1V1语音服务器设计

# 简介

# 框架设计

语音服务器架构设计，主要分为媒体前端，媒体集群，媒体中心和统计业务4块：

* 媒体前端(mproxy)负责提供用户接入连接和数据包转发服务，具体来说，负责语音和视频通话的数据链路。
* 媒体集群(mgroup)负责机房级别的所有媒体前端管理，包含具体前端分配和流管理。
* 媒体中心(mcenter)负责媒体集群的管理，包含机房分配，机房监控。
* 媒体统计(mstat)负责信息的收集，包含并发量，每条链路的rtt, 语音延迟，丢包率，接通速度和通话时长等。

媒体前端对服务器接入速度要求比较高，属于高并发业务，一般来说，需要在每个isp(电信，网通，移动等) 的机房部署服务，每台服务器（4核16G, 2G网卡）大概能支持10000个并发通话。

统计进程对实时性要求低，服务器部署非常自由。

开始呼叫

MCenter：分配机房

ＭGroup：分配前端

开始通话

根据isp/area分配机房

根据负载分配前端

双方通过MProxy转发数据

mproxy使用UDP协议做所有数据处理，其它进程使用TCP做了标准协议。mstat数据可靠性要求不高，可以考虑使用UDP。

服务器之间的链接，使用白名单机制保证安全性, (iptable层来做，不依赖代码控制)。mproxy和客户端之间，暂时用cooki机制来确保安全。考虑到企业用户的特性，需要考虑使用链路加密(AES)。

服务器设计的主要原则是：

1. 去中心化，防止单点失效，防止中心节点负载瓶颈。如mcenter这一类数据中心节点，需要确保所有数据在mcenter挂了以后，能快速从mgroup重建。
2. 解耦合，服务之间减少交互，每个服务对外暴露简单接口，部署上无顺序依赖。必要的时候，将数据同步出来，通过http提供对外接口，降低长链接带来的耦合。
3. 服务化，每个服务之间保持协议交互接口问题，简单易用。
4. 考虑省电和省流量的需求，减少使用push模型，尽量使用pull模型。

# 协议定义

协议定义求简化，分为控制协议和数据协议2套。控制协议用来进行通话指令的传递，而数据协议用来传送媒体数据报。

总体上，协议分为包头和数据去，包头对于所有协议都是一致的，8个字节：svid, uri, seq, res.

|  |  |  |
| --- | --- | --- |
| 名称 | Byte数目 | 描述 |
| Len | 4 | 包长（不需要4字节，可用做flag位） |
| svid | 2 | Service id. |
| uri | 2 | Protocol id in service |

需要注意，由于媒体前端采用udp，不需要考虑分包，所以包头不包含length字段。

包头之后是各个协议的内容，各不相同。为了简化处理过程，定义了string, int, short, char的marshall/unmarshall方法。比如对于string类型，marshall方法如下：

pushInt16(len);

pushString(bytes);

以此确保不同编程语言，不同endian平台下，都可以正确解释协议内容。具体每个协议的定义和marshall/unmarshall实现，请参考源代码。

# 媒体前端(mproxy\_d)

媒体前端负责用户的媒体接入功能，从根本上来说是一个实时的消息队列：

* 说话方，publish数据到服务器stream上。
* 接听方，subscribe数据到stream上。

前端服务器的职责就是，根据stream id把publisher和subscriber联系起来。注意媒体前端并不假设pub/sub是1v1的关系，也就是说设计本身就是支持多人通话的。

考虑到链接用户数目比较高，媒体前端开发使用单线程epoll轮询机制，在同一台物理机，启动多个进程，每个进程监听一个端口。由于使用单线程方案，进程内不需要考虑并发。

媒体前端主要需要处理3个消息：

1. Publish Request

Publish请求包含uid/nick/stream信息，用来说明说话人的信息。注意stream的定义必须为唯一的，不然会出现串音现象。

*struct PMPJoinReq : Packet {*

*enum { uri=URI\_MPROXY\_JOIN\_REQ };*

*int uid;*

*std::string nick;*

*std::string stream;*

*virtual void unmarshall(Unpack& up) {*

*uid = up.popInt32();*

*nick = up.popString();*

*stream = up.popString();*

*}*

*virtual void marshall(Pack& pk) {*

*pk.pushInt32(uid);*

*pk.pushString(nick);*

*pk.pushString(stream);*

*}*

*};*

1. Subscribe Request

语音接收方，通过stream与publisher连接起来。

*struct PMPJoinRes : Packet {*

*enum { uri=URI\_MPROXY\_JOIN\_RES };*

*int res;*

*int sid;*

*int uid;*

*virtual void unmarshall(Unpack& up) {*

*res = up.popInt32();*

*sid = up.popInt32();*

*uid = up.popInt32();*

*}*

*virtual void marshall(Pack& pk) {*

*pk.pushInt32(res);*

*pk.pushInt32(sid);*

*pk.pushInt32(uid);*

*}*

*};*

1. Stream Data

Stream data是具体的语音视频的payload信息，seq用来做媒体相关的丢包重传。Stamp表示语音视频数据的录制时间，客户端Jitter Buffer播放依赖于stamp。

媒体前端自动根据同一条链接publish时候的ip/port把这个数据报映射到stream，然后找到stream上所有subscriber，进行数据转发。此过程不需要其它服务器参与。

*struct PMPStreamData : Packet {*

*enum { uri=URI\_MPROXY\_STREAM\_DATA };*

*int from;*

*int seq;*

*int stamp;*

*std::string payload;*

*virtual void unmarshall(Unpack& up) {*

*from = up.popInt32();*

*seq = up.popInt32();*

*stamp = up.popInt32();*

*//this is a stupid hack, trying to hook up a memory pool.*

*//later I should implement the memory pool here.*

*//payload = up.popString();*

*}*

*virtual void marshall(Pack& pk) {*

*pk.pushInt32(from);*

*pk.pushInt32(seq);*

*pk.pushInt32(stamp);*

*pk.pushString(payload);*

*}*

*};*

媒体服务器属于UDP服务，因此需要反攻击能力，目前来说主要根据包格式是否合法来判断客户端，并基于此建立黑名单。黑名单的数据过来直接抛弃，然后通知上层的反外挂平台处理。如果协议本身被破解，UDP服务会面临攻击，目前来说，主要的防御手段是ticket机制，开发工作量比较高，可以考虑在今后加入。

# 媒体集群(mgroup\_d)

mgroup负责管理某个机房的所有前端机器，mgroup统计归总本机房内所有的用户，流信息，负载信息，前端分配管理，然后向mcenter汇报。对于mcenter来说，mgroup代表的是一种集群服务能力。

mgroup收集本机房内所有前端(mproxy)的当前信息，包含load, load\_max等。即使mcenter失效，mgroup也可以提供完整的语音通信能力。

用户需要一条连接到mgroup，来分配具体到哪个mproxy前端做语音数据交换。每个机房的用户链接数可能很高(40server\*1.5=60万)，因此mgroup本身并不能提供用户直接连接。mgroup存在的目标只是作为中心的数据收集节点。具体用户访问的任务交给loadtable进程。

mgroup收到任何mproxy的负载报告，就把这部分数据同步给loadtable进程。Loadtable进程维护一张分配表，根据load/load\_max来进行分配。一般来说，load越低,load\_max越高，表示前端的服务能力越强，在分配算法里就会更优先。

Loadtable实现master/slave 1vN能力，可以无限扩展。另外loadtable对外提供基于tcp和http2套访问接口，以简化客户端设计(不需要长链接)。每个loadtable可以提供大概1500的并发量，一台机器4个进程，应该上就可以完成整个集群的分配任务。 每个load\_table进程的分配表数据是完全一样的，需要防止预分配冲突。

mproxy通过如下协议向mgroup进行汇报：

*struct PMGReportReq : Packet {*

*enum { uri=URI\_MGROUP\_REPORT\_REQ };*

*std::string name;*

*std::string ip;*

*short port;*

*short udp;*

*int load;*

*int load\_max;*

*virtual void unmarshall(Unpack& up) {*

*name = up.popString();*

*ip = up.popString();*

*port = up.popInt16();*

*udp = up.popInt16();*

*load = up.popInt32();*

*load\_max = up.popInt32();*

*}*

*virtual void marshall(Pack& pk) {*

*pk.pushString(name);*

*pk.pushString(ip);*

*pk.pushInt16(port);*

*pk.pushInt16(udp);*

*pk.pushInt32(load);*

*pk.pushInt32(load\_max);*

*}*

*};*

mgroup收到每个mproxy的report之后，其实就知道当前mproxy的负载能力。把这部分数据复制给loadtable，然后由loadtable对外提供服务。客户端需要访问loadtable，来实时查询本mgroup集群哪台机器的负载最低，最适合提供服务。一旦查询完成并进行分配，loadtable会通过mgroup向相应的mproxy发送一条指令，进行资源预留。(细节请参考SIP协议的资源分配方案。)

mproxy向mgroup汇报频率要求高（<=1s），以确保mgroup和loadtable里的数据是实时的。具体汇报的间隔需要实时测试，适当设置load\_max小于实际负载能力，能解决大部分分配冲突。

# 媒体中心(mcenter\_d)

mcenter\_d是媒体业务对外的接口层，主要负责进行用户的机房调度。mcenter\_d接受mgroup\_d的定期汇报各个机房负载状况，并根据负载来挑选机房给用户使用。

mcenter\_d并不主动给mgroup\_d发消息，以保持相对的独立性，防止单点失效。mcenter\_d意外重启之后，将根据mgroup\_d的定期汇报重建机房信息数据。

mcenter\_d负责所有机房分配，而且因为保存的全量服务数据，因此对性能要求高。为了优化访问速度，在mcenter\_d之前架设mcproxy\_d, 提供cache访问。

## mcenter与mcproxy之间的层次如下

接入服务器

mcproxy

集群

mcproxy

mcproxy

集群

mcproxy

集群

DNS(mcproxy.xxx.com)

mcenter(媒体中心)

交互图如下：

mcproxy

mcenter

mgroup

接入服务

PMCReportReq

PMCAllocReq

PMCPreAllocReq

PMCPreAllocRes

PMCAllocRes

每次有用户请求分配机房，请求从接入服务到达mcproxy.

* 如果mcproxy本地cache存在，直接返回给用户。
* 如果mcproxy本地cache不存在，mcproxy向mcenter请求预分配大概300个资源.
* mcenter收到mcproxy的预分配请求，分配资源并返回给mcproxy.
* mcproxy把预分配资源保存，然后根据用户请求返回给接入服务器

遇到突发状况，被mcproxy本地cache下来的机房离线了，则会导致大概300个访问失败。考虑到机房整体下线可能性很低，300次访问重试是可以接受的。

## 机房选择策略：

机房分配策略如下：

用户输入：isp1, area1, isp2, area2

机房信息：isp, area, load/load\_max

输出：isp, area, grp\_ip, grp\_port.

1. 如果isp1==isp2
   1. 如果area1==area2, 优先选择grp.isp==isp1的机房，如果没有找到，则寻找(grp.isp&isp1==isp1)的机房。
   2. 如果area1!=area2, 优先选择grp.area==area1的机房，其次选择grp.area==area2的机房。
2. 如果isp1!=isp2
   1. 首先计算isp\_all=isp1|isp2, 然后按照(isp\_all, area1,area2)根据规则1进行选择。
   2. 如果isp\_all机房不存在，则选择(grp.isp&isp\_all)==isp\_all的机房。
3. 如果仍然没有找到机房，则选择最贵最好的fallback机房，fallback机房最好设定在能满足绝大部分用户连结性的多线机房。

有效性测试，实例：

void fill\_groups(MCMgr\* mgr) {

mgr->getGrpMgr()->add(1, "group11", "127.0.0.1", 2021, 1, 1, 100, 5000);

mgr->getGrpMgr()->add(1, "group12", "127.0.0.1", 2021, 1, 1, 300, 5000);

mgr->getGrpMgr()->add(1, "group13", "127.0.0.1", 2021, 1, 1, 300, 5000);

mgr->getGrpMgr()->add(1, "group14", "127.0.0.1", 2021, 1, 2, 300, 5000);

mgr->getGrpMgr()->add(1, "group15", "127.0.0.1", 2021, 1, 3, 300, 5000);

mgr->getGrpMgr()->add(2, "group21", "127.0.0.1", 2021, 2, 1, 2100, 5000);

mgr->getGrpMgr()->add(2, "group22", "127.0.0.1", 2021, 2, 2, 2100, 5000);

mgr->getGrpMgr()->add(2, "group23", "127.0.0.1", 2021, 2, 3, 2100, 5000);

mgr->getGrpMgr()->add(3, "group41", "127.0.0.1", 2021, 4, 1, 300, 5000);

mgr->getGrpMgr()->add(3, "group42", "127.0.0.1", 2021, 4, 2, 300, 5000);

mgr->getGrpMgr()->add(4, "group81", "127.0.0.1", 2021, 8, 1, 700, 5000);

mgr->getGrpMgr()->add(4, "group82", "127.0.0.1", 2021, 8, 2, 500, 5000);

mgr->getGrpMgr()->add(5, "group161", "127.0.0.1", 2021, 16, 1, 100, 5000);

mgr->getGrpMgr()->add(5, "group162", "127.0.0.1", 2021, 16, 2, 100, 5000);

mgr->getGrpMgr()->add(5, "group163", "127.0.0.1", 2021, 16, 3, 100, 5000);

mgr->getGrpMgr()->add(6, "group31", "127.0.0.1", 2021, 3, 1, 100, 5000);

mgr->getGrpMgr()->add(6, "group32", "127.0.0.1", 2021, 3, 2, 100, 5000);

mgr->getGrpMgr()->add(7, "group7", "127.0.0.1", 2021, 5, 1, 100, 5000);

mgr->getGrpMgr()->add(8, "group8", "127.0.0.1", 2021, 6, 1, 100, 5000);

}

void select\_group(MCMgr\* mgr) {

GrpInfo\* info = NULL;

info = mgr->getGrpMgr()->fetchGroup(1, 1, 1, 1);

info = mgr->getGrpMgr()->fetchGroup(1, 1, 1, 2);

info = mgr->getGrpMgr()->fetchGroup(1, 1, 2, 1);

info = mgr->getGrpMgr()->fetchGroup(1, 1, 2, 2);

info = mgr->getGrpMgr()->fetchGroup(1, 2, 2, 2);

info = mgr->getGrpMgr()->fetchGroup(1, 2, 2, 3);

info = mgr->getGrpMgr()->fetchGroup(1, 1, 8, 1);

info = mgr->getGrpMgr()->fetchGroup(1, 1, 8, 2);

}

最终数据，还需要添加fallback的多线机房。

## mcenter\_d主要处理2条协议：

* PMCReportReq
* PMCPreAllocReq

Mcenter\_d应该同时对外提供web监控接口，以实施检测整个媒体服务的运作状况。比较独立的做法是，mcenter\_d把机房数据同时写入到数据库，然后web通过php从数据库取数据显示。

# 媒体中心前端(mcproxy\_d)

所有节点分配都需要经过mcenter\_d, 会导致mcenter\_d压力过大，并存在单点故障的可能，因此在mcproxy\_d和接入进程之间，需要架设mcproxy\_d。

mcproxy\_d本身是一个cache，简单的cache mcenter分配好好的机房信息，本身不做任何策略选择。把分配请求从mcenter分离出来，能极大提高mcenter的负载能力。更进一步，如果mcenter+mcproxy仍然不能满足负载需求，可以形成mcenter+reds+mcproxy的模式，进一步去中心化，提高负载能力。

每次用户请求道mcproxy, mcproxy会根据isp/area信息，查找是否有预分配的机房，如果有直接返回给用户；如果没有则发送PMCPreAllocReq给mcenter, 并把用户请求缓存。等到mcenter返回PMCPreAllocRes，根据返回的信息处理缓存的请求。

mcproxy理论处理能力为：4万

mcproxy与mcenter的部署比例上限大致为：1:100

按照此结构，并发负载上线大概是：400万

超过1:100部署量，会因为预分配冲突，会开始出现机房资源分配不均衡，需要更加精细的分配方案。

mcproxy\_d的分区表cache, 3秒过期。

# 媒体监控(mstat\_d)

# 节点发现(deamon\_d)