**Ceph消息通信设计原理与实现**

目录

[1. socket编程基础 3](#_Toc9694488)

[1.1. socket通信函数 3](#_Toc9694489)

[1.2. TCP通信流程 5](#_Toc9694490)

[2. RDMA编程基础 6](#_Toc9694491)

[2.1. RDMA概述 6](#_Toc9694492)

[2.2. 核心概念 7](#_Toc9694493)

[2.3. RDMA数据传输 8](#_Toc9694494)

[1. 常见操作 8](#_Toc9694495)

[2. 一个例子 8](#_Toc9694496)

[2.4. 理解SGL 10](#_Toc9694497)

[1. 数据结构 10](#_Toc9694498)

[2. 创建SGL 11](#_Toc9694499)

[3. 内存保护 12](#_Toc9694500)

[4. 发送 SGL 12](#_Toc9694501)

[3. Ceph基本通信流程 14](#_Toc9694502)

[3.1. Simple通信流程 14](#_Toc9694503)

[3.2. Async通信流程 17](#_Toc9694504)

[4. Ceph消息数据结构 20](#_Toc9694505)

[4.1. Messenger结构与接口 20](#_Toc9694506)

[4.2. Messenger类型 21](#_Toc9694507)

[4.3. AsyncMessenger 22](#_Toc9694508)

[1. AsyncMessenger结构 22](#_Toc9694509)

[2. NetworkStack 23](#_Toc9694510)

[3. Worker、EventCenter 24](#_Toc9694511)

[4. Processor、AsyncConnection 25](#_Toc9694512)

[4.4. RDMAStack、Infiniband 29](#_Toc9694513)

[1. 类结构图 29](#_Toc9694514)

[2. 创建与初始化 31](#_Toc9694515)

[3. 连接建立 32](#_Toc9694516)

[4. 消息发送 33](#_Toc9694517)

[5. 消息接收 34](#_Toc9694518)

[6. 清除连接 35](#_Toc9694519)

[4.5. 与业务模块的交互 35](#_Toc9694520)

[5. 网络异常处理（待整理、补充） 36](#_Toc9694521)

[1.1. 消息保序 36](#_Toc9694522)

[1.2. 连接竞争 37](#_Toc9694523)

[1.3. 重建连接 37](#_Toc9694524)

[1.4. 错误处理 37](#_Toc9694525)

[6. Ceph集群网络 37](#_Toc9694526)

[7. 附录 39](#_Toc9694527)

# socket编程基础

Socket（套接字）即网络进程的ID，网络通信归根到底即为进程间的通信；套接字中包含了IP地址和端口号，IP用来确定计算机，端口用来确定进程（一个端口号一次只能分配给一个进程）。这样就可以寻找到网络中的进程。

## socket通信函数

1. socket 函数

　　为了执行网络I/O，进程必须做的第一件事就是执行socket函数，指定期望的通信协议类型。套接字是通信端点的抽象，实现端对端之间的通信，访问套接字需要套接字描述符。套接字描述符通过socket 函数获得，这样才能对套接字进行操作。

/\*

\* 函数功能：创建套接字描述符；

\* 返回值：若成功则返回套接字非负描述符，若出错返回-1；

\* 函数原型：

\*/

#include <sys/socket.h>

int socket(int family, int type, int protocol);

说明：

socket类似与open对普通文件操作一样，都是返回描述符，后续的操作都是基于该描述符。family 表示套接字的通信域，不同的取值决定了socket的地址类型；type确定socket的类型；protocol指定协议。

1. bind 函数

　　bind()函数用于将一个网络地址赋予一个套接字，因为套接字在创建之初是没有地址的需要进行赋值，这里的地址一般为网络IP和端口号。

/\*

\* 函数功能：将协议地址绑定到一个套接字；其中协议地址包含IP地址和端口号；

\* 返回值：若成功则返回0，若出错则返回-1；

\* 函数原型：

\*/

#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

说明：sockfd 为套接字描述符；addr是一个指向特定协议地址结构的指针,通用套接字结构；addrlen是地址结构的长度；

1. listen 函数

　　listen函数只用于服务器端，服务器进程不知道要与谁连接，因此，它不会主动地要求与某个进程连接，只是一直监听是否有其他客户进程与之连接，然后响应该连接请求，并对它做出处理，一个服务进程可以同时处理多个客户进程的连接。主要就两个功能：将一个未连接的套接字转换为一个被动套接字（监听），规定内核为相应套接字排队的最大连接数。

/\*

\* 函数功能：接收连接请求；

\* 若成功则返回0，若出错则返回-1；

\* 函数原型：

\*/

#include <sys/socket.h>

int listen(int sockfd, int backlog);

/\*

\* sockfd是套接字描述符；

\* backlog是该进程所要入队请求的最大请求数量；

\*/

　　内核为任何一个给定监听套接字维护两个队列：未完成连接队列，每个这样的 SYN 报文段对应其中一项：已由某个客户端发出并到达服务器，而服务器正在等待完成相应的 TCP 三次握手过程。这些套接字处于 SYN\_REVD 状态;已完成连接队列，每个已完成 TCP 三次握手过程的客户端对应其中一项。这些套接字处于 ESTABLISHED 状态;

1. accept 函数

　　accept 函数由 TCP 服务器调用，用于从已完成连接队列队头返回下一个已完成连接。如果已完成连接队列为空，那么进程被投入睡眠。

/\* 函数功能：从已完成连接队列队头返回下一个已完成连接；若已完成连接队列为空，则进程进入睡眠；

\* 返回值：若成功返回套接字描述符，出错返回-1；

\* 函数原型：

\*/

#include <sys/socket.h>

int accept(int sockfd, struct sockaddr \*cliaddr, socklen\_t \*addrlen);

/\*

说明：参数 cliaddr 和 addrlen 用来返回已连接的对端（客户端）的协议地址；该函数返回套接字描述符，该描述符连接到调用connect函数的客户端；这个新的套接字描述符和原始的套接字描述符sockfd具有相同的套接字类型和地址族，而传给accept函数的套接字描述符sockfd没有关联到这个链接，而是继续保持可用状态并接受其他连接请求；

　　该函数的返回值是一个新的套接字描述符，返回值是表示已连接的套接字描述符，而第一个参数是服务器监听套接字描述符。一个服务器通常仅仅创建一个监听套接字，它在该服务器的生命周期内一直存在。内核为每个由服务器进程接受的客户连接创建一个已连接套接字(表示 TCP 三次握手已完成)，当服务器完成对某个给定客户的服务时，相应的已连接套接字就会被关闭。

1. connect 函数

　　TCP用户用connect函数与服务器建立连接。

/\*

\* 函数功能：建立连接，即客户端使用该函数来建立与服务器的连接；

\* 返回值：若成功则返回0，出错则返回-1；

\* 函数原型：

\*/

#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr \*servaddr, socklen\_t addrlen);

/\*

说明：sockfd是系统调用的套接字描述符，即由socket函数返回的套接字描述符； servaddr是目的套接字的地址，该套接字地址结构必须包含目的IP地址和目的端口号，即想与之通信的服务器地址；addrlen是目的套接字地址的大小；

1. close 函数

/\*

\* 函数功能：关闭套接字，若是在 TCP 协议中，并终止 TCP 连接；

\* 返回值：若成功则返回0，若出错则返回-1；

\* 函数原型：

\*/

#include <unistd.h>

int close(int sockfd);

说明：close之后套接字描述符不能再由调用进程使用，即不能作为read和write参数；在多进程并发服务器中，父子进程共享着套接字，套接字描述符引用计数记录着共，享着的进程个数，当父进程或某一子进程close掉套接字时，描述符引用计数会相，应的减一，当引用计数仍大于零时，这个close调用就不会引发TCP的四路握手断连过程。shutdown函数也有类似的功能；

## TCP通信流程

　　TCP 是一种面向连接的字节流套接字，服务端先创建socket()，然后绑定套接字bind()，然后通过 listen() 转变为被动 socket，通过 accept() 等待连接。客户端创建socket()，然后通过connect()去连接服务器。当连接成功后，二者便可以互相发送消息，直到关闭close()连接，流程如下图所示：

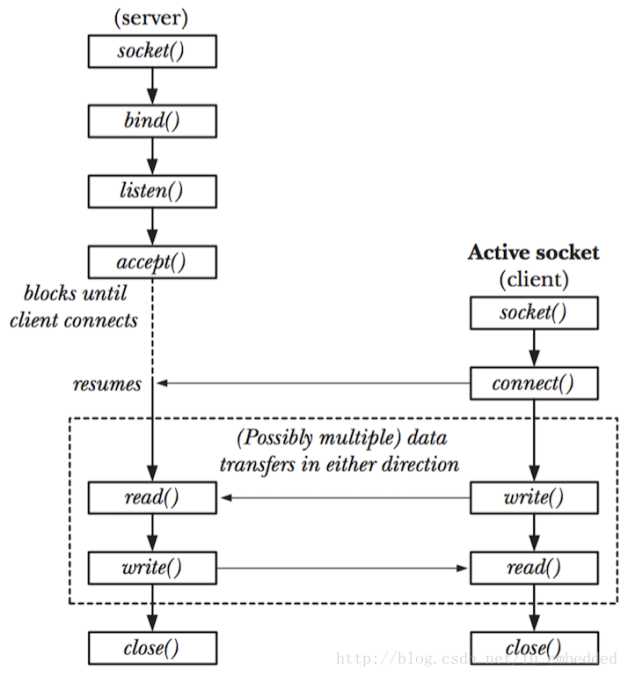


图 1 TCP客户/服务器通信流程图

# RDMA编程基础

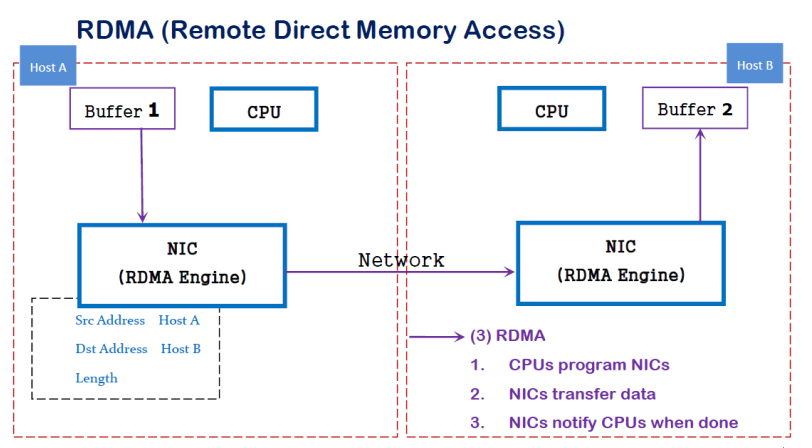
## RDMA概述

RDMA是一种host-offload, host-bypass技术，允许应用程序(包括存储)在它们的内存空间之间直接做数据传输。具有RDMA引擎的以太网卡(RNIC)--而不是host--负责管理源和目标之间的可靠连接。使用RNIC的应用程序之间使用专注的QP和CQ进行通讯：

每一个应用程序可以有很多QP和CQ

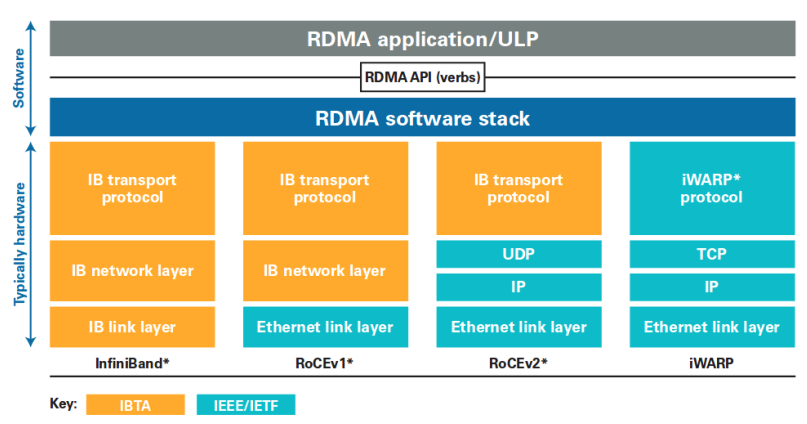
每一个QP包括一个SQ和RQ

每一个CQ可以跟多个SQ或者RQ相关联



RDMA作为一种host-offload, host-bypass技术，使低延迟、高带宽的直接的内存到内存的数据通信成为了可能。目前支持RDMA的网络协议有：

1. InfiniBand(IB): 从一开始就支持RDMA的新一代网络协议。由于这是一种新的网络技术，因此需要支持该技术的网卡和交换机。
2. RDMA过融合以太网(RoCE): 即RDMA over Ethernet, 允许通过以太网执行RDMA的网络协议。这允许在标准以太网基础架构(交换机)上使用RDMA，只不过网卡必须是支持RoCE的特殊的NIC。
3. 互联网广域RDMA协议(iWARP): 即RDMA over TCP, 允许通过TCP执行RDMA的网络协议。这允许在标准以太网基础架构(交换机)上使用RDMA，只不过网卡要求是支持iWARP(如果使用CPU offload的话)的NIC。否则，所有iWARP栈都可以在软件中实现，但是失去了大部分的RDMA性能优势。



## 核心概念

1. Memory Registration(MR) | 内存注册

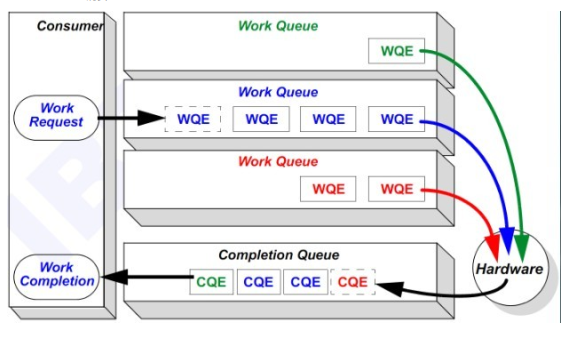
用RDMA的前提就是"搞内存"。 怎么“搞”？ 很简单，注册。 因为RDMA硬件对用来做数据传输的内存是有特殊要求的。在数据传输过程中，应用程序不能修改数据所在的内存。  
操作系统不能对数据所在的内存进行page out操作 -- 物理地址和虚拟地址的映射必须是固定不变的。注意无论是DMA或者RDMA都要求物理地址连续，这是由DMA引擎所决定的。 那么怎么进行内存注册呢？创建两个key (local和remote)指向需要操作的内存区域注册的keys是数据传输请求的一部分

1. Queues | 队列

RDMA一共支持三种队列，发送队列(SQ)和接收队列(RQ)，完成队列(CQ)。其中，SQ和RQ通常成对创建，被称为Queue Pairs(QP)。RDMA是基于消息的传输协议，数据传输都是异步操作。 RDMA操作其实很简单，可以理解为：

1. Host提交工作请求(WR)到工作队列(WQ): 工作队列包括发送队列(SQ)和接收队列(RQ)。工作队列的每一个元素叫做WQE, 也就是WR。
2. Host从完成队列(CQ）中获取工作完成(WC): 完成队列里的每一个叫做CQE, 也就是WC。
3. 具有RDMA引擎的硬件(hardware)就是一个队列元素处理器。 RDMA硬件不断地从工作队列(WQ)中去取工作请求(WR)来执行，执行完了就给完成队列(CQ)中放置工作完成(WC)。从生产者-消费者的角度理解就是：

* Host生产WR, 把WR放到WQ中去
* RDMA硬件消费WR
* RDMA硬件生产WC, 把WC放到CQ中去
* Host消费WC



## RDMA数据传输

### 常见操作

1. RDMA Send | RDMA发送(/接收)操作 （Send/Recv）

跟TCP/IP的send/recv是类似的，不同的是RDMA是基于消息的数据传输协议（而不是基于字节流的传输协议），所有数据包的组装都在RDMA硬件  
上完成的，也就是说OSI模型中的下面4层(传输层，网络层，数据链路层，物理层)都在RDMA硬件上完成。

1. RDMA Read | RDMA读操作 (Pull)

RDMA读操作本质上就是Pull操作, 把远程系统内存里的数据拉回到本地系统的内存里。

1. RDMA Write | RDMA写操作 (Push)

RDMA写操作本质上就是Push操作，把本地系统内存里的数据推送到远程系统的内存里。

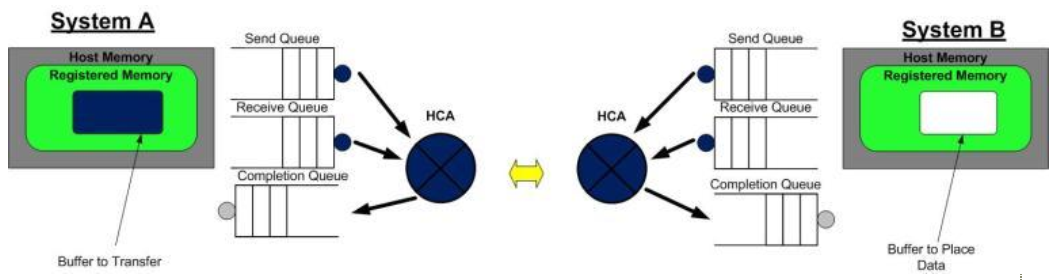
1. RDMA Write with Immediate Data | 支持立即数的RDMA写操作

支持立即数的RDMA写操作本质上就是给远程系统Push(推送)带外(OOB)数据, 这跟TCP里的带外数据是类似的。

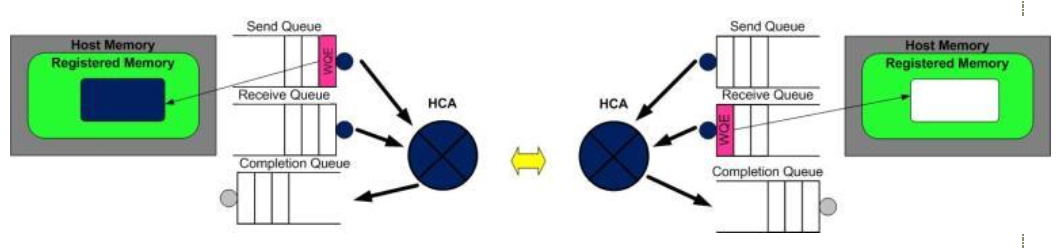
### 一个例子

让我们看个简单的例子。在这个例子中，我们将把一个缓冲区里的数据从系统A的内存中搬到系统B的内存中去。这就是我们所说的消息传递语义学。接下来我们要讲的一种操作为SEND，是RDMA中最基础的操作类型。

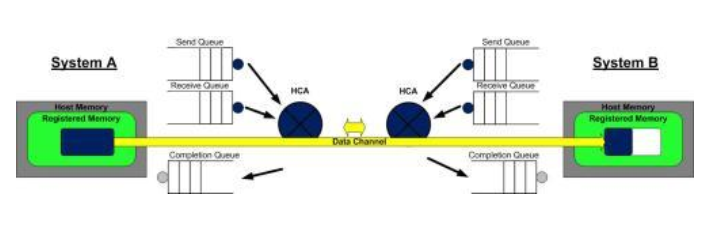
第1步：系统A和B都创建了他们各自的QP的完成队列(CQ), 并为即将进行的RDMA传输注册了相应的内存区域(MR)。 系统A识别了一段缓冲区，该缓冲区的数据将被搬运到系统B上。系统B分配了一段空的缓冲区，用来存放来自系统A发送的数据。



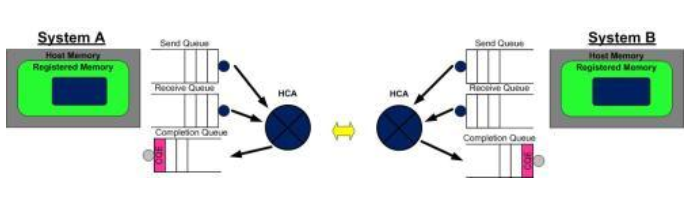
第2步：系统B创建一个WQE并放置到它的接收队列(RQ)中。这个WQE包含了一个指针，该指针指向的内存缓冲区用来存放接收到的数据。系统A也创建一个WQE并放置到它的发送队列(SQ)中去，该WQE中的指针执行一段内存缓冲区，该缓冲区的数据将要被传送。



第3步：系统A上的HCA总是在硬件上干活，看看发送队列里有没有WQE。HCA将消费掉来自系统A的WQE, 然后将内存区域里的数据变成数据流发送给系统B。当数据流开始到达系统B的时候，系统B上的HCA就消费来自系统B的WQE，然后将数据放到该放的缓冲区上去。在高速通道上传输的数据流完全绕过了操作系统内核。



第4步：当数据搬运完成的时候，HCA会创建一个CQE。 这个CQE被放置到完成队列(CQ)中，表明数据传输已经完成。HCA每消费掉一个WQE, 都会生成一个CQE。因此，在系统A的完成队列中放置一个CQE,意味着对应的WQE的发送操作已经完成。同理，在系统B的完成队列中也会放置一个CQE，表明对应的WQE的接收操作已经完成。如果发生错误，HCA依然会创建一个CQE。在CQE中，包含了一个用来记录传输状态的字段。



## 理解SGL

### 数据结构

在RDMA编程中，SGL(Scatter/Gather List)是最基本的数据组织形式。 SGL是一个数组，该数组中的元素被称之为SGE(Scatter/GatherElement)，每一个SGE就是一个Data Segment(数据段)。其中，SGE的定义如下(参见verbs.h)：

struct ibv\_sge {

uint64\_t addr;

uint32\_t length;

uint32\_t lkey;

};

* addr: 数据段所在的虚拟内存的起始地址 (Virtual Address of the Data Segment (i.e. Buffer))
* length: 数据段长度(Length of the Data Segment)
* lkey: 该数据段对应的L\_Key (Key of the local Memory Region)

而在数据传输中，发送/接收使用的Verbs API为：

* ibv\_post\_send() - post a list of work requests (WRs) to a send queue 将一个WR列表放置到发送队列中
* ibv\_post\_recv() - post a list of work requests (WRs) to a receive queue 将一个WR列表放置到接收队列中

下面以ibv\_post\_send()为例，说明SGL是如何被放置到RDMA硬件的线缆(Wire)上的。ibv\_post\_send()的函数原型如下：

#include <infiniband/verbs.h>

int ibv\_post\_send(struct ibv\_qp \*qp,struct ibv\_send\_wr \*wr,struct ibv\_send\_wr \*\*bad\_wr);

struct ibv\_send\_wr {

uint64\_t wr\_id; /\* User defined WR ID \*/

struct ibv\_send\_wr \*next; /\* Pointer to next WR in list, NULL if last WR \*/

struct ibv\_sge \*sg\_list; /\* Pointer to the s/g array \*/

int num\_sge; /\* Size of the s/g array \*/

enum ibv\_wr\_opcode opcode; /\* Operation type \*/

int send\_flags; /\* Flags of the WR properties \*/

uint32\_t imm\_data; /\* Immediate data (in network byte order) \*/

union {

struct {

uint64\_t remote\_addr; /\* Start address of remote memory buffer \*/

uint32\_t rkey; /\* Key of the remote Memory Region \*/

} rdma;

struct {

uint64\_t remote\_addr; /\* Start address of remote memory buffer \*/

uint64\_t compare\_add; /\* Compare operand \*/

uint64\_t swap; /\* Swap operand \*/

uint32\_t rkey; /\* Key of the remote Memory Region \*/

} atomic;

struct {

struct ibv\_ah \*ah; /\* Address handle (AH) for the remote node address \*/

uint32\_t remote\_qpn; /\* QP number of the destination QP \*/

uint32\_t remote\_qkey; /\* Q\_Key number of the destination QP \*/

} ud;

} wr;

};

struct ibv\_sge {

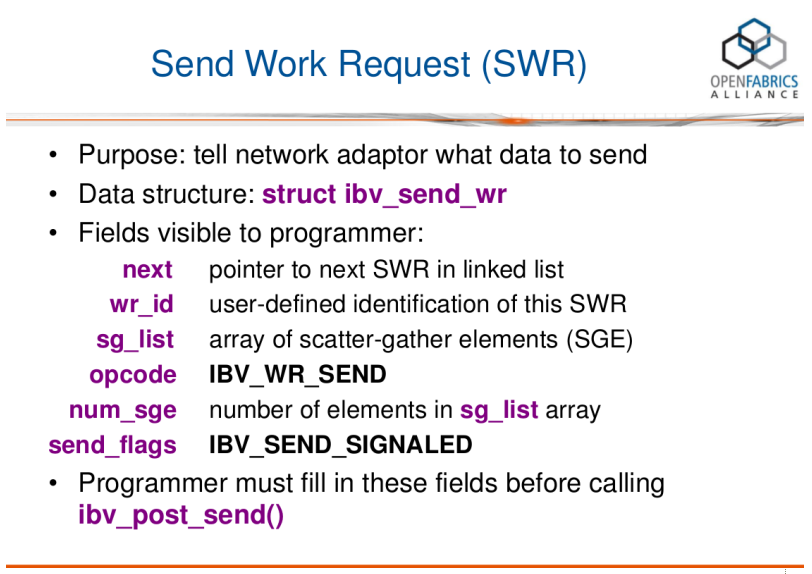
uint64\_t addr; /\* Start address of the local memory buffer \*/

uint32\_t length; /\* Length of the buffer \*/

uint32\_t lkey; /\* Key of the local Memory Region \*/

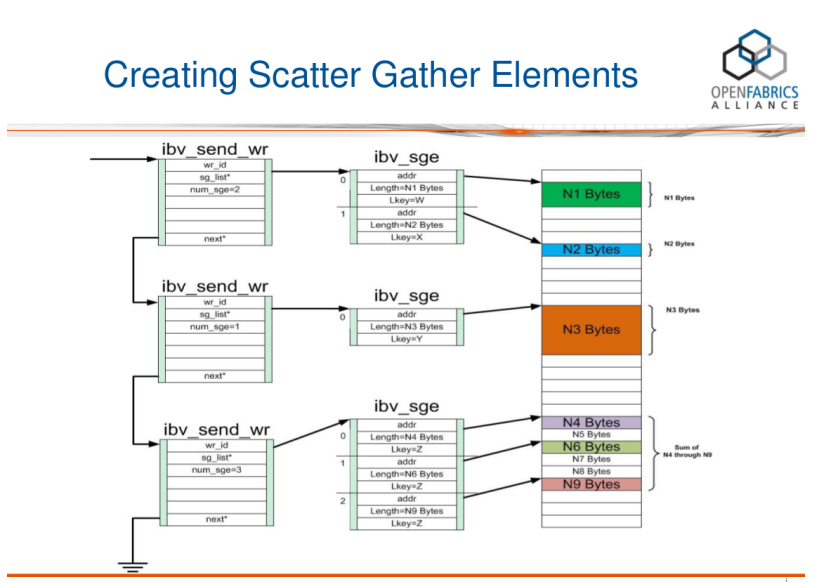
};

在调用ibv\_post\_send()之前，必须填充好数据结构wr。 wr是一个链表，每一个结点包含了一个sg\_list(i.e. SGL: 由一个或多个SGE构成的数组), sg\_list的长度为num\_sge。



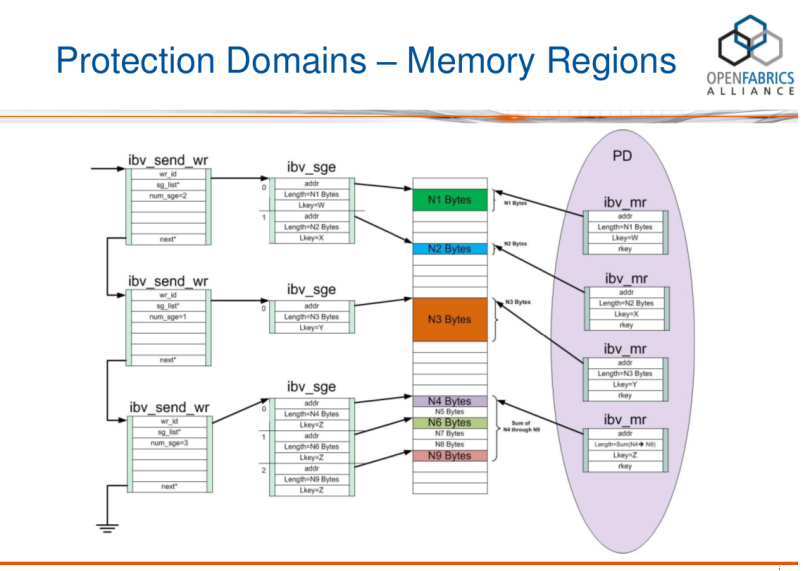
下面图解一下SGL和WR链表的对应关系，并说明一个SGL (struct ibv\_sge \*sg\_list)里包含的多个数据段是如何被RDMA硬件聚合成一个连续的数据段的。

### 创建SGL



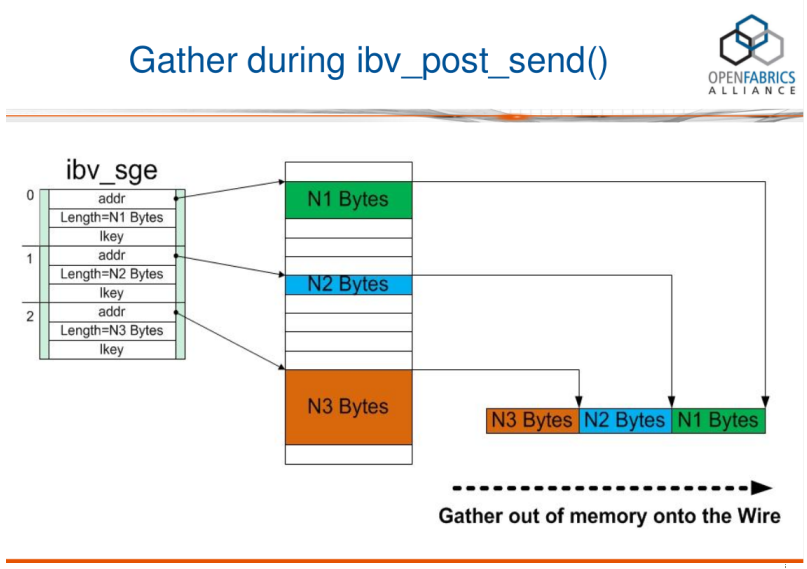
从上图中，我们可以看到wr链表中的每一个结点都包含了一个SGL，SGL是一个数组，包含一个或多个SGE。

### 内存保护



一个SGL至少被一个MR保护, 多个MR存在同一个PD中。

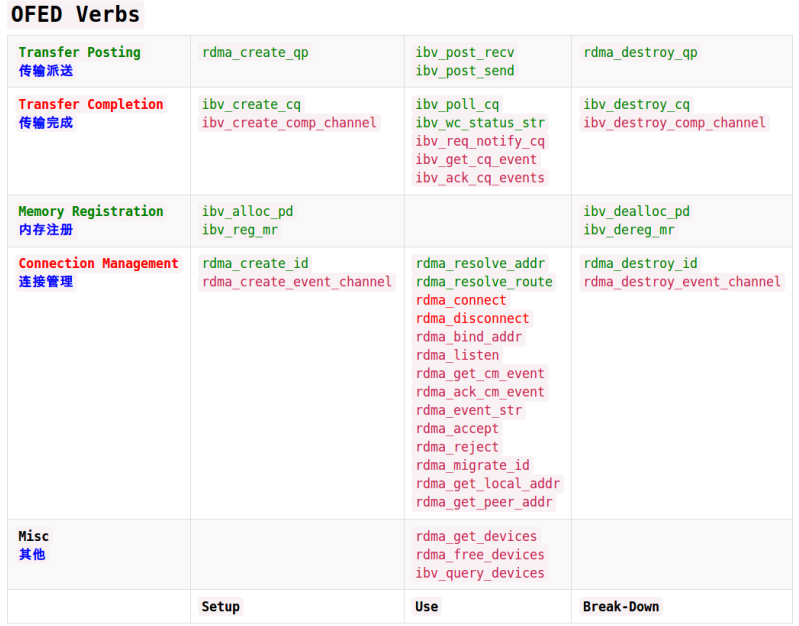
### 发送 SGL



在上图中，一个SGL数组包含了3个SGE, 长度分别为N1, N2, N3字节。我们可以看到，这3个buffer并不连续，它们Scatter(分散)在内存中的各个地方。RDMA硬件读取到SGL后，进行Gather(聚合)操作，于是在RDMA硬件的Wire上看到的就是N3+N2+N1个连续的字节。换句话说，通过使用SGL, 我们可以把分散(Scatter)在内存中的多个数据段(不连续)交给RDMA硬件去聚合(Gather)成连续的数据段。最后，作为一个代码控（不喜欢纸上谈兵），贴一小段代码展示一下如何为调用ibv\_post\_send()准备SGL和WR以加深理解。

1 #define BUFFER\_SIZE 1024  
2  
3 struct connection {  
4 struct rdma\_cm\_id \*id;  
5 struct ibv\_qp \*qp;  
6

7 struct ibv\_mr \*recv\_mr;  
8 struct ibv\_mr \*send\_mr;  
9  
10 char recv\_region[BUFFER\_SIZE];  
11 char send\_region[BUFFER\_SIZE];  
12  
13 int num\_completions;  
14 };  
15  
16 void foo\_send(void \*context)  
17 {  
18 struct connection \*conn = (struct connection \*)context;  
19  
20 /\* 1. Fill the array SGL having only one element \*/  
21 struct ibv\_sge sge;  
22  
23 memset(&sge, 0, sizeof(sge));  
24 sge.addr = (uintptr\_t)conn->send\_region;  
25 sge.length = BUFFER\_SIZE;  
26 sge.lkey = conn->send\_mr->lkey;  
27  
28 /\* 2. Fill the singly-linked list WR having only one node \*/  
29 struct ibv\_send\_wr wr;  
30 struct ibv\_send\_wr \*bad\_wr = NULL;  
31  
32 memset(&wr, 0, sizeof(wr));  
33 wr.wr\_id = (uintptr\_t)conn;  
34 wr.opcode = IBV\_WR\_SEND;  
35 wr.sg\_list = &sge;  
36 wr.num\_sge = 1;  
37 wr.send\_flags = IBV\_SEND\_SIGNALED;  
38  
39 /\* 3. Now send ... \*/  
40 ibv\_post\_send(conn->qp, &wr, &bad\_wr);  
41  
42 ...<snip>...  
43 }



# Ceph基本通信流程

网络通信模块实现在src/msg目录下，定义了三个通信框架，分别为：

Smiple：比较简单，目前比较稳定的实现

Async：使用了基于时间的I/O多路复用模式（使用DPDK+RDMA）

XIO：使用开源的网络通信库accelio来实现

在这里先主要介绍一下simple和async的基本通信流程。让大家有一个直观的认识。

## Simple通信流程

1、服务端的启动流程如下图所示：



图 2 Simple服务端流程图

说明：OSD1表示服务端、OSD2表示客户端

从上图可以看出：

1. 服务端会创建一个单独的线程（accepter线程）专门用来接收客户端的connect；
2. 当有客户端来connect时，accepter线程又会创建一个读线程（reader线程）
3. 该读线程会与客户端进行信息认证，如果认证成功，该读线程又会创建一个写线程（writer线程）。
4. 后续与该客户端的通信就由该读线程和写线程来进行交互。

从上述流程可知：每一条新的连接都会创建2个线程，当集群中连接很多时，线程数也就会成倍的增长。

2、客户端建立连接的流程如下图所示：



图 3 Simple客户端流程图

说明：OSD1表示客户端、OSD2表示服务端

从上图可以看出：

1. 当客户端需要向服务端发送消息时，直接调用接口函数send\_message\_osd\_cluster，他根据OSD编号在OSDMAP中查找该OSD的IP地址和端口。
2. 然后再根据IP地址和端口查找是否已经建立连接，如果不存在建立新的连接。
3. 在建立新的连接时首先创建一个写线程（writer线程），该线程向服务器发起connect。
4. 服务器accept后，写线程与服务器开始交互信息，进行认证。
5. 认证成功后，写线程会创建一个读线程（reader线程），后续与服务端的通信就由该读线程和写线程来进行交互。

从上述流程也可以看出，客户端每建立一条新的连接都会创建2个线程，当集群中连接很多时，线程数也就会成倍的增长

Simple通信流程的致命缺陷就是线程数太多。比如一个集群中有36个OSD，那么一个OSD就有可能与其他35个OSD建立连接，一个连接2个线程。那么就意味着单个osd进程中就会存在35 \* 2 = 70个读写线程。线程数过多会导致系统开销过大，线程切换时延增大，影响系统性能，于是就有了下面的async通信方式。

## Async通信流程

1、服务端的启动流程如下图所示：



图 4 Async服务端流程图

说明：OSD1表示服务端、OSD2表示客户端

从上图可以看出：

1. OSD在初始化的时候就会创建N（默认3）worker线程，专门用于消息的发送
2. 服务端选取一个负载最小的worker线程进行accept客户端的连接。
3. 当客户端向其connect时，该worker线程再找一个负载最小的worker线程（也可能是自己），与客户端进行数据交互。

从上述流程可知：一个进程只有N（默认3）个线程用于消息的发送。

2、客户端建立连接的流程如下图所示：



图 5 Async客户端流程图

说明：OSD1表示客户端、OSD2表示服务端

从上图可以看出：

1. OSD在初始化的时候就会创建N（默认3）worker线程，专门用于消息的发送
2. 客户端选取一个负载最小的worker线程connect服务端。
3. 服务端accept后，客户端继续用该worker客户服务相互数据

通过async与simple对比后，可以发现：在大规模集群下，async可以有效的降低线程数。

# Ceph消息数据结构

## Messenger结构与接口

Messenger是个抽象层，定义了提供应用模块使用的消息处理接口。

Messenger主要有4个类：Messegner、Connection、Message、Dispatcher。



图 6 Messenger类图

1. Messenger类

Mesenger实例对应一个通信实体,往往是一类消息的集合，例如业务网络、集群网络、心跳网络。在每个实例中，可以有很多同类型的连接(Connection).

Messenger类定义了Messenger创建，连接管理、消息发送、消息分发接口。

add\_dispacher\_xxx接口是供应用注册消息接口处理函数。

Messenger接收消息进行分发时，可根据应用是否对该消息支持快速处理，而调用fast dispatch或diliver dispatch进行分发。ms\_deliver\_handle\_xxx类接口是将connection的状态变化通知应用。

1. Connection类

Connection实例对应本messenger与另一个messenger的连接（可理解为socket连接），消息收发都是基于Connection的. Connection是属于Messenger的，与其的对应关系为N:1的关系。

1. Message类

Message实例对应一个消息。

Message类定义了消息的结构，header中记录了消息的类型、序号、数据长度等信息，footer中记录消息的校验信息，payload中保存操作请求的元数据信息，data中保存读写的数据，middle很少使用。

Message类还定义了消息的编解码接口，实现讲Message数据编码为字节流，或反之。应用程序的每一个消息类型都要定义基于Message的子类，并提供payload的编解码接口。

1. Dispatcher类

Dispatcher类定义了对接收消息分发处理相关的接口，也包括处理connection事件通知接口。

应用程序类都继承Dispacher类，实现消息的分发和处理。

## Messenger类型

1. 三种Messenger类型

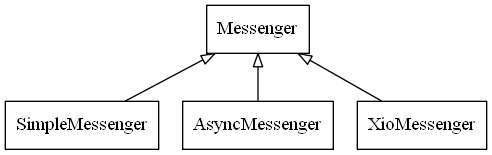


图 7 Messenger类型图

|  |  |  |  |
| --- | --- | --- | --- |
|  | SimpleMessenger | AsyncMessenger | XioMessenger |
| 发布状态 | 生产  V11版本之前，是缺省ms type。 | 生产  V11（2016-12-13）成为缺省 ms type。 | 试验 |
| 线程设计 | 每连接两线程 | 线程池 | 线程池 |
| 网络协议 | TCP | TCP、DPDK、RDMA | RDMA |

1. 线程设计对比

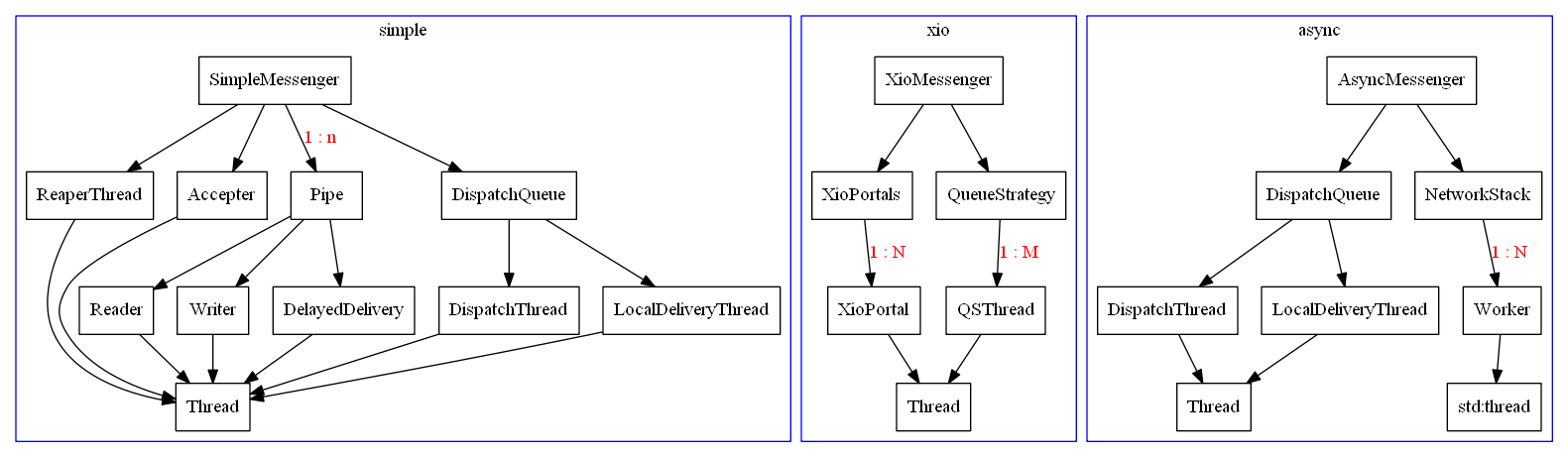


图 8 线程数对比图

AsyncMessenger采用worker线程池模式处理所有连接的Message收发，避免SimpleMessenger每连接线程模式导致线程过多引起的线程切换开销问题。

## AsyncMessenger

### AsyncMessenger结构



图 9 AsyncMessenger类图

1. AsyncMessenger类

Messenger的子类，具体实现Messenger定义的功能。

控制着NetworkStack（包括Worker及EventCenter）、Processer、DispatchQueue的创建、初始化和启动停止。

1. NetworkStack类

代表网络功能，采用单例模式，全局只实例化一个对象。

是Worker线程的容器，负责Worker的创建、启动。负责Worker的分配管理。

1. Worker类

抽象类，定义Worker控制接口。每个Worker拥有一个EventCenter。

1. EventCenter类

网络事件处理中心，监视文件事件、定时事件、外部事件，并调用事件注册的处理函数。EventCenter是AsyncMessenger的支持异步处理的核心。

1. Processer类

创建ServerSocket，绑定网络地址和端口并监听、接受连接请求。

1. ServerSocket类、ServerSocketImpl类

接受连接请求，创建ConnectSocket，创建AsyncConnection。

1. AsyncConnection类

Connection子类，实现连接具体的功能：建立连接、状态管理，消息收发。

1. ConnectedSocket类、ConnectedSocketImpl类

Connection Socket的数据收发。

1. DispachQueue类

AsyncMessenger对象包含一个DispachQueue对象。DispachQueue有两个线程，分别用于处理分发本地连接和外部连接接收的消息。

AsyncConnection接收消息后，对于应用能快速处理的消息，调用DispachQueue的快速分发接口直接处理消息；对于不能快速处理的消息，加入DispachQueue的消息队列，有DispachQueue线程进行分发。

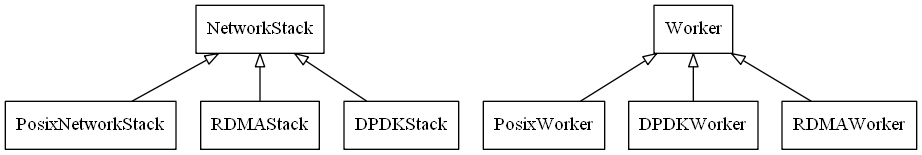
### NetworkStack

NetworkStack基本抽象类，定义了网络控制接口，网络功能由具体网络类型的子类实现。NetworkStack本身只是实现Worker线程池的创建和管理。

三种网络类型

AsyscMessenger支持插件式网络后端，目前支持三种类型：Posix、DPDK和RDMA。

每种类型都实现自己的NetworkStack、Worker、ServerSocketImpl、ConnectedSocketImpl子类。



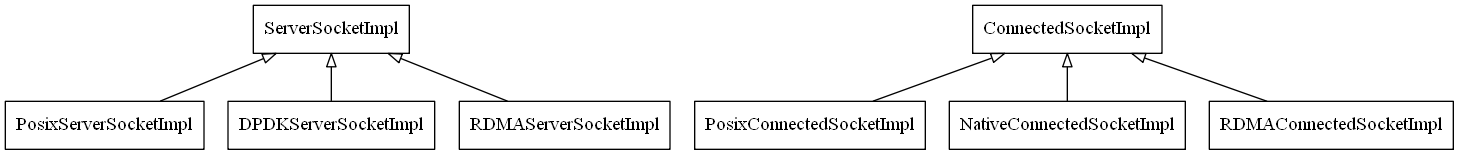


图 10 NetworkStack类图

### Worker、EventCenter

1. Worker进程创建、启动
2. 异步事件处理机制

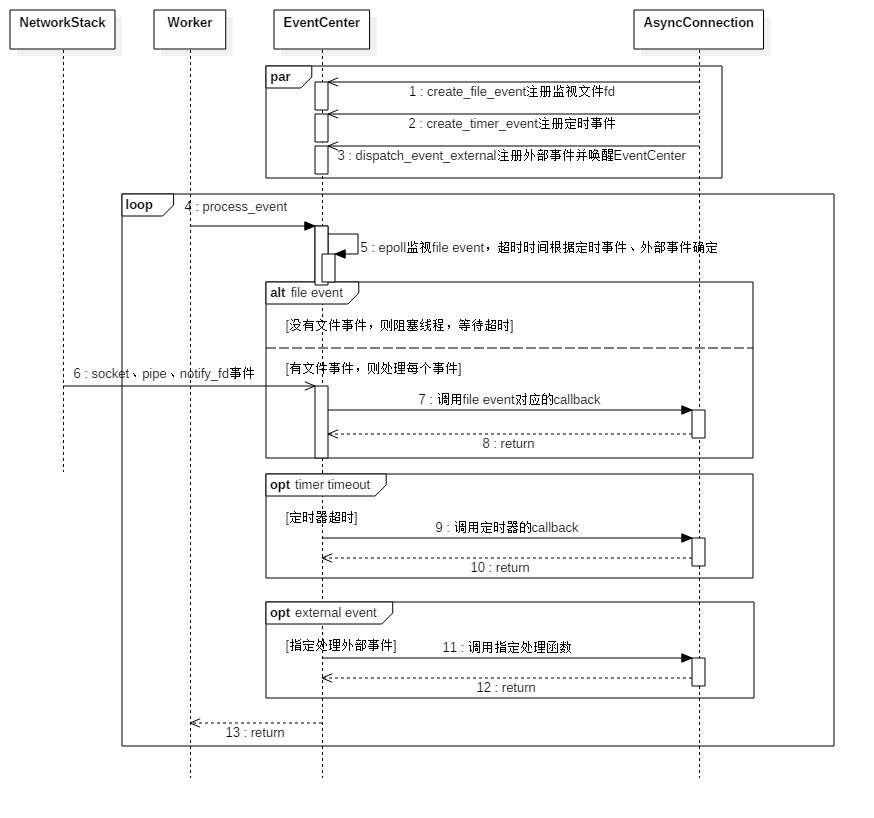


图 11 异步事件处理流程图

每个Worker包含一个EventCenter，Worker线程主函数中不停地循环调用EventCenter的事件处理函数（process\_event），直到Worker线程退出。EventCenter的事件处理机制，是AsyscMessenger得以async运行的基础。

EventCenter处理三类事件：文件事件（文件可读可写事件），定时器事件，外部事件（立即调用指定的处理函数）。

AsyncConnection、AsyncMessenger等对象可向指定的Worker的EventCenter添加指定事件的回调函数或指示执行指定的函数，由EventCenter异步执行相关事件的处理。当没有事件发生时，EventCenter会阻塞在等待文件事件。等待时间根据是否有外部事件、定时事件确定：有外部事件时不等待，有定时事件时等待时间不超过定时器剩余时间，没有注册事件时等待30秒。

EventCenter本身会创建一个pipe，并监视pipe输入事件。当需要时，可通过写pipe输出文件唤醒EventCenter。如指示执行外部事件时，即通过该方式唤醒EventCenter。

当向EventCenter添加了文件事件（如socket文件），当相关事件发生时（socket接收数据）也会唤醒EventCenter。

### Processor、AsyncConnection

1. Connection建立流程

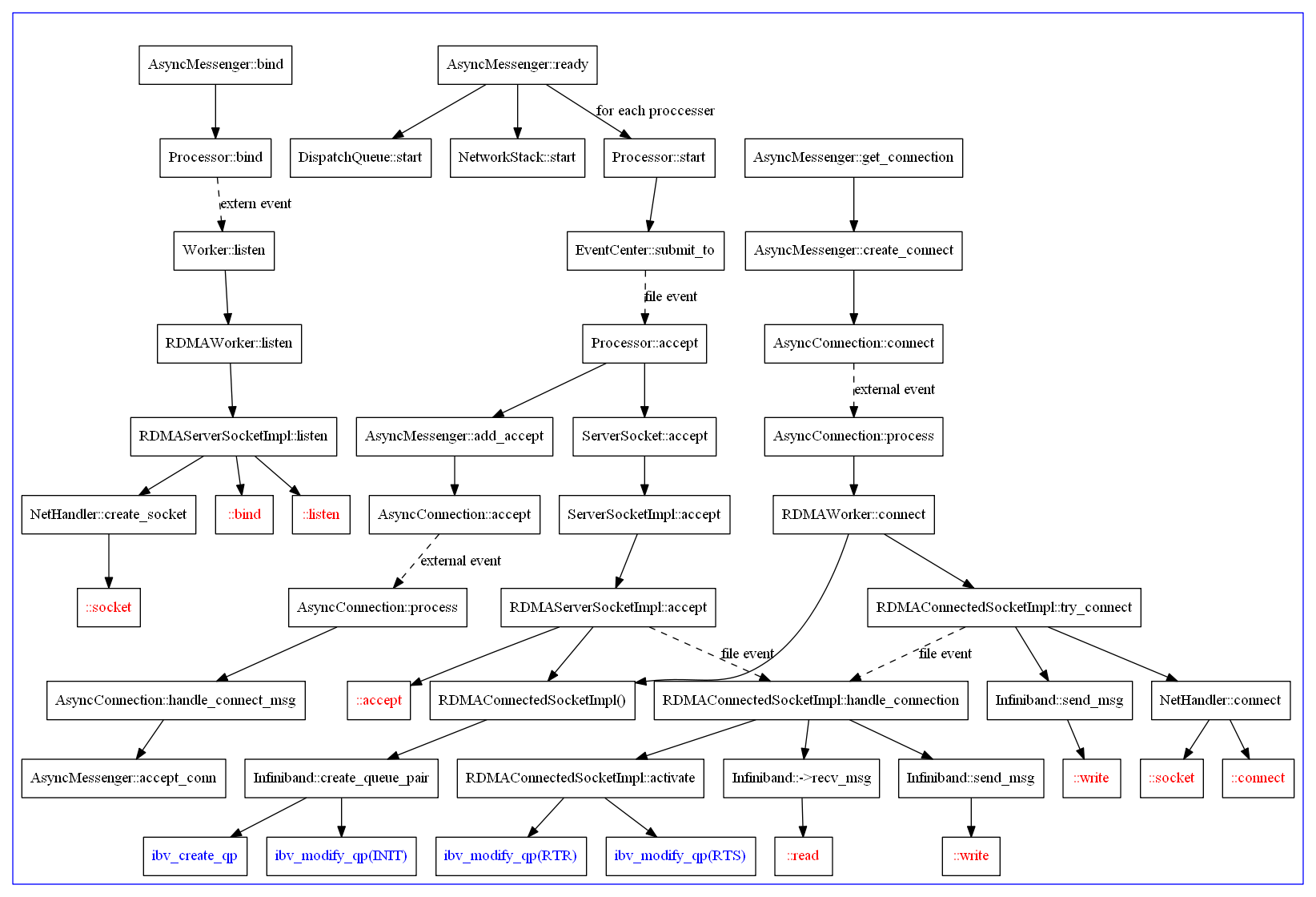


图 12 connect函数调用关系图

Server Messenger调用bind接口，Processor worker创建ServerSocket，bind，listen。

Server Messenger调用ready接口，Processor worker调用accept接受Client连接请求，创建connect socket和AsyncConnection。

Client Messenger调用create\_connect接口，创建AsyncConnection，调用connect请求连接。

1. Ceph Network Protocol协商过程

Socket连接建立后，两端开始Ceph Network协议协商。



图 13 协议协商流程图

1. Server accept过程状态迁移：

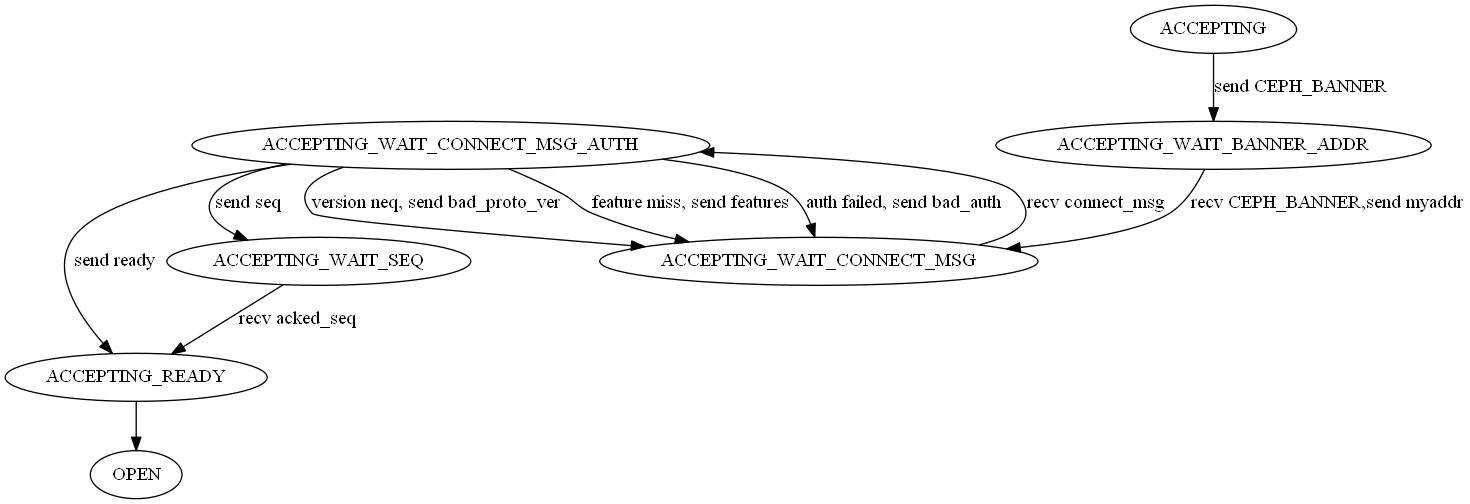


图 14 accept 状态迁移图

1. Client connect过程状态迁移：

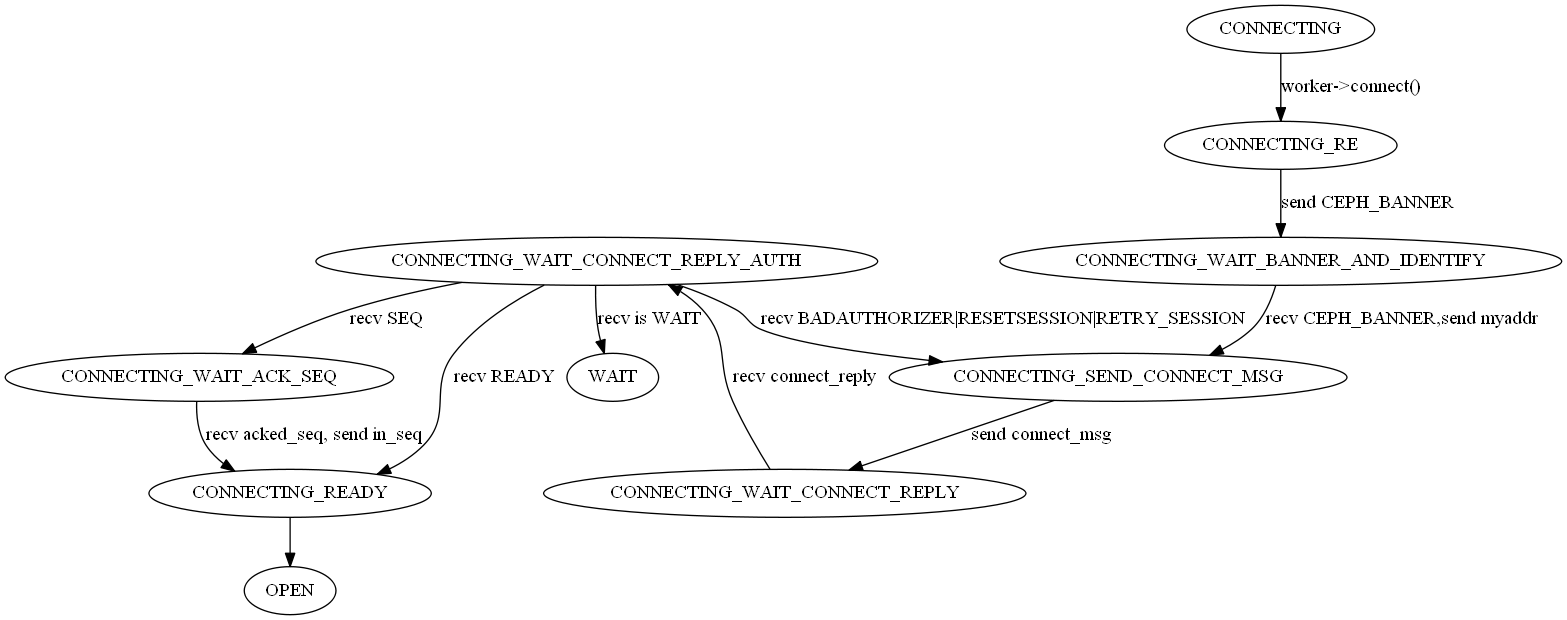
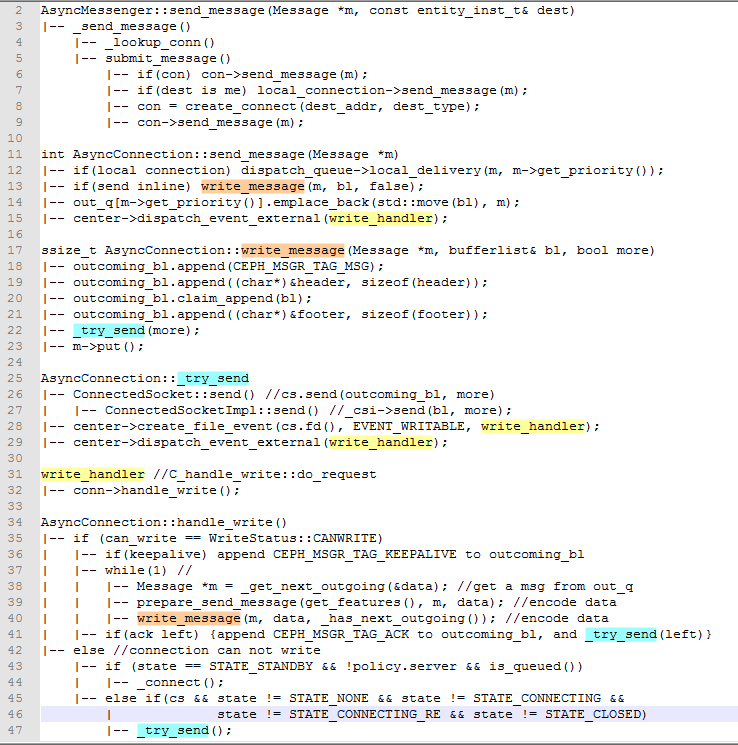
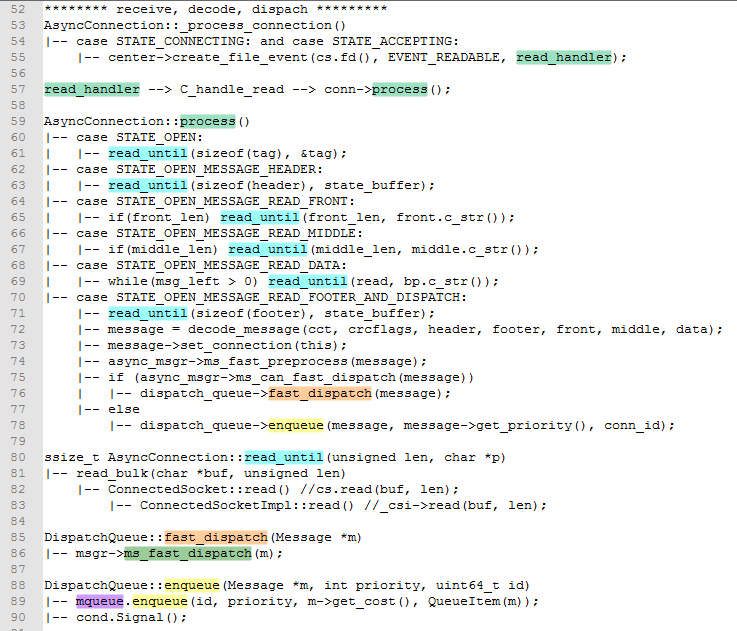


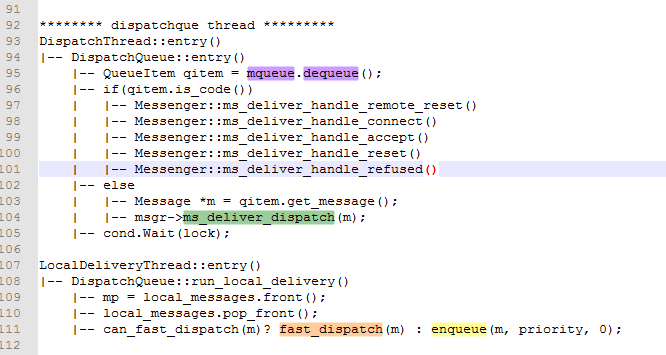
图 15 connect状态迁移图

1. 消息发送流程

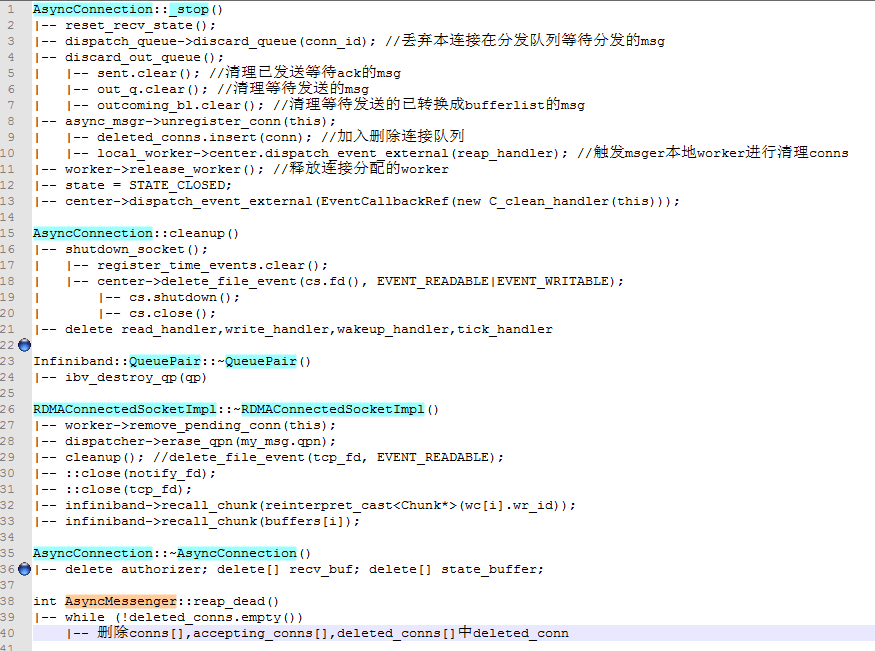


1. 消息接收、分发流程





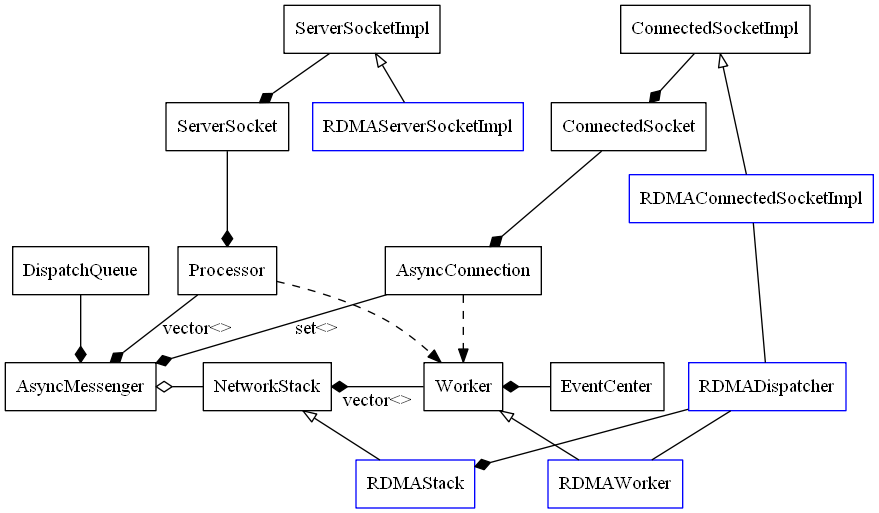
* 连接清除流程



## RDMAStack、Infiniband

### 类结构图

1. RDMAStack在AsyncMessenger中的位置



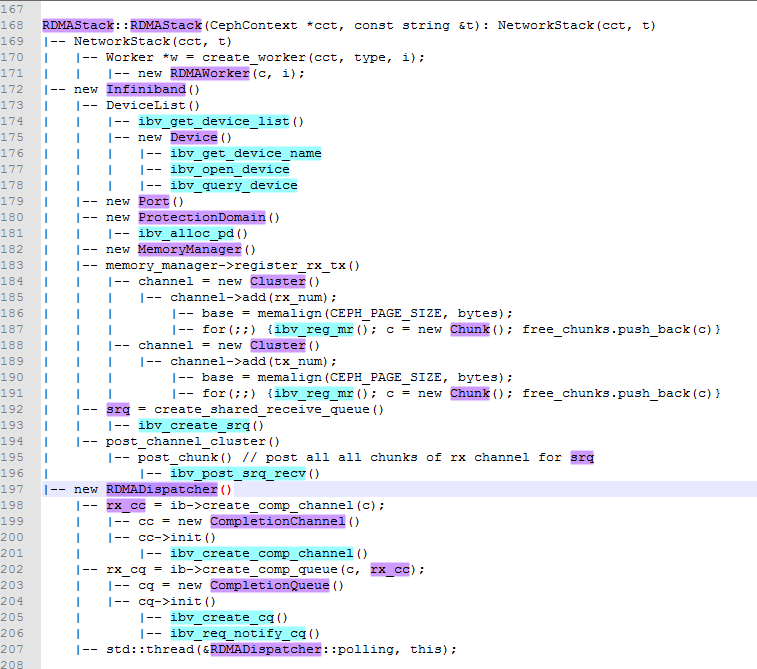
1. RDMAStack类结构图

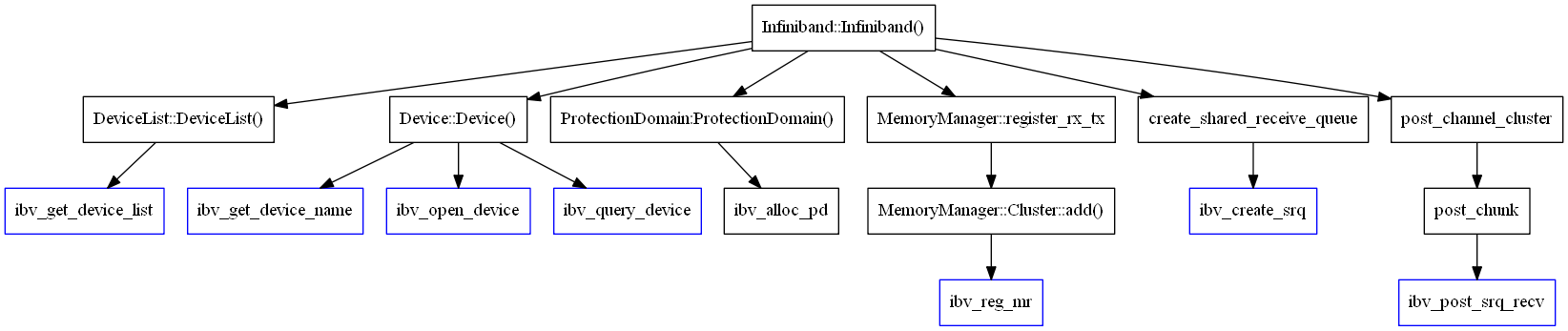


1. Infiniband类结构图



### 创建与初始化





Infiniband QP（Queue Pair）由一个TxQueue和RxQueue组成，RDMAStack采用所有QP共享一个RxQueue（SRQ）。所有QP关联一个全局完成队列CQ，接收、发送完成都会向CQ插入任务完成表项（WC）。CQ关联一个完成通道Channel，当CQ有新WC时，会写Channel fd，通知RDMADispatcher有新CQ事件。



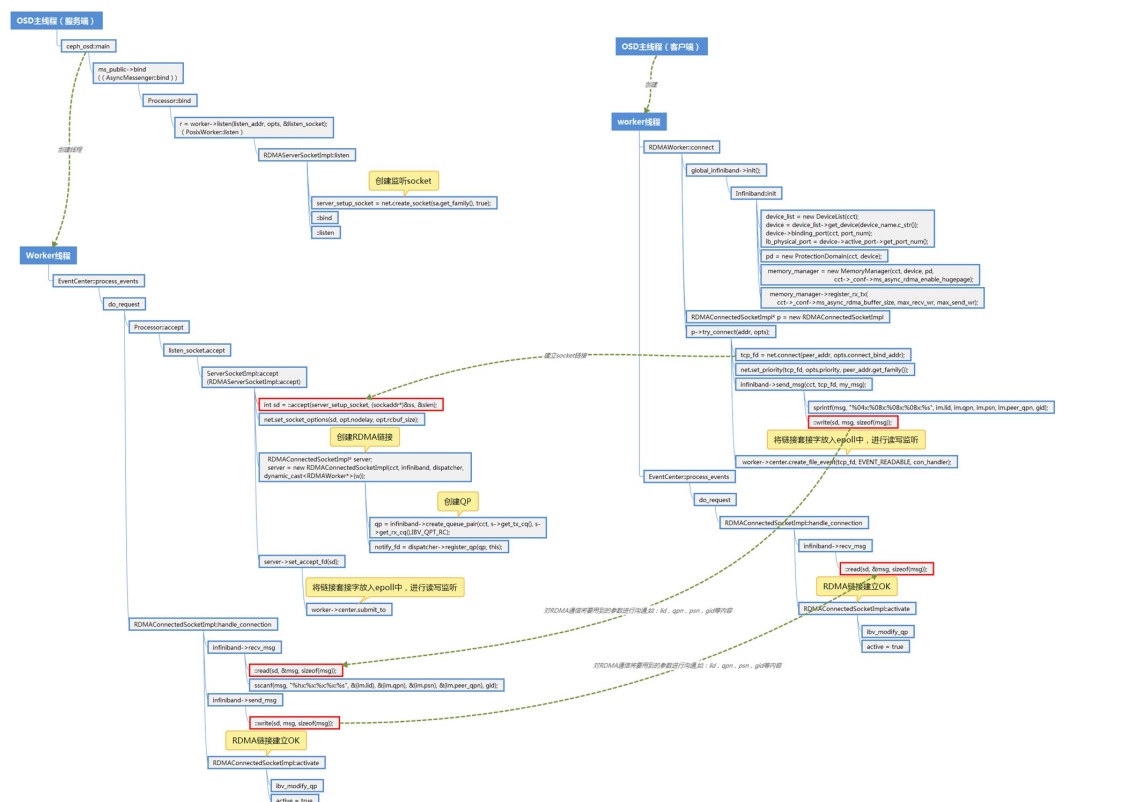
### 连接建立

Server监听并接收Client的连接请求，创建连接socket：RDMAConnectedSoketImpl。RDMAConnectedSoketImpl对象在创建时会自动创建Infiniband QP，和tcp Socket。先通过tcp socket交换QP的lid、QpNumber后，就可激活QP，之后就通过QP进行通信。

1. 时序图



1. 代码调用关系图

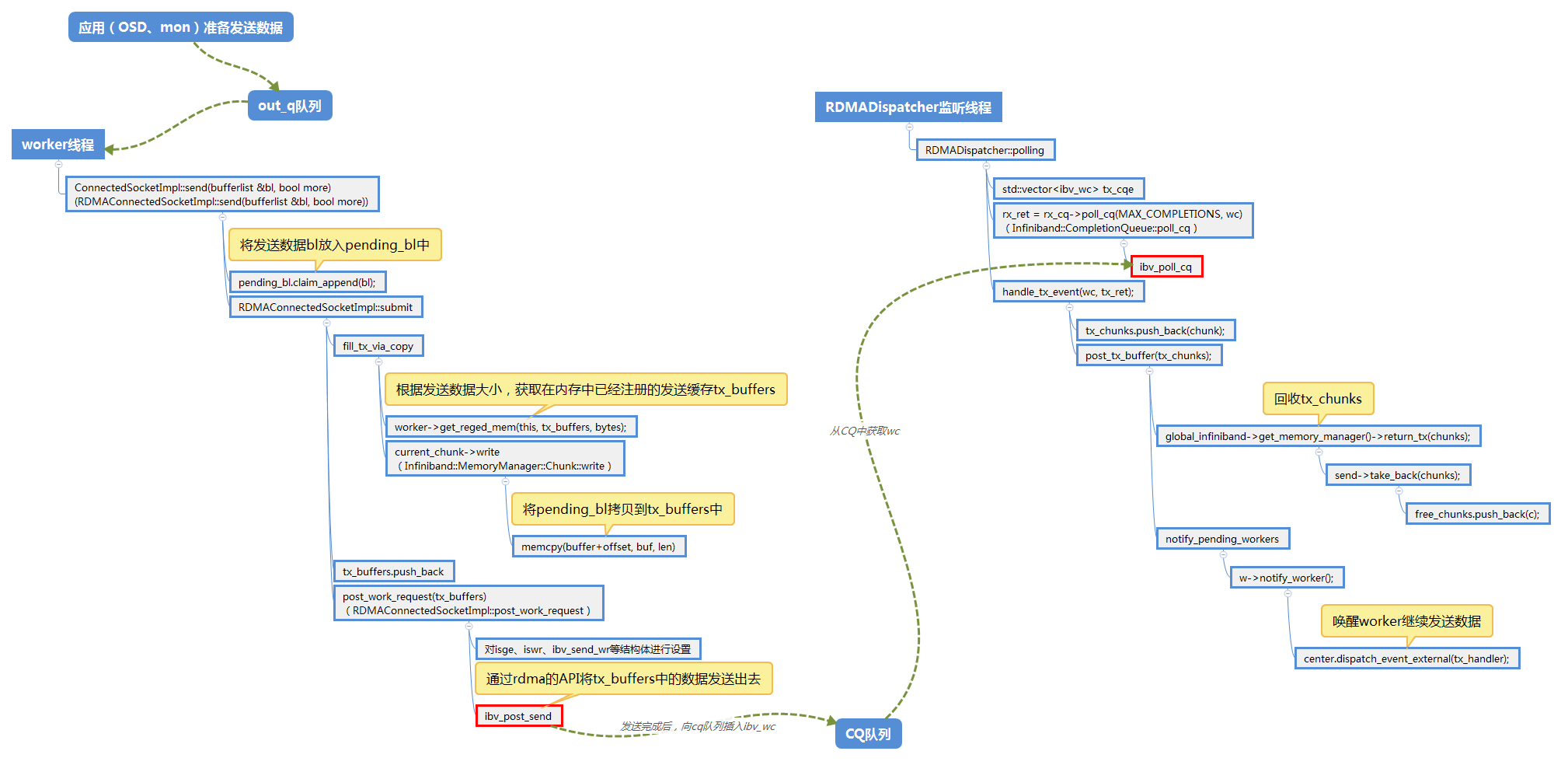


### 消息发送

1. 消息发送数据流



1. 代码调用关系图



### 消息接收

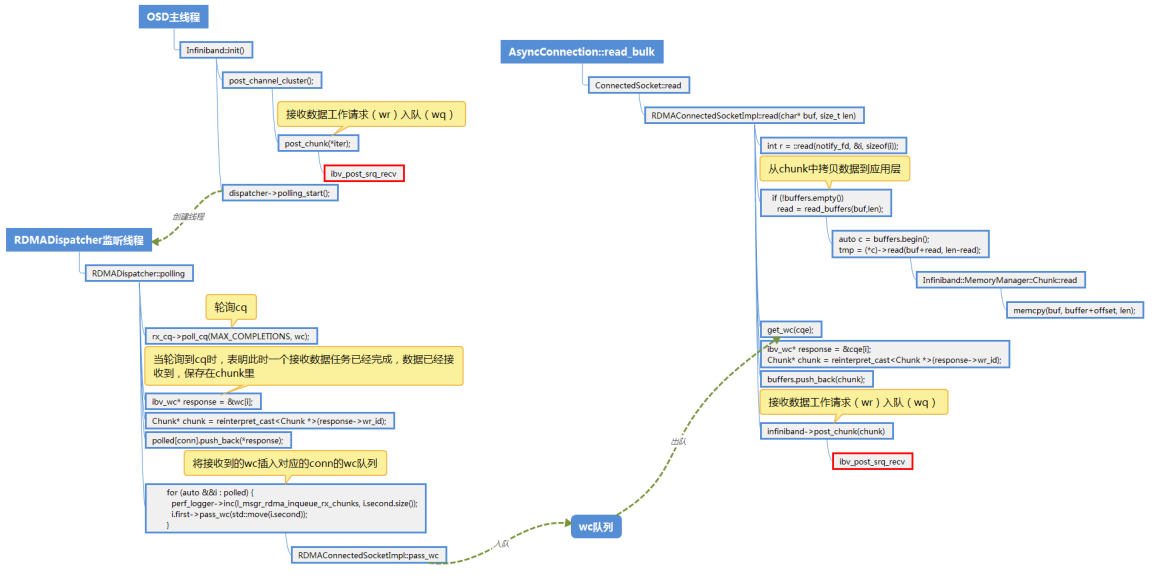
1. 消息接收数据流



1. 消息接收时序图



1. 代码调用关系图



### 清除连接

~QueuePair()中删除infiniband qp，

~RDMAConnectedSocketImpl()中释放接收wc的chunk资源。

## 与业务模块的交互

1. 当业务模块需要发数据时，只需要调用消息发送接口函数即可。
2. 消息发送接口函数会将消息放入消息发送队列out\_q。
3. 消息发送线程从队列out\_q中取消息，发送给对方。
4. 消息接收线程从网络中获取到数据，然后放入工作队列op\_shardewq.
5. 业务处理线程从该工作队列op\_shardewq中取出数据，然后进行处理



图 16 与业务模块的交互

# 网络异常处理（待整理、补充）

## 消息保序

\_\_u32 connect\_seq, peer\_global\_seq;

std::atomic<uint64\_t> in\_seq{0};

std::atomic<uint64\_t> out\_seq{0};

std::atomic<uint64\_t> ack\_left{0};

out\_seq是发送端进行设置，一般新建一个连接时随机产生一个序号，后续在发送消息时out\_seq递增赋值给消息的序号m->seq.

in\_seq是接收方进行设置，当收到消息后，判断该消息是否合法。如果合法，就将in\_seq设置为收到消息的m->seq。

判断是否合法：

当m->seq <= in\_seq；说明收到了重复的老消息，丢弃。

当m->seq > in\_seq + 1; 说明收到了跳过了某些信息，目前是继续处理。

实际上接收端期望的是m->seq = in\_seq + 1

ack\_left表示收到的消息没有ack的数量。

当网络异常导致tcp连接中断后，会调用Pipe::fault进行处理，就是关闭socket，调用requeue\_sent把没有收到ack的消息重新放入out\_q队列头（放在头部以便于可以优先处理），而且out\_seq会递减，后续会不断尝试重新建立连接。这样再重新建立连接重发的消息所带的seq还是跟之前的一样。

因为发送端连接异常调用Pipe::fault里会关闭socket，进行tcp的连接关闭的处理，在接收端继续读的时候读到0认为tcp\_read失败，因此也会调用Pipe::fault()从而调用shutdown\_socket去关闭socket。因此对于连接异常断开后再重新建立连接的情况，in\_seq也不会接着之前的序号，仍然是取决于发送端生成的out\_seq。因而可以保证消息的顺序。

requeue\_sent 在连接异常时，重新入队。从sent中回到out\_q中，待重新发送

discard\_requeued\_up\_to 表示从out\_q中丢掉某个序列号之前的消息。因为此时接收端可能已经收到该序号的消息了，不需要重新发送了

## 连接竞争

## 重建连接

## 错误处理

# Ceph集群网络

Ceph集群网络包括前端公共网络和后端集群网络，前端网络用于业务访问和集群管理，后端网络主要用于OSD间数据处理（复制、均衡、恢复、检查）。（管理网络在此省略）



图 17 网络拓补图

Ceph功能模块（MON、OSD、MDS、RadosClient）通过网络通信模块（Messenger）实现（节点间和节点内）进程间通信。

Messenger可分为Server Messenger和Client Messenger。Server Messenger监听端口，被动接受连接；Client Messenger主动请求建立连接（Server Messenger也可以主动向另一个Server Messenger请求建立连接）。Messenger之间建立连接后，可进行双向通信。



图 18 网络连接图

Ceph messenger主要有：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Messenger | 类型 | 网络 | 监听 | 进程 | 用途 |
| "mon" | MON | public | 6789 | ceph-mon |  |
| "mds" | MDS | public | 6800+ | ceph-mds |  |
| "client" | OSD | public | 6800+ | ceph-osd | 响应OSD对象操作 |
| "cluster" | OSD | cluster | 6800+ | ceph-osd | OSD复制/均衡/恢复/回填/检查 |
| "hb\_front\_server" | OSD | public | 6800+ | ceph-osd | OSD心跳 |
| "hb\_back\_server" | OSD | cluster | 6800+ | ceph-osd | OSD心跳 |
| "hbclient" | OSD | both |  | ceph-osd | OSD心跳 |
| "ms\_objecter" | OSD | public |  | ceph-osd | OSD向其他OSD请求对象操作 |
| "server" | MGR | public | 6800+ | mgr |  |
| "radosclient" | CLIENT | public |  |  | mon client，osd client，mgr client |
| "client" | CLIENT | public |  |  | mds client |

待补充。。。。。。

# 附录

参考文档：

《NAS V1R2 RDMA预研 报告.docx》