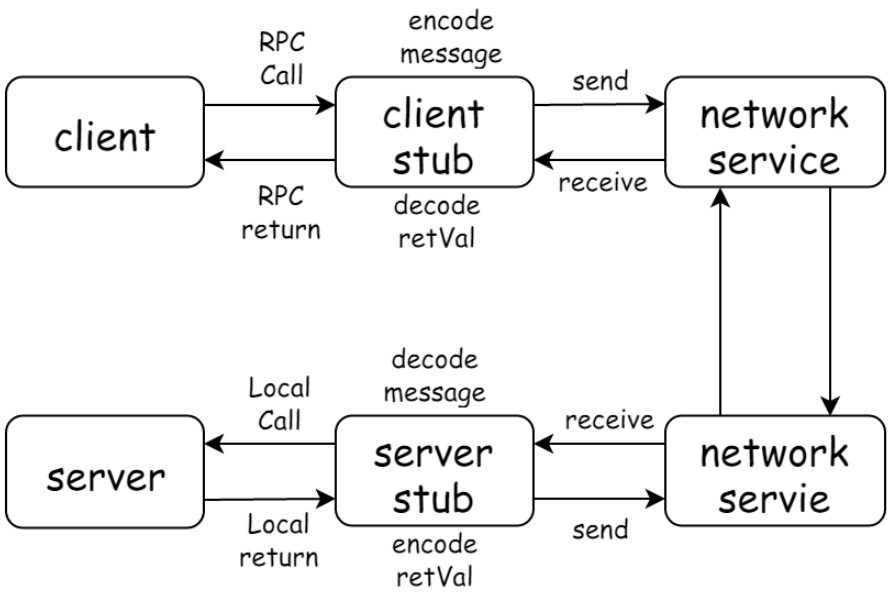
RPC调研

什么是 RPC

RPC即Remote Procedure Call，是指对远程Server 提供的功能进行调用的过程。常见的 RPC 通信过程如下图所示：

RPC 框架通常要满足以下条件：



1. 可用性，远程调用的难度与本地调用尽可能接近，尽可能做到用户透明，即用户无需明显区分本地调用和远程调用；
2. 高性能，远程调用因为网络通信的原因，调用延迟一定高于本地调用，而一个优秀的框架会尽可能降低调用延迟，降低性能开销；

根据 RPC通信的过程，可将 RPC 调用栈分为以下几层：

1. 网络层：负责二进制数据的传输；
2. 传输层：将二进制数据流封装成 frame 或者 message 供上层使用；
3. 逻辑层：对 frame 序列的定义，比如 request<>reply模式，pub<>sub模式，routing 模式等；
4. API 层（数据）：负责将用户数据转换成 frame，涉及对象数据的序列化，序列化数据的压缩等问题；
5. API 层（调用）：负责向用户提供调用API，设计同步异步机制以及多语言支持；

常见 RPC 框架

以下是常见的 RPC 框架及其在相关调用栈上的设计：

框架	网络层	传输层	逻辑层	API层-数据	API层-调用
erpc	tcp	私有	req<>rep	protobuf	protobuf service
grpc	tcp	http2	req<>rep	protobuf	protobuf service
zeromq	tcp udp ipc	zmtp	req<>rep pub<>sub route	可选，比如 pb（ps-lite） msgpack(zero-rpc)	-
brpc	tcp	私有 http	req<>rep	pb	protobuf service
hsf	tcp	私有	req<>rep	hession	-
thrift	tcp	私有	req<>rep	thrift	thrift
akka	tcp	私有	req<>rep	java序列化	actor模型

	udp	zmtp +扩展	pub<>sub	kyro protobuf	

关键技术

RPC 框架要为整个系统的性能负责，因此有一些关键特性要为性能负责：

1. 多路复用，基于 epoll、kqueue 这些 API 实现的 reactor 模式通信是现代化 RPC 的标准底层方案；
2. 长连接，TCP 协议受用了 Slow Start <https://en.wikipedia.org/wiki/TCP_congestion_control#Slow_start> 策略来进行拥塞控制，以避免发送速度快于网络带宽导致网络拥塞。这意味着 TCP 对于每个链接都从非常小的传输窗口开始传输，并在传输过程中缓慢增长传输窗口。这种 slow start 策略导致短链接面临比较严重的性能问题。因此主流 RPC 框架都会支持长连接的工作方式，不但避免建立 TCP 链接的开销，也会避免 slow start 的问题；
3. 零拷贝；
4. 服务发现、负载均衡与容错处理；

技术调研

传输层设计

传输层主要负责 Server 与 Client 端之间的通信，通信的主要内容是 RPC 调用的参数一级结果。常见的传输层设计一般会在 TCP 的二进制字节流上抽象出帧（frame）与消息（message）的逻辑概念供上层使用。以下是几种常见的传输层设计：

TCP 私有协议

TCP 私有协议一般实现比较简单，但兼容性差，认可度也比较低。私有协议常见两种设计：

1. TLV 帧设计，即每帧消息包含 Type、Length 和 Value 三个段，并以 Length 决定帧长度，若需要提高消息的可靠性，可以在帧尾追加一个 crc32 段做数据校验；
2. 变长帧设计，即每一帧长度不定，需要理解消息内容才能知道每一帧结尾在何处，优点是降低了 Type 和 Length 的开销，缺点是较难实现，而且和上层业务耦合较紧。但如果结合具体业务进行优化，变长帧可以达到非常不错的吞吐速度；

HTTP 协议

最早时候，http 协议只支持短链接。http 请求头较大，并且需要文本解析，所以用在 RPC 场景效率较低。但随着 http 长连接，以及复杂传输编码技术的出现，这种情况很大程度上得到了解决。另外http协议在生态上比较完整，比如接入网关、鉴权认证、代理转发等，都非常成熟，因此基于 http 长连接技术设计 RPC消息层也是可行的。

http 1.1 协议中有两个header字段用于传输过程中数据编码：

1. Content-Encoding <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Encoding> ： 控制整个entity-body传输过程中的编码压缩；
2. Transfer-Encoding <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Transfer-Encoding> ： 控制每个 entity 的编码，其中 chunked encoding 对与我们实现 RPC 有较大帮助；

chunked encoding 是指将大块数据切分成小片，并逐片发送，发送格式为 `<length>\r\n<content>\r\n`，type 通过 header 中的 `Content-Type` 来实现。以下是一个示例：

```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked

7\r\n
Mozilla\r\n
9\r\n
Developer\r\n
7\r\n
Network\r\n
0\r\n
\r\n
```

可见 chunked encoding 的 LV 数据格式与 TLV 的设计十分相似。但基于文本的 结束符 `\r\n` 会有些额外开销。不过redis 协议也使用 `\r\n` 结束符，并可达到很高的 QPS 和吞吐，所以这点不会成为瓶颈。

ZMTP 协议

zmtp <http://zmtp.org/page:read-the-docs> 是 ZeroMQ 所使用的 Message Transort Prorocol，其定义如下

```
;   A message is one or more frames
message = *message-more message-last
message-more = ( %x01 short-size | %x03 long-size ) message-body
message-last = ( %x00 short-size | %x02 long-size ) message-body
message-body = *OCTET
```

ZMTP使用一种自定义的 TLV 格式，共4种 type，用于区分是否多帧消息，是否长消息。ZMTP对于短消息做了额外优化，使用更少的字节来编码消息长度。此外，ZMTP 还定义了一组信令用于版本校验、认证和通信控制。基于这些信令，ZMTP 上可以比较方便的实现一系列通信模式。

二进制帧格式+信令的设计使得 ZMTP 更像是 TCP 协议上的另一层网络协议，而其作者也一直致力于将这种基于消息的通信协议和 API 标准化成 Linux 内核的一部分。

HTTP/2协议

http/2 协议是在 Google 的 SPDY 基础上发展起来的新一代 HTTP 协议，主要从多路复用、头压缩等角度对 http/1.1 进行了改进。http/2 的帧格式如下：

```
+-----------------------------------------------+
 |                 Length (24)                   |
 +---------------+---------------+---------------+
 |   Type (8)    |   Flags (8)   |
 +-+-------------+---------------+-------------------------------+
 |R|                 Stream Identifier (31)                      |
 +=+=============================================================+
 |                   Frame Payload (0...)                      ...
 +---------------------------------------------------------------+
```

每一帧有一个固定9字节72比特的头，与常规 TLV 格式相比，头部可能会大3，4个字节。

gRPC 协议

gRPC 以 HTTP/2 的帧为载体，在之基础上定义了如下帧格式：

```
Request → Request-Headers *Delimited-Message EOS
Response → (Response-Headers *Delimited-Message Trailers) / Trailers-Only

Delimited-Message → Compressed-Flag Message-Length Message
Compressed-Flag → 0 / 1 # encoded as 1 byte unsigned integer
Message-Length → {length of Message} # encoded as 4 byte unsigned integer
Message → *{binary octet}

Response → (Response-Headers *Delimited-Message Trailers) / Trailers-Only
Response-Headers → HTTP-Status [Message-Encoding] [Message-Accept-Encoding] Content-Type *Custom-Metadata
Trailers-Only → HTTP-Status Content-Type Trailers
Trailers → Status [Status-Message] *Custom-Metadata
HTTP-Status → “:status 200”
Status → “grpc-status”
Status-Message → “grpc-message”
```

RESP 协议

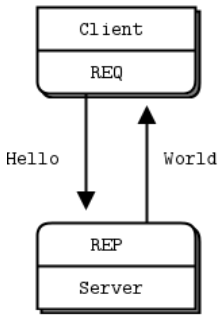
RESP <https://redis.io/topics/protocol> 协议是 redis 所使用的序列化协议，由于和 redis 的场景绑定，所以平时使用不多。RESP 是一种变长协议，即无固定帧结束符，无显式帧长度，因此解析较以上几种协议更复杂。但由于 redis 的广泛使用，使得 redis 客户端非常方便，使用RESP 协议的好处就是可以复用高性能 redis 客户端，减少 RPC 框架的开发成本。

逻辑层设计

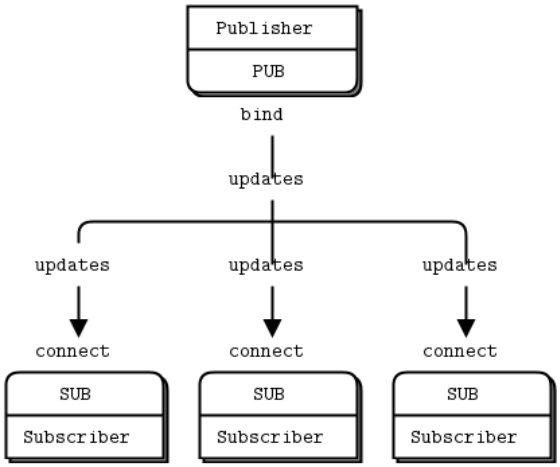
逻辑层主要负责管理底层通信模式，关于通信模式，在 ZGuide <http://zguide.zeromq.org/> 中已经有比较好的叙述了，这里只抽出比较基础的几个模式介绍。

通信模式

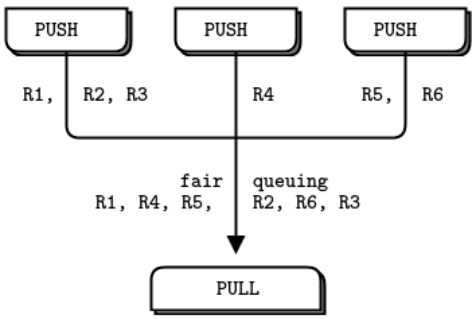
Request-Reply 模式



Publish-Subsccribe 模式



Push-Pull 模式



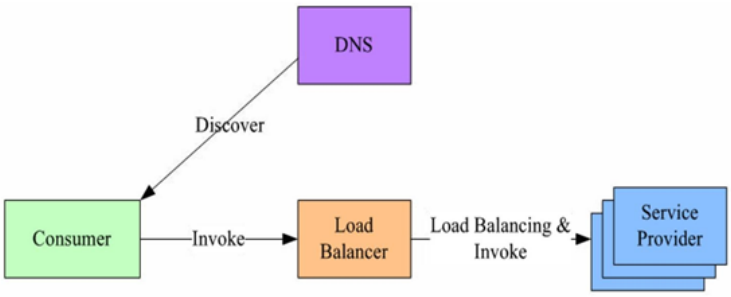
一些说明

互联网公司使用的现代化 RPC 一般是从 protobuf 的 service 接口以及 Java 的 RMI 演进而来的，一般更侧重于函数语义，即更看重request-reply模式。但随着线上系统的复杂化，pub-sub 模式以及 push-pull 模式的应用场景也越来越多，特别是涉及一些离线数据系统时。

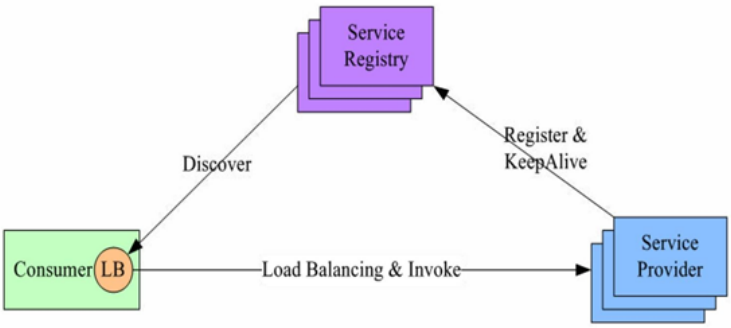
服务发现与负载均衡

对于高可用高性能服务，服务发现、负载均衡与容错处理是不可缺少的组成部分，常见的实现结构有三种 <https://segmentfault.com/a/1190000008672912>：

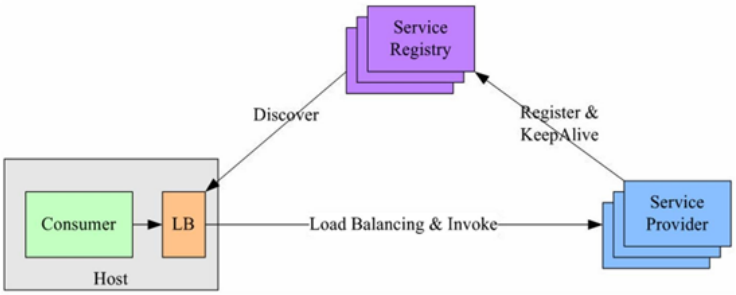
Proxy 端 load balance



Client 端 load balance



Client 端独立 load balance 服务



API 层设计-数据

序列化方法

在 RPC 过程中，最核心的一点，是通信双方相互理解，Client 需要发送 Server 能够理解的数据，Server 也要给出 Client 能够理解的响应。为了简化这个过程，RPC 框架一般将Client 发送的请求数据和 Server 给出的响应数据都理解成 **对象** 或者 **消息**。序列化方法就是将编程语言中的对象，转化成网络通信能够理解的二进制字节流，进行传输的方法。

我们一般从如下几个方面考虑序列化方法：

1. 可靠，经过序列化和反序列化后，数据不变。这里的不变是广义的，比如一个对象在C里序列化，在 Java 中反序列化，那么 java 对象的二进制内存数据可能与C对象不同，但两个对象在含义上是相同的，比如每个属性值都相等；
2. 效率，在序列化的过程中，不光会编码对象本身的数据，还会编码一些分割符、指示符和结束符到二进制字节流中。我们定义对象相关字节数与对象无关字节数之比为序列化方法的效率。过多的额外数据会导致二进制数据膨胀，降低编码效率，进而会产生内存浪费、CPU 耗时以及浪费网络带宽的问题。一个好的序列化方法能够尽可能少的产生额外数据。
3. 速度，在 RPC 过程中，需要频繁进行数据的序列化和反序列化，因此这两个过程的计算开销也是影响 RPC 程序速度的主要因素；

由于序列化和反序列化过程中必须要理解对象，我们也有可能有需求去理解序列化后的数据，下表从对象数据的描述方法和序列化数据是否能够自省来对比几种序列化方法。

序列化方法	描述对象	序列化数据自省
protobuf	IDL 语言	自省 API
thrift	IDL 语言	-
hession	-	-
json	过程式定义序列化过程	通过字段名
msgpack	过程式定义序列化过程	通过字典结构

压缩方法

压缩是 RPC 框架中一种时间换空间的策略，通过压缩降低收发数据的体积，从而让网卡在单位时间内收发更多数据。虽然网络带宽越来越不成为问题，但序列化后的数据大小对整个 RPC 的性能还是有着关键性的性能影响。压缩能够让 RPC 框架在单位时间内交换更多的数据给网卡，让网卡收发更多的请求。

压缩算法我们从效率和速度个方面来评估：

1. 效率，主要看压缩比，压缩比越高，则 RPC 框架吞吐能力越强；
2. 速度，主要看压缩时间，和解压缩时间；

一般来说，高压缩比的算法会开销更多的 CPU，压缩算法的选择即是一种 CPU 时间和传输时间的 trade off。适当选择压缩方案能够较好的降低整体处理时间，提高 QPS 能力。

常见几种压缩算法的压缩比一般排序为：
lzma > bzip2 > gzip = zip > lzo = lz4 =snappy
而计算速度的一般排序为：
snappy=lz4=lzo>zip=gzip > bzip2 > lzma

以下是在ubuntu 官方 iso 镜像数据上的压缩效率与时间对比：

压缩方法	原文件大小	压缩大小	压缩比	压缩时间	解压时间
zlib	1501102080	1481348881		51s	11s
gzip	1501102080	1481538797		44.4s	3.1s
bzip2	1501102080	1492225272		3m55s	2m3s
lzma	1501102080	1479042010		10m44s	1m38s
snappy	1501102080	1492138718		9.5s	6.4s
lz4	1501102080	1491800719		3.3s	3.3s
lzo	1501102080	1492659179		3.9s	1.9s
zstd	1501102080	1485924528		8.4s	2.0s

API 层设计-调用

目前常见的 RPC 框架通常以 protobuf 定义message 和 service，比如以下 pb 定义：

```
option cc_generic_services = true;

message EchoRequest {
      required string message = 1;
};
message EchoResponse {
      required string message = 1;
};

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

通过 protoc 编译后，会给出 service 的定义，比如：

```
#include "echo.pb.h"
...
class MyEchoService : public EchoService  {
public:
    void Echo(::google::protobuf::RpcController* cntl_base,
              const ::example::EchoRequest* request,
              ::example::EchoResponse* response,
              ::google::protobuf::Closure* done) {
        // 这个对象确保在return时自动调用done->Run()
        brpc::ClosureGuard done_guard(done);

        brpc::Controller* cntl = static_cast<brpc::Controller*>(cntl_base);

        // 填写response
        response->set_message(request->message());
    }
};
```

其中 `EchoService` 是 protoc 生成的 service 定义，在实现 service 时候，需要继承该类，并提供 Echo 方法的实现。Echo 方法由四个参数，其中 `cntl_base` 提供RPC 的一些额外控制参数， `request` 与 `response` 是RPC 的请求和相应， `done` 是框架提供的回调接口， `done->Run()` 将会执行 `response` 的校验、序列化和打包发送。实现异步 service 时，只需要将 `done` 保存下来，在完成请求后再调用 `done->Run()` 即可。

ZeroCopy