

FreeSWITCH WIRESHARK

杜金房 著

INVITE sip:+861234567890@example.com SIP/2.0
Via: SIP/2.0/UDP 10.20.30.40:5060;rport;branch=z9hG4bKj9Nm7v3047Ha
Max-Forwards: 70 FreeSWITCH-CN 出品
From: "Seven Du" <sip:+861234567890@example.com>;tag=B0U9ZQZQ124SK
To: <sip:+861234567890@example.com>
Call-ID: eb9f724e-9fed-1233-88ac-525400272cd0
CSeq: 77777777 INVITE

FreeSWITCH WIRESHARK

杜金房

版权所有，侵权必究

图书不在版编目 (NCIP) 数据

FreeSWITCH WIRESHARK / 杜金房 著 / 2015.10

ISBN 7-DU-777777-7

本书包含用 Wireshark 解决 VoIP 问题的一些实例, ...

FreeSWITCH WIRESHARK

作 者 杜金房

封面设计 杜金房

校 对 杜金房

排 版 杜金房

开 本 1890 毫米 × 2360 毫米

印 张 7.5

印 数 7

版 数 2015 年 10 月第 1 版 2016 年 8 月第 1 次发布

电子邮箱 freeswitch@dujinfang.com

前 言

一直想写一本关于 Wireshark 的书，但是一直没有完成。2015 年国庆节的时候，突击了一把，但还是没有完成。

看看马上又到另一个国庆节了，时间过得真快。

与其烂在手里，倒不如拿出来献丑。

本书没有废话，都是些真实的例子。

其实本书核心内容是用 Wireshark 分析 VoIP 包，跟 FreeSWITCH 本身关系不是很大，但—

- 如果你喜欢 FreeSWITCH，你肯定喜欢本书。
- 如果你喜欢 Wireshark，本书或许也对你有帮助。

本书内容可能会有更新，请关注本书的网站：<http://book.dujinfang.com>。你也可以订阅 FreeSWITCH-CN 微信公众号以随时了解 FreeSWITCH 和本书动态。



图 1: FreeSWITCH-CN 微信公众号

目 录

前 言	1
1 从电话开始说起	4
1.1 SIP 协议	5
1.2 RTP 协议	8
1.3 SDP 协议	9
1.4 小结	11
2 用 Wireshark 分析 SIP 呼叫	13
2.1 分析 SIP 包	13
2.2 分析 RTP 包	15
2.3 理解 UDP 重发	16
2.3.1 预防性重发	16
2.3.2 按需重发	16
3 看不到视频	18
4 解析 H264 数据包	21
5 异地多活	25
6 音视频不同步问题	28
7 NAT	31
7.1 为什么呼不通?	31

7.2 为什么我活不过 30 秒?	32
7.3 通过实例深入理解 NAT	35
7.3.1 默认配置	36
7.3.2 使用 Stun 服务器	37
7.3.3 使用 ICE	40
7.4 更多实例	41
7.5 UPnP	45
8 不是你的错	49
9 暗无天日的日子	52
9.1 多人电话会议	52
9.2 谎言终会被揭穿	55
9.3 迟到的 RTP	57
9.4 疯狂的按键音	61
9.5 诡异的 ACK	62
9.6 小结	64
写在最后	65
作者简介	66
版权声明	67
广告	68
关于广告的广告	68
FreeSWITCH 第五届开发者沙龙将于 2016 年 8 月 27 日在京举行	68
FreeSWITCH 培训 2016 夏季班（北京站）将于 2016 年 8 月 28-30 日在京举行	68
烟台小樱桃网络科技有限公司提供商业 FreeSWITCH 及 OpenSIPS 技术支持	68
烟台小樱桃网络科技有限公司是潮流网络（GrandStream）山东总代理	68
FreeSWITCH 相关图书	69
	71

第一章 从电话开始说起

众所周知，人类最原始的通信方式当然是“吼”啦。但吼传输距离实在太有限。最早的有代表性的远程通信方式普遍认为是“烽火台”。

1876 年，苏格兰人亚历山大 · 贝尔（Alexander Granham Bell）发明了电话，大大增加了通信的距离。随着电子计算机的出现和数字化技术的成熟，20 年代 60 年代以来，脉冲编码调制（PCM）技术成功地应用于传输系统中，数字电路逐渐取代原来的模拟通信线路，提高了通话质量、增加了传输距离，同时，节约了许多线路成本。随着分组交换技术的成熟及因特网的发展，人们认识到了将原始的基于电路交换的语音网络与基于分组交换的因特网络进行融合（即语音通信和数据通信相结合）的必要性，因此一个称为 NGN（Next Generation Network）的概念被提了出来。NGN 其实也就是 VoIP。

VoIP 的全称 Voice Over IP，即承载于 IP 网上的语音通信。大家还得家庭用来上网的 ADSL 吧？或许有些人还记得前些年用过的吱吱叫的老“猫”（Modem，调制解调器）？技术日新月异，前面的技术都是用电话线上网，现在，VoIP 技术使我们可以在网上打电话，生活大概就是这样。

比较有代表性的 VoIP 协议有 H.323 和 SIP。

H.323 是 ITU 多媒体通信系列标准 H.32x 的一部份，该系列标准使得在现有通信网络上进行音视频会议成为可能，其中，H.320 是在 N-ISDN 上进行多媒体通信的标准：H.321 是在 B-ISDN 上进行多媒体通信的标准：H.322 是在有服务质量保证的 LAN 上进行多媒体通信的标准：H.324 是在 GSTN 和无线网络上进行多媒体通信的标准。H.323 为现有的分组网络 PBN（如 IP 网络）提供多媒体通信标准。若和其他的 IP 技术如 IETF 的资源预留协议 RSVP 相结合，就可以实现 IP 网络的多媒体通信。

SIP（Session Initiation Protocol，会话发起协议）是由 IETF（Internet 工程任务组）提出的 IP 电话信令协议。正如其名字所隐含的，SIP 用于发起会话，它能控制多个参与者参加的多媒體会话的建立和终结，并能动态调整和修改会话属性，如会话带宽要求、传输的媒体类型（语音、视频和数据等）、媒体的编解码格式、对组播和单播的支持等。

H.323 和 SIP 设计之初都是作为多媒体通信的应用层控制（信令）协议，目前一般用于 IP 电话。

它们能实现的信令功能基本相同，也都利用 RTP 作为媒体传输的协议。但两者的设计风格截然不同，这是由于其推出的两大阵营（电信领域与 Internet 领域）都想沿袭自己的传统。H.323 是由国际电信联盟提出来的，它企图把 IP 电话当作是众所周知的传统电话，只是传输方式由电路交换变成

了分组交换，就如同模拟传输变成数字传输、同轴电缆传输变成了光纤传输。而 SIP 侧重于将 IP 电话作为 Internet 上的一个应用，较其他应用（如 FTP，E-mail 等）增加了信令和 QoS 的要求。H.323 推出较早，协议发展得比较成熟；由于其采用的是传统的实现电话信令的模式，便于与现有的电话网互通，但相对复杂得多。SIP 借鉴了其他 Internet 标准和协议的设计思想，有其突出的优点，在互联网大发展的时代已逐渐代替 H.323 成为事实上的标准。

首先，SIP 是基于文本的协议，而 H.323 采用基于 ASN.1 和压缩编码规则的二进制方法表示其消息，因此，SIP 对以文本形式表示的消息的词法和语法分析就比较简单。其次，SIP 会话请求过程和媒体协商过程等是一起进行的，因此呼叫建立时间短，而在 H.323 中呼叫建立过程和进行媒体参数等协商的信令控制过程是分开进行的。再次，H.323 为实现补充业务定义了专门的协议，如 H.450.1、H.450.2 和 H.450.3 等，而 SIP 只要充分利用已定义的头域，必要时对头域进行简单扩展就能很方便地支持补充业务或智能业务。最后，H.323 进行集中、层次式控制。尽管集中控制便于管理（如便于计费和带宽管理等），但是当用于控制大型会议电话时，H.323 中执行会议控制功能的多点控制单元很可能成为瓶颈。而 SIP 类似于其他的 Internet 协议，设计上就为分布式的呼叫模型服务的，具有分布式的组播功能。

除了电信运营商使用的“正式”的信令协议外，像 Skype、QQ、微信等也都有自己私有的信令协议支持包括文字、语音和视频等在内的多媒体通话。后面这些业务在电信网络中称为 OTT（Over The Top）。近年来，OTT 发展迅猛，被认为是对电信运营商很大的威胁。

SIP 和 RTP 是目前电信领域最为常用的协议。我们本章大多数的例子都是基于这两种协议来讨论的。为了方便不了解这两种协议的读者，我们先做一个简要的介绍。

1.1 SIP 协议

SIP 的全称是 Session Initiation Protocol，即会话初始协议。它是一个控制发起、修改和终结交互式多媒体会话的信令协议。由 IETF (Internet Engineering Task Force, Internet 工程任务组) 在 RFC 2543 中定义的。最早发布于 1999 年 3 月，后来在 2002 年 6 月又发布了一个新的标准 RFC 3261。

SIP 协议与大家熟知的 HTTP 协议有很多相似之处，因此，我们在这里与 HTTP 对比介绍一下。

SIP 是一个基于文本的协议，我们先来对比一组简单的 HTTP 请求与 SIP 请求：

- HTTP: GET /index.html HTTP/1.1
- SIP: INVITE sip:seven@freeswitch.org.cn SIP/2.0

两者类似，请求都是由客户端（称为 UA，即 User Agent）发起的，请求文本的起始行均有三部分组成。在 HTTP 请求中，“GET”指明一个获取资源（文件）的动作，“/index.html”则是资源的地址，最后，“HTTP/1.1”是协议版本号。而在 SIP 中，“INVITE”表示发起一次呼叫请求，“seven@freeswitch.org.cn”为请求的地址，也称为 SIP URI 或 AOR (Adress of Record, 用户的公开地址)，第 3 部分“SIP/2.0”也是版本号。其中，SIP URI 很类似一个电子邮件地

址，其格式为“协议:名称 @ 主机”。“协议”与 http 和 https 相对应，有 sip 和 sips（后者是加密的，如sips:seven@freeswitch.org.cn）；“名称”可以是一串数字的电话号码，也可以是字母表示的名称；而“主机”可以是一个域名，也可以是一个 IP 地址。

两者的响应消息也类似：

- HTTP: HTTP/1.1 200 OK
- SIP: SIP/2.0 200 OK

响应消息也分为三个部分：其中第一部分是协议版本号；200 表示状态码；“OK”是一个可读的描述字符串（称为 Reason Phrase）。

与 HTTP 类似，SIP 的状态码也是由三位数字构成：

- 1xx 表示临时响应；
- 2xx 表示成功的响应；
- 3xx 表示重定向；
- 4xx 表示客户端引起的错误（如请求一个不存在的地址时就会收到著名的 404）或需要客户端认证（如 401，SIP 也使用与 HTTP 一样的 Digest 认证）；
- 5xx 表示服务器端的错误（服务器脚本出错等）。
- 6xx 全局错误

SIP 是一个对等的协议，不像 HTTP 一样是客户端服务器结构。在 HTTP 中，客户端能请求服务器资源，但服务器不能反过来请求客户端。而 SIP 双方都可以互发请求，即每一个 UA 都可以互做服务器和客户端（主动发起请求的一方称为 UAC，即客户端；被动接收请求并做出响应的一方称为 UAS，即服务端）。如图1.1，Bob 给 Alice 发送一个 INVITE 请求，说“Hi, 一起吃饭吧…”，Alice 说“好哇，OK”，电话就接通了。

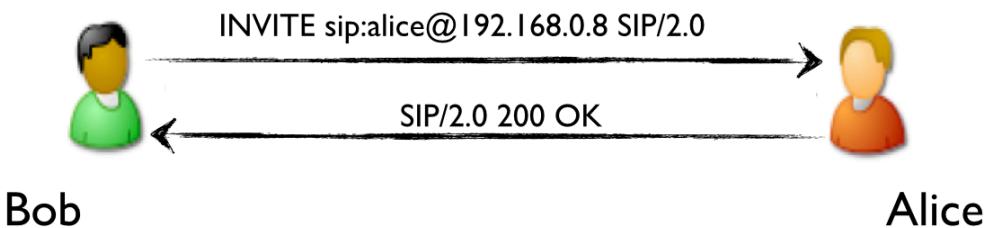


图 1.1: SIP 点对点通信

当然，在现实世界中，每对相互通信的 UA 间互相知道对方的地址是不可能的，另外也由于运营商管控和计费的需要，通信的双方一般都通过一个或几个服务器（称为交换机，注意与普通的以太网交换机不同）。服务器一般由两种形式，Proxy 和 B2BUA。如图1.2。

虽然从上图中看不出两者的区别，但它们还是有区别的。其中 Proxy 称为代理服务器，它会将 Bob 发送的 SIP 消息“转发”给 Alice；而 B2BUA 称为背靠背的用户代理，它相当于一对 UA 的串联，

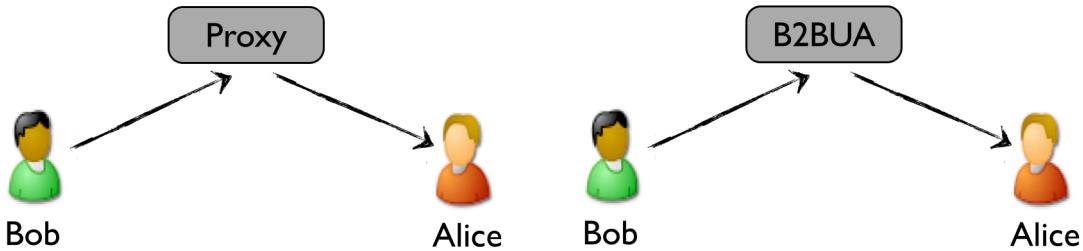


图 1.2: SIP 通信 Proxy 或 B2BUA 通信

如果 Bob 通过 B2BUA 给 Alice 发请求，B2BUA 会响应该请求并重新产生一个新的 SIP 消息发给 Alice，相当于 B2BUA 同时建立了两路通话（称为 Session 或 Channel），只不过 B2BUA 会在自己内部将两路通话“桥接”起来，以便 Bob 和 Alice 能相互通话。

SIP UA 间的消息流程如图1.3所示：

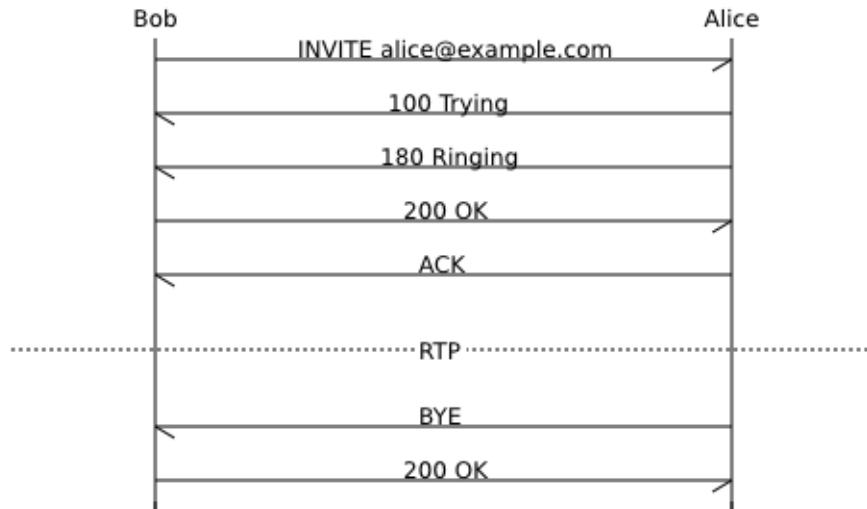


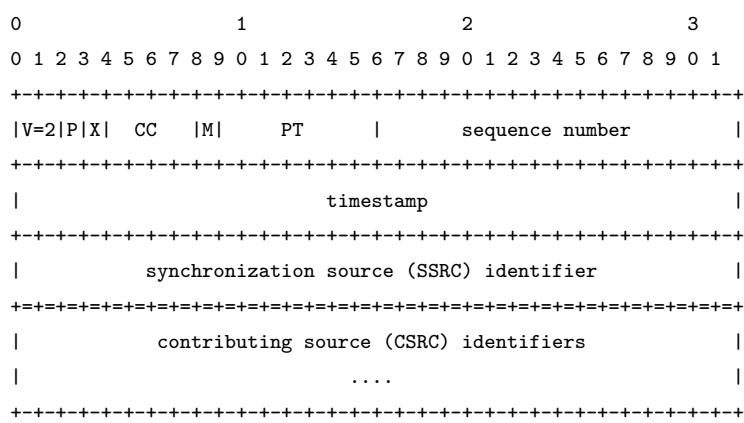
图 1.3: SIP UA 间直接呼叫流程图

首先 Bob 向 Alice 发送 INVITE 消息请求建立 SIP 会话。Alice 的 UA 回 100 Trying 消息，意思是说我收到你的请求了，先等一会儿。接着 Alice 的电话开始振铃，并给对方回消息 180 Ringing，说我这边已经振铃了，Alice 听到后可能一会就过来接电话。Bob 的 UA 收到该消息后即可以播放回铃音，以提示 Bob 对方的话机正在振铃。接着 Alice 接了电话，她发送 200 OK 消息给 Bob，该消息是对 INVITE 消息的最终响应（所有首位大于 1 的状态码都是最终响应），而先前的 100 和 180 消息都是临时状态，只是表明呼叫进展的情况。Bob 收到 200 后向 Alice 回 ACK 证实消息。INVITE - 200 - ACK 完成“三次握手”（与 TCP 的三次握手类似）的操作，保证了呼叫可以正常进行。这时候 Bob 已经在跟 Alice 通话了，他们通话的内容（语音数据）是在 SIP 之外的 RTP 包中传递的。

1.2 RTP 协议

RTP 协议的全称是 Real-time Transport Protocol，即实时传输协议，它是由 IETF 的多媒体传输工作小组在 RFC 3550 中定义的。RTP 协议详细说明了在互联网上传递音频和视频的标准数据包格式。它一开始被设计为一个组播协议，但后来被用在很多单播应用中。RTP 协议是建立在 UDP 协议之上的，常用于流媒体系统、视频会议和一键通（Push to Talk）系统（配合 H.323 或 SIP），它已成为 IP 电话产业的技术基础。

RFC 3550 中规定的 RTP 封包格式如下：



一般来说，在没有任何扩展的情况下，RTP 包头的长度为 12 字节。其中主要字段的含义简介如下：

- version (V): 1 bit。标明 RTP 版本号。RFC3550 中规定的版本号为 2。
 - padding (P): 1 bit。如果该位被设置，则在该 packet 末尾包含了额外的附加信息，附加信息的最后一个字节表示额外附加信息的长度（包含该字节本身）。该字段之所以存在是因为一些加密机制需要固定长度的数据块，或者为了在一个底层协议数据单元中传输多个 RTP packets。
 - extension (X): 1 bit。如果该位为 1，则在固定的头部后存在一个扩展头部，格式定义在 RFC3550 5.3.1 节。
 - CSRC count (CC): 4 bits。在固定头部后存在多少个 CSRC 标记。
 - marker (M): 1 bit。该位的功能依赖于 RTP 中实际传输媒体类型的 profile 的定义。profile 可以改变该位的长度，但是要保持 marker 和 payload type 总长度不变（一共是 8 bit）。一般来说，音频跟视频中该位的定义就是不同的。在视频中，marker 值为 1 通常标明一帧的结束。
 - payload type (PT): 7 bits。标记着 RTP packet 所携带信息的类型，标准类型列出在 RFC3551 中。如果接收方不能识别该类型，必须忽略该包。
 - sequence number: 16 bits。序列号，每个 RTP 包发送后该序列号加 1，接收方可以根据该序列号检测丢包，或重新排列数据包顺序等。

- timestamp: 32 bits。时间戳。反映 RTP 包中所携带信息包中第一个字节的采样时间。
- SSRC: 32 bits。数据源标识。在一个 RTP Session 期间每个数据流都应该有一个不同的 SSRC。如果视频源发生变化，则应该改变 SSRC，以让接收端感知到该变化。
- CSRC list: 0 to 15 项，每个源标识 32 bits。贡献数据源标识。只有存在多路混发的时候才有效。多路混发的情况如一个将多声道的语音流合并到一个 RTP 流中发送，在这里就可以列出原来每个声道的 SSRC。

值得一提的是，RTP 还有一个姊妹协议叫 RTCP，即实时传输控制协议，它由 RFC 3551 定义。一般来说，RTP 使用一个偶数 UDP 端口，而 RTCP 则使用 RTP 的下一个端口，也就是一个奇数端口。RTCP 为 RTP 媒体流提供信道外 (out-of-band) 控制。RTCP 本身并不传输数据，但和 RTP 一起协作将多媒体数据打包和发送。RTCP 定期在流多体会话参与者之间传输控制数据。RTCP 的主要功能是为 RTP 所提供的服务质量 (Quality of Service) 提供反馈。RTCP 收集相关媒体连接的统计信息，例如传输字节数，传输分组数，丢失分组数，抖动 (jitter)，单向和双向网络延迟等等，网络应用程序即可利用 RTCP 的统计信息来控制传输的品质，比如当网络拥塞比较严重时可以限制信息流量或改用压缩比较小的编解码器，在视频应用中重新请求关键帧等。

1.3 SDP 协议

那么，SIP 协议和 RTP 协议是怎么结合起来的呢？这就靠 SDP。SDP 称为会话描述协议，它寄生在 SIP 消息内部 (Body 中)，用于协商 RTP 协议将要用到的 IP 地址、端口以及媒体编码、带宽等。图1.4是一次呼叫的数据包在 Wireshark 里的分析界面。

从图1.4中可以看出，主叫一方的 UA (左) 发送 INVITE 消息，里面携带了 SDP；被叫方的应答消息 200 OK 中也带了 SDP。SDP 消息中包含将来 RTP 消息中传送媒体数据时将要用到的双方的 IP 地址和端口号以及媒体类型 (音频、视频、编码) 等信息。从图中可以看出，客户端的 SIP 端口号是 35526，音频端口号是 50452，视频端口号是 52974；FreeSWITCH 端的端口号则分别是 5060，31988 和 19008。

下面是完整的 SIP INVITE 消息：

```
INVITE sip:9196@192.168.1.9 SIP/2.0
Via: SIP/2.0/UDP 192.168.1.118:35526;branch=z9hG4bK-d8754z-0a09c74c6345dc09-1---d8754z;rport
Max-Forwards: 70
Contact: <sip:607@192.168.1.118:35526>
To: <sip:9196@192.168.1.9>
From: "607"<sip:607@192.168.1.9>;tag=f49f383a
Call-ID: ZTQON2Y2NzI2ZjMxZTcwZTY0YTA50DUyZDUzNWM2YjM
CSeq: 1 INVITE
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, NOTIFY, MESSAGE, SUBSCRIBE, INFO
Content-Type: application/sdp
Supported: replaces
```

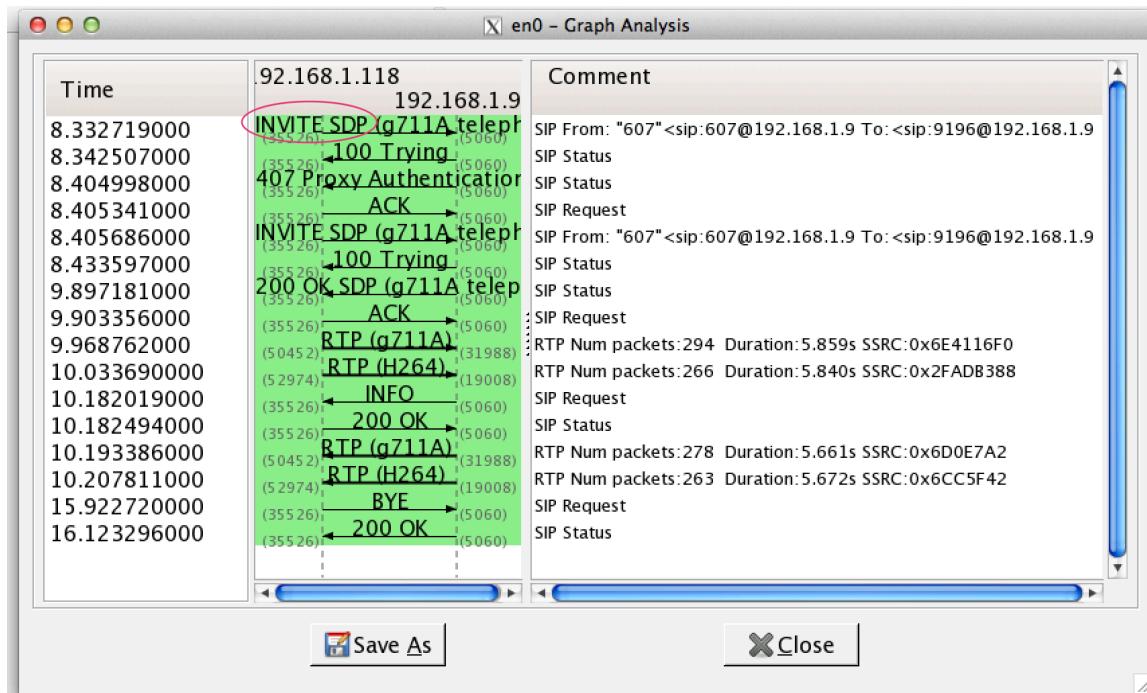


图 1.4: 带 SDP 的 SIP 呼叫

```
User-Agent: Bria 3 release 3.5.0b stamp 69410
Content-Length: 381

v=0
o=- 1371880105304943 1 IN IP4 192.168.1.118
s=Bria 3 release 3.5.0b stamp 69410
c=IN IP4 192.168.1.118
b=AS:2064
t=0 0
m=audio 50452 RTP/AVP 8 0 98 101
a=rtpmap:98 ILBC/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
a=sendrecv
m=video 52974 RTP/AVP 123
b=TIAS:2000000
a=rtpmap:123 H264/90000
a=fmtp:123 profile-level-id=428014;packetization-mode=0
a=rtcp-fb:* nack pli
a=sendrecv
```

其中，第 1 ~ 13 行为 SIP 消息，第 15 ~ 31 行为 SDP 消息。SDP 简要解释如下：

- v= Version，表示协议的版本号。

- o= Origin, 表示源。各项的涵义依次是 username、sess-id、sess-version、nettype、addrtype、unicast-address。
- s= Session Name, 表示本 SDP 所描述的 Session 的名称。
- c= Connecton Data, 连接数据。两个字段分别是网络类型和网络地址，以后的 RTP 流就为发到该地址上。注意，在 NAT 环境中我们要解决透传问题就是要看这个地址，这在以后的章节中也会讲到。
- b= Badwidth Type, 带宽类型。
- t= Timing, 起止时间。0 表示无限。
- m=audio Media Type, 媒体类型。audio 表示音频，50452 表示音频的端口号，应该跟上面图中的一致；RTP/AVP 是传输协议，这是里 RTP；后面是支持的 Codec 类型，与 RTP 流中的 Payload Type（负荷类型）相对应，在这里分别是 8、0、98 和 101。8 和 0 分别代表 PCMA 和 PCMU，它们属于静态编码，因而有一一对应的关系。而对于大于 95 的编码都属于动态编码，需要在后面使用“a=”进行说明。
- a= Attributes, 属性，它用于描述上面音频的属性，如本例中 98 代表 8000Hz 的 ILBC 编码，101 代表 RFC2833 DTMF 事件。a=sendrecv 表示该媒体流可用于收和发，其他的还有 sendonly（仅收）、recvonly（仅发）和 inactive（不收不发）。
- v= Video, 视频。可以看出它的端口号 52974 也是跟上面一致的。而且 H264 的视频编码对应的也是一个动态 Payload，在本例中是 123。

1.4 小结

从本节的介绍可以看出，SIP 消息与 HTTP 消息是很类似的。不同的是，在实际应用中，HTTP 消息多承载以 TCP 上，而 SIP 消息则多承载在 UDP 上（当然，也有相当多的应用中 SIP 消息会承载在 TCP 上）。为了能在 UDP 上可靠的传输 SIP，SIP 协议中在应用层有相应的重传和超时机制。

RTP 协议主要用于传输媒体，为了保证实时性，绝大部分实现都是基于 UDP 的。关于实时性，可以考虑视频点播与直播的区别。对于点播，用户一般期望看到完整的视频，中间“卡”一点也问题不大，所以可以用 TCP 实现；但对于直播来说，如果视频出现“卡”的情况，那就要使用降低码率或丢弃数据包的方式，以避免影响后面的视频，保证实时性，所以使用 UDP 实现比较合适。

UDP 是无连接的，但在 SIP 和 SDP 协议中，都有相关的域名和 IP 地址、端口参数，这使得穿越 NAT 变得困难，在实际应用中也是很令人头痛的地方。

SIP 协议称为信令协议，它为了建议 UA 间的会话，主要是为了协商媒体如何传输。而 RTP 称为媒体协议，用于真正的传输媒体。实际上，信令协议也不一定使用 SIP，更早的 H323 协议也可以进行媒体协商并最终使用 RTP 协议传输媒体。而 Google 主导的 WebRTC 更是没有规定信令是如何实现的，只要信令能交换双方的 SDP 就能建立 RTP 媒体传输。

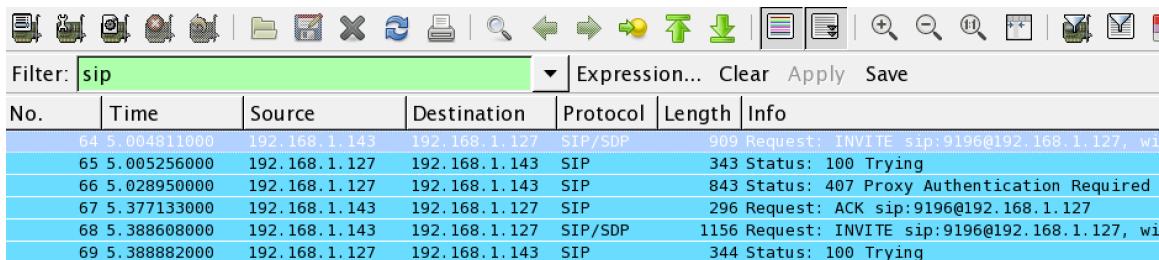
篇幅所限，关于这些基本知识的介绍就到这里了。讲理论一般比较枯燥，还是看些实际的案例好玩一些。

第二章 用 Wireshark 分析 SIP 呼叫

Wireshark 专门有一个 Telephony 菜单用于分析各种电话协议，对 SIP/RTP 的支持也是非常的完善。我们先来熟悉一下。

2.1 分析 SIP 包

首先，我们先来看 SIP 信令。由于我们抓到了很多包，为了看起来方便，我们可以使用“sip”过滤器，只看我们关心的包。如图2.1所示，我们可以在“Filter:”一栏中输入“sip”，便可以只显示所有的 SIP 包。



The screenshot shows the Wireshark interface with a list of captured SIP packets. The 'Filter:' field at the top is set to 'sip'. The packet list table has columns: No., Time, Source, Destination, Protocol, Length, and Info. The 'Info' column displays the SIP message details.

No.	Time	Source	Destination	Protocol	Length	Info
64	5.004811000	192.168.1.143	192.168.1.127	SIP/SDP	909	Request: INVITE sip:9196@192.168.1.127, wi
65	5.005256000	192.168.1.127	192.168.1.143	SIP	343	Status: 100 Trying
66	5.028950000	192.168.1.127	192.168.1.143	SIP	843	Status: 407 Proxy Authentication Required
67	5.377133000	192.168.1.143	192.168.1.127	SIP	296	Request: ACK sip:9196@192.168.1.127
68	5.388608000	192.168.1.143	192.168.1.127	SIP/SDP	1156	Request: INVITE sip:9196@192.168.1.127, wi
69	5.388882000	192.168.1.127	192.168.1.143	SIP	344	Status: 100 Trying

图 2.1: 使用了“sip”过滤器只显示 SIP 包

我们可以通过鼠标定位到 INVITE 消息，如下图，读者可以在下面的框里看到消息的内容。虽然它看起来不如直接在 FreeSWITCH 中直接用 siptrace 抓出来的纯文本的包直观，但是，Wireshark 可以对 SIP 消息做深入的分析，可以显示更多的帮助信息。如图2.2，显示了 SIP 中的 INVITE 消息的详细情况。

Wireshark 甚至有专门分析 VoIP 通话的工具。在主菜单中，选择“Telephony”->“VoIP Calls”可以看到所有的 VoIP 呼叫，如图2.3所示。

此时，选择一路呼叫，并点击“Flow”按钮，就可以看到详细的 SIP 呼叫流程。如图2.4。

有了前面的 SIP 基础，读者就可以按照这个流程自己试一下了。打几个正常的或者不正常的电话，比较一下它们的异同。

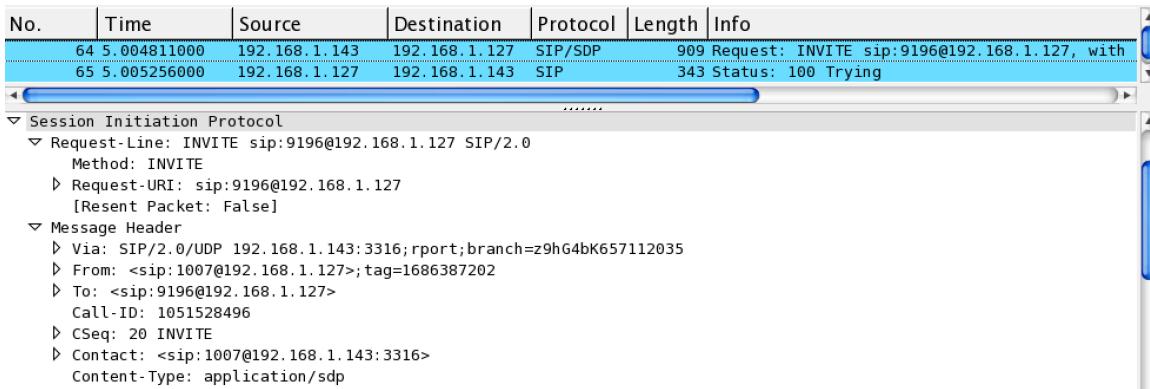


图 2.2: 在 Wireshark 中查看 INVITE 消息

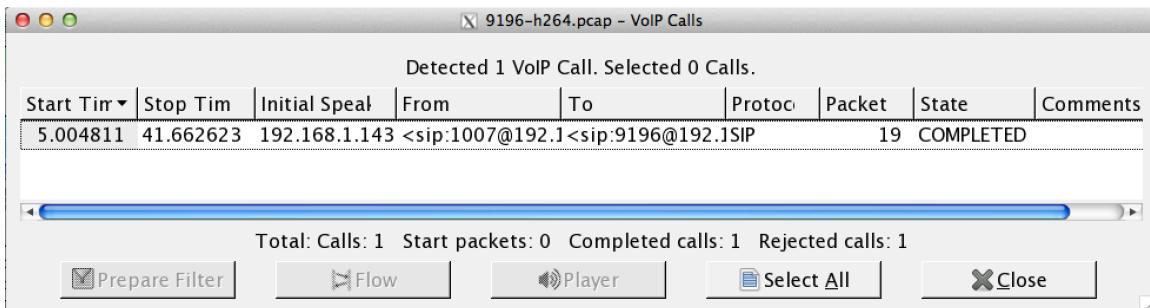


图 2.3: 一路 VoIP 通话

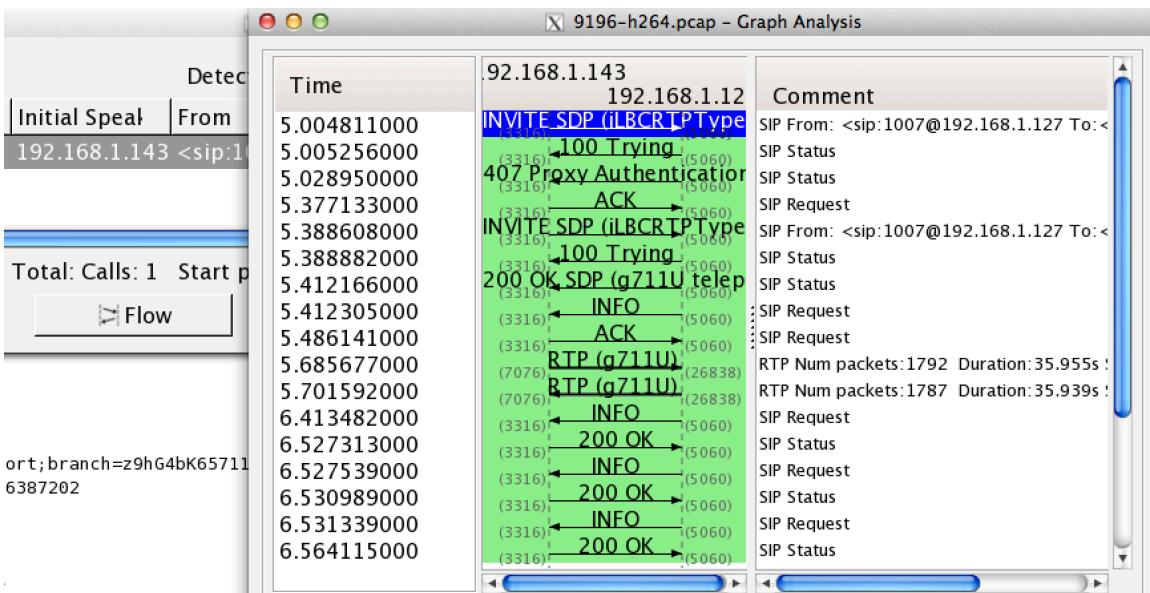


图 2.4: Wireshark 中分析呼叫流程

另外，在上一步中，“Flow”按钮的旁边有一个“Play”按钮，打开以后点击 Decode 可以将音视频流的数据进行解码，并可以播放音频流里的声音。注意，如果音频不是以 PCMU 或 PCMA 格式编码的，则不能播放。关于 RTP 流的详细分析我们将在下一节进行讨论。

2.2 分析 RTP 包

在 Wireshark 中也可以分析 RTP 包。在 Filter 栏中输入“RTP”就可以看到所有的 RTP 包。一般来说，可以先肉眼大体看一下收发是否规律。详细的分析可以使用“Telephony”->“RTP”->“Show All Streams”显示所有的 RTP 流，如图2.5，它显示了我们这次呼叫中涉及到的两路 PCMU 的音频流和两路 H264 的视频流：

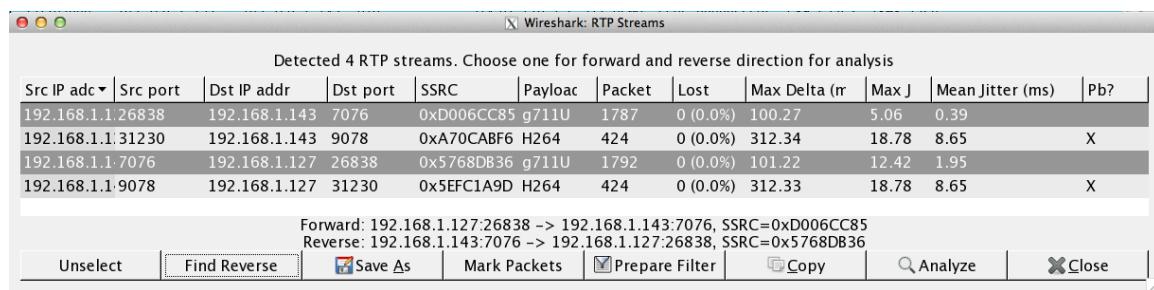


图 2.5: 一个呼叫中的 4 路 RTP 流

先选择一路音频流，通过点击“Find Reverse”按钮，可以找到选择到反向的流。然后点击“Analysis”就可以看到两路 RTP 流的详细分析。如图2.6所示。

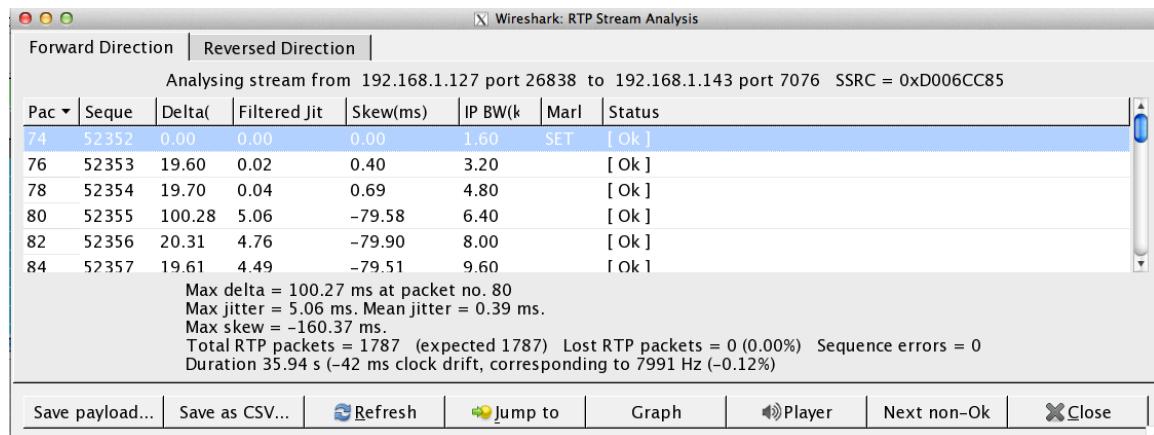


图 2.6: Wireshark 中的 RTP 流分析

在分析中，也可以通过上一节提到的“Play”播放音频流，或者使用“Save Payload …”按钮将音频数据存到单独的文件中。如果使用 au 格式，则可以在同一个文件中保存双向的音频流；但如果使用 raw 格式，只能将单向的音频流分别存到不同的文件里。另外，如上一节所说，如果语音编码不是 PCM 格式的话（如 G729），则音频流只能存成 raw 格式，然后再用其他工具去分析。

除此之外，还可以使用“Graph”按钮生成某一路流相关的统计图，如果图表显示比较规律，那一般说明是比较好的，如果生成的图表不是很规律，那就可能是网络上有丢包、延迟或抖动等。解决音频或视频质量的问题需要具体问题具体分析，读者可以在学习中慢慢积累经验。

最后值得一提的是，如果是分析视频流，可能会在视频分析结果中发现“Incorrect Timestamp”之类的提示，有时候这个提示不一定准确，因为该功能目前只适合分析音频流。

2.3 理解 UDP 重发

UDP 是不可靠的数据传输协议。所以，要基于 UDP 做点靠谱的事情，就需要在应用层适当的发送冗余数据或进行重发。数据冗余一般涉及比较高深的算法，在此，我们只讨论重发。

UDP 重发一般有两种：预防性重发和基于反馈机制的按需重发。

2.3.1 预防性重发

我们常说，“一切艺术都是来源于生活又高于生活”，其实技术也不例外。下面我们就看一则生活中的例子。如图。



图中，路人甲喊贾君鹏的时候，第一遍没收到贾的回应，很自然地又喊了一遍，这便是预防性重发的例子。

SIP 就是基于预防性重发设计的。如果对照 SIP 协议，就会发现路人甲喊贾君鹏就相当于发 INVITE，而贾的回应“唉……”就相当于 100 Trying。如果主叫用户发出 INVITE 超过一定时间没有收到 100 Trying 时，便会预防性的重新发 INVITE……

2.3.2 按需重发

在 TCP 协议中，使用 ACK 机制已经内置了确认和重发的机制，如果某个包没得到确认，就会重发。而在 UDP 协议中，没有相应的机制，因此，确认和重发需要在应用层实现。

用于传输媒体的 RTP 配合 RTCP 可以实现按需重发。RTCP 使用 NACK 机制，即不像 TCP 协议中采用明确的确认收到了哪些包，而是在接收端检测到丢包时（RTP 中的 Seq 号不连续），通过 NACK 向发送者请求重新发送某个 RTP 包。关于这一点具体可参考图6.3。

第三章 看不到视频

有一次使用我们的系统跟国内某大厂 MCU 设备对接时，无论如何都看不对方的视频，对方收我们的视频倒是正常的。我们不能确定是我们的系统问题还是对方问题。因此，我们便换了一个国际上用得比较多的 SIP 客户端软件 (Bria) 与该 MCU 对接，发现问题依旧。当时我们的水平也不高，但我们更倾向于相信 Bria。抓了一个包，发给 MCU 设备厂家请他们帮忙排查问题。然后，就没有然后了。

就在写本书时，笔者到处搜索以前的抓包记录，又把原来的数据包翻出来。用 Wireshark 分析一看，哈，一目了然，明显是大厂的设备有问题么。

如下图。通过菜单“Telephony” -> “VoIP Calls”显示数据包中只有一路通话，继续点击“Flow”显示详细的流程。很明显 MCU 返回的数据包（高亮的一行）显示的RTPType-109比较可疑。

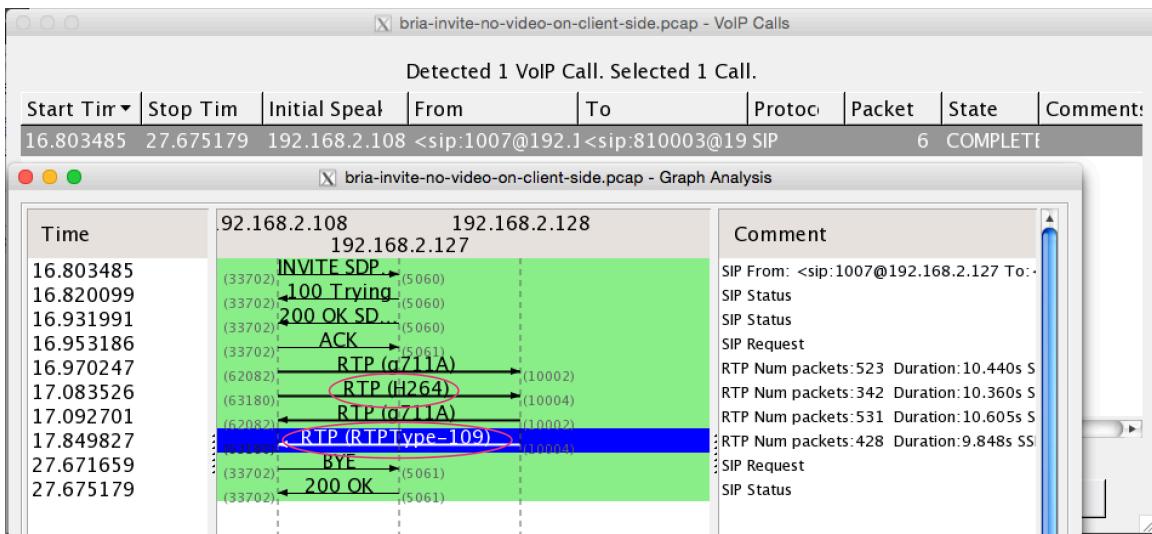


图 3.1: 通话流程图

在上图中，左侧是 Bria，右侧是 MCU。MCU 的 SIP 地址是 192.168.2.127，而媒体地址是 192.168.2.128，这都是正常的。Bria 向 MCU 发送的视频数据包显示为 RTP(H264)，看起来也正常。那为什么对于 MCU 返回的视频数据包，Wireshark 不显示 RTP(H264) 而是显示奇怪的 RTPType-109 呢？

选中流程图中的 INVITE 消息，打开详细的选项可以看到 INVITE 携带的 SDP 信息，如下图。Bria 发送的 SDP 消息中有两个不同的编码选项分别是123和124，该值称为 Payload Type（简称 PT）。`rtpmap`指明它们都代表 H264，只是`fmtpt`字段指明两种 H264 编码的打包方式略有不同。总之这些内容指定了 PT 的值与实例代表的视频编码的对应关系（这里是 H264）。

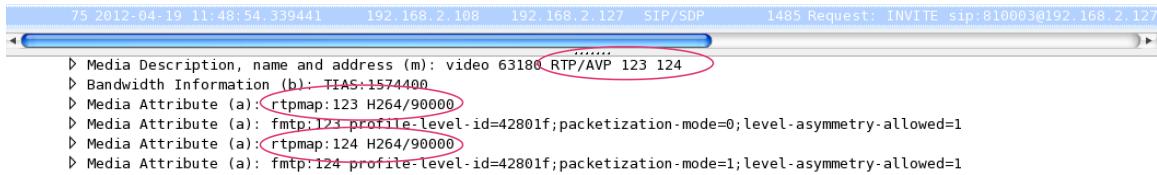


图 3.2: 主叫 Bria 发送的 INVITE 消息及 SDP

我们再看 MCU 回复的 200 OK 消息。它里面也带了 SDP，如下图。MCU 在 SDP 中指出它将用 PT 值109代表 H264 编码。

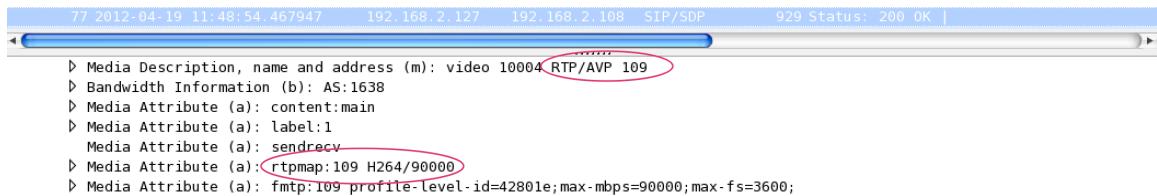


图 3.3: 被叫 MCU 回复的 200 OK 消息及 SDP

SIP 协商完毕后，双方就互相发 RTP 消息。Bria 按照约定，给对方发送 H264 视频流时在 RTP 的 PT 字段中写上了 109（图 X）；但 MCU 却没有理会 Bria 只认识123或124的事实，在发送 H264 视频流时固执地写上了只有自己才认识的 PT 值，也是 109（图 Y）。

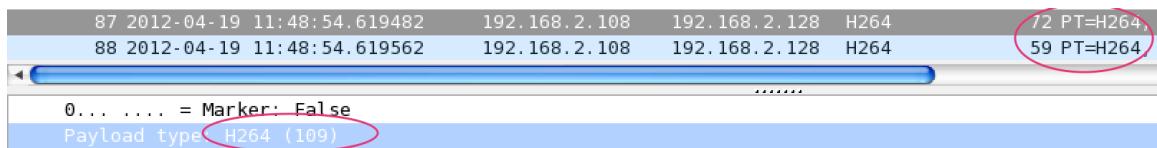


图 3.4: 主叫发送正确的 PT 值

所以，虽然双方都互相发送 RTP 视频流，但 Bria 不认识，自然就不能显示视频了。

这是一个常见的对 SIP/RTP 协议的实现错误。RTP 协议中只有一个字节表示 RTP 中实际传输的媒体数据类型，这一个字节就是 PT。有一些媒体编码，PT 值与编码之间有一一对应的关系，如 PCMU（音频）的 PT 值为 0，PCMA（音频）为 8，H263（视频）为 34。但为每种编码都指定一一对应的关系是不可能的，因此，规定所有大于 95 的 PT 值为动态编码，只能在实现时在 SDP 中通过`rtpmap`来动态映射。而且，如果通信双方的映射不一致时（就是我们本例中遇到的情况），发送方要使用对方指定的 PT 值发送，而用本方的 PT 值接收。但遗憾的是，很多开发人员没有意识到这一点，因此跟其它厂家的设备对接就出现了问题。大厂的人也不例外。

在本例中，MCU 在 RTP 中发送了错误的 PT 值，Bria 不认识（不显示视频），Wireshark 也不

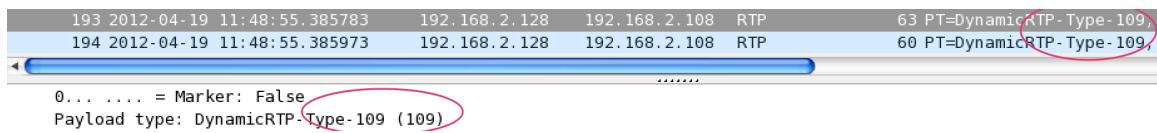


图 3.5: 被叫发送错误的 PT 值

认识 (在流程图上不显示 H264)。

第四章 解析 H264 数据包

笔者在给 FreeSWITCH 添加 H264 视频功能时，费了不少劲到处找资料。后来发现在 Wireshark 中可以很方便地分析 H264 数据包，真是舍近求远啊。下面，就跟大家分享一下这一发现。

还是以我们上一节数据包为例。在呼叫流程图上点击 RTP(H264)那一条就可以定位到第一个 RTP 数据包。H264 的数据包有很多 NALU 组成，其中第一个字节表示 NALU 的类型。一般来说，第一个 NALU 包的类型是 7，即 Sequence Parameter Set，简称 SPS，它描述了 H264 视频流的一些基本参数。如下图：

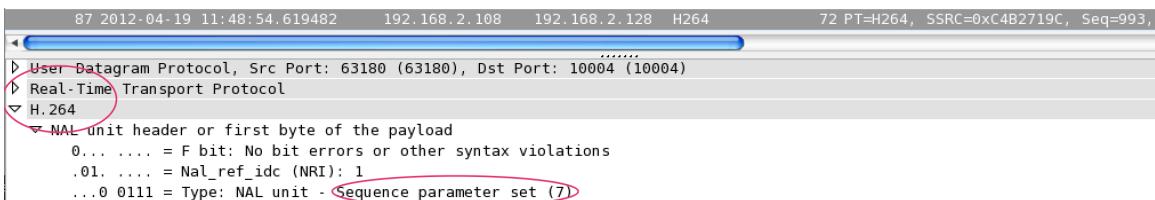


图 4.1: H264 包

如下图，我们比较常用的就是 H264 的 Profile、Level ID，以及视频的分辨率等。

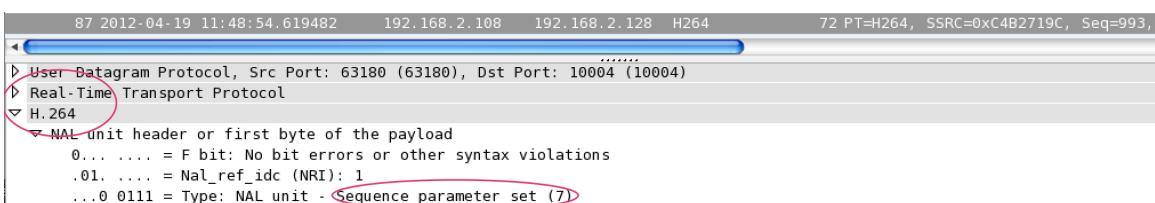


图 4.2: H264 包

其中，视频的分辨率是由 `pic_width_in_mbs_minus1` 和 `pic_height_in_mbs_minus1` 计算出来的，计算公式是：

```
宽度 = (pic_width_in_mbs_minus1 + 1) * 16
高度 = (pic_height_in_mbs_minus1 + 1) * 16
```

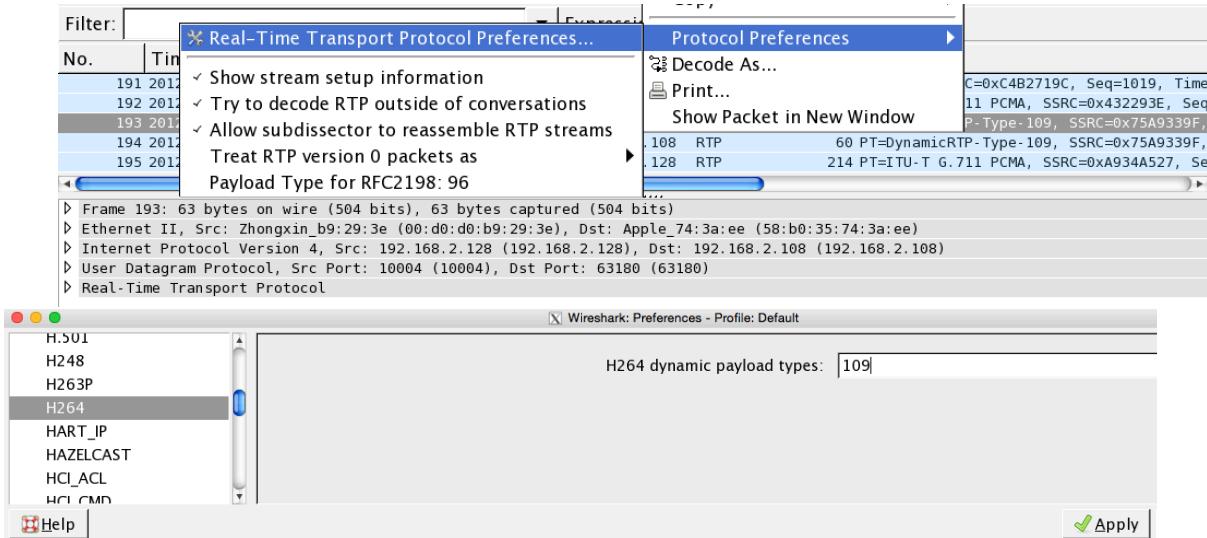
所以，上图中的视频分辨率为 172×144 ，即：

$$(10 + 1) * 16 = 176$$

$$(8 + 1) * 16 = 144$$

那么，问题来了。我们在上一节说到，由于 MCU 发送的数据包中 PT 值错误，Wireshark 根本不知道 RTP 里传的就是 H264，进而无法解包。如果我们想知道视频的参数怎么办呢？

其实也简单。选中 PT 值为 109 的一条 RTP 记录，点击右键，然后选择“Protocol Preference”->“Realtime Transport Protocol Preference”，在弹出的窗口中再找到“H264”，把“H264 dynamic payload types”中填入“109”，然后点击“Apply”即可。关闭弹出窗口，就可以看到 H264 包已经能正确的解析了。如图。



这种方法对纯 RTP 的数据包也管用。在整个抓包中没有 SIP 消息可以关联时，Wireshark 不仅不认识 H264，连 RTP 包也认不出来。方法是找到一条疑似是 RTP 的包（如何找就靠经验了，当然肯定是 UDP 包），然后点击右键菜单中的“Decode As”，在弹出的窗口中选择 RTP 就可以将类似的包解析成 RTP。如果你知道 RTP 里带的是 H264，可以进而用上述方法解析 H264。当然，其它编码如 VP8 等也类似。

既然说到这里，我们不妨继续分析一下 H264 中的各种 NALU。如图。

网络层加上 RTP 的包头一共是 54 个字节。

87 号包，是 Sequence Parameter Set，简称 SPS，我们上面已经分析过了。它的 nri 值是 1，NAL Type 是 7，所以，第一个字节是 $(1 << 5) || 7 = 0x27$ ，一共是 $72 - 54 = 18$ 字节。

88 号包，是 Picture Parameter Set，简称 PPS，它的 NAL Type 是 8，因此第一个字节是 $0x28$ ，一共是 $59 - 54 = 5$ 字节。

SPS 和 PPS 是 H264 编码器的参数，必须有这些参数才可以解码。

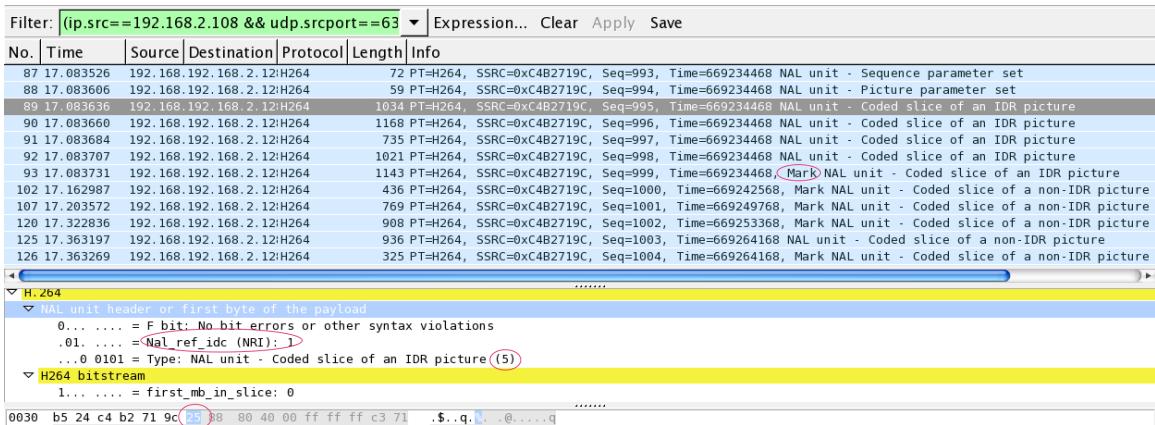


图 4.3: H264 包

89 号包到 93 号包，Coded slice of an IDR Picture，关键帧。NAL Type 是 5，因此第一个字节是 0x25。

我们知道，视频其实是一帧一帧的图像。每秒种发送多少帧就做帧频或帧率，即 Frames Per Second，简称 FPS。FPS 越大，视频越流畅。H264 压缩算法是将一帧帧的图像进行压缩，并得到压缩后的 NALU。其中，第一帧称为关键帧，或 I 帧，全压缩；后续的帧则只压缩两帧图像间变化的部分，称为非关键帧或 P 帧；除此之外还有 B 帧，但在 VoIP 中一般不使用。

所以，我们看到，在紧接着 SPS 和 PPS 以后，就是一幅关键帧。由于压缩后的数据大于网络的 MTU，所以，将压缩后的数据分成多个 RTP 包传送，每个 RTP 包一般都不大于 1200。我们看到，93 号包有一个 Mark，即 RTP 头域中的 mark 值为 1，标志一帧图像的结束。

因为 89 到 93 号包表示的是同一帧图像，所以，所有的包的时间戳（Time）都是相等的。

接下来，102 号包，non-IDR 表示非关键帧，NAL Type 是 1，因此第一个字节是 0x21。它只有 $1143 - 54 = 1089$ 个字节，因此不用分包，因为有 Mark，所以单独是一帧，同时时间戳变了。

107、120 号包也 102 类似。

125 和 126 号包时间戳相同，125 没有 Mark，126 有 Mark，因此这两个包是一帧图像。

数一数 1 秒内有多少个 Mark 可以大致估算 FPS。另外也可以通过时间戳估算。视频的采样频率固定为 90000（可以在 SDP 中看到），我们取相临两个包算出两个时间间隔：

- 93 到 102 号包的时间间隔为： $669242568 - 669234468 = 8100$
- 102 到 103 号包的时间间隔为： $669249768 - 669242568 = 7200$

可以分别得出两个值：

- $90000 / 8100 = 11.11$
- $90000 / 7200 = 12.5$

因此，FPS 大约为 12 帧每秒。

根据图 X，用同样的方法我们可以计算出 MCU 发送给我们的视频参数：

No.	Time	Source	Destination	Protocol	Length	Info
193	17.849827	192.168.192.168.2.10:H264			63	PT=DynamicRTP-Type-109, SSRC=0x75A9339F, Seq=1, Time=10000 NAL unit - Sequence parameter set
194	17.850017	192.168.192.168.2.10:H264			60	PT=DynamicRTP-Type-109, SSRC=0x75A9339F, Seq=2, Time=10000 NAL unit - Picture parameter set
196	17.850511	192.168.192.168.2.10:H264			1419	PT=DynamicRTP-Type-109, SSRC=0x75A9339F, Seq=3, Time=10000 NAL unit - Coded slice of an IDR picture
197	17.851182	192.168.192.168.2.10:H264			1031	PT=DynamicRTP-Type-109, SSRC=0x75A9339F, Seq=4, Time=10000, Mark NAL unit - Coded slice of an IDR pic
203	17.890200	192.168.192.168.2.10:H264			831	PT=DynamicRTP-Type-109, SSRC=0x75A9339F, Seq=5, Time=13600, Mark NAL unit - Coded slice of a non-IDR
209	17.931193	192.168.192.168.2.10:H264			763	PT=DynamicRTP-Type-109, SSRC=0x75A9339F, Seq=6, Time=17200, Mark NAL unit - Coded slice of a non-IDR
213	17.972304	192.168.192.168.2.10:H264			1397	PT=DynamicRTP-Type-109, SSRC=0x75A9339F, Seq=7, Time=20800 NAL unit - Coded slice of a non-IDR pictur
215	17.974259	192.168.192.168.2.10:H264			177	PT=DynamicRTP-Type-109, SSRC=0x75A9339F, Seq=8, Time=20800, Mark NAL unit - Coded slice of a non-IDR

.... = gap_size_in_map_units_minus1: 21
.... . . . 0 0001 0110 = pic_width_in_mbs_minus1: 21
0000 1001 0... . . . = pic_height_in_map_units_minus1: 17

图 4.4: H264 包

- SPS 长度: $63 - 54 = 9$ 字节
- PPS 长度: $60 - 54 = 6$ 字节
- 图像大小: 352×288 即 $(21 + 1) * 16 = 352$, $(17 + 1) * 16 = 288$
- FPS: $90000 / 3600 = 25$ 帧每秒

看来，MCU 确实给我们发送了正确的 H264 数据，只是 RTP 中的 PT 值用错了。

好玩吧？

第五章 异地多活

今年（2015）前一段时间，互联网上出了几个大事，从携程的程序被误删、到支付宝服务器被挖光缆等，使人们越来越意识到多个数据中心互为备份的重要性。当发生紧急事件时能快速切换到可用的数据中心，以最大程度上保障业务。然而，鉴于目前互联网业务的复杂性，真正实现所谓的异地多活可谓非常困难，主要的问题就是交易状态和数据的实时复制。

然而，难归难，小概率事件也不是不可能事件，因此，对于重要的业务，还是要做到异地多活才行。笔者搭建了一个 SIP 服务平台，从最开始就是基于这个考虑来的。当然，笔者的主要目的并不是为了高可用，更多的是为了练手。

我们先来看一下笔者测试的一个异地双活的截图。这个图是在客户端侧抓的。

No.	Time	Source	Destination	Protocol	Length	Info
302	2015-10-04 14:37:32	192.168.7.6	sip2.sipsip.cn	SIP/SDP	1210	Request: INVITE sip:test@sipsip.cn
304	2015-10-04 14:37:33	sip2.sipsip.cn	192.168.7.6	ICMP	590	Destination unreachable (Port unreachable)
309	2015-10-04 14:37:33	192.168.7.6	sip2.sipsip.cn	SIP/SDP	1210	Request: INVITE sip:test@sipsip.cn
310	2015-10-04 14:37:34	sip2.sipsip.cn	192.168.7.6	ICMP	590	Destination unreachable (Port unreachable)
315	2015-10-04 14:37:35	192.168.7.6	sip2.sipsip.cn	SIP/SDP	1210	Request: INVITE sip:test@sipsip.cn
316	2015-10-04 14:37:36	sip2.sipsip.cn	192.168.7.6	ICMP	590	Destination unreachable (Port unreachable)
337	2015-10-04 14:37:39	192.168.7.6	sip2.sipsip.cn	SIP/SDP	1210	Request: INVITE sip:test@sipsip.cn
338	2015-10-04 14:37:40	sip2.sipsip.cn	192.168.7.6	ICMP	590	Destination unreachable (Port unreachable)
386	2015-10-04 14:37:47	192.168.7.6	sip2.sipsip.cn	SIP/SDP	1210	Request: INVITE sip:test@sipsip.cn
392	2015-10-04 14:37:48	sip2.sipsip.cn	192.168.7.6	ICMP	590	Destination unreachable (Port unreachable)
609	2015-10-04 14:38:03	192.168.7.6	sip2.sipsip.cn	SIP/SDP	1210	Request: INVITE sip:test@sipsip.cn
611	2015-10-04 14:38:04	sip2.sipsip.cn	192.168.7.6	ICMP	590	Destination unreachable (Port unreachable)
622	2015-10-04 14:38:05	192.168.7.6	sip1.sipsip.cn	SIP/SDP	1210	Request: INVITE sip:test@sipsip.cn
637	2015-10-04 14:38:05	sip1.sipsip.cn	192.168.7.6	SIP	388	Status: 100 Trying
638	2015-10-04 14:38:06	sip1.sipsip.cn	192.168.7.6	SIP/SDP	1150	Status: 200 OK
639	2015-10-04 14:38:06	192.168.7.6	sip1.sipsip.cn	SIP	418	Request: ACK sip:test@121.40.231.235:5080;transport=udp
1218	2015-10-04 14:38:11	sip1.sipsip.cn	192.168.7.6	SIP	619	Request: BYE sip:mod_sofia@27.216.184.13:5060
1219	2015-10-04 14:38:11	192.168.7.6	sip1.sipsip.cn	SIP	548	Status: 200 OK

图 5.1: 异地双活

从图中可以看出，SIP 客户端向服务器 test@sipsip.cn 发起呼叫请求 (INVITE, 302 号包)。sipsip.cn 首先被 DNS 解析到 sip2.sipsip.cn，由于服务器上的软交换应用服务没有开启，因此，服务器回复 ICMP 包表明目的地不可达 (304 号包)。如此重试几次后，客户端自动切换到另一条 DNS 记录所指的服务器 sip1.sipsip.cn 上 (622 号包)。呼叫正常建立和释放。从下图的呼叫流程图上能更直观的看到这一过程。

另外，从图中我们也可以看到客户端重发 INVITE 消息的策略是每隔 1、2、4、8、16 秒重发，并在 $1 + 2 + 4 + 8 + 16 = 31$ 秒后超时，尝试下一个可用的服务器。本例是基于 UDP 的，这也是 UDP 为了保证消息可靠性在收不到对方的响应时主动重发的一个例子。

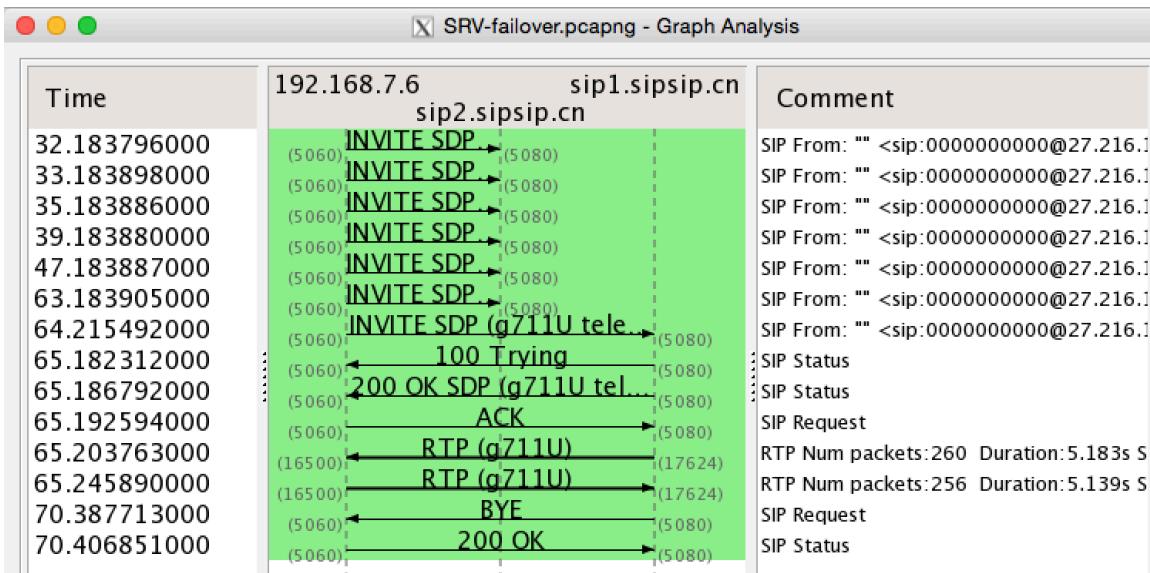


图 5.2: 异地双活

在本例中，DNS 的双备份是使用 SRV 记录来实现的。SRV 记录其实是一种服务记录，类似于邮件服务的 MX 记录。MX 记录诞生比较早，因此，单独占了一种类型，而随着 SIP、XMPP 等其它越来越多的服务的出现，为每一种服务都单独创造一种记录类型显然不太明智，因此，就有了 SRV 记录。下图就是 SRV 记录的查询过程。

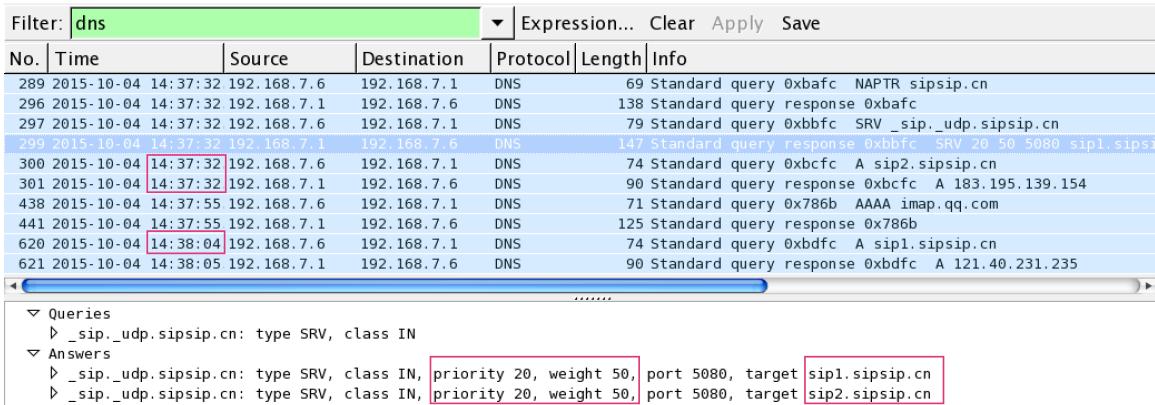


图 5.3: SRV 记录查询过程

从图中可以看到，我们的 SIP 客户端首先查找 `sipsip.cn` 的 NAPTR 记录（289 号包），由于我们没有使用 NAPTR 记录，接下来它又中查找 SRV 记录（299 号包），并得到两个 `target` 分别指向 `sip1.sipsip.cn` 和 `sip2.sipsip.cn`。由于两条记录的优先级（priority）权重（weight）都一样，因此客户端再随机选择一个，查询其 A 记录所指向的 IP 地址（300 号包）。向第一个地址呼叫失败后大约在 32 秒后查询另外一个（620 号包）。

当然，客户端应该会缓存 DNS，它知道某个 DNS 指向的 IP 地址不可用后在一段时间内就不会再次发送请求，以提升用户体验。

最后，需要说明的是，如果不使用 SRV 记录而仅使用 A 记录也能完成 DNS 的轮循备份，但是 SRV 记录可以指定更多的参数，如优先级和端口号等。

第六章 音视频不同步问题

有一天收到一个紧急故障工单，一个客户即将上线的系统上在最后测试阶段发现音视频有很大的概率不同步，视频比音频有时会有 3 秒延迟。

这个问题确实比较严重，而且也比较棘手。大家都知道，处理问题时遇到最令人头痛的问题就是时好时坏，指不定你在排查的时候就好了，回头客户又告诉你不好。处理这种问题也没有什么捷径，只能靠多花时间测试。好在经过几天的测试后，逐渐缩小了范围——用了客户的专用 SIP 客户端就会出这个问题，而我们使用其它的 SIP 客户端却不出问题。所以问题基本出在客户端上，建议他们找做客户端的人去解决。但是，拿人钱财，与人消灾，做为服务端的维护人员，还需要配合客户排查。

用户的拓扑情况如下图：

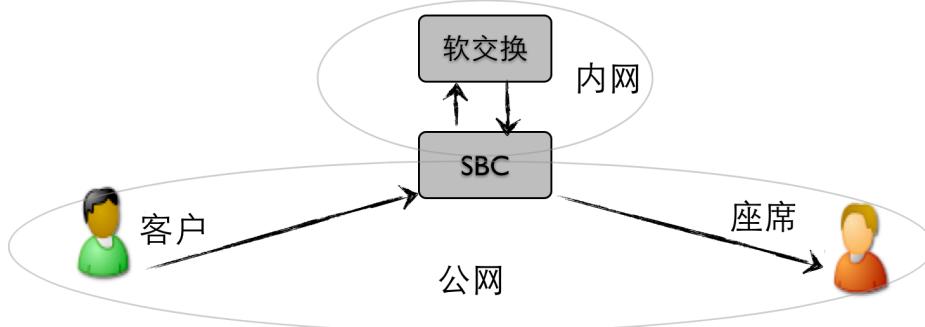


图 6.1: 网络拓扑

其中，我们的客户运营的是一个视频呼叫中心系统。座席端使用 Windows 版的 SIP 客户端，而客户的客户则使用 Pad (平板) 版的 SIP 客户端。两种客户端均基于同一厂家的 SIP 产品 SDK 开发。系统运行在公网上，所以，软交换放在企业内网上，通过 SBC (有信令和媒体转发及防火墙功能) 与公网通。

数据包通过这么多设备，每个点都有可能出问题。好在客户已经被我们培训的很有经验了，在报问题的时候就给我们提供了从 Pad 到 SBC、软交换经座席端的所有抓包。分析这么多包不是一件容易的事，尤其是各参考点的时钟还不一致，更增加了分析比对的难度，但我们还是得做。

经过分析我们发现一个问题。如下图。在 Pad 端的抓包中，使用 `rtp.seq>=10417 and rtp.seq<=10420` 过滤，可以看到有一个 Seq 为 10418 的包比 Seq 为 10419 的包大约晚了 3 秒到达（后面还有

一个 Seq 为 10418 的包更是在 12 秒后到达，但由于与前面的 10418 重复会被丢弃，在此我们就不用考虑了），是不是这就是视频延迟的原因呢？

Filter: (rtp.seq>=10417 and rtp.seq <=10420)							Expression... Clear Apply Save
No.	Time	Source	Destination	Protocol	Length	Info	
643	2015-08-05 15:48:19.646922	sbc	pad	H264	733	PT=H264, SSRC=0x15807737, Seq=10417, Time=2346699882	
644	2015-08-05 15:48:19.646939	sbc	pad	H264	718	PT=H264, SSRC=0x15807737, Seq=10419, Time=2346705642	
665	2015-08-05 15:48:19.736618	sbc	pad	H264	1214	PT=H264, SSRC=0x15807737, Seq=10420, Time=2346712842	
1353	2015-08-05 15:48:22.556656	sbc	pad	H264	1214	PT=H264, SSRC=0x15807737, Seq=10418, Time=2346705642	
1442	2015-08-05 15:48:22.877094	sbc	pad	H264	1214	PT=H264, SSRC=0x15807737, Seq=10420, Time=2346712842	
3372	2015-08-05 15:48:31.039017	sbc	pad	H264	1214	PT=H264, SSRC=0x15807737, Seq=10418, Time=2346705642	

图 6.2: Pad 端抓包分析

接着我们使用(rtp.seq>=10417 and rtp.seq <=10420) or rtcp.rtpfb.nack_pid==10418为条件对包进行过滤，发现 Pad 端在收到 Seq 为 10419 的包后认为 10418 发生了丢包，然后发送了一个 RTCP NACK 数据包请求对方重新将 Seq 为 10418 的包发送一遍。如图6.3。

Filter: (q <=10420) or rtcp.rtpfb.nack_pid==10418							Expression... Clear Apply Save
No.	Time	Source	Destination	Protocol	Length	Info	
643	2015-08-05 15:48:19.646922	sbc	pad	H264	733	PT=H264, SSRC=0x15807737, Seq=10417, Time=2346699882,	
644	2015-08-05 15:48:19.646939	sbc	pad	H264	718	PT=H264, SSRC=0x15807737, Seq=10419, Time=2346705642,	
645	2015-08-05 15:48:19.651704	pad	sbc	RTCP	146	Sender Report Source description Payload-specific	
647	2015-08-05 15:48:19.661836	pad	sbc	RTCP	146	Sender Report Source description Payload-specific	

Media source SSRC: 0x15807737 (360740663)
RTCP Transport Feedback NACK PID: 10415
RTCP Transport Feedback NACK BLP: 0x0004 (Frames 10418 lost)
[RTCP frame length check: OK - 104 bytes]

图 6.3: NACK

对方果然重发了一遍 10418 包，不过，这个包晚了 3 秒到达，因此，在 Pad 端就晚上 3 秒播放视频，造成了音视频不一致的现象。

为了验证是否是 RTCP 引起的问题，我们在服务端禁用了 RTCP 转发，发现不同步的情况有明显改善，但视频画质出现问题，经常会花屏（一般是由于丢包引起的）。

重新打开 RTCP 转发继续排查问题。那么，10418 为什么晚了 3 秒呢？简要分析有以下两种可能：

- 1) RTCP 用了差不多 3 秒的时间才传到对端，对端又急忙重发 10418 包，因此晚了
- 2) RTCP 很快就到了对端，而对端重发 10418 晚了或者是到达晚了

这里所说的对端包括数据包经过的所有路径——SBC，软交换，以及座席端。接下来的事情就是分析所有的参考点上的包，查一查到底是哪部分造成了包转发慢。

限于篇幅，详细的过程我们就不讲了，最终我们发现问题出在软交换，软交换在转发 RTCP 时用了将近 3 秒的时间。

但在排查软交换的转发性能时我们又发现了一个现象。我们随机抽了一些通话开始时和快结束时的 RTCP 包（没有丢包也会发其它的 RTCP），发现转发并不慢。所以转发慢只出现在中间某

个过程。又经过大量的分析和调试，最后发现原因是 Pad 端短时间内发送了大量的 RTCP 包（符合`rtcp.rtpfb.nack_pid==10418`条件的包有上百个），导致出现阻塞。后来修改了客户端软件，降低了 RTCP 包的发送频率，问题解决。

本例中根据 RTCP 重发 RTP 包也是 UDP 被动重传的一个很好的例子。

第七章 NAT

SIP 和 SDP 的设计就决定了它们与 NAT 割舍不断的孽缘。NAT 的出现是为了解决 IP 地址不足问题，但 SIP 协议设计的太灵活，以至于很多地方都可以定义 IP 地址和端口，而且，SIP 和 RTP 又大都是基于 UDP 实现的，这使得位于 NAT 后面的 SIP 设备穿越 NAT 变得困难。

当然，协议设计出来就是应该能使用的，因此，聪明的人们又想出了无数的方法解决 NAT 的问题。下面，我们就看一下实例以及解决办法。

7.1 为什么呼不通？

在这个例子中，客户端和服务器都没有开启 NAT 支持（就是设备不够聪明，根本不知道 NAT 存在）。所以，当服务器呼叫客户端时，SIP 消息从服务器端无法送达客户端的私网地址，因而无法呼通。

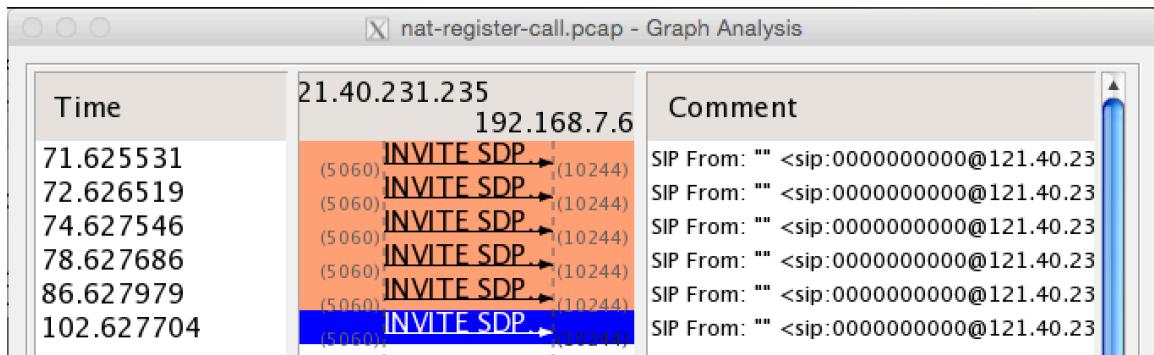


图 7.1: NAT 被叫不可达

我们来看一看客户端向服务器注册的包。如图。客户端向服务器注册时，在 Contact 头域中指定自己的联系地址，后续服务器向客户端发送请求都会发送到该联系地址上。由于此处联系地址是一个位于 NAT 后面的私网地址，服务器主动发送的消息就无法送达。

解决这一问题有两个思路。一是在客户端，通过 Stun 服务获取自己的公网地址，然后在发送注册请求时在 Contact 头域内填入自己的公网地址即可。另一种方法是在服务器端开启 NAT 探测，当服务器端检测到它是一个私网地址 (RFC1918，如 10.x.x.x、192.168.x.x 等) 时，即判定该私网

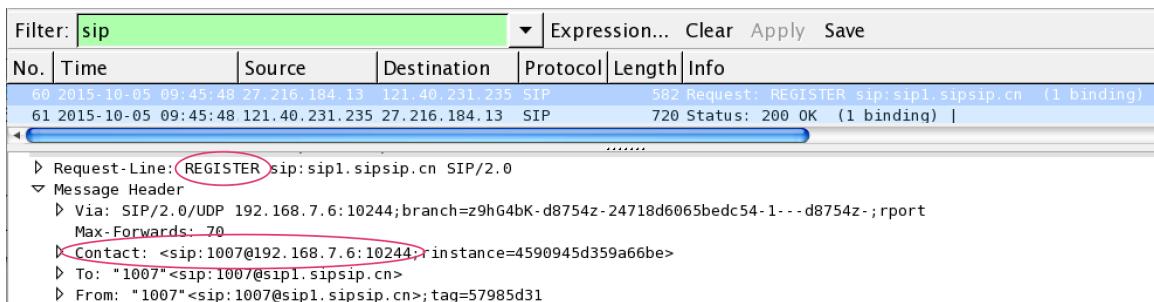


图 7.2: NAT 使用内网地址注册

地址是不可达的，转而使用其公网地址（因为服务器端收到的包的实际来源是 NAT 设备的公网地址）。关于这些解决方案，我们将在后面的章节中再详细说明。

7.2 为什么我活不过 30 秒？

社区中经常会有人问到这样的问题：在 SIP 通话中，为什么我都正常建立通话了，声音也正常，但电话却在 30 秒钟左右自动断了呢？

看来是个很悲剧的故事，好不容易把电话弄通了，却不能早畅聊。

其实这个问题笔者遇到的多了，但“授人以鱼不如授人以渔”，笔者一般都会告诉他们，抓个包看看。等到提问者抓了包，还是不得要领的时候，再给他们提示。

不过遗憾的是，笔者没有把他们抓的包保存下来。因此，为了阐述这一问题，专门搭建了一个模拟环境。服务器端使用 FreeSWITCH，搭在公网上；客户端使用 Bria，在 NAT 后面，私网地址是 192.168.7.6。从服务器向客户端发起一路通话，在客户端抓包如下图。

从图中却实可以看出，客户端在 18 秒处接听了电话，向服务器发送 200 OK，但在 48 秒的时候自动发送了 BYE 请求，挂断了电话。

我们说：要想查电话为什么断，就要查到底是谁发了 BYE。我们点击 BYE 消息查看详细信息，结果我们发现了一个 Reason 字段，内容是“ACK not received”。至此问题已经很明确了，Bria 超过 30 秒没有收到 ACK 证实消息，就挂断了电话。而且，Bria 在这 30s 内，不断地重发 200 OK 消息，这是 UDP 通信的一个特点，在不确定自己发的包是否正确到达对方的情况下，在应用层预防性地重发。

当然，Reason 是一个可选的头域，并不是所有的 SIP 终端都会有该头域，如果在你抓到的 BYE 消息里没有该消息的话，那多半是得靠经验了。不过，如果你读到这里，你已经很有经验了。经验就是——如果电话正常通话 30 秒断，那十之八九就是 NAT 环境下 ACK 的问题。

不过，服务器为什么没有给我们发 ACK 呢？解铃还须系铃人，我们得找服务器要说法。

为了跟服务器对质，解决这样的问题一定要在客户端和服务器端同时抓包。我们从服务端的包中看到呼叫流程如下图。图中左侧是服务端，右侧是客户端。

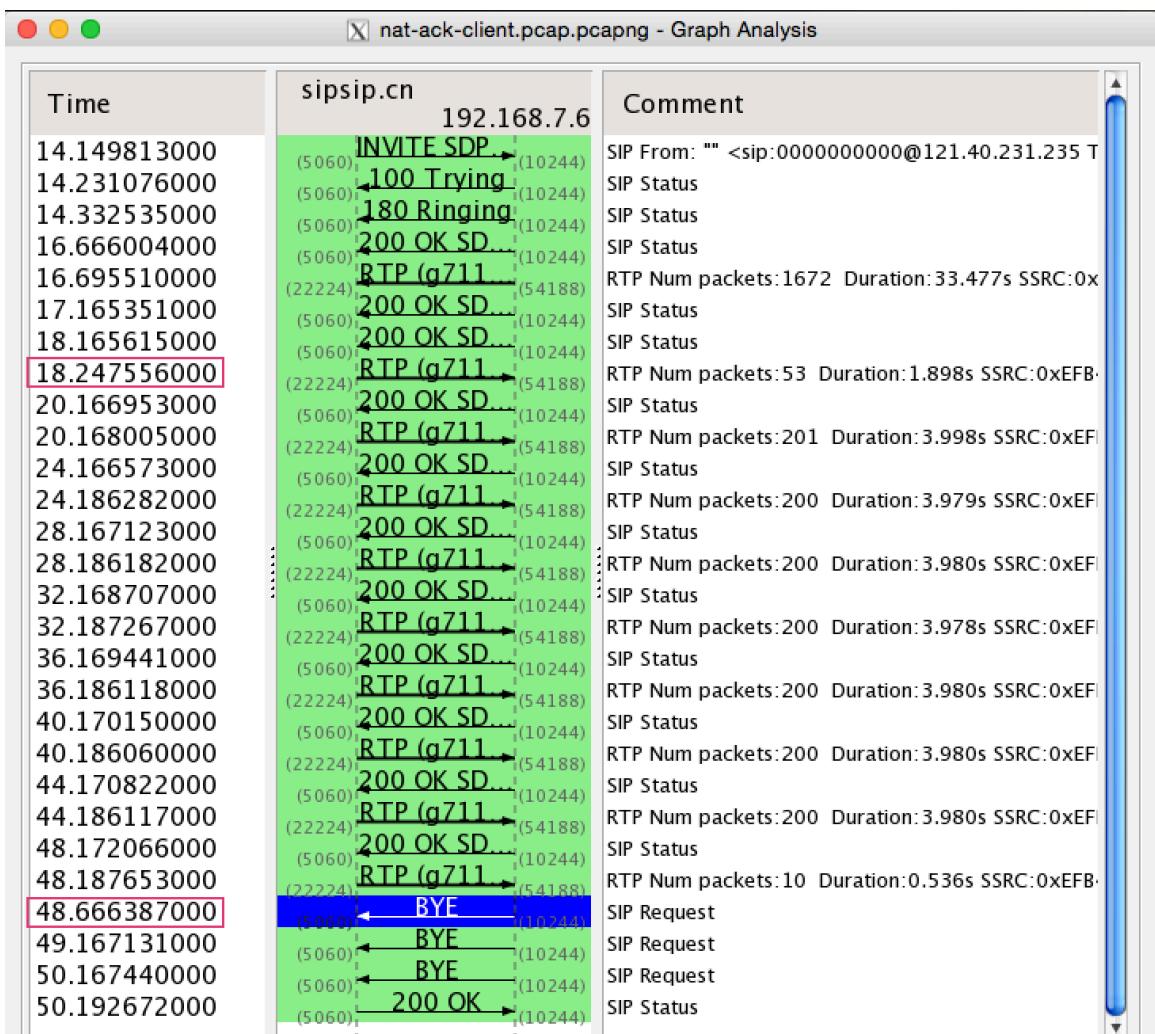


图 7.3: 30 秒断

```

    ▼ Session Initiation Protocol (BYE)
      ▷ Request-Line: BYE sip:mod_sofia@121.40.231.235:5060 SIP/2.0
    ▼ Message Header
      ▷ Via: SIP/2.0/UDP 192.168.7.6:10244;branch=z9hG4bK-d8754z-3984210f39bd7b6f-1---d8754z-;rport
        Max-Forwards: 70
      ▷ Contact: <sip:1007@192.168.7.6:10244>
      ▷ To: <sip:0000000000@121.40.231.235>;tag=XjZ39rQ6ZH05H
      ▷ From: <sip:1007@27.216.184.13:10244>;tag=53c86a02
        Call-ID: 0e32edf5-e5ab-1233-f885-00163e004c12
      ▷ CSeq: 2 BYE
        User-Agent: Bria 3 release 3.5.5 stamp 71243
      ▷ Reason: SIP;description="ACK not received"
        Content-Length: 0
  
```

图 7.4: Bye 消息中的原因值

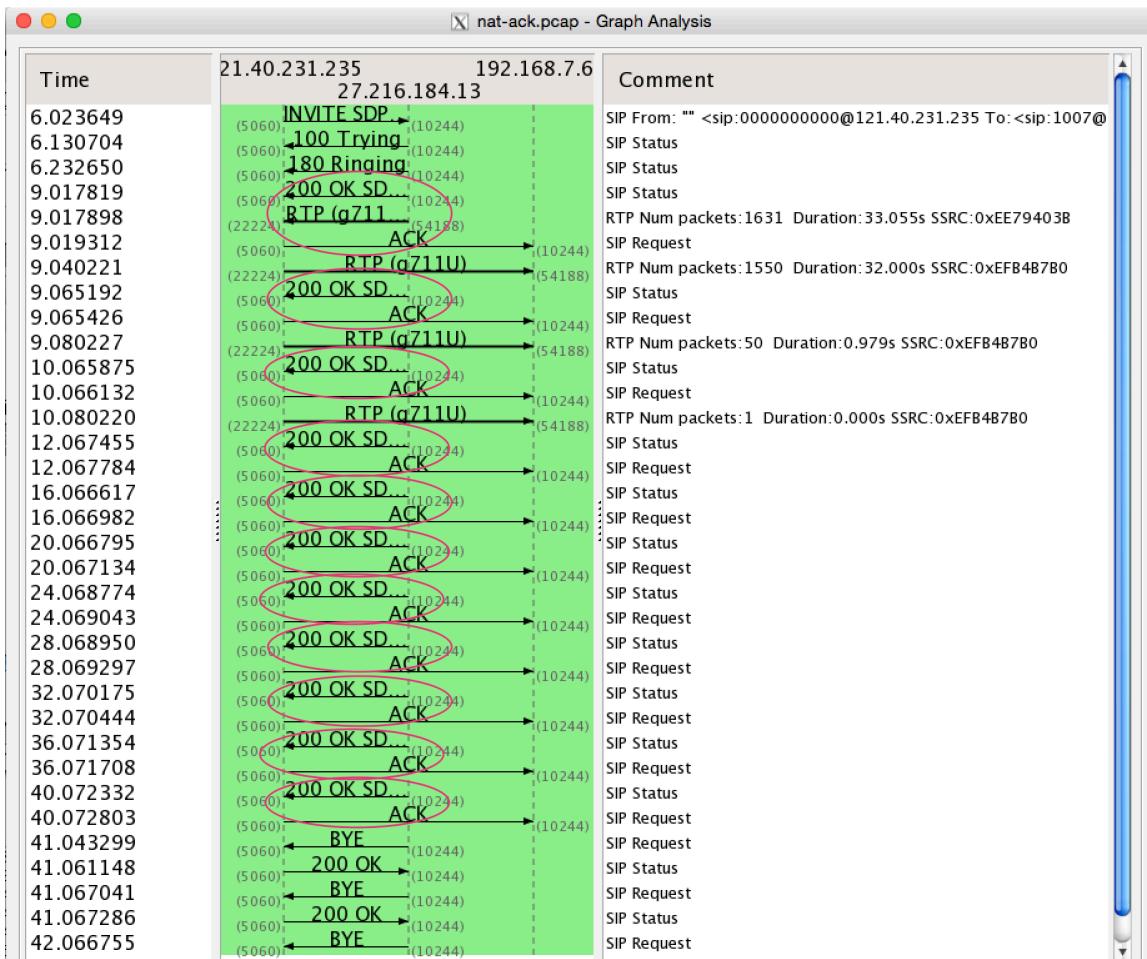


图 7.5: 服务端的呼叫流程

从图中看，我们有点冤枉服务端了，服务端确实针对每一个 200 OK 都发送了 ACK 消息，只是……有点不正常，ACK 为什么发到了客户端的私网地址上？

这，还得看规定。RFC 规定 ACK 消息要发到 200 OK 消息的 Contact 指定的地址上。我们选中一条 200 OK 消息，展开看到详细信息如下图。图中，Contact 头域指定的地址就是客户端的私网地址。

No.	Time	Source	Destination	Protocol	Length	Info
16	2015-10-05 10:24:35	27.216.184.13	121.40.231.235	SIP/SDP	768	Status: 200 OK
↓ Status-Line: SIP/2.0 200 OK						
↳ Message Header						
↳ Via: SIP/2.0/UDP 121.40.231.235;rport=5060;branch=z9hG4bKZagS3NDc7Nvc						
↳ Contact: <sip:1007@192.168.7.6:10244>						
↳ To: <sip:1007@27.216.184.13:10244>;tag=53c86a02						
↳ From: <sip:0000000000@121.40.231.235>;tag=XjZ39rQ6ZH05H						

图 7.6: 200 OK 消息中的 Contact

解决的方法无外乎两种，在客户端开启 STUN 或在服务端启用 NAT 探测。呃……我记得你上一节是不是说过了啊？

最后，对比图 X 和图 Y，细心的读者可能还会发现客户端发送了 3 个 BYE，仅收到一个 200 OK 响应。这个问题就不用纠结了，UDP 本来就是不可靠的，这也是为什么客户端要预防性的重复发 BYE 的原因。

7.3 通过实例深入理解 NAT

为了便于理解 NAT 对 SIP 通话的影响，我们一起来做几个测试。

首先，软交换服务器位于公网上，IP 地址是 121.40.231.235。客户端是 Bria，位于 NAT 后面，IP 地址是 192.168.7.6，对应的公网地址是 27.216.184.13。

我们在服务器端抓包，从客户 1007 呼叫服务器 test，共做了三次测试，抓包用 Wireshark 分析可以看到三次通话，如下图：

nat-server.pcap - VoIP Calls									
Detected 3 VoIP Calls. Selected 0 Calls.									
Start Tim	Stop Tim	Initial Speal	From	To	Protocol	Packet	State	Comm	
10.196497	16.192922	27.216.184.13 "1007"<sip:1007@sip1.sips<sip:test@sip1.SIP				6	COMPLETE		
78.576958	85.234422	27.216.184.13 "1007"<sip:1007@sip1.sips<sip:test@sip1.SIP				8	COMPLETE		
36.264044	42.269888	27.216.184.13 "1007"<sip:1007@sip1.sips<sip:test@sip1.SIP				10	COMPLETE		

图 7.7: Wireshark 中显示有三次通话

7.3.1 默认配置

Bria 客户端采用默认配置，我们点击“Flow”按钮可以看到如的流程图。

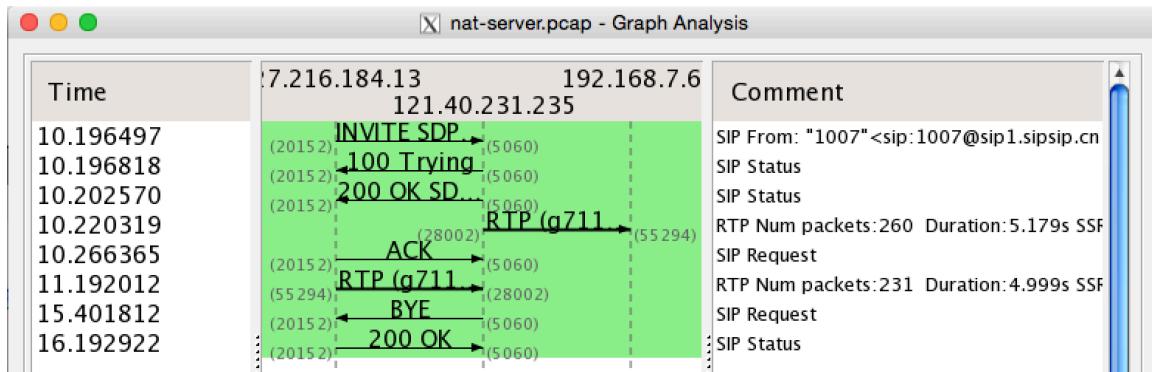


图 7.8: 默认配置

从图中可以看出，由于该数据包是在服务器端的抓包，服务器看到的第一个 INVITE 请求来自 Bria 的公网地址 27.216.184.13，后续的所有 SIP 消息都是服务器跟这个地址交互的。唯独有一部分 RTP 流直接发往了 Bria 的私网地址 192.168.7.6。当然，这些 RTP 流是无法真正到达客户端的，所有的包由于在服务器端无法路由而被丢弃。那么，为什么服务器会往 Bria 的私网地址发消息呢？

点击流程图上的 INVITE 消息，然后在主窗口中展开，我们可以看到其中的 SDP 消息。如图。从图中我们可以看到，SDP 中对媒体的描述全部都是 Bria 客户端的私网地址。也难怪，Bria 其实不知道它位于 NAT 后面，所以，只好填上自己所在主机的 IP 地址。

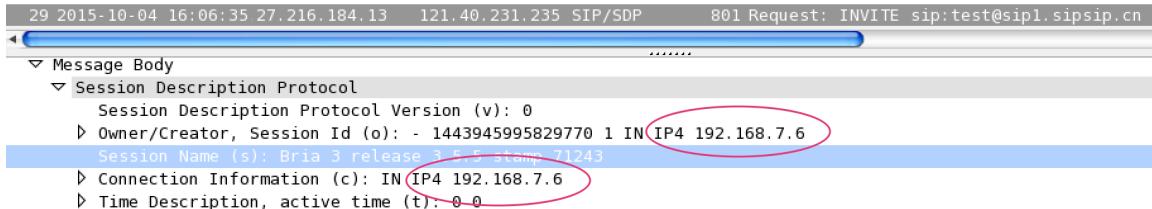


图 7.9: INVITE 中的 SDP

所以，从上图中看我们是无法收到服务器端的 RTP 的，因而无法听到声音。但事实上，在测试中我们听到了声音，为什么呢？继续看。

点击主菜单“Telephony”->“RTP”->“Show All Streams”可以看到如下窗口，该窗口中列出了很多路 RTP 流，其中最上面两路的目标地址 (Dest) 分别是 192.168.7.6 和 27.216.184.13，这两路 RTP 流的源 IP、端口以及 SSRC 都是相同的，因而我们可以判断是同一路 RTP 流。

选中这两路 RTP 流并点击“Prepare Filter”按钮，Wireshark 就会准备好一个过滤器，我们可以在主窗口中看到过滤器的内容是“(ip.src==121.40.231.235 && udp.srcport==28002 && ip.dst==192.168.7.6 && udp.dstport==55294 && rtp.ssrc==0xEF3B65B) || (ip.src==121.40.231.235 && udp.srcport==28002 && ip.dst==27.216.184.13 &&

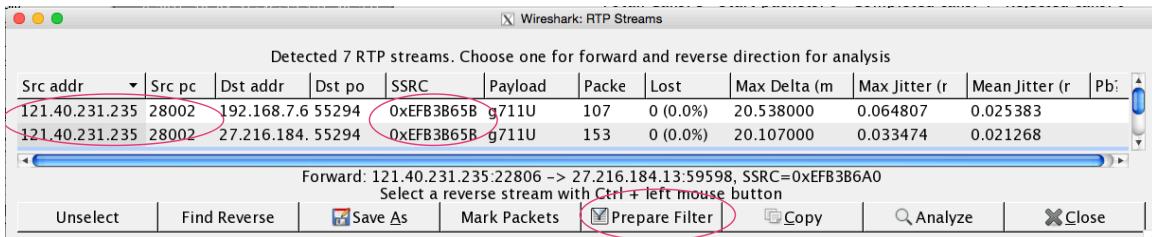


图 7.10: INVITE 中的 SDP

`udp.dstport==55294 && rtp.ssrc==0xEF83B65B”`。点击主窗口上过滤器这边的“Apply”按钮就可以把这两路 RTP 流过滤出来。如图。

Filter: <code>dstport==55294 && rtp.ssrc==0xEF83B65B</code>							Expression...	Clear	Apply	Save
No.	Time	Source	Destination	Protocol	Length	Info				
226	2015-10-04 16:06:37	121.40.231.235	192.168.7.6	RTP	216	PT=ITU-T G.711 PCMU, SSRC=0xEF83B65B, Seq=21054, Time=16				
228	2015-10-04 16:06:37	121.40.231.235	192.168.7.6	RTP	216	PT=ITU-T G.711 PCMU, SSRC=0xEF83B65B, Seq=21055, Time=17				
230	2015-10-04 16:06:38	121.40.231.235	27.216.184.13	RTP	216	PT=ITU-T G.711 PCMU, SSRC=0xEF83B65B, Seq=21056, Time=17				
232	2015-10-04 16:06:38	121.40.231.235	27.216.184.13	RTP	216	PT=ITU-T G.711 PCMU, SSRC=0xEF83B65B, Seq=21057, Time=17				
235	2015-10-04 16:06:38	121.40.231.235	27.216.184.13	RTP	216	PT=ITU-T G.711 PCMU, SSRC=0xEF83B65B, Seq=21058, Time=17				

图 7.11: INVITE 中的 SDP

在图中我们可以发现，从 Seq 为 20156 的包开始，服务器将 RTP 包发往了 Bria 客户端的公网地址。Bria 端的 NAT 设备会把从公网地址上收到的 RTP 消息转发到 Bria 所在的主机（192.168.7.6）上去，我们便听到了声音。

所以，实际上私密在于服务器。回头看图 X。在服务器端向 192.168.7.6 发送 RTP 的同时，服务器也会收到来自 Bria 公网地址的 RTP 包。服务器端软交换软件具有“学习”功能，它通过一系列的判断发现 Bria 可能位于 NAT 后面（因为它没有收到来自 192.168.7.6 的包反而收到了来自 27.216.184.13 的包），转而开始将 RTP 流发往 27.216.184.13，以便“穿越”NAT 设备，正常通信。

此处服务端用的软件是 FreeSWITCH，该功能默认是开启的，以便帮助大部分客户端设备穿越 NAT。当然，该功能由于违反了 SDP 协议（它应该严格按照 SDP 中规定的地址发包，不应该发给其它人），有点小小的安全性问题。不过，在实际应用中问题不大（除非遇到了精心伪造的一系列 RTP 包，而这是比较困难的）。但如果在对通信保密非常严格的场合，就需要关掉这一功能，用下面提到的两种方式了。

7.3.2 使用 Stun 服务器

第二次测试的截图如下。

从图中可以看到，这次由于我们在 Bria 客户端上开启了 Stun，Bria 通过 Stun 服务获取到了自己的外网 IP 地址和端口号，因而在 SDP 中直接填入了自己的外网地址和端口，就不需要依赖服务器的“学习”功能也能轻松穿越 NAT 了。

在 Bria 上开启 Stun 的配置界面如下。注意，需要有一台运行于公网上的 Stun 服务器。

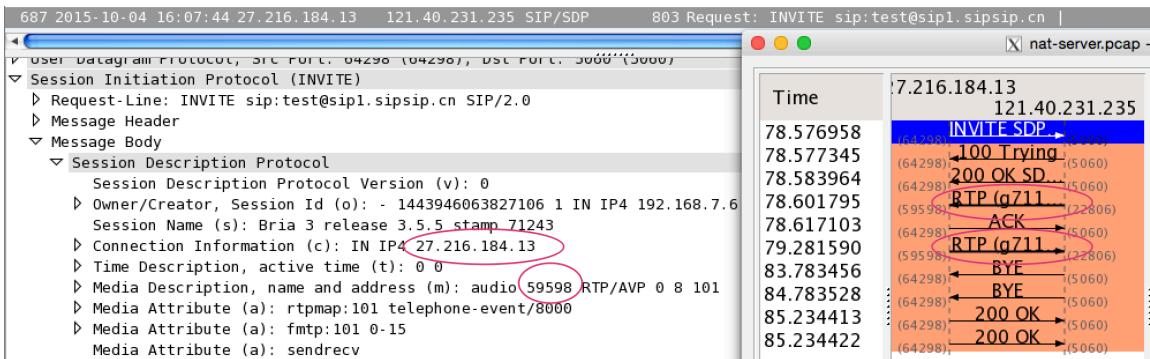


图 7.12: Stun

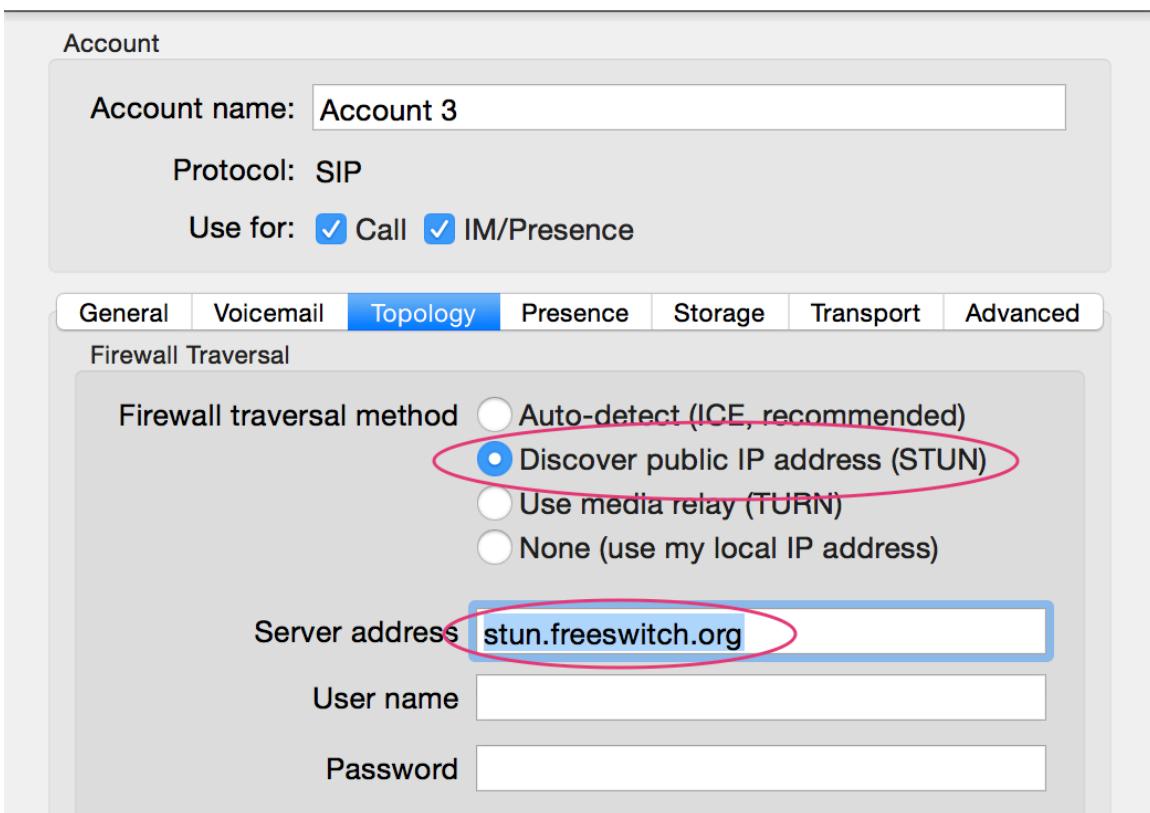


图 7.13: Stun

那么 Stun 是如何工作的呢?

如下图所示。23 号包，客户端向 Stun 服务器发送一个绑定请求 (Binding Request)，服务端回复了一个成功映射的 IP 地址和端口号对。注意，此处的 Stun 请求与上面的例子没有关联，因而端口号不一致。主要是因为我们在做上面的例子时忘了在客户端抓包。另外，至于为什么我们使用 `stun.freeswitch.org` 却出现了 `stun.bitleap.net`，是因为它只是一个 DNS CNAME (第 22 号包)。

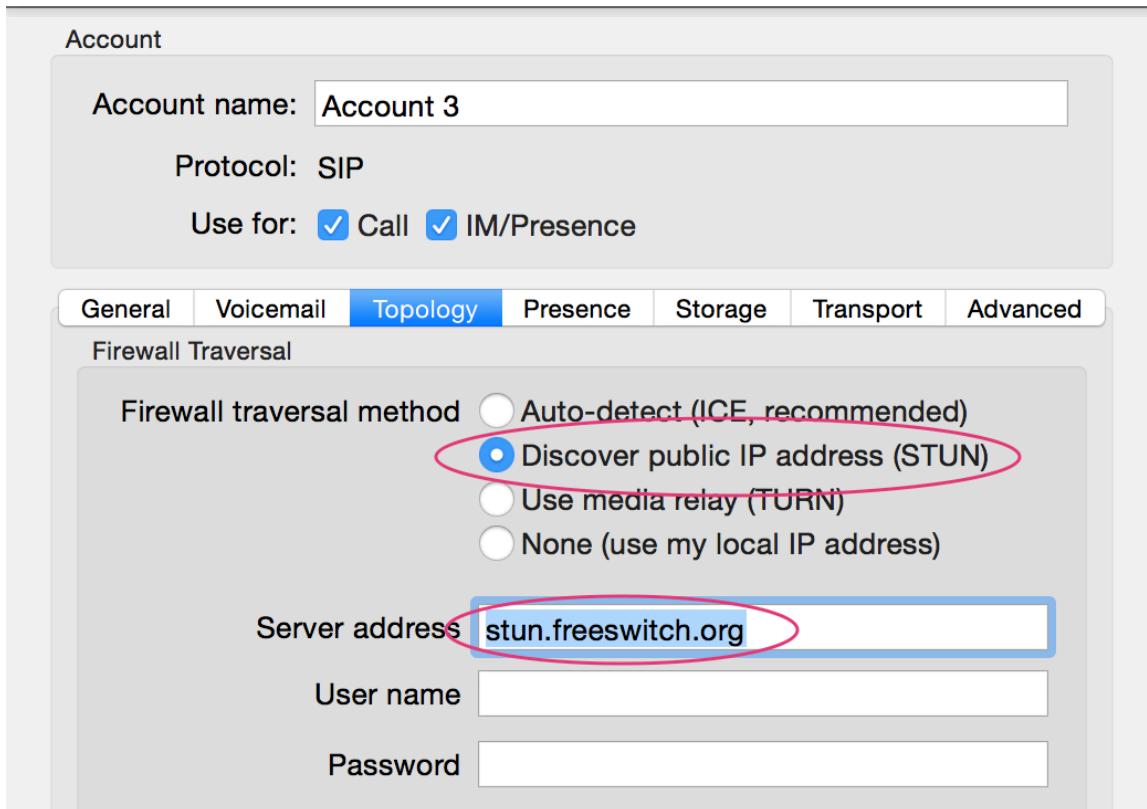


图 7.14: Stun

所以，其实 Stun 的原理很简单。NAT 后面的主机向公网上的 Stun 服务器发送一个 UDP 消息，就在 NAT 设备上钻出一个“洞”，并且，Stun 服务器会告诉客户端这个洞对应的 IP 地址和端口号。客户端再把这个洞的地址告诉别人，别人也可以从该洞中钻进来，上面的例子中的 RTP 包就是这样钻进来的。

当然，NAT 有很多种类型，并不是所有类型的 NAT 都让你乱钻洞，也并不是所有的洞都是对所有人开放的。如，你在对称型 NAT 的设备上利用 Stun 服务钻了个洞，当你再向真正的软交换服务器发 RTP (UDP) 包时，它会再给你生成一个新洞，因而你从 Stun 服务器上获取的洞的映射地址是不准确的，因而软交换也无法利用这个洞。Stun 服务不能解决对称型 NAT 的穿越问题。

7.3.3 使用 ICE

ICE (Interactive Connectivity Establishment) 是 NAT 穿越的终极解决方案。ICE 其实不是一种技术，而是多种现有技术的组合，其中就包括 Stun。别急，我们先看图再说话。

第三次实验的截图如下。从图中可以看出，Bria 客户端发出的 SDP 消息中不仅 Connection Information 使用了从 Stun 服务器获得的公网 IP 地址，还多出来两个候选地址（Candidate）。这两个候选地址包括自己的私网地址和公网地址。

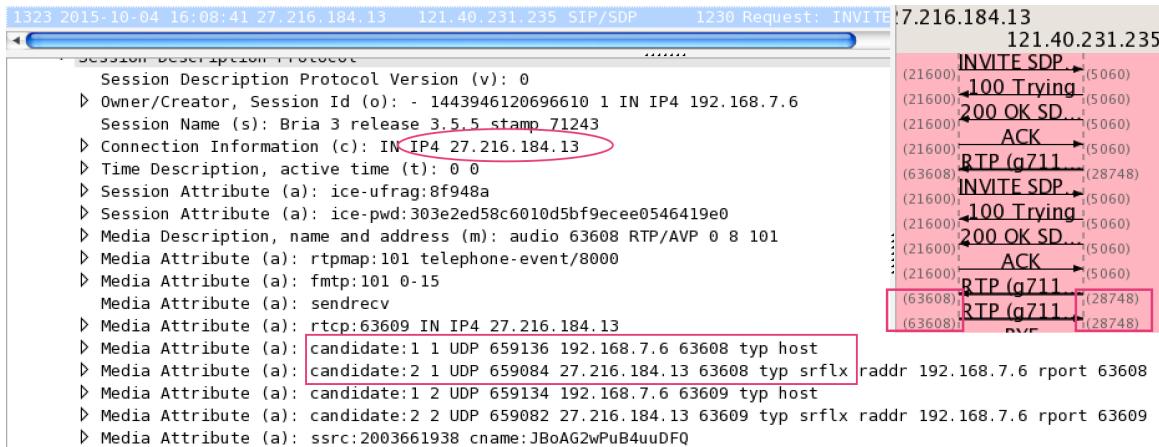


图 7.15: ICE

通过在 ICE 中引入 Candidate，通信的双方就可在以 SDP 中向对方提供多个 IP 地址和端口对，这样，双方在互发 RTP 时就可以从这些候选地址中一个一个地试，直到能够试通。通过这种机制，理论上也可以解决最短路径问题——如果两个相互通信的客户端都在同一个 NAT 子网内，理论上 RTP 或可以点对点地传送，不需要经过服务器中转，从而节省资源。

当然，如果在 ICE 中引入 TURN 服务（一种数据中转服务），ICE 就可以具有更强的穿透能力。比如，很多企业防火墙直接不允许 UDP 包通过，更甚至只允许内网主机访问外网的特定端口如 80、443 等。那么，只要客户端、服务端以入 TURN 服务支持通过 TCP 的 80 端口中转数据，则这种网络也可以穿透。由于笔者没有这样的环境，在此就不举例了。

在这一节的最后，我们再来看一下双方在 Candidate 中的协商过程。从上图中我们已经知道双方的端口号分别是 63608 和 28748，因此，我们以 `udp.port == 28748` 过滤一下。如下图。

图中，1328 号包服务器给客户端的 RTP 端口发送了一个 STUN 请求（但没收到回应，此时不通），同时，客户端也给服务端发送了一个 STUN 请求（1331 号包），服务端立即在 1332 号包回应 Binding Success 消息，这条路（上行）算走通了。接着，1334 号包，服务端重新向客户端发送 STUN 请求，并于 1347 号包收到了客户端的 Binding Success 消息，这条路（下行）也通了。接下来，从 1349 行开始，服务器就正常的给客户端发送 RTP 包了。

注意，此处的 STUN 请求和响应并不是为了获取自己的公网地址在 NAT 设备上钻洞，而是 ICE 重用了 STUN 协议在多个候选地址之间测试连通性的一种手段。

No.	Time	Source	Destination	Protocol	Length	Info
1328	2015-10-04 16:08:41	121.40.231.235	27.216.184.13	STUN	200	Binding Request user: 8f948a;2eHKLdXMMOGhU8si
1331	2015-10-04 16:08:42	27.216.184.13	121.40.231.235	STUN	196	Binding Request user: 2eHKLdXMMOGhU8si:8f948a
1332	2015-10-04 16:08:42	121.40.231.235	27.216.184.13	STUN	108	Binding Success Response XOR-MAPPED-ADDRESS: 27.216.184.13:63608
1333	2015-10-04 16:08:42	121.40.231.235	27.216.184.13	RTP	216	PT=ITU-T G.711 PCMU, SSRC=0xEF83B6D9, Seq=44928, Time=640
1334	2015-10-04 16:08:42	121.40.231.235	27.216.184.13	RTP	216	PT=ITU-T G.711 PCMU, SSRC=0xEF83B6D9, Seq=44929, Time=800
1337	2015-10-04 16:08:42	121.40.231.235	27.216.184.13	RTP	216	PT=ITU-T G.711 PCMU, SSRC=0xEF83B6D9, Seq=44930, Time=960
1338	2015-10-04 16:08:42	121.40.231.235	27.216.184.13	RTP	216	PT=ITU-T G.711 PCMU, SSRC=0xEF83B6D9, Seq=44931, Time=1120
1344	2015-10-04 16:08:42	121.40.231.235	27.216.184.13	STUN	200	Binding Request user: 8f948a;2eHKLdXMMOGhU8si
1347	2015-10-04 16:08:42	27.216.184.13	121.40.231.235	STUN	156	Binding Success Response XOR-MAPPED-ADDRESS: 121.40.231.235:28748
1349	2015-10-04 16:08:42	121.40.231.235	27.216.184.13	RTP	216	PT=ITU-T G.711 PCMU, SSRC=0xEF83B6D9, Seq=44932, Time=1760
1350	2015-10-04 16:08:42	121.40.231.235	27.216.184.13	RTP	216	PT=ITU-T G.711 PCMU, SSRC=0xEF83B6D9, Seq=44933, Time=1920

图 7.16: ICE

7.4 更多实例

我们在一次搭建测试环境时，遇到了一些 NAT 问题，简要记录了一下，就是个很好的例子。

FreeSWITCH 部署在公网上做为软交换服务器，开放 5060 端口。客户端呼入后，先放一段回铃音，然后应答，然后放一段音频（大约 14.5 秒），服务端挂机。

笔者的客户端在 NAT 环境中，IP 地址是 192.168.7.6，对应的外网 IP 地址是 27.213.99.220。客户端软件是 X-Lite 4.7.1 (74250) Mac 版。配置如图 7.17 所示：

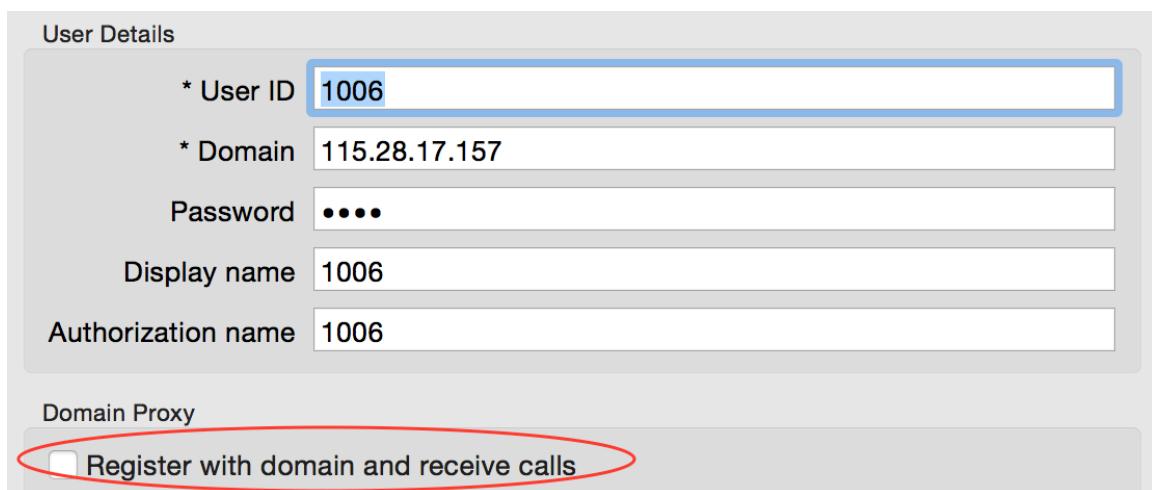


图 7.17: X-Lite 配置

注意，当时笔者并没有使用『Register Domain And Receive Calls』，因为我们这里不讨论注册，在客户端做主叫时就无需向服务器注册。

设置好后在客户端侧开启 Wireshark 抓包，在客户端上呼叫 play，一切正常。经 Wireshark 分析，呼叫流程如图 7.18。从图上可以看出，X-Lite（左）向 FreeSWITCH（右）发送 INVITE 请求，FreeSWITCH 返回 183（带回铃音）以及 200（应答）消息（32.21秒处），应答后 FreeSWITCH 即播放音频，完毕后，FreeSWITCH 发送 BYE 消息通知 X-Lite 挂机（46.89秒处），从应答到挂机，总时长约为 46.89 - 32.21 = 14.5 秒。一切看起来都很正常。

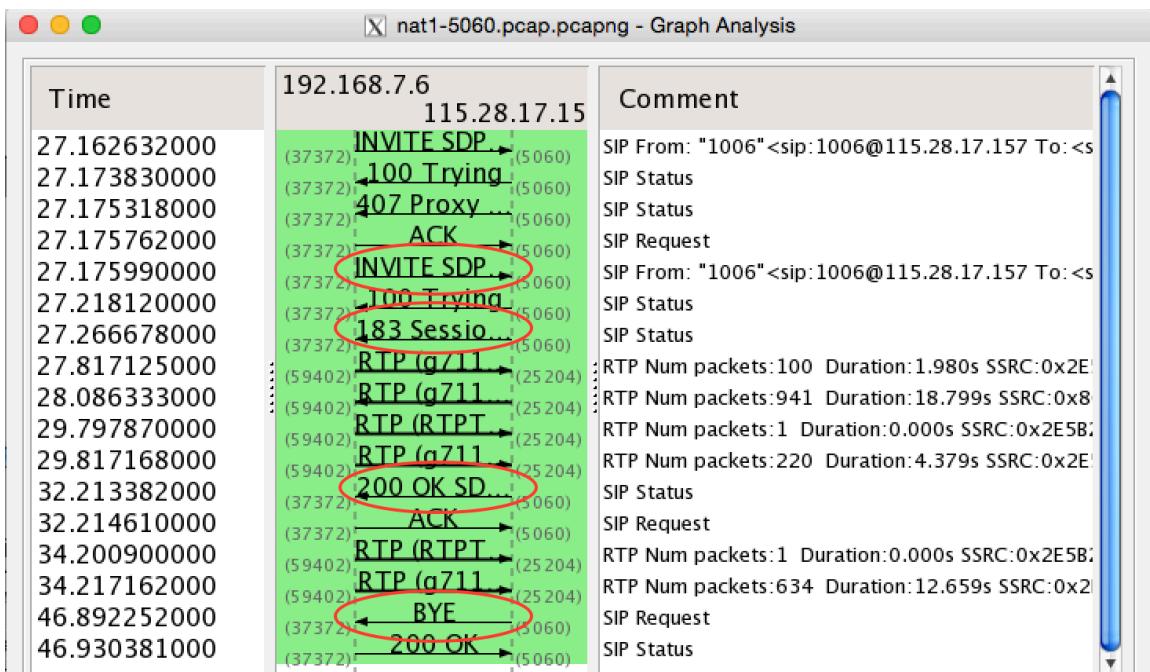


图 7.18: 第一个呼叫流程

后来，笔者在服务器又增加了一个配置，使用 5080 端口，这时候笔者改变 X-Lite 配置中的 Domain，在 IP 地址后增加：5080，即准备把 INVITE 消息发送到 FreeSWITCH 的 5080 端口。

配好后重新呼叫 play，这次还是可以正常听到声音的。不正常的是在声音播放完毕后电话并没有立即挂断。而是等了一会挂断的。而且，与上次的情况不同，从呼叫流程图 7.19 上看，X-Lite 只发送了一个 INVITE 请求，而图 7.18 中是两个。这是由于 FreeSWITCH 的默认配置中 5080 端口是不需要鉴权的，任何人都可以向其发起呼叫，因此这部分也没什么问题。真正与令人费解的是，图 1.3 挂断是由客户端主动发起的，而不是像图 1.2 那样由服务器端发起的。

另外，从图 7.19 中我们也可以看到，客户端连发 4 个 BYE，而服务端却没有反应，客户端主动将呼叫状态变为挂机状态。

不合逻辑，我们期望这次能与第一次得到一样的结果。

由于前半段呼叫流程好像没什么问题，我们从 BYE 消息开始查。

点开 BYE 消息，我们可以看到以下 SIP 头域：

```
Reason: SIP;description="RTP stream closed due to inactivity; mediaType=AUDIO"
```

这个 Reason 头域在 SIP 中是可选的，不过在这里它对我们很有用。字面意思是说，客户端检测到 RTP 不活动了（就是超过一定时间没收到 RTP 流），进而挂断了电话。

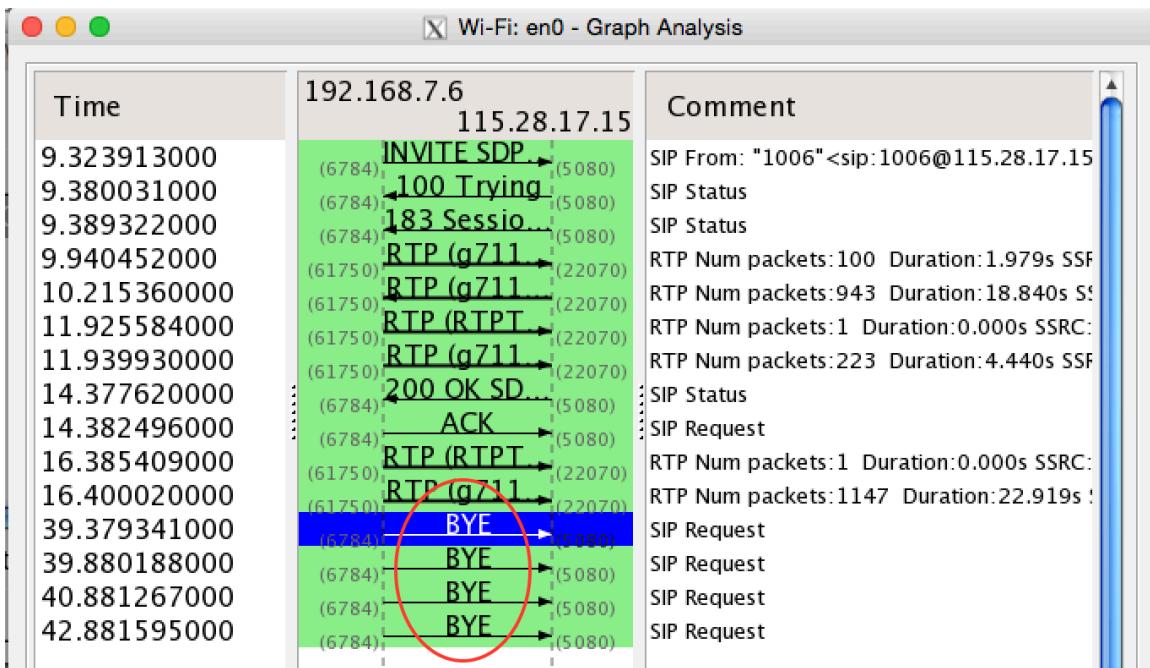


图 7.19: 第二个呼叫流程

问题似乎已经很明确了。服务端在14.37秒处开始放音，至 $14.37 + 14.5 = 29$ 秒时就应该已经播放完毕，因此没有后续的音频，客户端在等待了约10秒后，在39.37秒处发送BYE挂机。

这其实是 SIP 终端的一种保护机制。正常情况下，通话的建立和释放都要靠 SIP 来实现，但在不正常的情况下（如网络中断等原因），SIP 消息可能无法送达，SIP 终端在超过一定时间检测不到 RTP 信息后，认为网络中断或发生了异常，便主动挂断电话。这个超时时间值在大部分终端上是可以设置的，X-Lite 有一个设置如图7.20。把『Enable inactivity timers』前面的勾去掉，就可以禁用这个功能。当然，感兴趣的读者也可以试一下修改后面『RTP Timers』的值看有什么效果。

去掉我们上面说的勾以后，再重新打个电话，Wireshark 显示的流程图还是一样的，只是，X-Lite 不会再自动挂断电话了。

我们再从服务器端找找原因，看能不能发现什么。在服务端抓包，如图7.21（注意，抓包时忽略了 RTP 数据，帮在图中无显示）。

这个图很有意思，即使你稍微有点看图的经验，也可以一眼看出它是不正常的。其中99.220是 X-Lite 的公网地址，115.28.17.15是 FreeSWITCH 的地址，而192.168.7.6是 X-Lite 的私网地址。从图上可以看出。虽然 INVITE 请求是在 X-Lite 的私网地址 7.6 上发出来的，但在服务器端，只能看到来自其公网地址27.213.99.220。前面的交互流程都很正常。4.99秒的时候服务器发送 200 OK 应答，并开始播放音频，约14.5秒后播放完毕，并于19.67秒处向客户端发送 BYE 消息请求挂机，但由于客户端一直没有反应，因此，BYE 消息一直重发。

那么，在这里，为什么前面的流程都正常，而服务端的 BYE 发到192.168.7.6这个私网地址上去了呢？

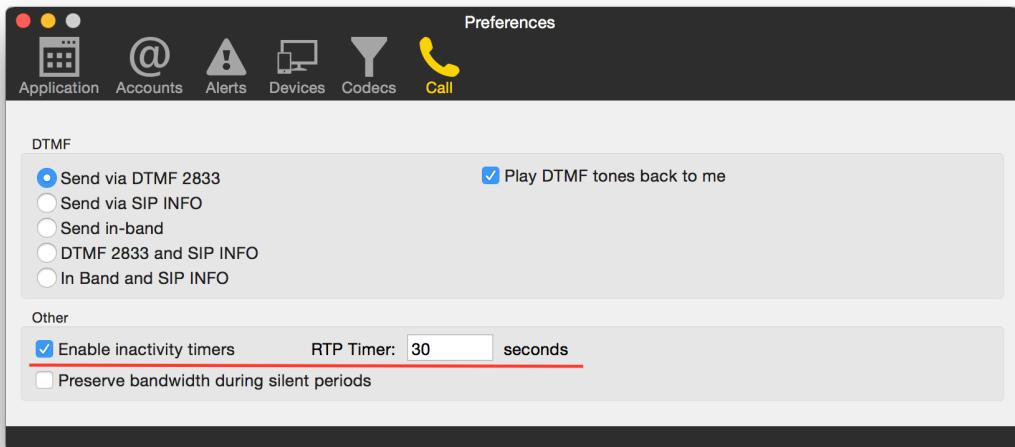


图 7.20: X-Lite RTP timer 设置

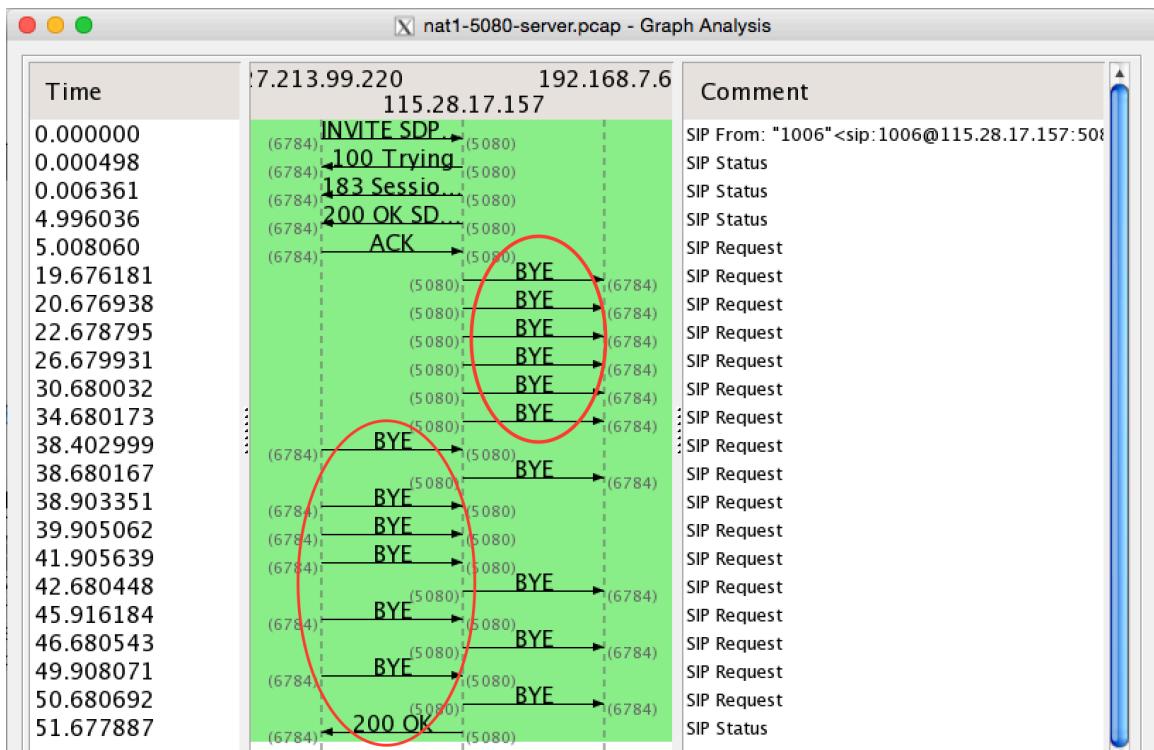


图 7.21: 服务器端的抓包

为了找到答案，我们从 BYE 前面的一个消息入手。点开 BYE 前面的 ACK 消息（5.00秒处），可以看到 SIP 头域里有一个 Contact 地址正是192.168.7.6。如图7.22。

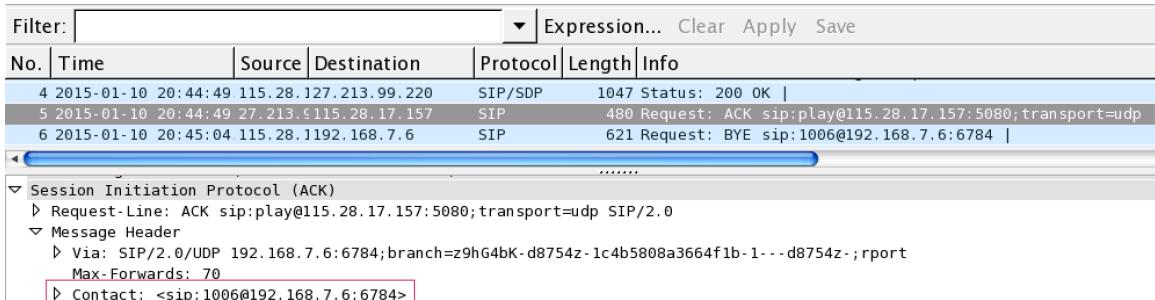


图 7.22: 服务器端的抓包

FreeSWITCH 就是根据这个 Contact 地址发送的 BYE。由于 BYE 消息没有到达真正的目的地，因而对方也没有任何回应。

后来，在第38.40秒处，笔者手工按下了客户端的挂机键（客户端没收到来自服务端的 BYE 消息，还不知道服务端正在挂机呢），因此，客户端主动给服务端发 BYE。但由于这里服务端正忙着处理一个事务（不断的重发 BYE），所以无法给客户端响应，故此时客户端也在重复的发送 BYE。当服务端的事务在50.68秒处超时后（超时时间为58.68 - 19.67 = 31 秒），得响应了客户端的 BYE 请求（51.67秒处）。

找到了问题的所在，就好解决了。解决的办法前面的章节已经说过了，无非是从客户端还是从服务端入手的问题。在本例中，由于我们服务端的 5060 端口是正常的，而 5080 端口不正常，因此，我们对比了服务端的配置，修改了一个 NAT 相关的参数，服务端就能自动探测到客户端位于 NAT 之后，因而以后再发送 BYE 时目的地便不使用 ACK 中的 Contact 地址，而使用探测到的客户端的公网地址，问题就解决了。

7.5 UPnP

前面我们说了，如果要在客户端解决 NAT 穿越问题，就需要 STUN 服务器的支持。那么，如果没有 STUN 服务器，或者说我的客户端不支持 STUN，那还有其它的解决办法吗？

答案是肯定的，人的智慧就在于可以发明无数不同的方法解决同一个问题。处于 NAT 内网的机器要想知道自己的外网 IP 地址，可以直接问路由器啊。

问？怎么问？当然需要有一种协议。这种协议就叫做 UPnP。笔者使用的是是一款 TP Link 路由器，点击“转发规则”->“UPnP 设置”并开启 UPnP 服务后，可以看到如下界面。笔者在内网开启了 FreeSWITCH，FreeSWITCH 通过 UPnP 把自己的私网地址和端口映射到了路由器的外部端口上。

它是怎么实现的呢？好办，在客户端上启用 Wireshark 抓包，然后重新启动 FreeSWITCH，就可以抓到 UPnP 的协商过程。UPnP 使用 HTTP 协议，因此直接在 Wireshark 上用 http 过滤器，就可以看到 UPnP 的内容。如下图。



图 7.23: 路由器 UPnP 设置

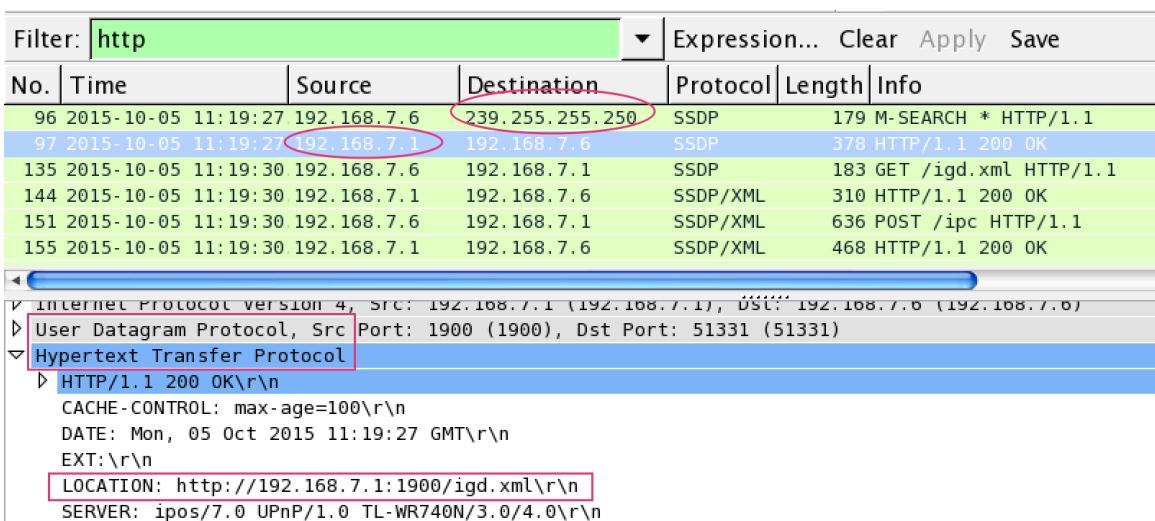


图 7.24: UPnP Discover 过程

96 号包，客户端（FreeSWITCH）在启动时，首先向一个组播地址239.255.255.250使用 UDP 协议发送一个 M-Search 的 HTTP 请求。不错，这是一个 HTTP 请求，但谁说 HTTP 请求不可以走在 UDP 上呢。

由于请求发组播地址，局域网内所有主机都可以收到这条请求消息。但只有我们的路由器做出了回应。97 号包，路由器（192.168.7.1）说：“啊，想找 UPnP 服务啊，我有啊，访问这个地址……”。接着，它便在 HTTP 响应消息中 Location 头域中回应了一个 HTTP 地址。这次，也是使用的 UDP 协议。

在获得 UPnP 服务地址后，135 号包，客户端就开始用 HTTP 请求查询服务器相关的信息了。从此，协议也都开始用 TCP 了。如客户端通过 HTTP POST 请求发送一个 SOAP 消息请求获取路由器的公网 IP（GetExternalIPAddress），并得到一个 NewExternalIPAddress 响应。如图。

Filter: http							Expression... Clear Apply Save
No.	Time	Source	Destination	Protocol	Length	Info	
96	2015-10-05 11:19:27	192.168.7.6	239.255.255.250	SSDP	179	M-SEARCH * HTTP/1.1	
97	2015-10-05 11:19:27	192.168.7.1	192.168.7.6	SSDP	378	HTTP/1.1 200 OK	
135	2015-10-05 11:19:30	192.168.7.6	192.168.7.1	SSDP	183	GET /igd.xml HTTP/1.1	
144	2015-10-05 11:19:30	192.168.7.1	192.168.7.6	SSDP/XML	310	HTTP/1.1 200 OK	
151	2015-10-05 11:19:30	192.168.7.6	192.168.7.1	SSDP/XML	636	POST /ipc HTTP/1.1	
155	2015-10-05 11:19:30	192.168.7.1	192.168.7.6	SSDP/XML	468	HTTP/1.1 200 OK	

Filter: http							Expression... Clear Apply Save
No.	Time	Source	Destination	Protocol	Length	Info	
96	2015-10-05 11:19:27	192.168.7.6	239.255.255.250	SSDP	179	M-SEARCH * HTTP/1.1	
97	2015-10-05 11:19:27	192.168.7.1	192.168.7.6	SSDP	378	HTTP/1.1 200 OK	
135	2015-10-05 11:19:30	192.168.7.6	192.168.7.1	SSDP	183	GET /igd.xml HTTP/1.1	
144	2015-10-05 11:19:30	192.168.7.1	192.168.7.6	SSDP/XML	310	HTTP/1.1 200 OK	
151	2015-10-05 11:19:30	192.168.7.6	192.168.7.1	SSDP/XML	636	POST /ipc HTTP/1.1	
155	2015-10-05 11:19:30	192.168.7.1	192.168.7.6	SSDP/XML	468	HTTP/1.1 200 OK	

Internet Protocol Version 4, Src: 192.168.7.1 (192.168.7.1), Dst: 192.168.7.6 (192.168.7.6)
 User Datagram Protocol, Src Port: 1900 (1900), Dst Port: 51331 (51331)
 Hypertext Transfer Protocol
 HTTP/1.1 200 OK\r\n
 CACHE-CONTROL: max-age=100\r\n
 DATE: Mon, 05 Oct 2015 11:19:27 GMT\r\n
 EXT: \r\n
 LOCATION: http://192.168.7.1:1900/igd.xml\r\n
 SERVER: iPos/7.0 UPnP/1.0 TL-WR740N/3.0/4.0\r\n

Internet Protocol Version 4, Src: 192.168.7.1 (192.168.7.1), Dst: 192.168.7.6 (192.168.7.6)
 User Datagram Protocol, Src Port: 1900 (1900), Dst Port: 51331 (51331)
 Hypertext Transfer Protocol
 HTTP/1.1 200 OK\r\n
 CACHE-CONTROL: max-age=100\r\n
 DATE: Mon, 05 Oct 2015 11:19:27 GMT\r\n
 EXT: \r\n
 LOCATION: http://192.168.7.1:1900/igd.xml\r\n
 SERVER: iPos/7.0 UPnP/1.0 TL-WR740N/3.0/4.0\r\n

继续看下看还可以看到端口映射的包，如

与 UPnP 类似的还有一种协议叫 PMP，由于笔者没有相应的设备，就不举例了。

No.	Time	Source	Destination	Protocol	Length	Info
275	2015-10-05 11:19:36	192.168.7.6	192.168.7.1	SSDP/XML	936	POST /ipc HTTP/1.1
278	2015-10-05 11:19:36	192.168.7.6	192.168.7.1	SSDP/XML	936	POST /ipc HTTP/1.1

```
<u:AddPortMapping
    xmlns:u="urn:schemas-upnp-org:service:WANIPConnection:1">
    <NewRemoteHost>
        </NewRemoteHost>
    <NewExternalPort>
        5060
    </NewExternalPort>
    <NewProtocol>
        UDP
    </NewProtocol>
    <NewInternalPort>
        5060
    </NewInternalPort>
    <NewInternalClient>
        192.168.7.6
    </NewInternalClient>
```

图 7.25: UPnP 获取公网 IP 响应

第八章 不是你的错

笔者学习 FreeSWITCH 的时候曾经遇到个诡异的问题。FreeSWITCH 做为 SIP 服务器运行在公网上，服务器托管在某省运营商的机房里。笔者在北京往服务器上打电话就经常打不通。在跟服务器相同的省内就能通。

在客户端用 Wireshark 抓包看到如下流程通话流程。从图中可以看出，客户端发出呼叫请求后，服务器回复了 502 消息。

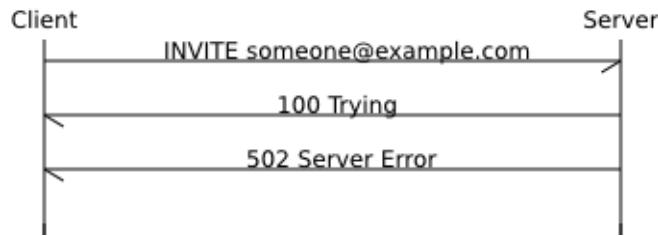


图 8.1: 客户端呼叫流程

当时笔者刚开始学习 SIP，对 SIP 流程也是一知半解。后来慢慢知道正常的通话应该是这样的。



肯定是服务器哪里配错了。但怎么都检查不出问题。服务器端的日志也显得很诡异。在服务器端抓包，看到的呼叫流程是这样的。

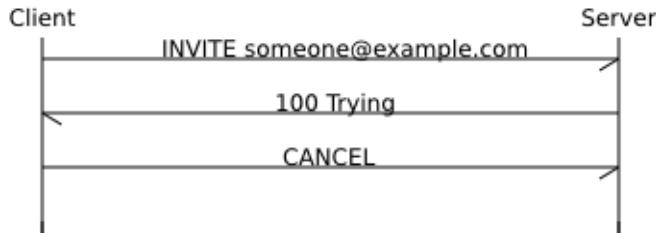


图 8.3: 服务端看到的呼叫流程

咦，服务端根本没有发 502，却收到个 Cancel？又去检查客户端的抓包，确实没看到客户端发送 Cancel，却收到个 502。

百思不得其解。后来经过仔细研究，发现 502 消息中，From 客段中的电话号码少了一个 0。后来才想明白，看来，是中间有人捣鬼。所以，实际的呼叫流程应该是这样的。其中，BAD 便是中间的坏人。

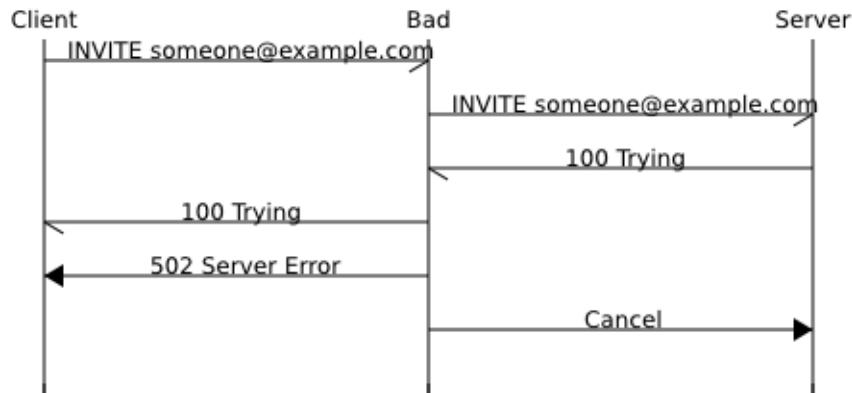


图 8.4: BAD 在中间捣鬼

虽然无法精确定位，但肯定在客户端和服务器中间，有个设备（Bad）伪造了消息，给客户端（主叫）发了 502 假装服务器不通，同时给服务端（被叫）发送 CANCEL 造成客户端取消通话的假象。

后来，跟运营商的人员求证，确实有这样的设备在中间默默地捣鬼。不过，鉴于我搞的是 VoIP，没人愿意透露更多的信息，也没有人能帮我开绿灯。VoIP 怎么了？再说了，当时笔者只是在学习，又不是在搞运营。事实上，坏人总是怯懦的、猥琐的。如果说笔者的数据包不合法，那直接禁止掉就得了，又何必在中间搞这些下三烂的招数。更何况实现的还有 Bug，在伪造的消息中电话号码还少了个 0。

当然，这是几年前的事了。后来笔者也遇到过几次这样的问题，一看少了个 0 就知道是同一种设备在做怪。近几年好像没这个问题了。而且，我们也看到像 Skype、QQ、微信、钉钉之类的业务都具有了实时通话甚至视频的能力，以及随着各种新型的 VoIP 电话的出现，VoIP 大势所趋已经不是谁能阻止的了的了。

由于年代久远，笔者没有找到原先保存下来的 Wireshark 抓包，只好凭记忆用流程图代替了。但无论如何笔者想借这个例子说明，有些情况下一定要双方同时抓包比对，才能找到中间的坏人。

最后，有时候达不到目的也不要太自责，因为，这根本不是你的错。

第九章 暗无天日的日子

几年前笔者应邀到国外帮客户解决一个问题。由于客户还是跨国运营所以跟地球另一侧的队友同步解决问题是很正常的事情，为此我们基本上都是晚上干活。那几天简直就是暗无天日的工作。不过，好在，问题基本都找出来并解决了。现在，挑几个有代表性的讲一下。

9.1 多人电话会议

客户提出的首先一个很严重的问题就是多人会议问题。在多人电话会议中，经常会听不到声音。

客户使用 FreeSWITCH 做为会议服务器，由于不确定是 FreeSWITCH 本身的还是网络的问题，客户感到很迷茫，客户的客户更是感觉到用起很不爽。

笔者一到场给出的解决方案就是抓包，跟录音做比对。但是客户是直接线上运营的系统，数据量非常大，而且，由于系统整个是一个很大的服务器集群，在各各点上都要抓包就更加困难。更让人抓狂的是系统时好时坏，指不定你在抓包时，就没有问题，不抓时反而有问题了。我们现场测了很多遍都未能重现问题，倒是有些客户很配合做测试，最好几个客户经理配合总算重现了一把，得到些零零散散的抓包和录音。

说实话，光把这些零零散散的抓包和录音对起来就费了我一两天的时间。客户的客户催客户，客户也在催我，真是压力山大。但压力再大也得慢慢来。我根据前两天的整理把我掌握的信息画了一个图。

图中的字母都是跟抓包文件名的首字母，该图描述了在一个会议里面各个成员的发言状态。由于抓包和录音很不完整，所以看起来就是图中的样子。

不过，有了这个图，至少笔者能够知道每个包里的语音发生的相对时间，以便查找问题。

从其中一个 MP3 录音开始，如下图。用 Audacity 打开，发现在 1:15 到 1:45 之间，是一段空白。接着就听到有人说：“听不见了……”。

笔者有幸找到了它对应的抓包，在 Wireshark 中打开，找到对应的 RTP，点击“Analyze”并点击“Player”后，可以看到如图所示的波形。

由于这段语音使用了 PCMU 编码，所以可以直接在 Wireshark 中播放。通过听取录音，我们发现这段时间内确实没有人在说话，似乎有人在敲键盘。不过，由于敲键盘的声音很小，几乎听不到。

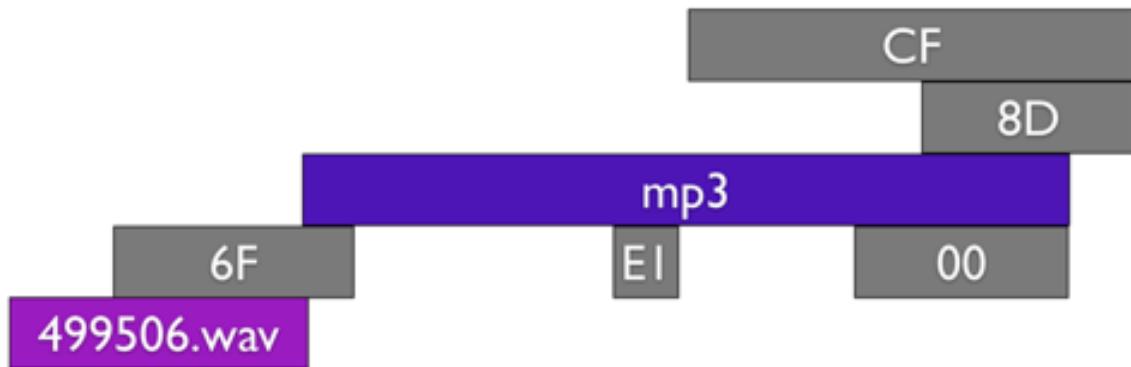


图 9.1: 会议成员抓包和录音位置图

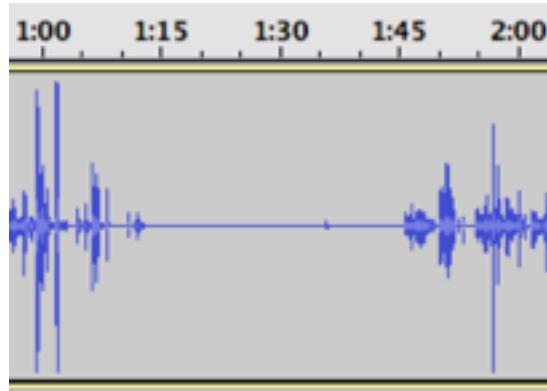


图 9.2: MP3

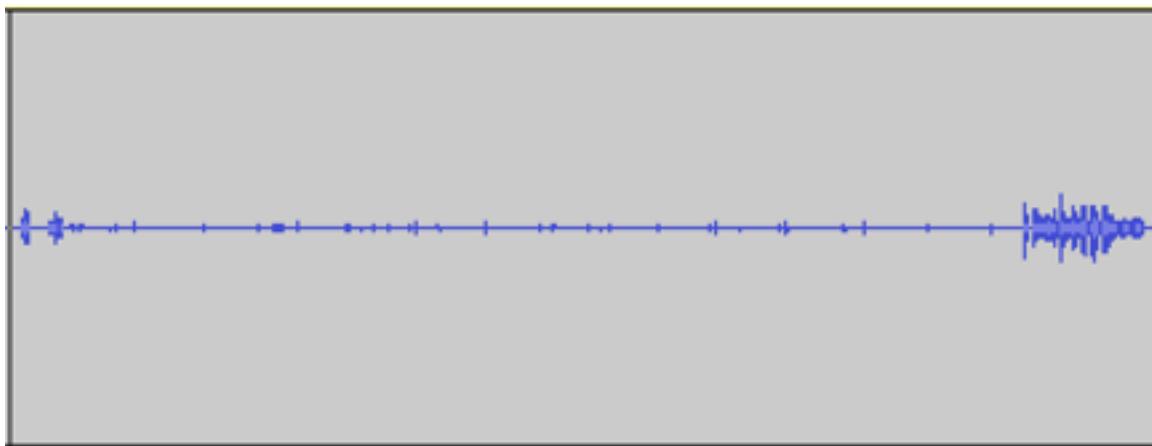
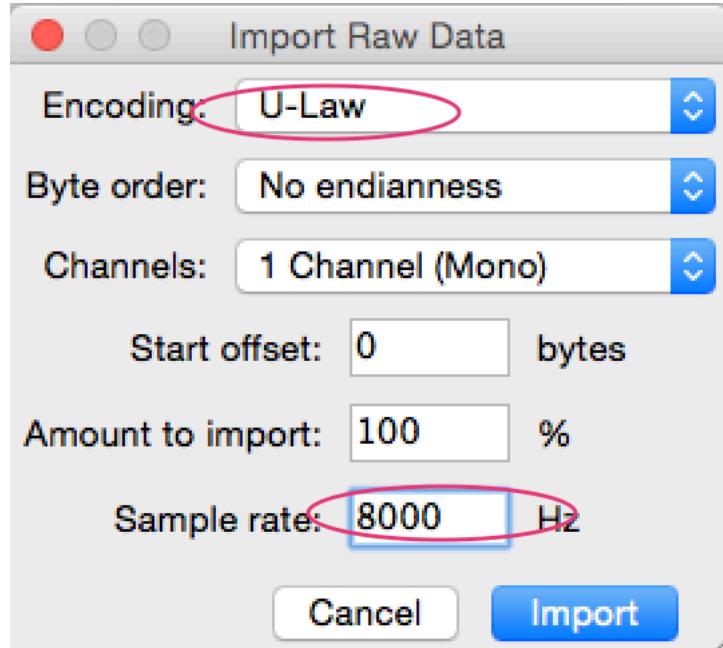
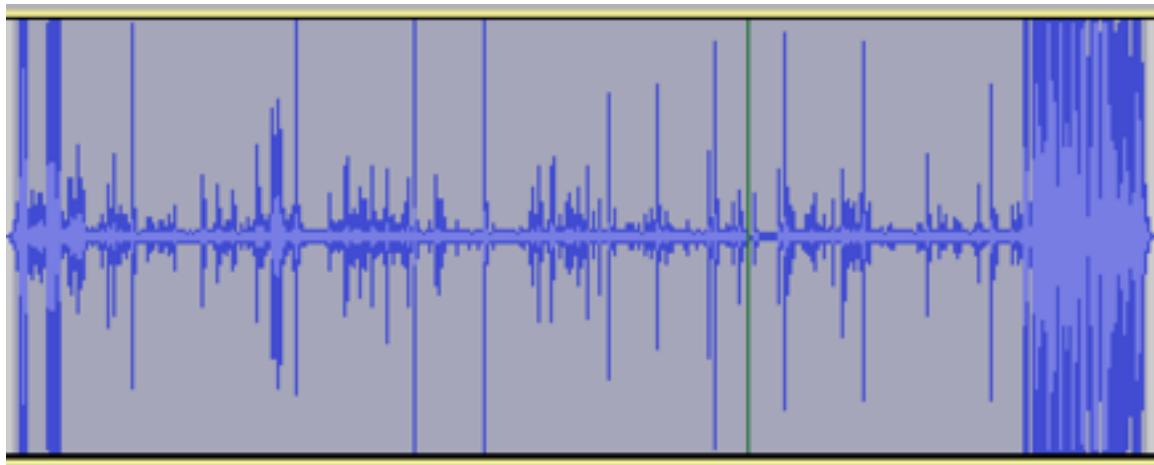


图 9.3: Wireshark 中显示的波形

为了听一听里面究竟有什么声音，笔者在 Wireshark 的“RTP Stream Analyser”窗口中点击 Save Payload 把这段语音存成文件 (RAW 类型)。再打开 Audacity，依次点击“File”->“Import”->“Raw Data”，选择 Encoding 类型为 U-Law (即 率，PCMU)，Sample Rate 为 8000，其它都采用默认值，最后点击 Import 完成导入。如图



选中我们想听的录音部分，再点击“Effect”->“Amplify”将信号放大 30dB，我们可以看到如图所示的波形。从中可以清楚的听到键盘的敲击声。



所以，我们分析，在该会议中，至少这部分没有任何问题，只是键盘的敲击声由于声音太小以至于在会议中它被忽略了，所以与会人员听不到任何声音，以为会议服务器本身有问题。

注：在 FreeSWITCH 的会议中，有一个能量 (Energy) 参数可以调整参会方的声音阈值。为了保证会议中的人数比较多的情况下会场声音不至于太乱，因此，低于一定阈值的声音会被丢掉，这也算是简单的降噪技术。

分析了半天，最后确定没有问题，难道是客户错了？我们还得继续分析才行。后面，我们又找到了一些类似的场景，但通过对录音和抓包的对比，问题都是类似的原因，声音能量值太小而未进入会议混音，以致于让参会者觉得会有声音中断的现象。

除此之外，客户还抱怨会议里经常莫名其妙有些细微的噪音（卡卡声）。我们经过对 RTP 的分析，发现这些噪音是由用户的线路传过来的，而不是会议系统产生的。至于用户的线路是如何产生了这些噪音，那就不得而知了。如图。

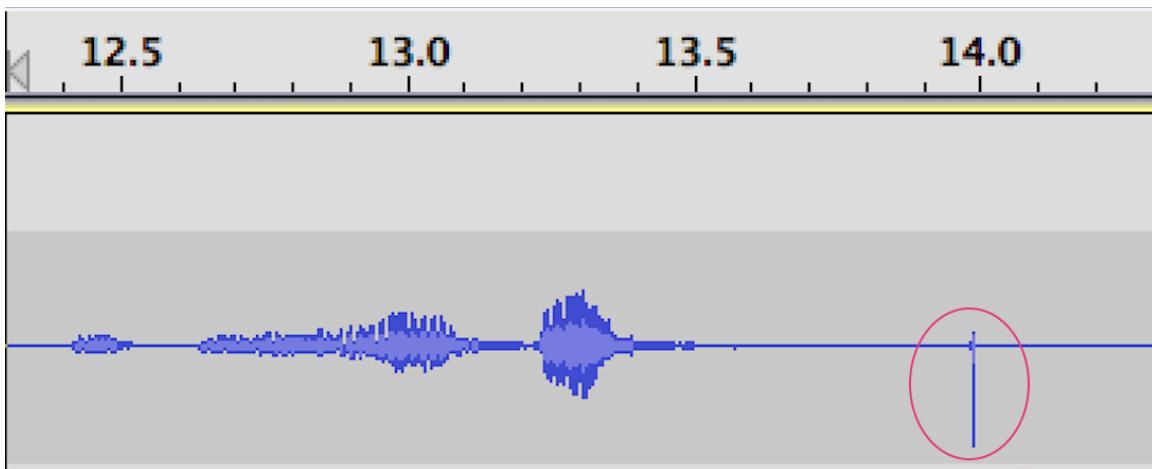


图 9.4: 细微噪音

注：用户的线路是从运营商 PSTN 经过一系列的网关又转成 VoIP 进入会议系统的，噪音产生一般发生在用户端的设备上，或者是在从 PSTN 到 VoIP 转换的环节。

最后，我们给客户的建议是：

- 1) 系统没有错，但是客户也没有错。客户感受到会议中长时间的停顿是由于“绝对静音”引起的，建议在会议系统中增加“舒适噪音”；
- 2) 针对个别的参会方调整会议系统中的能量阈值，以适当其音量大小；
- 3) 更换客户线路，以排查线路引入的细微噪音问题。

最后，客户听取了我们的建议，问题全部解决，客户的客户也满意了。而这一切，都是 Wireshark 的功劳。

小知识：我们在打电话时，如果双方都不说话，在听筒中什么都听不到的话，我们会误以为电话断掉了。所以，一般的电话系统中都会有“舒适噪音”。就是在系统检测到线路上没有声音时，主动加入一定的“吃吃”声，声音不太大，恰好能听见，又不太大，以致影响到人的心情，因此称为舒适噪音。

9.2 谎言终会被揭穿

问题不断，上一个问题还没忙完，就又来了一个。现象是，客户的 FreeSWITCH 系统跟某线路提供商（Carrier）对接时没有声音。拿过抓包，看 SIP 呼叫流程图（略），无异常，SIP 和 RTP 都有。继

续发析 RTP，发现 RTP 流虽然有，但表现很可疑。如图。从图中看，对端的服务器（remote-server）给我们服务器（our-server）发送的 RTP 包是我们的服务器发给对方的 2 倍；并且，对方发送的包大小是 134 字节，而我们发送的是 214 字节。经验告诉我们，对方发错了。

Filter: udp.port == 25290		Expression... Clear Apply Save				
No.	Time	Source	Destination	Protocol	Length	Info
3210	2013-09-14 14:09:19	remote-server	our-server	RTP	134	PT=ITU-T G.711 PCMA, SSRC=0x5A394B9A, Seq=38612, Time=192123936
3211	2013-09-14 14:09:19	remote-server	our-server	RTP	134	PT=ITU-T G.711 PCMA, SSRC=0x5A394B9A, Seq=38613, Time=192123944
3213	2013-09-14 14:09:19	our-server	remote-server	RTP	214	PT=ITU-T G.711 PCMA, SSRC=0x9EE157D7, Seq=56267, Time=123040
3215	2013-09-14 14:09:19	remote-server	our-server	RTP	134	PT=ITU-T G.711 PCMA, SSRC=0x5A394B9A, Seq=38614, Time=192123952
3216	2013-09-14 14:09:19	remote-server	our-server	RTP	134	PT=ITU-T G.711 PCMA, SSRC=0x5A394B9A, Seq=38615, Time=192123960
3219	2013-09-14 14:09:19	our-server	remote-server	RTP	214	PT=ITU-T G.711 PCMA, SSRC=0x9EE157D7, Seq=56268, Time=123200
3222	2013-09-14 14:09:19	remote-server	our-server	RTP	134	PT=ITU-T G.711 PCMA, SSRC=0x5A394B9A, Seq=38616, Time=192123968
3223	2013-09-14 14:09:19	our-server	remote-server	RTP	134	PT=ITU-T G.711 PCMA, SSRC=0x5A394B9A, Seq=38617, Time=192123976
3225	2013-09-14 14:09:19	our-server	remote-server	RTP	214	PT=ITU-T G.711 PCMA, SSRC=0x9EE157D7, Seq=56269, Time=123360
3226	2013-09-14 14:09:19	remote-server	our-server	RTP	134	PT=ITU-T G.711 PCMA, SSRC=0x5A394B9A, Seq=38618, Time=192123984
3227	2013-09-14 14:09:19	remote-server	our-server	RTP	134	PT=ITU-T G.711 PCMA, SSRC=0x5A394B9A, Seq=38619, Time=192123992
3230	2013-09-14 14:09:19	our-server	remote-server	RTP	214	PT=ITU-T G.711 PCMA, SSRC=0x9EE157D7, Seq=56270, Time=123520

众所周知，一般来说 UDP 层的包头是42个字节，即数据链路层14个字节，IP 层20个字节，UDP 层8个字节，一共是 $14 + 20 + 8 = 42$ 。那么，剩余的 172 ($214 - 42 = 172$) 个字节就是 RTP 了。普通的 RTP 包头是12个字节，剩余160个字节的音频采样数据。

本例中使用的是 PCMU 编码，每个采样数据1个字节，因此160个字节就是160个采样数据。PCMU 使用8000赫兹的采样频率，即1秒钟会得到 8000 个字节的数据。所以 $160/8000 = 0.2$ ，即160个字节正好是20毫秒的数据。这个20毫秒称为打包时间，简称ptime。说白了就是每20毫秒发送一个数据包，正常情况下每秒种要发送50 ($8000 / 160 = 50$) 个数据包。

那么，134 是怎么来的呢？第一次见，不妨算一算。 134 (总数) $- 42$ (网络层包头) $- 12$ (RTP 包头) $= 80$ ，80正好是160的一半，即10毫秒的数据。也就是说，远端服务器发给我们的实际上是10毫秒的数据包，而我们的服务器发送的是20毫秒的数据包，怪不得对方发送的包比我们多一倍。

所以上面的情况看起来一切都合情合理，没有什么错误。但是，笔者知道，我们的服务器使用的是 FreeSWITCH，而 FreeSWITCH 不支持不对称的打包方式（即 FreeSWITCH 要求双方的 ptime 一致）。

看来问题就出在这里了。进一步查找原因要看 SIP 中的 SDP。我们服务器在 INVITE 中的 SDP 如下图。ptime=20，正常。

```
▷ Media Description, name and address (m): audio 25290 RTP/AVP 0 8 101
▷ Media Attribute (a): rtpmap:101 telephone-event/8000
▷ Media Attribute (a): fmtp:101 0-16
▷ Media Attribute (a): silenceSupp:off - - -
▷ Media Attribute (a): ptime:20
```

图 9.5: SDP 中 Offer 20ms ptime

再看对方服务器回复的 SDP，里面根本没有ptime。ptime可以省略吗？是的。如果没有ptime，则默认ptime = 20。

所以，对方的服务器没带ptime默认应该发送20毫秒的数据包，它却实际发了10毫秒的，对方撒谎了。量在强大的 Wireshark 面前，谎言是无处遁形的。

```

> Media Description, name and address (m): audio 25290 RTP/AVP 0 8 101
> Media Attribute (a): rtpmap:101 telephone-event/8000
> Media Attribute (a): fmtp:101 0-16
> Media Attribute (a): silenceSupp:off - - -
> Media Attribute (a): ptimetime:20

```

图 9.6: SDP 中 Answer 没有 ptimetime

9.3 迟到的 RTP

这个案例颇有代表性。

问题的现象是客户抱怨通话质量不好，有时听不见声音。

我们拿到了两个抓包文件，其中一个是纯 SIP 的，另一个是纯 RTP。虽然我们可以分别分析这两个文件，但是换来换去的比较复杂。而且，由于纯 RTP 的抓包中没有相关的 SIP 关联信息，分析起来比较困难。好在 Wireshark 为我们想到了这点，在主菜单上点击“File”->“Merge...”就可以将两个抓包文件合并在一起。顿时感觉 Wireshark 真的好强大。

点击“Telephony”->“RTP”->“Show All Streams”可以看到两路 RTP 流。其中，上面一路是我们发给远端的（local->remote），下面一路是远端发给我们的。上面一路各项指标看起来都正常，丢包率0%（当然应该这样，如果发送前就丢包，那就没得玩了），因此，我们重点分析下面一路，即我们接收到的包。

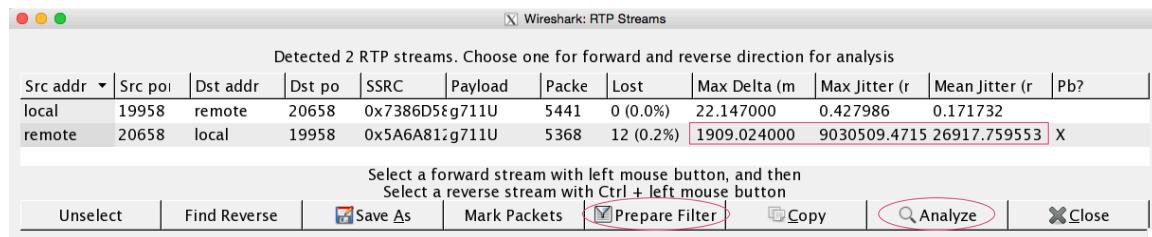


图 9.7:

我们看到下面一路的 Max Delta, Jitter 等指标与上面一路相比相差太大，明显有问题。选中这一路，点击“Prepare Filter”按钮先准备好一个过滤器，并在主窗口中点击“Apply”应用此过滤器，就可以过滤出这一路 RTP。然后点击 Analyze 看详细分析。

No.	Time	Source	Destination	Protocol	Length	Info
143	1.905771	remote	local	RTP	214	PT=ITU-T G.711 PCMU, SSRC=0x5A6A8128, Seq=38542, Time=7680
145	1.925820	remote	local	RTP	214	PT=ITU-T G.711 PCMU, SSRC=0x5A6A8128, Seq=38543, Time=7840
147	1.945801	remote	local	RTP	214	PT=ITU-T G.711 PCMU, SSRC=0x5A6A8128, Seq=38544, Time=8000
157	2.125752	remote	local	RTP	214	PT=ITU-T G.711 PCMU, SSRC=0x5A6A8128, Seq=38545, Time=1155914950
160	2.165773	remote	local	RTP	214	PT=ITU-T G.711 PCMU, SSRC=0x5A6A8128, Seq=38546, Time=1155914970
162	2.185775	remote	local	RTP	214	PT=ITU-T G.711 PCMU, SSRC=0x5A6A8128, Seq=38547, Time=1155915130

图 9.8:

通过浏览 RTP 列表，我们发现在 157 号包处字段的统计值突然变大。在主窗口中找到 157 号包，果然，157 号包的时间戳突变。

这里其实有两个问题：

1) 时间戳突变是允许的，但这个地方应该有一个 Mark 标记，标志时间戳变化了。2) 该 RTP 使用的是 20 毫秒的打包时间，理论上包的 Seq 值以 1 递增，Time 值以 160 递增 ($8000 * 0.20 = 160$)，如 143 到 145，以及 145 到 147 号间；但 157 到 160 号间的 Time 值增幅为 320，错。

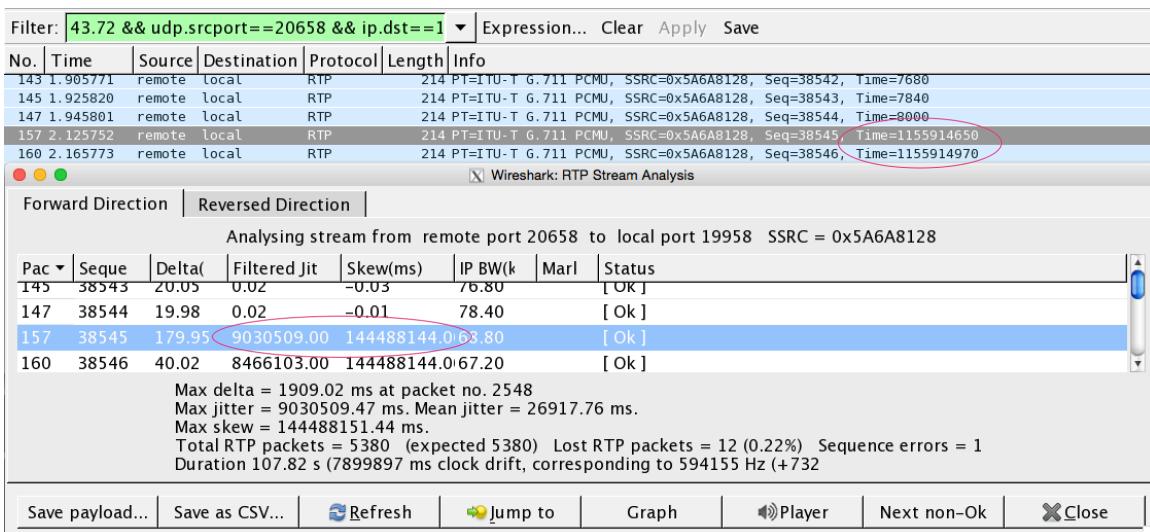


图 9.9:

由于这个时间戳的变化严重影响统计信息，所以，我们做一个过滤，在过滤器中增加条件“`&& frame.number >= 157`”就可以只保留时间戳变化后的 RTP 数据。我们把这些数据通过“Files”->“Export”功能导出到新的文件中，如图。其中“Displayed”表示只有经过我们过滤器过滤后显示的内容被导出。

打开新导出的数据包文件，再重新分析，发现这次的统计数据正常多了，但还是不理想。其中，Delta 的值为系统收到两个包的时间差。由于对方使用 20 毫秒的时间间隔发包，因此，访值应该在 20 左右。如，1147 和 1148 号包的 Delta 值看起来是正常的。但 1149 号包的 Delta 值就一下子到了 1.9 秒，表示收到这个包比预期大约 1.7 秒 (1.9 - 0.2) 的延迟。这一延迟也可以从主窗口上看出来 (从 22.95 到 24.86 秒)。

一般来说，如果实时通话中在大于 0.5 秒的延迟人们听起来就很不舒服了，1.7 秒的延迟就很难实时交流了。

无论如何我们先点击“Player”听一听声音效果。Player 界面如图。

图中有一些参考竖线。画圈的部分，左上角是一个黄色的“W”标记，表示是 Wrong Timestamp，即错误的时间戳。后面红色的“D”表示 Dropped By Jitter Buffer，即被 Jitter Buffer 丢弃。再后面是更多的“W”，如果细心的数的话，这个的“W”差不多应该是 33 个 (见图右下角)。(有时候还会看到 S, Silence Inserted，即插入静音，本例中没有)。

FreeSWITCH WIRESHARK

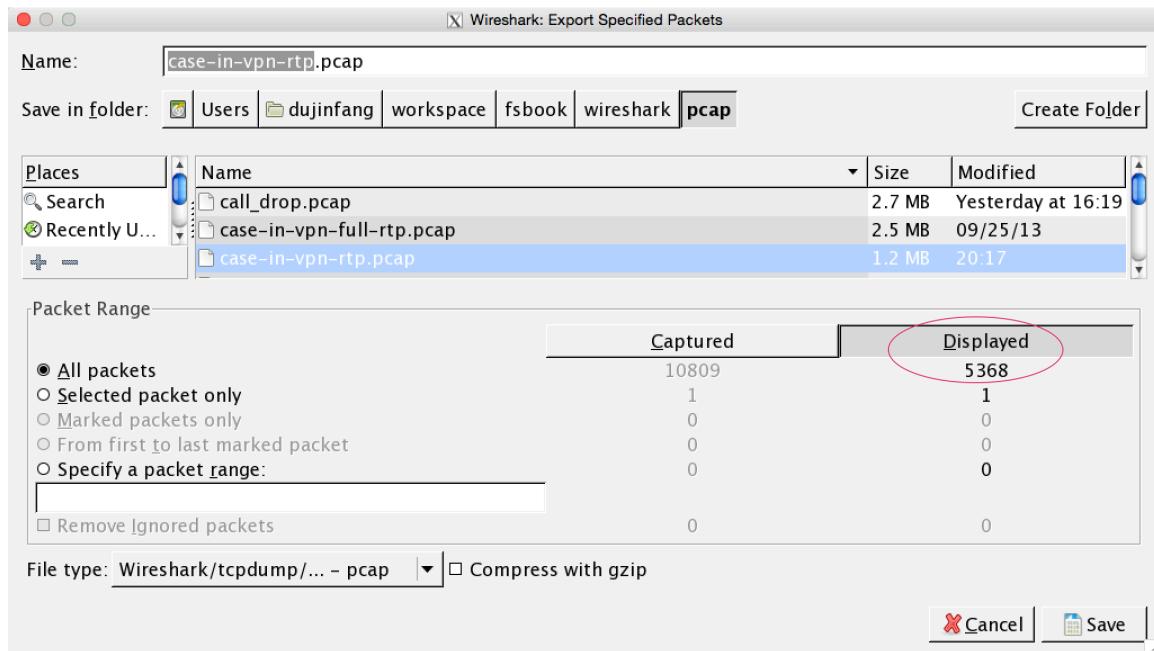


图 9.10:

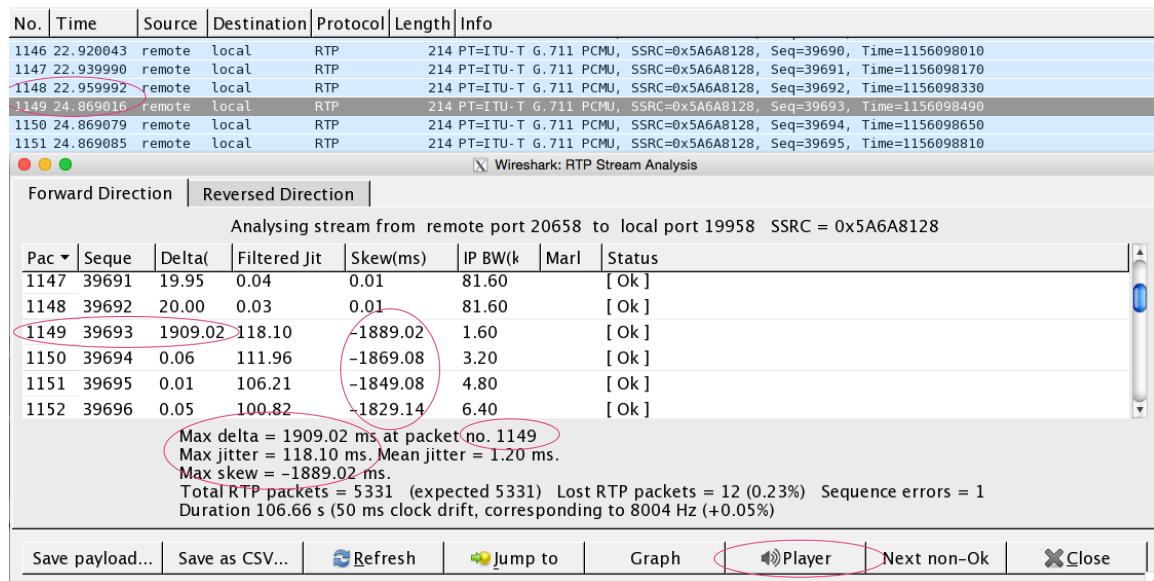


图 9.11:

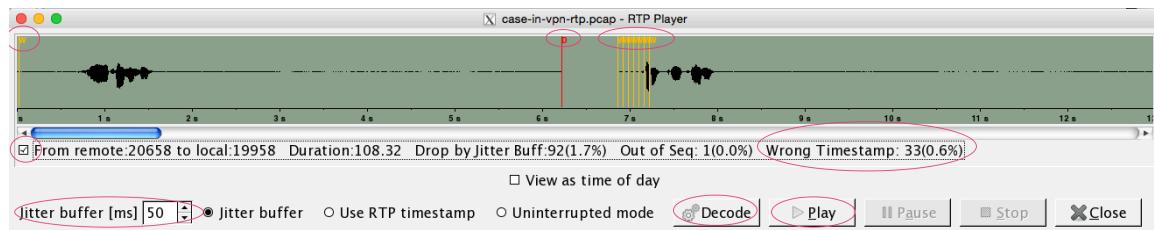


图 9.12:

我们回忆一下刚才讲过的 157 号包和 160 号包，当时我们说时间戳增加了 320 是错误的。我们增加“`&& frame.number >= 157`”为过滤条件导出 RTP 包后，原来的 157 号包就变成了我们这里的 1 号包，而 160 号包则变成了 2 号包，2 号包时间戳有误，因此，这也是为什么左上角会有一个“W”标记。

Jitter Buffer 是一个抖动缓冲区，即在数据接收端要维护一个缓冲区，典型的缓冲区是几十到几百毫秒，以便在遇到时延或抖动（收到乱序的包可在缓冲时间范围内重排）。Wireshark 默认使用 50ms 的缓冲区，可以手动调整缓冲区的值重新“Decode”，减少被缓冲区丢掉的包数，得到更好的音质。在实际的软交换中也通常会有一定大小的缓冲区。缓冲区能提高声音质量，但在延迟和抖动比较多的网络中会增加时延。

选中左下角的小方块，然后点击“Play”按钮，我们就可以听到当时通话的声音了。如图。在 23 秒到 27 秒之间我们听到了明确的停顿。图上也显示很多问题。再回头看图 X，问题就是出在 1149 号包，22.9 到 24.8 秒之间的位置。

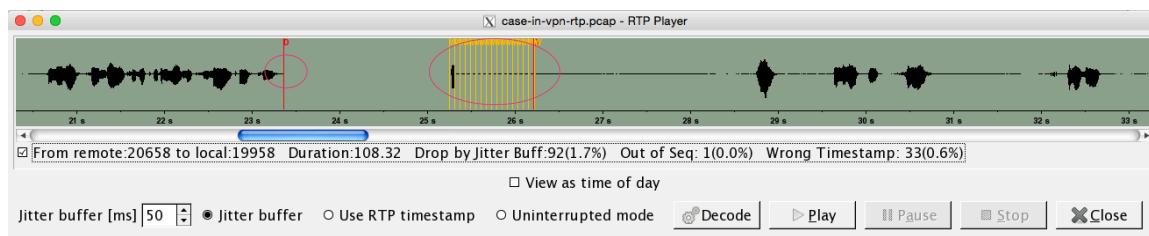


图 9.13:

由于 1149 号包晚到，导致后面其它包都晚到。这个晚到有 Skew 值体现。1149 号包的 Skew 是 -1889.02，该值越接近 0 越好。从图 X 中可以看到，Skew 的值直到 24.88 秒处的 1232 号包才回到接近 0 的状态。但此时我们立即看到了一个 Wrong sequence nr (number)。从 1231 到 1232 号包，Seq 从 39775 变成 39788，发生了丢包 ($39788 - 39775 - 1 = 12$ ，即有 12 个包丢失了)。

No.	Time	Source	Destination	Protocol	Length	Info
1229	24.071552	remote	local	RTP	214	PT=ITU-T G.711 PCMU, SSRC=0x5A6A8128, Seq=39775, Time=1156111290
1230	24.871554	remote	local	RTP	214	PT=ITU-T G.711 PCMU, SSRC=0x5A6A8128, Seq=39774, Time=1156111450
1231	24.871607	remote	local	RTP	214	PT=ITU-T G.711 PCMU, SSRC=0x5A6A8128, Seq=39775, Time=1156111610
1232	24.880068	remote	local	RTP	214	PT=ITU-T G.711 PCMU, SSRC=0x5A6A8128, Seq=39788, Time=1156113690
1233	24.900028	remote	local	RTP	214	PT=ITU-T G.711 PCMU, SSRC=0x5A6A8128, Seq=39789, Time=1156113850

Wireshark: RTP Stream Analysis

Forward Direction		Reversed Direction					
Analysing stream from remote port 20658 to local port 19958 SSRC = 0x5A6A8128							
Pac	Seque	Delta(Filtered Jit	Skew(ms)	IP BW(k	Marl	Status
1230	39774	0.00	20.50	-271.55	131.20		[Ok]
1231	39775	0.05	20.46	-251.61	132.80		[Ok]
1232	39788	8.46	34.90	-0.07	134.40		Wrong sequence nr.
1233	39789	19.96	32.73	-0.03	136.00		[Ok]

图 9.14:

晚到也好，丢包也罢。只要 Skew 恢复到 0，通话还可以正常进行，只是不幸的是，在图 X 中看到，25 秒后又出现了很多“W”……

其实，不管怎么样，从总的统计数据来看，该通话的问题不算严重。而且从录音中我们也听到，

对话的双方互相沟通和理解是没什么问题的，只不是偶尔停下来说几句“Hello……Hello……能听见吗？”之类的。即使不用 VoIP，我们在信号不好的时候用 PSTN 手机打电话不也是经常这样么？

但无论如何用户（客户的客户）觉得不爽啊。就在我们把分析结果告诉客户的时候，客户告诉我们：“是啊，难怪，其实这路通话走的是 VPN，我们有时候直接 Ping 也会发现很大的延时……”

我们建议客户用抗丢包和抖动更好的 iLBC 编码替换 PCMU 试下下，可是客户说对端的设备不支持 iLBC。

最后，说明一点，上面分析的只是抓包所在的服务器收端的情况，至于从服务器发出去的包是否正常到达对端，是否有时延和抖动，那就无法确认了，但从抓包的录音中听，好像也不是很乐观。

9.4 疯狂的按键音

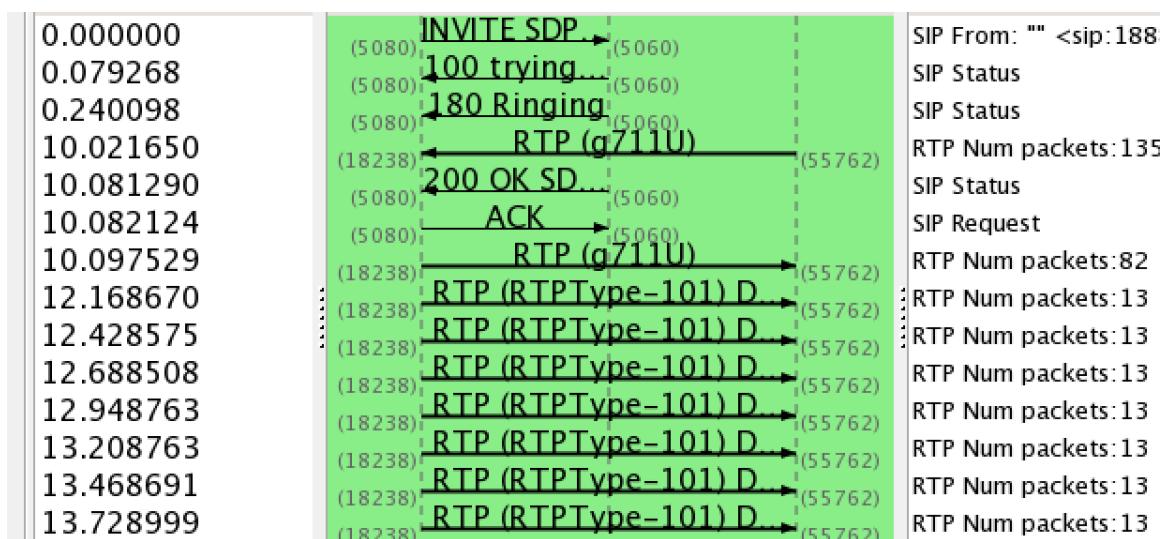


图 9.15: 太多的 DTMF 包

This one was because the far end (Sonus) keep sending us DTMF like crazy.

Attachments:

- calluuid 88f23490-2547-11e3-913b-e7c4e0d777d5.pcap
- calluuid da1fd4bc-2547-11e3-9408-d35ad703072a.pcap
- calluuid 6abce588-2547-11e3-bd59-f7be00057409.pcap

我们来对比一个正确的 DTMF 抓包如图9.16。

最后确认是 Sonus 发包有问题。

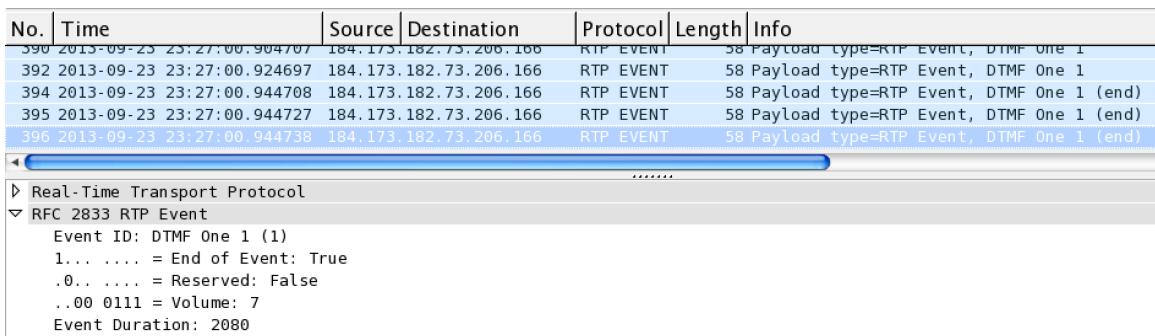


图 9.16: 正确的 DTMF

9.5 谬误的 ACK

先看图9.17。

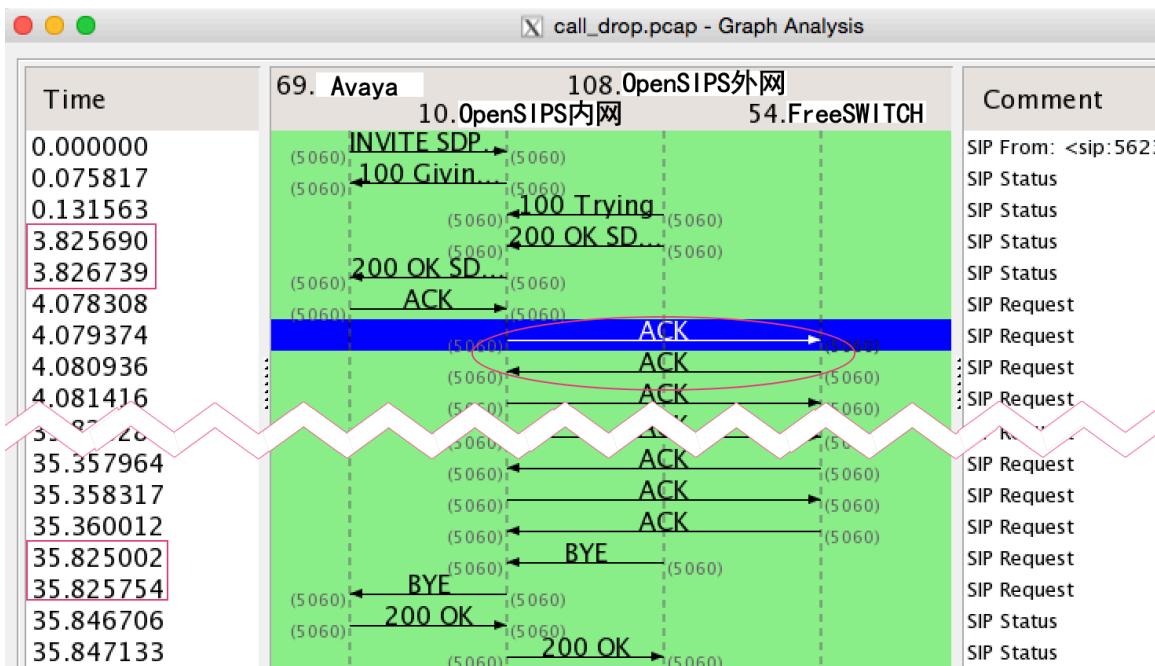


图 9.17: 谬误的 ACK

其中，SIP 呼叫来自 Avaya (一种 SIP 服务器，IP 为 69.Avaya)，到达 OpenSIPS 服务器 (OpenSIPS 是 SIP 代理服务器软件，用于转发 SIP 消息)。OpenSIPS 服务器有两个 IP 10.x.x.x (10.0.OpenSIPS 内网) 和 108.x.x.x (108.0.OpenSIPS 外网)，最后 SIP 会被 OpenSIPS 路由到 FreeSWITCH (54.x.x.x)。

从图中可以看出，该抓包是不完整的。由于它是在 OpenSIPS 服务器上抓的包，因此，不包含完整的呼叫信息。但我们只能根据这些信息进行分析。

图中白线的折线部分把整个图拦腰截断，是因为图太长了，我们省略了中间部分数十条的 ACK

消息。

虽然信息不大完整，但非常像我们在第7.2节讲的由于收不到 ACK 导致 30 秒断的问题。初步判断 FreeSWITCH 并没有收到 ACK。事实上，在图中省略的部分我们可以看到重复发送的 200 OK 消息，更表明 FreeSWITCH 没有收到 ACK 而预防性的重发。另外，从 3.8 秒左右的 200 OK 消息到 35.8 秒左右的 BYE 消息，也差不多符合“30 秒”断的情况。

那么，如果我们分析不错的话，图中那么多 ACK 明明是指向 FreeSWITCH 服务器的，到底去了哪儿呢？

事实上，仔细分析一下可以发现，**10.x.x.x**是一个私网地址，从该地址发往**54.x.x.x**地址的消息可能是无法路由的，或者

观察由**10.0penSIPS**内网发出的 ACK 消息，它好像被路由器反射回来，因为它收到的下一个 ACK 消息里的 CSeq 值和 Max-Forward 值都是一样的。至于哪个地方发生了反射，我们不得而知，从抓包中的数据也分析不出来。

```
Call-ID: 0c2395ef53ee31523b504111b800
▷ CSeq: 1 ACK
User-Agent: Avaya CM/R016x.00.1.510.1 AVAYA-SM-6.1.4.0.614005
Content-Length: 0
Max-Forwards: 65
```

继续看**10.0penSIPS**内网发出的下一个 ACK，CSeq 值没变，但 Max-Forward 值由 65 变成了 64。表明该包经过一次转发。SIP 中的 Max-Forward 类似于 IP 层的 TTL (Time To Live) 值，该值每经过一次转发，都会减 1，如果减到 0，就会终止转发，以防止产生死循环。所以，这里很显示，ACK 消息从某处反射回 OpenSIPS 后，又被 OpenSIPS 处理了一次，重新向 FreeSWITCH 转发，只是……不幸的是，又被原样反射了回来……

```
▷ CSeq: 1 ACK
User-Agent: Avaya CM/R016x.00.1.510.1 AVAYA-SM-6.1.4.0.614005
Content-Length: 0
Max-Forwards: 64
```

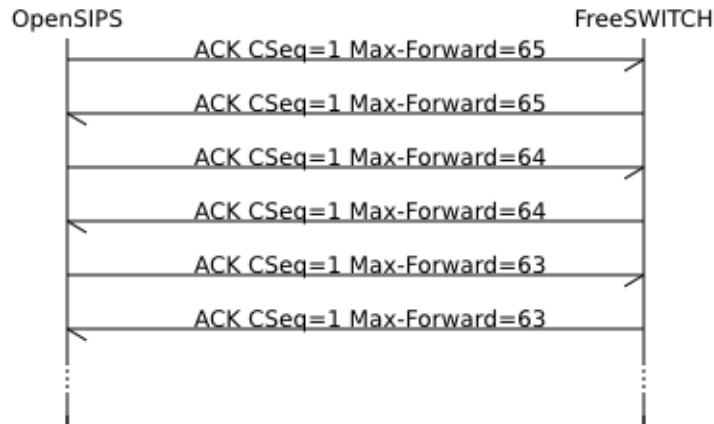
所以，看起来这些循环的 ACK 包就如下图所示。

分析出这里，出现问题的原因基本上有两个：

- 1) Avaya 发送的 ACK 消息有误，导致转发时发生循环
- 2) OpenSIPS 脚本配置有问题，导致发生循环

我们检查了 OpenSIPS 配置脚本，没发现明显的问题。遂继续检查 ACK 消息。从第 X 节我们知道，Avaya 应该根据收到的 200 OK 中的 Contact 地址发包。200 OK 包中的 Contact 消息如图。

但是，从下图中我们看到，Avaya 发送的 ACK 消息中，第一行的 Request Line 中确写的是 **sip:408xxxx@54.FreeSWITCH** 的地址，因而，ACK 消息到达 OpenSIPS 后，引起了转发错误。



No.	Time	Source	Destination	Protocol	Length	Info
13	2013-10-15 01:48:58	OpenSIPS-Private	Avaya	SIP/SDP	1253	Status: 200 OK
↓ CSeq: 1 INVITE						
↓ Contact: <sip:408@108.160.141.14>;transport=udp						

所以该问题是由于 Avaya 发送错误的 ACK 引起的，理应由 Avaya 侧修正。但是，如果 Avaya 侧就是不修正，OpenSIPS 侧也有容忍这种错误的方案。后来，客户在 OpenSIPS 中增加了一些配置，绕过了该问题。

9.6 小结

本来笔者是冒充 FreeSWITCH 专家去帮客户解决问题的，没想到后来成了练成了半个 Wireshark 专家……

No.	Time	Source	Destination	Protocol	Length	Info
15	2013-10-15 01:48:58	Avaya	OpenSIPS-Private	SIP	516	Request: ACK sip:408@54.160.141.14;transport=udp
↓ Session Initiation Protocol (ACK)						
↓ Request-Line: ACK sip:408@54.160.141.14;transport=udp SIP/2.0						

写在最后

本书将持续更新，这就是电子版的好处…

如果你对书中的内容和章节安排等有什么意见或建议，欢迎与我联系。如果你建议的内容适合放在本书里，我会考虑写进去；如果不适合放到本书中，我也会考虑写其它主题的书。

如果你的公司想在本书是植入广告或者赤裸裸地做广告，也欢迎与我们联系。

电子邮件：info@x-y-t.cn。

作者简介

杜金房 (网名: Seven) 资深网络通信技术专家，在网络通信领域耕耘近 15 年，精通 VoIP、SIP 和 FreeSWITCH 等各种网络协议和技术，经验十分丰富。有超过 7 年的 FreeSWITCH 应用和开发经验，不仅为国内大家大型通信服务厂商提供技术支持和解决方案，而且客户还遍及美国、印度等海外国家。

FreeSWITCH-CN 中文社区创始人兼执行主席，被誉为国内 FreeSWITCH 领域的『第一人』；在 FreeSWITCH 开源社区非常活跃，不仅经常为开源社区提交补丁和新功能、新特性，而且还开发了很多外围模块和外围软件；此外，他经常在 FreeSWITCH 的 Wiki 上分享自己的使用心得和经验、在 FreeSWITCH IRC、QQ 及微信群中热心回答网友提问，并不定期在国内不同城市举行 FreeSWITCH 技术培训；自 2011 年起每年都应邀参加在美国芝加哥举办的 ClueCon 大会，并发表主题演讲。

此外，他还精通 C、Erlang、Ruby、Lua 等语言相关的技术。

著有《FreeSWITCH 权威指南》，2014 年出版。

创办了[北京信悦通科技有限公司](#)和[烟台小樱桃网络科技有限公司](#)，提供 FreeSWITCH 培训和商业技术支持服务。

版权声明

本书版权归作者所有，任何人未经书面授权均不得分发此书。

本书电子版仅在 SelfStore (<https://selfstore.io/~dujinfang>) 上发布。如果您不小心从其它渠道获得本书，请删除您的版本并到 SelfStore 上购买正版。

本书纸质版仅随电子版赠送。SelfStore 支持买家自由定价。如果您购买电子书时支付**超过**一定金额，可以通过电子邮件联系我们**赠送**一本精美打印的纸质书。

广告

关于广告的广告

请允许我在本书中发布广告。广告合作联系邮箱：info@x-y-t.cn。

FreeSWITCH 第五届开发者沙龙将于 2016 年 8 月 27 日在京举行

<http://www.freeswitch.org.cn/2016.html>

FreeSWITCH 培训 2016 夏季班（北京站）将于 2016 年 8 月 28-30 日在京举行

<http://x-y-t.cn/fst1608.html>

烟台小樱桃网络科技有限公司提供商业 FreeSWITCH 及 OpenSIPS 技术支持

网址：<http://x-y-t.cn> 邮箱：info@x-y-t.cn

烟台小樱桃网络科技有限公司是潮流网络（GrandStream）山东总代理

深圳市潮流网络技术有限公司（Grandstream）是全球知名的统一通讯和整体解决方案厂商和全球 TOP3 VoIP 终端供应商，是国内 IMS、统一通信、呼叫中心、调度市场、视频监控的主要 SIP 终端供货厂商，成功案例包括京东、网易、凡客诚品、平安、中集、东航、国家电网、中石化、中石油、

外交部、中移动、中电信等等，公司是中国三大运营商 IMS 市场主要核心合作 SIP 终端厂商，入围中国电信集团 IMS SIP 电话短名单，持续配合运营商研究院 SIP 硬终端的核心业务及协议定制，参与及进入中电信、中移动多个省试点应用和产品供应厂商。在全球与渠道及增值合作伙伴成功服务将近千万终端用户，包括全球主流运营商数十万套终端以及美国共和党和美国民主党数万套 SIP 电话和网关等大型成功部署案例，连续 10 年保持近 50% 年复合增长率。

烟台小樱桃网络科技有限公司，是潮流全线产品在山东省的总代理。

联系方式：邮箱：info@x-y-t.cn 电话 0535-6753997

FreeSWITCH 相关图书

《FreeSWITCH 文集》收集了一些 FreeSWITCH 文章，相比其它 FreeSWITCH 书来说，技术内容比较少，便于非技术人员快速了解 FreeSWITCH。

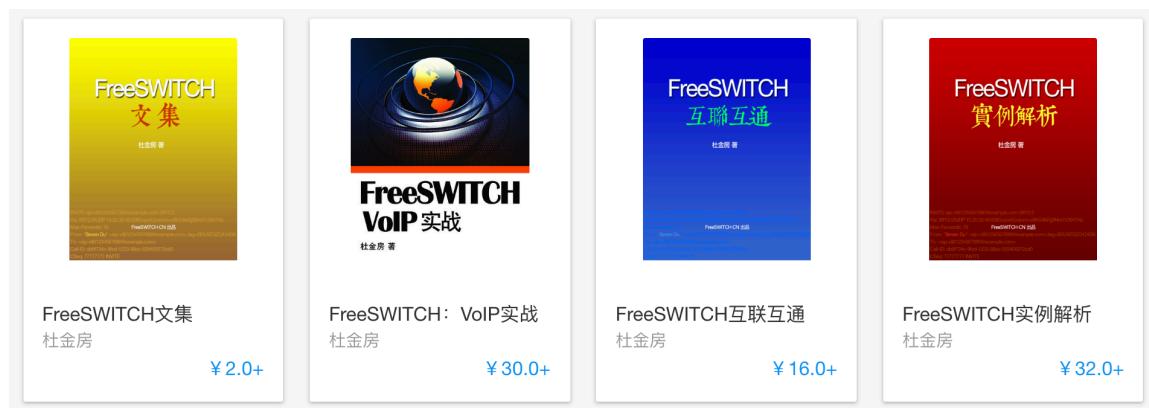
《FreeSWITCH 互联互通》主要收集了一些互联互通的例子。书中有些例子来自《FreeSWITCH 权威指南》。

《FreeSWITCH 实例解析》收集了一些如何使用 FreeSWITCH 的实际例子，方便读者参考。书中有些内容来自《FreeSWITCH 权威指南》。

《FreeSWITCH 实战》是《FreeSWITCH 权威指南》的前身，不再更新，但该书有其历史意义。

以上图书均为电子书，可在 SelfStore (<https://selfstore.io/~dujinfang>) 上购买。SelfStore 支持买家自由定价。如果您购买电子书时支付超过一定金额，可以联系通过电子邮件联系我们赠送一本精美打印的纸质书。

《FreeSWITCH 权威指南》是正式出版的纸质书，纸质版和电子版均可在以下网站购买：<http://book.dujinfang.com>。





作者简介



杜金房（网名：Seven Du）资深网络通信技术专家，拥有十多年通信行业经验。精通VoIP、SIP和消息领域相关技术，精于VAD、SIP和FreeSWITCH等多种协议的协议设计、经验十分丰富。有超过6年的FreeSWITCH应用和开发经验，不仅为阿里云打造了全球最大的VoIP呼叫中心解决方案，还为阿里巴巴集团、蚂蚁金服、钉钉等提供解决方案，以及众多企业和跨国企业等多家FreeSWITCH-CNP中文社区创始人和执行主席，被誉为国内FreeSWITCH领域的“第一人”。拥有丰富的开源项目开发经验，不仅参与过Apache、FreeBSD等开源项目的贡献，还开发了诸多开源项目和框架。此外，他还经常在FreeSWITCH CHINA上分享自己的核心技术经验和自由开源的精神，自FreeSWITCH INC及CNP中国开源社区成立以来，一直担任核心贡献者和FreeSWITCH中文社区执行主席。2013年，2014年和2015年连续被评为FreeSWITCH中文社区贡献者。2016年，被评为FreeSWITCH中文社区贡献者，并发表主题演讲。此外，他还精通与C、Erlang、Roby、Lua等语言相关的技术。



FreeSWITCH 权威指南

FreeSWITCH: The Definitive Guide

FreeSWITCH权威指南



内容简介

FreeSWITCH是世界上最一个平台的、伸缩性最好的、最全面的、多层次的软交换系统。本书是FreeSWITCH领域最为权威的著作之一，在这本书面前，FreeSWITCH无秘密！

由中国的FreeSWITCH第一人——金牌FreeSWITCH开源项目执行长名专家、FreeSWITCH-CNP中文社区创始人执行主席Sevan Du执笔，FreeSWITCH之父布力维·帕雷利耶倾力全副，倾尽心血，倾尽才智，倾尽力量，从企划到完成，历经数年，各阶段都倾注了极大的热情。本书从架构与编程、理论与实践和应用、系统的性能优化、调试与开发、若干进阶功能的二次开发和高深实践原理，深入探讨了FreeSWITCH的底层架构、实践性能数据、实用性数据，从开明坦诚、深入人心的会话、话单与事件记录的配置文件的生产环境中的需求实现，从单个的FreeSWITCH应用到多个FreeSWITCH集群，从跨群应用到整个企业的二次开发，各种案例均用配有大量案例分析的代码加以举重若轻地使用。很多案例中的代码都可以拿来直接使用。



出版地：北京·三里屯路
出版时间：2013-05-06
ISBN：978-7-111-46625-9
定价：52.00元

THIS PAGE INTENTIONALLY LEFT BLANK.