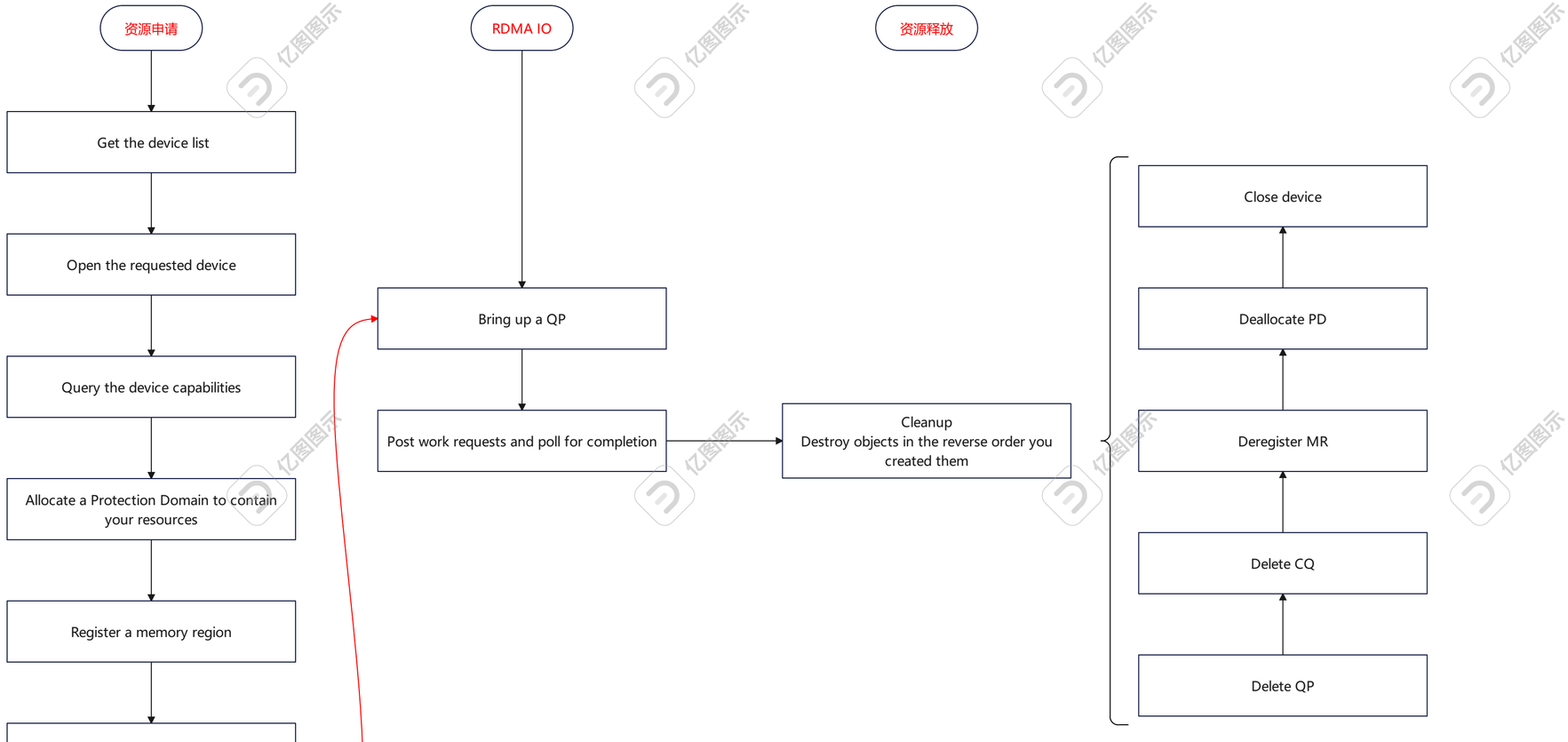
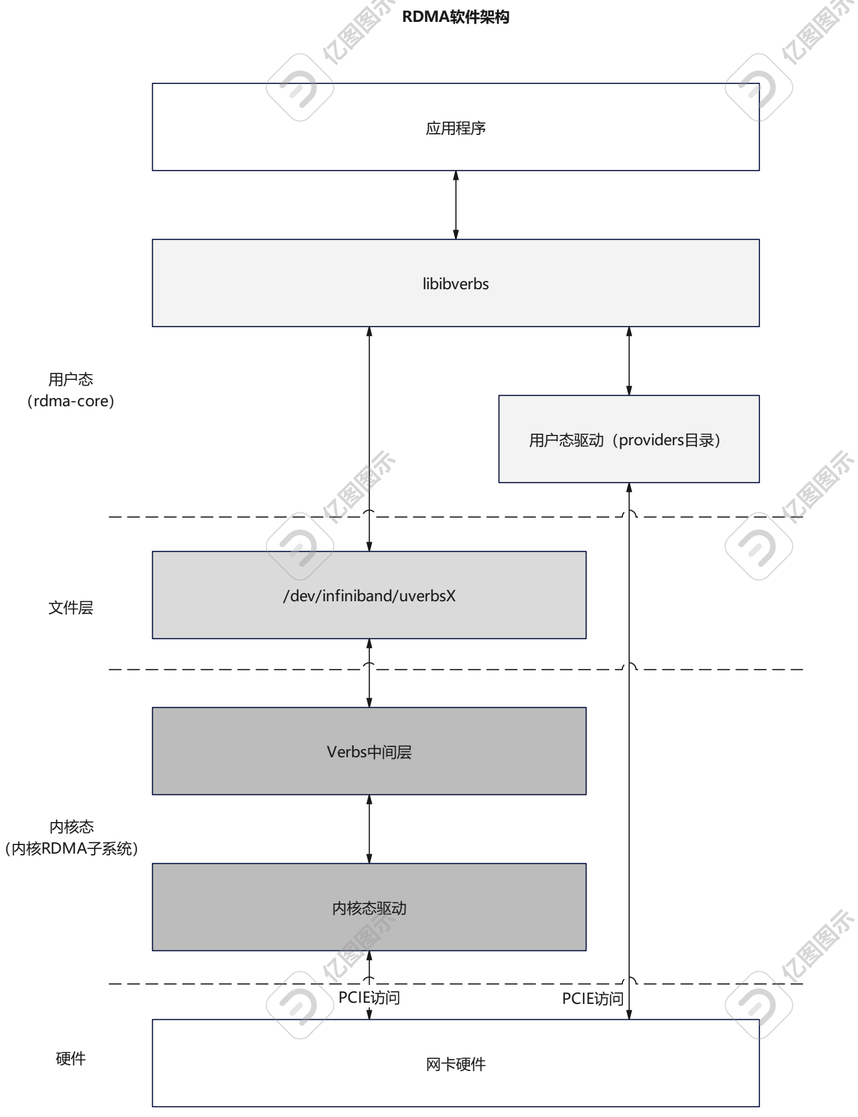
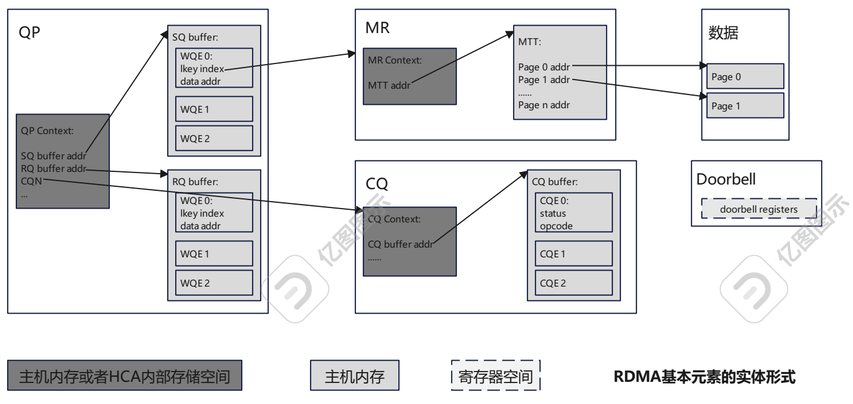
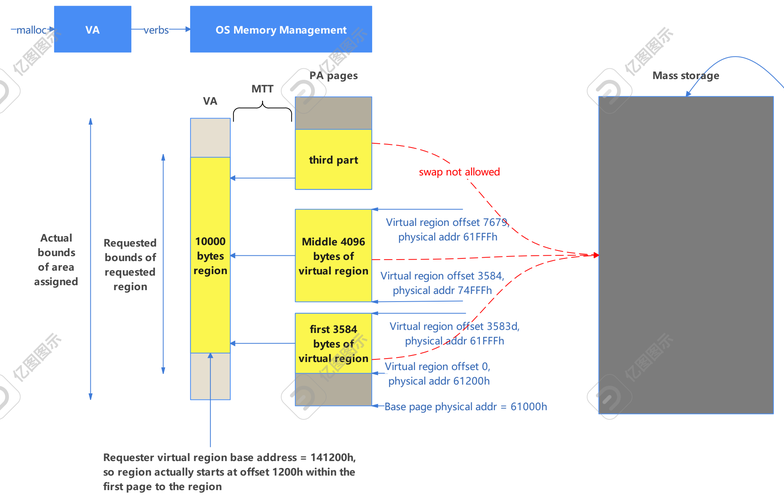
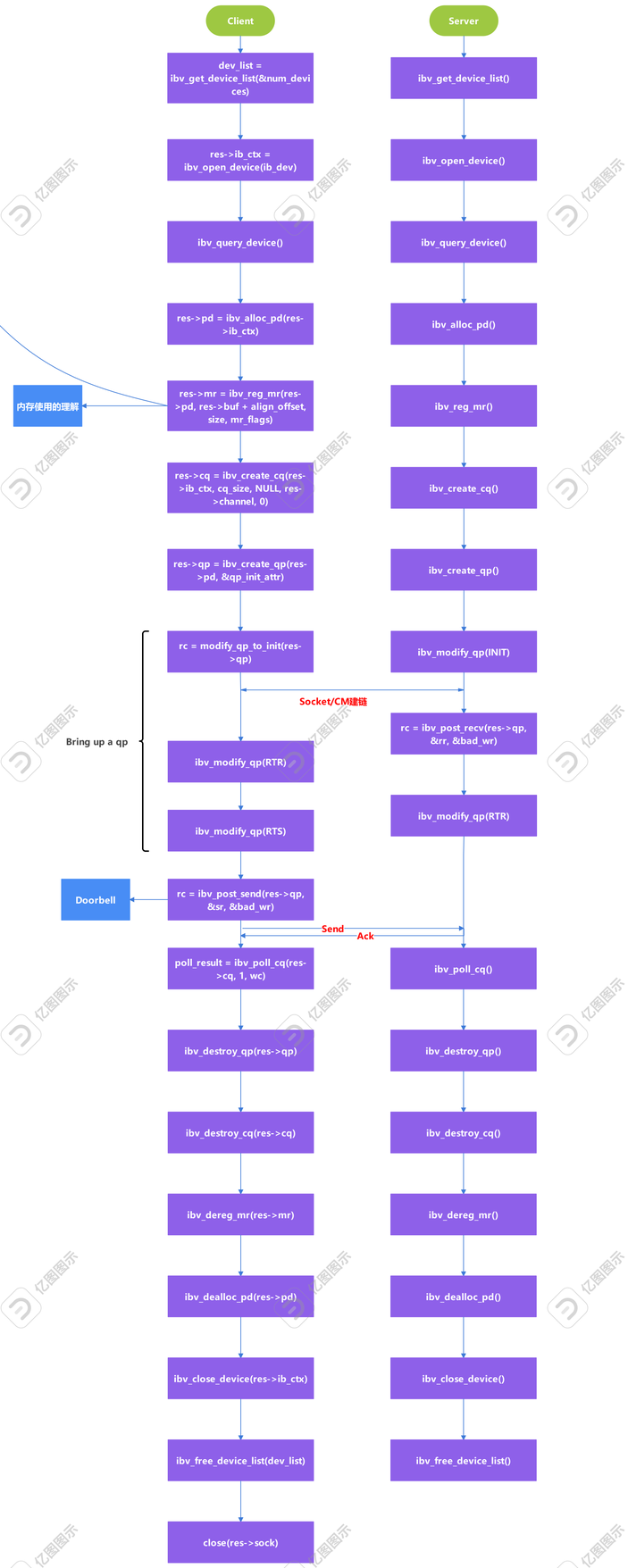






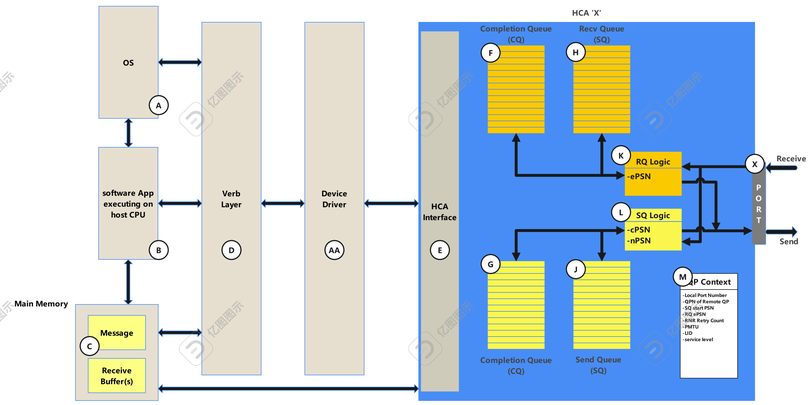
RDMA基本元素的实体形式

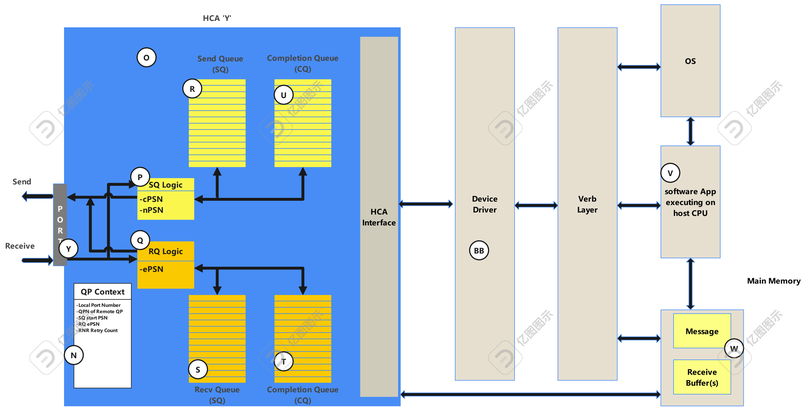

malloc
VA
verbs
OS Memory Management
PA pages
MTT
VA
Mass storage
third part
swap not allowed
Actual bounds of area assigned
Requested bounds of requested region
10000 bytes region
Middle 4096 bytes of virtual region
Virtual region offset 7679, physical addr 61FFFh
Virtual region offset 3584, physical addr 74FFFh
first 3584 bytes of virtual region
Virtual region offset 3583d, physical addr 61FFFh
Virtual region offset 0, physical addr 61200h
Base page physical addr = 61000h
Requester virtual region base address = 141200h, so region actually starts at offset 1200h within the first page to the region


Client
Server
内存使用的理解
dev_list = ibv_get_device_list(&num_devices)
ibv_get_device_list()
res->ib_ctx = ibv_open_device(ib_dev)
ibv_open_device()
ibv_query_device()
ibv_query_device()
res->pd = ibv_alloc_pd(res->ib_ctx)
ibv_alloc_pd()
res->mr = ibv_reg_mr(res->pd, res->buf + align_offset, size, mr_flags)
ibv_reg_mr()
res->cq = ibv_create_cq(res->ib_ctx, cq_size, NULL, res->channel, 0)
ibv_create_cq()
res->qp = ibv_create_qp(res->pd, &qp_init_attr)
ibv_create_qp()
Bring up a qp
rc = modify_qp_to_init(res->qp)
ibv_modify_qp(INIT)
Socket/CM建链
rc = ibv_post_recv(res->qp, &rr, &bad_wr)
ibv_modify_qp(RTR)
ibv_modify_qp(RTR)
ibv_modify_qp(RTS)
Doorbell
rc = ibv_post_send(res->qp, &sr, &bad_wr)
Send
Ack
poll_result = ibv_poll_cq(res->cq, 1, wc)
ibv_poll_cq()
ibv_destroy_qp(res->qp)
ibv_destroy_qp()
ibv_destroy_cq(res->cq)
ibv_destroy_cq()
ibv_dereg_mr(res->mr)
ibv_dereg_mr()
ibv_dealloc_pd(res->pd)
ibv_dealloc_pd()
ibv_close_device(res->ib_ctx)
ibv_close_device()
ibv_free_device_list(dev_list)
ibv_free_device_list()
close(res->sock)

解释逐渐深入

定性描述
流程图示



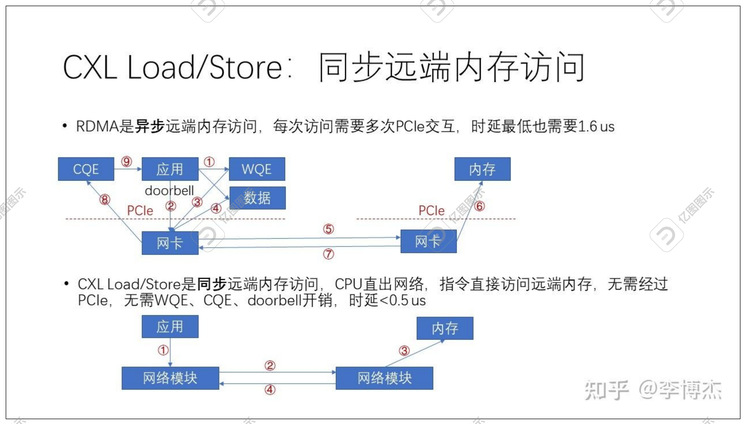


step1:Responder端，分配一块特定大小内存，准备存储发过来的message。通过ibv_post_recv()调用，将WR post到RQ的next entry中，成为了WQE；
step2:Requester端，软件调用ibv_post_send() 把一个WR post到SQ中，成为一个WQE，在WR中表明了本地的message的存放地址等信息；
step3:Requester端，the QP's SQ Logic (item L) starts processing the top SQ entry, 决定是否将message分片成多个message. Responder端，接收packet，进行PSN，opcode的校验，然后回应ack给Requester端，然后再进行写数据到Local memory, 并更新WQE中的addr；
step4:解释了Responder 端对Ack报文的处理；
step5:中间片的处理逻辑，两端
step6:尾片处理完成之后，Responder端就会产生CQE。
step7:Requester端收到Ack之后，进行SQ中的top WQE的retire处理，然后生成CQE

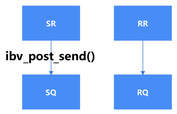
RDMA 中如果想发送一个数据，那么：
1.软件首先会生成一个 WQE（work queue element），就是工作队列里边的一个工作任务。
2.然后这个任务再下发一个 doorbell，就是按一个门铃到网卡告诉说我有事情要做了。
3.接着，网卡在收到这个门铃之后会从内存里面把这个工作任务取到网卡里面。
4.然后再根据工作任务当中的地址，访问内存中的数据，把它 DMA 到网卡。
5.再接下来，网卡会把这个数据封装成一个网络报文，从本地发送到远端。
6.然后，接收端的网卡在收到了这个数据之后，再把它写到远端的内存。
7.接着，接收端的网卡返回一个完成消息说我干完了。
8.发起端的网卡收到了这个完成消息之后，它就在本地内存中生成一个CQE。
9.最后，应用需要去 poll 这个CQE，也就是说它要获取这个完成队列里的完成事件才能够完成整个过程。

作者：李博杰
链接：https://www.zhihu.com/question/392146033/answer/3151675927

——当我把这个过程描述的时候其认为深度远远不够！

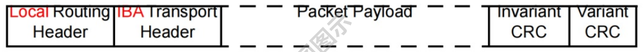


知乎参考

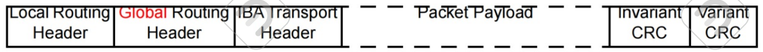


IBA 的包就分为以下几类

Local (within a subnet) Packets



Global (routing between subnets) Packets



Raw Packet with Raw Header

Raw Packet with IPv6 Header

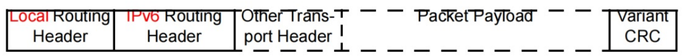


Figure 45 IBA Packet Overview

其中BTH比较重要：

Table 6 Base Transport Header Fields

Field Name	Field Abbreviation	Field Size (in bits)	Description
Opcode	OpCode	8	This field indicates the IBA packet type. The OpCode also specifies which extension headers follow the Base Transport Header
Solicited Event	SE	1	This bit indicates that an event should be generated by the responder.
MigReq	M	1	This bit is used to communicate migration state.
Pad Count	PadCnt	2	This field indicates how many extra bytes are added to the payload to align to a 4 byte boundary.
Transport Header Version	TVer	4	This field indicates the version of the IBA Transport Headers.
Partition Key	P_KEY	16	This field indicates which logical Partition is associated with this packet (see 10.9 Partitioning on page 565)
F/Res1*	F/R	1	F (FECN): 0 indicates that a FECN indication was not received. 1 indicates that the packet went through a point of congestion. Res1*: Transmitted as 0, ignored on receive. This field is not included in the invariant CRC. see 7.8 CRCs on page 220 for details.
B/Res1*	B/R	1	B (BECN): 0 the packet did not go through a point of congestion or went through a point of congestion but was not marked.1 indicates that the packet indicated by this header was subject to forward congestion. The B bit is set in an ACK or CN BTH. Res1*: Transmitted as 0, ignored on receive. This field is not included in the invariant CRC. see 7.8 CRCs on page 220 for details.
Reserved (variant)		6	Transmitted as 0, ignored on receive. This field is not included in the invariant CRC. see 7.8 CRCs on page 220 for details.
Destination QP	DestQP	24	This field indicates the Work Queue Pair Number (a.k.a. QP) at the destination
Acknowledge Request	A	1	This bit is used to indicate that an acknowledge (for this packet) should be scheduled by the responder.
Reserved		7	Transmitted as 0, ignored on receive. This field is included in the invariant CRC.
Field Name	**Field Abbreviation**	**Field Size (in bits)**	**Description**
Packet Sequence Number	PSN	24	This field is used to detect a missing or duplicate Packet. See 9.7.1 Packet Sequence Numbers (PSN) on page 314 for a detailed description of PSN.

The Previous Part

Part 1 introduced basic concepts and terminology and consisted of the following chapters:

- Chapter 1—Basic Terms and Concepts.
- Chapter 2—Intro to Attributes and Managers.
- Chapter 3—QP: Message Transfer Mechanism.
- Chapter 4—Intro to Transport Types.
- Chapter 5—Intro to Send/Receive Operations.
- Chapter 6—Division of Labor.
- Chapter 7—Subnet-Local Addressing.
- Chapter 8—Global Addressing.
- Chapter 9—Intro to the Managers.
- Chapter 10—Intro to Connection Establishment.
- Chapter 11—PSN Usage.


Setup_client
Setup_Server
exchange_r_key(&conn)
Master Side
Slave Side
wait_cq(conn, 0);
ibv_post_send(conn->cm_id->qp, &conn->send_wr[0], &failed_send_wr)
ibv_post_recv(conn->cm_id->qp, &conn->recv_wr[0], &failed_recv_wr)
wait_cq(conn, 0)
test_write_or_read(struct connection *conn, int write_only)
ibv_post_send(conn->cm_id->qp, &conn->rw_wr[i], &failed_send_wr
ibv_poll_cq(conn->cq, 1, &wc)
test_write_read(struct connection *conn)
fill_rdma_buffer(conn, i, total_loop)
conn->rw_wr[i].opcode = IBV_WR_RDMA_WRITE; ibv_post_send(conn->cm_id->qp, &conn->rw_wr[i], &failed_send_wr)
wait_cq(conn, total_loop)
reset_rdma_buffer(conn, i, total_loop)
conn->rw_wr[i].opcode = IBV_WR_RDMA_READ; ibv_post_send(conn->cm_id->qp, &conn->rw_wr[i], &failed_send_wr)
wait_cq(conn, total_loop)
verify_rdma_buffer(conn, i, total_loop)
创建Channel 和 id
rdma_create_event_channel()
rdma_create_id(conn->channel, &conn->cm_id, &conn, RDMA_PS_TCP)
rdma_getaddrinfo(prog_opt.connect_addr, buf, &conn->hints, &conn->rai)
rdma_freeaddrinfo(conn->rai)
解析Address和Route
rdma_resolve_addr(conn->cm_id, conn->rai->ai_src_addr, conn->rai->ai_dst_addr, 2000)
rdma_get_cm_event(conn->channel, &event) event->event == RDMA_CM_EVENT_ADDR_RESOLVED ?
rdma_resolve_route(conn->cm_id, 2000)
rdma_ack_cm_event(event)
rdma_get_cm_event(conn->channel, &event) event->event == RDMA_CM_EVENT_ROUTE_RESOLVED ?
prepare_qp(conn)
setup_ack_timeout(conn)
链接 Connect
rdma_connect(conn->cm_id, &conn_param)
rdma_ack_cm_event(event)
rdma_get_cm_event(conn->channel, &event) event->event == RDMA_CM_EVENT_ESTABLISHED ?
rdma_ack_cm_event(event)
print_peer_addr(conn)
rdma_create_event_channel()
rdma_create_id(conn->channel, &conn->listen_id, &conn, RDMA_PS_TCP)
rdma_getaddrinfo(prog_opt.listen_addr, buf, &conn->hints, &conn->rai)
rdma_bind_addr(conn->listen_id, conn->rai->ai_src_addr)
rdma_listen(conn->listen_id, 0)
rdma_get_cm_event(conn->channel, &event) event->event = RDMA_CM_EVENT_CONNECT_REQUEST ?
wait for client
prepare_qp(conn)
setup_ack_timeout(conn)
rdma_ack_cm_event(event)
rdma_accept(conn->cm_id, NULL)
rdma_get_cm_event(conn->channel, &event) event->event == RDMA_CM_EVENT_ESTABLISHED ?
rdma_ack_cm_event(event)
print_peer_addr(conn)
ibv_alloc_pd(conn->cm_id->verbs)
ibv_create_comp_channel(conn->cm_id->verbs)
ibv_create_cq(conn->cm_id->verbs, conn->cqe_num, conn, conn->cc, 0 /* comp_vector */)
Alloc PD
rdma_create_qp(conn->cm_id, conn->pd, &init_qp_attr)
ibv_reg_mr(conn->pd, buffer, conn->data_size, IBV_ACCESS_LOCAL_WRITE)
Prepare Receive Buffer
ibv_post_recv(conn->cm_id->qp, &conn->recv_wr[i], &recv_wr_failed)
ibv_reg_mr(conn->pd, buffer, conn->data_size, IBV_ACCESS_LOCAL_WRITE)
PrePare Send Buffer
RDMA Slave
RDMA Master
ibv_reg_mr(conn->pd, offset_buffer, rdma_data_size, IBV_ACCESS_LOCAL_WRITE| IBV_ACCESS_REMOTE_READ| IBV_ACCESS_REMOTE_WRITE)
ibv_reg_mr(conn->pd, offset_buffer, conn->rdma_data_size, IBV_ACCESS_LOCAL_WRITE

CM建链的底层仍然调用的是ibv_modify_qp()来改变QP的状态，client与server之间传递的是UD报文，通过QP1进行传递！
那么需要梳理清楚的就是哪里传递的信息，哪里迁移的状态！
CM建链有自己的一套！

务必要先启动server，setup的时候进行rdma_bind_addr(), rdma_listen.

对端client进行address和route的resolve，然后才启动connect，
server收到了request之后才往下运行。
后面二者才能完成 connect eatablished的event的确认！

可改进点：
1.程序中的rdma_ack_cm_event()可以在最后执行！

Server

rdma_create_event_channel()

rdma_create_id(conn->channel, &conn->listen_id, &conn, RDMA_PS_TCP)

rdma_getaddrinfo(prog_opt.listen_addr, buf, &conn->hints, &conn->rai)

rdma_getaddrinfo provides transport independent address translation.
It resolves the destination node and service address
and returns information required to establish device communication.
It is the functional equivalent of getaddrinfo.

get a addr and bind

rdma_bind_addr(conn->listen_id, conn->rai->ai_src_addr)

rdma_bind_addr associates a source address with an rdma_cm_id.
The address may be wildcarded.
If binding to a specific local address,
the rdma_cm_id will also be bound to a local
RDMA device.
将source address 和 cm_id连接起来！

rdma_listen(conn->listen_id, 0)

rdma_listen initiates a listen for incoming connection
requests or datagram service lookup. The
listen is restricted to the locally bound source address.

听着对端有没有来请求信息

rdma_get_cm_event(conn->channel, &event)

Server 端执行到这里的时候就卡在这里等着，等待rdma_get_cm_event()执行返回值，
此时需要client端来执行rdma_connect()的操作，
从而在这里的Server端才能产生CONNECT_REQUEST. 这个函数才能执行结束！

event->event = RDMA_CM_EVENT_CONNECT_REQUEST ?

wait for client

prepare_qp(conn)

Prepare a qp when received a connect request from cma

rdma_ack_cm_event(event)

rdma_accept(conn->cm_id, NULL)

ibv_modify_qp(rtr)
ibv_modify_qp(rts)

rdma_get_cm_event(conn->channel, &event)

执行到这里也得等着，只有当Client 端进行了ack的释放操作，
这里才能得到event, 执行完并进行返回！

accept

event->event == RDMA_CM_EVENT_ESTABLISHED

rdma_ack_cm_event(event)

Client

rdma_create_event_channel()

rdma_create_id(conn->channel, &conn->cm_id, &conn, RDMA_PS_TCP)

rdma_getaddrinfo(prog_opt.connect_addr, buf, &conn->hints, &conn->rai)

rdma_resolve_addr(conn->cm_id, conn->rai->ai_src_addr, conn->rai->ai_dst_addr, 2000)

rdma_get_cm_event(conn->channel, &event)
event->event == RDMA_CM_EVENT_ADDR_RESOLVED ?

rdma_resolve_route(conn->cm_id, 2000)

rdma_ack_cm_event(event)

rdma_get_cm_event(conn->channel, &event)
event->event == RDMA_CM_EVENT_ROUTE_RESOLVED

prepare_qp(conn)

Prepare a qp to connect to remote side

rdma_connect(conn->cm_id, &conn_param)

rdma_ack_cm_event(event)

rdma_ack_cm_event frees a communication event. All events which are allocated by
rdma_get_cm_event must be released, there should be a one-to-one correspondence between successful gets and acks. This call
frees the event structure and any memory that it references.

rdma_get_cm_event(conn->channel, &event)

event->event == RDMA_CM_EVENT_ESTABLISHED

rdma_ack_cm_event(event)

Client端必须先发起connect，
发送一个ConnectRequest报文

返回一个
ConnectReply报文

发送一个
ReadyToUse报文

应用显示过滤器 … <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.0.5	10.0.0.4	RRoCE	322	CM: ConnectRequest
2	0.017853	10.0.0.4	10.0.0.5	RRoCE	322	CM: ConnectReply
3	0.018281	10.0.0.5	10.0.0.4	RRoCE	322	CM: ReadyToUse
4	0.018286	10.0.0.5	10.0.0.4	RRoCE	130	RC RDMA Write Only Immediate QP=0x0004e9
5	0.018287	10.0.0.5	10.0.0.4	RRoCE	94	RC RDMA Write Only Immediate QP=0x0004e9
6	0.018509	10.0.0.4	10.0.0.5	RRoCE	94	RC RDMA Write Only Immediate QP=0x000003
7	0.018514	10.0.0.5	10.0.0.4	RRoCE	62	RC Acknowledge QP=0x0004e9
8	0.018679	10.0.0.4	10.0.0.5	RRoCE	62	RC Acknowledge QP=0x000003

test_send_recv(&conn)

ibv_req_notify_cq(conn->cq, 0)

这里的wait_cq()得停一下，等对面client发送完成之后，这里才能执行结束
输出结果为：
wr_id=0 opcode=0x80 (recv) status=0

wait_cq(conn, total_loop)

ibv_post_send(conn->cm_id->qp, &conn->send_wr[i], &failed_send_wr)

发送数据要进行一个标志判断和数据填充

if (prog_opt.verify)
    fill_send_buffer(conn, i, total_loop);

ret = ibv_post_send(conn->cm_id->qp, &conn->send_wr[i], &failed_send_wr);

verify_recv_buffer(conn, i, total_loop)

wait_cq(conn, total_loop)

执行wait_cq的返回结果是
wr_id=0 opcode=0x00 (send) status=0

ibv_post_recv(conn->cm_id->qp, &conn->recv_wr[i], &failed_recv_wr)

wait_cq(conn, total_loop)

返回结果是
wr_id=0 opcode=0x80 (recv) status=0

ibv_post_send(conn->cm_id->qp, &conn->send_wr[i], &failed_send_wr)

verify_recv_buffer(conn, i, total_loop)

根据 prog_opt.verify 的值进行验证数据。
输出结果是
wr_index=0 sge=0 mark=1

POST receive 和 post receive send 的测试！
是一个send和receive流程的循环，
这里列出了2次

wait_cq(conn, total_loop)

输出结果为：
wr_id=0 opcode=0x00 (send) status=0

ibv_post_recv(conn->cm_id->qp, &conn->recv_wr[i], &failed_recv_wr)

输出结果为
wr_id=1 opcode=0x80 (recv) status=0

wait_cq(conn, total_loop)

ibv_post_send(conn->cm_id->qp, &conn->send_wr[i], &failed_send_wr)

verify_recv_buffer(conn, i, total_loop)

输出是：
wr_index=1 sge=0 mark=2

wait_cq(conn, total_loop)

输出结果为：
wr_id=1 opcode=0x00 (send) status=0

ibv_post_recv(conn->cm_id->qp, &conn->recv_wr[i], &failed_recv_wr)

wait_cq(conn, total_loop)

输出为：
wr_id=1 opcode=0x80 (recv) status=0

ibv_post_send(conn->cm_id->qp, &conn->send_wr[i], &failed_send_wr)

verify_recv_buffer(conn, i, total_loop)

根据 prog_opt.verify 进行验证数据
输出结果为：
wr_index=1 sge=0 mark=2

wait_cq(conn, total_loop)

输出结果是：
wr_id=1 opcode=0x00 (send) status=0

ibv_post_recv(conn->cm_id->qp, &conn->recv_wr[i], &failed_recv_wr)

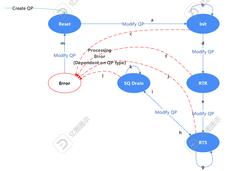
共循环100次

共循环100次

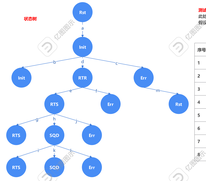
亿图图示

这些状态的变迁图都调用modify verb 函数
InfiniBand network architecture第7章，
Page 237- 那状态要产生的变迁都用哪些所指的必须所引发的参数值？

状态机图



状态树



测试策略

总结出verb是一条有始有终的对问题状态变迁，
假设有8个verb，每一个对应下面的一条case测试。

序号	状态迁移case（0-Switch）	描述
1	Rst->a->Init->b->Init	
2	Rst->a->Init->d->RTR->e->RTS->g->RTS	
3	Rst->a->Init->d->RTR->e->RTS->h->SQD->i->RTS	
4	Rst->a->Init->d->RTR->e->RTS->h->SQD->k->SQD	
5	Rst->a->Init->d->RTR->e->RTS->h->SQD->l->Err	
6	Rst->a->Init->d->RTR->e->RTS->j->Err	
7	Rst->a->Init->d->RTR->f->Err	
8	Rst->a->Init->c->Err->m->Rst	

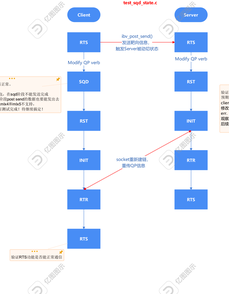
基本函数

基本动作	函数
a	rst_to_init()
b	init_to_init()
c	init_to_error()
d	init_to_rtr()
e	rtr_to_rts()
f	rtr_to_err()
g	rts_to_rts()
h	rts_to_sqd()
i	sqd_to_rts()
j	rts_to_err()
k	sqd_to_sqd()
l	sqd_to_err()
m	err_to_rst()

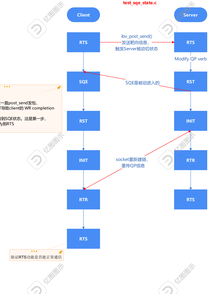
不是全面测试，只需测试会影响到的部分！
三个case：
1. 从RTS（不能出现的）-> ERR（验证是否进入ERR状态，能否产生CQ）-> RTS（ready to work）
2.RTS -> SQD（验证发现功能）-> RTS
3.RTS -> RST -> RTS
——
三个case，先完成一次测试一个case测试。
两端都要验证接收和发送，（同一台客户端程序，能数据能够成功，有可能需要重新建链（参考pingpong程序中的worker），一般最简短环境（post_send 和比较正式一个环境发现问题，只能创造入问题的状态）。

CASE 1:
test_error_state.c

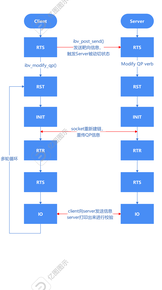
CASE 2:
test_sqd_state.c



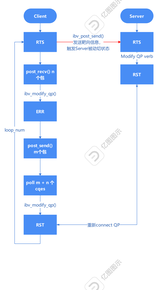
CASE 3:
test_sqe_state.c



CASE 4:
test_rst_state.c



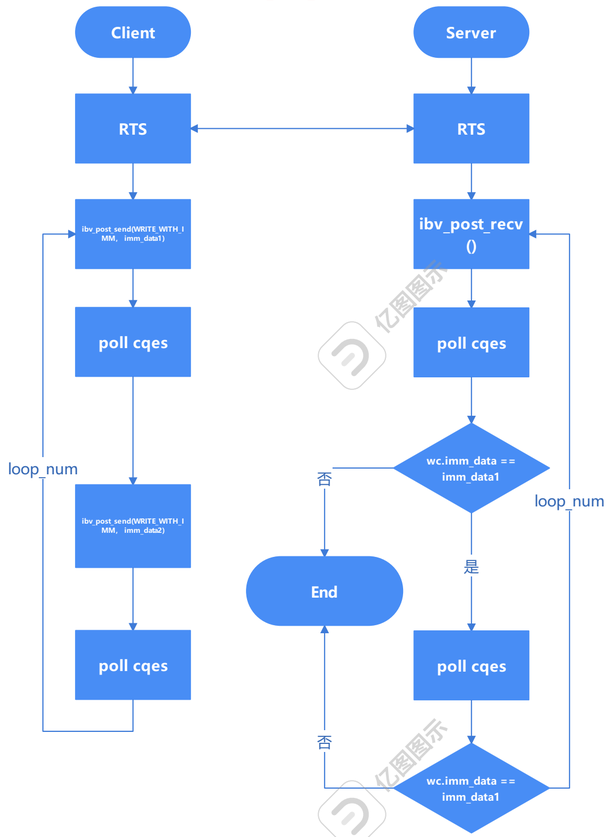
CASE 5:
test_error_state_plus.c



三. IO传输参数测试

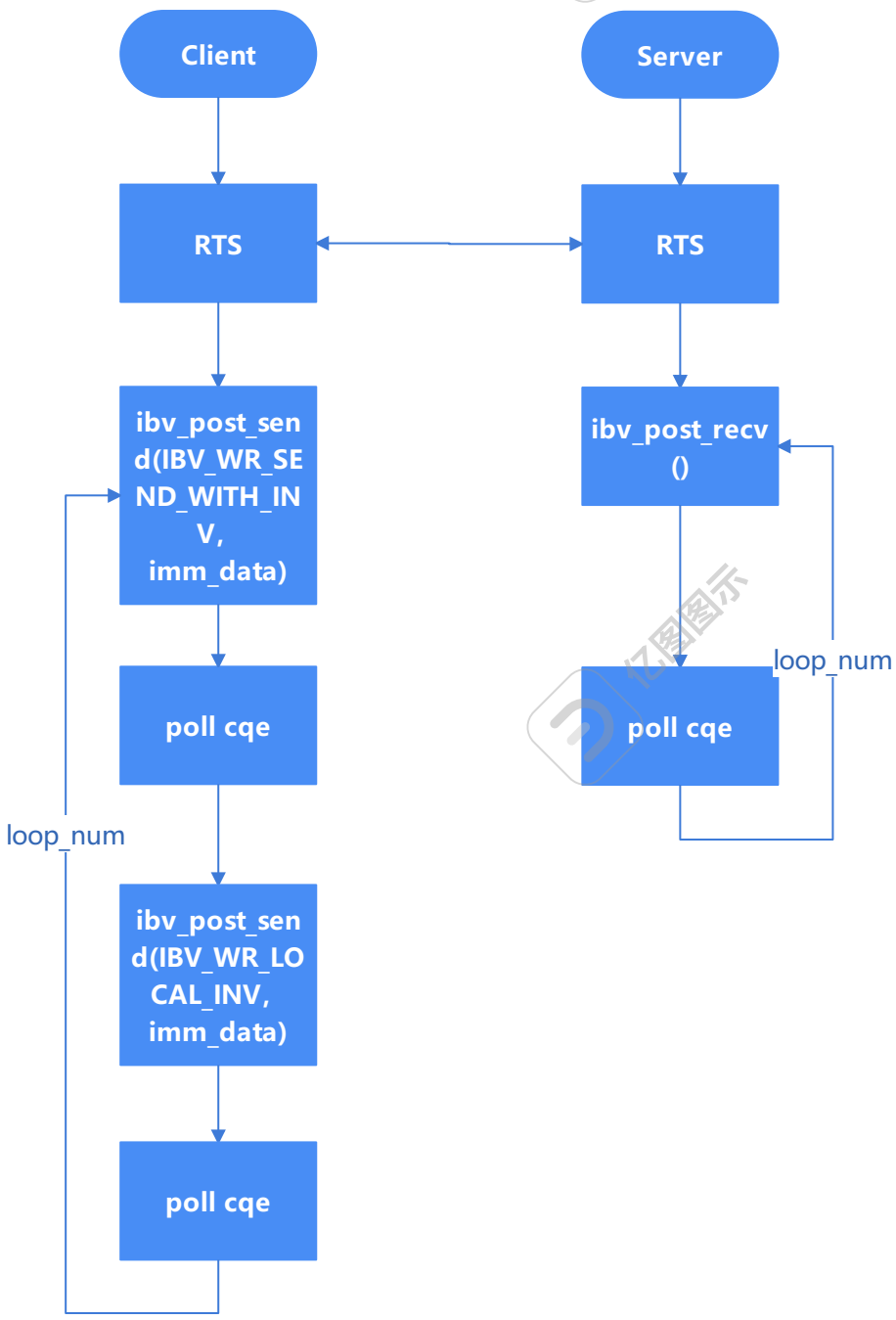
序号	测试内容	工具	测试步骤与要求
1	IBV_WR_SEND_WITH_IMM	内部用户态工具	
2	IBV_WR_RDMA_WRITE_WITH_IMM		
3	IBV_WR_SEND_WITH_INV		仅检查命令能否成功（暂无实际功能）
4	IBV_WR_LOCAL_INV		检查命令是否成功，本地key能否使用
5	IBV_SEND_SIGNALED		每次发2个包， 第一个包不设IBV_SEND_SIGNALED,第2个包设置。 通过标准是抓包能看到所有包， 但cqe只有第2个的。 (总循环次数为3*队列深度)
6	IBV_SEND_SOLICITED & Solicited Completion Event		1. ibv_notify_req_cq设置solicited_only 2. 每组发送2个包，带ibv_send_solicited的和不带的。 3. 抓包能看到所有包，但接受侧中断只产生solicated的包

CASE 1：
test_imm_data.c



亿图图示

CASE 1:
test_wr_inv.c

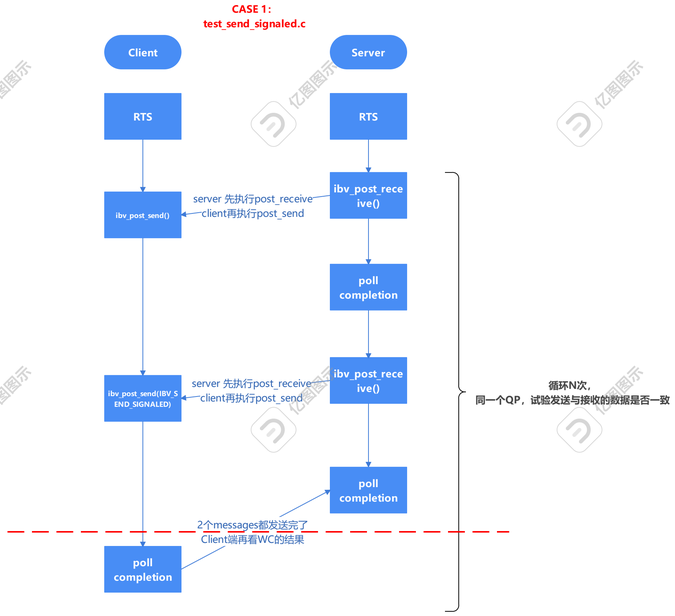


- **IBV_SEND_SIGNALED** - Set the completion notification indicator for this WR. This means that if the QP was created with *sq_sig_all*=0, a Work Completion will be generated when the processing of this WR will be ended. If the QP was created with *sq_sig_all*=1, there won't be any effect to this flag

创建QP的时候，让sq_sig_all = 0：（如果等于1, 那么产生CQE就不会受这个IBV_SEND_SIGNALED的影响了）；
post_send(IBV_SEND_SIGNALED)才会产生CQE;
post_send()不会产生CQE;

所以sq_sig_all 和 IBV_SEND_SIGNALED 是配合使用的！用于控制本端发送的时候要不要产生CQE的！

测试方案：
1.创建QP的时候让sq_sig_all = 0;
2.Client连续发两个包，第一个包不设置SEND_SIGNALED, 第二个包设置；
3.Client 出口（Server入口）可以抓到两个包，client设置poll 2个CQE的配置（队列深度为2），但是预期只能得到一个CQE，并且是第二个包（通过wr_id判断）的。第一个包的行为是：无法poll 到CQE。



> - **IBV_SEND_SOLICITED** - Set the solicited event indicator for this WR. This means that when the message in this WR will be ended in the remote QP, a solicited event will be created to it and if in the remote side the user is waiting for a solicited event, it will be woken up. Relevant only for the Send and RDMA Write with immediate opcodes

理解：
参考OneNote中的相关设计总结
总体业务代码逻辑：
server端，post_receive()之后准备接收了，就可以使用下面的机制了，然后等待events的产生
ibv_req_notify_cq(pp_cq(ctx), solicited_flag); // 使能一种机制，何种情况下产生notification

Client端，post_send()（发两个，先普通，后solicited）

```
while() {
        ibv_get_cq_event(ctx->channel, &ev_cq, &ev_ctx);  // 等待这个notification告诉我们产生了CQE
        num_cq_events++; // 计数增加
        ibv_req_notify_cq(pp_cq(ctx), 0);   // 重复执行

        ibv_poll_cq(pp_cq(ctx), 2, wc); // 处理CQE
}
```

ibv_ack_cq_events(pp_cq(ctx), num_cq_events); // 确认知道了，get_cq_events收到了events
判断event的次数是否符合预期。
组合验证发送端的ibv_send_solicited与接收端的solicited_flag。

本业务要求覆盖四种组合测试：
-o 0：只测IBV_SEND_SOLICITED 不测cq event（IBV_SEND_SIGNALED =1）
——client端：post_send(IBV_SEND_SOLICITED)，
——Server端：不使用 ibv_req_notify_cq() 机制
预期是就是最普通的场景，两端都只产生CQE，没有event

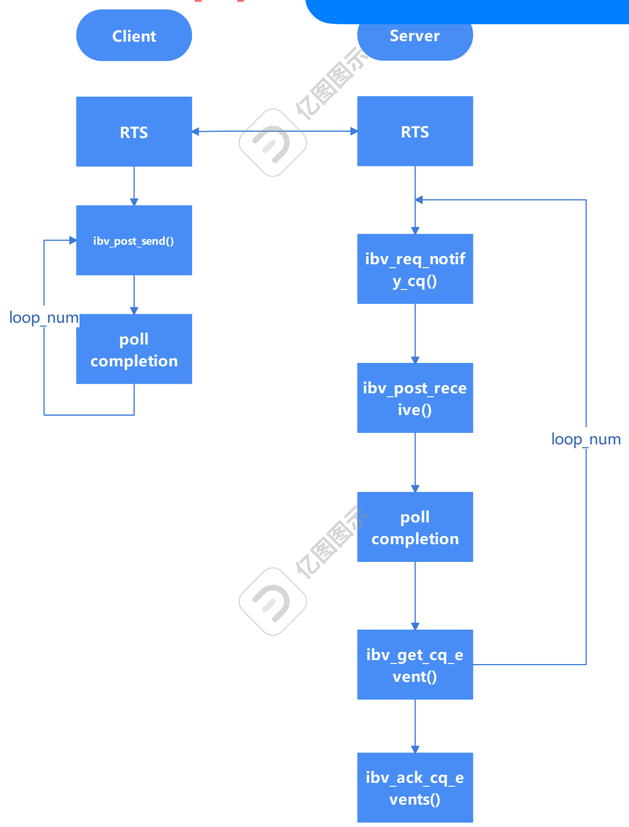
-o 1：IBV_SEND_SOLICITED + ibv_req_notify_cq(struct ibv_cq *cq, int solicited_only) (IBV_SEND_SIGNALED = 1) (solicited_only = 1)
——Client端，post_send(IBV_SEND_SOLICITED);
——Server端调用ibv_req_notify_cq(1)机制，
预期两端都产生CQEs，server端有event

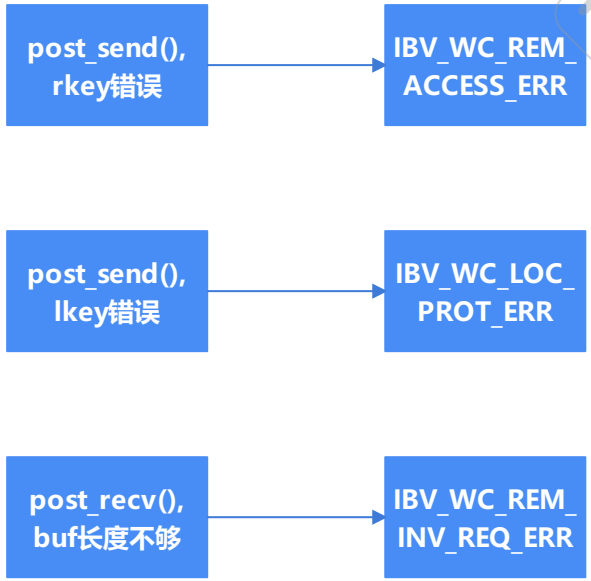
-o 2: 只测IBV_SEND_SOLICITED 不测cq event（IBV_SEND_SIGNALED =0）
——client端：post_send(0)，sleep(1)，预期不产生CQE，
——Server端：不调用 ibv_req_notify_cq() 机制；
预期在client端不产生CQE，在server端会产生CQE，没有event
——因为在client端，对于sq_sig_all = 0 的情况，只有IBV_SEND_SIGNALED才能完成整个过程，产生CQE

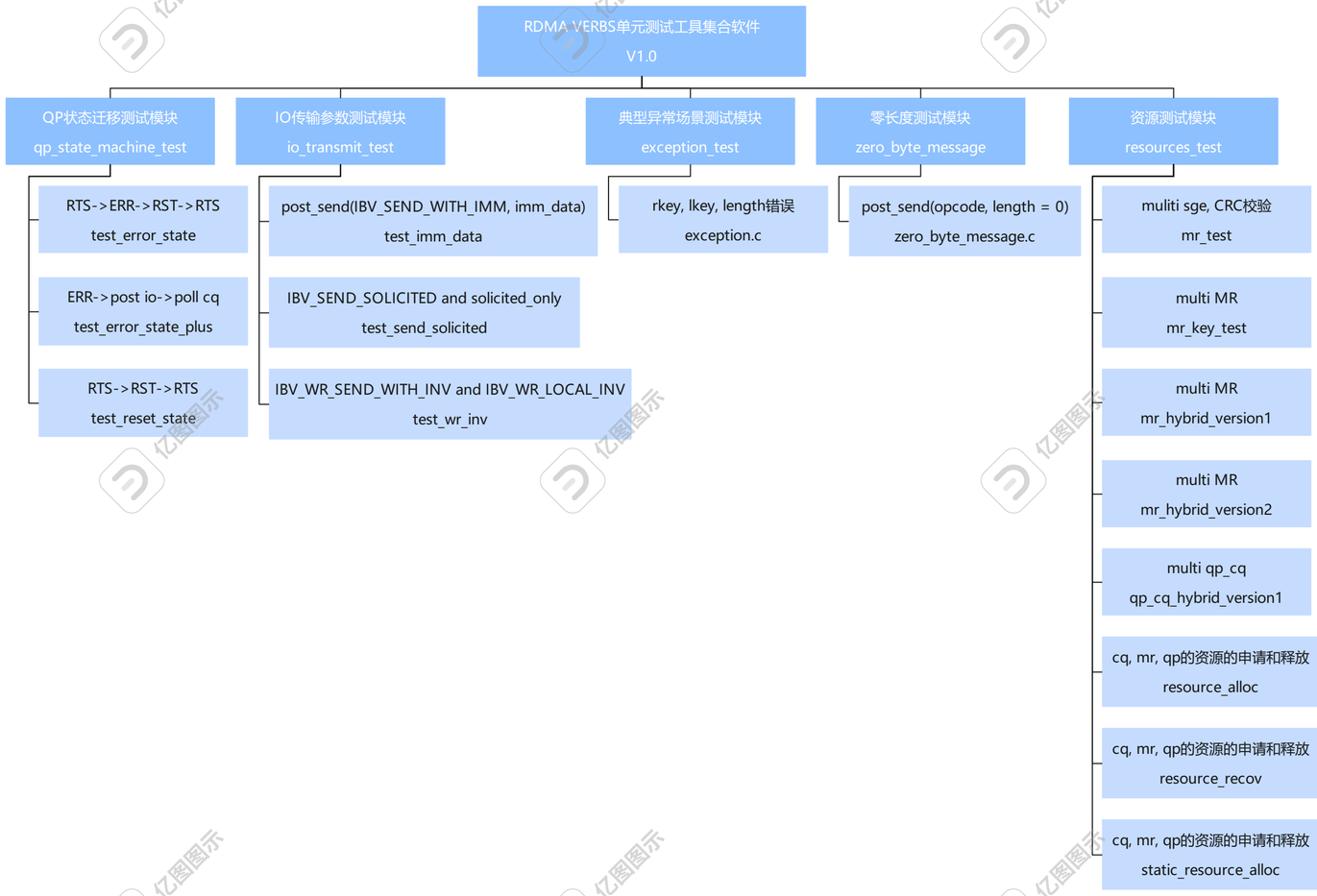
-o 3: IBV_SEND_SOLICITED + ibv_req_notify_cq(struct ibv_cq *cq, int solicited_only) (IBV_SEND_SIGNALED = 0)
——Client端，post_send(0);
——Server端调用ibv_req_notify_cq(1)；
预期client不会产生CQE，server会产生CQE，没有event，不进行预期event

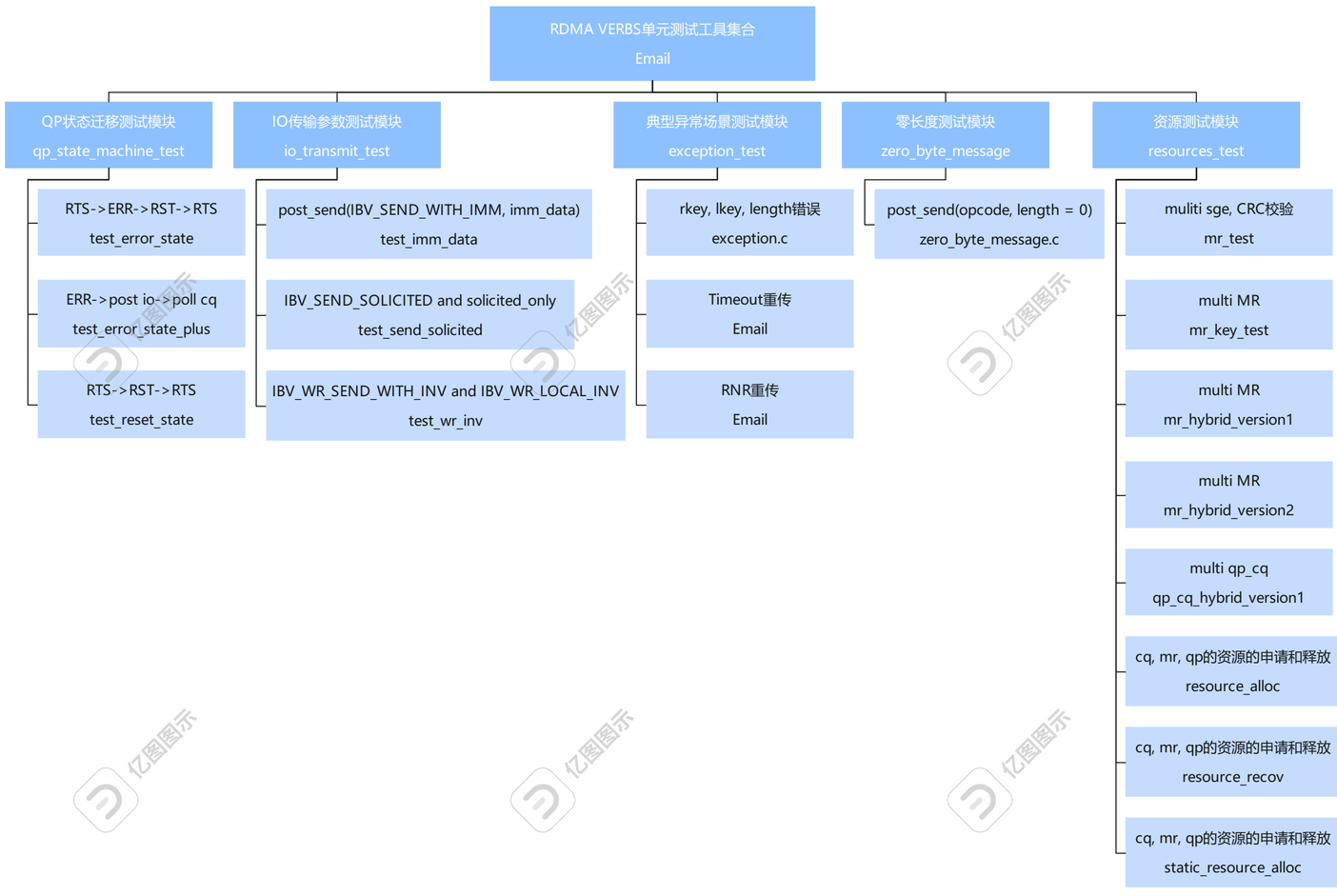
此流程图总结一个solicited功能正
和记录！

CASE 1:
test_send_solicited




post_send(),
rkey错误
IBV_WC_REM_
ACCESS_ERR
post_send(),
lkey错误
IBV_WC_LOC_
PROT_ERR
post_recv(),
buf长度不够
IBV_WC_REM_
INV_REQ_ERR


RDMA VERBS单元测试工具集合软件
V1.0
QP状态迁移测试模块
qp_state_machine_test
IO传输参数测试模块
io_transmit_test
典型异常场景测试模块
exception_test
零长度测试模块
zero_byte_message
资源测试模块
resources_test
RTS->ERR->RST->RTS
test_error_state
ERR->post io->poll cq
test_error_state_plus
RTS->RST->RTS
test_reset_state
post_send(IBV_SEND_WITH_IMM, imm_data)
test_imm_data
IBV_SEND_SOLICITED and solicited_only
test_send_solicited
IBV_WR_SEND_WITH_INV and IBV_WR_LOCAL_INV
test_wr_inv
rkey, lkey, length错误
exception.c
post_send(opcode, length = 0)
zero_byte_message.c
muliti sge, CRC校验
mr_test
multi MR
mr_key_test
multi MR
mr_hybrid_version1
multi MR
mr_hybrid_version2
multi qp_cq
qp_cq_hybrid_version1
cq, mr, qp的资源的申请和释放
resource_alloc
cq, mr, qp的资源的申请和释放
resource_recov
cq, mr, qp的资源的申请和释放
static_resource_alloc


RDMA VERBS单元测试工具集合
Email
QP状态迁移测试模块
qp_state_machine_test
IO传输参数测试模块
io_transmit_test
典型异常场景测试模块
exception_test
零长度测试模块
zero_byte_message
资源测试模块
resources_test
RTS->ERR->RST->RTS
test_error_state
ERR->post io->poll cq
test_error_state_plus
RTS->RST->RTS
test_reset_state
post_send(IBV_SEND_WITH_IMM, imm_data)
test_imm_data
IBV_SEND_SOLICITED and solicited_only
test_send_solicited
IBV_WR_SEND_WITH_INV and IBV_WR_LOCAL_INV
test_wr_inv
rkey, lkey, length错误
exception.c
Timeout重传
Email
RNR重传
Email
post_send(opcode, length = 0)
zero_byte_message.c
muliti sge, CRC校验
mr_test
multi MR
mr_key_test
multi MR
mr_hybrid_version1
multi MR
mr_hybrid_version2
multi qp_cq
qp_cq_hybrid_version1
cq, mr, qp的资源的申请和释放
resource_alloc
cq, mr, qp的资源的申请和释放
resource_recov
cq, mr, qp的资源的申请和释放
static_resource_alloc

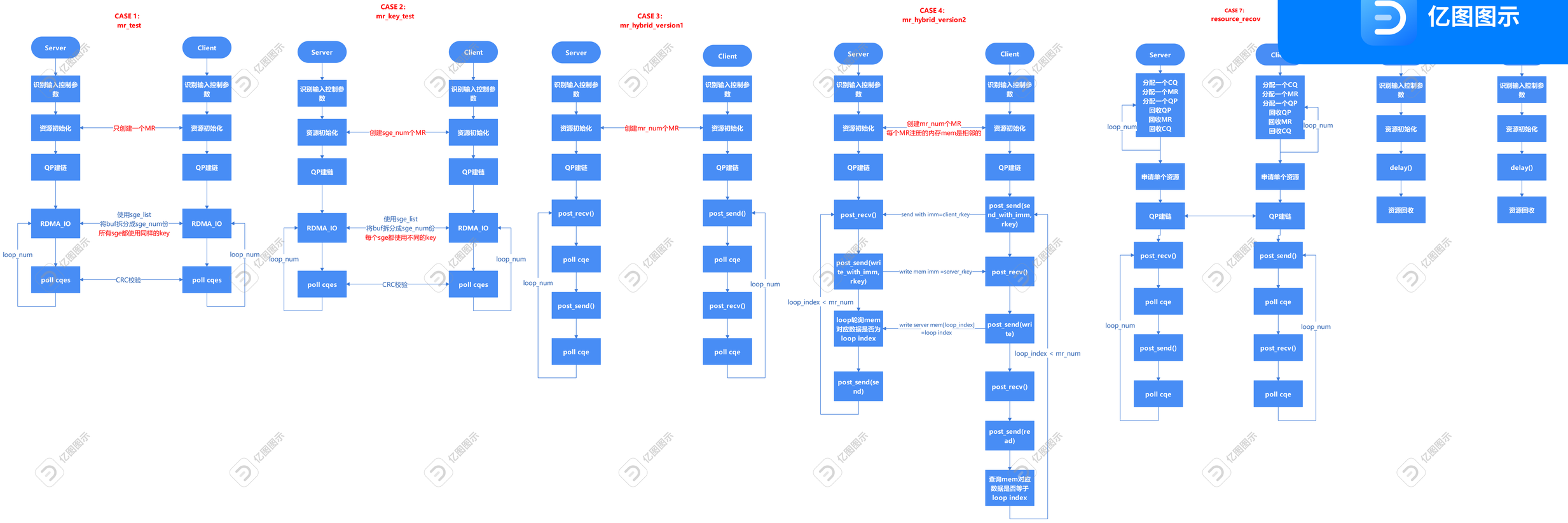

CASE 1:
mr_test
Server
Client
识别输入控制参数
资源初始化
只创建一个MR
QP建链
RDMA_IO
使用sge_list
将buf拆分成sge_num份
所有sge都使用同样的key
loop_num
poll cqes
CRC校验
CASE 2:
mr_key_test
创建sge_num个MR
使用sge_list
将buf拆分成sge_num份
每个sge都使用不同的key
CASE 3:
mr_hybrid_version1
创建mr_num个MR
post_recv()
poll cqe
post_send()
CASE 4:
mr_hybrid_version2
创建mr_num个MR
每个MR注册的内存mem是相邻的
send with imm=client_rkey
post_send(write_with_imm, rkey)
write mem imm =server_rkey
loop_index < mr_num
loop轮询mem对应数据是否为loop index
write server mem[loop_index] =loop index
post_send(send)
post_send(send_with_imm, rkey)
post_recv()
post_send(write)
post_send(read)
查询mem对应数据是否等于loop index
CASE 7:
resource_recov
分配一个CQ
分配一个MR
分配一个QP
回收QP
回收MR
回收CQ
申请单个资源
delay()
资源回收

Table 77 Post Send Bind WR Rules

MW Type	MW State	PD Rule
Type 2A	Free	PD associated with R_Key in the Post Send Bind WR must be the same as the PD associated with the QP on which the WR was posted.
Type 2B	Free	

Table 79 Type 2 Memory Window Access Rules

MW Type	MW State	QP Rule	PD Rule
Type 2A	Valid	MW access operations only allowed via QP on which MW was bound	N/A
Type 2B	Valid	MW access operations only allowed via QP on which MW was bound	MW access operations only allowed if the PD associated with the R_Key is the same as the PD associated with the QP

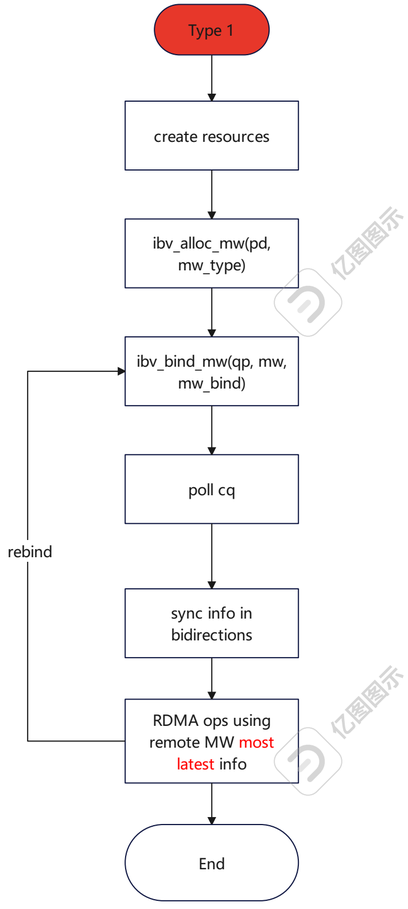
Table 78 Type 2 Memory Window Invalidation Rules

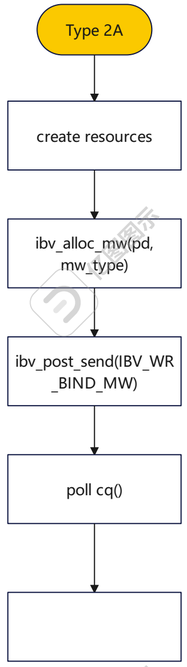
MW Type	MW State	QP Rule	PD Rule
Type 2A	Valid	Invalidate operation only allowed via QP on which MW was bound	N/A
Type 2A	Free	N/A	Invalidate operation only allowed via QP associated with same PD as MW
Type 2B	Valid	Invalidate operation only allowed via QP on which MW was bound	
Type 2B	Free	N/A	

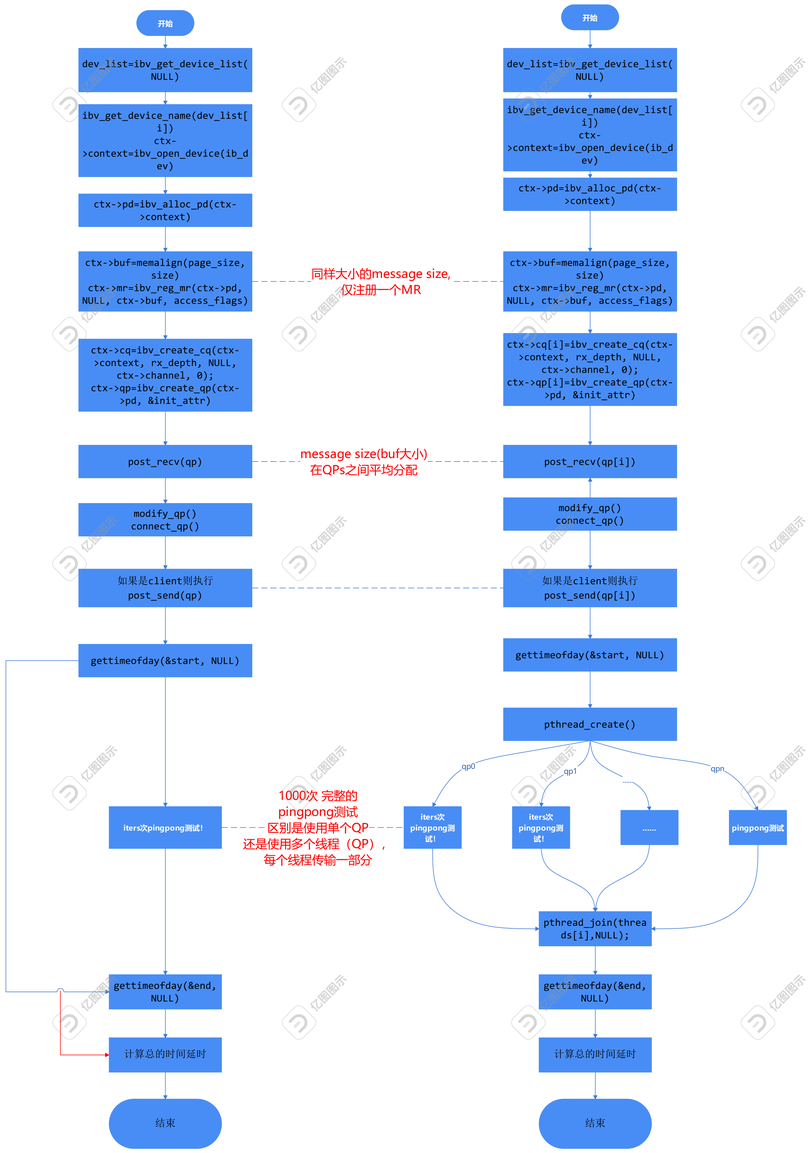
Table 80 Memory Windows Invalidation Rules

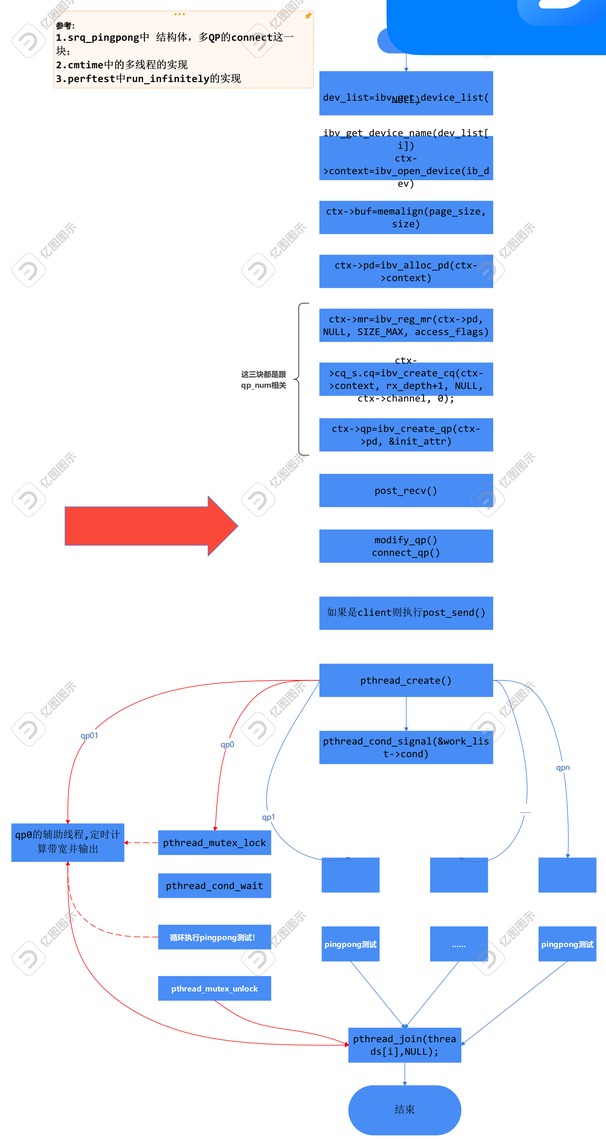
MW Type	Key Ownership	Local and Remote Invalidation	Bind using already Bound MW
Type I	CI	Not Allowed	Allowed
Type II	Consumer	Allowed	Not Allowed

区别点	Type 1	Type 2A	Type 2B
Bind method	Bind MW verbs	Post Send Bind MW WR	Post Send Bind MW WR
Post Send Bind WR Rules	-	Free state	Free state
access rules	PD	QP	PD + QP
R_key ownership	CI 返回R_key值	consumer 自己拼凑R_key	consumer 自己拼凑R_key
Bind using already bound MW	Allowed 绑定后之前的R_Key自动失效	Not Allowed 绑定前需要先使之前的R_Key无效化	Not Allowed 绑定前需要先使之前的R_Key无效化
是否支持零长度	是 通过零长度的方式使得mw的bindings失效	否	否
Local and Remote Invalidation operation	Not Allowed 采用deallocate mw, 或者绑定到新的区域。	Allowed	Allowed
关联的QP是否可以被销毁	-	Not Allowed	Allowed

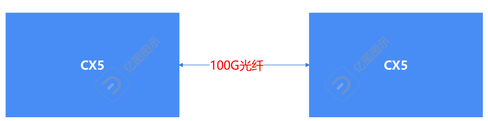





开始
dev_list=ibv_get_device_list(NULL)
ibv_get_device_name(dev_list[i])
ctx->context=ibv_open_device(ib_dev)
ctx->pd=ibv_alloc_pd(ctx->context)
ctx->buf=memalign(page_size, size)
ctx->mr=ibv_reg_mr(ctx->pd, NULL, ctx->buf, access_flags)
ctx->cq=ibv_create_cq(ctx->context, rx_depth, NULL, ctx->channel, 0);
ctx->qp=ibv_create_qp(ctx->pd, &init_attr)
post_recv(qp)
modify_qp()
connect_qp()
如果是client则执行
post_send(qp)
gettimeofday(&start, NULL)
iters次pingpong测试!
gettimeofday(&end, NULL)
计算总的时间延时
结束
同样大小的message size,
仅注册一个MR
message size(buf大小)
在QPs之间平均分配
1000次 完整的
pingpong测试
区别是使用单个QP
还是使用多个线程（QP），
每个线程传输一部分
开始
dev_list=ibv_get_device_list(NULL)
ibv_get_device_name(dev_list[i])
ctx->context=ibv_open_device(ib_dev)
ctx->pd=ibv_alloc_pd(ctx->context)
ctx->buf=memalign(page_size, size)
ctx->mr=ibv_reg_mr(ctx->pd, NULL, ctx->buf, access_flags)
ctx->cq[i]=ibv_create_cq(ctx->context, rx_depth, NULL, ctx->channel, 0);
ctx->qp[i]=ibv_create_qp(ctx->pd, &init_attr)
post_recv(qp[i])
modify_qp()
connect_qp()
如果是client则执行
post_send(qp[i])
gettimeofday(&start, NULL)
pthread_create()
qp0
qp1
.....
qpn
iters次pingpong测试!
iters次pingpong测试!
.....
pingpong测试
pthread_join(threads[i],NULL);
gettimeofday(&end, NULL)
计算总的时间延时
结束


参考:
1.srq_pingpong中 结构体，多QP的connect这一块;
2.cmtime中的多线程的实现
3.perftest中run_infinitely的实现
dev_list=ibv_get_device_list(
ibv_get_device_name(dev_list[i])
ctx->context=ibv_open_device(ib_dev)
ctx->buf=memalign(page_size, size)
ctx->pd=ibv_alloc_pd(ctx->context)
这三块都是跟qp_num相关
ctx->mr=ibv_reg_mr(ctx->pd, NULL, SIZE_MAX, access_flags)
ctx->cq_s.cq=ibv_create_cq(ctx->context, rx_depth+1, NULL, ctx->channel, 0);
ctx->qp=ibv_create_qp(ctx->pd, &init_attr)
post_recv()
modify_qp()
connect_qp()
如果是client则执行post_send()
pthread_create()
qp01
qp0
qp1
qpn
qp0的辅助线程,定时计算带宽并输出
pthread_mutex_lock
pthread_cond_wait
循环执行pingpong测试!
pthread_mutex_unlock
pingpong测试
.....
pingpong测试
pthread_cond_signal(&work_list->cond)
pthread_join(threads[i],NULL);
结束


pthread_cond_signal(&work_list->cond)
pthread_mutex_lock
pthread_cond_wait
pthread_mutex_unlock


CX5
100G光纤
CX5