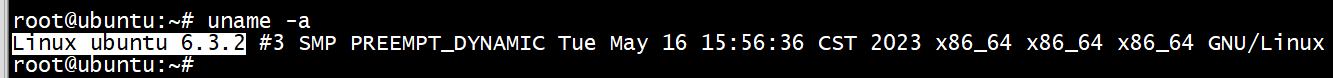
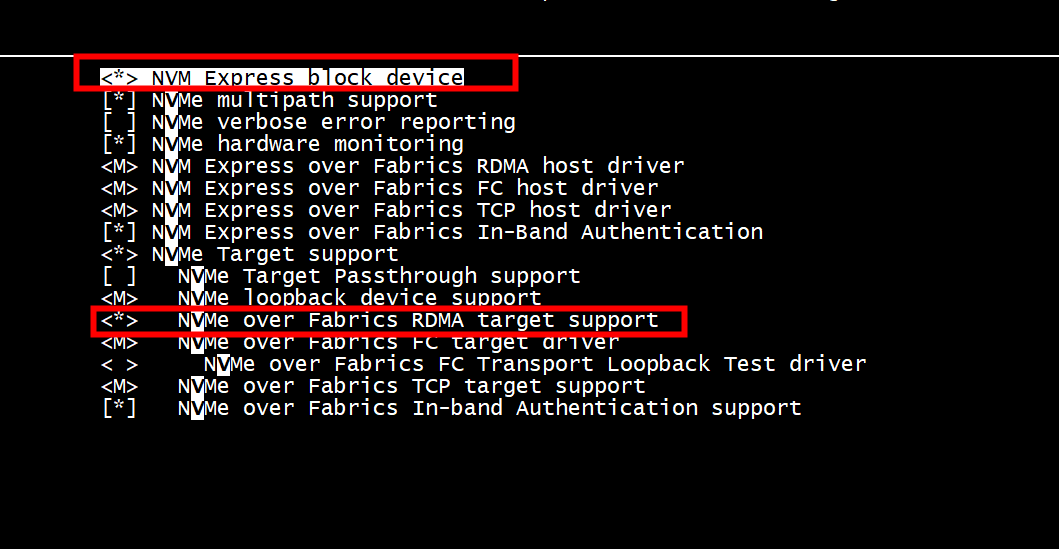
# 内核版本



## 内核config



### Debug config



# NVMeoF

## NVMeoF 队列结构

NVMeoF 将 NVMe 队列映射到 RDMA 队列（Queue Pairs）上，映射过程如图

所示：

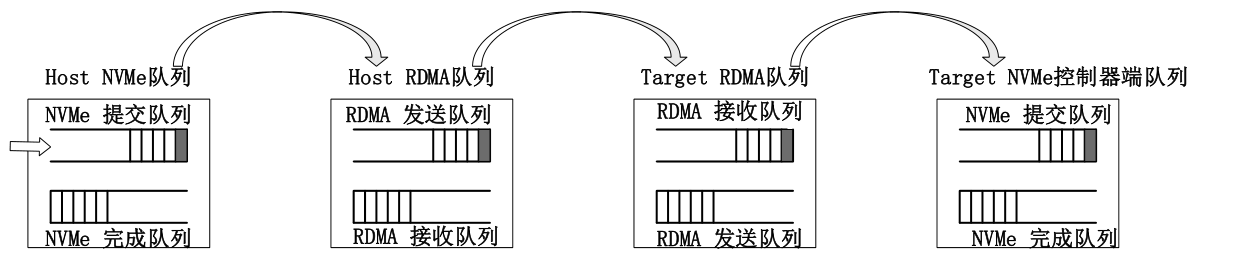
Host 端 RINC 将 NVMe SQE（Submission Queue Entry）封装成网络命令包，

作为 RDMA 报文的内容（payload），通过 RDMA 传输到 Target 端；Target 端的

RNIC 解包后，将 SQE 和数据放入 Target 端主机的内存中，随后 Target 处理 NVMe

命令和数据；处理完之后，将 NVMe CE（Completion Entry）封装成 RDMA 响应

（Response）包，放入 RDMA 完成队列中，返回给 Host 端。



NVMeoF 的提交队列（SQ）和完成队列（CQ）是基于消息的，而不是 NVMe

中基于内存的队列。因此，在 NVMeoF 中，不再使用门铃寄存器。在 NVMeoF 中，

Host 和 NVMe 控制器之间传输封装的通道是单向的。Host 通过 Connect 命令在建

立连接之初就成对的创建了 SQ 和 CQ。Host 利用 SQ 将命令封装发送给控制器，

NVMe 控制器利用 CQ 将应答封装返回 Host。

在 NVMeoF 中，SQ 的尾指针位于本地的 Host 端，对 NVMe 控制器是透明的。

NVMe 传输层负责传输命令封装，并在封装到达之后通知控制器。SQ 的头指针由

NVMe 控制器进行维护更新，并将 SQ 头指针的位置封装在在应答封装中，这样

Host 就获知了当前 SQ 头的位置，并根据本地的尾指针的信息对 SQ 进行管理，判

断队列是否已满。当 NVMe 的控制器检测到 Host 向一个已经满了的 SQ 中提交命

令时，控制器将会停止命令的处理并报错。这样就构成了提交队列的简单流控。

与 SQ 队列不同的是，在 NVMeoF 协议中，CQ 队列没有头尾指针的存在。

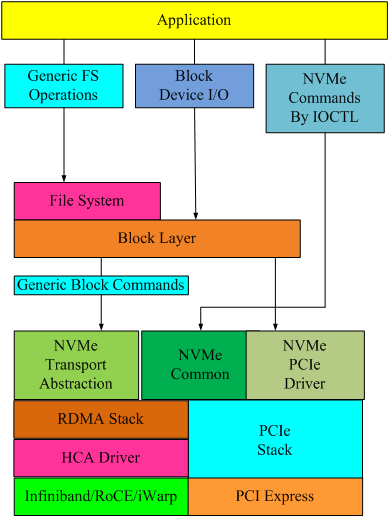
Host 应该配置好每个 CQ 的深度，以应对在某一时刻有大量来自 SQ 的命令同时完

成。实际操作过程中，当 NVMe 控制器已经在处理命令时，SQ 中仍存在大量潜在

的即将完成的命令，因此 CQ 队列的深度一般比对应 SQ 队列的深度大很多。如果

CQ 的队列深度比较小，那么存在同一时刻完成的大量命令将 CQ 写满的情况。

## Linux 的 NVMeoF 驱动



Host 端在共享命令的基础之上，同时实现了 NVMe over PCIe 和 NVMe over

Fabric，主要包括 Core、PCIe、Fabrics 三个模块。Host 既可以访问网络另一端挂

载的 NVMe 设备，也可以访问本地挂载的 NVMe 设备，各自的访问路径由各自的

非公有部分控制实现（图中的 NVMeoF 驱动和 NVMe PCIe 驱动）。Host 端的 NVMeoF 支持了多种 RDMA 传输协议，如 InfiniBand、RoCE

和 iWARP，对网络的适配性好；支持 Host 端和多个控制器进行动态的连接与断开

操作；提供了基础的多路径支持。

和 Host 端类似，Target 端也以类似的方式在 Linux 内核中实现了 NVMe 的控

制器，有 Core 和 Fabrics 模块，支持基于网络的访问和本地的访问。NVMe Target

支持绝大部分 Linux 块设备，例如 NVMe、SCSI、SATA、SAS 等，具有良好的兼

容性；利用块设备层和设备进行通信。Target 端不仅支持 NVMe 的基础命令，也

添加了对 DSM（Data Set Management）的支持，并且允许多主机、多命名空间以

及多控制；但是目前还不支持混合命令、安全发送和接收等功能。

基于 Linux 实现的 NVMeoF 具有以下特点：

1) Host 端和 Target 端之间通过标准的 RDMA 建立相互的连接；

2) 通过内核的 IB verbs 管理和控制数据的访问路径和资源的分配；

3) CQ 的处理过程和读写操作通过公有的 IB 栈完成；

4) NVMe 的队列映射到 RDMA 的队列中，NVMe 的 Admin 队列、I/O 队列

映射到 RDMA 的 QP、CQ 队列中；

5) 块设备层的 SGEs（Scatter/Gather Elements）映射到 RDMA 的 SGEs、内存区

域中；

6) NVMe 的命令通过 RDMA 的 SQ 队列进行传输；

7) 无论是 Host 端，还是 Target 端，驱动代码绝大部分是公共部分，相比 SCSI

的驱动实现，精简很多；

8) 核心部分

a) nvme-core：需要在本地端执行的 nvme 命令实现，与网络无关；

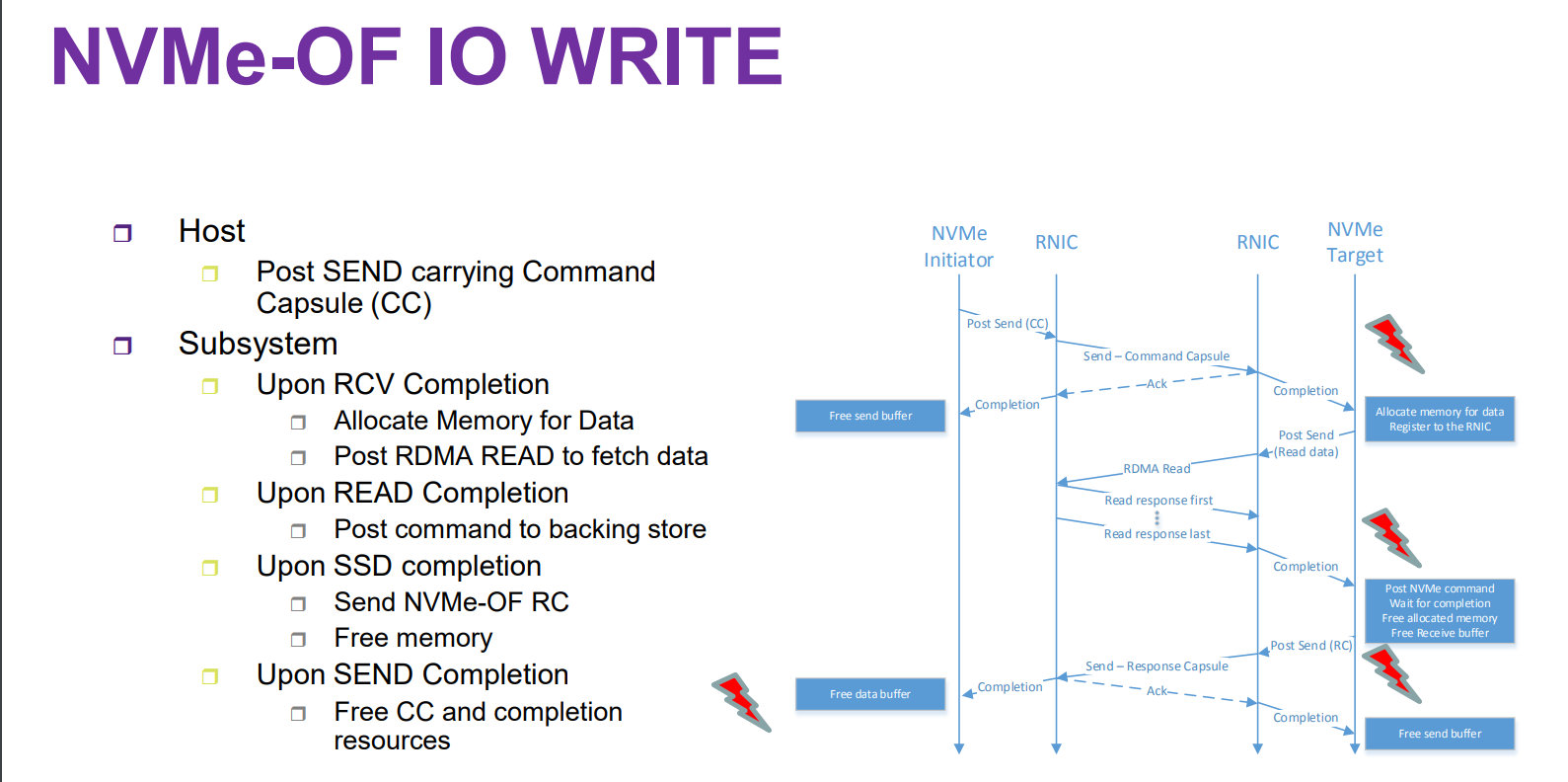
b) nvme-fabrics：实现网络中的独立传输。

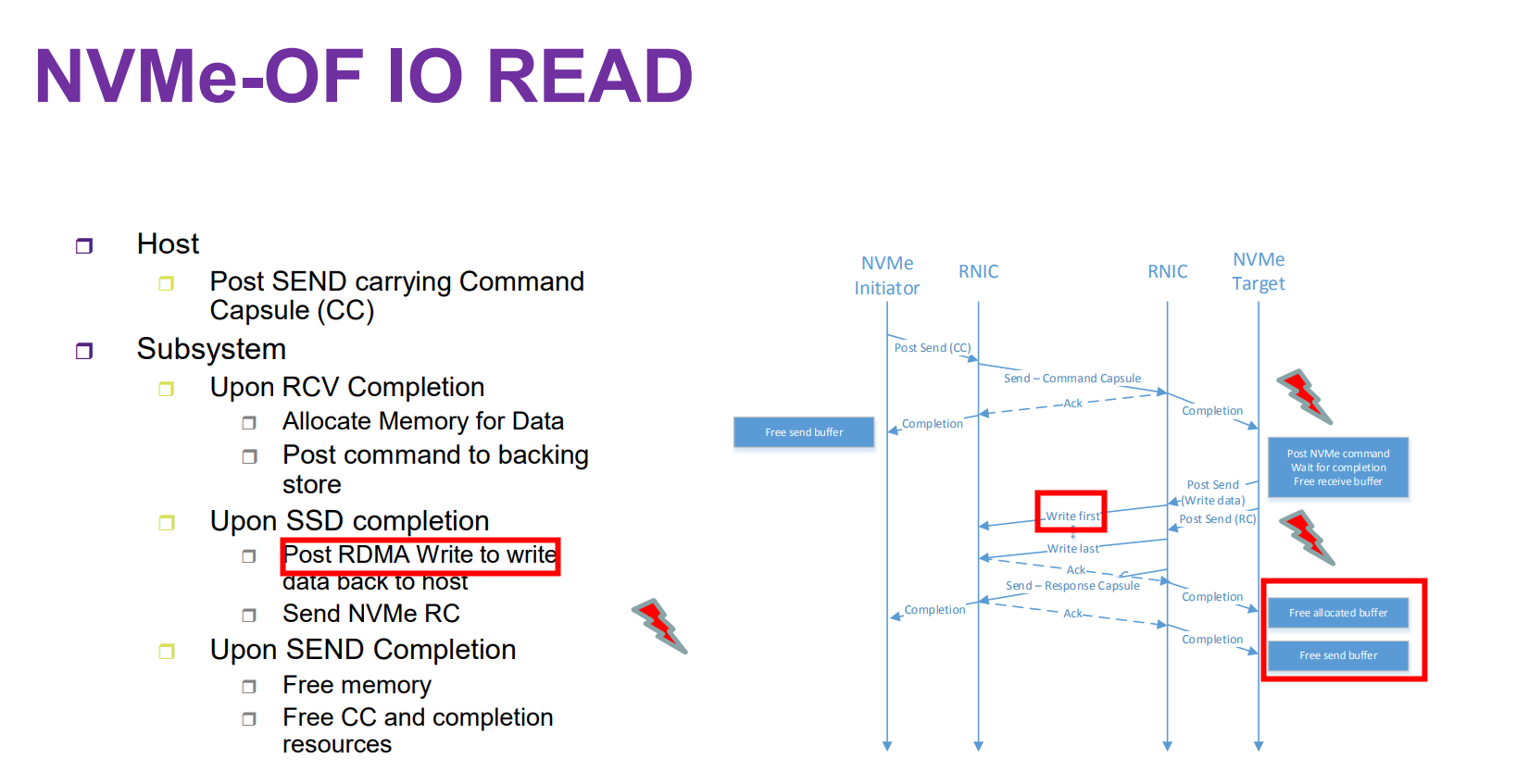
9) 驱动

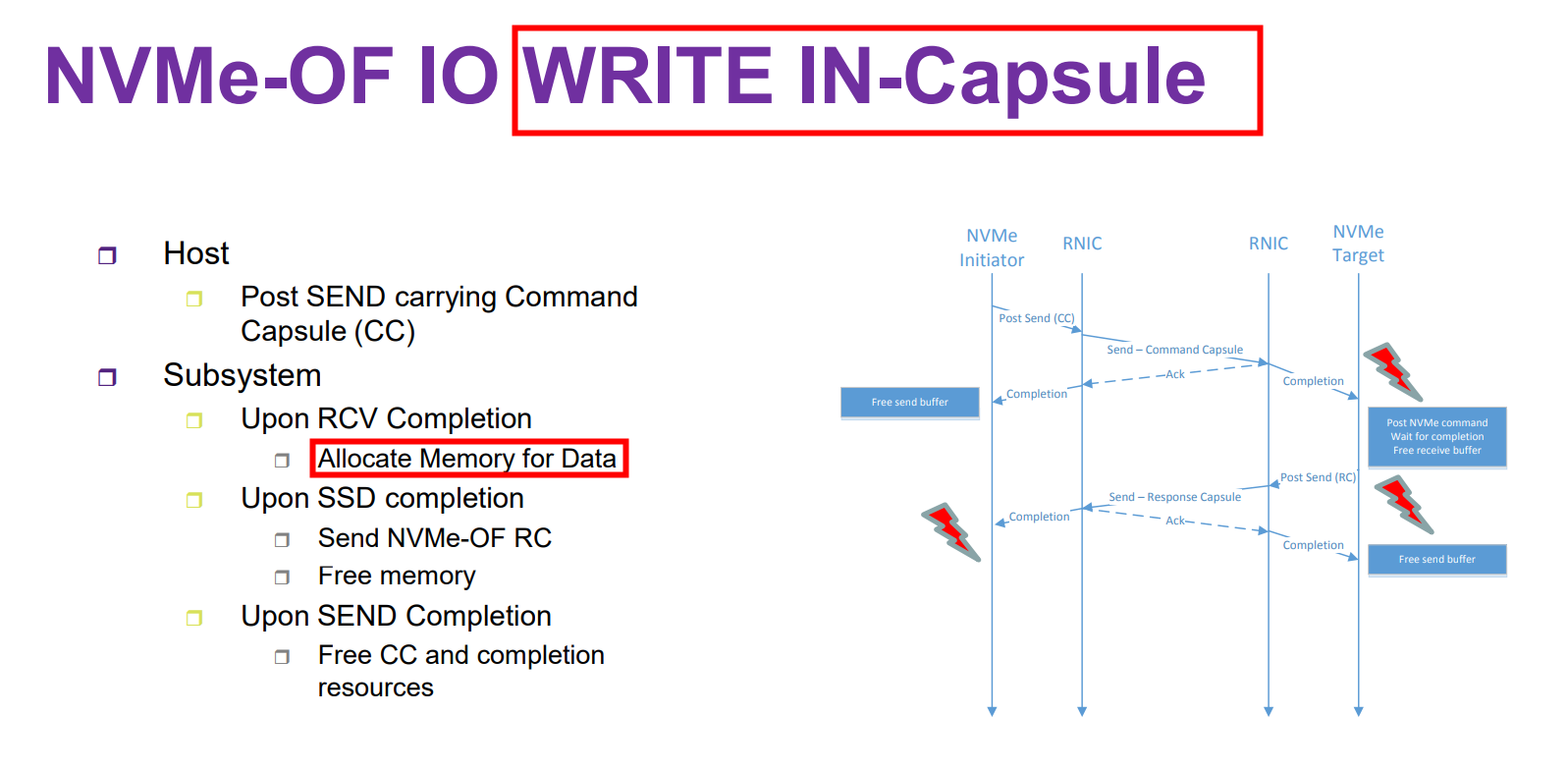
a) nvme-pci：Host 端的 PCIe 实现；

b) nvme-rdma：Host 端的 RDMA 实现。

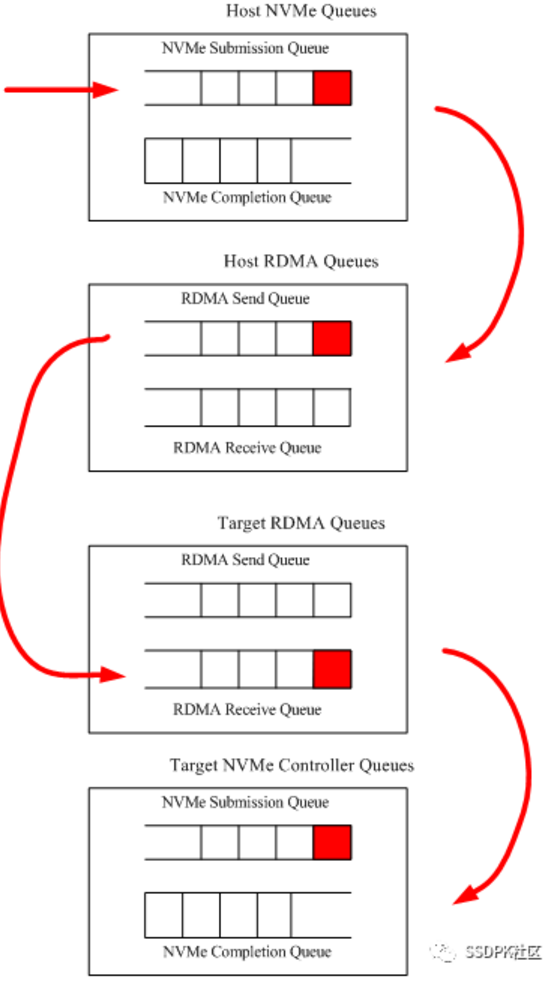
## NVMe over RDMA







首先，host的command被封装进capsule后压入Host NVMe Submission Queue；接着该capsule会被放入Host RDMA Send Queue变成RDMA\_SEND消息的消息负载；接着消息被host的RDMA网卡发包，当被target网卡接收后，capsule被放到target网卡的RDMA ReceIve Queue；接下来该capsule被放置到target端的内存中并由target处理该command；待处理完后，target生成回复信息（Respond Command）并封装进capsule然后将该回复capsule压入target端的NVMe Controller的Submission Queue，并经由target的网卡发送给host。这样，一次NVMe over RDMA的通信就完成了。



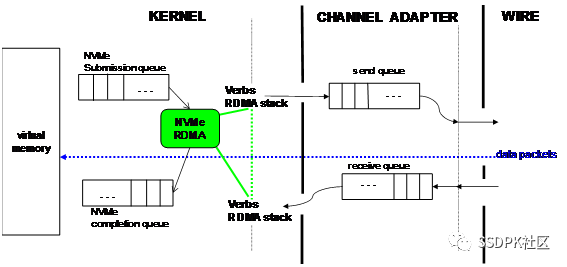
NVMe over RDMA读写文件原理

 下面以host读文件为例讨论数据流的传输。

事实上，在NVMe over Fabrics协议中，提出了两种数据传输方式，一种是将data附到capsule中，只要在传输command时负载需要传输的data即可。另一种则是直接的内存传输方式。这种方式原理上是将要传输的数据的地址和长度等元信息负载到capsule中。如用户发出read请求后，最终的capsule消息负载中会包含数据的存储地址、要读取的数据的长度以及要读到的内存地址。利用RDMA技术，这种传输的特性将表现的更加淋漓尽致。由Linux内核知识可知，虚拟文件系统（VFS）向上层提供统一的文件操作接口，当用户空间进程读取文件时便调用read API；接着VFS调用到真正的文件系统（例如Ext4等）真正的read函数，真正文件系统的作用是确定数据的位置，简言之就是根据用户要读取的文件、长度、偏移量等确定文件的逻辑块号（在真实的文件存储中，一个文件是被分割成若干块存储的）；接下来内核利用通用块层（Block Layer）启动IO操作来传送所请求的数据，通用块层为所有的块设备提供了一个抽象视图，隐藏了硬件块设备间的差异，而每次IO操作由一个“块I/O”结构（struct bio）的对象来描述。至此，所有的读文件操作都是这套同样的流程。由于NVMe或AHCI都是更底层的接口标准，因此差异从通用块层之下才开始。

此后，进入到NVMe Transport Abstraction（即NVMe over Fabrics协议的实现层）或者NVMe over PCIe，这时便会由此生成NVMe（base或over Fabrics）标准的Read Command，由NVMe Base标准对Read Command的定义，Read Command中包含了数据应该读到的内存区域的地址（Read Command的Data Pointer字段）、以及存储数据的逻辑块号的起始地址（Starting LBA字段）等其他在read操作中必需的字段（显然地，这些信息从上层通用块层传递的bio对象中获取），并将该Read Command封装进NVMe的通信单元capsule中。接着便进入到下一层的RDMA stack。RDMA Stack之下便是支持RDMA技术的网卡驱动，最底层便是不同技术（IB/RoCE/iWarp）实现的RDMA网卡。内核的NVMe RDMA层代码完成调用RDMA的接口verbs将该capsule封装为RDMA\_SEND消息的负载。Target的controller处理完read command后，数据流传输便开始了。数据流的传输则完全借助于RoCE技术。由于此前在target端注册了host的内存，接下来从target的SSD中读出的数据便直接封装为RDMA\_WRITE消息的负载（注意host的Read Command触发的却是target的RDMA\_WRITE操作），然后将这部分数据直接从target端写入到host的内存中，而写入的地址在Read Command中便已指明。该过程的实现得益于RDMA的数据零复制技术。

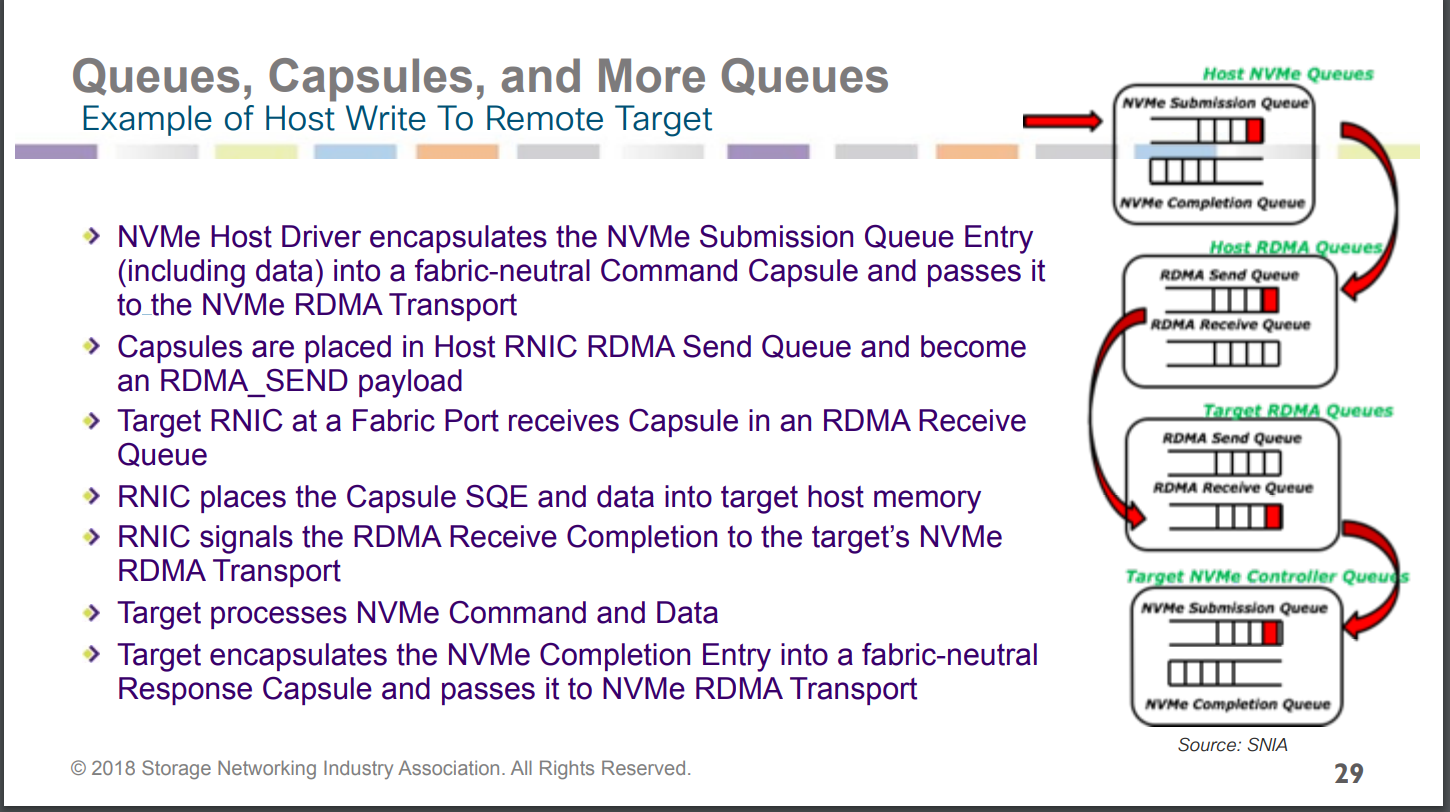
 以下是NVMe over RDMA读文件的过程



### Write to target

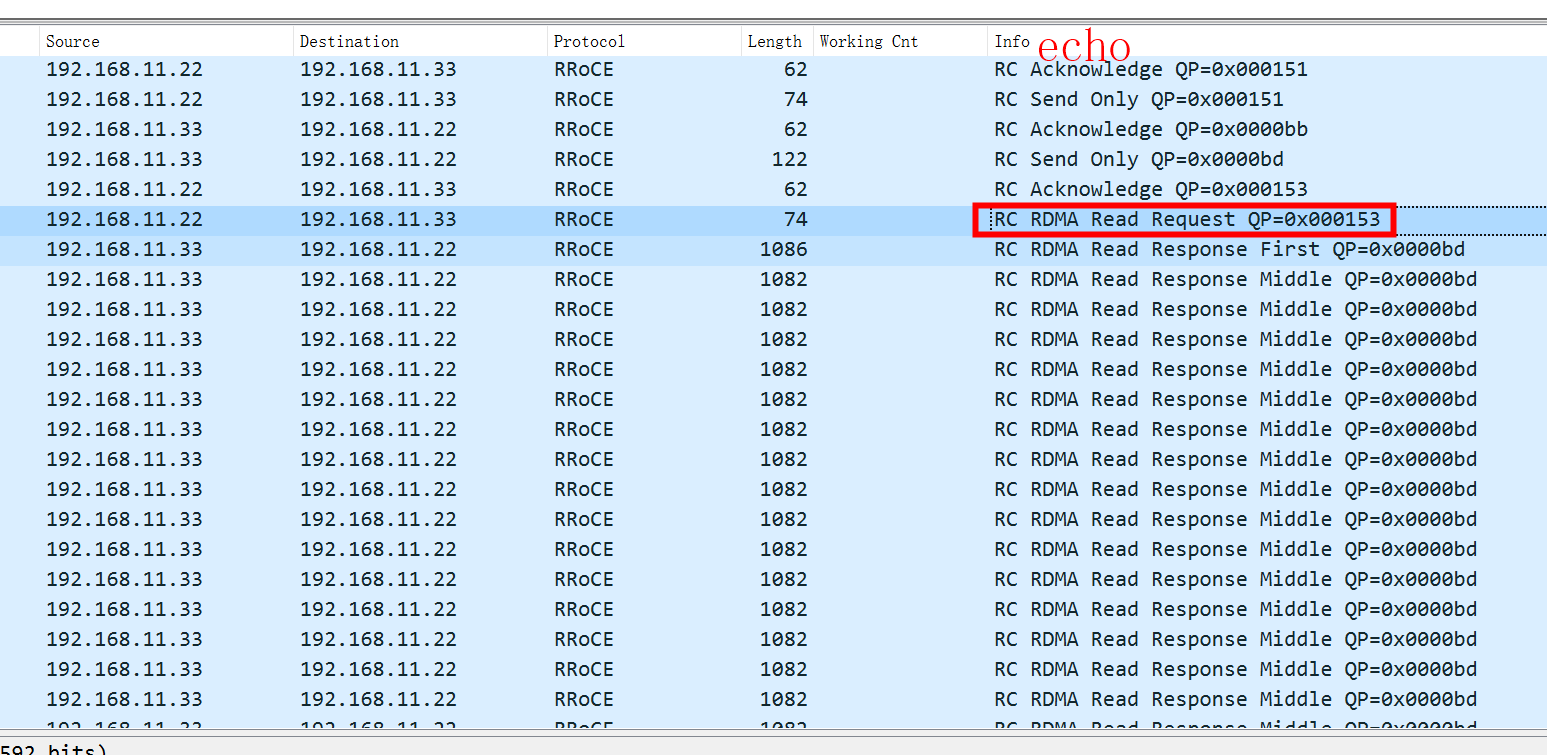
NVMe over Fabrics，以client的一个写请求的处理过程来展示NVMf如何利用RDMA技术。

1. NVMe Queue与Client端RDMA QP一一对应，把NVMe Submission Queue中的NVMe Command存放到RDMA QP注册的内存地址中（有可能带上I/O Payload），然后通过RDMA Send Queue发送出去；
2. 当Target的NIC收到Work Request后把NVMe Command DMA到Target RDMA QP注册的内存中，并在Targe的QP的Completion Queue中设置一个Work Completion。
3. Target处理Work Completion时，把NVMe Command发送到后端PCIe NVMe驱动，如果本次传输没有带上I/O Payload，则使用RDMA Read获取；
4. Target收到PCIe NVMe驱动处理结果后通过QP的Send Queue把NVMe Completion结果返回给client。



Host 执行

echo '123446fffffffffffffffffffffffffffffffffffffffffff' > /work/data/hello.txt



## [Wireshark support for NVMe over Fabrics](https://support.mellanox.com/s/article/wireshark-support-for-nvme-over-fabrics)

Packet Format

1. Here are the expected headers seen on the wire for NVMe over RDMA over UDP/IP.

Ethernet

IP

UDP

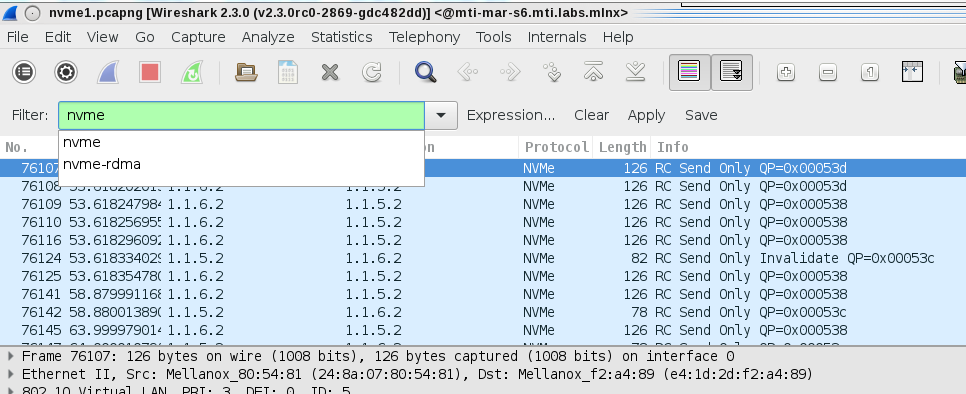
InfiniBand - this header includes the BTH, and may have other sub-headers such as RETH, IETH.

NVMe-oF - this header encapsulates the NVMe command capsule.

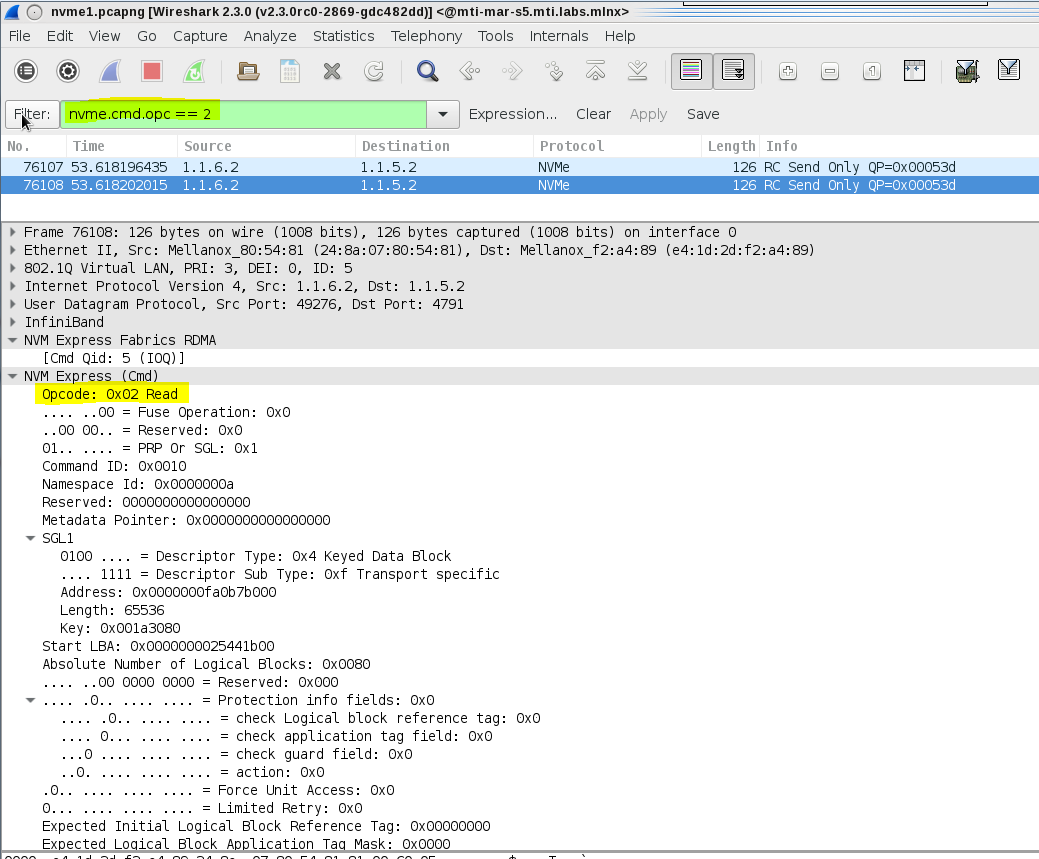
NVMe



2. Once Wireshark is opened, start NVMe-oF recording. Use the Filter options to filter the traffic to nvme or nvme-rdma headers.



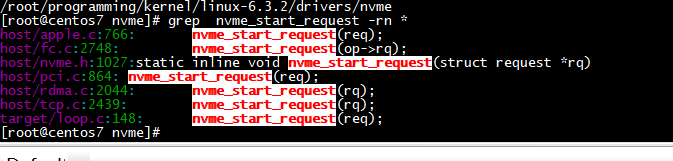
3. Here is an NVMe read command example. Use the filter nvme.cmd.opc ==2 for NVMe read.



# Nvme 通用函数

|  |  |
| --- | --- |
|  | nvme\_start\_request |
|  | struct blk\_mq\_ops nvme\_rdma\_mq\_ops ：  nvme\_rdma\_queue\_rq |
|  |  |
|  |  |

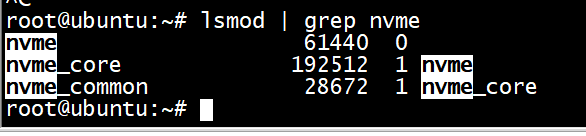
* nvme\_start\_request

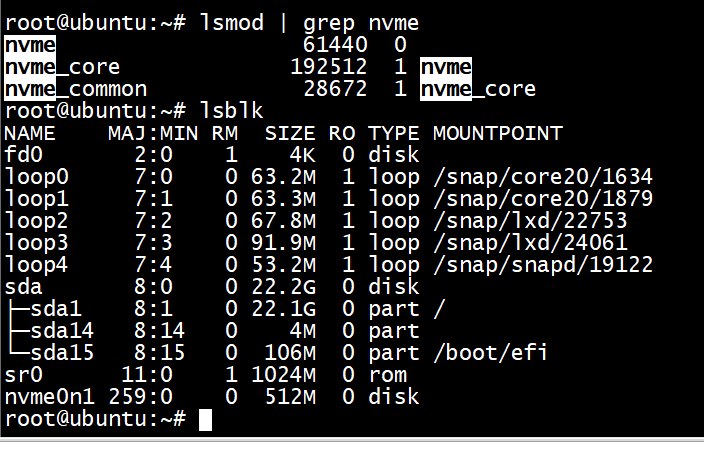


# Target

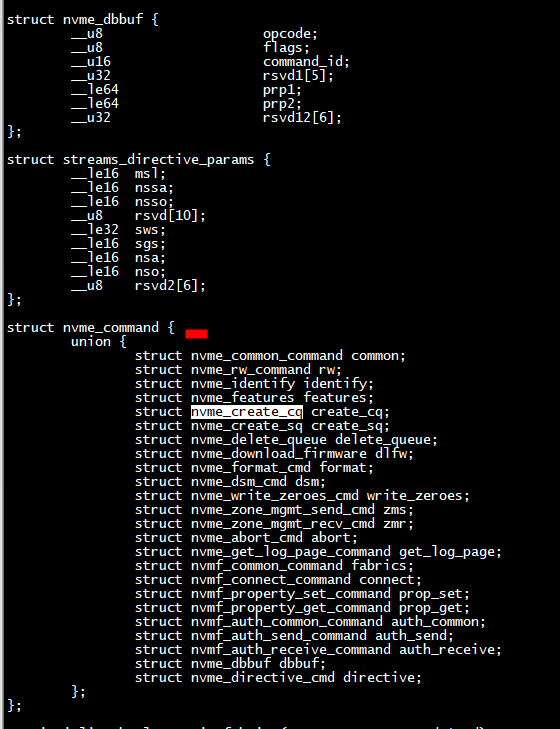
## Lsblk

有/dev/nvme0n1





## struct nvme\_command



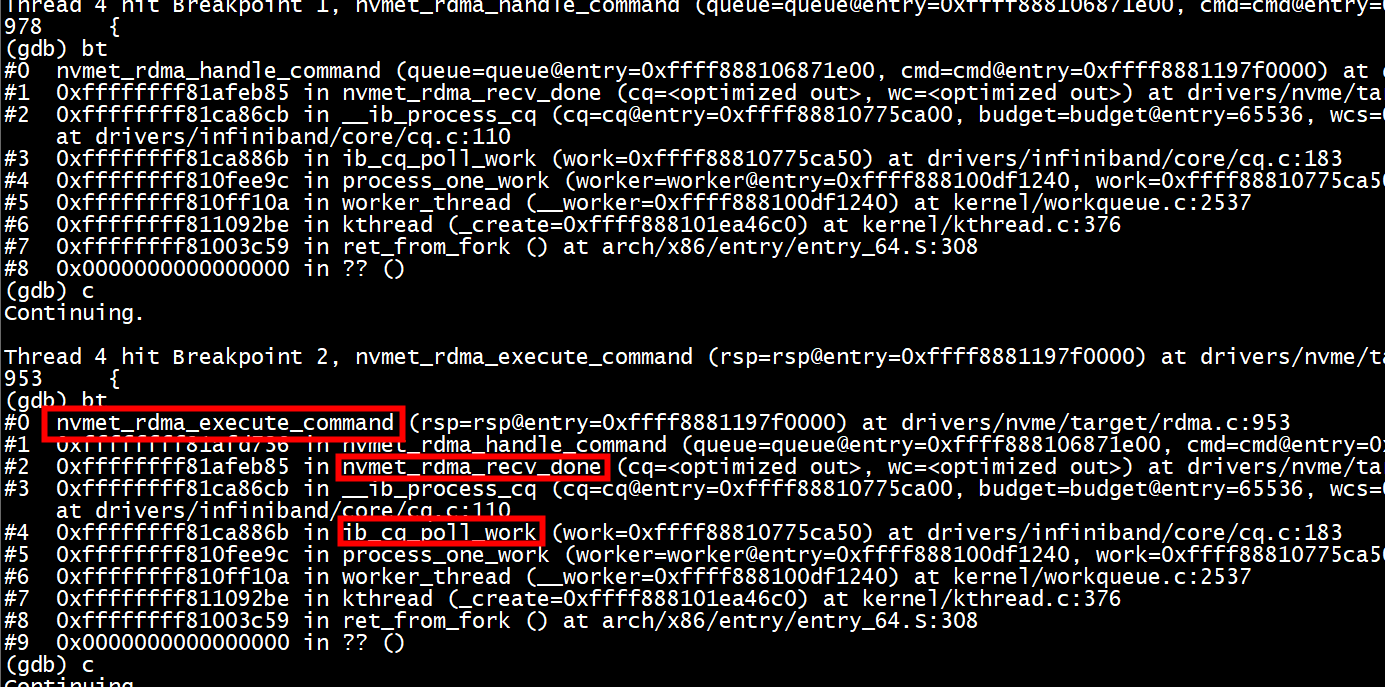
1. host调用nvme\_setup\_rw会设置read/write请求
2. Target 调用nvmet\_bdev\_execute\_rw执行read/write请求

## Qp

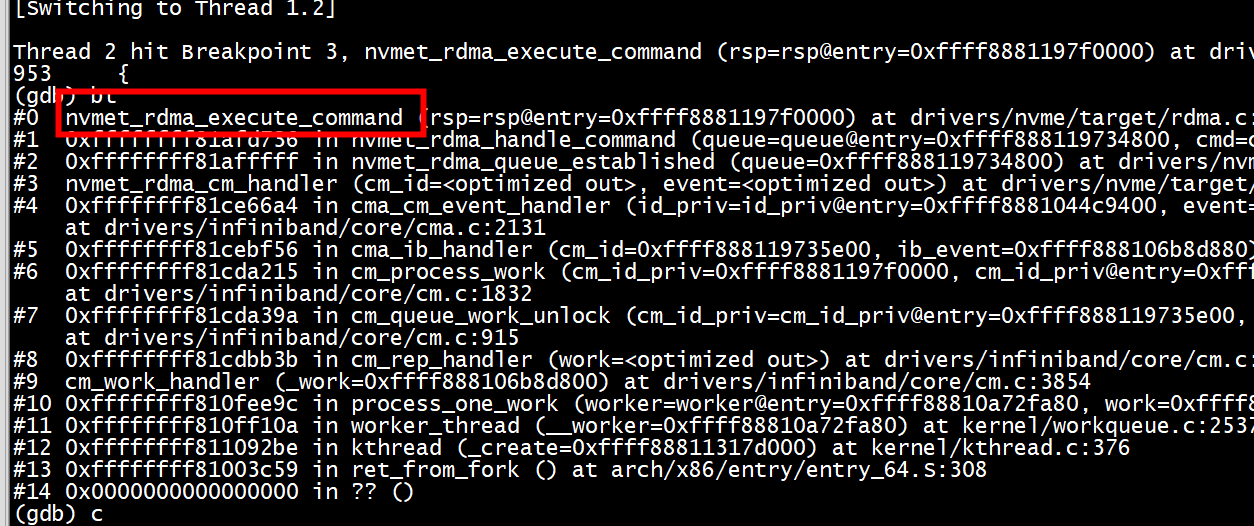
qp\_attr.event\_handler = nvmet\_rdma\_qp\_event

## nvmet\_rdma\_handle\_command

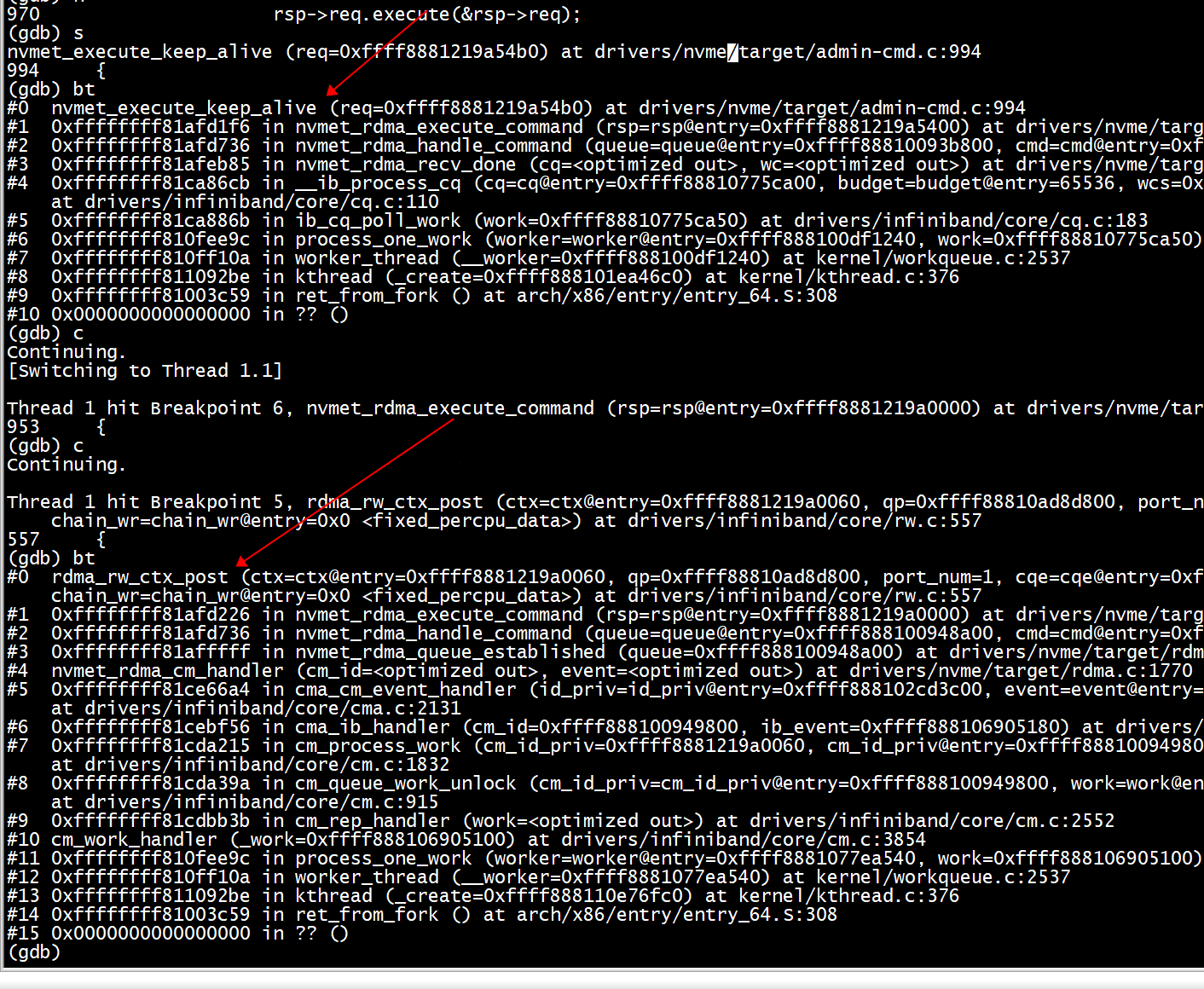
客户端执行nvme connect -t rdma -n data\_8 -a 192.168.11.22 -s 6600



路径2



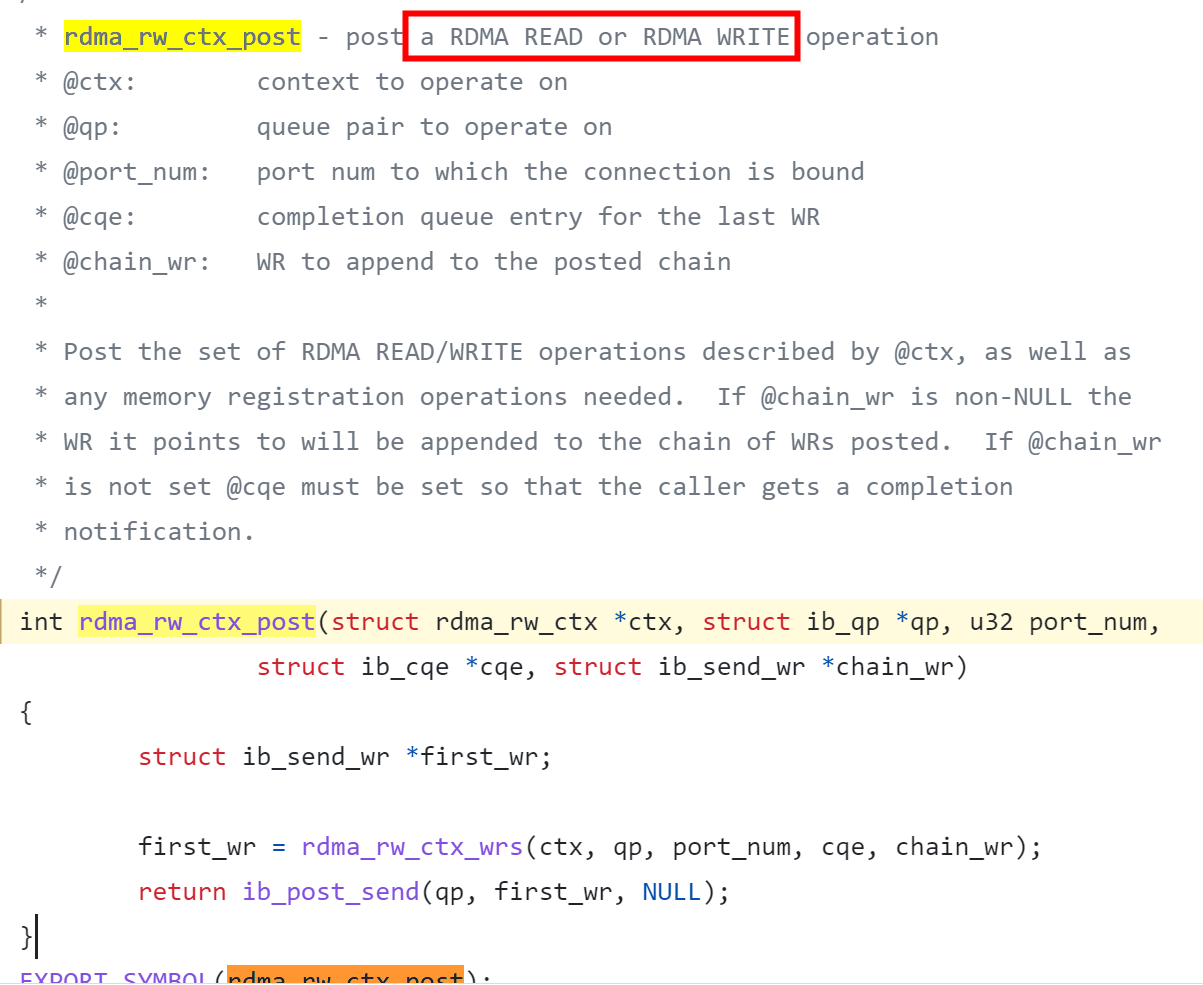
### nvmet\_execute\_keep\_alive

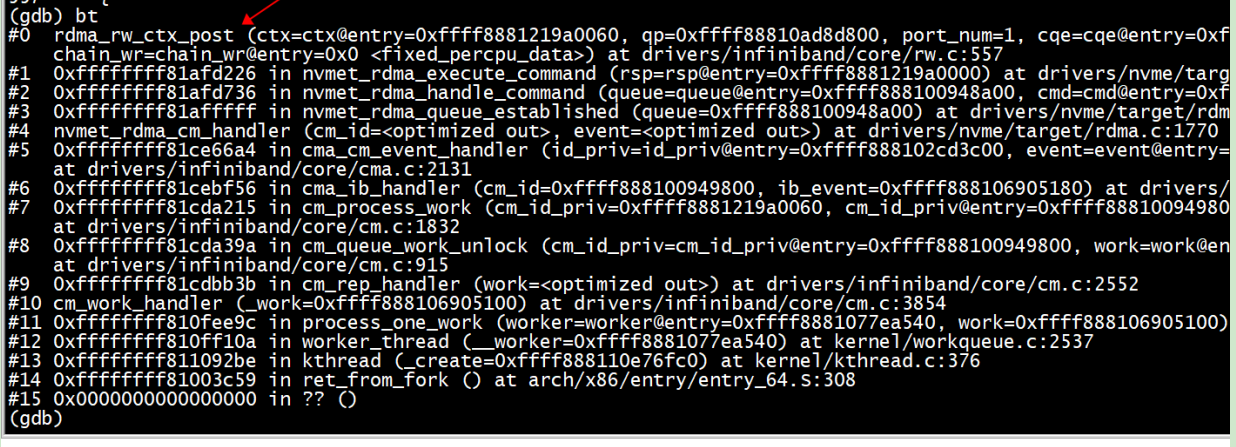


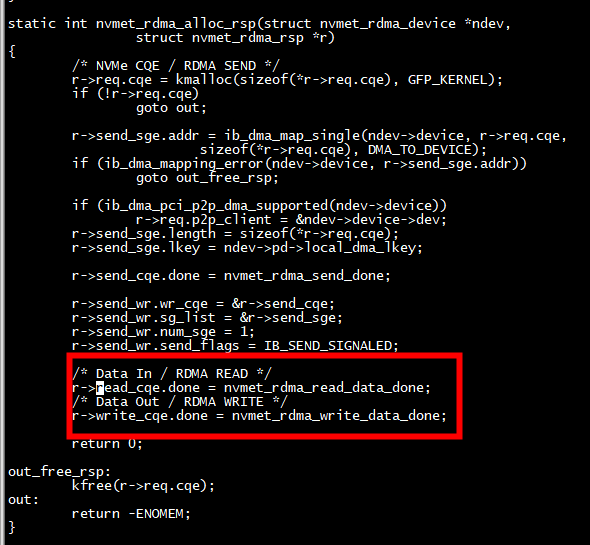
## Opcode

## **RDMA READ or RDMA WRITE**

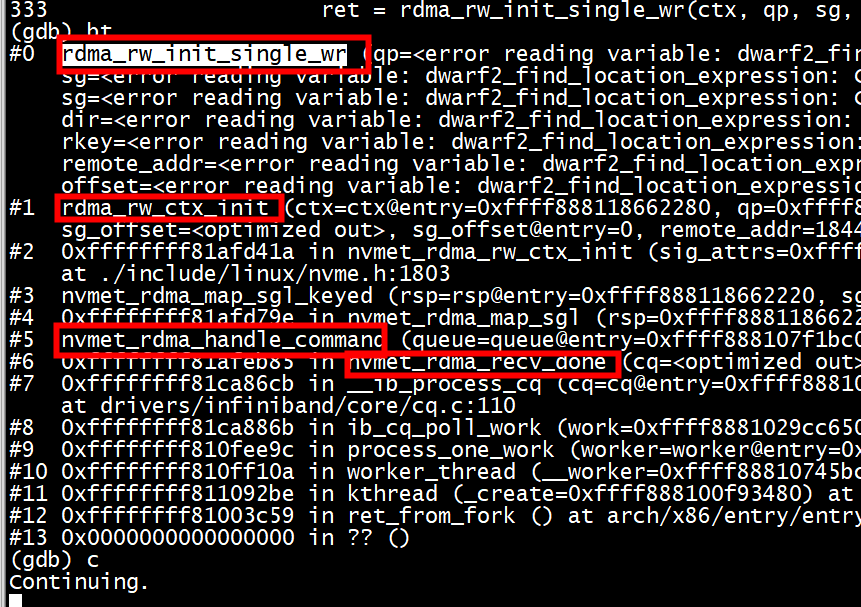
|  |  |
| --- | --- |
|  | rdma\_rw\_ctx\_wrs |
|  | 发送：rdma\_rw\_ctx\_post |
|  | rdma\_rw\_ctx\_init |

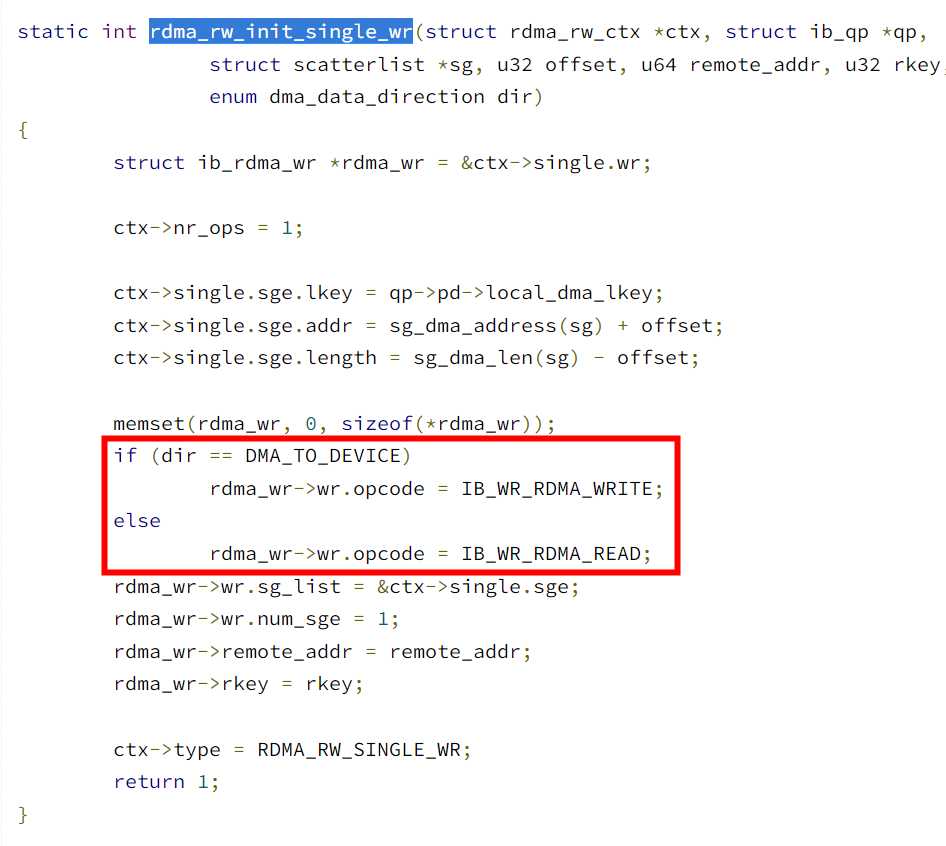






### rdma\_rw\_init\_single\_wr 初始化read/write

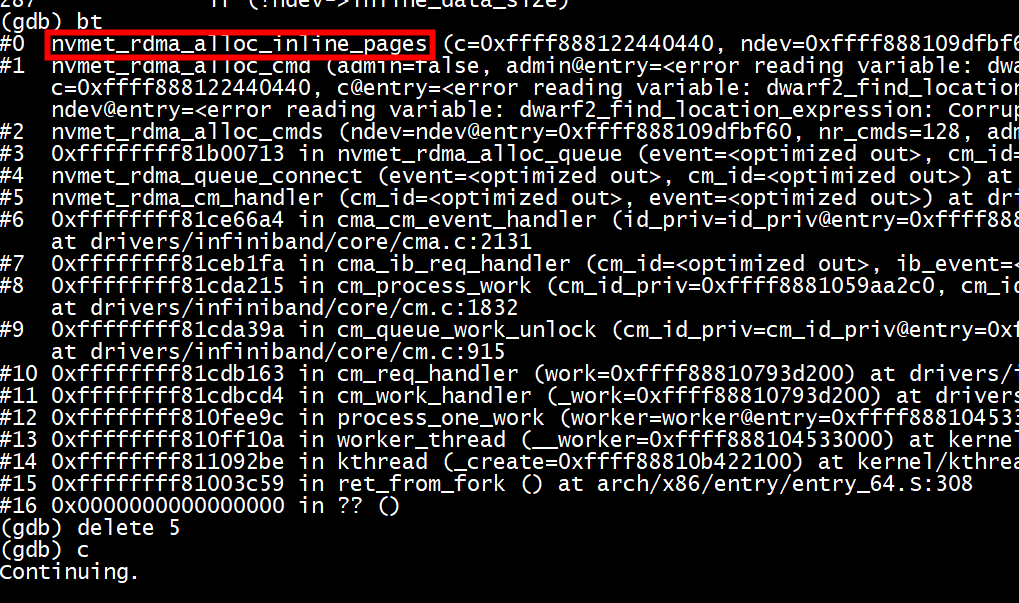




### IB\_WR\_REG\_MR

1. struct scatterlist \*sg的内存由 nvmet\_rdma\_alloc\_inline\_pages分配，
2. Key等于 ndev->pd->local\_dma\_lkey
3. rdma\_rw\_init\_single\_wr

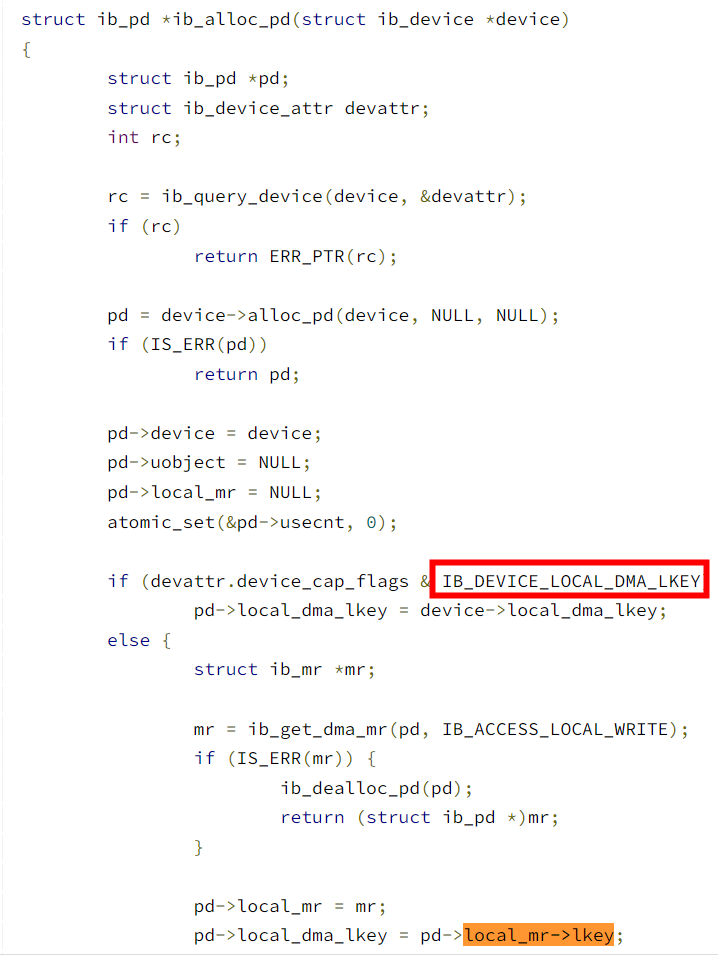
* ctx->single.sge.lkey = qp->pd->local\_dma\_lkey;
* ctx->single.sge.addr = sg\_dma\_address(sg) + offset



#### local\_dma\_lkey

A protection domain object provides an association between QPs, shared receive queues, address handles, memory regions, and memory windows.

Every PD has a local\_dma\_lkey which can be used as the lkey value for local memory operations。

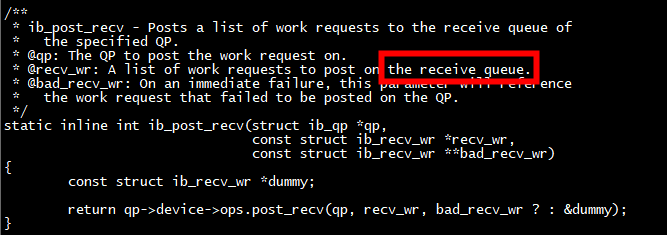


### Send and recv Opcode

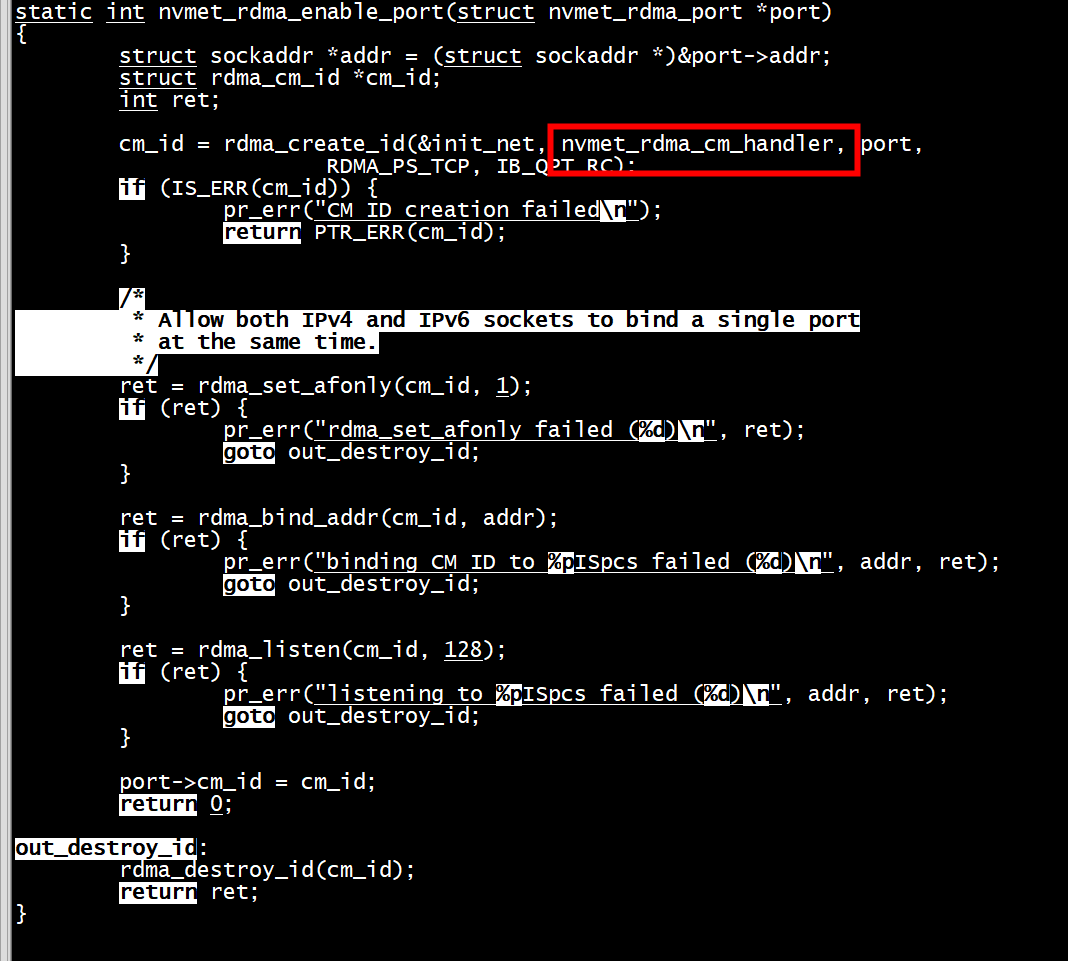
* IB\_WR\_SEND\_WITH\_INV、IB\_WR\_SEND

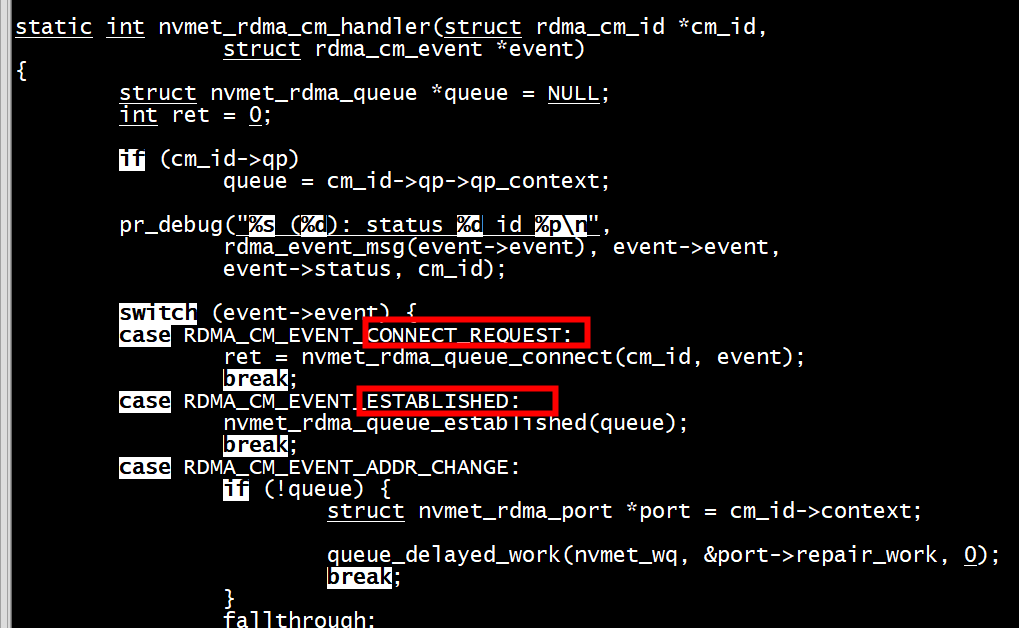
rxe\_post\_send 、 rxe\_post\_recv

* IB\_WC\_XX
* ib\_post\_recv



## rdma event





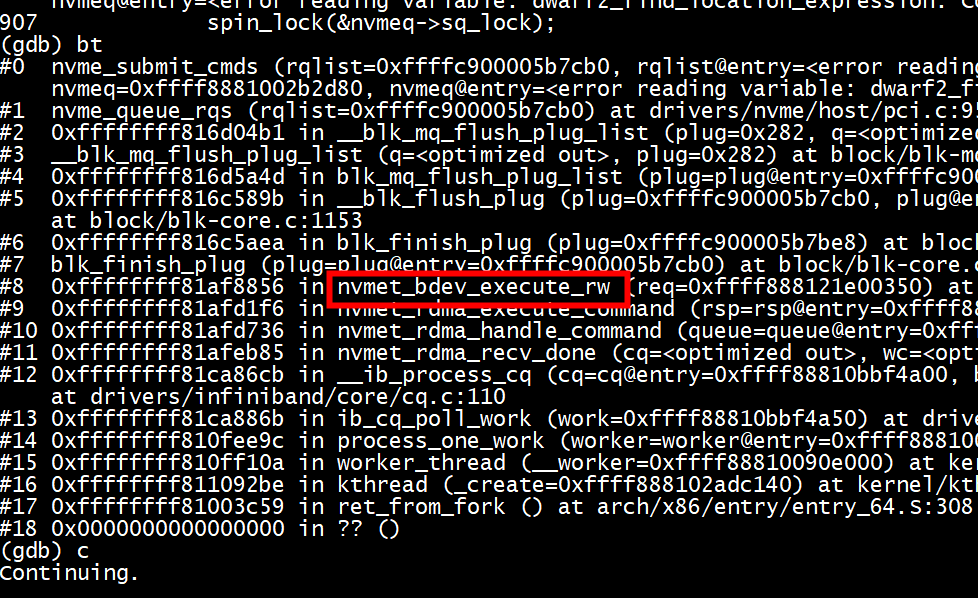
## nvme\_submit\_cmds

### Recv\_done

nvme\_rdma\_queue\_rq--> nvme\_setup\_cmd:设置命令nvme\_cmd\_write或nvme\_cmd\_read

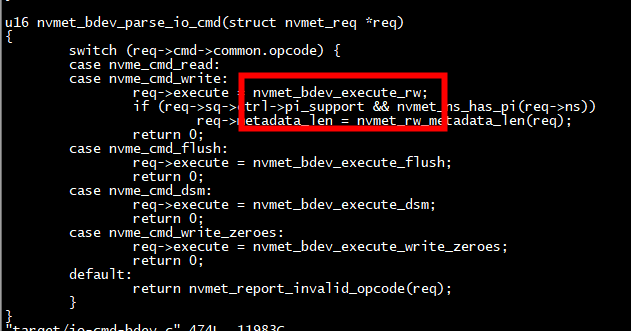
nvme\_rdma\_queue\_rq-->nvme\_rdma\_post\_send： wr.opcode = IB\_WR\_SEND

target/io-cmd-bdev.c:458: req->execute = nvmet\_bdev\_execute\_rw;



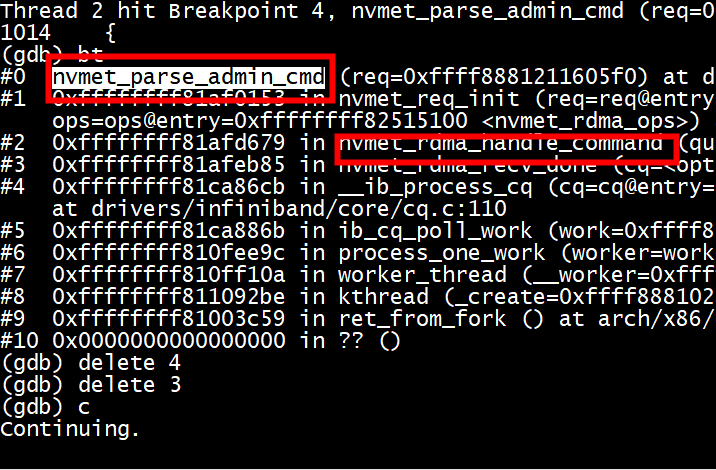
nvmet\_rdma\_execute\_command-->

nvmet\_req\_init--> nvmet\_parse\_io\_cmd--> nvmet\_bdev\_parse\_io\_cmd:



### nvmet\_parse\_admin\_cmd

nvme connect -t rdma -n data\_8 -a 192.168.11.22 -s 660



# Host

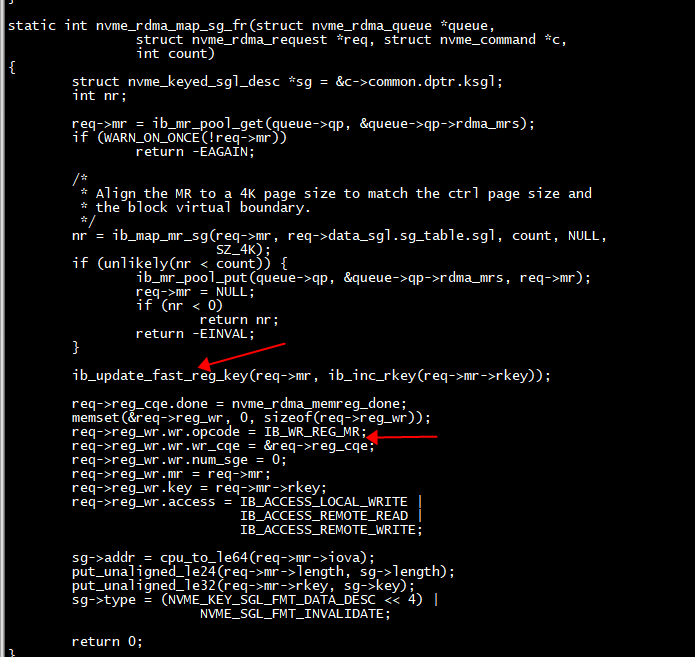
nvme\_rdma\_queue\_rq:

1. nvme\_setup\_cmd:设置命令nvme\_cmd\_write或nvme\_cmd\_read
2. nvme\_rdma\_post\_send： wr.opcode = IB\_WR\_SEND
3. nvme\_rdma\_map\_data->nvme\_rdma\_map\_sg\_fr：设置 wr->access =IB\_ACCESS\_REMOTE\_XX
4. nvme\_rdma\_post\_send

# Opcode

|  |  |
| --- | --- |
| wr->access | wr->access = IB\_ACCESS\_LOCAL\_WRITE |  IB\_ACCESS\_REMOTE\_READ |  IB\_ACCESS\_REMOTE\_WRITE; |
| wr.opcode | IB\_WR\_REG\_MR、IB\_WR\_SEND、ib\_post\_recv、IB\_WR\_REG\_MR\_INTEGRITY |
|  |  |

## IB\_WR\_REG\_MR



# 每个IO queue建立qp

1. nvme\_rdma\_alloc\_io\_queues

for (i = 1; i < ctrl->ctrl.queue\_count; i++) {

ret = nvme\_rdma\_alloc\_queue(ctrl, i,ctrl->ctrl.sqsize + 1);

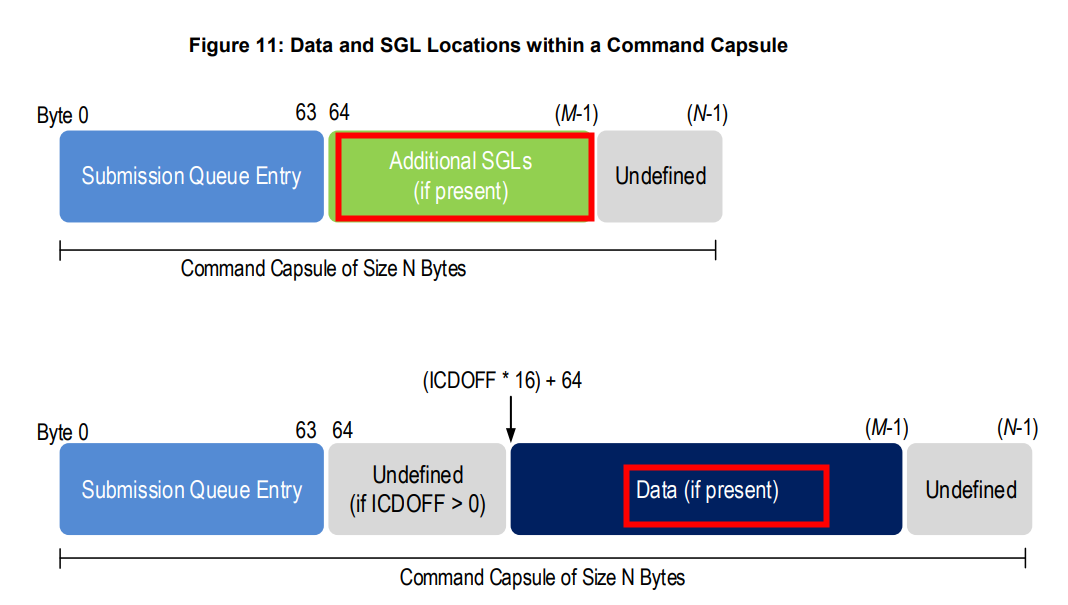
1. nvme\_rdma\_alloc\_queue

queue->cm\_id = rdma\_create\_id(&init\_net, nvme\_rdma\_cm\_handler, queue,

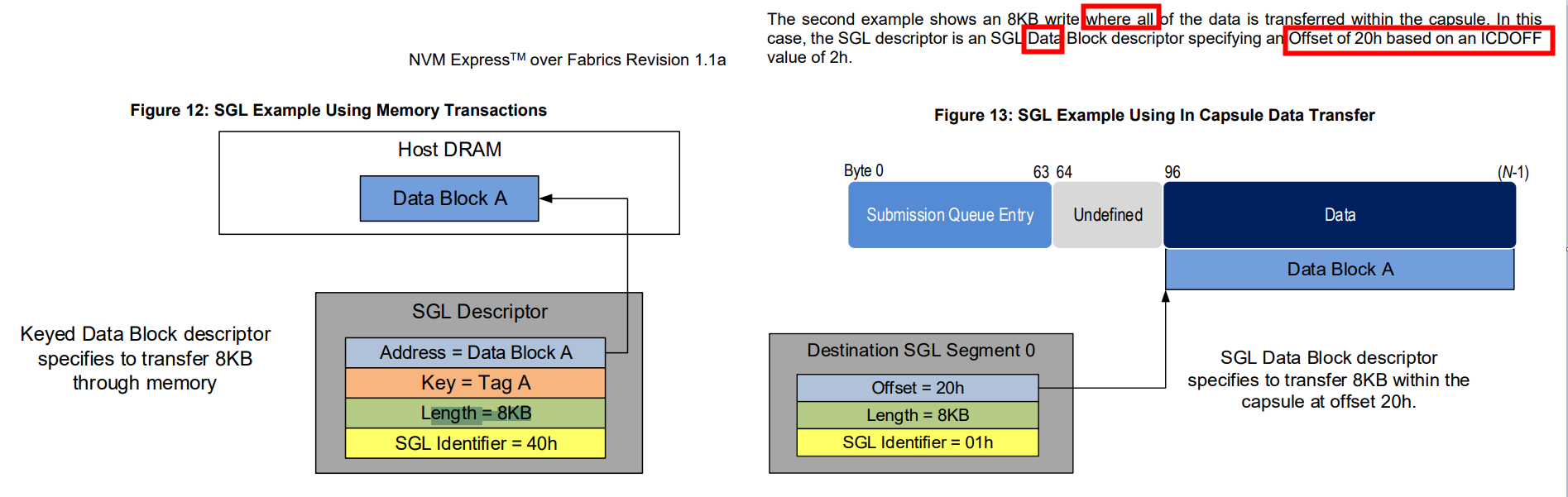
RDMA\_PS\_TCP, IB\_QPT\_RC);



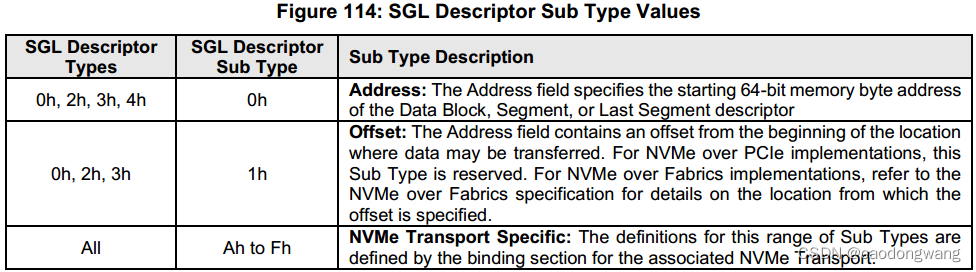
# Sgl



The data transfer examples in this section show SGL examples for a Write command where data is transferred via a memory transaction or within the capsule. The SGL may use a key as part of the data transfer depending on the requirements of the NVMe Transport used. The first example shows an 8KB write where all of the data is transferred via memory transactions. In this case, there is one SGL descriptor that is contained within the Submission Queue Entry at CMD.SGL1. The SGL descriptor is a Keyed SGL Data Block descriptor. If more SGLs are required to complete the command, the additional SGLs are contained in the command capsule.

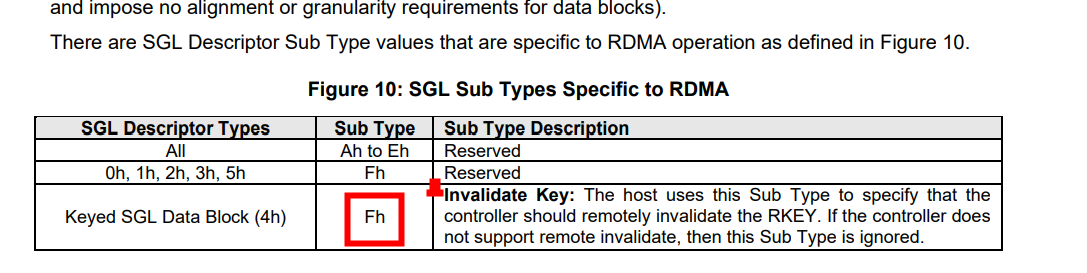


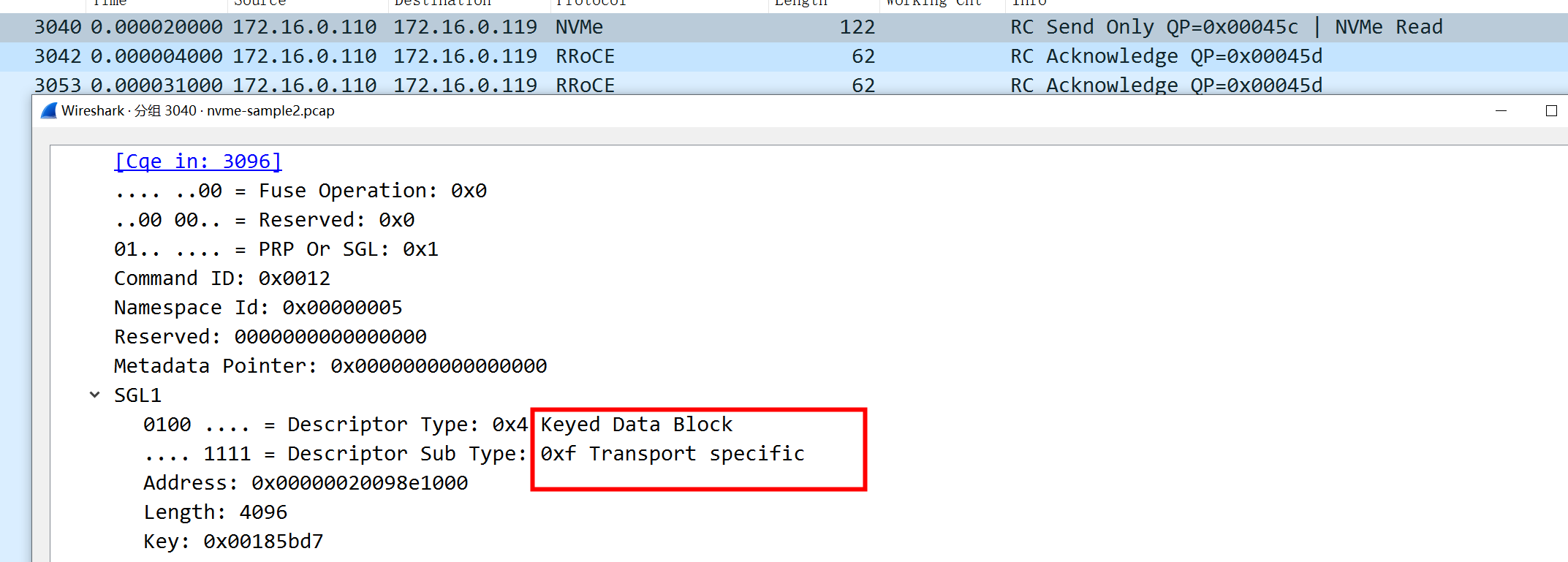
## Subtype



对于NVMe over PCIe来说，这个sub type是保留的，或者说默认值为0。sub type主要服务于NVMe over fabric，举个栗子（蜂蜜栗子 = = ），以NVMe over RDMA为例，type类型为0时，那么sub type必须为1，表示数据就在nvme命令的尾部，偏移为offset，简单来说，就是initiator使用RDMA发送一个send，其数据前64byte是nvme命令，64byte后面是真正的数据。type类型为4时且sub type为0时，表示数据还在远端，地址为addr、长度为len、密钥为key，需要target使用RDMA read把数据读回来。

## keyed data block sgl subtype 0xF



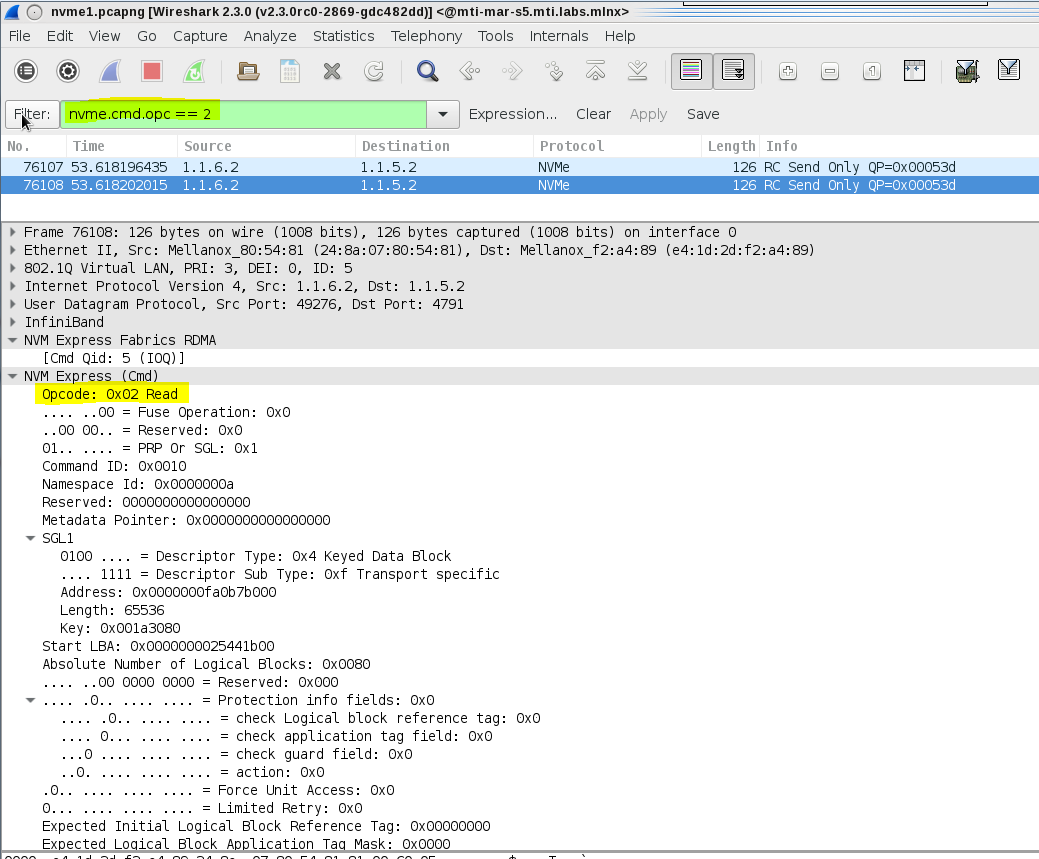


# Nvme rdma



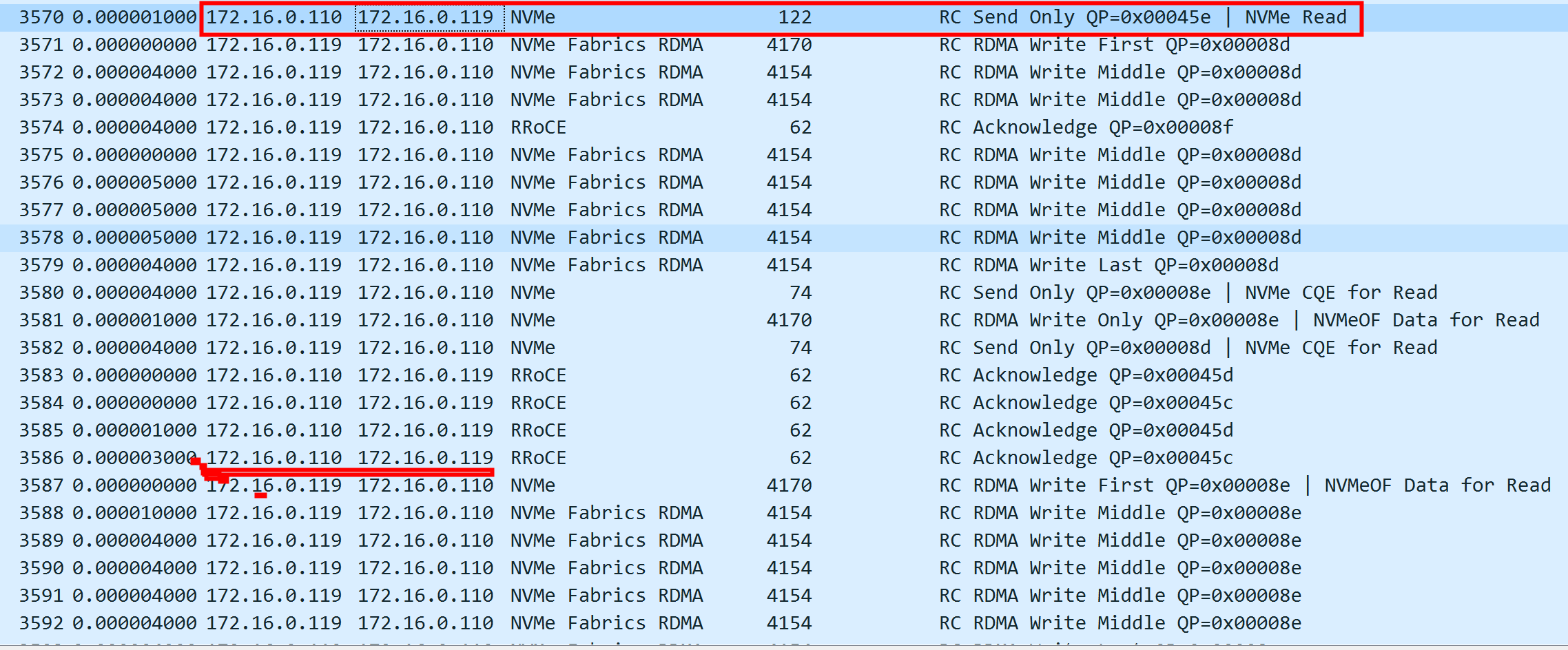
 nvme.cmd.opc ==2 for NVMe read

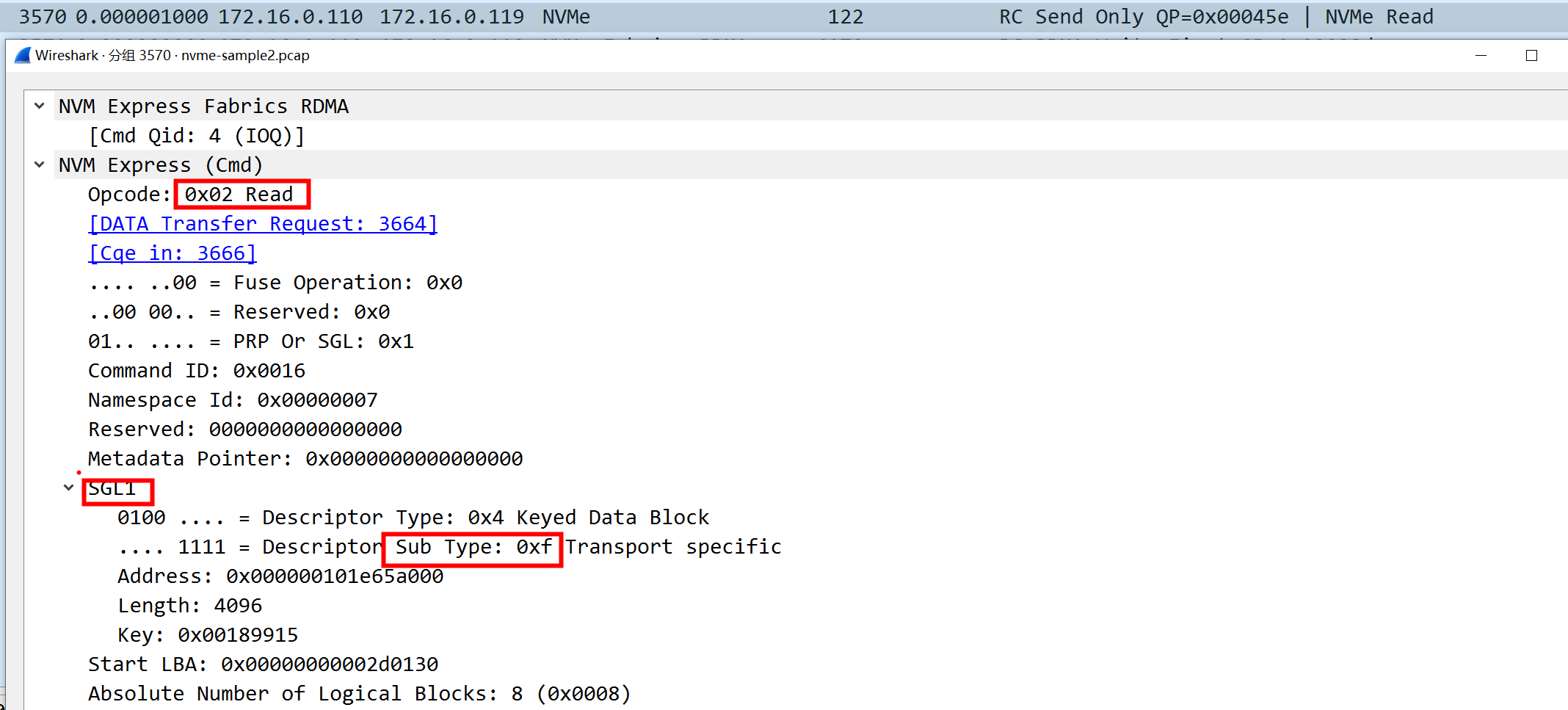
 nvme.cmd.opc ==1 for NVMe write



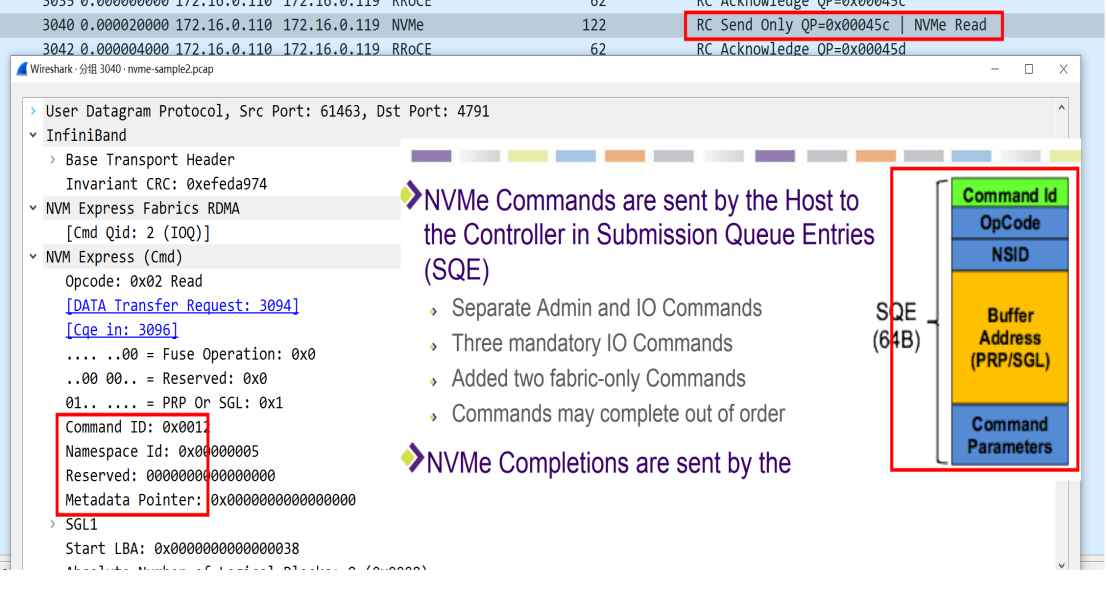
## read

### Host 发送read命令

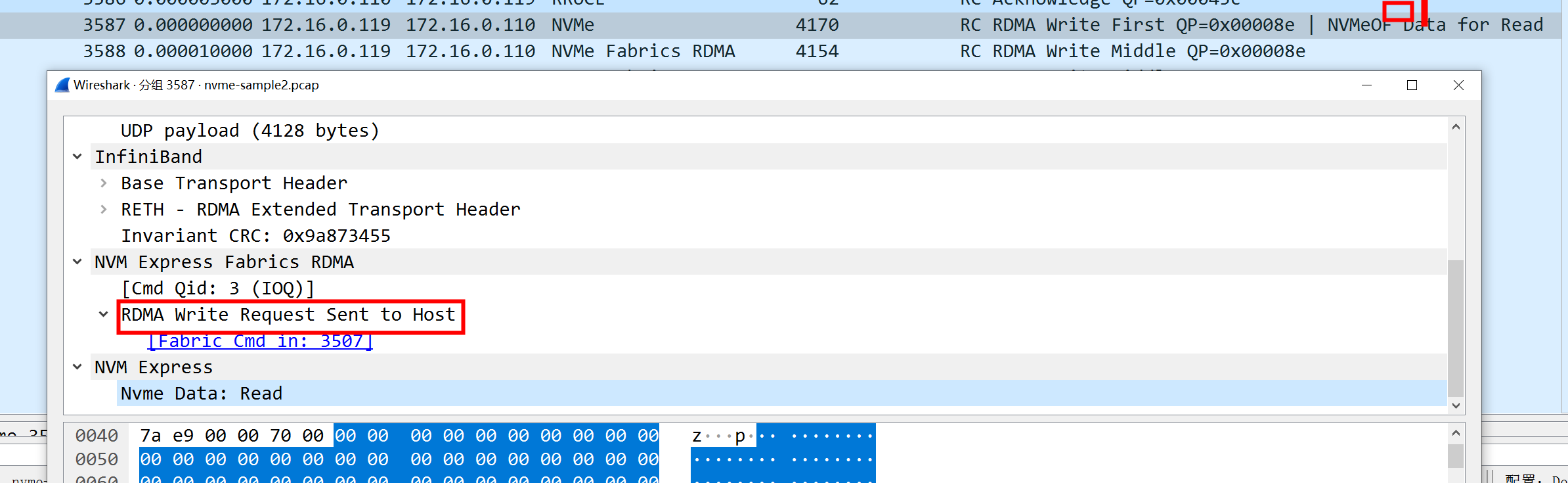




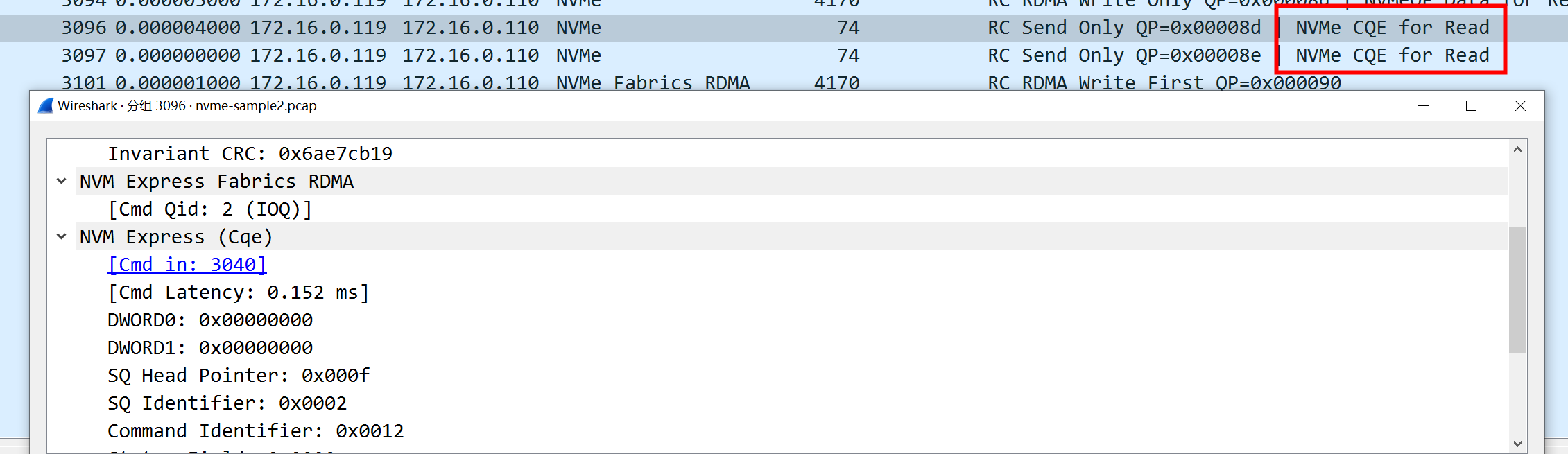
### Host发送SQE



### Target 发送data到host



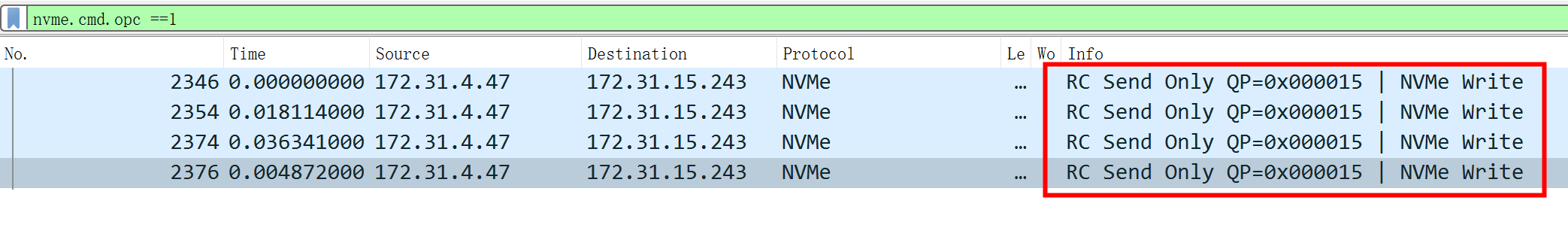
### Target 发送CQE到host



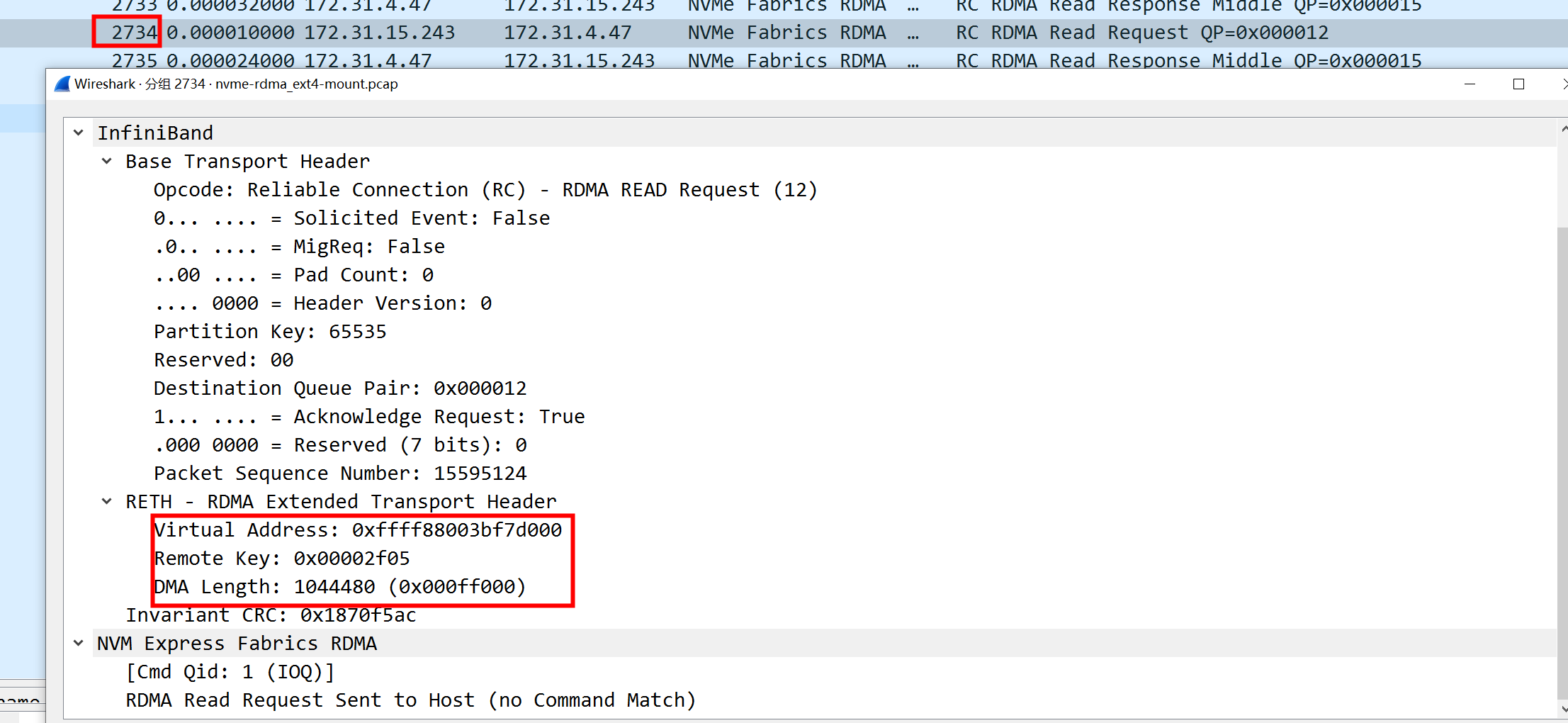


## write

nvme.cmd.opc ==1



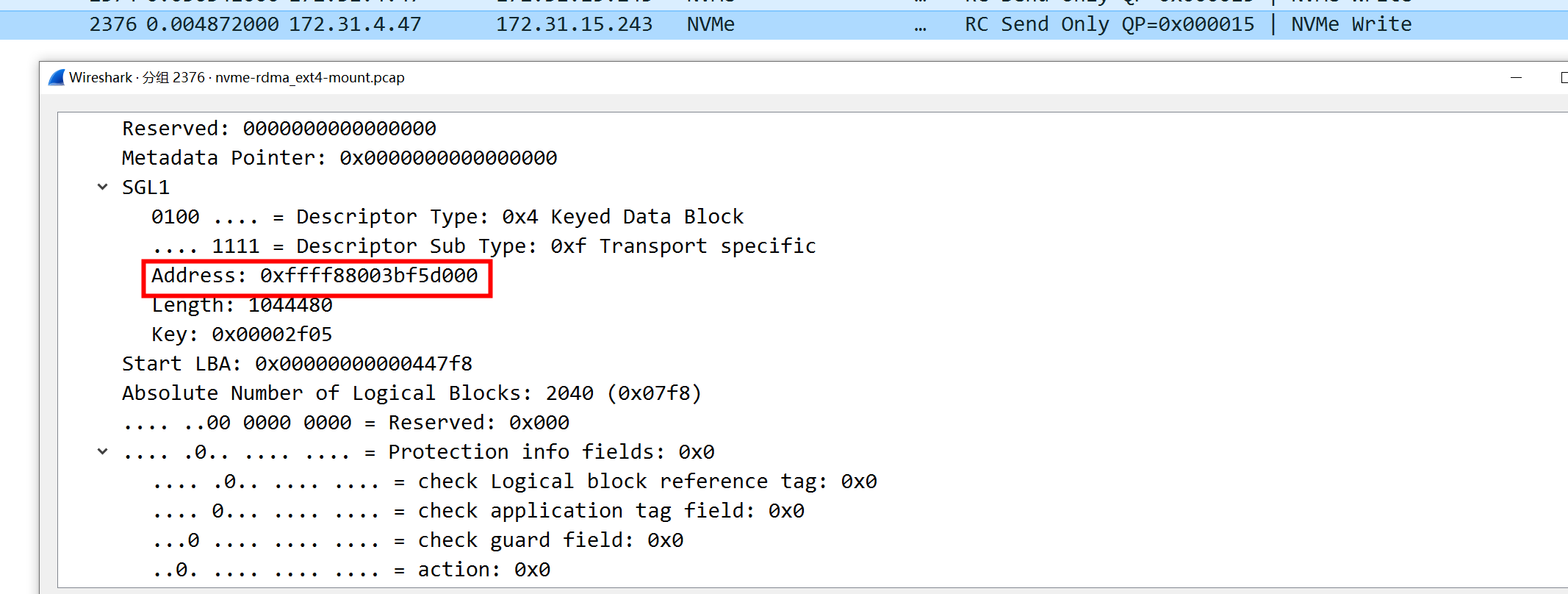
### Target 发送read请求1（非nvme write）



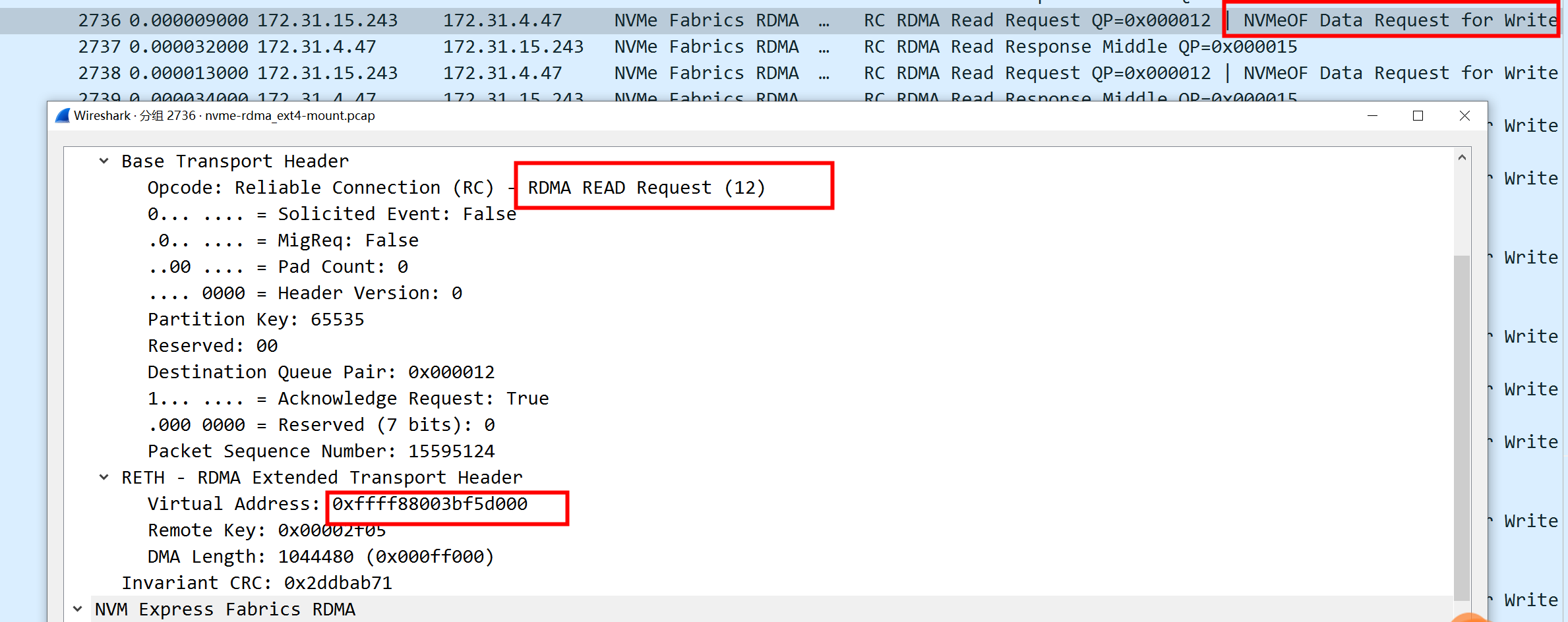
Virtual Address: 0xffff88003bf7d000

### Host 发送write请求2（nvme write）

Address: 0xffff88003bf5d000



#### Target 发送read请求2



Virtual Address: 0xffff88003bf5d000是host发送的sgl中的Address: 0xffff88003bf5d000

# References

|  |  |
| --- | --- |
|  | Efficient Crash Consistency for NVMe over PCIe and RDMA  <https://dl.acm.org/doi/fullHtml/10.1145/3568428#fig2> |
|  |  |
|  |  |