动 `rdma_rxe` 与用户级库 `librxe`
  - 上游Linux自版本4.8合并 `rdma_rxe` 模块，上游libibverbs已整合 `librxe` 插件
  - 近来的Linux发行版（如Ubuntu，CentOS≥7.4）大都提供安装包：

```
# yum/dnf install rdma-core libibverbs libibverbs-utils
# apt-get install rdma-core libibverbs1 ibverbs-utils
```

RXE配置

- 加载内核模块

```
# rxe_cfg start
# lsmod | grep rdma_rxe
```

- 添加以太网接口并启动RXE实例（以 *eno1* 为例）

```
# rxe_cfg add eno1
# rxe_cfg status
Name Link Driver Speed NMTU IPv4_addr    RDEV RMTU
eno1 yes  e1000e       1500 xx.xx.xx.xx rxe0 1024 (3)
```

- 查看RXE设备状态

```
$ ibv_devinfo
$ ibv_devices
```

- 验证RXE连通性

```
$ ibv_rc_pingpong -d rxe0 -g 1
$ ibv_rc_pingpong -d rxe0 -g 1 ${SERVER}
```

- 删除RXE设备并卸载内核模块

```
# rxe_cfg remove eno1
# rxe_cfg stop
```

示例程序 - hello_verbs.c

- 两个进程之间通过 RDMA WRITE 发送“hello verbs”字符串消息
- 使用socket传输连接信息：LID、QPN、PSN、(GID、RKEY、VADDR)
- 编译依赖

```
# dnf/yum install rdma-core-devel (libibverbs-devel)
# apt-get install libibverbs-dev
```

- 代码中选择合适的HCA设备、端口、GID索引

```
#define IB_DEV      (0)
#define IB_PORT     (1)
#define GID_INDEX   (1)
```

- 编译执行

```
$ gcc hello_verbs.c -libverbs
$ ./a.out
$ ./a.out ${SERVER}
```

- 程序中假定IB只使用LID；注意IB亦支持GID编址与路由
- 更多参考例子：libibverbs/examples

课程作业 - 网卡非连续数据通信

- 科学计算、企业负载中存在大量非连续数据通信，例如高维数组数据交换、对象序列化/反序列化等
- ibv_send_wr 中的 sg_list 可由网卡卸载 Scatter/Gather 操作以实现0拷贝非连续数据通信

```
struct ibv_sge {
  uint64_t  addr;
  uint32_t  length;
  uint32_t  lkey;
};
```

```
struct ibv_device_attr {
  ...
  int   max_sge;   // MLNX CX-5最多支持30个SGE
  ...
};
```

- Verbs支持 Gather WRITE、Scatter READ、Gather SEND Scatter RECV
- 作业要求：自定非连续数据布局(例 block_size、block_num、stride)
  - 任选 G-WR/S-RD/GSSR 编写Verbs非连续数据通信的测试用例
  - 对不同消息大小，对比网卡0拷贝与CPU有拷贝通信的性能区别
    - 有拷贝即先将非连续数据“打包”至连续缓冲后再进行通信