

用UCX实现BRPC对RDMA的支持

徐逸锋

BRPC简介

- BRPC是Curve的基础通讯框架
- 支持远程过程调用
 - C++
 - TCP传输
 - bthread协程(m:n调度，减少基于内核的下文切换，减少cache miss)
- 多协议支持
 - baidu_std, http, grpc...
 - protobuf

BRPC简介

- Client/Server架构
- 使用Protobuf定义协议文件
 - 例如：echo.proto:

```
syntax="proto2";
package example;

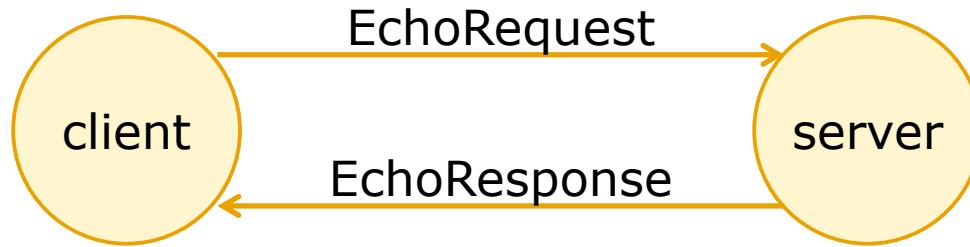
option cc_generic_services = true;

message EchoRequest {
    required string message = 1;
};

message EchoResponse {
    required string message = 1;
};

service EchoService {
    rpc Echo(EchoRequest) returns (EchoResponse);
};
```

BRPC简介



BRPC简介

- Channel类

- 代表一个连接，Client通过Channel发送请求和接收应答

- Server类

- 代表一个服务器，可以注册不同的接口服务，例如上面的EchoService

BRPC SERVER

```
// Generally you only need one Server.  
brpc::Server server;  
  
// Instance of your service.  
example::EchoServiceImpl echo_service_impl;  
  
// Add the service into server. Notice the second parameter, because the  
// service is put on stack, we don't want server to delete it, otherwise  
// use brpc::SERVER_OWNS_SERVICE.  
if (server.AddService(&echo_service_impl,  
                      brpc::SERVER_DOESNT_OWN_SERVICE) != 0) {  
    LOG(ERROR) << "Fail to add service";  
    return -1;  
}
```

BRPC SERVER

```
// Your implementation of EchoService
class EchoServiceImpl : public EchoService {
public:
    EchoServiceImpl() {}
    ~EchoServiceImpl() {};
    void Echo(google::protobuf::RpcController* cntl_base,
              const EchoRequest* request,
              EchoResponse* response,
              google::protobuf::Closure* done) {
        brpc::ClosureGuard done_guard(done);
        brpc::Controller* cntl =
            static_cast<brpc::Controller*>(cntl_base);

        // Echo request and its attachment
        response->set_message(request->message());
        if (FLAGS_echo_attachment) {
            cntl->response_attachment().append(cntl->request_attachment());
        }
    }
};
```

BRPC client

```
int log_id = 0;
while (!brpc::IsAskedToQuit()) {
    example::EchoRequest request;
    example::EchoResponse response;
    brpc::Controller cntl;

    request.set_message(g_request);
    cntl.set_log_id(log_id++);
    cntl.request_attachment().append(g_attachment);

    stub.Echo(&cntl, &request, &response, NULL);
    if (!cntl FAILED()) {
        g_latency_recorder << cntl.latency_us();
```

BRPC EndPoint

EndPoint是一个代表通讯地址的数据结构，是一个C++类。

字段： ip , port

.在Socket创建时需要提供EndPoint

.Socket::Connect时需要RemoteEndPoint

.Accept的Socket可以获得RemoteEndPoint

BRPC Socket对象

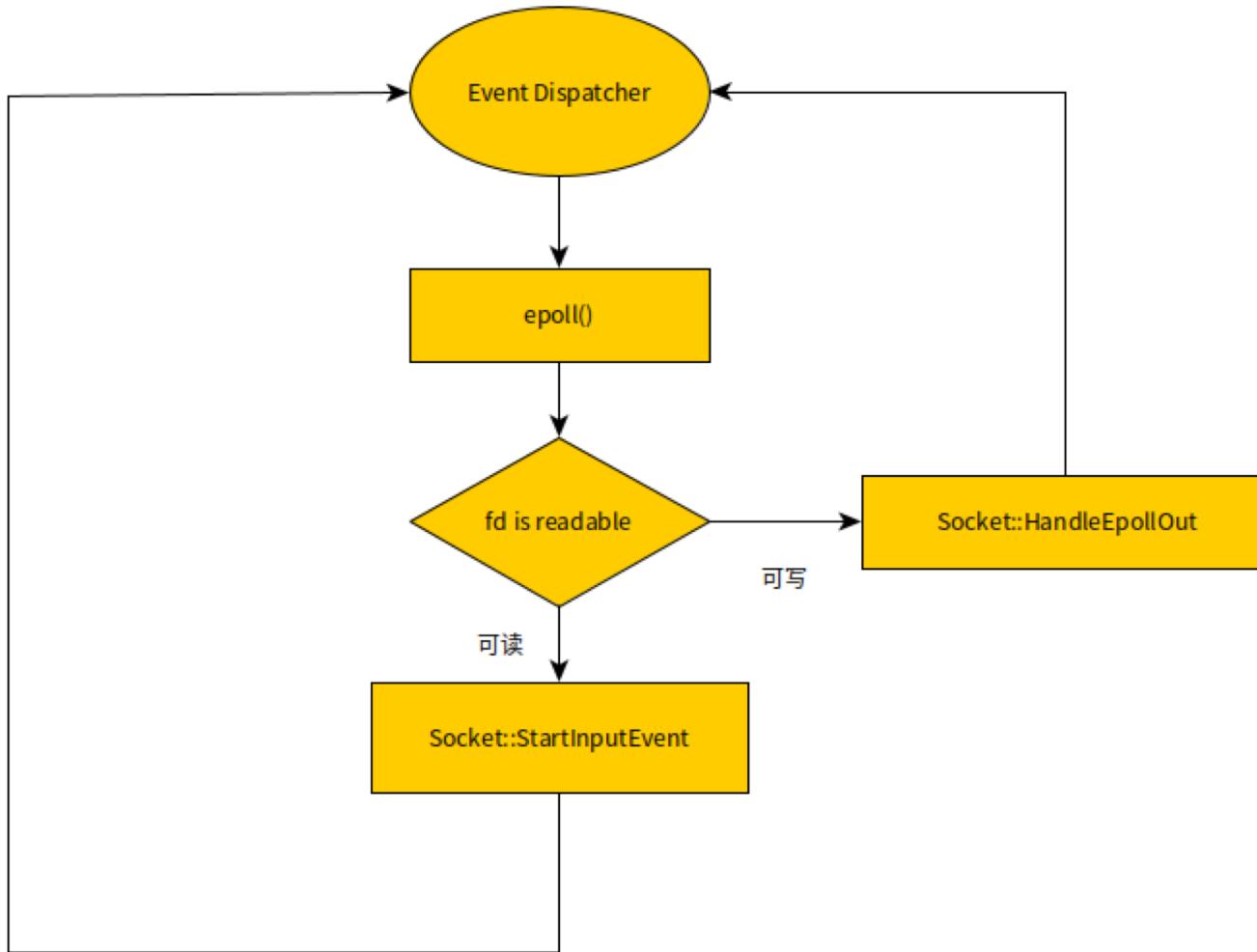
- brpc最终的网络通讯都集中在socket对象里面
- 读socket通过EventDispatcher触发
- 上层发送网络数据通过写socket完成，不能立刻完成的，则去启动后台bthread去完成。

BRPC SocketMap

- 根据EndPoint作为一个map的Key，Value是Socket对象
- Socket对象引用计数，多个Channel可以共享一个Socket对象
- 往SocketMap里调用Insert，要么返回已经存在的Socket对象（引用计数加一），要么创建一个新的

BRPC EventDispatcher

- 是socket事件分发的中心
- 使用epoll和边沿触发
- 提供监视一个fd是否可读写，并调用对应socket对象的成员函数



Socket 输入事件处理



Socket options

- 是创建socket的参数
- 主要成员：
 - fd 是socket文件句柄
 - void (*on_edge_triggered_events)(Socket*)
- 可读事件的回调函数

Server创建Socket Listener

把系统调用创建的listen socket fd传给Socket::Create，获得一个Socket对象

```
SocketOptions options;
options.fd = listened_fd;
options.user = this;
options.on_edge_triggered_events = OnNewConnections;
if (Socket::Create(options, &_acception_id) != 0) {
    LOG(FATAL) << "Fail to create _acception_id";
    return -1;
}
```

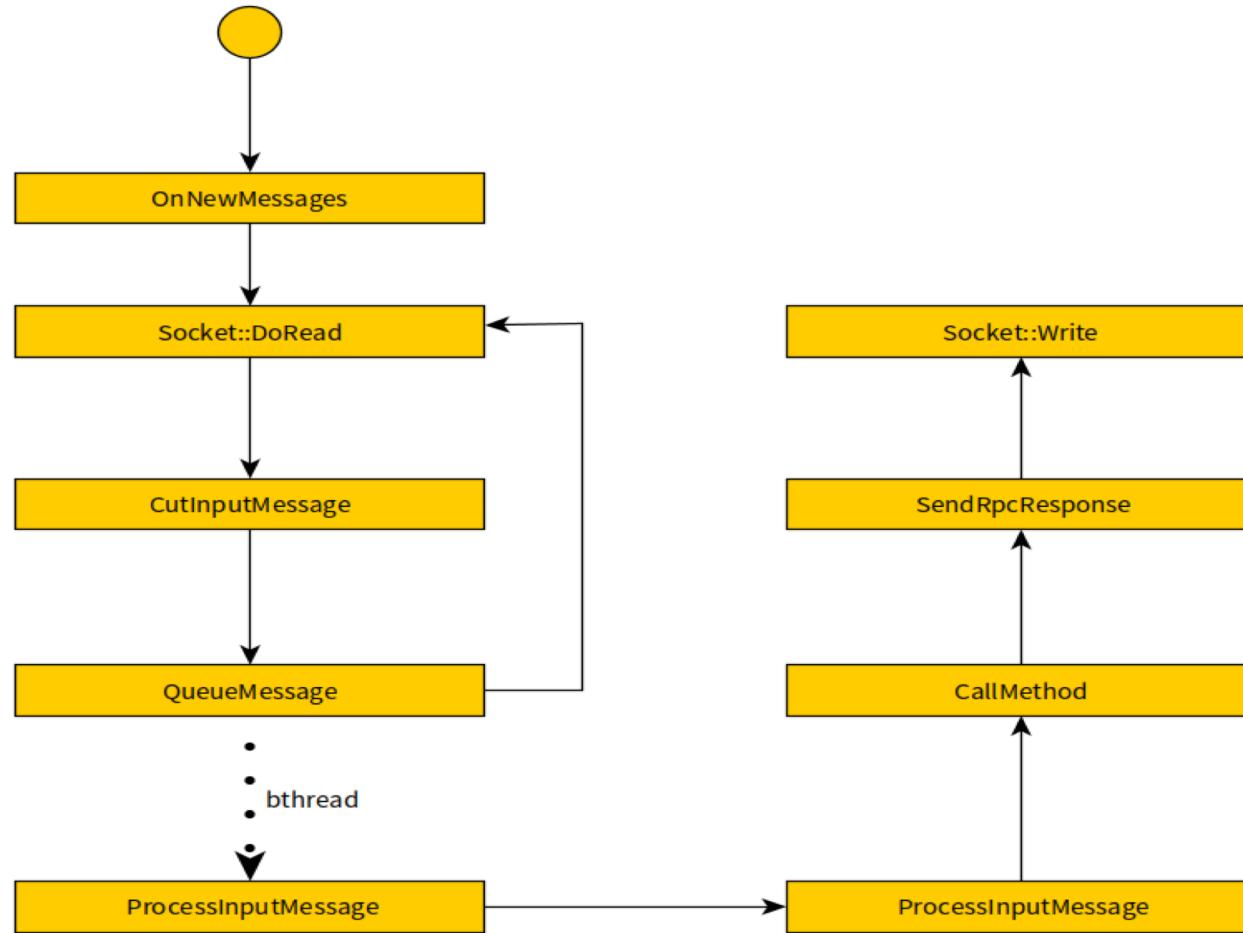
Socket Listener::OnNewConnections

Listener 获得一个socket fd后，创建通讯Socket。

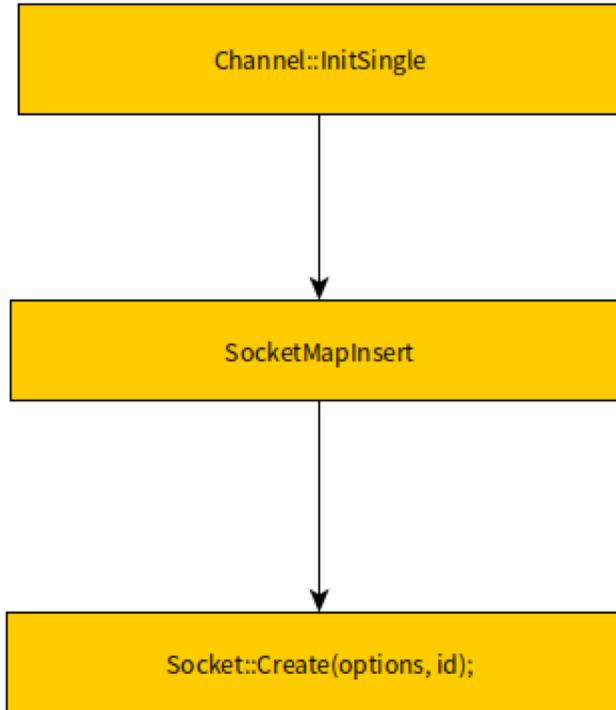
SocketOptions关键字段： fd, on_edge_triggered_events

```
SocketId socket_id;
SocketOptions options;
options.keytable_pool = am->_keytable_pool;
options.fd = in_fd;
options.remote_side = butil::EndPoint(*(sockaddr_in*)&in_addr);
options.user = acceptor->user();
options.on edge triggered events = InputMessenger::OnNewMessages;
options.initial_ssl_ctx = am->_ssl_ctx;
if (Socket::Create(options, &socket_id) != 0) {
    LOG(ERROR) << "Fail to create Socket";
    continue;
}
```

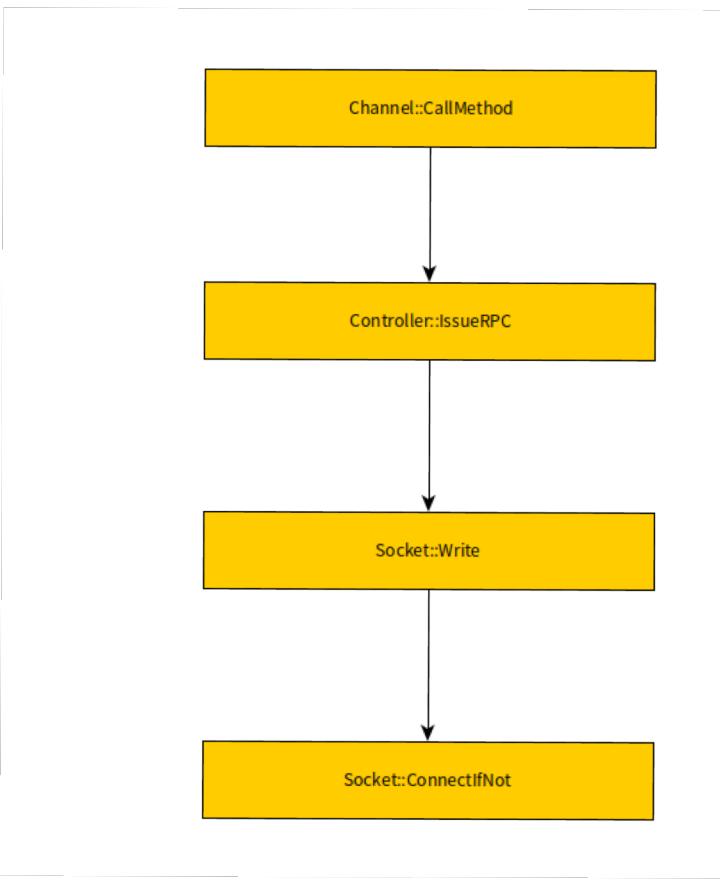
例子：Request输入处理



Channel创建Socket



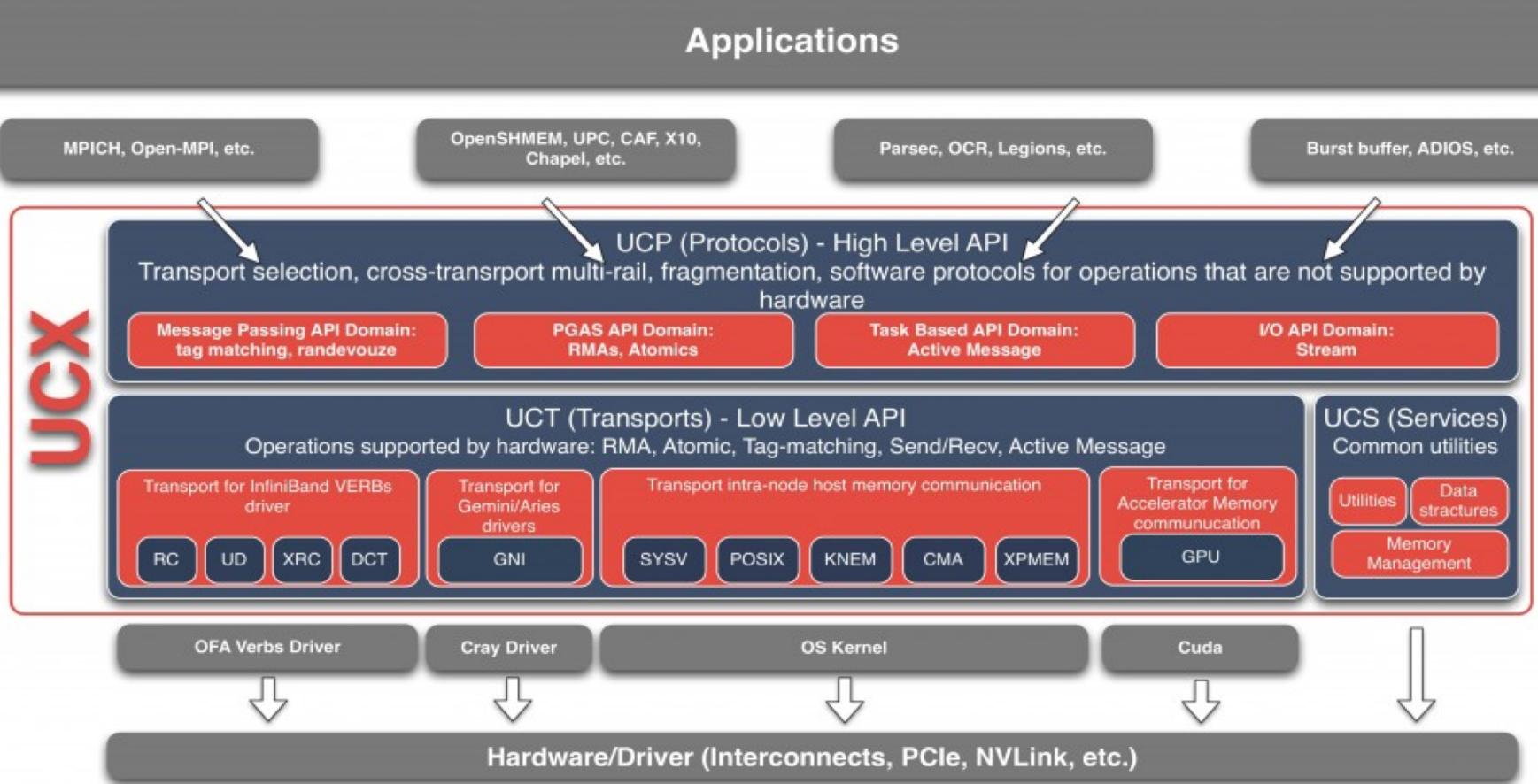
Channel远程调用的发起



UCX

- NVIDIA Mellanox 开源项目
- 支持RDMA，TCP，Shared memory等
- 能透明支持多个链路传输，例如多网卡bond
- 编译成.so或lib的方式，可以集成到应用程序里
- 有完善的配置功能，ucx_info可以dump配置信息
- 有性能测试工具
- 比较详细的文档

Architecture



UCS

- 是一些工具代码，例如

- 链表

- hash table

- epoll event loop

- memory register cache

- config file

UCT

- 特点是比较原始，开销小，但是没有很强的功能
- 是网络接口层，主要功能是网卡发现和远程内存传输支持，提供component查询和memory domain的打开
- 一个component包含若干 memory domain resource,一个memory domain又可以包含若干个transport
- 当前支持的memory domain : ibverbs, tcp, shared memory
- 详情可以见examples/uct_hello_world.c

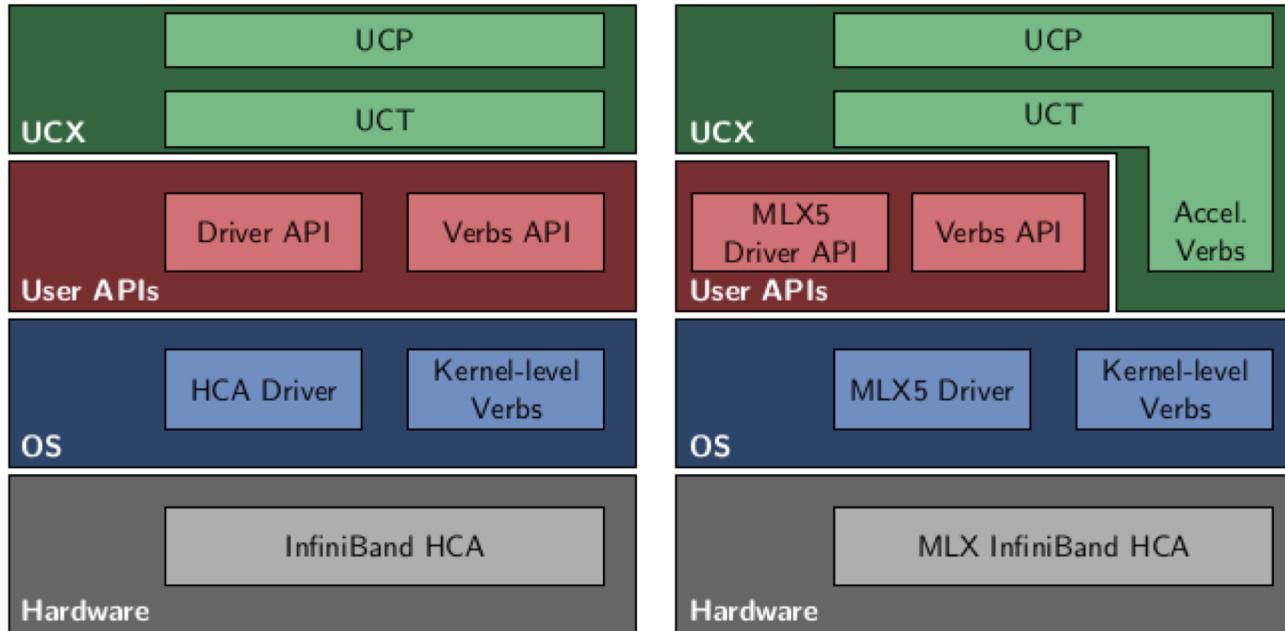
UCT

- Memory domain open/close
- Active message
- Memory get/put
- Memory atomic fetch, compare and set
- Tag match
- client/server模式的Listener, Ep(endpoint)

UCP

- 构建于uct之上，实现更加高级的功能，容易使用，但有一定开销。
- UCT和UCP两者都有context概念，但是UCT只对一块网卡，而UCP把若干个UCT组合起来，自动选择最快路径传输。
- 高级特性
 - 大消息报文的自动分片传输
 - Active message, atomic operation, tag match, stream

典型的RDMA栈



(a) Common InfiniBand OFED stack
(b) MLX5 InfiniBand OFED stack and UCX with accelerated Verbs.

UCX 编程的一些基本概念

• Context

- 收集机器资源（内存，网卡等），在应用的各个部分共享

• Worker

- 完成ucx的功能，可以在应用程序中调用的函数（不是单独执行的线程）

• Listener

- 接收连接请求

• Ep

- 连接对象，在ep上请求发送和接收

UCP 消息接口类型

- Active message

- 速度最快，被brpc使用作为消息传递
- 消息通过回调函数接收
- 消息异步发送

- Tag

- MPI使用

- Stream

- 官方不推荐

WORKER

- .worker是UCX通讯中的核心概念，它是一个进度引擎(progress engine)
- .worker既不是协程也不是线程，而是一个状态机，可以通过不停地调用ucp_worker_progress(worker)完成功能。如果你用过libuv或者libevent的evbuffer，它们有点像proactor，使用libuv时不停调用uv_run(UV_RUN_NOWAIT)，就可完成buffer自动读写。

WORKER

- Busy poll

- Busy poll可以有效降低时延，但是在空闲时浪费CPU

- Wait

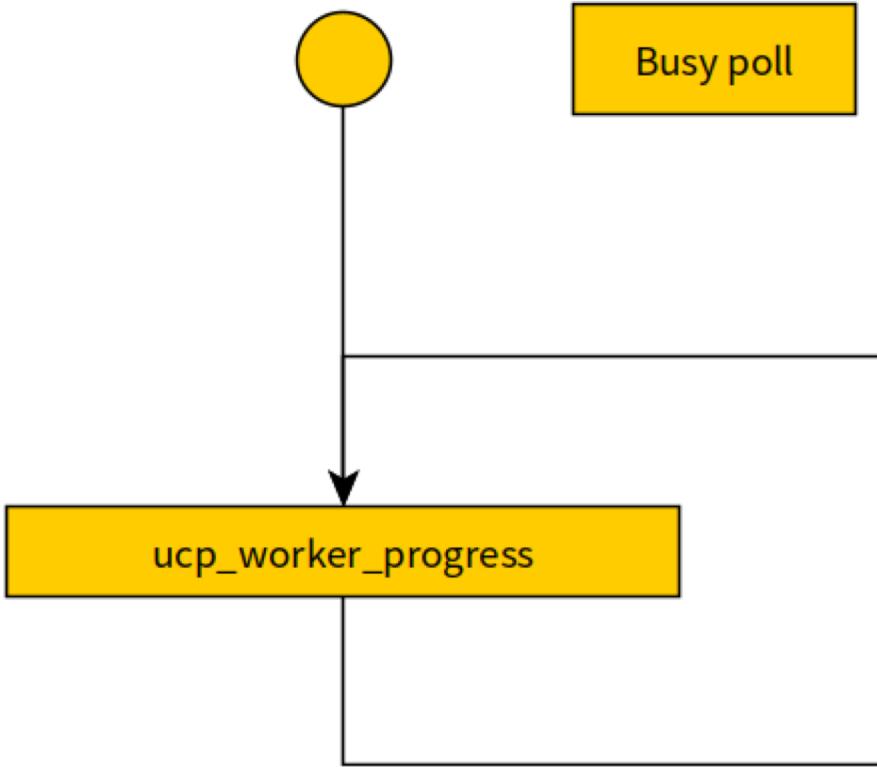
- 会增加时延，但是节省CPU使用

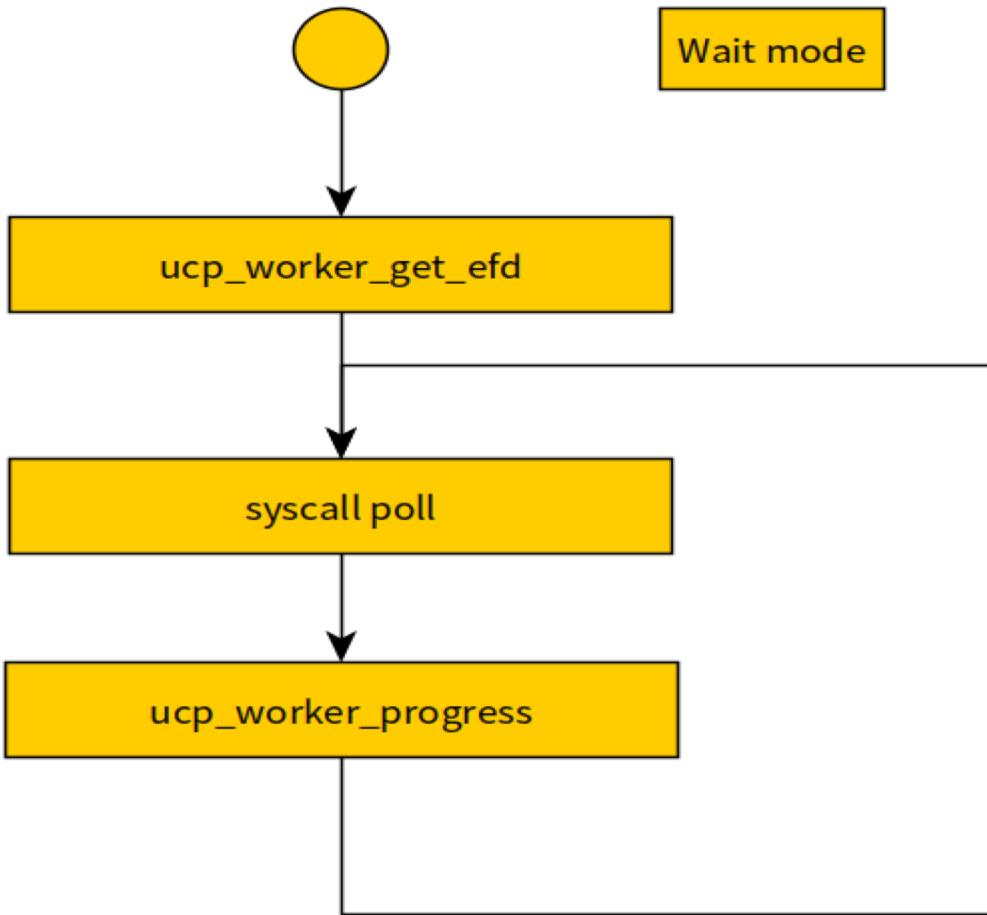
- 通过ucp_worker_get_efd(*ucp_worker, efd)获得轮询文件句柄

- 调用pollefd等待有任务执行，然后再调用ucp_worker_progress()

- /dev/cpu_dma_latency 禁止power-saving模式

- 由于rdma速度很快，内核调度时延对性能影响很大。关键应用应开启busy poll。





BRPC怎么指定使用UCX?

```
class Channel : public ChannelBase {  
friend class Controller;  
friend class SelectiveChannel;  
public:  
    Channel(ProfilerLinker = ProfilerLinker());  
    ~Channel();  
  
    // Connect this channel to a single server whose address is given by the  
    // first parameter. Use default options if `options` is NULL.  
    int Init(butil::EndPoint server_addr_and_port, const ChannelOptions* options);  
    int Init(const char* server_addr_and_port, const ChannelOptions* options);  
    int Init(const char* server_addr, int port, const ChannelOptions* options);
```

修改 BRPC ChannelOptions

增加字段：

```
// Use ucp transport  
bool use_ucp;
```

BRPC的Server开启RDMA

server类有如下成员函数, 如何指定开启ucx连接?

```
// Start on an address in form of "0.0.0.0:8000".
int Start(const char* ip_port_str, const ServerOptions* opt);
int Start(const buutil::EndPoint& ip_port, const ServerOptions* opt);
// Start on IP_ANY:port.
int Start(int port, const ServerOptions* opt);
// Start on `ip_str` + any useable port in `range`
int Start(const char* ip_str, PortRange range, const ServerOptions *opt);
```

修改 BRPC ServerOptions

- .ServerOptions添加成员
- 当前取舍的：TCP总是可用的， UCX作为选项

```
// Enable ucp listener
bool enable_ucp;

// Ucp listener ip address
std::string ucp_address;

// Ucp listener port
uint16_t ucp_port;
```

Ucp Context

- 只有一个全局对象，使用下列函数获取
- `UCP_Context* get_or_create_ucp_ctx()`
- 指定了FEATURE_AM, 多线程共享

命令行参数控制context的属性

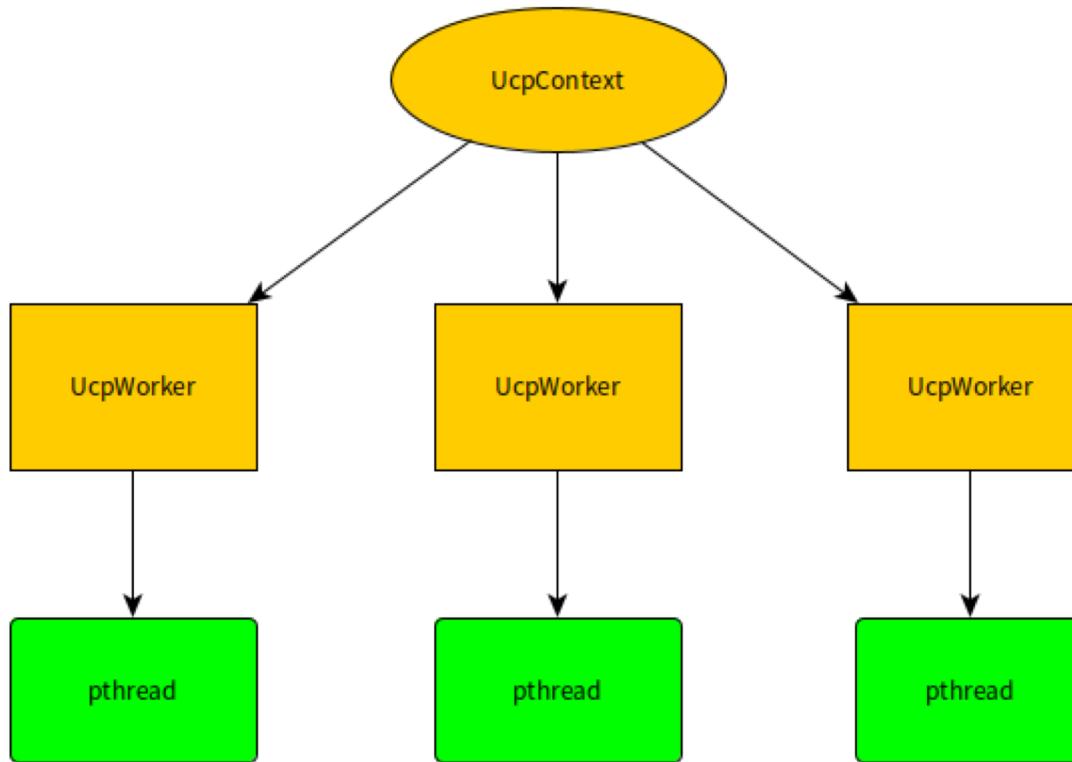
```
DEFINE_int32(brpc_set_cpu_latency, -1, "Set cpu latency in microseconds");
DEFINE_string(brpc_ucp_error_mode, "none", "Ucp error mode(none,peer");
```

- --brpc_ucp_error_mode 缺省是none，是的本地通讯使用shared memory成为可能
- --brpc_set_cpu_latency 非-1, 设置intel cpu节电模式，可能减少硬件延迟。
- /dev/cpu_dma_latency
- root特权
- Busy poll下不要开启，可能导致电耗过高、cpu降速

Ucp Worker

- 创建UcpWorker，封装ucp worker和逻辑。
- 是整个ucp实现RDMA的核心。
- 系统可以有多个worker，共享使用一个UcpContext。
- 不同的连接分配到不同的worker,一般情况下只需要一个worker足够应付网络通讯。
- worker逻辑在一个pthread中运行。

1个 UcpContext: N个 UcpWorker



连接管理器UcpCm

- 连接管理类
 - 全局唯一对象
 - 通过 UcpCm * get_or_create_ucp_cm(void) 获取
 - 完成连接的接受
 - 完成连接的创建
 - 监视 brpc::Socket 类关闭文件句柄
 - 连接以文件句柄表示

连接管理器UcpCm

- 连接以文件句柄返回

```
-int Accept(ucp_conn_request_h req);  
-int Connect(const butil::EndPoint &peer);  
-UcpConnectionRef GetConnection(int fd1);
```

连接管理器UcpCm

- .Brpc socket代码不少地方需要文件句柄表示连接，使用句柄可以减少代码修改。例如 SocketOptions.fd为-1表示尚未连接。
- .UcpCm返回的文件句柄实际上是pipe的写端句柄
- .记得brpc的event dispatcher是边沿触发
- .写端句柄永远不会触发可读事件
- .写端句柄第一次epoll会返回可写，可写是brpc判断连接成功的措施
- .UcpCm从来不会写入pipe，如果pipe有可读字节，会打印错误，说明有地方遗漏了修改。
- .Socket通过关闭UcpCm返回的句柄来关闭连接。此举和Socket原来代码一样，减少了修改。 UcpCm检测到pipe读端可读，关闭UcpConnection。
- .以上修改实际上绕过了BRPC的Event dispatcher触发读写机制，UCX自己完成发送接收 44

连接管理器UcpCm

- 连接管理类
 - 全局唯一对象
 - 通过 UcpCm * get_or_create_ucp_cm(void) 获取
 - 完成连接的接受
 - 完成连接的创建
 - 监视 brpc::Socket 类关闭文件句柄
 - 连接以文件句柄表示

UcpAcceptor

- 连接接收器类
- 独占一个ucp_worker
- 部分代码从brpc::Acceptor类拷贝
- ucp部分重新设计

UcpAcceptor

• 处理收到的连接

```
SocketOptions options;  
-options.fd = get_or_create_ucp_cm()->Accept(conn_request);
```

UcpConnection

- 封装ucp_ep

- 支持读写接口

- `ssize_t Read(butil::IOBuf *out, size_t n);`
- `ssize_t Write(butil::IOBuf *buf);`
- `ssize_t Write(butil::IOBuf *data_list[], int ndata);`

UcpWorker的实现

- 封装 ucp_worker_h
- 一个UcpWorker对用多个UcpConnection，由UcpCm分配。
- 在pthread运行，busy poll or wait mode。

UcpWorker实现

提供Accept

```
int UcpWorker::Accept(UcpConnection *conn, ucp_conn_request_h req)
{
    BAIDU_SCOPED_LOCK(mutex_);
    int ret = CreateUcpEp(conn, req);
    if (ret == 0) {
        AddConnection(conn);
        MaybeWakeup();
    }
    return ret;
}
```

UcpWorker的实现

• 提供 Connect。

```
int UcpWorker::Connect(UcpConnection *conn, const butil::EndPoint &peer)
{
    BAIDU_SCOPED_LOCK(mutex_);
    int ret = create_ucp_ep(ucp_worker_, peer, ErrorCallback, this, &conn->ep_)
    if (ret == 0) {
        AddConnection(conn);
        conn->remote_side_ = peer;
        auto str = butil::endpoint2str(peer);
        conn->remote_side_str_ = str.c_str();
        MaybeWakeup();
    }
    return ret;
}
```

UcpWorker的实现

- 提供Release Connection。
 - 在UcpCm决定关闭连接时

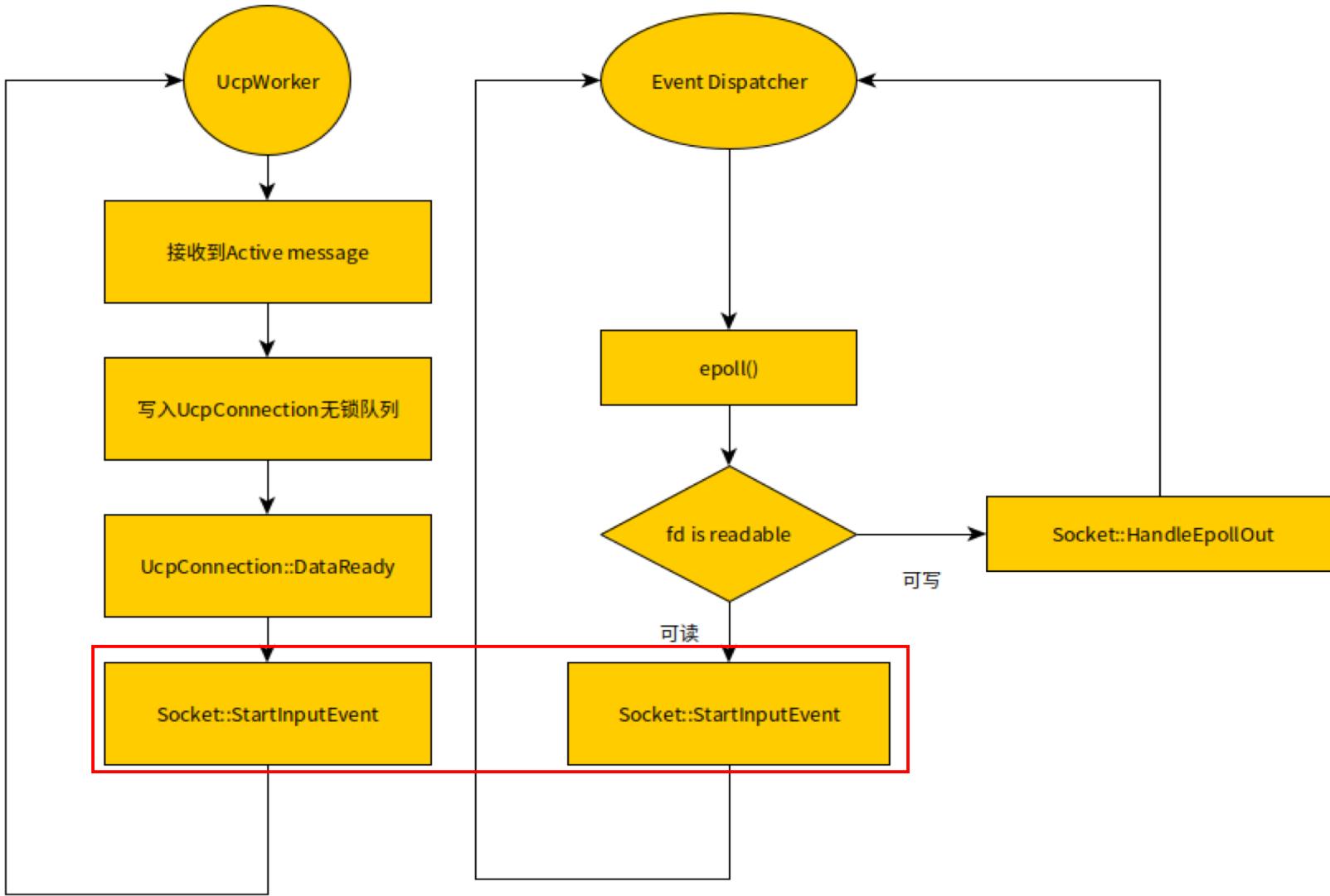
```
// Release the connection object indexed by ep, and gracefully disconnect it
void UcpWorker::Release(UcpConnectionRef conn)
{
    ucs_status_ptr_t request;
    ucp_ep_h ep = conn->ep_;

    // Lock the worker
    BAIDU_SCOPED_LOCK(mutex_);

    conn->state_ = UcpConnection::STATE_CLOSED;
```

UcpWorker的实现

- 使用了ucp active message
 - 当消息很短时， ucx使用内部缓冲提供给brpc(比较快)
 - 当消息很大时， 由brpc提供接收缓冲区 (rndv,rendezvous)
- 阀值可调
 - 接收和发送使用无锁队列
 - UcpWorker接收时写入UcpConnection的无锁队列
 - brpc发送时写入UcpWorker的无锁发送队列



UcpWorker实现

- 提供命令行参数调整ucp worker的行为

- brpc_ucp_worker_busy_poll为true时使用busy poll

- brpc_ucp_worker_poll_time为wait模式时，一次突发的poll的时长

- brpc_ucp_deliver_out_of_order为true，接收端乱序提交

- brpc_ucp_close_flush, release connection时发送未完成的报文，不是必须的，因为brpc通常是需要接收应答的，服务器端一般不主动关闭连接，客户端主动关闭，自己负责是否有未接收完的应答。

修改BRPC的EndPoint

- 原始的EndPoint类，不能识别网络连接的类型，默认只有TCP
 - 现在有了UCX, 需要添加一个字段说明是UCX连接，防止在函数传递参数时丢失连接类型信息。
 - UCX的地址依然是tcp地址。

```
struct EndPoint {
    enum { TCP, UCP, UNIX };
    EndPoint() : ip(IP_ANY), port(0), kind(TCP), socket_file("") {}
    EndPoint(ip_t ip2, int port2) : ip(ip2), port(port2), kind(TCP), socket_file("") {}
    EndPoint(ip_t ip2, int port2, int k) : ip(ip2), port(port2), kind(k), socket_file("") {}
```

修改 Socket Connect

```
int Socket::Connect(const timespec* abstime,
                    int (*on_connect)(int, int, void*), void* data) {
    if (_ssl_ctx) {
        _ssl_state = SSL_CONNECTING;
    } else {
        _ssl_state = SSL_OFF;
    }
    butil::fd_guard sockfd;

    if (is_ucp_connection()) {
        ssl_state = SSL_OFF; // FIXME
        sockfd.reset(get_or_create_ucp_cm()->Connect(_remote_side()));
    } else {
        if (butil::is_unix_sock_endpoint(remote_side())) {
            sockfd.reset(socket(AF_LOCAL, SOCK_STREAM, 0));
        } else {
            sockfd.reset(socket(AF_INET, SOCK_STREAM, 0));
        }
    }
}
```

修改Socket Connect

```
if (is_uctp_connection()) {
    UcpConnectionRef conn = get_or_create_uctp_cm()->GetConnection(sockfd);
    if (conn) {
        int rc = conn->Ping(abstime);
        if (rc)
            return rc;
        goto out;
    } else {
        errno = ENOTCONN;
    }
    return -1;
}

if (WaitEpollOut(sockfd, false, abstime) != 0) {
    PLOG(WARNING) << "Fail to wait EPOLLOUT of fd=" << sockfd;
    return -1;
}
if (CheckConnected(sockfd) != 0) {
    return -1;
}
```

修改Socket::DoRead

```
if (is_ucp_connection())
    return _ucp_conn->Read(&_read_buf, size_hint);
return _read_buf.append_from_file_descriptor(fd(), size_hint);
```

.UcpWorker在接收到ucp_ep上的断开事件时，设置UcpConnection处于Error状态，再调用UcpConnection::DataReady，进而调用Socket::StartInputEvent

.UcpConnection的Read函数发现了错误状态，于是返回读错误，进而导致Brpc关闭socket，而我们的socket里的fd是pipe的写端，当比关闭时，UcpCm检测到pipe读端fd可读并且EOF，进而检测到UcpConnection需要关闭。

修改Socket::StartWrite

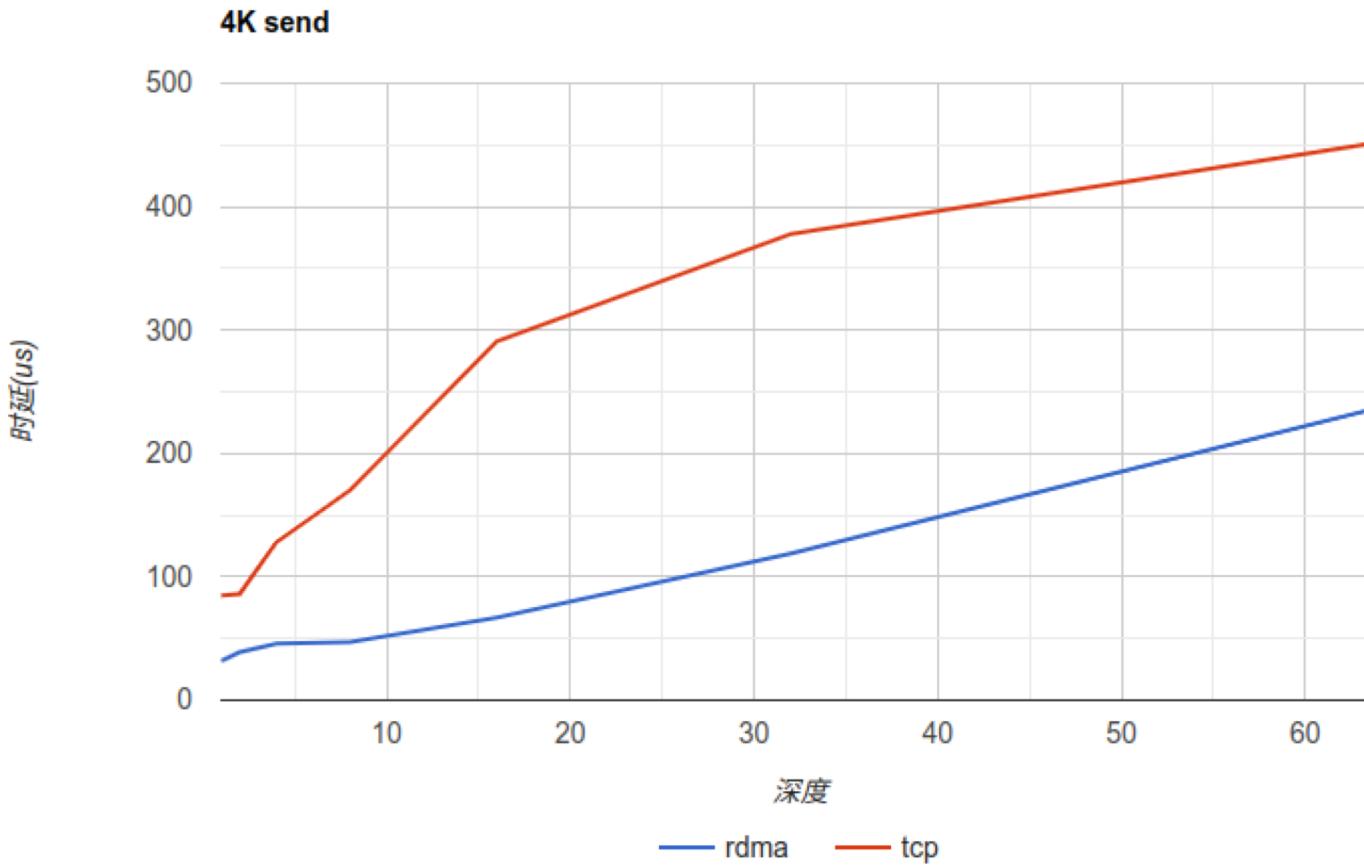
```
if (is_ucp_connection())
    nw = _ucp_conn->Write(&req->data);
else
    nw = req->data.cut_into_file_descriptor(fd());
```

ucp_conn的写总是提交给ucp worker的，不会阻塞，所以不会用到 brpc eve

4K 数据RPC时延比较(us) (双网卡bond)

深度	RDMA	TCP
1	32	85
2	39	86
4	46	128
8	47	170
16	67	291
32	119	378
64	237	452

4K大小时延比较(us)



CurveBS FIO

FIO 4K 随机写

UCX 配置双网卡 (UCX_MAX_EAGER_RAILS=2, UCX_MAX_RNDV_RAILS=2)

时延单位: us

QD	RDMA-IOPS	clat avg	clat 90%	TCP-IOPS	clat avg	clat 90%	scale
1	3983	247	258	2611	379	424	1.525
2	7908	249	258	5313	372	408	1.488
4	15.9k	247	251	10.5k	376	404	1.514
8	30.8k	256	249	19.7k	403	429	1.563
16	55.3k	286	260	35.3k	450	474	1.567
20	65.2k	304	269	41.5k	478	506	1.571
24	73.8k	322	273	47.5k	502	529	1.553
28	80.6k	344	285	52.3k	532	562	1.541
32	87.3k	363	297	57.0k	558	578	1.532
64	116k	549	510	78.8k	808	1020	1.472
96	128k	745	1254	89.9k	1063	1909	1.423
128	137k	932	2180	96.3k	1324	2999	1.422

现状

- 开源分支:<https://github.com/opencurve/incubator-brpc.git>

- curve主干分支

- ucx_am当前rdma分支

- .对brpc的改动不大，加入的模块基本上独立

- .降低了开发难度

- .ucx满足我们对rdma支持的需求

- .已经测试通过了curve验证，取得了不错的性能提升

Thank You !



扫一扫上面的二维码图案，加我为朋友