

WebRTC 零基础开发者教程

版本 0.3 (2014 年 2 月)
康林 (16614119@qq.com)

版本	内容	时间	作者
V0.1	初建文档	2013 年 11 月	康林
V0.2	增加协议文档	2014 年 2 月 8 日	康林
V0.3	细化了 windows 下编译过程。 增加 IDE 工具用法。 重新排版整个文档。	2014 年 2 月 20 日	康林

目录

1	工具	4
1.1	depot_tools	4
1.1.1	目标	4
1.1.2	Chromium	4
1.1.3	使用说明在这儿	4
1.1.4	下载	4
1.1.5	使用	4
1.1.6	具体使用例子:	5
1.2	Gyp 工具	5
1.3	Python 工具	5
1.4	本地集成开发环境 (IDE)	5
1.4.1	Visual studio	5
1.4.2	Kdevelop	11
1.4.3	Eclipse	12
2	Webrtc	12
2.1	下载、编译	12
2.1.1	Windows 下	12
2.1.2	ubuntu 下编译	16
2.1.3	编译 Android(只能在 linux 下)	19
3	webrtc 开发	19
3.1	开发 P2P 视频软件需要处理的问题	19
3.1.1	用户列的获取、交换、信令的交换	19
3.1.2	P2P 通信	19
3.1.3	多媒体处理	20
3.2	webrtc 架构:	20
3.2.1	WebRTC 架构组件介绍	20
3.2.2	WebRTC 核心模块 API 介绍	22
3.2.3	webRTC 核心 API 详解	27
4	Libjingle 详细介绍	35
4.1	重要组件	35
4.1.1	信号	35
4.1.2	线程和消息	35
4.1.3	名称转换	35
4.1.4	Ssl 支持	35
4.1.5	连接	35
4.1.6	传输, 通道, 连接	36
4.1.7	候选项	36
4.1.8	数据包	36
4.2	如何工作	36
4.2.1	Application 模块	37
4.2.2	XMPP Messaging Component 模块	37
4.2.3	Session Logic and management component 模块	38
4.2.4	Peer to peer Component 模块	38

4.2.5	其他.....	39
4.3	建立 libjingle 应用程序.....	39
5	代码分析.....	39
5.1	音频通道建立过程.....	39
5.2	音频接收播放过程.....	40
5.3	视频接收播放过程.....	43
6	协议:.....	45
6.1	XMPP 协议.....	45
6.1.1	原理介绍.....	45
6.1.2	XMPP 协议网络架构.....	46
6.1.3	XMPP 协议的组成.....	48
6.1.4	Xmpp 介绍.....	49
6.1.5	协议内容.....	49
6.2	Stun 协议.....	52
6.2.1	P2P 实现的原理.....	53
6.2.2	P2P 的常用实现.....	55
6.2.3	Stun URI.....	58
6.2.4	内容.....	58
6.2.5	中文内容.....	59
6.2.6	开源服务器.....	61
6.2.7	公开的免费 STUN 服务器.....	61
6.3	Turn 协议.....	62
6.3.1	概念.....	62
6.3.2	Turn uri.....	63
6.3.3	开源服务器工程.....	63
6.3.4	开源库.....	63
6.4	交互式连接建立 (Interactive Connectivity Establishment)	63
6.4.1	IETF 规格.....	68
6.4.2	开源工程:	69
6.5	XEP-0166 Jingle.....	70
6.5.1	绪论.....	71
6.5.2	需求.....	72
6.6	Sctp 协议.....	95
6.7	Rtp 协议.....	98
7	附件.....	100
7.1	Gyp 工具.....	100
7.2	Google test 程序.....	102
7.3	Webrtc 库介绍.....	103
7.4	webrtc 代码相关基础知识.....	104
7.5	STUN 和 TURN 技术浅析.....	106
7.6	基于 ICE 的 VoIP 穿越 NAT 改进方案.....	107
7.7	ubuntu 安装使用 stuntman.....	112
7.8	一个开源的 ICE 库——libnice 介绍.....	112
7.9	4 种利用 TURN 穿越对称型 NAT 方案的设计与实现.....	114
7.10	基于 ICE 方式 SIP 信令穿透 Symmetric_NAT 技术研究.....	115

1 工具:

1.1 depot_tools:

chromium 自己整了一套开发工具系统, 原来叫 gclient (名字好像让位给 google 桌面客户端了), 现在改名 depot_tools.

1.1.1 目标:

Wrapper script for checking out and updating source code from multiple SCM repository locations.

chromium 使用了 (目前 @159834) 107 个代码仓库的代码, 这些分散在多个代码仓库, chromium 不需要某些仓库的东西, google 就封装个工具, 这个工具既支持 svn, 也支持 git, 不光能 down 代码, 也支持了

- patch
- cpplint, pylint
- apply_issue
- junction
- codereview

1.1.2 chromium 使用它来

- 更新 chromium 代码
- 生成工程文件, windows 上生产 sln, mac 生产 xcode 工程, linux 生成 scons 或者 makefile
- 其他的 patch, codereview, 管理分散开发人员的修改

1.1.3 使用说明在这儿

<http://www.chromium.org/developers/how-tos/depottools>

1.1.4 下载:

1.1.4.1 linux 下:

```
sudo apt-get install git
```

```
git clone https://chromium.googlesource.com/chromium/tools/depot\_tools.git
```

1.1.4.2 window 下:

- 已装 cygwin:

```
git clone https://chromium.googlesource.com/chromium/tools/depot\_tools.git
```

- 无 cygwin:

https://src.chromium.org/svn/trunk/tools/depot_tools.zip

1.1.5 使用:

1.1.5.1 用 gclient 获取代码

- 首先会更新 depot_tools, 有两种 bat 和 sh, 目的都一样
更新 depot_tools, 然后运行 python 版 gclient.py, 参数都传给 gclient.py
这里解决了鸡生蛋还是蛋生鸡的问题, 更新了 gclient.py
- 生成 gclient 文件, gclient 指定了某个版本的 chromium 代码
- 执行 gclient sync, 更新代码, 生成工程文件, 这里使用了另一个工具 GYP

1.1.5.2 gclient 命令:

Commands are:

cleanup	Cleans up all working copies.
config	Create a .gclient file in the current directory.
diff	Displays local diff for every dependencies.
fetch	Fetches upstream commits for all modules.
help	Prints list of commands or help for a specific command.
hookinfo	Output the hooks that would be run by `gclient runhooks`
pack	Generate a patch which can be applied at the root of the tree.

recurse	Operates on all the entries.
revert	Revert all modifications in every dependencies.
revinfo	Output revision info mapping for the client and its dependencies.
runhooks	Runs hooks for files that have been modified in the local working copy.
status	Show modification status for every dependencies.
sync	Checkout/update all modules.
update	Alias for the sync command. Deprecated.

Prints list of commands or help for a specific command.

Options:

--version	show program's version number and exit
-h, --help	show this help message and exit
-j JOBS, --jobs=JOBS	Specify how many SCM commands can run in parallel; default=8
-v, --verbose	Produces additional output for diagnostics. Can be used up to three times for more logging info.
--gclientfile=CONFIG_FILENAME	Specify an alternate .gclient file
--spec=SPEC	create a gclient file containing the provided string. Due to Cygwin/Python brokenness, it probably can't contain any newlines.

1.1.6 具体使用例子:

1.1.6.1 安装工具

<http://www.chromium.org/developers/how-tos/install-depot-tools>

1.1.6.2 .配置

主要是写

.gclient 和 DEPS python 语法(精确点就是 json 语法+ “#” 型注释, list 最末元素可以有, 执行时使用 python 的 eval 来解释的)

.gclient

1.2 Gyp 工具

Gyp 工具简介见附件。Google 自己搞的玩意。Webrtc 不是直接用的 gyp, 而是又封装了一下。

Webrtc 中的 gyp 工具是 build/ gyp_chromium

1.3 Python 工具

整个 webrtc 工程是由 python 程序进行维护。而整个 webrtc 工程现在处理一个完全不稳定的阶段。所以需要了解一些基本的 python 语法知识。

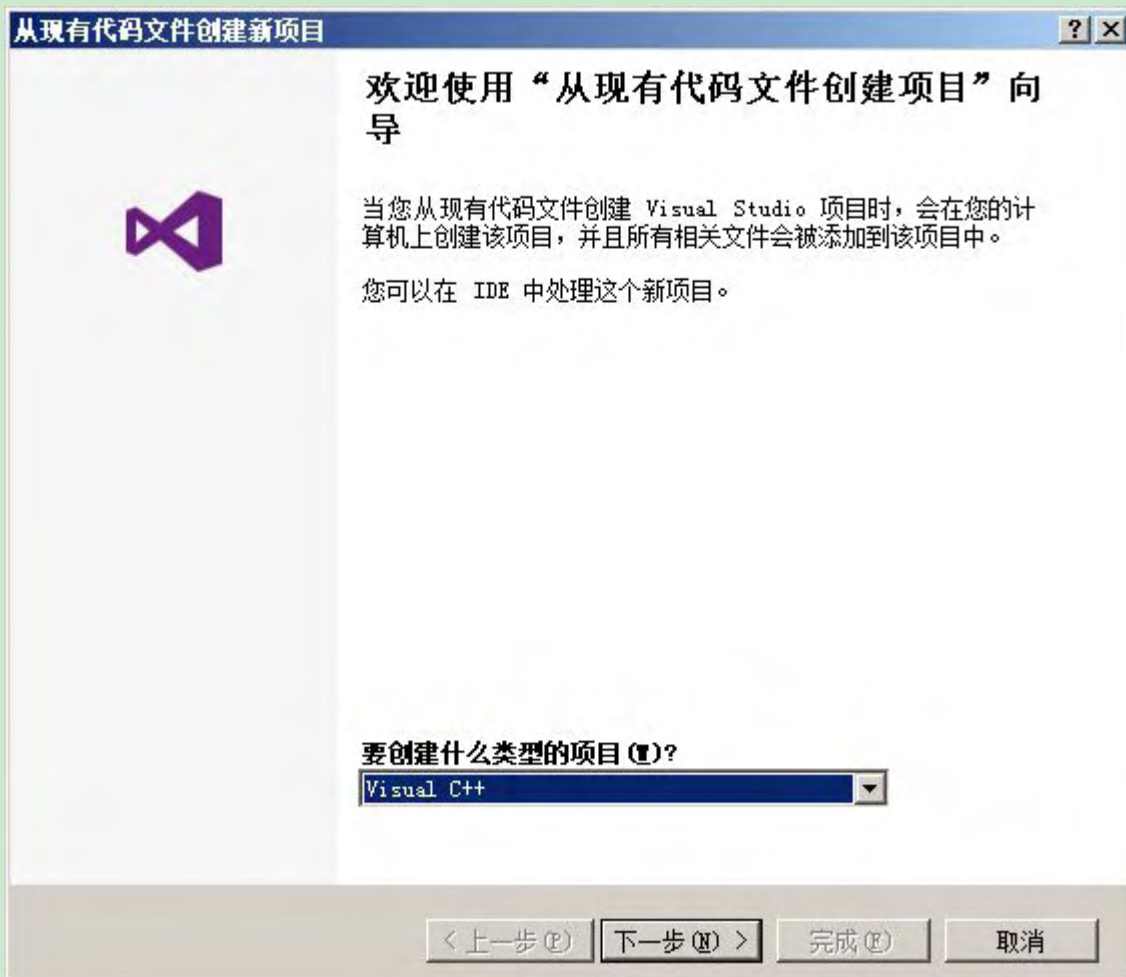
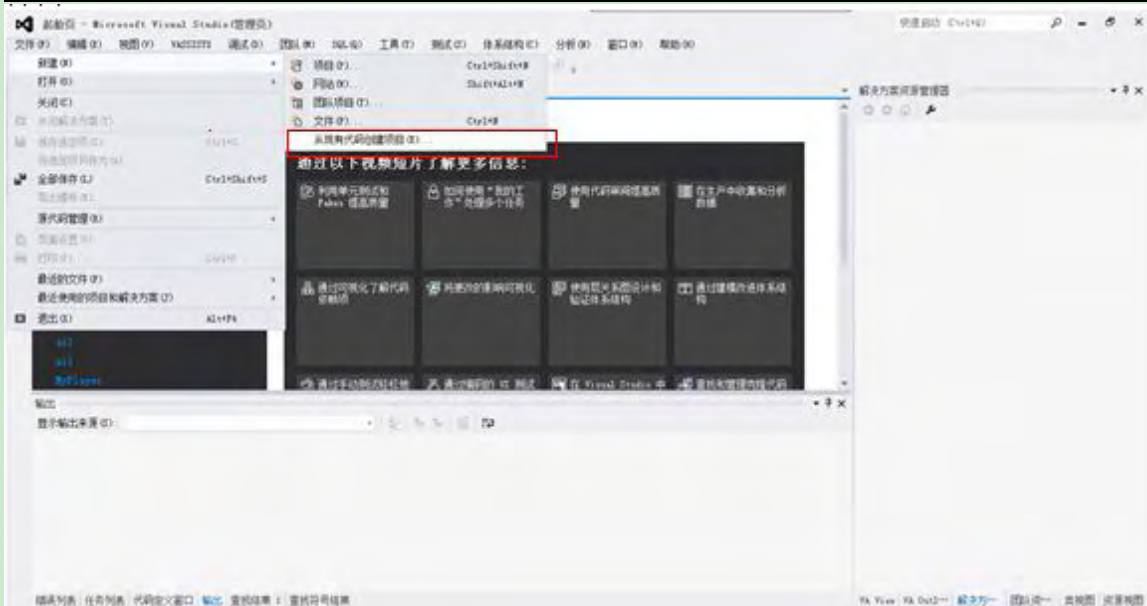
1.4 本地集成开发环境 (IDE)

工欲善其事, 必先利其器。所以下面介绍下本地集成开发环境。

1.4.1 Visual studio

一般的在 windows 本地开发环境 (IDE) 是 visual studio。然而 webrtc 默认的是 ninja。虽然可以手工生成 vs 工程, 但是经常出现兼容性问题。所以下面介绍下 vs 加 ninja 开发环境。Webrtc 默认生成的工程位于源码根目录下的 out 目录中。

打开 vs2012:



从现有代码文件创建新项目

指定项目位置和源文件

您可以从一个或多个文件夹中选择文件。

选择webrtc目录

项目文件位置 (L):

E:\source\muli\google\trunk

浏览 (O)...

项目名称 (R):

webrtc

☒ 将文件从这些文件夹添加到项目中 (T)

文件夹 (O):

添加子文件夹	文件夹
<input checked="" type="checkbox"/>	E:\source\muli\google\trunk

添加 (A)...

移除 (R)

要添加到项目中的文件类型 (Y):

.cpp;.cxx;*.cc;*.c;*.inl;*.h;*.hpp;*.hxx;*.hm;*.inc;*.rc;*.r

☒ 在解决方案资源管理器中显示所有文件 (X)

< 上一步 (P)

下一步 (N) >

完成 (F)

取消

从现有代码文件创建新项目

?

×

指定项目设置

这些详细信息决定了将如何生成项目以及所创建的项目类型。

您希望如何生成项目?

☐ 使用 Visual Studio (V)

项目类型 (T):

Windows 应用程序项目

☐ 添加对 ATL 的支持 (A)

☐ 添加对 MEC 的支持 (M)

☐ 添加对公共语言运行时的支持 (C)
公共语言运行时支持:

公共语言运行时支持

☒ 使用外部生成系统 (E)

要指定生成命令行, 请单击“下一步”在“指定调试配置设置”页和“指定发布配置设置”页上进行设置。

< 上一步 (P)

下一步 (N) >

完成 (F)

取消

从现有代码文件创建新项目

指定调试配置设置

设置“调试”配置的设置。

要对“调试”配置进行哪些设置?

生成命令行 (L):

ninja -C out\Debug All

重新生成命令行 (M):

ninja -C out\Debug All

清除命令行 (C):

输出 (用于调试) (O):

预处理器定义 (/D) (P):

包括搜索路径 (/I) (I):

强制包含的文件 (/FI) (E):

.NET 程序集搜索路径 (/AI) (S):

强制使用 .NET 程序集 (/FU) (U):

使用外部生成系统时，命令行中包含的是在生成操作发生时要执行的命令，而输出选项指定要调试的可执行程序名称。

预处理器定义 (包括搜索路径、强制包含的文件、程序集搜索路径和强制使用程序集) 用于 IntelliSense。当不使用外部生成系统时，这些设置还控制 Visual Studio 生成项目的方式。

< 上一步 (P)

下一步 (N) >

完成 (F)

取消

从现有代码文件创建新项目

?

×

指定发布配置设置

设置“发布”配置的设置。

要对“发布”配置进行哪些设置?

生成命令行 (L):

ninja -C out\Release All

预处理器定义 (/D) (P):

重新生成命令行 (M):

ninja -C out\Release All

包括搜索路径 (/I) (I):

清除命令行 (C):

强制包含的文件 (/FI) (E):

输出 (用于调试) (O):

.NET 程序集搜索路径 (/AI) (S):

☐ 与“调试”配置相同 (D)

强制使用 .NET 程序集 (/FU) (U):

使用外部生成系统时，命令行中包含的是在生成操作发生时
要执行的命令，而输出选项指定要调试的可执行程序名称。

预处理器定义 (包括搜索路径、强制包含的文件、程序集搜索路径和强制使用程序集)
用于 IntelliSense。当不使用外部生成系统时，
这些设置还控制 Visual Studio 生成项目的方式。

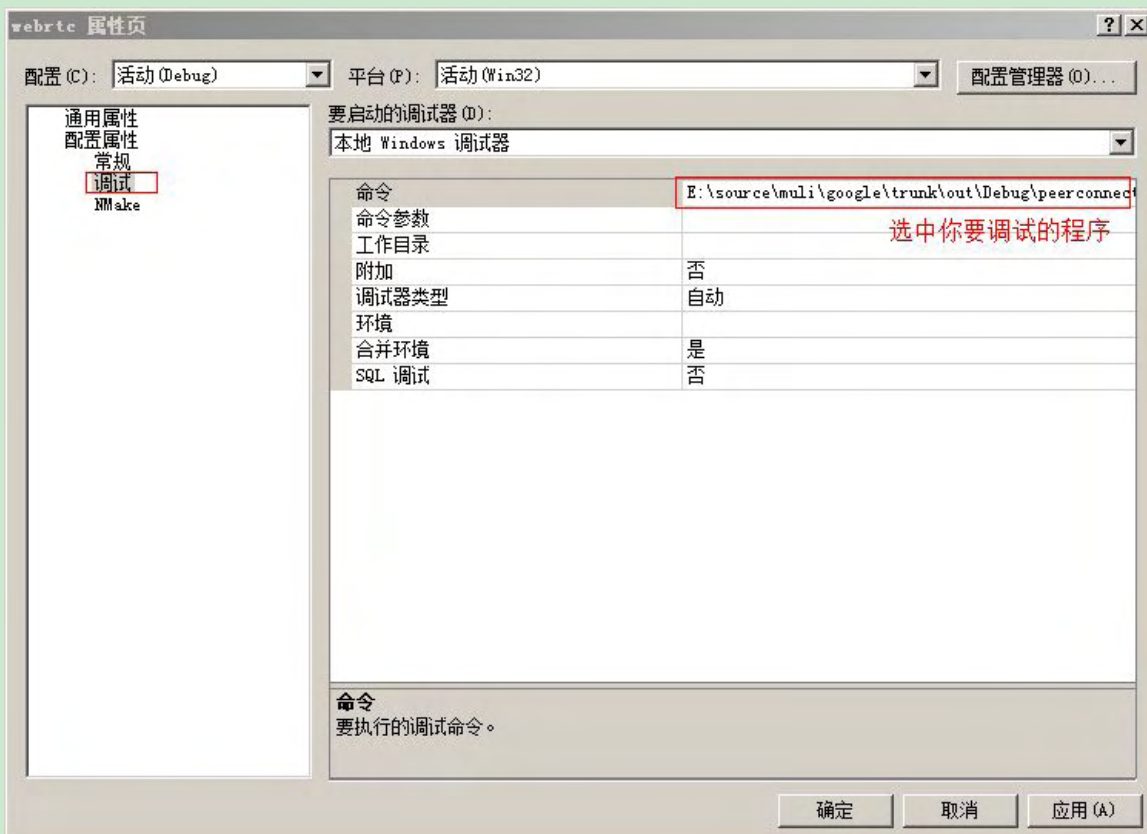
< 上一步 (P)

下一步 (N) >

完成 (F)

取消

第 10 页 共 116



1.4.2 Kdevelop

官网地址: <http://kdevelop.org/>

它是一个跨平台开源的集成开发环境。用 c++ 开发。所以效率比较高。本人比较喜欢用它在 linux 下开发, 或用它开发跨平台程序。它默认支持 ninja。

1.4.3 Eclipse

官网地址: <http://eclipse.org/>

Eclipse 是一个用 java 写的跨平台集成开发环境。支持 java、c、c++ 等。它由于是用 java 开发的, 所以效率比用 c、c++ 开发的低。如果你的机器比较好, 用这个也挺文便的。用于 android 开发。

2 Webrtc

原文地址: <http://www.webrtc.org/reference/getting-started>

2.1 下载、编译:

注意: 由于天朝的限制, 所以下载是一个艰难的过程 (最好翻墙)。由于 google 采用 ninja 做为默认编译工程。所以当你用其它本地建立方式 (VS、make 等) 时, 你要自己解决兼容问题。关于建立工具、编译工具的版本, 最好采用最新的版本, 否则也会出现兼容性问题。

相关工具安装, 最好采用默认值。否则也可能出现兼容问题。

2.1.1 Windows 下:

1) 安装 vs2012 旗舰版

<http://www.microsoft.com/zh-cn/download/details.aspx?id=30678>

注册号:

Visual Studio 2012 Ultimate 旗舰版序列号:

YK CW6-BPFPF-BT8C9-7DCTH-QXGWC
RBCXF-CVBGR-382MK-DFHJ4-C69G8
YQ7PR-QTHDM-HCBCV-9GKGG-TB2TM

建立用最新版本 vs, vs2008 以前版本不能保证百分之百兼容。因为开发者用的是最新版本。

2) 安装依赖库:

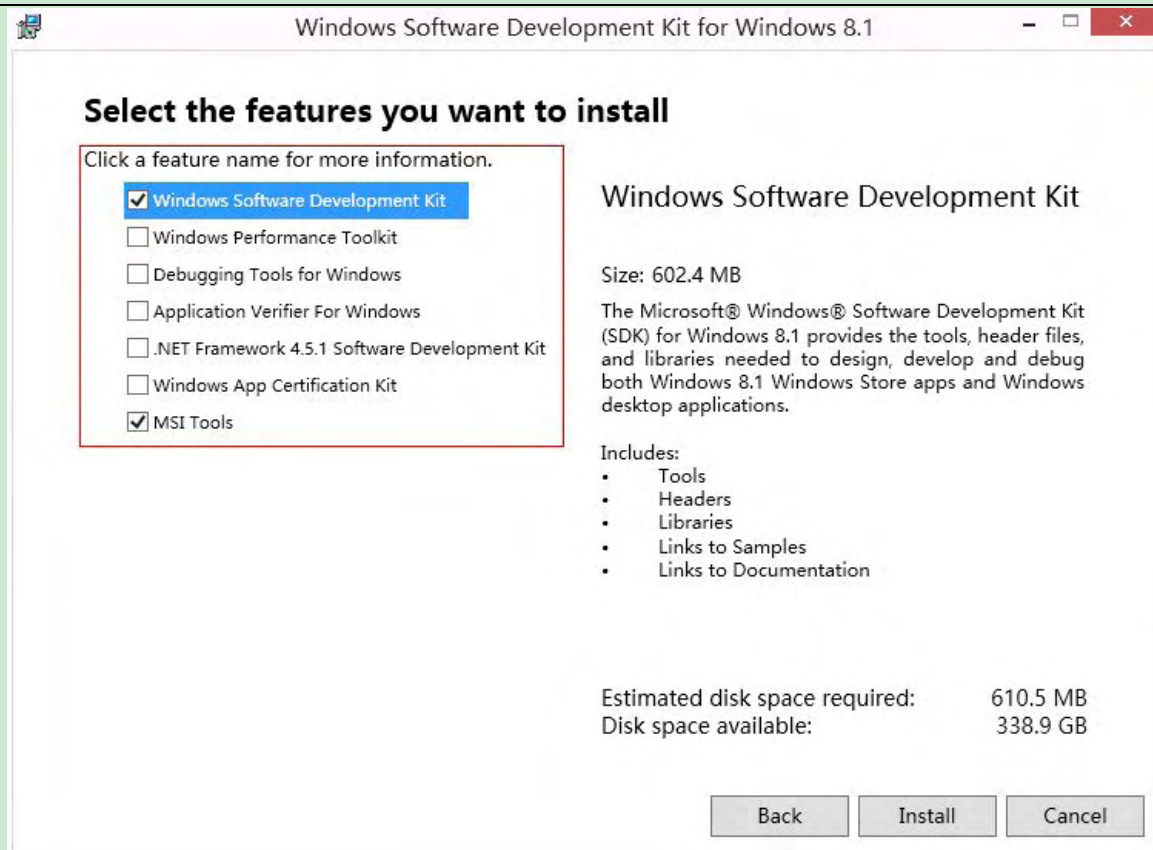
a) Windows SDK (主要是需要 DirectX sdk, 主要是捕获、Render 时用的是 DirectX)

windows sdk 介绍:

官网:

<http://msdn.microsoft.com/zh-cn/windows/bb980924.aspx>

从 Windows 7 开始, DirectX SDK 就作为 Windows SDK 的一部分包含在内。



注意：然而 webrtc 现在用的是 directx9.0 sdk。所以只要在下面地址下载：

<http://www.microsoft.com/en-us/download/details.aspx?id=6812>

b) Windows DDK（好象可以不用安装）

从 windows vista 开始，ddk 改成 wdk。

官网：<http://msdn.microsoft.com/zh-cn/windows/hardware/gg454513>

Vista 以后的版本，可以用最新的 [windows wdk 8.0](#)

如果是 windows xp，则需要 [wdk 7.1.0](#) 以前的版本。

3) 下载 depot_tools 工具

a) 先装 cygwin:

`git clone https://chromium.googlesource.com/chromium/tools/depot_tools.git`

b) 无 cygwin:

https://src.chromium.org/svn/trunk/tools/depot_tools.zip

4) 把路径设置到环境变量 PATH 中。

5) 下载 webrtc 代码，最好选择稳定代码下载，trunk 是当前开发代码库。

6) 配置下载代码库：

`E:\source\muli@google>gclient config http://webrtc.googlecode.com/svn/trunk/`

这一步主要下载 git、svn、python 和配置文件.gconfig。

默认配置下载与平台相应的代码，如果要下其它平台代码。修改.gconfig 文件，加入 target_os =


```
['windows', 'android', 'unix']
```

7) 设置产生者和版本号。此步可选

```
E:\source\muli\google\trunk>set GYP_GENERATORS=msvs #设置产生者 (可选)
```

指定工程文件类型, 如果没有这一步, 默认使用 ninja

make for Makefiles

msvs for Visual Studio

msvs-ninja for Visual Studio project building with ninja

xcode for Xcode

(Windows: ninja/Visual Studio, OS X: ninja/XCode, Linux: ninja/make, Android: ninja)

```
E:\source\muli\google\trunk>set GYP_MSVS_VERSION=2012 #设置 vs 产生者版本 (可选)
```

8) 同步源代码。

```
E:\source\muli\google>gclient sync --force #同步源码
```

更新 depot_tools 工具、git、svn、python 工具、下载 webrtc 代码及相关工具, 有 1G 多大小。注意: 如果下载中卡住了, 需要翻墙。

这步完成时, 会自动调用 gyp 产生工程。如果没有设置前面两步, 则默认的为 ninja 工程。

(Windows: ninja/Visual Studio, OS X: ninja/XCode, Linux: ninja/make, Android: ninja)

如果用默认工程, 就可以直接到第 10) 操作。

9) 重新产生工程

```
E:\source\muli\google>gclient runhooks --force #运行 hooks, 重新产生工程
```

10) 手工生成工程

```
E:\source\muli\google\trunk>python build\gyp_chromium.py --depth . -G msvs_version=2012  
all.gyp #手工产生指定版本的工程
```

在 trunk 目录下生 all.sln

-G msvs_version :vs 的版本号。产生者由环境变量 **GYP_GENERATORS** 设置

--generator-output : 工程产生目录, 但是生成出来的工程有问题, 所以不选。

Msvs 版本: 建议用最新的版本号, 因为开发者用的是最近的版本, 所以老版本的兼容性可能会有问题。

11) 编译:

用 vs2012 打开工程文件 all.sln。

用 ninja 编译

```
ninja -C out/Debug All
```

12) 编译时可能出现的问题:

问题一:

```
_____ running 'C:\openmeetings\google\depot_tools\python_bin\python.exe trunk  
/webrtc/build/gyp_webrtc -Dextra_gyp_flag=0' in 'C:\openmeetings\google'  
ERROR at //build/config/win/visual_studio_version.gni:33:24: Script returned non  
-zero exit code.
```

```
visual_studio_path = exec_script("get_visual_studio_path.py",  
^-----
```

Current dir: C:\openmeetings\google\trunk\out\gn_build.Debug64\

Command: "C:\openmeetings\google\depot_tools\python_bin\python.exe" "C:\openmeetings\google\trunk\build\config\win\get_visual_studio_path.py" auto
Returned 1 and printed out:

原因是由于没有安装 visual studio 引起的。解决方法是安装 vs2012。

问题三:

```
E:\source\muli\google\trunk>python build\gyp_chromium.py --depth . -G msvs_version=2012 all.gyp
```

Traceback (most recent call last):

```
File "build\gyp_chromium.py", line 18, in <module>
```

```
    execfile(os.path.join(path, 'gyp_chromium'))
```

```
File "E:\source\muli\google\trunk\build\gyp_chromium", line 42, in <module>
```

```
    import find_depot_tools
```

ImportError: No module named find_depot_tools

原因是开发人员把这个模块给删掉了。但不知道为什么这么明显的 BUG 却没有测试到并修复? 也许是开发人员从不用第 10 步生成工程。Webrtc 代码处理开发阶段, 很不稳定。在项目管理方面比其它开源项目 (例如: ffmpeg、vlc、libnice) 差很多。当然相应不久将来, 这个 BUG 也许会被修复。

解决方法:

在 gyp_chromium.py 42 行前加入下面行。

```
sys.path.insert(1, os.path.join(chrome_src, 'webrtc', 'build'))
```

从下面地址下载 find_depot_tools 工具到 trunk\webrtc\build 下:

https://github.com/adobe/chromium/blob/master/tools/find_depot_tools.py

问题二:

用 vs2008 编译, 出现下面问题:

```
3>. \video_coding\main\source\receiver.cc(159) : error C2668: "abs": 对重载函数的调用不明确
```

```
3>          C:\Program Files\Microsoft Visual Studio 9.0\VC\include\math.h(539): 可能是 "long double abs(long double)"
```

```
3>          C:\Program Files\Microsoft Visual Studio 9.0\VC\include\math.h(491): 或 "float abs(float)"
```

```
3>          C:\Program Files\Microsoft Visual Studio 9.0\VC\include\math.h(487): 或 "double abs(double)"
```

```
3>          C:\Program Files\Microsoft Visual Studio 9.0\VC\include\math.h(485): 或 "long abs(long)"
```

```
3>          C:\Program Files\Microsoft Visual Studio 9.0\VC\include\stdlib.h(380): 或 "int abs(int)"
```

```
3>          试图匹配参数列表 "(const int64_t)" 时
```

```
3>. \video_coding\main\source\receiver.cc(164) : error C2668: "abs": 对重载函数的调用不明确
```

```
3>          C:\Program Files\Microsoft Visual Studio 9.0\VC\include\math.h(539): 可能是 "long double abs(long double)"
```

```
3>          C:\Program Files\Microsoft Visual Studio 9.0\VC\include\math.h(491): 或 "float abs(float)"
```

```
3>          C:\Program Files\Microsoft Visual Studio 9.0\VC\include\math.h(487): 或 "double abs(double)"
```

```
3>          C:\Program Files\Microsoft Visual Studio 9.0\VC\include\math.h(485): 或 "long abs(long)"
```

```
3>          C:\Program Files\Microsoft Visual Studio 9.0\VC\include\stdlib.h(380): 或 "int abs(int)"
```

以上问题是由于 vs2008 对 C99 支持不够, 用最新版本 vs2012 就可以解决。也可以修改代码进行类型强

制转换。

2.1.2 ubuntu 下编译:

1) 安装 depot_tools:

```
svn co http://src.chromium.org/svn/trunk/tools/depot\_tools
```

或者:

```
git clone https://chromium.googlesource.com/chromium/tools/depot\_tools.git
```

我的 depot_tools 下到了 /data/google/depot_tools 中。

2) 设置环境变量, 把这个目录加入到 PATH 中:

```
export PATH=$PATH:/data/google/depot_tools
```

3) 下载 webrtc 代码, 最好选择稳定代码下载, trunk 是当前开发代码库。

```
gclient config http://webrtc.googlecode.com/svn/trunk/ (生成 gconfig 文件)
```

默认配置下载与平台相应的代码, 如果要下其它平台代码. 修改 gconfig 文件, 加入 target_os = ['windows', 'android', 'unix']

gclient sync --force (同步项目文件, 要下载 1 个多 G 的文件, 网速不好的, 可以去玩一会再回来), 注意: 如果下载中卡住了, 需要翻墙。

gclient runhooks --force (重新生成相应的工程文件, Linux 的 MakeFile 文件)

4) 安装依赖开发库:

```
apt-get install libasound2-dev
```

```
apt-get install libpulse-dev
```

```
apt-get install libx11-dev
```

```
apt-get install libxext-dev
```

```
apt-get install libnss3-dev
```

5) 生成工程文件

a) 生成 make 工程

```
k@k-C410:/data/google/webrtc/trunk$ export GYP_GENERATORS=make
```

指定工程文件类型, 如果没有这一步, 默认使用 ninja

make for Makefiles

msvs for Visual Studio

msvs-ninja for Visual Studio project building with ninja

xcode for Xcode

```
k@k-C410:/data/google/webrtc/trunk$ build/gyp_chromium --depth=. all.gyp
```

如果你没有安装依赖库, 可能会出现下面错误:

```
Updating projects from gyp files...
```

```
Package nss was not found in the pkg-config search path.
```

```
Perhaps you should add the directory containing `nss.pc'
```

```
to the PKG_CONFIG_PATH environment variable
```

```
No package 'nss' found
```

```
gyp: Call to 'pkg-config --libs-only-L --libs-only-other nss' returned exit status 1. while loading dependencies of all.gyp while trying to load all.gyp
```

安装 libnss 库:

```
k@k-C410:/data/google/webrtc/trunk$ sudo apt-get install libnss3-dev
```

```
_____ running '/usr/bin/python trunk/webrtc/build/gyp_webrtc -Dextra_gyp_flag=0' in '/home/google'
Generating gyp files from GN...
Updating projects from gyp files...
Package gconf-2.0 was not found in the pkg-config search path.
Perhaps you should add the directory containing `gconf-2.0.pc'
to the PKG_CONFIG_PATH environment variable
No package 'gconf-2.0' found
gyp: Call to 'pkg-config --cflags gconf-2.0' returned exit status 1.
Error: Command /usr/bin/python trunk/webrtc/build/gyp_webrtc -Dextra_gyp_flag=0 returned non-zero exit
status 1 in /home/google
安装 gconf-2.0 库:
k@k-C410:/home/google$ sudo apt-get install gconf-2.0
```

然后再生成编译工程:

```
k@k-C410:/data/google/webrtc/trunk$ build/gyp_chromium --depth=. all.gyp
```

在当前目录下产生 Makefile 文件。

编译:

```
k@k-C410:/data/google/webrtc/trunk$ make peerconnection_server
k@k-C410:/data/google/webrtc/trunk$ make peerconnection_client
生成的文件放在 out 目录下。
k@k-C410:/data/google/webrtc/trunk/out/Debug$ ls
```

genmacro	libvpx_obj_int_extract	obj.target	re2c
genmodule	libyuv.a	peerconnection_client	yasm
genperf	linker.lock	peerconnection_server	
genstring	obj	protoc	
genversion	obj.host	pyproto	

你可以看到:

peerconnection_client、 peerconnection_server 两个应用程序。

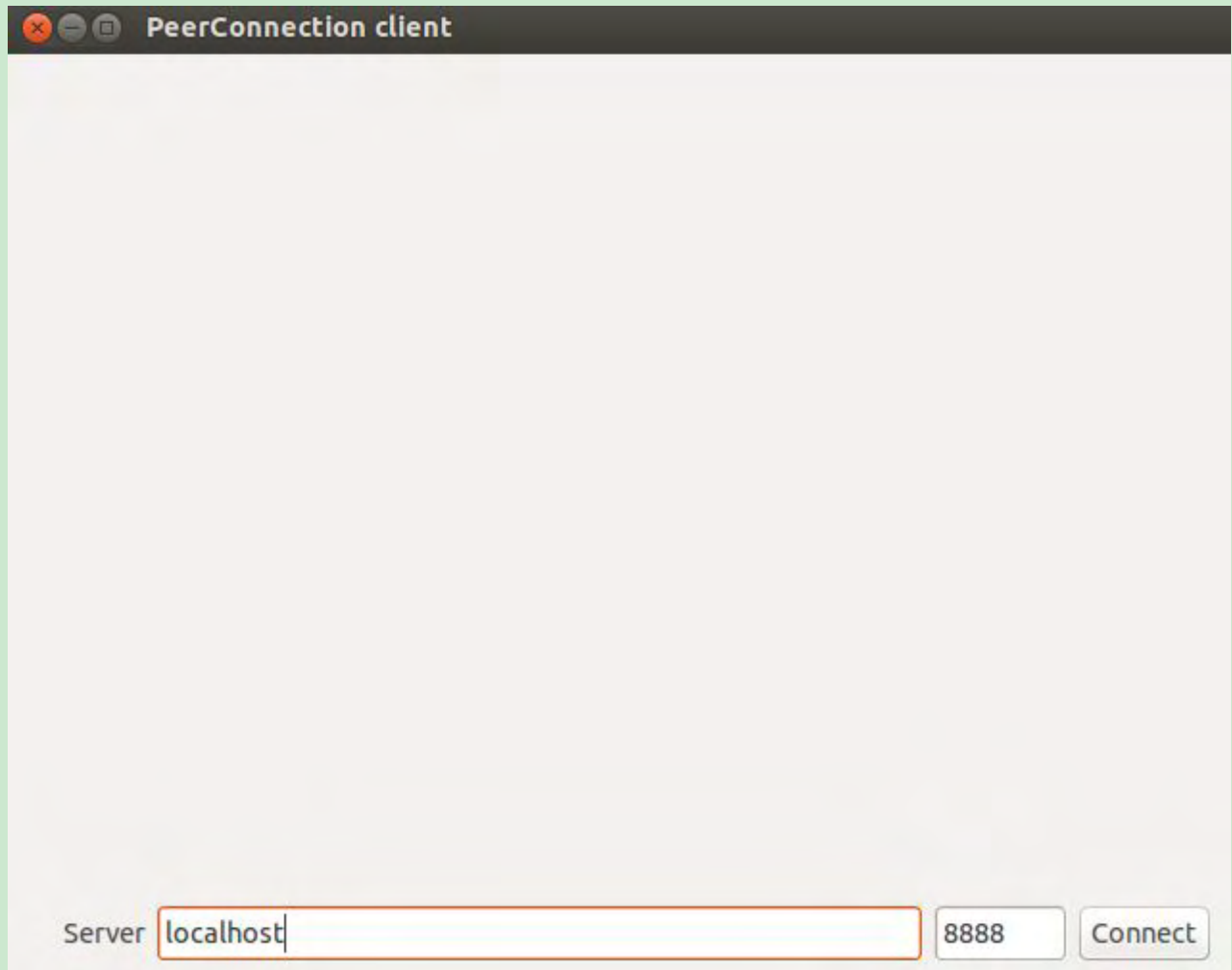
运行:

启动服务器:

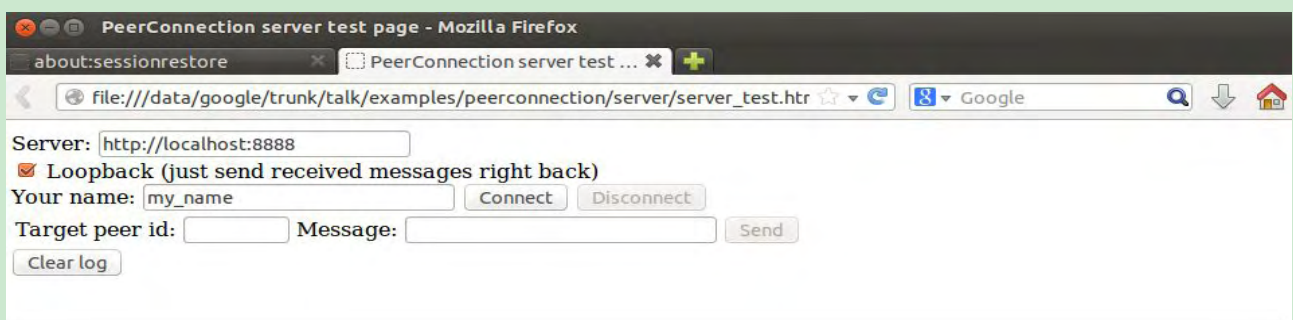
```
k@k-C410:/data/google/webrtc/trunk/out/Debug$ ./peerconnection_server
Server listening on port 8888
```

启动客户端:

```
k@k-C410:/data/google/webrtc/trunk/out/Debug$ ./peerconnection_client
```



也可以用 trunk/talk/examples/peerconnection/server/server_test.html 目录下的页面进行测试。



a) 生成默认工程，默认为 ninja

```
k@k-C410:/data/google/webrtc/trunk$ build/gyp_chromium --depth=. all.gyp
```

如果你没有安装依赖库，可能会出现下面错误：

```
Updating projects from gyp files...
```

```
Package nss was not found in the pkg-config search path.
```

```
Perhaps you should add the directory containing `nss.pc'
to the PKG_CONFIG_PATH environment variable
```

```
No package 'nss' found
```

```
gyp: Call to 'pkg-config --libs-only-L --libs-only-other nss' returned exit status 1. while loading dependencies of
all.gyp while trying to load all.gyp
```

安装 libnss 库:

```
k@k-C410:/data/google/webrtc/trunk$ sudo apt-get install libnss3-dev
```

然后再生成编译工程:

```
k@k-C410:/data/google/webrtc/trunk$ build/gyp_chromium --depth=. all.gyp
```

在当前目录下产生 out 目录, 在 out 目录中有 Debug、Release 两个子目录, 在子目录中有 ninja 工程文件 build.ninja

编译指定的目标:

```
k@k-C410:/data/google/webrtc/trunk$ ninja -C out peerconnection_server
```

```
k@k-C410:/data/google/webrtc/trunk$ ninja -C out peerconnection_client
```

编译所有工程目标:

```
k@k-C410:/data/google/webrtc/trunk$ ninja -C out All
```

在 Debug 下可以看到许多的测试程序。

2.1.3 编译 Android(只能在 linux 下):

设置环境变量: ANDROID_NDK_ROOT、ANDROID_SDK_ROOT、JAVA_HOME

```
k@k-C410:/data/google /webrtc/trunk$ export JAVA_HOME=/data/jdk1.7.0_45
```

```
k@k-C410:/data/google/webrtc/trunk$ export ANDROID_SDK_ROOT=/data/adt-bundle-linux-x86_64-20130917/sdk
```

```
k@k-C410:/data/google/webrtc/trunk$ export ANDROID_NDK_ROOT=/data/android-ndk-r9
```

```
k@k-C410:/data/google/webrtc/trunk$ source build/android/envsetup.sh
```

```
k@k-C410:/data/google/webrtc/trunk$ build/gyp_chromium --depth=. all.gyp #此步可省
```

```
k@k-C410:/data/google/webrtc/trunk$ ninja -C out/Debug All #此步可省
```

在编译过程中会有些类实例调用静态方法的警告错误。需要提示修改代码。

生成所有程序。

```
k@k-C410:/data/google/webrtc/trunk/out/Debug$ ls *.apk
```

```
AppRTCDemo-debug.apk  OpenSslDemo-debug.apk  WebRTCDemo-debug.apk
```

在 Debug 下可以看到许多的测试程序。

3 webrtc 开发:

在编译环境搭建完成后, 可以开始干活了。我们用 webrtc 的主要目的是应用它的音 / 视频捕获、视频显示、音频播放、音视频压缩、网络通信。如果你只是想在 HTML5 中用音 / 视频解决方案, 可以跳过本教程, 因为 webrtc 的最终目标是实现 HTML5 的音视频解决方案。本教程主要讲解在应用程序中如何使用 webrtc 库。

3.1 开发 P2P 视频软件需要处理的问题

3.1.1 用户列的获取、交换、信令的交换

XMPP 协议主要用于解决获取用户列表、交换用户数、信令交换。

3.1.2 P2P 通信

现实网络环境有三种情况:

1. 公共网络: 这类网络 IP 之间可以不受限制的进行直接访问。

2. NAT 网络: 这类网络主机在私有内网中, 没有单独的公网 IP。但可以通过打洞来找到它在公网中固定的网络地址。STUN 协议就是解决些网络问题。

3. 严格的受限 NAT 网络: 这类网络中的主机在私网内, 只能单向访问外网。外网不能直接访问它。所以这类网络需要通过在公共网络上的 TURN 服务器来进行数据中转。TURN 协议就是解决此网络问题的。

为了解决地址转换、防火墙限制访问等问题。所以提供了 **ICE 协议**, ICE 协议是一个框

架，它依赖 **STUN 协议** 解决完全锥形 NAT、以及受限锥形 NAT。**TURN 协议** 用于解决严格受限 NAT 网络的问题。

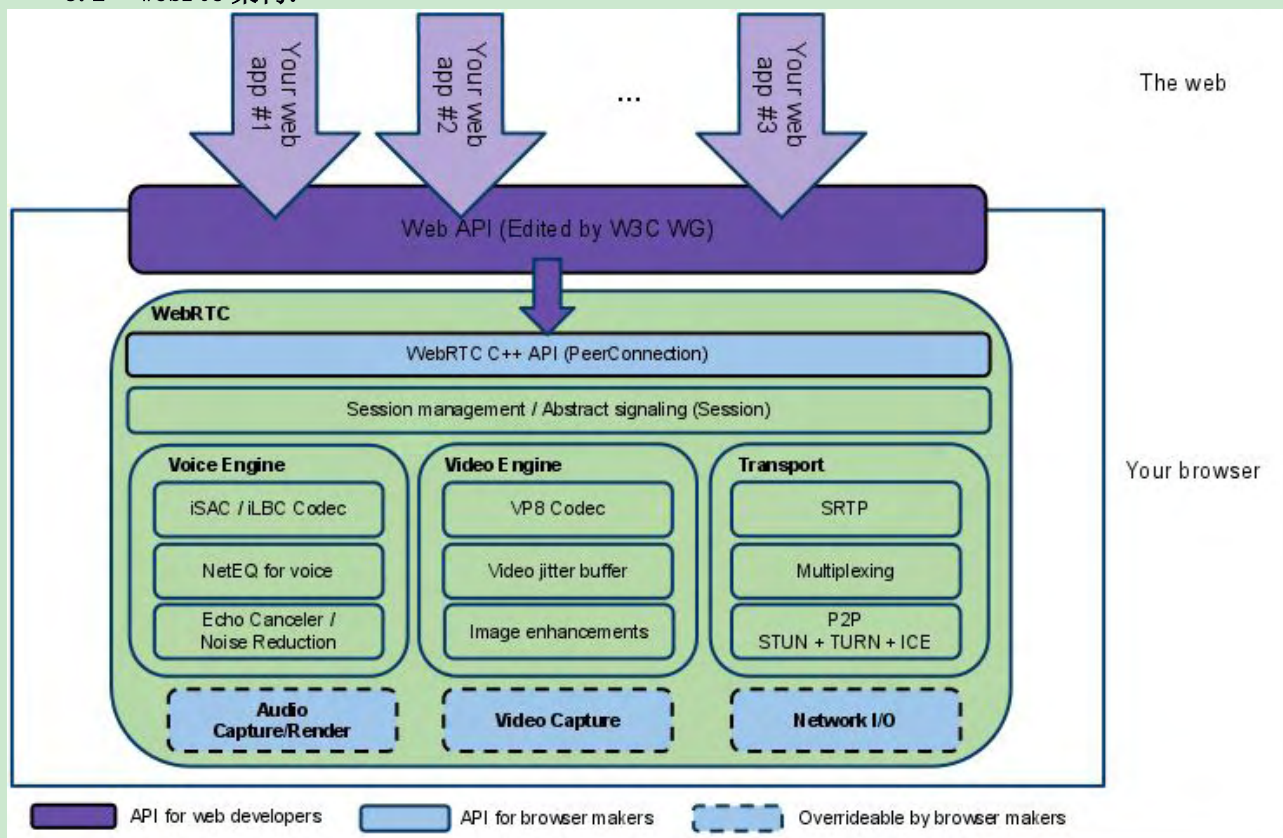
3.1.3 多媒体处理

3.1.3.1 音视频捕获、播放

3.1.3.2 音视频编解码

3.1.3.3 音视频效果优化

3.2 webrtc 架构:



(1) 紫色部分是 Web 开发者 API 层;

(2) 蓝色实线部分是面向浏览器厂商的 API 层 (本教程主要讲解的部分)

(3) 蓝色虚线部分浏览器厂商可以自定义实现

3.2.1 WebRTC 架构组件介绍

(1) 你的 web 应用程序

Web 开发者开发的程序，Web 开发者可以基于集成 WebRTC 的浏览器提供的 web API 开发基于视频、音频的实时通信应用。

(2) Web API

面向第三方开发者的 WebRTC 标准 API (Javascript)，使开发者能够容易地开发出类似于网络视频聊天的 web 应用，最新的标准化进程可以查看这里。

(3) WebRTC Native C++ API

本地 C++ API 层，使浏览器厂商容易实现 WebRTC 标准的 Web API，抽象地对数字信号过程进行处理。主要是 PeerConnection 对象。

(4) Transport / Session

传输/会话层

会话层组件采用了 libjingle 库的部分组件实现，无须使用 xmpp/jingle 协议。

参见: https://developers.google.com/talk/talk_developers_home

现在 libjingle 已经移到 webrtc 项目中维护了。位于源码根目录的 talk 目录下。

a. RTP Stack 协议栈

Real Time Protocol

b. STUN/ICE

可以通过 STUN 和 ICE 组件来建立不同类型网络间的呼叫连接。现实网络环境有三种情况:

1. 公共网络: 这类网络 IP 之间可以不受限制的进行直接访问。
 2. NAT 网络: 这类网络主机在私有内网中, 没有单独的公网 IP。但可以通过打洞来找到它在公网中固定的网络地址。STUN 协议就是解决些网络问题
 3. 严格的受限 NAT 网络: 这类网络中的主机在私网内, 只能单向访问外网。外网不能直接访问它。所以这类网络需要通过在公共网络上的服务器来进行数据中转。TRUN 协议就是解决此网络问题的。
- ICE 是一个框架, 在这个框架中, 它会判断主机是上面三种类型之一, 并用相应的办法来建立主机之间的连接。

c. Session Management

一个抽象的会话层, 提供会话建立和管理功能。该层协议留给应用开发者自定义实现。

(5) VoiceEngine

音频引擎是包含一系列音频多媒体处理的框架, 包括从视频采集卡到网络传输端等整个解决方案。

PS: VoiceEngine 是 WebRTC 极具价值的技术之一, 是 Google 收购 GIPS 公司后开源的。在 VoIP 上, 技术业界领先, 后面的文章会详细了解

a. iSAC

Internet Speech Audio Codec

针对 VoIP 和音频流的宽带和超宽带音频编解码器, 是 WebRTC 音频引擎的默认的编解码器

采样频率: 16khz, 24khz, 32khz; (默认为 16khz)

自适应速率为 10kbit/s ~ 52kbit/;

自适应包大小: 30~60ms;

算法延时: frame + 3ms

b. iLBC

Internet Low Bitrate Codec

VoIP 音频流的窄带语音编解码器

采样频率: 8khz;

20ms 帧比特率为 15.2kbps

30ms 帧比特率为 13.33kbps

标准由 IETF RFC3951 和 RFC3952 定义

c. NetEQ for Voice

针对音频软件实现的语音信号处理元件

NetEQ 算法: 自适应抖动控制算法以及语音包丢失隐藏算法。使其能够快速且高解析度地适应不断变化的网络环境, 确保音质优美且缓冲延迟最小。

是 GIPS 公司独步天下的技术, 能够有效的处理由于网络抖动和语音包丢失时候对语音质量产生的影响。

PS: NetEQ 也是 WebRTC 中一个极具价值的技术, 对于提高 VoIP 质量有明显效果, 加以 AEC\NR\AGC 等模块集成使用, 效果更好。

d. Acoustic Echo Canceler (AEC)

回声消除器是一个基于软件的信号处理元件，能实时的去除 mic 采集到的回声。

e. Noise Reduction (NR)

噪声抑制也是一个基于软件的信号处理元件，用于消除与相关 VoIP 的某些类型的背景噪声（嘶嘶声，风扇噪音等等… …）

(6) VideoEngine

WebRTC 视频处理引擎

VideoEngine 是包含一系列视频处理的整体框架，从摄像头采集视频到视频信息网络传输再到视频显示整个完整过程的解决方案。

a. VP8

视频图像编解码器，是 WebRTC 视频引擎的默认的编解码器

VP8 适合实时通信应用场景，因为它主要是针对低延时而设计的编解码器。

PS:VPx 编解码器是 Google 收购 ON2 公司后开源的，VPx 现在是 WebM 项目的一部分，而 WebM 项目是 Google 致力于推动的 HTML5 标准之一

b. Video Jitter Buffer

视频抖动缓冲器，可以降低由于视频抖动和视频信息包丢失带来的不良影响。

c. Image enhancements

图像质量增强模块

对网络摄像头采集到的图像进行处理，包括明暗度检测、颜色增强、降噪处理等功能，用来提升视频质量。

3.2.2 WebRTC 核心模块 API 介绍

(1) 网络传输模块：libjingle

WebRTC 重用了 libjingle 的一些组件，主要是 network 和 transport 组件，关于 libjingle 的文档资料可以查看[这里](https://developers.google.com/talk/talk_developers_home)。参见：https://developers.google.com/talk/talk_developers_home

现在这个模块已经放到 webrtc 中进行维护了。位于源码根目录的 talk 目录下。

(2) WebRTC Native C++ API

代码位于源码根目录的 talk\app\webrtc 目录下。webrtc 封装了 peerconnection 对象，这个对象把流媒体传输也封装进去了。

PeerConnection 类厂接口：

```
// PeerConnectionFactoryInterface is the factory interface use for creating
// PeerConnection, MediaStream and media tracks.
// PeerConnectionFactoryInterface will create required libjingle threads,
// socket and network manager factory classes for networking.
// If an application decides to provide its own threads and network
// implementation of these classes it should use the alternate
// CreatePeerConnectionFactory method which accepts threads as input and use the
// CreatePeerConnection version that takes a PortAllocatorFactoryInterface as
// argument.
```

PeerConnectionFactoryInterface

PeerConnectionInterface 接口:

peerconnection API:talk/app/webrtc/peerconnectioninterface.h

webrtc 本地 API: talk/app/webrtc/mediastreaminterface.h

(3) 音频、视频图像处理的主要数据结构 常量\VideoEngine\VoiceEngine

注意: 以下所有的方法、类、结构体、枚举常量等都在 `webrtc` 命名空间里

类、结构体、枚举常量	头文件	说明
Structures	common_types.h	Lists the structures common to the

		VoiceEngine & VideoEngine
Enumerators	common_types.h	List the enumerators common to the VoiceEngine & VideoEngine
Classes	common_types.h	List the classes common to VoiceEngine & VideoEngine
class VoiceEngine	voe_base.h	How to allocate and release resources for the VoiceEngine using factory methods in the VoiceEngine class. It also lists the APIs which are required to enable file tracing and/or traces as callback messages
class VideoEngine	vie_base.h	How to allocate and release resources for the VideoEngine using factory methods in the VideoEngine class. It also lists the APIs which are required to enable file tracing and/or traces as callback messages

(4) 音频引擎 (VoiceEngine) 模块 APIs

下表列的是目前在 *VoiceEngine* 中可用的 *sub APIs*

sub-API	头文件	说明
VoEAudioProcessing	voe_audio_processing.h	Adds support for Noise Suppression (NS), Automatic Gain Control (AGC) and Echo Control (EC). Receiving side VAD is also included.
VoEBase	voe_base.h	Enables full duplex VoIP using G.711. NOTE: This API must always be created.
VoECallReport	voe_call_report.h	Adds support for call reports which contains number of dead-or-alive detections, RTT measurements, and Echo metrics.
VoECodec	voe_codec.h	Adds non-default codecs (e.g. iLBC, iSAC, G.722 etc.), Voice Activity Detection (VAD) support.
VoEDTMF	voe_dtmf.h	Adds telephone event transmission, DTMF tone generation and telephone event detection. (Telephone events include DTMF.)
VoEEncryption	voe_encryption.h	Adds external encryption/decryption support.
VoEErrors	voe_errors.h	Error Codes for the VoiceEngine
VoEExternalMedia	voe_external_media.h	Adds support for external media processing and enables utilization of an external audio resource.
VoEFile	voe_file.h	Adds file playback, file recording and

		file conversion functions.
VoEHardware	voe_hardware.h	Adds sound device handling, CPU load monitoring and device information functions.
VoENetEqStats	voe_neteq_stats.h	Adds buffer statistics functions.
VoENetwork	voe_network.h	Adds external transport, port and address filtering, Windows QoS support and packet timeout notifications.
VoERTP_RTCP	voe_rtp_rtcp.h	Adds support for RTCP sender reports, SSRC handling, RTP/RTCP statistics, Forward Error Correction (FEC), RTCP APP, RTP capturing and RTP keepalive.
VoEVideoSync	voe_video_sync.h	Adds RTP header modification support, playout-delay tuning and monitoring.
VoEVolumeControl	voe_volume_control.h	Adds speaker volume controls, microphone volume controls, mute support, and additional stereo scaling methods.

(5) 视频引擎 (VideoEngine) 模块 APIs

下表列的是目前在 *VideoEngine* 中可用的 sub APIs

sub-API	头文件	说明
ViEBase	vie_base.h	Basic functionality for creating a VideoEngine instance, channels and VoiceEngine interaction. NOTE: This API must always be created.
ViECapture	vie_capture.h	Adds support for capture device allocation as well as capture device capabilities.
ViECodec	vie_codec.h	Adds non-default codecs, codec settings and packet loss functionality.
ViEEncryption	vie_encryption.h	Adds external encryption/decryption support.
ViEErrors	vie_errors.h	Error codes for the VideoEngine
ViEExternalCodec	vie_external_codec.h	Adds support for using external codecs.
ViEFile	vie_file.h	Adds support for file recording, file playout, background images and snapshot.
ViEImageProcess	vie_image_process.h	Adds effect filters, deflickering, denoising and color enhancement.
ViENetwork	vie_network.h	Adds send and receive functionality, external transport, port and address filtering, Windows QoS support, packet timeout notification and changes to network settings.
ViERender	vie_render.h	Adds rendering functionality.

ViERTP_RTCP	vie_rtp_rtcp.h	Adds support for RTCP reports, SSRS handling RTP/RTCP statistics, NACK/FEC, keep-alive functionality and key frame request methods.
-------------	----------------	---

3.2.3 webRTC 核心 API 详解

webRTC 本地 API: <http://www.webrtc.org/reference/native-apis>。

翻译: <http://www.cnblogs.com/longrenle/archive/2012/03/04/2378433.html>。

webRTC 本地 API 是基于 [WebRTC spec](#) 的实现。webRTC 的实现代码 (包括流和 PeerConnection API) 是在 libjingle 中实现。

线程模型

WebRTC native APIs 拥有两个全局线程: 信令线程 (signaling thread) 和工作者线程 (worker thread)。取决于 PeerConnection factory 被创建的方式, 应用程序可以提供这两个线程或者直接使用内部创建好的线程。

Stream APIs 和 PeerConnection APIs 的调用会被代理到信令线程, 这就意味着应用程序可以在任何线程调用这些 APIs。

所有的回调函数都在信令线程调用。应用程序应当尽快地跳出回调函数以避免阻塞信令线程。严重消耗资源的过程都应当其他的线程执行。

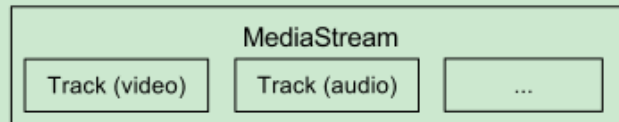
工作者线程被用来处理资源消耗量大的过程, 比如说数据流传输。

3.2.3.1 libjingle_peerconnection

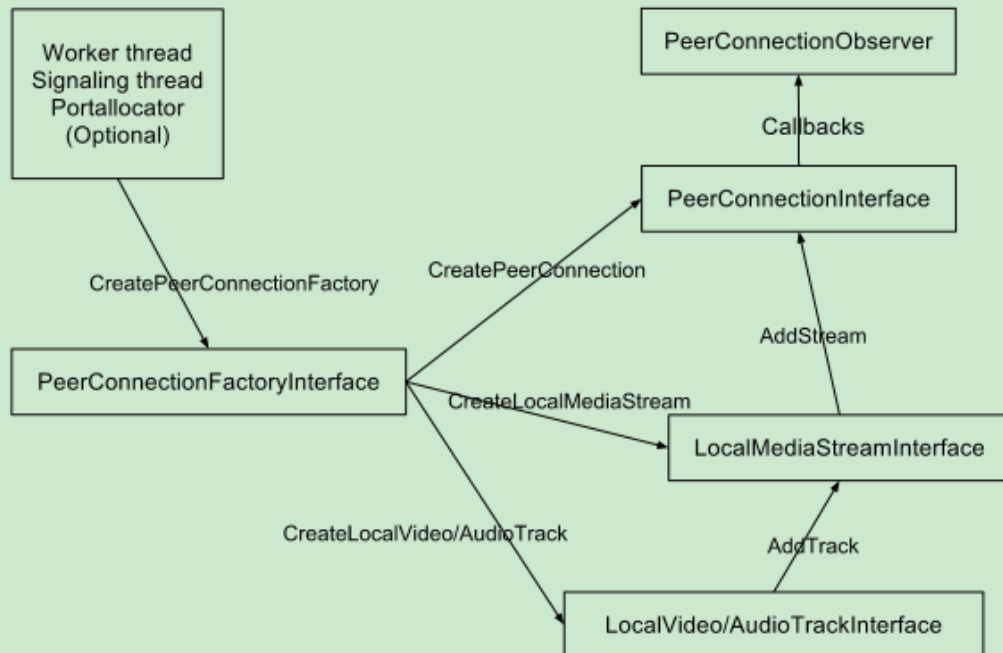
这个模块封装了对等网络连接通信过程。

块图:

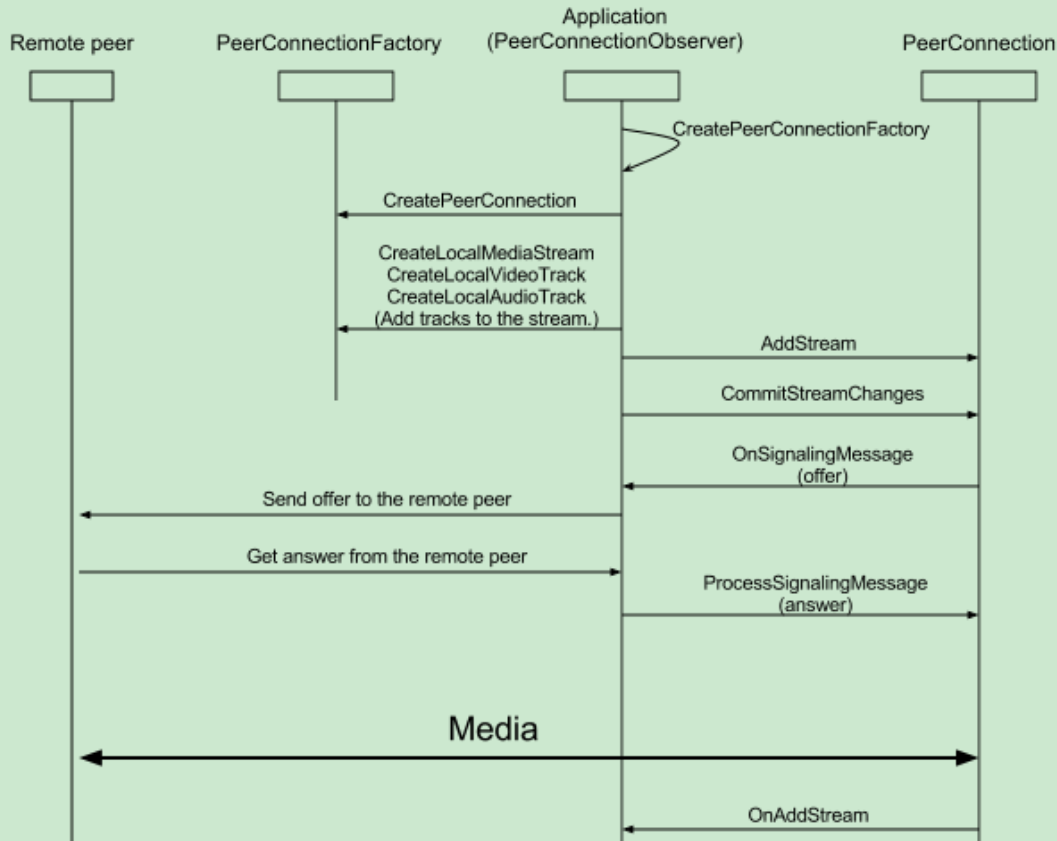
Stream



PeerConnection



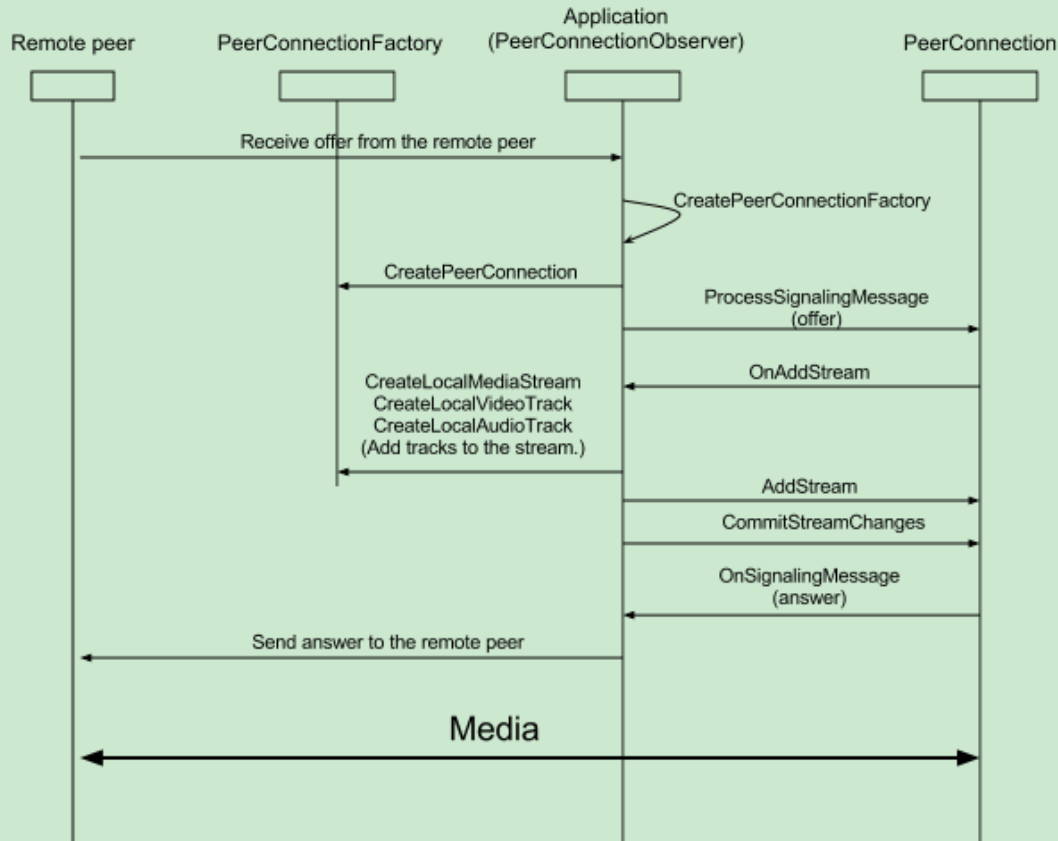
调用顺序:
安装调用:



```

// The Following steps are needed to setup a typical call using Jsep.
// 1. Create a PeerConnectionFactoryInterface. Check constructors for more
// information about input parameters.
// 2. Create a PeerConnection object. Provide a configuration string which
// points either to stun or turn server to generate ICE candidates and provide
// an object that implements the PeerConnectionObserver interface.
// 3. Create local MediaStream and MediaTracks using the PeerConnectionFactory
// and add it to PeerConnection by calling AddStream.
// 4. Create an offer and serialize it and send it to the remote peer.
// 5. Once an ice candidate have been found PeerConnection will call the
// observer function OnIceCandidate. The candidates must also be serialized and
// sent to the remote peer.
// 6. Once an answer is received from the remote peer, call
// SetLocalSessionDescription with the offer and SetRemoteSessionDescription
// with the remote answer.
// 7. Once a remote candidate is received from the remote peer, provide it to
// the peerconnection by calling AddIceCandidate.
    
```

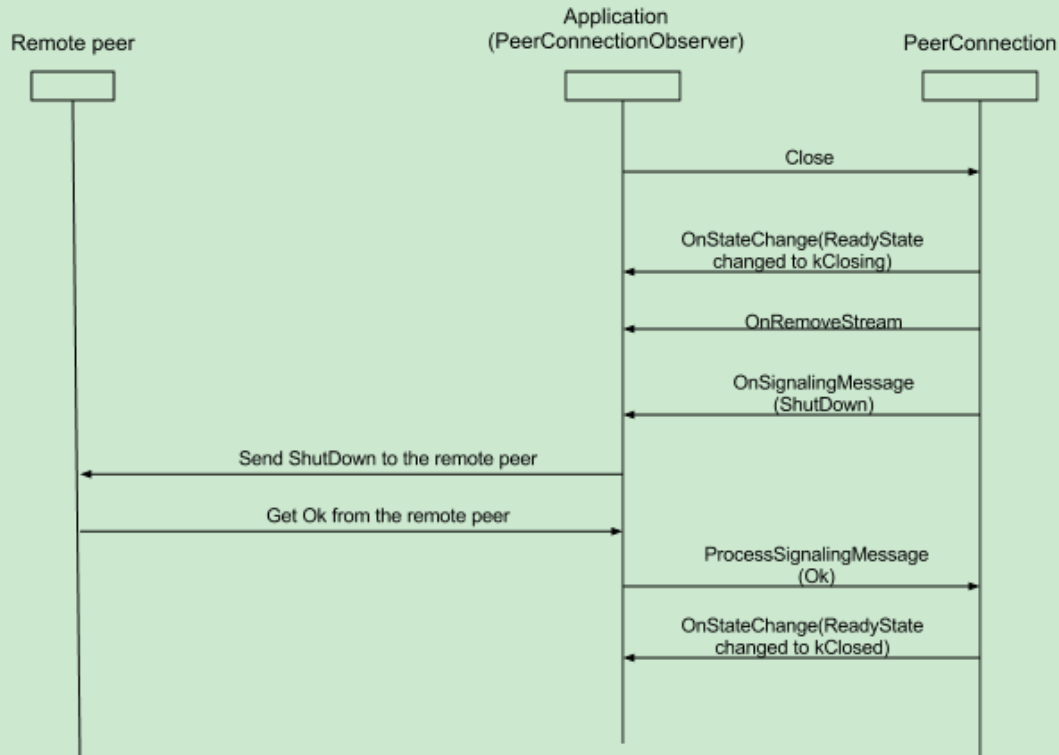
接收调用:



```

// The Receiver of a call can decide to accept or reject the call.
// This decision will be taken by the application not peerconnection.
// If application decides to accept the call
// 1. Create PeerConnectionFactoryInterface if it doesn't exist.
// 2. Create a new PeerConnection.
// 3. Provide the remote offer to the new PeerConnection object by calling
// SetRemoteSessionDescription.
// 4. Generate an answer to the remote offer by calling CreateAnswer and send it
// back to the remote peer.
// 5. Provide the local answer to the new PeerConnection by calling
// SetLocalSessionDescription with the answer.
// 6. Provide the remote ice candidates by calling AddIceCandidate.
// 7. Once a candidate have been found PeerConnection will call the observer
// function OnIceCandidate. Send these candidates to the remote peer.
    
```

关闭调用:



3.2.3.2 libjingle_media 库:

3.2.3.2.1 视频采集，处理、渲染类:

视频捕获类:

● 得到 VideoCapture 过程:

```
cricket::VideoCapturer* Conductor::OpenVideoCaptureDevice() {
    talk_base::scoped_ptr<cricket::DeviceManagerInterface> dev_manager(
        cricket::DeviceManagerFactory::Create());
    if (!dev_manager->Init()) {
        LOG(LS_ERROR) << "Can't create device manager";
        return NULL;
    }
    std::vector<cricket::Device> devs;
    if (!dev_manager->GetVideoCaptureDevices(&devs)) {
        LOG(LS_ERROR) << "Can't enumerate video devices";
        return NULL;
    }
    std::vector<cricket::Device>::iterator dev_it = devs.begin();
    cricket::VideoCapturer* capturer = NULL;
    for (; dev_it != devs.end(); ++dev_it) {
        capturer = dev_manager->CreateVideoCapturer(*dev_it);
        if (capturer != NULL)
            break;
    }
    return capturer;
}
```

```
}
```

说明:

➤ 捕获设备的建立:

建立 DeviceManagerFactory 实例。Windows 下在实例初始化时并实例化 DefaultVideoCapturerFactory。

调用 dev_manager->GetVideoCaptureDevices 得到设备。

调用 dev_manager->CreateVideoCapturer 建立一个视频捕获对象。

在 dev_manager->CreateVideoCapturer 中,

先检查是否是文件捕获对象。如果, 则返回文件捕获对象指针。

如果不是, 调用 device_video_capturer_factory->Create(device); 建立视频捕获对象。这里 device_video_capturer_factory=DefaultVideoCapturerFactory

```
class DefaultVideoCapturerFactory : public VideoCapturerFactory {
public:
    DefaultVideoCapturerFactory() {}
    virtual ~DefaultVideoCapturerFactory() {}

    VideoCapturer* Create(const Device& device) {
#ifdef VIDEO_CAPTURER_NAME
        VIDEO_CAPTURER_NAME* return_value = new VIDEO_CAPTURER_NAME;
        if (!return_value->Init(device)) {
            delete return_value;
            return NULL;
        }
        return return_value;
#else
        return NULL;
#endif
    }
};
```

这个 DefaultVideoCapturerFactory 类厂很简单, 在 Create 函数中建立 WebRtcVideoCapturer 对象。并调用 Init 初始化此对象。在这个初始化中, 会建立调用 module_ = factory_->Create(0, vcm_id); 其中 factory_ 建立过程如下:

WebRtcVcmFactory-> VideoCaptureFactory-> videocapturemodule::VideoCaptureImpl->

VideoCaptureDS

类关系详见捕获设备类:

VideoCaptureModuleV4L2: linux 下 v4l2 捕获设备。
VideoCaptureDS: windows 下 DirectXShow 捕获设备。
VideoCaptureAndroid: android 下捕获设备。

➤ 捕获数据流:

在 WebRtcVideoCapturer 中的 Start 函数中, 会把回调处理对象 VideoCaptureDataCallback 与视频捕获设备对象 VideoCaptureModule 进行关联。这样, 当视频捕获设备对象有视频帧被捕获时。会调用 VideoCaptureDataCallback 中的 OnIncomingCapturedFrame。WebRtcVideoCapturer 继承 VideoCaptureDataCallback, 所以会调用 WebRtcVideoCapturer::OnIncomingCapturedFrame。在其中调用 SignalFrameCaptured(this, &frame); 发送捕获信息。会调用基类 VideoCapturer::OnFrameCaptured。在这里会对视频格式由 BITMAP 转换为 I420。并且会调用已注册的 VideoProcessors 对象。还会

SignalVideoFrame(this, &i420_frame);发送视频帧信号。

- 渲染类:

GdiVideoRenderer: windows gdi 渲染

GtkVideoRenderer: GTK 库渲染, 这个用在 linux 平台。

4 Libjingle 详细介绍

Libjingle 现已转到 webrtc 项目中维护了。位于源码 talk 目录中。

4.1 重要组件:

- 4.1.1 信号:
- 4.1.2 线程和消息
- 4.1.3 名称转换
- 4.1.4 Ssl 支持
- 4.1.5 连接

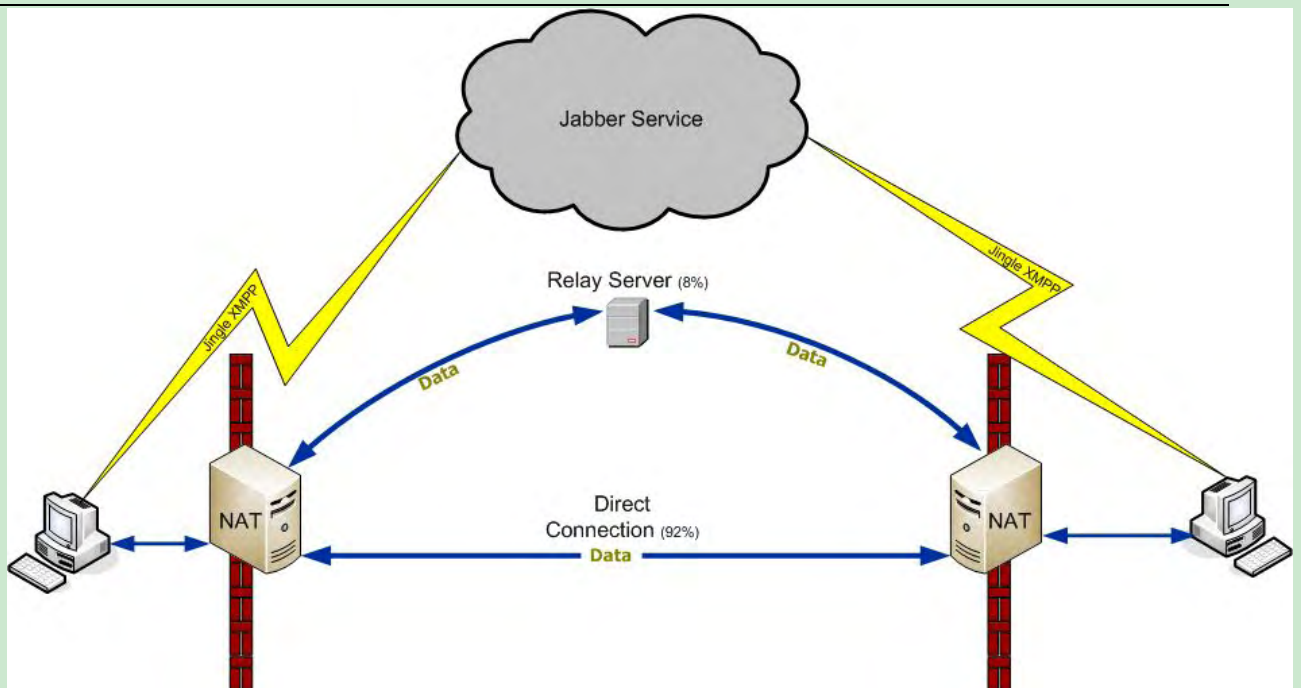
一对 libjingle 的对等连接实际上是由两个通道组成:

会话协商通道 (也称为信令信道) 是用于协商数据连接的通信链路。此通道用于请求连接, 交换人选, 并协商会话的详细信息 (如套接字地址, 需要编解码器, 以交换文件, 连接变更请求, 并终止请求)。这是计算机之间进行的第一次连接, 并且仅在由该连接可以将数据信道来建立。对 libjingle 采用先导, 以 jingle 指定要建立数据连接所需的节和响应 (见 jingle 和对 libjingle), 该通道通过中介 XMPP 服务器发送节; 示例代码中使用了谷歌 Talk 服务器作为中介。

数据通道是在点对点会话之间交换实际数据 (音频、视频、文件等)。数据通道的数据依赖于传输协商, 被捆绑在 TCP 或 UDP 包中。并且不经过 XMPP 服务器。

首先会话协商信道被建立, 作为计算机协商数据信道的细节; 数据连接后, 大部分活动发生在数据信道, 除非一个编解码器的改变, 一个新的文件请求, 偶尔请求重定向请求或终止请求。

下图显示了这两种途径。在该图中, 两个备用数据路径被示出, 虽然只有一个数据通路将被激活, 在一个连接。这是因为数据通路可以是直接连接 (连接尝试的 92%, 可以直接发生), 或者通过中继服务器 (连接尝试的 8% 所需要的中介中继服务器)。第三个数据通路, 没有显示, 是从计算机到计算机的直接连接时, 有没有中介防火墙。



注意事项:

对 libjingle 发出偶尔 STUN 数据包, 以保持可写性, 保持防火墙和 NAT 地址绑定活跃, 并检查连接延迟。

对 libjingle 分配一个用户名和密码来连接端口。这确保了连接上的数据信道的计算机是同一个, 协商通过信令信道的连接。由于这些用户名和密码值发送 XMPP 协议可能会或可能不会使用 TLS 进行加密, 在 STUN 包这些值仅作识别, 未加密的身份验证使用。

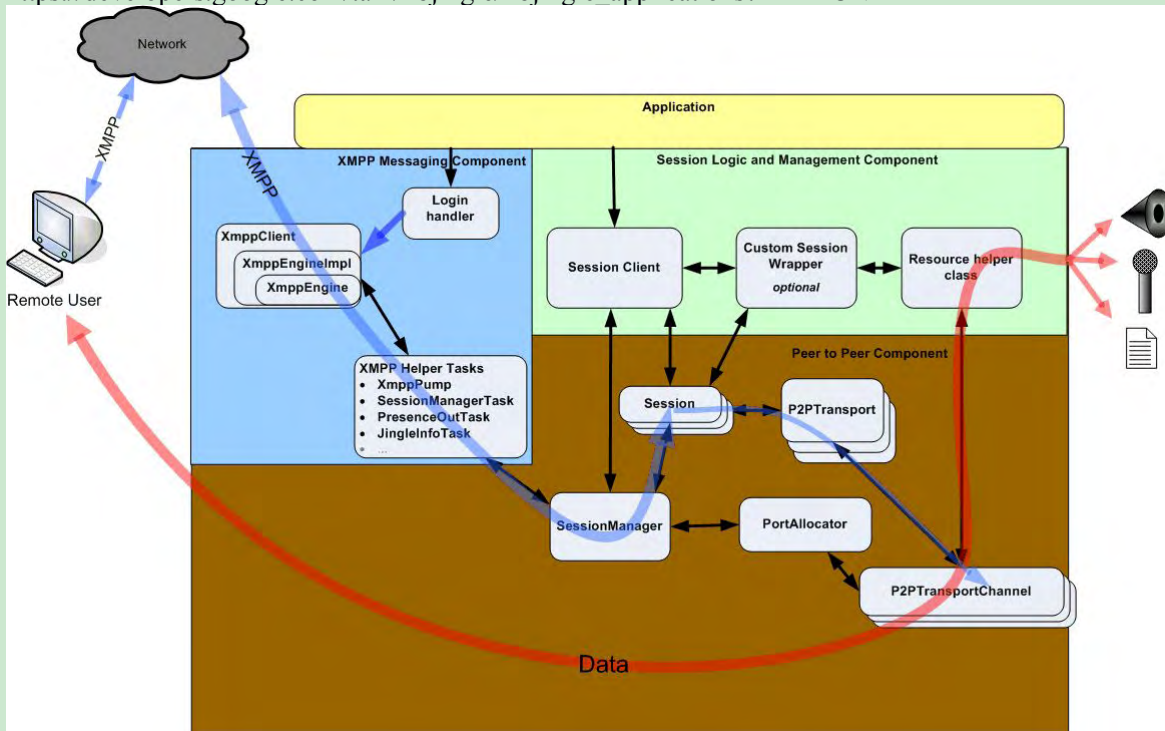
4.1.6 传输, 通道, 连接

4.1.7 候选项

4.1.8 数据包

4.2 如何工作:

https://developers.google.com/talk/libjingle/libjingle_applications?hl=zh-CN



4.2.1 Application 模块

Libjingle 的应用程序首先调用 XMPP Messaging Component 的 XmppClient 对象进行登录，然后做一些 message, iq, presence 等 request/respond 操作。

其次，每个 application 可能包含一个或多个 session client 用来做 P2P 操作，比如远程协助，视频会议，音频连接，文件共享等等。

4.2.2 XMPP Messaging Component 模块

此模块主要由三个部分组成：XmppClient, LoginHandler 和 Xmpp Helper Task. 此模块主要做相当于一个 peer 的防火墙的功能，连接服务器和客户端，负责发送所有本地的 stanza 请求（即 XMPP 协议内容），并负责接收服务器的 stanza 请求，并分发到各个 Helper Task 里。

- XmppClient 主要是代理登录，发送 stanza，接收 stanza。之所以说是代理，是因为真正发送，接收的消息都是通过 XmppEngine 来实现的。
- XmppEngine 能注册多个 XmppStanzaHandler 回调，然后所有从服务器接收的 Stanza 都转发到已绑定的 XmppStanzaHandler 进行过滤，而实际上是回调 XmppTask 对象。
- XmppStanzaHandler 类定义如下：

```
1.  //!< Callback to deliver stanzas to an Xmpp application module.
2.  //!< Register via XmppEngine.SetDefaultSessionHandler or via
3.  //!< XmppEngine.AddSessionHandler.
4.  class XmppStanzaHandler {
5.  public:
6.      virtual ~XmppStanzaHandler() {}
7.      //!< Process the given stanza.
8.      //!< The handler must return true if it has handled the stanza.
9.      //!< A false return value causes the stanza to be passed on to
10.     //!< the next registered handler.
11.     virtual bool HandleStanza(const XmlElement * stanza) = 0;
12. };
```

- XmppTask 是所有 XmppHelperTask 的基类，并继承自 XmppStanzaHandler，主要有监听，过滤 XmppEngine 对象转发过来的 Stanza 消息。XmppTask 有多种类型，当取类型为 HL_PEEK 时，只有监听功能，无法做到过滤；而其他类型可以做到过滤。过滤是有 HandlerStanza 函数来完成，当返回为 true 时，过滤，否则 XmppEngine 枚举下一个绑定的 XmppTask 继续尝试分发、过滤。
- 所有 XmppHelperTask 都要继承 XmppTask 并要重载 HandlerStanza 函数和 ProcessStart 函数。
 - HandlerStanza 是用来过滤，相当于 windows 消息处理的 GetMessage()
 - 而 ProcessStart 是用来处理 HandlerStanza 过滤的消息。
 - 比如在源代码 example/call/presencepushtask.h 里：

[cpp] [view plaincopy](#)

```
1.  class PresencePushTask : public XmppTask {
2.  public:
3.      PresencePushTask(XmppTaskParentInterface* parent, CallClient* client)
4.      : XmppTask(parent, XmppEngine::HL_TYPE),
5.        client_(client) {}
6.      virtual int ProcessStart();
7.
8.      sigslot::signal1<const Status&> SignalStatusUpdate;
9.      sigslot::signal1<const Jid&> SignalMucJoined;
10.     sigslot::signal2<const Jid&, int> SignalMucLeft;
11.     sigslot::signal2<const Jid&, const MucStatus&> SignalMucStatusUpdate;
12. }
```

```
13. protected:
14.     virtual bool HandleStanza(const XmlElement * stanza);
15.     void HandlePresence(const Jid& from, const XmlElement * stanza);
16.     void HandleMucPresence(buzz::Muc* muc,
17.                             const Jid& from, const XmlElement * stanza);
18.     static void FillStatus(const Jid& from, const XmlElement * stanza,
19.                             Status* status);
20.     static void FillMucStatus(const Jid& from, const XmlElement * stanza,
21.                             MucStatus* status);
22.
23. private:
24.     CallClient* client_;
25. };
```

- 这里 PresencePushTask 类, 通过 HandleStanza 过滤所有 presence 相关的 stanza 并在 ProcessStart 里处理所有来自服务器的用户状态更新消息。
- LoginHandler 部分是由 XmppPump 来负责的。主要调用 XmppClient 的 connect 和 disconnect 方法建立、断开连接, 监听 SignalStateChange 事件来获取连接信息, 类型为 STATE_OPENED 的事件表示连接成功。

4.2.3 Session Logic and management component 模块。

所有 p2p session 逻辑相关的部分都放在了这个模块。可以 session 可能是处理文件传输的连接, 或者可能是视频会话, 或者音频会话等等。

- 我们需要继承 SessionClient 来处理每个 Session 相关具体任务, 比如文件传输 Session: 当接收对端客户端建立一个文件传输 session 的时候, 如果此 Session 是新创建的, SessionManager 对象会回调所有注册的 SessionClient 的 OnSessionCreate 的接口, 并以 SessionManger 创建的 Session 对象为参数穿进去;如果是已有的 Session 则会调用 Session 的 OnIncomingMessage 方法。
- Session 对象则抽象了两个 peer 之间的数据传输接口。当收到 OnSessionCreate 回调时, SessionClient 可以通过 Session 的方法 Accept 来接受创建, Reject 来拒绝。
- 那怎么读写 p2p 数据呢?
 - 首先需要调用 session 的 CreateChannel 方法获取 TransportChannel 对象指针, 然后监听 TransportChannel 的事件 SignalReadPacket 来接收数据, 通过 SendPacket 方法来发送数据。

```
1. class TransportChannel: public sigslot::has_slots<> {
2. public:
3.     //.....
4.
5.     // Attempts to send the given packet. The return value is < 0 on failure.
6.     virtual int SendPacket(const char *data, size_t len) = 0;
7.     // Signalled each time a packet is received on this channel.
8.     sigslot::signal3<TransportChannel*, const char*, size_t> SignalReadPacket;
9.     //.....
10. };
```

4.2.4 Peer to peer Component 模块。

此模块才是 libjingle 核心, libjingle 项目的初衷也是能够把模块设计得完美, 使得所有需要通过 P2P 传输数据的应用层调用 libjingle 时, 不用担心数据传输的稳定性, 可靠性, 高效性。

- 刚才上面提到, 当服务器发送 stanza 时 XmppEngine 把 Stanza 发送到 XmppTask 过滤, 在这个模块, SessionManagerTask 代理 SessionManager 过滤 session 相关的 stanza, 并转发到 SessionManager 对象, 如下:

```
1. class SessionManagerTask : public buzz::XmppTask {
2. public:
3.     .....
4.
5.     virtual int ProcessStart() {
6.         const buzz::XmlElement *stanza = NextStanza();
7.         if (stanza == NULL)
8.             return STATE_BLOCKED;
9.         session_manager_ ->OnIncomingMessage(stanza);
10.        return STATE_START;
11.    }
12.
13. protected:
14.     virtual bool HandleStanza(const buzz::XmlElement *stanza) {
15.         if (!session_manager_ ->IsSessionMessage(stanza))
16.             return false;
17.         QueueStanza(stanza);
18.         return true;
19.     }
20.
21.
22. } // namespace cricket
```

- SessionManager 类在这里起到连接上述 3 个模块的桥梁作用。
- 当上层调用 SessionManager 创建的 Session 对象的 CreateChannel 时，实际上是调用 P2PTransport 的 CreateChannel 方法。
- 上层通过 P2PTransport 创建 P2pTransportChannel 的类的。
- P2pTransportChannel 继承自 TransportChannel，并创建多个不同的 Connection，每个 Connection 代表一个 TCP 或者 UDP 或者 SSL 连接。上层传输数据最终是调到 P2pTransportChannel 的相关方法，当发送，接收数据时，P2pTransportChannel 选择表现最好的 Connection 进行传输。

4.2.5 其他

LigJingle 提供了很多接口供我们继承，用于特定的个性化 Session，同时也提供了不少实例(如 pcp, login, call) 让调用者更容易的理解框架思路。当需要着手研究 libjingle 时，如果能够充分的利用已成的实例，对于缩短熟悉时间，很有帮助。

4.3 建立 libjingle 应用程序

5 代码分析:

5.1 音频通道建立过程:

```
peerconnection_client.exe!webrtc::RtpReceiverImpl::RtpReceiverImpl(int id=65536, webrtc::Clock *
clock=0x017d4e68, webrtc::RtpAudioFeedback * incoming_audio_messages_callback=0x0417a7d4,
webrtc::RtpFeedback * incoming_messages_callback=0x0417a7c4, webrtc::RTPPayloadRegistry *
rtp_payload_registry=0x0417d060, webrtc::RTPReceiverStrategy * rtp_media_receiver=0x0417d338) 行 91
C++
```

```
peerconnection_client.exe!webrtc::RtpReceiver::CreateAudioReceiver(int id=65536, webrtc::Clock *
clock=0x017d4e68, webrtc::RtpAudioFeedback * incoming_audio_feedback=0x0417a7d4, webrtc::RtpData *
incoming_payload_callback=0x0417a7c0, webrtc::RtpFeedback * incoming_messages_callback=0x0417a7c4,
webrtc::RTPPayloadRegistry * rtp_payload_registry=0x0417d060) 行 61 + 0x47 字节 C++
```

```
peerconnection_client.exe!webrtc::voe::Channel::Channel(int channelId=0, unsigned int instanceId=1,
const webrtc::Config & config={...}) 行 931 + 0x1e7 字节 C++
peerconnection_client.exe!webrtc::voe::Channel::CreateChannel(webrtc::voe::Channel * &
channel=0xcccccccc, int channelId=0, unsigned int instanceId=1, const webrtc::Config & config={...}) 行 753 +
0x2a 字节 C++
peerconnection_client.exe!webrtc::voe::ChannelManager::CreateChannelInternal(const webrtc::Config &
config={...}) 行 64 + 0x1f 字节 C++
peerconnection_client.exe!webrtc::voe::ChannelManager::CreateChannel(const webrtc::Config &
external_config={...}) 行 59 + 0x10 字节 C++
peerconnection_client.exe!webrtc::VoEBaseImpl::CreateChannel(const webrtc::Config & config={...})
行 552 C++
peerconnection_client.exe!cricket::WebRtcVoiceEngine::CreateVoiceChannel(cricket::VoEWrapper *
voice_engine_wrapper=0x01a572f8) 行 1652 + 0x23 字节 C++
peerconnection_client.exe!cricket::WebRtcVoiceEngine::CreateMediaVoiceChannel() 行 1657
C++

peerconnection_client.exe!cricket::WebRtcVoiceMediaChannel::WebRtcVoiceMediaChannel(cricket::We
bRtcVoiceEngine * engine=0x01a56e1c) 行 1765 + 0x3d 字节 C++
peerconnection_client.exe!cricket::WebRtcVoiceEngine::CreateChannel() 行 635 + 0x22 字节
C++

peerconnection_client.exe!cricket::CompositeMediaEngine<cricket::WebRtcVoiceEngine,cricket::WebRt
cVideoEngine>::CreateChannel() 行 191 C++
peerconnection_client.exe!cricket::ChannelManager::CreateVoiceChannel_w(cricket::BaseSession *
session=0x03fade50, const std::basic_string<char,std::char_traits<char>,std::allocator<char> > &
content_name="audio", bool rtc=true) 行 327 + 0x1d 字节 C++
peerconnection_client.exe!talk_base::MethodFunctor3<cricket::ChannelManager,cricket::VoiceChannel *
(__thiscall cricket::ChannelManager::*)(cricket::BaseSession
*,std::basic_string<char,std::char_traits<char>,std::allocator<char> > const &,bool),cricket::VoiceChannel
*,cricket::BaseSession *,std::basic_string<char,std::char_traits<char>,std::allocator<char> > const
&,bool>::operator>()() 行 294 + 0x2b 字节 C++
peerconnection_client.exe!talk_base::FunctorMessageHandler<cricket::VoiceChannel
*,talk_base::MethodFunctor3<cricket::ChannelManager,cricket::VoiceChannel * (__thiscall
cricket::ChannelManager::*)(cricket::BaseSession
*,std::basic_string<char,std::char_traits<char>,std::allocator<char> > const &,bool),cricket::VoiceChannel
*,cricket::BaseSession *,std::basic_string<char,std::char_traits<char>,std::allocator<char> > const
&,bool> >::OnMessage(talk_base::Message * msg=0x03a0fdc8) 行 58 + 0xb 字节 C++
peerconnection_client.exe!talk_base::Thread::ReceiveSends() 行 456 + 0x13 字节 C++
peerconnection_client.exe!talk_base::MessageQueue::Get(talk_base::Message * pmsg=0x03a0ff38, int
cmsWait=-1, bool process_io=true) 行 205 + 0xf 字节 C++
peerconnection_client.exe!talk_base::Thread::ProcessMessages(int cmsLoop=-1) 行 508 + 0x19 字节
C++
peerconnection_client.exe!talk_base::Thread::Run() 行 371 C++
peerconnection_client.exe!talk_base::Thread::PreRun(void * pv=0x01a551f0) 行 358 + 0x13 字节
C++
kernel32.dll!76033c45()
[下面的框架可能不正确和/或缺失，没有为 kernel32.dll 加载符号]
ntdll.dll!771a37f5()
ntdll.dll!771a37c8()
```

5.2 音频接收播放过程:

```
peerconnection_client.exe!webrtc::acm1::AudioCodingModuleImpl::IncomingPacket(const unsigned
char * incoming_payload=0x0026f5bc, const int payload_length=3, const webrtc::WebRtcRTPHeader &
rtp_info={...}) 行 2032 C++
peerconnection_client.exe!webrtc::voe::Channel::OnReceivedPayloadData(const unsigned char *
payloadData=0x0026f5bc, unsigned short payloadSize=3, const webrtc::WebRtcRTPHeader *
rtpHeader=0x03a5e0fc) 行 545 + 0x2a 字节 C++
```



```
peerconnection_client.exe!webrtc::RTPReceiverAudio::ParseAudioCodecSpecific(webrtc::WebRtcRTPHeader * rtp_header=0x03a5e0fc, const unsigned char * payload_data=0x0026f5bc, unsigned short payload_length=3, const webrtc::AudioPayload & audio_specific={...}, bool is_red=false) 行 395 + 0x22 字节 C++
peerconnection_client.exe!webrtc::RTPReceiverAudio::ParseRtpPacket(webrtc::WebRtcRTPHeader * rtp_header=0x03a5e0fc, const webrtc::PayloadUnion & specific_payload={...}, bool is_red=false, const unsigned char * payload=0x0026f5bc, unsigned short payload_length=3, __int64 timestamp_ms=16210844, bool is_first_packet=true) 行 206 + 0x1e 字节 C++
peerconnection_client.exe!webrtc::RtpReceiverImpl::IncomingRtpPacket(const webrtc::RTPHeader & rtp_header={...}, const unsigned char * payload=0x0026f5bc, int payload_length=3, webrtc::PayloadUnion payload_specific={...}, bool in_order=true) 行 244 + 0x65 字节 C++
peerconnection_client.exe!webrtc::voe::Channel::ReceivePacket(const unsigned char * packet=0x0026f5b0, int packet_length=15, const webrtc::RTPHeader & header={...}, bool in_order=true) 行 2095 + 0x44 字节 C++
peerconnection_client.exe!webrtc::voe::Channel::ReceivedRTPPacket(const char * data=0x0026f5b0, int length=15) 行 2076 + 0x19 字节 C++
peerconnection_client.exe!webrtc::VoENetworkImpl::ReceivedRTPPacket(int channel=0, const void * data=0x0026f5b0, unsigned int length=15) 行 128 + 0x10 字节 C++
peerconnection_client.exe!cricket::WebRtcVoiceMediaChannel::OnPacketReceived(talk_base::Buffer * packet=0x03a5e784, const talk_base::PacketTime & packet_time={...}) 行 2964 + 0x47 字节 C++
peerconnection_client.exe!cricket::BaseChannel::HandlePacket(bool rtc=false, talk_base::Buffer * packet=0x03a5e784, const talk_base::PacketTime & packet_time={...}) 行 643 + 0x23 字节 C++
peerconnection_client.exe!cricket::BaseChannel::OnChannelRead(cricket::TransportChannel * channel=0x040bf4a0, const char * data=0x04244008, unsigned int len=25, const talk_base::PacketTime & packet_time={...}, int flags=0) 行 392 C++
peerconnection_client.exe!cricket::VoiceChannel::OnChannelRead(cricket::TransportChannel * channel=0x040bf4a0, const char * data=0x04244008, unsigned int len=25, const talk_base::PacketTime & packet_time={...}, int flags=0) 行 1425 C++
peerconnection_client.exe!sigslot::connection5<cricket::BaseChannel,cricket::TransportChannel *,char const *,unsigned int,talk_base::PacketTime const &,int,sigslot::single_threaded>::emit(cricket::TransportChannel * a1=0x040bf4a0, const char * a2=0x04244008, unsigned int a3=25, const talk_base::PacketTime & a4={...}, int a5=0) 行 2047 + 0x2a 字节 C++
peerconnection_client.exe!sigslot::signal5<cricket::TransportChannel *,char const *,unsigned int,talk_base::PacketTime const &,int,sigslot::single_threaded>::operator()(cricket::TransportChannel * a1=0x040bf4a0, const char * a2=0x04244008, unsigned int a3=25, const talk_base::PacketTime & a4={...}, int a5=0) 行 2616 + 0x30 字节 C++
peerconnection_client.exe!cricket::TransportChannelProxy::OnReadPacket(cricket::TransportChannel * channel=0x0423b968, const char * data=0x04244008, unsigned int size=25, const talk_base::PacketTime & packet_time={...}, int flags=0) 行 243 C++
peerconnection_client.exe!sigslot::connection5<cricket::TransportChannelProxy,cricket::TransportChannel *,char const *,unsigned int,talk_base::PacketTime const &,int,sigslot::single_threaded>::emit(cricket::TransportChannel * a1=0x0423b968, const char * a2=0x04244008, unsigned int a3=25, const talk_base::PacketTime & a4={...}, int a5=0) 行 2047 + 0x2a 字节 C++
peerconnection_client.exe!sigslot::signal5<cricket::TransportChannel *,char const *,unsigned int,talk_base::PacketTime const &,int,sigslot::single_threaded>::operator()(cricket::TransportChannel * a1=0x0423b968, const char * a2=0x04244008, unsigned int a3=25, const talk_base::PacketTime & a4={...}, int a5=0) 行 2616 + 0x30 字节 C++
peerconnection_client.exe!cricket::DtlsTransportChannelWrapper::OnReadPacket(cricket::TransportChannel * channel=0x0423bb10, const char * data=0x04244008, unsigned int size=25, const talk_base::PacketTime & packet_time={...}, int flags=0) 行 463 C++
peerconnection_client.exe!sigslot::connection5<cricket::DtlsTransportChannelWrapper,cricket::TransportChannel *,char const *,unsigned int,talk_base::PacketTime const &,int,sigslot::single_threaded>::emit(cricket::TransportChannel * a1=0x0423bb10, const char * a2=0x04244008, unsigned int a3=25, const talk_base::PacketTime & a4={...}, int a5=0) 行 2047 + 0x2a 字节 C++
```

```
peerconnection_client.exe!sigslot::signal5<cricket::TransportChannel *,char const *,unsigned
int,talk_base::PacketTime const &,int,sigslot::single_threaded>::operator()(cricket::TransportChannel *
a1=0x0423bb10, const char * a2=0x04244008, unsigned int a3=25, const talk_base::PacketTime & a4={...}, int
a5=0) 行 2616 + 0x30 字节 C++
peerconnection_client.exe!cricket::P2PTransportChannel::OnReadPacket(cricket::Connection *
connection=0x05c2e260, const char * data=0x04244008, unsigned int len=25, const talk_base::PacketTime &
packet_time={...}) 行 1260 C++
peerconnection_client.exe!sigslot::_connection4<cricket::P2PTransportChannel,cricket::Connection
*,char const *,unsigned int,talk_base::PacketTime const &,sigslot::single_threaded>::emit(cricket::Connection *
a1=0x05c2e260, const char * a2=0x04244008, unsigned int a3=25, const talk_base::PacketTime & a4={...}) 行
1993 + 0x26 字节 C++
peerconnection_client.exe!sigslot::signal4<cricket::Connection *,char const *,unsigned
int,talk_base::PacketTime const &,sigslot::single_threaded>::operator()(cricket::Connection * a1=0x05c2e260,
const char * a2=0x04244008, unsigned int a3=25, const talk_base::PacketTime & a4={...}) 行 2544 + 0x2c 字节
C++
peerconnection_client.exe!cricket::Connection::OnReadPacket(const char * data=0x04244008, unsigned
int size=25, const talk_base::PacketTime & packet_time={...}) 行 962 C++
peerconnection_client.exe!cricket::UDPPort::OnReadPacket(talk_base::AsyncPacketSocket *
socket=0x05c30528, const char * data=0x04244008, unsigned int size=25, const talk_base::SocketAddress &
remote_addr={...}, const talk_base::PacketTime & packet_time={...}) 行 268 C++
peerconnection_client.exe!cricket::UDPPort::HandleIncomingPacket(talk_base::AsyncPacketSocket *
socket=0x05c30528, const char * data=0x04244008, unsigned int size=25, const talk_base::SocketAddress &
remote_addr={...}, const talk_base::PacketTime & packet_time={...}) 行 104 C++
peerconnection_client.exe!cricket::AllocationSequence::OnReadPacket(talk_base::AsyncPacketSocket *
socket=0x05c30528, const char * data=0x04244008, unsigned int size=25, const talk_base::SocketAddress &
remote_addr={...}, const talk_base::PacketTime & packet_time={...}) 行 1021 + 0x30 字节 C++

peerconnection_client.exe!sigslot::_connection5<cricket::AllocationSequence,talk_base::AsyncPacketSo
cket *,char const *,unsigned int,talk_base::SocketAddress const &,talk_base::PacketTime const
&,sigslot::single_threaded>::emit(talk_base::AsyncPacketSocket * a1=0x05c30528, const char * a2=0x04244008,
unsigned int a3=25, const talk_base::SocketAddress & a4={...}, const talk_base::PacketTime & a5={...}) 行 2047
+ 0x2a 字节 C++
peerconnection_client.exe!sigslot::signal5<talk_base::AsyncPacketSocket *,char const *,unsigned
int,talk_base::SocketAddress const &,talk_base::PacketTime const
&,sigslot::single_threaded>::operator()(talk_base::AsyncPacketSocket * a1=0x05c30528, const char *
a2=0x04244008, unsigned int a3=25, const talk_base::SocketAddress & a4={...}, const talk_base::PacketTime &
a5={...}) 行 2616 + 0x30 字节 C++
peerconnection_client.exe!talk_base::AsyncUDPSocket::OnReadEvent(talk_base::AsyncSocket *
socket=0x05c30044) 行 133 C++
peerconnection_client.exe!sigslot::_connection1<talk_base::AsyncUDPSocket,talk_base::AsyncSocket
*,sigslot::multi_threaded_local>::emit(talk_base::AsyncSocket * a1=0x05c30044) 行 1852 + 0x1a 字节
C++
peerconnection_client.exe!sigslot::signal1<talk_base::AsyncSocket
*,sigslot::multi_threaded_local>::operator()(talk_base::AsyncSocket * a1=0x05c30044) 行 2346 + 0x20 字节
C++
peerconnection_client.exe!talk_base::SocketDispatcher::OnEvent(unsigned int ff=1, int err=0) 行 1150
C++
peerconnection_client.exe!talk_base::PhysicalSocketServer::Wait(int cmsWait=32, bool process_io=true)
行 1671 + 0x23 字节 C++
peerconnection_client.exe!talk_base::MessageQueue::Get(talk_base::Message * pmsg=0x03a5ff38, int
cmsWait=-1, bool process_io=true) 行 271 + 0x21 字节 C++
peerconnection_client.exe!talk_base::Thread::ProcessMessages(int cmsLoop=-1) 行 508 + 0x19 字节
C++
peerconnection_client.exe!talk_base::Thread::Run() 行 371 C++
peerconnection_client.exe!talk_base::Thread::PreRun(void * pv=0x00275080) 行 358 + 0x13 字节
C++
kernel32.dll!76033c45()
```

[下面的框架可能不正确和/或缺失, 没有为 kernel32.dll 加载符号]

ntdll.dll!771a37f5()

ntdll.dll!771a37c8()

5.3 视频接收播放过程:

peerconnection_client.exe!webrtc::ModuleRTPUtility::RTPPayloadParser::ParseVP8(webrtc::ModuleRTPUtility::RTPPayload & parsedPacket={...}) 行 658 C++

peerconnection_client.exe!webrtc::ModuleRTPUtility::RTPPayloadParser::Parse(webrtc::ModuleRTPUtility::RTPPayload & parsedPacket={...}) 行 581 + 0xc 字节 C++

peerconnection_client.exe!webrtc::RTPReceiverVideo::ReceiveVp8Codec(webrtc::WebRtcRTPHeader * rtp_header=0x039ee0b8, const unsigned char * payload_data=0x03f7f206, unsigned short payload_data_length=129) 行 181 + 0xc 字节 C++

peerconnection_client.exe!webrtc::RTPReceiverVideo::ParseVideoCodecSpecific(webrtc::WebRtcRTPHeader * rtp_header=0x039ee0b8, const unsigned char * payload_data=0x03f7f206, unsigned short payload_data_length=129, webrtc::RtpVideoCodecTypes video_type=kRtpVideoVp8, __int64 now_ms=2915007, bool is_first_packet=true) 行 126 + 0x15 字节 C++

peerconnection_client.exe!webrtc::RTPReceiverVideo::ParseRtpPacket(webrtc::WebRtcRTPHeader * rtp_header=0x039ee0b8, const webrtc::PayloadUnion & specific_payload={...}, bool is_red=false, const unsigned char * payload=0x03f7f206, unsigned short payload_length=129, __int64 timestamp_ms=2915007, bool is_first_packet=true) 行 75 + 0x28 字节 C++

peerconnection_client.exe!webrtc::RtpReceiverImpl::IncomingRtpPacket(const webrtc::RTPHeader & rtp_header={...}, const unsigned char * payload=0x03f7f206, int payload_length=129, webrtc::PayloadUnion payload_specific={...}, bool in_order=false) 行 244 + 0x65 字节 C++

peerconnection_client.exe!webrtc::ViEReceiver::ReceivePacket(const unsigned char * packet=0x03f7f1ee, int packet_length=153, const webrtc::RTPHeader & header={...}, bool in_order=false) 行 243 + 0x44 字节 C++

peerconnection_client.exe!webrtc::ViEReceiver::OnRecoveredPacket(const unsigned char * rtp_packet=0x03f7f1ee, int rtp_packet_length=153) 行 184 C++

peerconnection_client.exe!webrtc::FecReceiverImpl::ProcessReceivedFec() 行 220 + 0x24 字节 C++

peerconnection_client.exe!webrtc::ViEReceiver::ParseAndHandleEncapsulatingHeader(const unsigned char * packet=0x03f81768, int packet_length=154, const webrtc::RTPHeader & header={...}) 行 259 + 0x1d 字节 C++

peerconnection_client.exe!webrtc::ViEReceiver::ReceivePacket(const unsigned char * packet=0x03f81768, int packet_length=154, const webrtc::RTPHeader & header={...}, bool in_order=true) 行 232 + 0x14 字节 C++

peerconnection_client.exe!webrtc::ViEReceiver::InsertRTPPacket(const unsigned char * rtp_packet=0x03f81768, int rtp_packet_length=154, const webrtc::PacketTime & packet_time={...}) 行 224 + 0x1c 字节 C++

peerconnection_client.exe!webrtc::ViEReceiver::ReceivedRTPPacket(const void * rtp_packet=0x03f81768, int rtp_packet_length=154, const webrtc::PacketTime & packet_time={...}) 行 156 C++

peerconnection_client.exe!webrtc::ViEChannel::ReceivedRTPPacket(const void * rtp_packet=0x03f81768, const int rtp_packet_length=154, const webrtc::PacketTime & packet_time={...}) 行 1670 C++

peerconnection_client.exe!webrtc::ViENetworkImpl::ReceivedRTPPacket(const int video_channel=0, const void * data=0x03f81768, const int length=154, const webrtc::PacketTime & packet_time={...}) 行 160 + 0x14 字节 C++

peerconnection_client.exe!cricket::WebRtcVideoMediaChannel::OnPacketReceived(talk_base::Buffer * packet=0x039ee7a4, const talk_base::PacketTime & packet_time={...}) 行 2570 + 0x5f 字节 C++

peerconnection_client.exe!cricket::BaseChannel::HandlePacket(bool rtc=false, talk_base::Buffer * packet=0x039ee7a4, const talk_base::PacketTime & packet_time={...}) 行 643 + 0x23 字节 C++

peerconnection_client.exe!cricket::BaseChannel::OnChannelRead(cricket::TransportChannel * channel=0x041d0500, const char * data=0x065e2008, unsigned int len=164, const talk_base::PacketTime & packet_time={...}, int flags=0) 行 392 C++


```
peerconnection_client.exe!sigslot::_connection5<cricket::BaseChannel,cricket::TransportChannel *,char
const *,unsigned int,talk_base::PacketTime const &,int,sigslot::single_threaded>::emit(cricket::TransportChannel
* a1=0x041d0500, const char * a2=0x065e2008, unsigned int a3=164, const talk_base::PacketTime & a4={...},
int a5=0) 行 2047 + 0x2a 字节 C++
peerconnection_client.exe!sigslot::signal5<cricket::TransportChannel *,char const *,unsigned
int,talk_base::PacketTime const &,int,sigslot::single_threaded>::operator()(cricket::TransportChannel *
a1=0x041d0500, const char * a2=0x065e2008, unsigned int a3=164, const talk_base::PacketTime & a4={...}, int
a5=0) 行 2616 + 0x30 字节 C++
peerconnection_client.exe!cricket::TransportChannelProxy::OnReadPacket(cricket::TransportChannel *
channel=0x041d2108, const char * data=0x065e2008, unsigned int size=164, const talk_base::PacketTime &
packet_time={...}, int flags=0) 行 243 C++

peerconnection_client.exe!sigslot::_connection5<cricket::TransportChannelProxy,cricket::TransportChan
nel *,char const *,unsigned int,talk_base::PacketTime const
&,int,sigslot::single_threaded>::emit(cricket::TransportChannel * a1=0x041d2108, const char * a2=0x065e2008,
unsigned int a3=164, const talk_base::PacketTime & a4={...}, int a5=0) 行 2047 + 0x2a 字节 C++
peerconnection_client.exe!sigslot::signal5<cricket::TransportChannel *,char const *,unsigned
int,talk_base::PacketTime const &,int,sigslot::single_threaded>::operator()(cricket::TransportChannel *
a1=0x041d2108, const char * a2=0x065e2008, unsigned int a3=164, const talk_base::PacketTime & a4={...}, int
a5=0) 行 2616 + 0x30 字节 C++

peerconnection_client.exe!cricket::DtlsTransportChannelWrapper::OnReadPacket(cricket::TransportChan
nel * channel=0x041d22b0, const char * data=0x065e2008, unsigned int size=164, const talk_base::PacketTime
& packet_time={...}, int flags=0) 行 463 C++

peerconnection_client.exe!sigslot::_connection5<cricket::DtlsTransportChannelWrapper,cricket::Transpo
rtChannel *,char const *,unsigned int,talk_base::PacketTime const
&,int,sigslot::single_threaded>::emit(cricket::TransportChannel * a1=0x041d22b0, const char * a2=0x065e2008,
unsigned int a3=164, const talk_base::PacketTime & a4={...}, int a5=0) 行 2047 + 0x2a 字节 C++
peerconnection_client.exe!sigslot::signal5<cricket::TransportChannel *,char const *,unsigned
int,talk_base::PacketTime const &,int,sigslot::single_threaded>::operator()(cricket::TransportChannel *
a1=0x041d22b0, const char * a2=0x065e2008, unsigned int a3=164, const talk_base::PacketTime & a4={...}, int
a5=0) 行 2616 + 0x30 字节 C++
peerconnection_client.exe!cricket::P2PTransportChannel::OnReadPacket(cricket::Connection *
connection=0x06675900, const char * data=0x065e2008, unsigned int len=164, const talk_base::PacketTime &
packet_time={...}) 行 1260 C++
peerconnection_client.exe!sigslot::_connection4<cricket::P2PTransportChannel,cricket::Connection
*,char const *,unsigned int,talk_base::PacketTime const &,sigslot::single_threaded>::emit(cricket::Connection *
a1=0x06675900, const char * a2=0x065e2008, unsigned int a3=164, const talk_base::PacketTime & a4={...}) 行
1993 + 0x26 字节 C++
peerconnection_client.exe!sigslot::signal4<cricket::Connection *,char const *,unsigned
int,talk_base::PacketTime const &,sigslot::single_threaded>::operator()(cricket::Connection * a1=0x06675900,
const char * a2=0x065e2008, unsigned int a3=164, const talk_base::PacketTime & a4={...}) 行 2544 + 0x2c 字
节 C++
peerconnection_client.exe!cricket::Connection::OnReadPacket(const char * data=0x065e2008, unsigned
int size=164, const talk_base::PacketTime & packet_time={...}) 行 962 C++
peerconnection_client.exe!cricket::UDPPort::OnReadPacket(talk_base::AsyncPacketSocket *
socket=0x041e4dd8, const char * data=0x065e2008, unsigned int size=164, const talk_base::SocketAddress &
remote_addr={...}, const talk_base::PacketTime & packet_time={...}) 行 268 C++
peerconnection_client.exe!cricket::UDPPort::HandleIncomingPacket(talk_base::AsyncPacketSocket *
socket=0x041e4dd8, const char * data=0x065e2008, unsigned int size=164, const talk_base::SocketAddress &
remote_addr={...}, const talk_base::PacketTime & packet_time={...}) 行 104 C++
peerconnection_client.exe!cricket::AllocationSequence::OnReadPacket(talk_base::AsyncPacketSocket *
socket=0x041e4dd8, const char * data=0x065e2008, unsigned int size=164, const talk_base::SocketAddress &
remote_addr={...}, const talk_base::PacketTime & packet_time={...}) 行 1021 + 0x30 字节 C++

peerconnection_client.exe!sigslot::_connection5<cricket::AllocationSequence,talk_base::AsyncPacketSo
cket *,char const *,unsigned int,talk_base::SocketAddress const &,talk_base::PacketTime const
```

```
&,sigslot::single_threaded>::emit(talk_base::AsyncPacketSocket * a1=0x041e4dd8, const char * a2=0x065e2008,
unsigned int a3=164, const talk_base::SocketAddress & a4={...}, const talk_base::PacketTime & a5={...}) 行
2047 + 0x2a 字节 C++
peerconnection_client.exe!sigslot::signal5<talk_base::AsyncPacketSocket *,char const *,unsigned
int,talk_base::SocketAddress const &,talk_base::PacketTime const
&,sigslot::single_threaded>::operator()(talk_base::AsyncPacketSocket * a1=0x041e4dd8, const char *
a2=0x065e2008, unsigned int a3=164, const talk_base::SocketAddress & a4={...}, const talk_base::PacketTime &
a5={...}) 行 2616 + 0x30 字节 C++
peerconnection_client.exe!talk_base::AsyncUDPSocket::OnReadEvent(talk_base::AsyncSocket *
socket=0x041e46bc) 行 133 C++
peerconnection_client.exe!sigslot::connection1<talk_base::AsyncUDPSocket,talk_base::AsyncSocket
*,sigslot::multi_threaded_local>::emit(talk_base::AsyncSocket * a1=0x041e46bc) 行 1852 + 0x1a 字节
C++
peerconnection_client.exe!sigslot::signal1<talk_base::AsyncSocket
*,sigslot::multi_threaded_local>::operator()(talk_base::AsyncSocket * a1=0x041e46bc) 行 2346 + 0x20 字节
C++
peerconnection_client.exe!talk_base::SocketDispatcher::OnEvent(unsigned int ff=1, int err=0) 行 1150
C++
peerconnection_client.exe!talk_base::PhysicalSocketServer::Wait(int cmsWait=477, bool process_io=true)
行 1671 + 0x23 字节 C++
peerconnection_client.exe!talk_base::MessageQueue::Get(talk_base::Message * pmsg=0x039eff38, int
cmsWait=-1, bool process_io=true) 行 271 + 0x21 字节 C++
peerconnection_client.exe!talk_base::Thread::ProcessMessages(int cmsLoop=-1) 行 508 + 0x19 字节
C++
peerconnection_client.exe!talk_base::Thread::Run() 行 371 C++
peerconnection_client.exe!talk_base::Thread::PreRun(void * pv=0x00262510) 行 358 + 0x13 字节
C++
kernel32.dll!76993c45()
[下面的框架可能不正确和/或缺失, 没有为 kernel32.dll 加载符号]
ntdll.dll!77d437f5()
ntdll.dll!77d437c8()
```

6 协议:

6.1 XMPP 协议:

6.1.1 原理介绍

XMPP (可扩展消息处理现场协议) 是基于可扩展标记语言 ([XML](#)) 的协议, 它用于即时消息 ([IM](#)) 以及在线现场探测。它在促进服务器之间的准即时操作。这个协议可能最终允许因特网用户向因特网上的其他任何人发送即时消息, 即使其操作系统和浏览器不同。

XMPP 的前身是 [Jabber](#), 一个开源形式组织产生的网络[即时通信](#)协议。XMPP 目前被 [IETF](#) 国际标准组织完成了标准化工作。标准化的核心结果分为两部分;

在 IETF 中, 把 IM 协议划分为四种协议, 即即时信息和出席协议 (Instant Messaging and Presence Protocol, IMPP)、出席和即时信息协议 (Presence and Instant Messaging Protocol, PRIM)、针对即时信息和出席扩展的会话发起协议 (Session Initiation Protocol for Instant Messaging and Presence Leveraging Extensions, SIMPLE), 以及可扩展的消息出席协议 (XMPP)。最初研发 IMPP 也是为了创建一种标准化的协议, 但是今天, IMPP 已经发展成为基本协议单元, 定义所有即时通信协议应该支持的核心功能集。

XMPP 和 SIMPLE 两种协议是架构, 有助于实现 IMPP 协议所描述的规范。PRIM 最初是基于即时通信的协议, 与 XMPP 和 SIMPLE 类似, 但是已经不再使用

1. XMPP 协议是公开的, 由 JSF 开源社区组织开发的。XMPP 协议并不属于任何的机构和个人, 而是属于整个社区, 这一点从根本上保证了其开放性。

2. XMPP 协议具有良好的扩展性。在 XMPP 中, 即时消息和到场信息都是基于 XML 的结构化信息, 这些信息以 XML 节(XML Stanza)的形式在通信实体间交换。XMPP 发挥了 XML 结构化数据的通用传输层的作用, 它将出席和上下文敏感信息嵌入到 XML 结构化数据中, 从而使数据以极高的效率传送给最合适的资源。基于 XML 建立起来的应用具有良好的语义完整性和扩展性。

3. 分布式的网络架构。XMPP 协议都是基于 Client/Server 架构, 但是 XMPP 协议本身并没有这样的限制。网络的架构和电子邮件十分相似, 但没有结合任何特定的网络架构, 适用范围非常广泛。

4. XMPP 具有很好的弹性。XMPP 除了可用在即时通信的应用程序, 还能用在网络管理、内容供稿、协同工具、档案共享、游戏、远端系统监控等。

5. 安全性。XMPP 在 Client-to-Server 通信, 和 Server-to-Server 通信中都使用 TLS (Transport Layer Security) 协议作为通信通道的加密方法, 保证通信的安全。任何 XMPP 服务器可以独立于公众 XMPP 网络 (例如在企业内部网络中), 而使用 SASL 及 TLS 等技术更加增强了通信的安全性。如下图所示:



6.1.2 XMPP 协议网络架构

XMPP 是一个典型的 C/S 架构, 而不是像大多数即时通讯软件一样, 使用 P2P 客户端到客户端的架构, 也就是说在大多数情况下, 当两个客户端进行通讯时, 他们的消息都是通过服务器传递的(也有例外, 例如在两个客户端传输文件时)。采用这种架构, 主要是为了简化客户端, 将大多数工作放在服务器端进行, 这样, 客户端的工作就比较简单, 而且, 当增加功能时, 多数是在服务器端进行。XMPP 服务的框架结构如下图所示。XMPP 中定义了三个角色, XMPP 客户端, XMPP 服务器、网关。通信能够在这三者的任意两个之间双向发生。服务器同时承担了客户端信息记录、连接管理和信息的路由功能。网关承担着与异构即时通信系统的互联互通, 异构系统可以包括 SMS(短信)、MSN、ICQ 等。基本的网络形式是单客户端通过 TCP / IP 连接到单服务器, 然后在之上传输 XML, 工作原理是:

- 1) 节点连接到服务器;
- 2) 服务器利用本地目录系统中的证书对其认证;
- 3) 节点指定目标地址, 让服务器告知目标状态;
- 4) 服务器查找、连接并进行相互认证;
- 5) 节点之间进行交互。

- **XMPP 客户端**

XMPP 系统的一个设计标准是必须支持简单的客户端。事实上，XMPP 系统架构对客户端只有很少的几个限制。一个 XMPP 客户端必须支持的功能有：

- 1) 通过 TCP 套接字与 XMPP 服务器进行通信；
- 2) 解析组织好的 XML 信息包；
- 3) 理解消息数据类型。

XMPP 将复杂性从客户端转移到服务器端。这使得客户端编写变得非常容易，更新系统功能也同样变得容易。XMPP 客户端与服务端通过 XML 在 TCP 套接字的 5222 端口进行通信，而不需要客户端之间直接进行通信。

基本的 XMPP 客户端必须实现以下标准协议（XEP-0211）：

- 1) RFC3920 核心协议 Core
- 2) RFC3921 即时消息和出席协议 Instant Messaging and Presence
- 3) XEP-0030 服务发现 Service Discovery
- 4) XEP-0115 实体能力 Entity Capabilities

- **XMPP 服务器**

XMPP 服务器遵循两个主要法则：

- 5) 监听客户端连接，并直接与客户端应用程序通信；
- 6) 与其他 XMPP 服务器通信；

XMPP 开源服务器一般被设计成模块化，由各个不同的代码包构成，这些代码包分别处理 Session 管理、用户和服务器之间的通信、服务器之间的通信、DNS（Domain Name System）转换、存储用户的个人信息和好友名单、保留用户在下线时收到的信息、用户注册、用户的身份和权限认证、根据用户的要求过滤信息和系统记录 等。另外，服务器可以通过附加服务来进行扩展，如完整的安全策略，允许服务器组件的连接或客户端选择，通向其他消息系统的网关。

基本的 XMPP 服务器必须实现以下标准协议

- 1) RFC3920 核心协议 Core
- 2) RFC3921 即时消息和出席协议 Instant Messaging and Presence
- 3) XEP-0030 服务发现 Service Discovery

- **XMPP 网关**

XMPP 突出的特点是可以和其他即时通信系统交换信息和用户在线状况。由于协议不同，XMPP 和其他系统交换信息必须通过协议的转换来实现，目前几种主流即时通信协议都没有公开，所以 XMPP 服务器本身并没有实现和其他协议的转换，但它的架构允许转换的实现。实现这个特殊功能的服务端在 XMPP 架构里叫做网关(gateway)。目前，XMPP 实现了和 AIM、ICQ、IRC、MSN Messenger、RSS0.9 和 Yahoo Messenger 的协议转换。由于网关的存在，XMPP 架构事实上兼容所有其他即时通信网络，这无疑大大提高了 XMPP 的灵活性和可扩展性。

6.1.3 XMPP 协议的组成

主要的 XMPP 协议范本及当今应用很广的 XMPP 扩展:

RFC 3920: XMPP 核心。全称: The Extensible Messaging and Presence Protocol, 即可扩展通讯和表示协议。说白了, 就是规定基于 XML 流传输指定节点数据的协议。这么做的好处就是统一(注: 大家都按照这个定义, 做的东西就可以相互通讯、交流, 这个应该很有发展前景!)。它是一个开放并且可扩展的协议, 包括 Jingle 协议都是 XMPP 协议的扩展。(注: 使用 Wireshark 抓包时, 早期的版本可能找不到这个协议, 这时候可以选择 Jabber, 它是 XMPP 协议的前身)。现在很多的 IM 都是基于 XMPP 协议开发的, 包括 gtalk 等。定义了 XMPP 协议框架下应用的网络架构, 引入了 XML Stream (XML 流) 与 XML Stanza (XML 节), 并规定 XMPP 协议在通信过程中使用的 XML 标签。使用 XML 标签从根本上说是协议开放性与扩展性的需要。此外, 在通信的安全方面, 把 TLS 安全传输机制与 SASL 认证机制引入到内核, 与 XMPP 进行无缝的连接, 为协议的安全性、可靠性奠定了基础。核心文档还规定了错误的定义及处理、XML 的使用规范、JID (Jabber Identifier, Jabber 标识符) 的定义、命名规范等等。所以这是所有基于 XMPP 协议的应用都必需支持的文档。

RFC 3921: 用户成功登陆到服务器之后, 发布更新自己的在线好友管理、发送即时聊天消息等业务。所有的这些业务都是通过三种基本的 XML 节来完成的: IQ Stanza (IQ 节), Presence Stanza (Presence 节), Message Stanza (Message 节)。RFC3921 还对阻塞策略进行了定义, 定义是多种阻塞方式。可以说, RFC3921 是 RFC3920 的充分补充。两个文档结合起来, 就形成了一个基本的即时通信协议平台, 在这个平台上可以开发出各种各样的应用。

XEP-0030 服务搜索。一个强大的用来测定 XMPP 网络中的其它实体所支持特性的协议。

XEP-0115 实体性能。XEP-0030 的一个通过即时出席的定制, 可以实时改变交互广告功能。

XEP-0045 多人聊天。一组定义参与和管理多用户聊天室的协议, 类似于 Internet 的 Relay Chat, 具有很高的安全性。

XEP-0096 文件传输。定义了一个从 XMPP 实体到另一个的文件传输。

XEP-0124 HTTP 绑定。将 XMPP 绑定到 HTTP 而不是 TCP, 主要用于不能够持久的维持与服务器 TCP 连接的设备。

XEP-0166 Jingle。规定了多媒体通信协商的整体架构。Jingle 协议是 XMPP 协议上的扩展协议, 它着手解决在 XMPP 协议框架下的点对点的连接问题, 也即 P2P 连接。在 Jingle 框架下, 即使用户在防火墙或是 NAT 网络保护之下, 也能够建立连接, 从而提供文件传送、视频、音频服务等。

XEP-0167 Jingle Audio Content Description Format。定义了一个从 XMPP 实体到另一个的语音传输过程。

TURN 协议: 全称: Traversal Using Relays around NAT, 顾名思义, 就是通过中继服务器来传输数据的协议。

STUN 协议: 全称: Simple Traversal of UDP over NATs, 即 NAT 的 UDP 简单穿越, 它允许位于 NAT (或双重 NAT) 后的客户端找出自己的公网地址, 查出自己位于哪种类型的 NAT 之后以及 NAT 为某一个本地端口所绑定的 Internet 端端口。知道 NAT 类型并且有了公网 IP 和 port, P2P 就方便多了。

XEP-0176 Jingle ICE (Interactive Connectivity Establishment) Transport。即交互式连接建立, 说白了, 它就是利用 STUN 和 TURN 等协议找到最适合的连接。

XEP-0177 Jingle Raw UDP Transport。纯 UDP 传输机制, 文档讲述了如何在没有防火墙且在同一网络下建立连接的。

XEP-0180 Jingle Video Content Description Format。定义了一个从 XMPP 实体到另一个的视频传输过程。

XEP-0181 Jingle DTMF (Dual Tone Multi-Frequency)。

XEP-0183 Jingle Telepathy Transport Method。

6.1.4 Xmpp 介绍

Jabber/XMPP 简介

Messia 2009-06-10

6.1.5 协议内容

中文版本:

RFC3920

可扩展的消息和出席信息协议 (XMPP): 核心协议

关于本文的说明

本文为互联网社区定义了一个互联网标准跟踪协议, 并且申请讨论协议和提出了改进的建议。请参照“互联网官方协议标准”的最新版本 (STD 1) 获得这个协议的标准化进程和状态。本文可以不受限制的分发。

版权声明

本文版权属于互联网社区 (C) The Internet Society (2004).

摘要

本文定义了可扩展消息和出席信息协议 (XMPP) 的核心功能, 这个协议采用 XML 流实现在任意两个网络终端接近实时的交换结构化信息。XMPP 提供一个通用的可扩展的框架来交换 XML 数据, 它主要用来建立即时消息和出席信息应用以实现 RFC 2779 的需求。

目录

1. [绪论](#)
2. [通用的架构](#)
3. [地址空间](#)
4. [XML 流](#)
5. [TLS 的使用](#)
6. [SASL 的使用](#)
7. [资源绑定](#)
8. [服务器回拨](#)
9. [XML 节](#)
10. [服务器处理 XML 节的规则](#)
11. [XMPP 中的 XML 用法](#)
12. [核心的兼容性要求](#)
13. [国际化事项](#)
14. [安全性事项](#)
15. [IANA 事项](#)
16. [参考](#)

RFC3921

出自 Jabber/XMPP 中文翻译计划

跳转到: [导航](#), [搜索](#)

本文的英文原文来自 [RFC 3921](#)

网络工作组 Saint-Andre, Ed.

申请讨论: 3921 Jabber 软件基金会

类别: 标准跟踪 2004 年 10 月

可扩展的消息和出席信息协议 (**XMPP**): 即时消息和出席信息

关于本文的说明

本文为互联网社区定义了一个互联网标准跟踪协议，并且申请讨论协议和提出了改进的建议。请参照“互联网官方协议标准”的最新版本（STD 1）获得这个协议的标准化进程和状态。本文可以不受限制的分发。

版权声明

本文版权属于互联网社区 (C) The Internet Society (2004).

摘要

本文定义了可扩展消息和出席信息协议（**XMPP**）的核心功能的扩展和应用，**XMPP** 提供了 [RFC 2779](#) 定义的基本的即时消息和出席信息功能。

英文版本:

Network Working Group
Request for Comments: 3921
Category: Standards Track

P. Saint-Andre, Ed.
Jabber Software Foundation
October 2004

Extensible Messaging and Presence Protocol (XMPP):
Instant Messaging and Presence

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (2004).

Abstract

This memo describes extensions to and applications of the core features of the Extensible Messaging and Presence Protocol (XMPP) that provide the basic instant messaging (IM) and presence functionality defined in RFC 2779.

Saint-Andre

Standards Track

[Page 1]

6.2 Stun 协议:

STUN (Session Traversal Utilities for NAT, NAT 会话传输应用程序) 是一种网络协议, 它允许位于 NAT (或多重 NAT) 后的客户端找出自己的公网地址, 查出自己位于哪种类型的 NAT 之后以及 NAT 为某一个本地端口所绑定的 Internet 端端口。这些信息被用来在两个同时处于 NAT 路由器之后的主机之间建立 UDP 通信。该协议由 RFC 5389 定义。

一旦客户端得知了 Internet 端的 UDP 端口, 通信就可以开始了。如果 NAT 是完全圆锥型的, 那么双方中的任何一方都可以发起通信。如果 NAT 是受限圆锥型或端口受限圆锥型, 双方必须一起开始传输。

需要注意的是, 要使用 STUN RFC 中描述的技术并不一定需要使用 STUN 协议——还可以另外设计一

个协议并把相同的功能集成到运行该协议的服务器上。

SIP 之类的协议是使用 UDP 分组在 Internet 上传输音频和 / 或视频数据的。不幸的是，由于通信的两个末端往往位于 NAT 之后，因此用传统的方法是无法建立连接的。这也就是 STUN 发挥作用的地方。

STUN 是一个客户机—服务器协议。一个 VoIP 电话或软件包可能会包括一个 STUN 客户端。这个客户端会向 STUN 服务器发送请求，之后，服务器就会向 STUN 客户端报告 NAT 路由器的公网 IP 地址以及 NAT 为允许传入流量传回内网而开通的端口。

以上的响应同时还使得 STUN 客户端能够确定正在使用的 NAT 类型——因为不同的 NAT 类型处理传入的 UDP 分组的方式是不同的。四种主要类型中有三种是可以使用的：完全圆锥型 NAT、受限圆锥型 NAT 和端口受限圆锥型 NAT——但大型公司网络中经常采用的对称型 NAT（又称为双向 NAT）则不能使用。

6.2.1 P2P 实现的原理

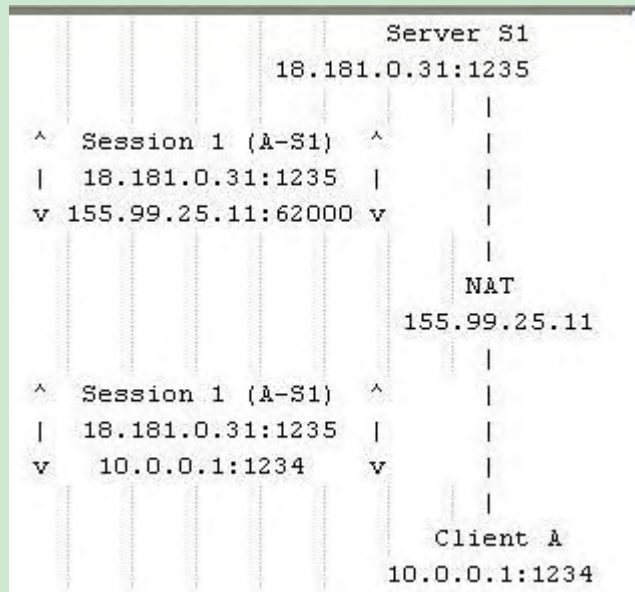
首先先介绍一些基本概念：

NAT(Network Address Translators)，网络地址转换：网络地址转换是在 IP 地址日益缺乏的情况下产生的，它的主要目的就是为了解决地址重用。NAT 从历史发展上分为两大类，基本的 NAT 和 NAPT(Network Address/Port Translator)。

最先提出的是基本的 NAT(注：刚开始其实只是路由器上的一个功能模块)，它的产生基于如下事实：一个私有网络（域）中的节点中只有很少的节点需要与外网连接（这是在上世纪 90 年代中期提出的）。那么这个子网中其实只有少数的节点需要全球唯一的 IP 地址，其他的节点的 IP 地址应该是可以重用的。因此，基本的 NAT 实现的功能很简单，在子网内使用一个保留的 IP 子网段，这些 IP 对外是不可见的。子网内只有少数一些 IP 地址可以对应到真正全球唯一的 IP 地址。如果这些节点需要访问外部网络，那么基本 NAT 就负责将这个节点的子网内 IP 转化为一个全球唯一的 IP 然后发送出去。（基本的 NAT 会改变 IP 包中的原 IP 地址，但是不会改变 IP 包中的端口）

关于基本的 NAT 可以参看 RFC 1631

另外一种 NAT 叫做 NAPT，从名称上我们也可以看得出，NAPT 不但会改变经过这个 NAT 设备的 IP 数据报的 IP 地址，还会改变 IP 数据报的 TCP/UDP 端口。基本 NAT 的设备可能我们见的不多（基本已经淘



汰了），NAPT 才是我们真正需要关注的。看下图：

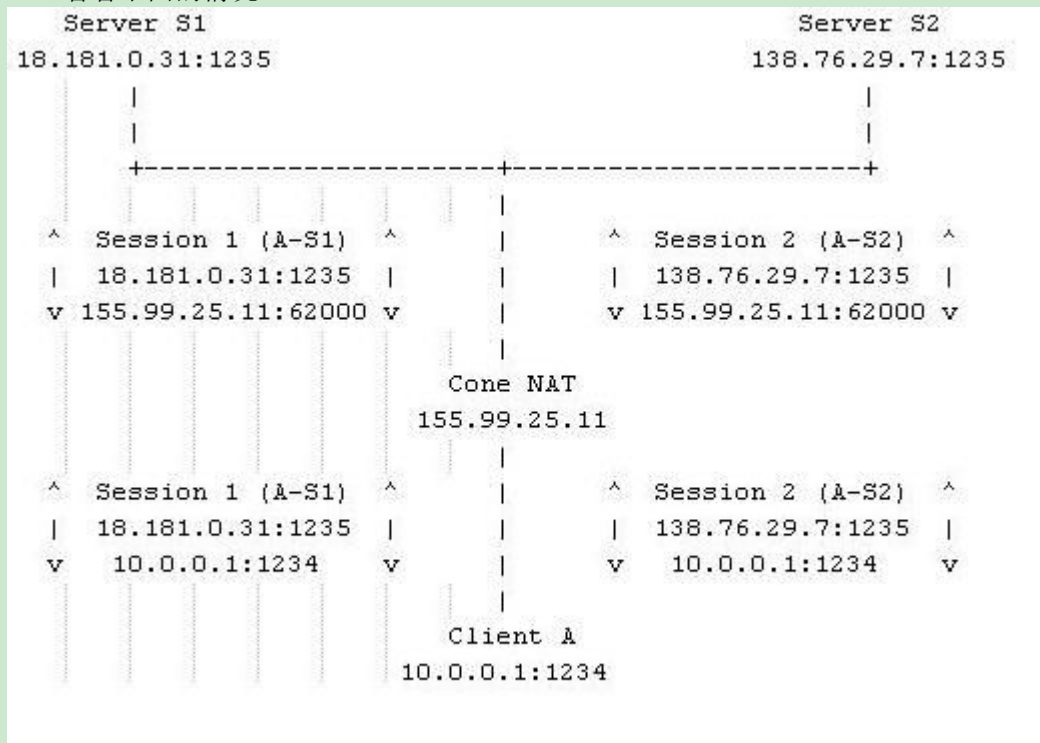
有一个私有网络 10.*.*.*，Client A 是其中的一台计算机，这个网络的网关（一个 NAT 设备）的外网 IP 是 155.99.25.11(应该还有一个内网的 IP 地址，比如 10.0.0.10)。如果 Client A 中的某个进程（这个进程创建了一个 UDP Socket, 这个 Socket 绑定 1234 端口）想访问外网主机 18.181.0.31 的 1235 端口，那么当数据包通过 NAT 时会发生什么事情呢？

首先 NAT 会改变这个数据包的原 IP 地址，改为 155.99.25.11。接着 NAT 会为此传输创建一个 Session（Session 是一个抽象的概念，如果是 TCP，也许 Session 是由一个 SYN 包开始，以一个 FIN 包结束。而 UDP 呢，以这个 IP 的这个端口的第一个 UDP 开始，结束呢，呵呵，也许是几分钟，也许是几小时，这要看具体的实现了）并且给这个 Session 分配一个端口，比如 62000，然后改变这个数据包的源端口为 62000。所以本来是（10.0.0.1:1234→18.181.0.31:1235）的数据包到了互联网上变为了（155.99.25.11:62000→18.181.0.31:1235）。

一旦 NAT 创建了一个 Session 后, NAT 会记住 62000 端口对应的是 10.0.0.1 的 1234 端口, 以后从 18.181.0.31 发送到 62000 端口的数据会被 NAT 自动的转发到 10.0.0.1 上。(注意: 这里是说 18.181.0.31 发送到 62000 端口的数据会被转发, 其他的 IP 发送到这个端口的数据将被 NAT 抛弃) 这样 Client A 就与 Server S1 建立以了一个连接。

上面的是一些基础知识, 下面的才是关键的部分了。

看看下面的情况:



接上面的例子, 如果 Client A 的原来那个 Socket(绑定了 1234 端口的那个 UDP Socket)又接着向另外一个 Server S2 发送了一个 UDP 包, 那么这个 UDP 包在通过 NAT 时会怎么样呢?

这时可能会有两种情况发生, 一种是 NAT 再次创建一个 Session, 并且再次为这个 Session 分配一个端口号(比如: 62001)。另外一种 NAT 再次创建一个 Session, 但是不会新分配一个端口号, 而是用原来分配的端口号 62000。前一种 NAT 叫做 Symmetric NAT, 后一种叫做 Cone NAT。如果你的 NAT 刚好是第一种, 那么很可能会有很多 P2P 软件失灵。(可以庆幸的是, 现在绝大多数的 NAT 属于后者, 即 Cone NAT)

注: Cone NAT 具体又分为 3 种:

(1) 全克隆(Full Cone): NAT 把所有来自相同内部 IP 地址和端口的请求映射到相同的外部 IP 地址和端口。任何一个外部主机均可通过该映射发送 IP 包到该内部主机。

(2) 限制性克隆(Restricted Cone): NAT 把所有来自相同内部 IP 地址和端口的请求映射到相同的外部 IP 地址和端口。但是, 只有当内部主机先给 IP 地址为 X 的外部主机发送 IP 包, 该外部主机才能向该内部主机发送 IP 包。

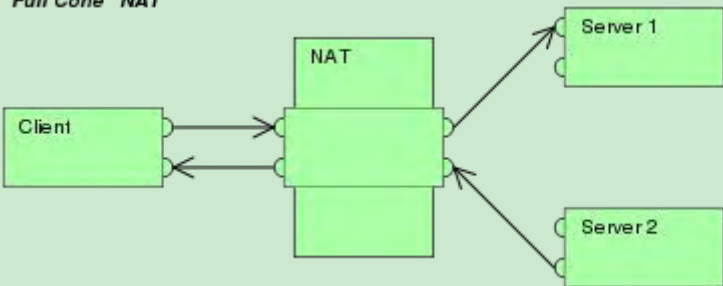
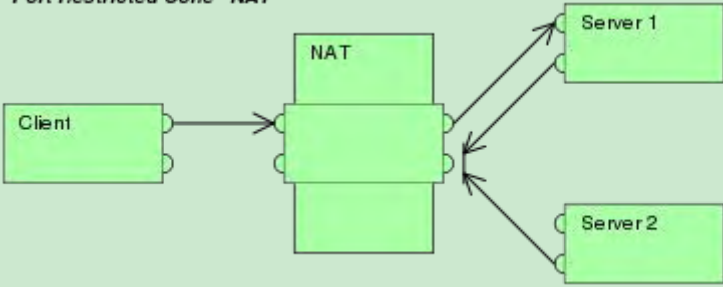
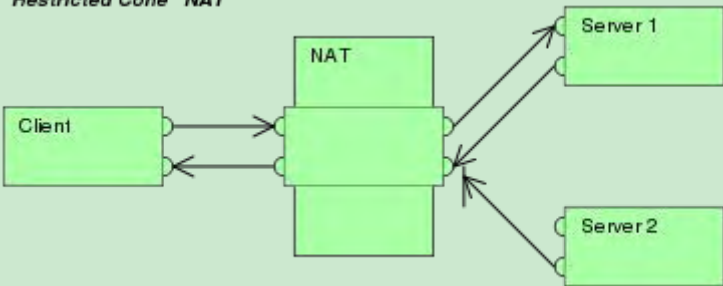
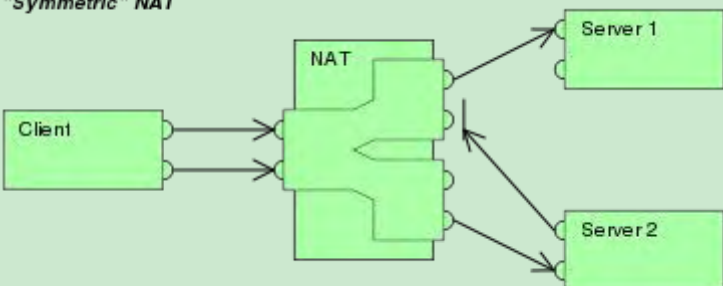
(3) 端口限制性克隆(Port Restricted Cone): 端口限制性克隆与限制性克隆类似, 只是多了端口号的限制, 即只有内部主机先向 IP 地址为 X, 端口号为 P 的外部主机发送 1 个 IP 包, 该外部主机才能够把源端口号为 P 的 IP 包发送给该内部主机。

好了, 我们看到, 通过 NAT, 子网内的计算机向外连结是很容易的(NAT 相当于透明的, 子网内的和外网的计算机不用知道 NAT 的情况)。但是如果外部的计算机想访问子网内的计算机就比较困难了(而这正是 P2P 所需要的)。那么我们如果想从外部发送一个数据报给内网的计算机有什么办法呢?

首先, 我们必须在内网的 NAT 上打上一个“洞”(也就是前面我们说的在 NAT 上建立一个 Session), 这个洞不能由外部来打, 只能由内网内的主机来打。而且这个洞是有方向的, 比如从内部某台主机(比如: 192.168.0.10)向外部的某个 IP(比如: 219.237.60.1)发送一个 UDP 包, 那么就在这个内网的 NAT 设备上打了一个方向为 219.237.60.1 的“洞”, (这就是称为 UDP Hole Punching 的技术)以后 219.237.60.1 就可以通过这个洞与内网的 192.168.0.10 联系了。(但是其他的 IP 不能利用这个洞)。

NAT 的四种类型

Full cone NAT, 亦即著名的一對一

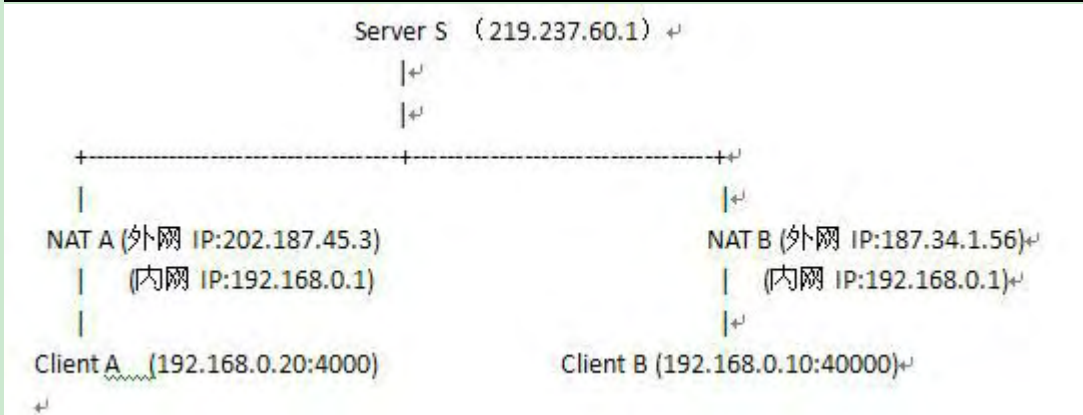
<p>(one-to-one) NAT</p> <ul style="list-style-type: none"> 一旦一个内部地址(iAddr:port1)映射到外部地址(eAddr:port2),所有发自 iAddr:port1 的包都经由 eAddr:port2 向外发送.任意外部主机都能通过给 eAddr:port2 发包到达 iAddr:port1 	<p>"Full Cone" NAT</p> 
<p>Address-Restricted cone NAT</p> <ul style="list-style-type: none"> 一旦一个内部地址(iAddr:port1)映射到外部地址(eAddr:port2),所有发自 iAddr:port1 的包都经由 eAddr:port2 向外发送.任意外部主机(hostAddr:any)都能通过给 eAddr:port2 发包到达 iAddr:port1 的前提是: iAddr:port1 之前发送过包到 hostAddr:any. "any"也就是说端口不受限制 	<p>"Port Restricted Cone" NAT</p> 
<p>Port-Restricted cone NAT</p> <p>类似受限制锥形 NAT (Restricted cone NAT), 但是还有端口限制。</p> <ul style="list-style-type: none"> 一旦一个内部地址(iAddr:port1)映射到外部地址(eAddr:port2),所有发自 iAddr:port1 的包都经由 eAddr:port2 向外发送.一个外部主机(hostAddr:port3)能够发包到达 iAddr:port1 的前提是: iAddr:port1 之前发送过包到 hostAddr:port3. 	<p>"Restricted Cone" NAT</p> 
<p>Symmetric NAT(對稱 NAT)</p> <ul style="list-style-type: none"> 每一個來自相同內部 IP 與 port 的請求到一個特定目的地的 IP 地址和端口, 映射到一個獨特的外部來源的 IP 地址和端口。同一個內部主機發出一個信息包到不同的目的端, 不同的映射使用 只有曾经收到过内部主机封包的外部主机, 才能够把封包发回来 	<p>"Symmetric" NAT</p> 

6.2.2 P2P 的常用实现

一、普通的直连式 P2P 实现

通过上面的理论, 实现两个内网的主机通讯就差最后一步了: 那就是鸡生蛋还是蛋生鸡的问题了, 两边都无法主动发出连接请求, 谁也不知道谁的公网地址, 那我们如何来打这个洞呢? 我们需要一个中间人来联系这两个内网主机。

现在来看看一个 P2P 软件的流程, 以下图为例:



首先, Client A 登录服务器, NAT A 为这次的 Session 分配了一个端口 60000, 那么 Server S 收到的 Client A 的地址是 202.187.45.3:60000, 这就是 Client A 的外网地址了。同样, Client B 登录 Server S, NAT B 给此次 Session 分配的端口是 40000, 那么 Server S 收到的 B 的地址是 187.34.1.56:40000。

此时, Client A 与 Client B 都可以与 Server S 通信了。如果 Client A 此时想直接发送信息给 Client B, 那么 he 可以从 Server S 那儿获得 B 的公网地址 187.34.1.56:40000, 是不是 Client A 向这个地址发送信息 Client B 就能收到了呢? 答案是不行, 因为如果这样发送信息, NAT B 会将这个信息丢弃 (因为这样的信息是不请自来的, 为了安全, 大多数 NAT 都会执行丢弃动作)。现在我们需要的是在 NAT B 上打一个方向为 202.187.45.3 (即 Client A 的外网地址) 的洞, 那么 Client A 发送到 187.34.1.56:40000 的信息, Client B 就能收到了。这个打洞命令由谁来发呢? 自然是 Server S。

总结一下这个过程: 如果 Client A 想向 Client B 发送信息, 那么 Client A 发送命令给 Server S, 请求 Server S 命令 Client B 向 Client A 方向打洞。然后 Client A 就可以通过 Client B 的外网地址与 Client B 通信了。

注意: 以上过程只适合于 Cone NAT 的情况, 如果是 Symmetric NAT, 那么当 Client B 向 Client A 打洞的端口已经重新分配了, Client B 将无法知道这个端口 (如果 Symmetric NAT 的端口是顺序分配的, 那么我们或许可以猜测这个端口号, 可是由于可能导致失败的因素太多, 这种情况下一般放弃 P2P)。

二、STUN 方式的 P2P 实现

STUN 是 RFC3489 规定的一种 NAT 穿透方式, 它采用辅助的方法探测 NAT 的 IP 和端口。毫无疑问的, 它对穿越早期的 NAT 起了巨大的作用, 并且还将继续在 NAT 穿透中占有一席之地。

STUN 的探测过程需要有一个公网 IP 的 STUN server, 在 NAT 后面的 UAC 必须和此 server 配合, 互相之间发送若干个 UDP 数据包。UDP 包中包含有 UAC 需要了解的信息, 比如 NAT 外网 IP, PORT 等等。UAC 通过是否得到这个 UDP 包和包中的数据判断自己的 NAT 类型。

假设有如下 UAC (B), NAT (A), SERVER (C), UAC 的 IP 为 IPB, NAT 的 IP 为 IPA, SERVER 的 IP 为 IPC1、IPC2。请注意, 服务器 C 有两个 IP, 后面你会理解为什么需要两个 IP。

(1) NAT 的探测过程

STEP1: B 向 C 的 IPC1 的 port1 端口发送一个 UDP 包。C 收到这个包后, 会把它收到包的源 IP 和 port 写到 UDP 包中, 然后把此包通过 IP1C 和 port1 发还给 B。这个 IP 和 port 也就是 NAT 的外网 IP 和 port, 也就是说你在 STEP1 中就得到了 NAT 的外网 IP。

熟悉 NAT 工作原理的应该都知道, C 返回给 B 的这个 UDP 包 B 一定收到。如果在你的应用中, 向一个 STUN 服务器发送数据包后, 你没有收到 STUN 的任何回应包, 那只有两种可能: 1、STUN 服务器不存在, 或者你弄错了 port。2、你的 NAT 设备拒绝一切 UDP 包从外部向内部通过, 如果排除防火墙限制规则, 那么这样的 NAT 设备如果存在, 那肯定是坏了,,,

当 B 收到此 UDP 后, 把此 UDP 中的 IP 和自己的 IP 做比较, 如果是一样的, 就说明自己是在公网, 下步 NAT 将去探测防火墙类型, 就不多说了 (下面有图)。如果不一样, 说明有 NAT 的存在, 系统进行 STEP2 的操作。

STEP2: B 向 C 的 IPC1 发送一个 UDP 包, 请求 C 通过另外一个 IPC2 和 PORT (不同与 STEP1 的 IP1) 向 B 返回一个 UDP 数据包 (现在知道为什么 C 要有两个 IP 了吧, 为了检测 cone NAT 的类型)。

我们分析一下, 如果 B 收到了这个数据包, 那说明什么? 说明 NAT 来着不拒, 不对数据包进行任何过滤, 这也就是 STUN 标准中的 full cone NAT。遗憾的是, full cone nat 太少了, 这也意味着你能收到这个数据包的可能性不大。如果没收到, 那么系统进行 STEP3 的操作。

STEP3: B 向 C 的 IPC2 的 port2 发送一个数据包, C 收到数据包后, 把它收到包的源 IP 和 port 写到 UDP 包中, 然后通过自己的 IPC2 和 port2 把此包发还给 B。和 step1 一样, B 肯定能收到这个回应 UDP 包。此包中的 port 是我们最关心的数据, 下面我们来分析:

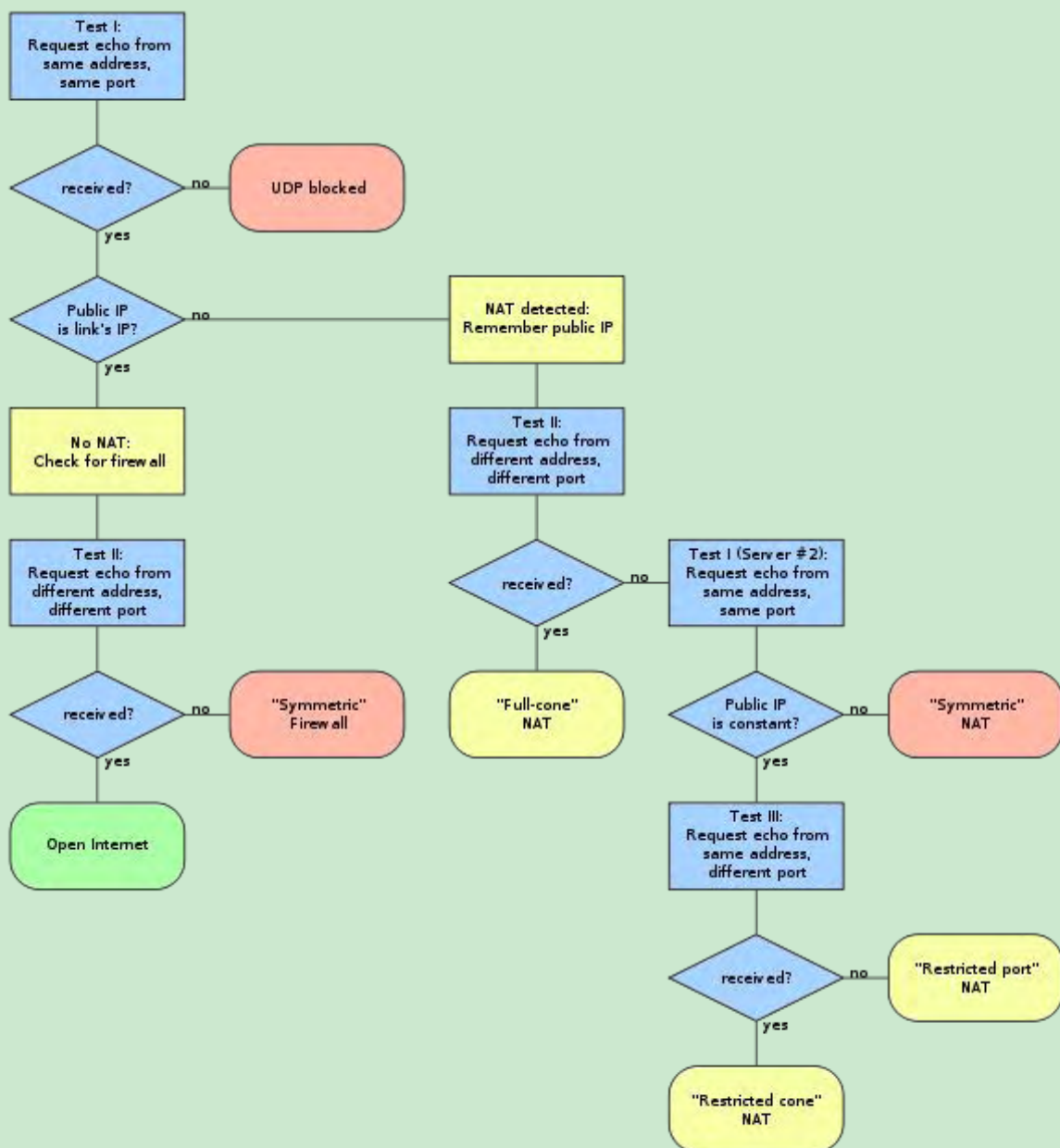
如果这个 port 和 step1 中的 port 一样, 那么可以肯定这个 NAT 是个 CONE NAT, 否则是对称 NAT。道理很简单: 根据对称 NAT 的规则, 当目的地址的 IP 和 port 有任何一个改变, 那么 NAT 都会重新分配一个 port 使用, 而在 step3 中, 和 step1 对应, 我们改变了 IP 和 port。因此, 如果是对称 NAT, 那这两个 port 肯定是不同的。

如果在你的应用中, 到此步的时候 PORT 是不同的, 那就只能放弃 P2P 了, 原因同上面实现中的一样。如果不同, 那么只剩下了 restrict cone 和 port restrict cone。系统用 step4 探测是是那一种。

STEP4: B 向 C 的 IP2 的一个端口 PD 发送一个数据请求包, 要求 C 用 IP2 和不同于 PD 的 port 返回一个数据包给 B。

我们来分析结果: 如果 B 收到了, 那也就意味着只要 IP 相同, 即使 port 不同, NAT 也允许 UDP 包通过。显然这是 restrict cone NAT。如果没收到, 没别的好说, port restrict NAT。

STUN 使用下列的算法 (取自 [RFC 3489](#)) 来发现 NAT gateways 以及防火墙 (firewalls):



一旦路径通过红色箱子的终点时, UDP 的连通是没有可能性的。一旦通过黄色或是绿色的箱子, 就有

连接的可能。

6.2.3 Stun uri

stunURI = scheme ":" host [":" port]
scheme = "stun" / "stuns"

6.2.4 内容

Internet Engineering Task Force
Internet Draft

MIDCOM WG
J. Rosenberg
dynamicsoft
J. Weinberger
dynamicsoft
C. Huitema
Microsoft
R. Mahy
Cisco

draft-ietf-midcom-stun-05.txt
December 19, 2002
Expires: June 2003

STUN - Simple Traversal of UDP Through Network Address
Translators

STATUS OF THIS MEMO

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress".

The list of current Internet-Drafts can be accessed at
<http://www.ietf.org/ietf/1id-abstracts.txt>

To view the list Internet-Draft Shadow Directories, see
<http://www.ietf.org/shadow.html>.

Abstract

Simple Traversal of UDP Through NATs (STUN) is a lightweight protocol that allows applications to discover the presence and types of Network Address Translators (NATs) and firewalls between them and the public Internet. It also provides the ability for applications to determine the public IP addresses allocated to them by the NAT. STUN works with many existing NATs, and does not require any special behavior from them. As a result, it allows a wide variety of

J. Rosenberg et. al.

[Page 1]

6.2.5 中文内容

RFC 3489:



STUN_RFC3489.pdf

RFC 5389:

1.简介.....	3
2.源于 RFC3489 的演变.....	3
3.操作概览.....	4
4.术语.....	5
5.定义.....	5
6. STUN 消息结构.....	6
7.基本的协议处理.....	8
7.1 形成一个请求消息或一个指示消息.....	8
7.2 发送请求消息或者指示消息.....	8
7.2.1 通过 UDP 发送.....	9
7.2.2 通过 TCP 或者 TCP 上的 TLS 发送.....	9
7.3 接受一个 STUN 消息.....	10
7.3.1 处理一个请求.....	11
7.3.2 处理一个指示.....	12
7.3.4 处理一个错误响应.....	13
8. FINGERPRINT 机制.....	13
9. server 的 DNS 发现.....	13
10.鉴权和消息完整性机制.....	14
10.1 短期证书机制.....	14
10.1.1 形成一个请求或者指示.....	15
10.1.2 接受一个请求或指示.....	15
10.1.3 接受一个响应.....	15
10.2 长期证书机制.....	15
10.2.1 形成一个请求.....	16
10.2.2 接受一个请求.....	16
10.2.3 接受一个请求.....	17
11. ALTERNATE-SERVER 机制.....	17
12.向后兼容 RFC3489.....	18
12.1client 处理的改变.....	18
12.2server 处理的改变.....	18
13.基本 server 行为.....	19
14. STUN 用法.....	19
15. STUN 属性.....	20
15.1MAPPED-ADDRESS.....	20
15.2 XOR-MAPPED-ADDRESS.....	21
15.3USERNAME.....	22
15.5FINGERPRINT.....	23
15.6ERROR-CODE.....	23
15.7 域.....	24
15.8 NONCE.....	25
15.9 未知属性.....	25
15.10 软件.....	25
15.11 ALTERNATE-SERVER.....	26
16.安全考虑.....	26

6.2.6 开源服务器

STUNTMAN: <http://www.stunprotocol.org/>
<https://github.com/jselbie/stunserver>
[rfc5766-turn-server](https://github.com/jselbie/stunserver)

<http://sourceforge.net/projects/stun/>

Stuntman - STUN server and client

High performance, production quality STUN server and client library

Vovida.org STUN server

- **mystun**: STUN server and client library from the iptel.org guys. Old but mature. License: GPL, Homepage: <http://developer.berlios.de/projects/mystun/>. You have to download the file via [CVS](#).
- **Vovida STUN server (stund)**: STUN server and client library/application for Linux and Windows from the Vovida guys. Old but mature. License: Vovida Software License 1.0, Homepage: <http://sourceforge.net/projects/stun/>.
- **WinSTUN**: A Windows STUN client, part of the Vovida STUN server (see above). A nice application to test your NAT box. Homepage: <http://sourceforge.net/projects/stun/files/WinStun/>.
- **reTurn**: STUN/TURN server and client library, part of the resiprocate project. Server application is provided as well, but it seems incomplete (authentication). License: 3-clause BSD license. Homepage: http://www.resiprocate.org/ReTurn_Overview.
- **restund**: STUN/TURN server, supports authentication against a mysql DB. License: 3-clause BSD license. Homepage: <http://www.creytiv.com/restund.html>.
- **TurnServer**: STUN/TURN server. License: GPL3. Homepage: <http://turnserver.sourceforge.net/>.
- **PJNATH**: Open Source ICE, STUN, and TURN Library, <http://www.pjsip.org/pjnath/docs/html/>
- **Numd**: a free STUN/TURN serve, <http://numb.viagenie.ca/>

6.2.7 公开的免费 STUN 服务器

//from origin post

stunserver.org

stun.xten.com

stun.fwdnet.net

stun.fwdnet.net:3478

stun.wirlab.net

stun01.sipphone.com

stun.iptel.org

stun.ekiga.net

stun.fwdnet.net

stun01.sipphone.com (no DNS SRV record)

stun.softjoys.com (no DNS SRV record)

stun.voipbuster.com (no DNS SRV record)

stun.voxgratia.org (no DNS SRV record)

stun.xten.com

stunserver.org

stun.sipgate.net:10000

stun.softjoys.com:3478

```
//from https://gist.github.com/zziuni/3741933
```

```
# source : http://code.google.com/p/natvpn/source/browse/trunk/stun_server_list
```

```
# A list of available STUN server.
```

stun.l.google.com:19302

stun1.l.google.com:19302

stun2.l.google.com:19302

stun3.l.google.com:19302

stun4.l.google.com:19302

stun01.sipphone.com

stun.ekiga.net

stun.fwdnet.net

stun.ideasip.com

stun.iptel.org

stun.rixtelecom.se

stun.schlund.de

stunserver.org

stun.softjoys.com

stun.voiparound.com

stun.voipbuster.com

stun.voipstunt.com

stun.voxgratia.org

stun.xten.com

6.3 Turn 协议

6.3.1 概念

TURN (全名 Traversal Using Relay NAT), 是一种资料传输协议 (data-transfer protocol)。允许在 TCP 或 UDP 的连线上跨越 NAT 或防火墙。

TURN 是一个 client-server 协议。TURN 的 NAT 穿透方法与 STUN 类似, 都是通过取得应用层中的公有地址达到 NAT 穿透。但实现 TURN client 的终端必须在通讯开始前与 TURN server 进行交互, 并要求 TURN server 产生 "relay port", 也就是 relayed-transport-address。这时 TURN server 会建立 peer, 即远端端点

(remote endpoints), 开始进行中继 (relay) 的动作, TURN client 利用 relay port 将资料传送至 peer, 再由 peer 转传到另一方的 TURN client。

6.3.2 Turn uri

```
turnURI = scheme ":" host [ ":" port ]
        [ "?transport=" transport ]
scheme = "turn" / "turns"
transport = "udp" / "tcp" / transport-ext
transport-ext = 1*unreserved
```

Table 1 shows how the <secure>, <port> and <transport> components are populated from various URIs. For all these examples, the <host> component is populated with "example.org".

URI	<secure>	<port>	<transport>
turn:example.org	false		
turns:example.org	true		
turn:example.org:8000	false	8000	
turn:example.org?transport=udp	false		UDP
turn:example.org?transport=tcp	false		TCP
turns:example.org?transport=tcp	true		TLS

6.3.3 开源服务器工程

- [Restund](#) OpenSource Modular STUN/TURN Server (BSD License)
- [Numb](#) is a free STUN/TURN server.
- [TurnServer](#) - OpenSource TURN server.
- [reTurn](#) - opensource STUN/TURN server and client library (C++)
- [TURN Server](#) - High-Performance Open Source TURN/STUN server (BSD license) and client library (C)
<https://code.google.com/p/rfc5766-turn-server/>
搭建教程: http://www.dialogic.com/den/developer_forums/f/71/t/10238.aspx
<http://zhangjunli177.blog.163.com/blog/static/138607308201341411384462/>

6.3.4 开源库

- [AnyFirewall](#) - STUN, TURN & ICE library.
- [Libnice](#) - STUN, TURN & ICE library used in Pidgin, GNOME, MeeGo, etc.
- [ice4j](#) - STUN, TURN & ICE library in Java

6.4 交互式连接建立 (Interactive Connectivity Establishment), 一种综合性的 NAT 穿越的技术。

交互式连接建立是由 IETF 的 MMUSIC 工作组开发出来的一种 framework, 可整合各种 NAT 穿透技术, 如 STUN、TURN (Traversal Using Relay NAT, 中继 NAT 实现的穿透)、RSIP (Realm Specific IP, 特定域 IP) 等。该 framework 可以让 SIP 的客户端利用各种 NAT 穿透方式打穿远程的防火墙。

一、ICE 产生的背景

基于信令协议的多媒体传输是一个两段式传输。首先，通过信令协议（如 SIP）建立一个会话连接，通过该连接，会话双方（Agent）通过 SIP 交互所承载的 SDP 消息彼此学习传输媒体时所必须的信息，针对媒体传输机制达成共识。然后，通常采用 RTP 协议进行媒体传输。

基于传输效率的考虑，通常在完成第一阶段的交互之后，通信双方另外建立一条直接的连接传输媒体。这样就会减少传输时延、降低丢包率并减少开销。这样，用于 SIP 传输的链路就不再用于传输媒体。现在，问题出现了，由于不采用原来的链路，当传输双方中任一方位于 NAT 之后，新的传输链接必须考虑 NAT 穿越问题。

通常有四种形式的 NAT，对于每一中 NAT 方式，都有相应的解决方案。然而，每一种 NAT 穿越解决方案都局限于穿越对应得 NAT 方式，对于复杂的网络环境来说，将会出现无法进行媒体传输的情况，同时这些方案给整个系统带来了在不同程度上的脆弱性和复杂性。

在这种背景下，Interactive Connectivity Establishment（交互式连通建立方式）也即 ICE 解决方案应运而生。ICE 方式能够在不增加整个系统的复杂性和脆弱性的情况下，实现对各种形式的 NAT 进行穿越，使得媒体流在通信双方顺利传输。

二、ICE 工作的基本原理及特性

ICE 是一种探索和更新式的解决方案。通过收集自己的和通信对端的尽可能多的网络信息（各种网络地址），尝试在这些地址之间建立数据通道，并在此过程中不断更新先前收集到的信息，从而找到和选择一条能够进行 NAT 穿越的数据通道。

其特性如下：ICE 实现不是很复杂，支持 TCP 穿透，对 NAT 设备没有要求，支持所有类型的 NAT，必须在客户端实现 ICE，在网络结构中需要 STUN/TURN 服务器，具有与协议无关性和良好的可扩展性，安全性和健壮性都比较好。

三、ICE 工作的核心

如下内容是 ICE 实现 NAT 穿透的所要完成的核心处理。包括收集地址，对地址进行排序、配对，然后执行连通性检查。

1、收集地址

Agent 必须确定所有的候选的地址。这些地址包括本地网络接口的地址和由它派生的其他所有地址。本地网络地址包括本地网卡地址、VPN 网络地址、MIP 网络地址等。派生地址指的是通过本地地址向 STUN 服务器发送 STUN 请求获得的网络地址，这些地址分为两类，一类是通过 STUN 的绑定发现用法得到的地址，称为服务器反向候选地址（Server Reflexive Candidates）或服务器反向地址。另一类是通过中继用法得到的，称为中继地址（RELAYED CANDIDATES）。上面提到的两种用法在相应的规范中提出。

服务器反向地址实际上就是终端的网络包经过一重或多重 NAT 穿透之后，由 STUN 服务器观察到的经过 NAT 转换之后的地址。中继地址是 STUN 服务器收到 STUN 请求后，为请求发起方在本机上分配的代理地址，所有被路由到该地址的网络包将会被转发到服务器反向地址，继而穿透 NAT 发送到终端，因此如名字所示，它是 STUN 服务器完成中继功能的地址。

为了找到服务器反向地址，Agent 通过每一个主机候选地址（通过绑定主机某个接口和端口而获取的候选地址），使用绑定发现用法（Binding Discovery Usage [11]）发送一个 STUN 绑定请求给 STUN 服务器（STUN 服务器的地址已经配置或者可以通过某种途径学习到）。当 Agent 发送绑定请求，NAT 将分配一个绑定，它映射该服务器反向地址到主机候选地址。这样，通过主机候选地址发送的外发包，将通过 NAT 转换为通过服务器反向地址发送的包。发往服务器反向候选地址的包，将被 NAT 转换为发往该主机候选地址的包，并转发给 Agent。

当 Agent 与 STUN 服务器之间存在多重 NAT，那么 STUN 请求将会针对每一个 NAT 创建一个绑定，但是，只有最外部的服务器反向地址会被 Agent 发现。如果 Agent 不在任何 NAT 之后，那么，基候选传输地址将与服务器反向地址相同，服务器反向地址可以忽略。

关于中继地址，STUN 中继用法允许 STUN 服务器作为一个媒体中继器进行工作，在 L 与 R 之间进行转发。为了发送消息到 L，R 必须发送消息给媒体中继器，通过媒体中继器转发给 L。反之亦然。

从 L 到 R 的消息其地址信息将两次被重写：第一次被 NAT，第二次被 STUN 中继服务器。这样，R 所了解的想与之通信的地址就是 STUN 中继服务器的地址。这个地址就是中继地址。

2、连通性检查

Agent L 收集到所有的候选地址后，就将它们按优先级高低进行排序，再通过信令信道发送给 Agent R。这些候选地址作为 SDP 请求的属性被传输。当 R 收到请求，它执行相同的地址收集过程，并且把它自己的候选地址作为响应消息发给请求者。这样，每个 Agent 都将有一个完整的包含了双方候选地址的列表，然后准备执行连通性检查。

连通性检查的基本原理是：

- 按照优先顺序对候选地址进行排序。
- 利用每个候选地址发送一个检查包。
- 收到另一个 Agent 的确认检查包。

首先，Agent 将本地地址集和远程地址集进行配对，如本地有 n 个地址，远程有 m 个地址，那么配成 $n*m$ 对。对这些地址对进行连通性检查是通过发送和接收 STUN 请求和响应完成的，此时，Agent 在每个地址对的本地地址上，必须同时充当 STUN 服务器和 STUN 客户端的角色。若通信双方以某一地址对通过一个完整的四次握手，那么该地址对就是有效地址对。

四次握手是指：当通过地址对中的本地地址向地址对中远程地址发送一个 STUN 请求，并成功收到 STUN 响应，称该地址对是可接收的；当地址对中的本地地址收到地址对中远程地址的一个 STUN 请求，并成功地响应，则称该地址对为可发送的。若一个地址对是可接收的，同时又是可发送的，则称该地址对是有效的，即通过连通性检查。则此地址对可用于媒体传输。

通常在对称 NAT 的情况下，在地址对验证过程中，会出现发现以前收集地址时没有收集到的地址对，这时就要对这些新的地址对进行连通性检查。

3、对候选地址进行排序

由于收集候选地址时，收集的是所有的候选地址，为了能够更快更好的找到能够正常工作的候选地址对，对所有组合进行排序是势在必行的。在此说明进行排序的两个基本原则，详细地排序算法将在后续文档中描述。

- Agent 为它的每个候选地址设置一个数值的优先级，这个优先级连同候选地址对一起发送给通信的对端。
- 综合本地的和远程的候选地址的优先级，计算出候选地址对的优先级，这样，双方的同一个候选地址对的优先级相同。以此排序，则通信双方的排序结果相同。

4、进行 SDP 编码

为了实现基于 ICE 的 NAT 穿越，对 SDP 进行了扩展，主要增加了四个属性。分别是 candidate 属性、ice-ufrag 属性、ice-pwd 属性和 remote-candidates 属性。

candidate 属性为通信提供多种可能的候选地址中的一个。这些地址是使用 STUN 的端到端的连通性检查为有效的。

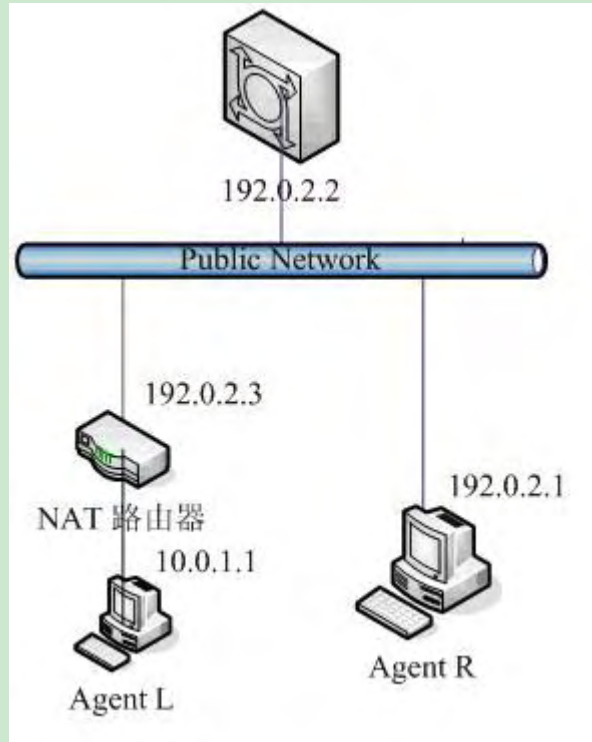
remote-candidates 属性提供请求者想要应答者在应答中使用的远程候选传输地址标识。

ice-pwd 属性提供用于保护 STUN 连通性检查的密码。

ice-ufrag 属性提供在 STUN 连通性检查中组成用户名的片断。

四、一个例子

两个 Agent, L 和 R, 使用 ICE。它们都有单个 IPv4 接口。对于 Agent L 地址为 10.0.1.1, 对于 R, 192.0.2.1。它们都配置了单独的 STUN 服务器（实际上是同一个），STUN 服务器在 192.0.2.2 地址的 3478 端口监听 STUN 请求。这个 STUN 服务器同时支持绑定发现和中继功能。Agent L 位于 NAT 之后，R 位于公网。NAT 有一个终端独立的映射特性和依靠地址的过滤特性。NAT 公网端的地址是 192.0.2.3。网络结构图如下所示。

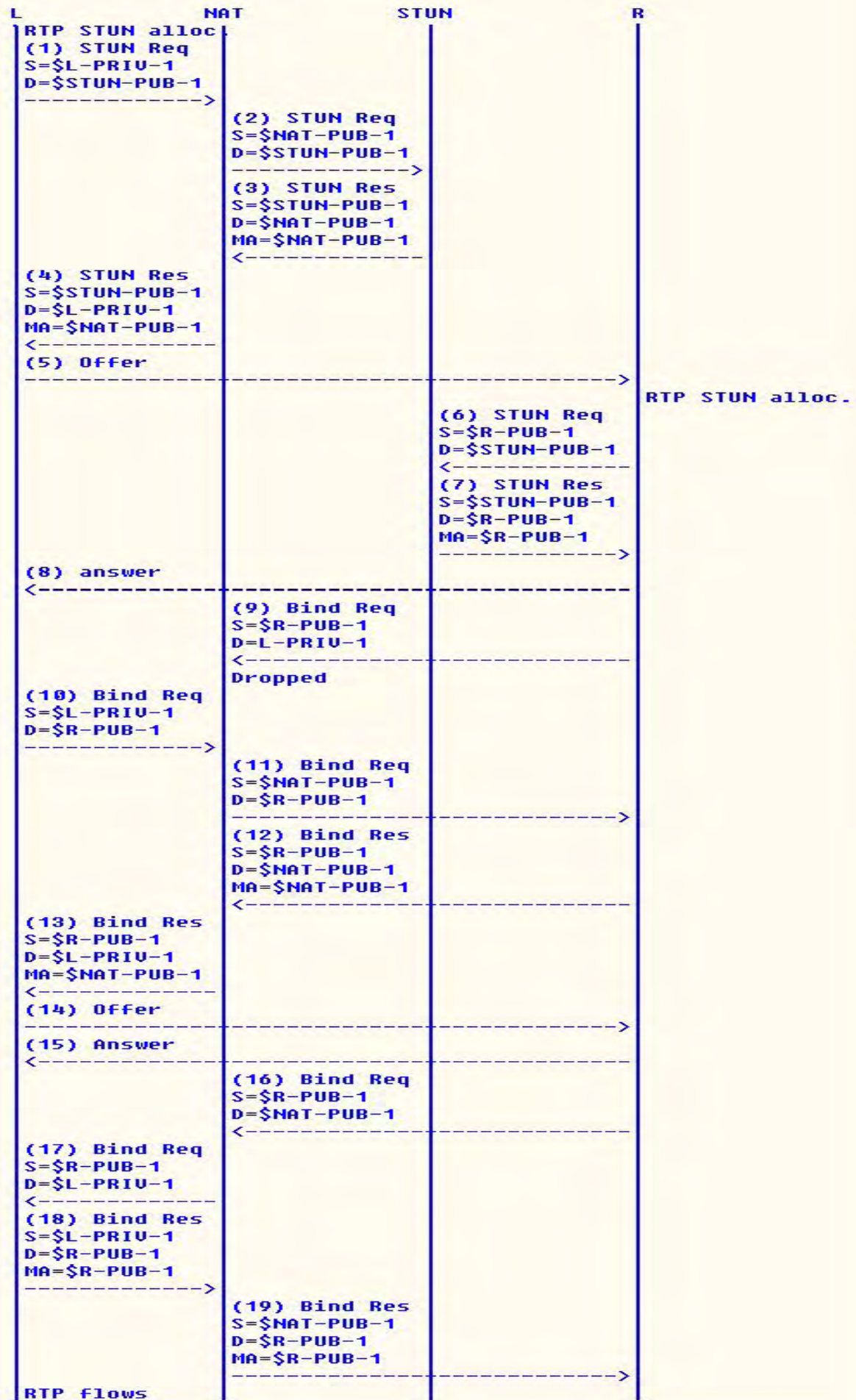


为了便于理解，传输地址用变量名代替。变量名的格式是 entity-type-seqno，其中 entity 是具有该传输地址的接口所在实体，具体为是 L、R、STUN 或 NAT 之一。type 不是 PUB（地址位于公网）就是 PRIV（地址位于内网）。seqno 是在实体上的相同类型的各传输地址各自的序列号。每个变量都有一个 IP 地址和端口号，分别用 varname.IP 和 varname.port 表示，varname 就是变量名。

STUN 服务器有公网的传输层地址 STUN-PUB-1（192.0.2.2: 3478），绑定发现用法和中继用法都使用这个地址。但在此处，两个 Agent 都不使用中继用法。

在呼叫过程中，STUN 消息有被许多属性注解。“S=”属性表明消息的源传输地址，“D=”属性表明消息的目标传输地址。“MA”属性用于 STUN 绑定响应消息，指明映射的地址。

基于以上规定，媒体传输的初始过程如下图所示。



消息

双方都对获取的传输地址进行配对，确定优先级并排序。之后，R 开始执行其连通性检查（消息 9），由于来自 L 的候选地址是一个私有地址，所以此检查必定失败，而被丢弃。

同时，L 收到应答后，除去包含了服务器反向地址的那对检查，只剩一对检查。基于此对地址，执行连通性检查（消息 10），经过 NAT 转换后发送给 R（消息 11）。R 收到之后发送响应给 L（消息 12），该消息中通过 MA 属性指明映射地址，经过 NAT 之后返回给 L（消息 13），这样 L 的连通性检查成功。L 检查收到的消息 13，以 NAT-PUB-1 为本地地址，R-PUB-1 为远程地址创建新的地址对，并添加到有效列表中。

ICE 查看有效列表，发现有一对存在，就发送一个更新请求（消息 14）给 R，这个请求用于删除没有被选中的候选地址，并且指示远程地址。

消息 11 到达 R 之后，会触发 R 执行一个相同地址对的检查，消息 16-19 反映了这个过程，在收到消息 19 的响应之后，R 会像 L 一样创建一对新的地址对（以 R-PUB-1 为本地地址，以 NAT-PUB-1 为远程地址），并添加到有效列表中。这样就可以进行媒体传输了。

五、总结

本文档从 ICE 的产生背景入手，讨论了 ICE 的基本原理及其特性，并对其工作的几个核心部位进行了简单的概述，在此基础上，分析了一个基于 ICE 通信的例子。所涉及的内容都是在宏观上的考虑，进一步的详细论述将在后续工作中展开。

1-4 获取服务器反向地址。消息 5 发送一个请求给 R，该请求包括了本地主机候选地址和服务器反向地址。R 收到消息 5 之后，通过消息 6-7 获取服务器反向地址（由于 R 不在 NAT 之后，服务器反向地址与主机候选地址相同），然后发送一个应答（消息 8）给 L，应答中包括主机候选地址。至此，通信双方都获取了彼此的网络信息。ICE 的典型应用环境

6.4.1 IETF 规格

- Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols [RFC 5245](#)
- Session Traversal Utilities for NAT (STUN): [RFC 5389](#)
- Traversal Using Relays around NAT (TURN): Relay Extensions to STUN [RFC 5766](#)

Internet Engineering Task Force (IETF)
Request for Comments: 5245
Obsoletes: 4091, 4092
Category: Standards Track
ISSN: 2070-1721

J. Rosenberg
jdrosen.net
April 2010

Interactive Connectivity Establishment (ICE):
A Protocol for Network Address Translator (NAT) Traversal for
Offer/Answer Protocols

Abstract

This document describes a protocol for Network Address Translator (NAT) traversal for UDP-based multimedia sessions established with the offer/answer model. This protocol is called Interactive Connectivity Establishment (ICE). ICE makes use of the Session Traversal Utilities for NAT (STUN) protocol and its extension, Traversal Using Relay NAT (TURN). ICE can be used by any protocol utilizing the offer/answer model, such as the Session Initiation Protocol (SIP).

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at <http://www.rfc-editor.org/info/rfc5245>.

Rosenberg

Standards Track

[Page 1]

6.4.2 开源工程:

[PJNATH – Open Source ICE, STUN, and TURN Library](#)

[libnice: Glib ICE library](#)

How to play libnice-ly with your NAT

Youness Alaoui



6.5 XEP-0166 Jingle

本文档定义了 Jabber/XMPP 客户间初始化及管理点对点的多媒体会话(sessions) (比如, 声音和图像的交换) 框架, 它在一定程度上与现有的 Internet 标准具有互操作性。

警告: 本标准跟踪文档是实验性的。作为 XMPP 扩展协议发表, 并不意味着 XMPP 标准基金会批准了这个协议。我们鼓励对本协议进行探索性的实现, 但在本协议的状态发展为草稿之前, 产品性的系统不应实现本协议。

文档信息

系列: [XEP](#)

序号: [0166](#)

发布者: [XMPP 标准基金会](#)

状态: [实验性的](#)

类型: [标准跟踪](#)

版本: 0.14

最后更新: 2007-04-17

批准机构: [XMPP 理事会](#)

依赖标准: XMPP 核心标准

被替代标准: 无

缩略名: 未指派

Wiki 页: [\[1\]](#)

作者信息

法律通告

讨论地点

相关的 XMPP

术语

6.5.1 绪论

从 Jabber/XMPP 客户内部初始化和点对点 (p2p) 互操作 (象声音、图像、或文件共享交换) 的未广泛采用的标准已经有了。虽然, 一些大 的服务提供商和 Jabber/XMPP 客户已经写出和实现了他们自己独有的用于点对点信号处理的 XMPP 扩展, 但这些技术没有公开, 并且总是没有考虑到与 公共转换电话网络 (PSTN) 或跨互联网声音协议 (VoIP) 的互操作性的需求。这些网络建立在 IETF 的 __会话初始化协议 (SIP) __ 上, 在 RFC3261[\[注 1\]|XMPP 文档列表/XMPP 扩展/XEP-0166]及其各种扩展中有详细说明。

与此相反, 唯一存在的开放协议是 {link:初始化及协商会话的传输
<http://www.xmpp.org/extensions/xep-0111.html>}\[[注 2\]|XMPP 文档列表/XMPP 扩展/XEP-0166], 它使得初始化及管理点对点会话成为可能, 却没有提供足够多的在 Jabber/XMPP 客户端中能轻松地实现的关键性的信号处理语义。其结果导致在 XMPP 社区里有关信号处理的协议支离破碎。基本上, 有两中方法可以解决这个问题:

1. 推荐所有的客户端开发者实现双重(XMPP+SIP)解决方案.
2. 定义一个 XMPP 信号处理的完整特征的协议。

实现经验表明, 双重方法也许不会在所有的计算平台上都可行-也许 Jabber 客户端已经写完了, 或者虽然可行但并不值当。因此, 定义一个 XMPP 信号处理协议似乎合情合理, 这个协议能提供所需的信号处理语义, 同时也使得与现有互联网标准的互操作性相对简单。

作为收到的 XEP-0111 反馈的一个结果, 文档的原作者 (Joe Hildebrand 和 Peter Saint-Andre) 开始定义这样的一个信号处理协议, 代码名为 Jingle。通过与 Google Talk 小组\[4\]成员交流, 发现形成的 Jingle 方法在概念上 (甚至在句法上) 都与在 Google Talk 程序中使用的信号处理协议非常相似。因此, 为了保持互操作性和适用性, 我们决定协调这两种方法。因此, 由本文详细说明了的信号处理协议基本上等同于 现有的 Google Talk 协议, 只是根据在 XMPP 标准基金会的标准化的实施及发表进程中收到的反馈作了一些调整。

Jingle 的目的不是排挤或替代 SIP。因为构建双重 XMPP+SIP 客户端非常困难, 导致本质上程序控制的两个中心, 所以, 我们将 Jingle 设计成纯的 XMPP 信号处理协议。Jingle 意欲与 SIP 相互作用, 这样数百万已布置的 XMPP 客户端能够加到现有的开放的 VoIP 网络之 中, 而不是将 XMPP 用户限制在某个分离的独特的 VoIP 网络中。

6.5.2 需求

这里定义的协议的目标是满足如下需求:

1. 使得 XMPP 内多种点对点会话 (不限于声音和视频) 的管理成为可能[6]。
2. 明确分离信号处理通道(XMPP)与数据通道 (例如, 在 RFC3550 中说明的实时传输协议[7])。
3. 明确分离内容描述格式(例如, 用于语音聊天的)与内容传输方法(比如, 在 RFC768[8]中说明的用户数据报协议)。
4. 使得从现有会话中加入、修改、删除内容类型成为可能。
5. 使得实现支持标准的 Jabber/XMPP 客户端中的协议相对容易。
6. 当需要与非 XMPP 实体通讯的时候, 尽可能将复杂性推到 XMPP 网络与非 XMPP 网络间的服务器端网关上。

本文档仅定义了信号处理协议。其他文档详细说明了如下内容:

- 各种内容描述格式 (音频、视频等), 如有可能, 将这些类型映射到会话描述协议 (SDP, 参见 RFC4566[9]) 中; 示例包括经由 RTP 的 Jingle 音频[10]及经由 RTP 的视频[11]。
- 各种内容传输方法; 示例包括 Jingle ICE 传输方法[12]及 Jingle 原生 UDP 传输[13]。
- 映射 Jingle 信号处理协议到现有信号处理标准的方法, 象 IETF 的会话初始化协议(SIP;参见 RFC2361[14]), ITU 的 H.323 协议 (见 H.323[15]); 这些文档即将完成。

1.1 2. 术语表 {anchor:术语表}

{table}

术语 | 定义

会话 | 连接两个实体的许多对已协商的内容传输方法和内容描述格式。它被限定在初始化请求和会话结束时间的时间段内。在一个会话的生命周期内, 可加入或删除成对的内容描述和内容传输方法。在某一时刻, 一个会话至少有一个已协商的内容类型。

内容类型 | 一个内容描述和一个内容传输方法的组合。

内容描述 | 内容类型将被建立的格式, 从形式上声明了会话的一种用途 (如, “voice”或“video”)。这是会话的 (即, 传输的比特 位) “是什么”, 象建立语音通话时可接受的编码器等。按照 Jingle XML 语法, 内容类型是元素<description/>的命名空间。

传输方法 | 实体间建立数据流的方法。可能的传输包括 ICE-TCP, 原生 UDP, 带内数据 (inband data) 等。这是关于会话 “怎样” 的部分。按照 Jingle XML 语法, 这是元素<transport/>的命名空间。内容传输方法定义了怎样将比特位从一台主机传到另一台。每种传输方法必须指定是有 损的 (允许丢包) 还是无损的 (不允许丢包)。

组件 | 组件是需要在端点间传输的特定会话上下文中特定内容类型的编号的数据流。协商每个组件的细节是由传输负责。根据内容类型和内容描述, 一个内容描述可能需要多个组件来通讯 (例如, 音频内容类型也许用两个组件: 一个传输 RTP 流, 另一个传输 RTCP 定时信息)。

{table}

1.1 4. 概念及方法 {anchor:概念和方法}

Jingle 有三部分组成，每部分有自己的语法、语义及状态机：

1. 总会话管理
1. 内容描述格式(“什么”)
1. 内容传输方法(“怎样”)

本文档定义了总会话管理的语义和语法。另有单独的文档，详细说明了用于内容描述和内容传输方法的可插入式“槽(slots)”。基于完整性的考虑，本文档也包含了与描述格式和传输方法有关的全部动作的示例。

从最根本上来说，协商 Jingle 会话的过程是这样的：

1. 一个用户（“发起方”）向另一个用户（“接收方”）发送一个带内容类型的会话请求，会话请求至少包含一个内容类型。
1. 如果接收者想要处理，它会通过发送一个 IQ 结果暂时接受这个请求。
1. 发起方和接收方以尽可能快的速度交换可能的传输候选方法（进一步协商前的传输候选方法的快速发送，是为了缩短媒体数据可流动前的必要时间）。
1. 检查这些传输候选方法的连通性。
1. 一旦接收方找到了媒体数据可流动的候选方法，接收方会向初始方发出一个“会话接受”动作。
1. 双发开始通过协商好的候选方法发送媒体数据。

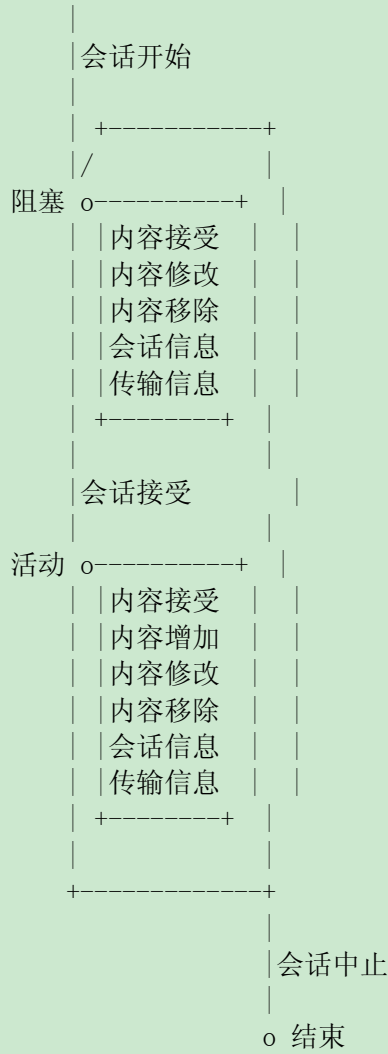
如果双方随后发现了更好的候选方法，他们会进行“内容修改”协商，然后转到这个更好的候选方法上。自然他们也会修改与会话相关的其他参数（如给语音聊天加入视频）。

1.1 4.1 总会话管理 {anchor:总会话管理}

总会话管理的状态机(也即每个 Session ID 的状态)如下：

{code}

o



{code}

总会话状态有三种:

1. 阻塞
1. 活动
1. 结束

与管理总体 Jingle 会话相关的动作如下:

__表 2: Jingle 动作__

{table}

动作|描述

内容接受|接受从另一方受到的内容增加或内容移除。

内容增加|增加一个或多个内容类型到会话中。这个动作 __不能__ 在会话的 __阻塞__ 状态时发出。当一方发出内容增加的时候，它 __必须__ 忽略从另一方收到的任何动作，直到收到内容增加的确认。

内容修改|改变现有的内容类型。接收方 __不能__ 以另一个内容修改来回应内容修改动作。

内容移除|从会话中移除一个或多个内容类型。

会话接受|最终接受会话协商。表明这个动作也适合内容接受（进而适合描述接受和传输接收）。

会话信息|发送会话级的信息/消息，如响铃消息（对 Jingle 音频来说）。

会话开始|请求一个新 Jingle 会话协商。

会话中止|结束现有会话。

传输信息|交换传输候选方法；主要用在 XEP-0176 中，也可以用在其他规范中。

{table}

1.1 5. 会话流 {anchor:会话流}

1.1 5.1 资源确定 {anchor:资源确定}

为了开始 Jingle 会话，发起方必须确定接收方的哪种 XMPP 资源最适合想要的内容描述格式。如果联系方仅有一种 XMPP 资源，这一任务 __必须__ 用服务发现\ [18\]或用在实体能力\ [19\]中说明的基于现身 (presence-based) 的简介 (profile) 的服务发现来完成。

很自然，用实体能力而不是向用户花名册中的每个联系人发送服务发现请求效率更高。由此，某个客户版本对 Jingle 及各种 Jingle 内容 描述格式和内容传输方法的支持等一般性信息（而不是每个的 JID 的）都能确定下来，这些信息随后被缓存。具体细节参考 EXP-0115。

如果联系方有不止一种 XMPP 资源，可能仅有一种资源支持 Jingle 和想要的内容描述格式，在这种情况下，用户 __必须__ 用这一资源初始化 Jingle 信号处理。

如果联系方有超过一种 XMPP 资源支持 Jingle 和想要的内容描述格式， __建议__ 用 ~资源应用优先权 ~ \ [20\]来确定哪种资源最适合初始化 Jingle 会话。

1.1 5.2 初始化 {anchor:初始化}

一旦发起方发现了接收方哪种 XMPP 资源适合想要的内容描述格式，就向接收方发送一个会话初始化请求。这个请求是个包含<jingle/>元素的 IQ 集，元素的命名空间为 '<http://www.xmpp.org/extensions/xep-0166.html#ns>'。<jingle/> 元素 __必须__ 有 'action'、'initiator' 和 'sid' 属性（两个字符唯一区分会话）。初始化时，'action' 属性 __必须__ 是 "session-initiate"，<jingle/> 元素 __必须__ 包含一个或多个<content/>元素，每个元素定义会话期间的要传输的内容类型；每个<content/>元素分别包含<description/>子元素，指定想要的内容描述格式，<transport/>子元素，指定了可能的内容传输方法。如果任何一方想要对相同的内容描述使用多种传输方法，则必须发送多个<content/>元素。

下面是一个 Jingle 会话初始化请求的例子，会话包含了音频和视频：

__例 1. 初始化示例__

```
{code:xml}

<iq from='romeo@montague.net/orchard' to='juliet@capulet.com/balcony' id='jingle1'
type='set'>

  <jingle xmlns='http://www.xmpp.org/extensions/xep-0166.html#ns'
    action='session-initiate'
    initiator='romeo@montague.net/orchard'
    sid='a73sjvkl37jfea'>
    <content creator='initiator' name='this-is-the-audio-content'>
      <description xmlns='http://www.xmpp.org/extensions/xep-0167.html#ns'>
        ...
      </description>
      <transport xmlns='http://www.xmpp.org/extensions/xep-0176.html#ns' />
    </content>
    <content creator='initiator' name='this-is-the-video-content'>
      <description xmlns='http://www.xmpp.org/extensions/xep-0180.html#ns'>
        ...
      </description>
      <transport xmlns='http://www.xmpp.org/extensions/xep-0176.html#ns' />
    </content>
  </jingle>

</iq>

{code}
```

注意：元素<description/>和<transport/>的语法、语义超出了本规范的范围，它们在相关的规范中定义。

元素<jingle/>有如下属性：

- ‘action’属性是__必需的__；它指定了本文中列出 Jingle 动作（如，“session-initiate”）。
- ‘initiator’属性是发起会话流的实体的完整 JID(可能与 IQ 集中地址‘from’不同)。
- ‘reasoncode’属性是__可选的__，指定机器可读的动作发送目的（如，用在会话中止动作的“connectivity-error”）。
- ‘reasoncode’属性是__可选的__，指定人可读的动作发送目的（如，用于会话中止动作的“Sorry,gotta go”）。
- ‘responder’属性（见下面的例子）是回应发起的实体的完整 JID(可能与 IQ 集中‘to’地址不同)。
- ‘sid’属性是由发起方产生的随机的会话识别符；它__应该__符合 XML Nmtoken production[21]，这样对&之类的字符就不需要进行 XML 字符的转义了。（注意：‘sid’属性可有效地映射到 SIP 的‘Call-ID’参数）

元素<content/>有如下属性：

- ‘creator’属性是__必需的__；它指明了那一方最初产生的内容描述(用于防止修改时的竞态条件(race conditions)的发生)。
- ‘name’属性是__必需的__；它指明了内容类型的独特的名字或识别符（这个识别符是不透明的，没有语义上的意义）。
- ‘profile’属性是__推荐的__；对有些内容类型，它说明了所用的简介（如，在实时传输协议上下文中的“RTP/AVP”）
- ‘senders’属性是__推荐的__；它说明了会话中那个实体将要产生内容；允许的值是“initiator”，“recipient”或“both”（缺省值是“both”）。

注意：为了加速会话建立，发起方__可以__在发送“session-initiate”消息后接到接收方响应之前，立即发送传输候选方法（如，用于 ICE 传输的协商），（也就是说，发起方__必须__认定会话是存在的，即使还没有收到响应）。如果按顺序传输的话，接收方应在收到“session-initiate”消息后，应收到诸如“transport-info”之类的消息（如果没收到，那么接收方返回<unknown-session/>错误是恰当的，因为按照它的状态机，会话并不存在）。

1.1 5.3 接收方响应 {anchor:接收方响应}

除非有错误发生，接收方__必须__收到的发起请求：

例 2. 接收方响应发起请求

```
{code:xml}
```

```
<iq type='result' from='juliet@capulet.com/balcony' to='romeo@montague.net/orchard'
id='jingle1' />
```

```
{code}
```

如果接收方响应了发起请求，双方都必须认定会话处在 __阻塞__ 状态。

有几种原因接收方会返回一个错误，而不是响应发起请求：

- 对接收方来说发起方是未知的（比如，通过在线(presence)订阅），接收方不能与未知实体通讯。
- 接收方希望转到另一个地址。
- 接收方不支持 Jingle。
- 接收方不支持任何指定的内容描述格式。
- 接收方不支持任何指定的内容传输方法。
- 发起请求格式是错误的。

如果对接收方来说发起方是未知的（比如，通过在线订阅），并且接收方有不与未知实体经由 Jingle 通讯的策略，则接收方 __应该__ 返回一个<service-unavailable/>错误。

例 3. 发起方未知

```
{code:xml}
```

```
<iq type='error' from='juliet@capulet.com/balcony' to='romeo@montague.net/orchard'
id='jingle1'>
```

```
  <error type='cancel'>
    <service-unavailable xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
  </error>
```

```
</iq>
```

```
{code}
```

如果接收方希望转向另一个地址，它 __应该__ 返回一个<redirect/>错误。

例 4. 接收方转向

```
{code:xml}
```

```
<iq type='error' from='juliet@capulet.com/balcony' to='romeo@montague.net/orchard'
id='jingle1'>
```

```
  <error type='cancel'>
    <redirect xmlns='urn:ietf:params:xml:ns:xmpp-
stanzas'>xmpp:voicemail@capulet.com</redirect>
  </error>
```

```
</iq>
```

```
{code}
```

如果接收方不支持 Jingle, 则 __必须__ 返回一个错误。

例 5. 接收方不支持 Jingle

```
{code:xml}
```

```
<iq type='error' from='juliet@capulet.com/balcony' to='romeo@montague.net/orchard'
id='jingle1'>
```

```
  <error type='cancel'>
    <service-unavailable xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
  </error>
```

```
</iq>
```

```
{code}
```

如果接收方不支持任何一种指定的内容描述格式, 它 __必须__ 返回一个<feature-not-implemented/>错误, 和具有 Jingle 特性的出错条件的<unsupported-content>。

例 6. 接收方不支持任何内容描述格式

```
{code:xml}
```

```
<iq type='error' from='juliet@capulet.com/balcony' to='romeo@montague.net/orchard'
id='jingle1'>
```

```
<error type='cancel'>
  <feature-not-implemented xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
  <unsupported-content xmlns='http://www.xmpp.org/extensions/xep-0166.html#ns-errors' />
</error>

</iq>

{code}
```

如果接收方不支持任何一种指定的内容传输方法，它 __必须__ 返回一个<feature-not-implemented/>错误，和具有 Jingle 特性的出错条件的<unsupported-transport>。

例 7. 接收方不支持任何内容传输方法

```
{code:xml}

<iq type='error' from='juliet@capulet.com/balcony' to='romeo@montague.net/orchard'
id='jingle1'>

  <error type='cancel'>
    <feature-not-implemented xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
    <unsupported-transport xmlns='http://www.xmpp.org/extensions/xep-0166.html#ns-errors' />
  </error>

</iq>

{code}
```

如果发起请求的格式错误，接收方 __必须__ 返回一个<bad-request/>错误。

例 8. 发起请求的格式错误

```
{code:xml}

<iq type='error' from='juliet@capulet.com/balcony' to='romeo@montague.net/orchard'
id='jingle1'>

  <error type='cancel'>
    <bad-request xmlns='urn:ietf:params:xml:ns:xmpp-stanzas' />
  </error>

</iq>

{code}
```

1.1 5.4 拒绝 {anchor:拒绝}

为拒绝会话发起请求，接收方 必须 响应收到的会话发起请求，然后按[中止|XMPP 文档列表/XMPP 扩展/XEP-0166#中止]中描述的方法中止会话。

1.1 5.5 协商 {anchor:协商}

一般情况下，双方在达成可接受的一系列内容类型、内容描述格式和内容传输方法前，协商是必要的。可能要协商的这些参数的组合是很多的，这里并没有列出全部（有些在各种内容描述格式和内容传输方法规范中列出）。

一个会话级的协商是移除一种内容类型。例如，让我们设想，有一天朱丽叶的心情很糟糕，她当然不想在和罗密欧的 Jingle 会话中包含视频，所以她给罗密欧发送了一个“内容移除”请求：

例 9. 内容类型移除

```
{code:xml}

<iq from='juliet@capulet.com/balcony' to='romeo@montague.net/orchard' id='reduce1'
type='set'>

  <jingle xmlns='http://www.xmpp.org/extensions/xep-0166.html#ns'
    action='content-remove'
    initiator='romeo@montague.net/orchard'
    sid='a73sjvkla37jfea'>
    <content creator='initiator' name='this-is-the-video-content'/>
  </jingle>

</iq>

{code}
```

实体收到了这个会话缩减请求，然后响应这个请求：

例 10. 响应

```
{code:xml}
```

```
<iq from='romeo@montague.net/orchard' to='juliet@capulet.com/balcony' id='reduce1'  
type='result' />  
  
{code}
```

如果缩减的结果是会话不再有任何内容类型，收到会话缩减的实体 应该 向另一方发送会话中止动作（因为没有内容类型的会话是无效的）。

另一个会话级的协商是增加一个内容类型；然而，这个动作 必不能 在会话处于 阻塞 状态时来做，只有在会话处于 活动 状态时才可以。

1.1 5.6 接受 {anchor:接受}

协商过内容传输方法和内容描述格式后，如果接收方确定能够建立连接，它将向发起方法送确定接受：

例 11. 接收方确定接受呼叫

```
{code:xml}  
  
<iq type='set' from='juliet@capulet.com/balcony' to='romeo@montague.net/orchard'  
id='accept1'>  
  
  <jingle xmlns='http://www.xmpp.org/extensions/xep-0166.html#ns'  
    action='session-accept'  
    initiator='romeo@montague.net/orchard'  
    responder='juliet@capulet.com/balcony'  
    sid='a73sjjvkl37jfea'>  
    <content creator='initiator' name='this-is-the-audio-content'>  
      <description xmlns='http://www.xmpp.org/extensions/xep-0167.html#ns'>  
        ...  
      </description>  
      <transport xmlns='http://www.xmpp.org/extensions/xep-0177.html#ns'>  
        <candidate .../>  
      </transport>  
    </content>  
  </jingle>  
  
</iq>  
  
{code}
```


<jingle/>元素 __必须__ 包含一个或多个<content/>元素, 后者 __必须__ 包含一个<description>元素和一个<transport/>元素。<jingle/>元素 __应该__ 有 ‘responder’ 属性, 以明确指明响应实体的完整 JID, 有关当前 Jingle 会话的所有通讯, 发起方应该向这个 JID 发送。

然后发起方响应接收方的确认接受:

例 12. 发起方响应确认接受

```
{code:xml}
```

```
<iq type='result' to='juliet@capulet.com/balcony' from='romeo@montague.net/orchard'
id='accept1' />
```

```
{code}
```

此时, 发起方和接收方可以通过协商好的连接开始发送内容了。

如果一方无法找到合适的内容传输方法, 它 __应该__ 下面描述的那样中止会话。

1.1 5.7 修改活动会话 {anchor:修改活动会话}

为修改一个活动会话, 任一方都可向另一方发送“content-remove”、“content-add”、“content-modify”、“description-modify”、“transport-modify”动作。然后接收方发送恰当的“-accept”或“-decline”动作, 也可能首先发送一个“-info”动作。

如果双方同时发送了修改消息, 那么会话发起方的修改消息 __必须__ 胜过接收方的修改消息, 发起方 __应该__ 返回一个<unexpected-request/>错误。

修改活动会话的一个例子是增加一个会话内容。例如, 设想一下朱丽叶的心情好了, 现在想加入视频。于是向罗密欧发送“content-add”请求:

例 13. 增加一个内容类型

```
{code:xml}
```

```
<iq from='juliet@capulet.com/balcony' to='romeo@montague.net/orchard' id='add1' type='set'>
```

```
<jingle xmlns='http://www.xmpp.org/extensions/xep-0166.html#ns'
  action='content-add'
  initiator='romeo@montague.net/orchard'
  sid='a73sjvkla37jfea'>
  <content creator='responder' name='video-is-back'>
    <description xmlns='http://www.xmpp.org/extensions/xep-0180.html#ns'>
      ...
    </description>
    <transport xmlns='http://www.xmpp.org/extensions/xep-0177.html#ns'>
      <candidate .../>
    </transport>
  </content>
</jingle>

</iq>
```

{code}

实体接收到会话扩展请求，响应这个请求，如果可接受，返回一个内容接受：

The entity receiving the session extension request then acknowledges the request and, if it is acceptable, returns a content-accept:

例 14. 响应

{code:xml}

```
<iq from='romeo@montague.net/orchard' to='juliet@capulet.com/balcony' id='add1'
type='result' />
```

{code}

例 15. 内容接受

{code:xml}

```
<iq from='romeo@montague.net/orchard' to='juliet@capulet.com/balcony' id='add2' type='set'>
```

```
<jingle xmlns='http://www.xmpp.org/extensions/xep-0166.html#ns'
  action='content-accept'
  initiator='romeo@montague.net/orchard'
  sid='a73sjvkla37jfea'>
  <content creator='responder' name='video-is-back'>
```

```
<description xmlns='http://www.xmpp.org/extensions/xep-0180.html#ns'>
    ...
</description>
<transport xmlns='http://www.xmpp.org/extensions/xep-0177.html#ns'>
    <candidate .../>
</transport>
</content>
</jingle>

</iq>

{code}
```

另一方响应接受。

例 16. 响应

```
{code:xmk}

<iq from='juliet@capulet.com/balcony' to='romeo@montague.net/orchard' id='add2'
type='result' />

{code}
```

1.1 5.8 中止 {anchor:中止}

为了顺利地结束会话（在响应了初始化请求后的任何时候都 __可以__ 这么做，包括立即想立刻拒绝请求的时候），无论接收方还是初始方都 __必须__ 向对方发送一个“中止”动作。

例 17 接收方中止会话

```
{code:xml}

<iq from='juliet@capulet.com/balcony'

    id='term1'
    to='romeo@montague.net/orchard'
    type='set'>
<jingle xmlns='http://www.xmpp.org/extensions/xep-0166.html#ns'
    action='session-terminate'
    initiator='romeo@montague.net/orchard'
    reason='Sorry, gotta go!'
```

```
sid='a73sjvkl37jfea' />
```

```
</iq>
```

```
{code}
```

另一方(这里是初始方)必须响应会话中止:

例 18. 初始方响应中止

```
{code:xml}
```

```
<iq type='result' to='juliet@capulet.com/balcony' from='romeo@montague.net/orchard'  
id='term1' />
```

```
{code}
```

注意: 一旦实体发送了“会话中止”动作, 它 __必须__ 认定会话已中止(即使在收到对方的响应之前)。如果中止方在发送了“会话中止”动作后收到了对方额外的 IQ 设置, 它 __必须__ 返回一个<unknown-session/>错误。

不幸的是, 并非所有会话都顺利地结束。下面的事件 __必须__ 认定为会话结束事件, 对内容描述格式和内容传输方法的进一步协商 __必须__ 通过协商一个新会话来完成:

- 从对方那里收到‘会话转向’或‘会话中止’动作。
- 从对方那收到<presence type='unavailable' />。

特别地, 如果一方从对方收到的在线 (presence) 类型是“未知 (unavailable)”的话, 则它 __必须__ 认定会话处于 __结束__ 状态。

例 19. 接收方离线

```
{code:xml}
```

```
<presence from='juliet@capulet.com/balcony' to='romeo@montague.net/orchard'  
type='unavailable' />
```

```
{code}
```

自然在这种情况下初始方没什么可响应的。

1.1 5.9 通知消息 {anchor:通知消息}

Jingle 会话开始后的任何时候，任一实体都 __可以__ 向对方发送通知消息，比如，改变内容传输方法或内容描述格式的参数，通知对方一个会话开始请求已经排队等待，设备正在响铃，或一个事先计划的事件已经发生 或将要发生。通知消息 __必须__ 是带有<jingle/>元素的 IQ 设置 (IQ-set) 指令，<jingle/>元素的 'action' 属性的值是 "session-info", "description-info" 或 "transport-info" 之一；<jingle/>元素 __必须__ 进一步包含有效的子元素 (会话，内容描述格式或内容传输方法) 来说明正在交流的信息。如果一个活动会话的任一方收到了一个空的 "session-info" 消息，它 __必须__ 返回一个空的 IQ 结果；这样，一个空的 "session-info" 消息可用作一个 "ping"，来确定会话的活性。(本规范的未来版本也许会定义与 "session-info" 动作相关的内容载荷。)

1.1 6. 出错处理 {anchor:出错处理}

Jingle 专有的出错条件如下。

表 3: 其他出错条件

{table}

Jingle 条件 | xmpp 条件 | 说明

<out-of-order/> | <unexpected-request/> | 请求不可能在状态机的这一点发生 (比如，会话接受后再次初始化)。

<unknown-session/> | <bad-request/> | 指定会话的 'sid' 属性对接收方未知 (例如，根据接收方的状态机会话已经不再有效，因为接收方先前已中止了会话)

<unsupported-content/> | <not-acceptable/> | 接收方不支持任何期望的内容描述格式。

<unsupported-transports/> | <not-acceptable/> | 接收方不支持任何期望的内容传输方法。

{table}

1.1 7. 支持性检测 {anchor:支持性检测}

如果一个实体支持 Jingle，它 __必须__ 在响应 {link:服务发现
<http://www.xmpp.org/extensions/xep-0030.html>} [22] 信息请求时，通过返回特性
“<http://www.xmpp.org/extensions/xep-0166.html#ns>” (见有关 {link:协议命名空间%7C<http://www.xmpp.org/extensions/xep-0166.html#ns>}) 将这一事实公布出去。

例 20. 服务发现信息请求

```
{code:xml}  
  
<iq from='romeo@montague.net/orchard'  
  
    id='disco1'  
    to='juliet@capulet.com/balcony'  
    type='get'>  
  <query xmlns='http://jabber.org/protocol/disco#info' />  
  
</iq>  
  
{code}
```

例 21. 服务发现信息响应

```
{code:xml}  
  
<iq from='juliet@capulet.com/balcony'  
  
    id='disco1'  
    to='romeo@montague.net/orchard'  
    type='result'>  
  <query xmlns='http://jabber.org/protocol/disco#info'>  
    ...  
    <feature var='http://www.xmpp.org/extensions/xep-0166.html#ns' />  
    ...  
  </query>  
  
</iq>  
  
{code}
```

1.1 8. 使用协议的一致性 {anchor:使用协议的一致性}

1.1 8.1 应用类型 {anchor:应用类型}

说明某种 Jingle 应用类型的文档（比如，经由 RTP 的音频）__必须__ 定义：

1. 为封装进 Jingle，如何成功地进行内容协商。
1. 用于表现内容的<description/>元素及相关语义。
1. 能否及怎样将内容描述映射到会话描述协议上。
1. 是通过可靠的还是有损的传输方式（或两者都是）来传输内容。
1. 通过可靠或有损传输来收发内容的精确说明。

1.1 8.2 传输方法 {anchor:传输方法}

说明 Jingle 传输方法的文档（比如，纯 UDP）__必须__ 定义：

1. 为封装进 Jingle，怎样成功地进行传输协商。
1. 用于表现传输类型的<transport/>元素及相关语义。
1. 传输是可靠的还是有损的。
1. 传输是否及怎样处理在这定义的组件（例如，对实时控制协议来说）。

1.1 9. 安全性事项 {anchor:安全性事项}

1.1 9.1 拒绝服务

Jingle 会话可能是资源密集型的。因此，有可能用增加过多 Jingle 会话负担的方法向一个实体发动拒绝服务攻击。必须小心地只从已知的实体那接受内容，并且只接受实体设备能处理的会话。

1.1 9.2 通过网关通讯 {anchor:通过网关通讯}

Jingle 通讯可通过网关与非 XMPP 网络完成，这些网络的安全特性与 XMPP 有很大的不同。（例如，在有些 SIP 网络中鉴定是可选的，“from”地址可轻易伪造。）与这些网通通讯时必须小心。

1.1 IANA 事项 {anchor:IANA 事项}

本文档要求不与 {link:互联网指派数字授权 (IANA) | <http://www.iana.org/>} \[23\] 相互作用。

1.1 11. XMPP 注册处事项 {anchor:XMPP 注册处事项}

1.1 1\1.1 协议命名空间 {anchor:协议命名空间}

在本规范成为草稿状态之前，其相关的命名空间是“<http://www.xmpp.org/extensions/xep-0166.html#ns>”和“<http://www.xmpp.org/extensions/xep-0166.html#ns-errors>”；随着规范的发展，{link:XMPP 注册员 \[24\] | <http://www.xmpp.org/registrar/>} 将按照 {link:XMPP 注册处功能 \[25\] | <http://www.xmpp.org/extensions/xep-0053.html>} 的第四节中定义的过程来发布永久命名空间。

1.1 11.2 Jingle 内容描述格式注册

XMPP 注册处会维护 Jingle 内容描述格式的注册。整个内容描述格式的注册会在单独的规范中定义（不在本文档中）。定义在 XEP 系列里的内容描述格式也__必须__在 XMPP 注册处注册，其结果是协议的 URN 的格式是“urn:xmpp:jingle:description:name”（其中“name”是内容描述格式的注册名）。

为提交注册的新值，注册人须按下面的格式定义一个 XML 段，内容包括相关的 XMPP 扩展协议，或者将它发送到<registrar@xmpp.org>。

```
{code:xml}

<content>

  <name>the name of the content description format</name>
  <desc>a natural-language summary of the content description format</desc>
  <transport>whether the content should be sent over a "reliable" or "lossy"
transport</transport>
  <doc>the document in which this content description format is specified</doc>

</content>

{code}
```

1.1 11.3 Jingle 内容传输方法注册

XMPP 注册处会维护 Jingle 内容传输方法的注册。整个内容传输方法的注册会在单独的规范中定义（不在本文档中）。定义在 XEP 系列里的内容传输方法也__必须__在 XMPP 注册处注册，其结果是协议的 URN 的格式是“urn:xmpp:jingle:transport:name”（其中“name”是内容传输方法的注册名）。

为提交注册的新值，注册人须按下面的格式定义一个 XML 段，内容包括相关的 XMPP 扩展协议，或者将它发送到<registrar@xmpp.org>。

```
{code:xml}

<transport>

  <name>the name of the content transport method</name>
  <desc>a natural-language summary of the content transport method</desc>
  <type>whether the transport method is "reliable" or "lossy"</type>
  <doc>the document in which this content transport method is specified</doc>

</transport>

{code}
```

1.1 11.4 Jingle 原因代码注册 {anchor:Jingle 原因代码注册}

1.1 11.4.1 过程 {anchor:过程}

XMPP 注册处会维护一份 Jingle 动作原因的注册。

为提交注册的新值，注册人须按下面的格式定义一个 XML 段，内容包括相关的 XMPP 扩展协议，或者将它发送到<registrar@xmpp.org>。

```
{code:xml}

<reason>

  the value of the 'reasoncode' attribute</name>
  <desc>a natural-language summary of the reason code</desc>
  <doc>the document in which this reason code is specified</doc>

</reason>

{code}
```

1.1 11.4.2 初始注册 {anchor:初始注册}

下面提交的原因代码注册从 2007 年 4 月开始使用。完整内容和最新的原因代码列表参见注册。

```
{code:xml}

<reason>

  <code>connectivity-error
  <desc>the action (e.g., session-terminate) is related to connectivity problems</desc>
  <doc>XEP-0166</doc>

</reason>

<reason>

  general-error
  <desc>the action (e.g., session-terminate) is related to a non-specific application
error</desc>
  <doc>XEP-0166</doc>

</reason>
```

```
<reason>
```

```
media-error
```

```
<desc>the action (e.g., session-terminate) is related to media processing problems</desc>
```

```
<doc>XEP-0166</doc>
```

```
</reason>
```

```
<reason>
```

```
no-error
```

```
<desc>the action is generated during the normal course of state management</desc>
```

```
<doc>XEP-0166</doc>
```

```
</reason>
```

```
{code}
```

1.1 12. XML 方案(Schemas) {anchor:XML Schemas}

1.1 12.1 Jingle {anchor:Jingle}

```
{code:xml}
```

```
<?xml version='1.0' encoding='UTF-8'?>
```

```
<xs:schema
```

```
  xmlns:xs='http://www.w3.org/2001/XMLSchema'  
  targetNamespace='http://www.xmpp.org/extensions/xep-0166.html#ns'  
  xmlns='http://www.xmpp.org/extensions/xep-0166.html#ns'  
  elementFormDefault='qualified'>
```

```
<xs:element name='jingle'>
```

```
<xs:complexType>
```

```
<xs:sequence minOccurs='1' maxOccurs='unlimited'>
```

```
<xs:element ref='content'/>
```

```
</xs:sequence>
```

```
<xs:attribute name='action' use='required'>
```

```
<xs:simpleType>
```

```
<xs:restriction base='xs:NCName'>
  <xs:enumeration value='content-accept' />
  <xs:enumeration value='content-add' />
  <xs:enumeration value='content-modify' />
  <xs:enumeration value='content-remove' />
  <xs:enumeration value='session-accept' />
  <xs:enumeration value='session-info' />
  <xs:enumeration value='session-initiate' />
  <xs:enumeration value='session-terminate' />
  <xs:enumeration value='transport-info' />
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name='initiator' type='xs:string' use='required' />
<xs:attribute name='reasoncode' type='xs:string' use='optional' />
<xs:attribute name='reasontext' type='xs:string' use='optional' />
<xs:attribute name='responder' type='xs:string' use='optional' />
<xs:attribute name='sid' type='xs:NMTOKEN' use='required' />
</xs:complexType>
</xs:element>

<xs:element name='content'>
  <xs:complexType>
    <xs:choice minOccurs='0'>
      <xs:sequence>
        <xs:any namespace='##other' minOccurs='0' maxOccurs='unbounded' />
      </xs:sequence>
    </xs:choice>
    <xs:attribute name='creator' use='required'>
      <xs:simpleType>
        <xs:restriction base='xs:NCName'>
          <xs:enumeration value='initiator'>
          <xs:enumeration value='responder' />
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name='name' use='required' type='xs:string' />
    <xs:attribute name='profile' use='optional' type='xs:string' />
    <xs:attribute name='senders' use='optional' default='both'>
      <xs:simpleType>
        <xs:restriction base='xs:NCName'>
          <xs:enumeration value='both'>
          <xs:enumeration value='initiator'>
          <xs:enumeration value='responder' />
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
```

```
</xs:schema>
```

```
{code}
```

1.1 12.2 Jingle 出错信息 {anchor:Jingle 出错信息}

```
{code:xml}
```

```
<?xml version='1.0' encoding='UTF-8'?>
```

```
<xs:schema
```

```
  xmlns:xs=' http://www.w3.org/2001/XMLSchema'
  targetNamespace=' http://www.xmpp.org/extensions/xep-0166.html#ns-errors'
  xmlns=' http://www.xmpp.org/extensions/xep-0166.html#ns-errors'
  elementFormDefault='qualified'>
```

```
  <xs:element name='out-of-order' type='empty' />
  <xs:element name='unknown-session' type='empty' />
  <xs:element name='unsupported-content' type='empty' />
  <xs:element name='unsupported-transport' type='empty' />
```

```
  <xs:simpleType name='empty'>
    <xs:restriction base='xs:string'>
      <xs:enumeration value=''/>
    </xs:restriction>
  </xs:simpleType>
```

```
</xs:schema>
```

```
{code}
```




XMPP

XEP-0166: Jingle

Scott Ludwig

<mailto:scottlu@google.com>

<xmpp:scottlu@google.com>

Joe Beda

<mailto:jbeda@google.com>

<xmpp:jbeda@google.com>

Peter Saint-Andre

<mailto:stpeter@jabber.org>

<xmpp:stpeter@jabber.org>

<https://stpeter.im/>

Robert McQueen

<mailto:robert.mcqueen@collabora.co.uk>

<xmpp:robert.mcqueen@collabora.co.uk>

Sean Egan

<mailto:seanegan@google.com>

<xmpp:seanegan@google.com>

Joe Hildebrand

<mailto:jhildebr@cisco.com>

<xmpp:hildjj@jabber.org>

2009-12-23

Version 1.1

Status	Type	Short Name
Draft	Standards Track	jingle

This specification defines an XMPP protocol extension for initiating and managing peer-to-peer media sessions between two XMPP entities in a way that is interoperable with existing Internet standards. The protocol provides a pluggable model that enables the core session management semantics (compatible with SIP) to be used for a wide variety of application types (e.g., voice chat, video chat, file transfer) and with a wide variety of transport methods (e.g., TCP, UDP, ICE, application-specific transports).

6.6 Sctp 协议



SCTP协议介绍



目 录

4 SCTP 协议	4-1
4.1 概述	4-2
4.2 SCTP 相关术语	4-2
4.3 SCTP 功能	4-5
4.3.1 偶联的建立和关闭	4-6
4.3.2 流内消息顺序递交	4-6
4.3.3 用户数据分段	4-7
4.3.4 证实和避免拥塞	4-7
4.3.5 消息块绑定	4-7
4.3.6 分组的有效性	4-7
4.3.7 通路管理	4-7
4.4 SCTP 原语	4-8
4.4.1 SCTP 用户向 SCTP 发送的请求原语	4-8
4.4.2 SCTP 向 SCTP 用户发送的通知原语	4-10
4.5 SCTP 协议消息	4-12
4.5.1 消息结构	4-12
4.5.2 SCTP 数据块的格式	4-16
4.5.3 SCTP 端点维护的参数和建议值	4-31
4.6 SCTP 基本信令流程	4-34
4.6.1 偶联的建立和发送流程	4-34
4.6.2 偶联关闭流程	4-37

中华人民共和国通信行业标准 流控制传送协议(SCTP)

YD/T 1194-2002

前 言

本标准是根据 RFC 2960(2000)建议制定的,它规定了流控制传送协议(SCTP)所使用的消息格式编码和程序。SCTP 协议主要用于在 IP 网中传送 PSTN 的信令消息,同时 SCTP 协议还可以用于其他的信息在 IP 网内传送。

本标准的附录 A、附录 B 和附录 C 是资料性的附录。

本标准由信息产业部电信研究院提出并归口。

本标准起草单位:信息产业部电信传输研究所
深圳市中兴通讯股份有限公司
华为技术有限公司
上海贝尔有限公司

本标准主要起草人:吕 军 续合元 张 宜 高 峰 走 超 林 铭 吕 严
中华人民共和国信息产业部 2002-06-21 发布 2002-06-21 实施

1 范围

本标准规定了流控制传送协议(SCTP)所使用的消息格式编码和程序, SCTP 协议主要用于在 IP 网中传送 PSTN 的信令消息和 IP 网内的信令消息。

本标准主要适用于完成 No. 7 信令与 IP 网互通的信令网关(SG)设备,以及 IP 网用于呼叫控制的软交换(Soft-Switch)交换机等设备的开发、生产、引进和购买。

2 规范性引用文件

下列文件中的条款通过在本标准中引用而成为本标准的条款,凡是注日期的引用文件,其随后所有的修改单(不包括勘误的内容)或修订版均不适用于本部分,然而鼓励根据本部分达成协议的各方研究是否可以使用这些文件的最小版本。凡是不注日期的引用文件,其最新版本适用于本标准。

- RFC 793 传送控制协议(TCP)
- RFC 1191 发现通路 MTU
- RFC 1123 对 Internet 主机的要求—应用和支持
- RFC 1700 分配的号码
- RFC 1750 出于安全目的的随机建议
- RFC 1981 用于 IPv6 的发现通路 MTU
- RFC 1982 串号的算法
- RFC 2373 IPv6 的地址结构
- RFC 2401 Internet 协议的安全结构
- RFC 2460 Internet 协议,版本 6
- RFC 2481 明确的拥塞通知
- RFe 2581 TCP 拥塞控制
- RFC 2960 流传送控制协议(SCTP)

3 名词术语和缩略语

6.7 Rtp 协议

RTP: 实时应用程序传输协议

摘要

本文描述 RTP (real-time transport protocol), 实时传输协议。RTP 在多点传送 (多播) 或单点传送 (单播) 的网络服务上, 提供端对端的网络传输功能, 适合应用程序传输实时数据, 如: 音频, 视频或者仿真数据。RTP 没有为实时服务提供资源预留的功能, 也不能保证 QoS (服务质量)。数据传输功能由一个控制协议 (RTCP) 来扩展, 通过扩展, 可以用一种方式对数据传输进行监测控制, 该协议 (RTCP) 可以升级到大型的多点传送 (多播) 网络, 并提供最小限度的控制和鉴别功能。RTP 和 RTCP 被设计成和下面的传输层和网络层无关。协议支持 RTP 标准的转换器和混合器的使用。

本文的大多数内容和旧版的 RFC1889 相同。在线路里传输的数据包格式没有改变, 唯一的改变是使用协议的规则和控制算法。为了最小化传输, 发送 RTCP 数据包时超过了设定的速率, 而在这时, 很多的参与者同时加入了一个会话, 在这样的情况下, 一个新加入到 (用于计算的可升级的) 计时器算法中的元素是最大的改变。

目录 (Table of Contents)

1. 引言 (Introduction)
 - 1 1 术语 (Terminology)
- 2 RTP 使用场景 (RTP Use Scenarios)
 - 2 1 简单多播音频会议 (Simple Multicast Audio Conference)
 - 2 2 音频和视频会议 (Audio and Video Conference)
 - 2 3 混频器和转换器 (Mixers and Translators)
 - 2 4 分层编码 (Layered Encodings)
- 3 定义 (Definitions)
- 4 字节序, 校正和时间格式 (Byte Order, Alignment, and Time Format)
- 5 RTP 数据传输协议 (RTP Data Transfer Protocol)
 - 5 1 RTP 固定头域 (RTP Fixed Header Fields)
 - 5 2 多路复用 RTP 会话 (Multiplexing RTP Sessions)
 - 5 3 RTP 头的配置文件详细变更 (Profile-Specific Modifications to the RTP Header)
 - 5 3 1 RTP 报头扩展 (RTP Header Extension)
- 6 RTP 控制协议 (RTP Control Protocol) -- RTCP
 - 6 1 RTCP 包格式 (RTCP Packet Format)
 - 6 2 RTCP 传输间隔 (RTCP Transmission Interval)
 - 6 2 1 维护会话成员数目 (Maintaining the number of session members)
 - 6 3 RTCP 包的发送与接收规则 (RTCP Packet Send and Receive Rules)
 - 6 3 1 计算 RTCP 传输间隔 (Computing the RTCP Transmission Interval)
 - 6 3 2 初始化 (Initialization)
 - 6 3 3 接收 RTP 或 RTCP (非 BYE) 包 (Receiving an RTP or Non-BYE RTCP Packet)
 - 6 3 4 接收 RTCP (BYE) 包 (Receiving an RTCP BYE Packet)
 - 6 3 5 SSRC 计时失效 (Timing Out an SSRC)
 - 6 3 6 关于传输计时器的到期 (Expiration of Transmission Timer)
 - 6 3 7 传输一个 BYE 包 (Transmitting a BYE Packet)
 - 6 3 8 更新 we_sent (Updating we_sent)
 - 6 3 9 分配源描述带宽 (Allocation of Source Description Bandwidth)
 - 6 4 发送方和接收方报告 (Sender and Receiver Reports)

7 附件:

7.1 Gyp 工具

GYP 简介:转载自: <http://blog.xiaogaozi.org/2011/10/29/introduction-to-gyp/>

说起项目构建工具, Linux 用户最熟悉的恐怕就是 [Autotools](#), 它将编译安装这个步骤大大简化。但对于项目作者来说, 想要使用 Autotools 生成有效的配置文件着实需要下一番功夫, 用现在流行的话来说就是用户体验不够友好。对 Unix shell 的依赖, 也使得 Autotools 天生对于跨平台支持不佳。

与其类似的有 [CMake](#), CMake 使用 C++ 编写, 原生支持跨平台, 不需要像 Autotools 那样写一堆的配置文件, 只需一个 CMakeLists.txt 文件即可。简洁的使用方式, 强大的功能使得我立马对 CMake 情有独钟。在后来的使用过程中, 虽然会遇到一些因为使用习惯带来的小困扰, 但我对于 CMake 还是基本满意的。直到我发现了 GYP。

[GYP](#) (Generate Your Projects) 是由 Chromium 团队开发的跨平台自动化项目构建工具, Chromium 便是通过 GYP 进行项目构建管理。为什么我要选择 GYP, 而放弃 CMake 呢? 功能上 GYP 和 CMake 很是相似, 在我看来, 它们的最大区别在于配置文件的编写方式和其中蕴含的思想。

编写 CMake 配置文件相比 Autotools 来说已经简化很多, 一个最简单的配置文件只需要写上源文件及生成类型(可执行文件、静态库、动态库等)即可。对分支语句和循环语句的支持也使得 CMake 更加灵活。但是, CMake 最大的问题也是在这个配置文件, 请看下面这个示例文件:

```
1 cmake_minimum_required(VERSION 2.8)
2 project(VP8 CXX)
3
4 add_definitions(-Wall)
5 cmake_policy(SET CMP0015 NEW)
6 include_directories("include")
7 link_directories("lib")
8 set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY "../lib")
9 set(VP8SRC VP8Encoder.cpp VP8Decoder.cpp)
10
11 if(X86)
12     set(CMAKE_SYSTEM_NAME Darwin)
13     set(CMAKE_SYSTEM_PROCESSOR i386)
14     set(CMAKE_OSX_ARCHITECTURES "i386")
15
16     add_library(vp8 STATIC ${VP8SRC})
17 elseif(IPHONE)
18     if(SIMULATOR)
19         set(PLATFORM "iPhoneSimulator")
20         set(PROCESSOR i386)
21         set(ARCH "i386")
22     else()
23         set(PLATFORM "iPhoneOS")
24         set(PROCESSOR arm)
25         set(ARCH "armv7")
26     endif()
27
28     set(SDKVER "4.0")
29     set(DEVROOT "/Developer/Platforms/${PLATFORM}.platform/Developer")
30     set(SDKROOT "${DEVROOT}/SDKs/${PLATFORM}${SDKVER}.sdk")
31     set(CMAKE_OSX_SYSROOT "${SDKROOT}")
32     set(CMAKE_SYSTEM_NAME Generic)
33     set(CMAKE_SYSTEM_PROCESSOR ${PROCESSOR})
34     set(CMAKE_CXX_COMPILER "${DEVROOT}/usr/bin/g++)")
35     set(CMAKE_OSX_ARCHITECTURES ${ARCH})
36
37     include_directories(SYSTEM "${SDKROOT}/usr/include")
38     link_directories(SYSTEM "${SDKROOT}/usr/lib")
39
40     add_definitions(-D_PHONE)
```



```
41 add_library(vp8-armv7-darwin STATIC ${VP8SRC})
42 endif()
```

你能一眼看出这个配置文件干了什么吗？其实这个配置文件想要产生的目标（target）只有一个，就是通过 `${VP8SRC}` 编译生成的静态库，但因为加上了条件判断，及各种平台相关配置，使得这个配置文件看起来很是复杂。在我看来，编写 CMake 配置文件是一种线性思维，对于同一个目标的配置可能会零散分布在各个地方。而 GYP 则相当不同，GYP 的配置文件更多地强调模块化、结构化。看看下面这个示例文件：

```
1 {
2   'targets': [
3     {
4       'target_name': 'foo',
5       'type': '<(library)',
6       'dependencies': [
7         'bar',
8       ],
9       'defines': [
10        'DEFINE_FOO',
11        'DEFINE_A_VALUE=value',
12      ],
13       'include_dirs': [
14         '..',
15       ],
16       'sources': [
17         'file1.cc',
18         'file2.cc',
19       ],
20       'conditions': [
21         ['OS=="linux"', {
22           'defines': [
23             'LINUX_DEFINE',
24           ],
25           'include_dirs': [
26             'include/linux',
27           ],
28         }],
29         ['OS=="win"', {
30           'defines': [
31             'WINDOWS_SPECIFIC_DEFINE',
32           ],
33         }, { # OS != "win",
34           'defines': [
35             'NON_WINDOWS_DEFINE',
36           ],
37         }],
38       ],
39     }
40   ],
41 }
```

我们可以立马看出上面这个配置文件的输出目标只有一个，也就是 `foo`，它是一个库文件（至于是静态的还是动态的这需要在生成项目时指定），它依赖的目标、宏定义、包含的头文件路径、源文件是什么，以及根据不同平台设定的不同配置等。这种定义配置文件的方式相比 CMake 来说，让我觉得更加舒服，也更加清晰，特别是当一个输出目标的配置越来越多时，使用 CMake 来管理可能会愈加混乱。

配置文件的编写方式是我区分 GYP 和 CMake 之间最大的不同点，当然 GYP 也有一些小细节值得注意，比如支持跨平台项目工程文件输出，Windows 平台默认是 Visual Studio，Linux 平台默认是 Makefile，

Mac 平台默认是 Xcode，这个功能 CMake 也同样支持，只是缺少了 Xcode。Chromium 团队成员也撰文详细比较了 GYP 和 CMake 之间的优缺点，在开发 GYP 之前，他们也曾试图转到 SCons（这个我没用过，有经验的同学可以比较一下），但是失败了，于是 GYP 就诞生了。

当然 GYP 也不是没有缺点，相反，我觉得它的「缺点」一大堆：

文档不够完整，项目不够正式，某些地方还保留着 Chromium 的影子，看起来像是还没有完全独立出来。大量的括号嵌套，很容易让人看晕，有过 Lisp 使用经验的同学可以对号入座。对于有括号恐惧症，或者不使用现代编辑器的同学基本可以绕行。

为了支持跨平台，有时不得不加入某些特定平台的配置信息，比如只适用于 Visual Studio 的 RuntimeLibrary 配置，这不利于跨平台配置文件的编写，也无形中增加了编写复杂度。

不支持 make clean，唯一的方法就是将输出目录整个删除或者手动删除其中的某些文件。

如果你已经打算尝试 GYP，那一定记得在生成项目工程文件时加上 --depth 参数，譬如：

```
$ gyp --depth=. foo.gyp
```

这也是一个从 Chromium 项目遗留下来的历史问题。

7.2 Google test 程序

[玩转 Google 开源 C++单元测试框架 Google Test 系列\(gtest\)\(总\)](#)

玩转 Google 开源 C++单元测试框架 Google Test 系列(gtest)之一 - 初识 gtest

一、前言

本篇将介绍一些 gtest 的基本使用，包括下载，安装，编译，建立我们第一个测试 Demo 工程，以及编写一个最简单的测试案例。

二、下载

如果不记得网址，直接在 google 里搜 gtest 第一个就是。目前 gtest 的最新版本为 1.3.0，从下列地址可以下载到该最新版本：

<http://googletest.googlecode.com/files/gtest-1.3.0.zip>

<http://googletest.googlecode.com/files/gtest-1.3.0.tar.gz>

<http://googletest.googlecode.com/files/gtest-1.3.0.tar.bz2>

三、编译

下载解压后，里面有个 msvc 目录：

7.3 Webrtc 库介绍

trunk\webrtc\modules

视频采集---video_capture

源代码在 webrtc/modules/video_capture/main 目录下，包含接口和各个平台的源代码。

在 windows 平台上，WebRTC 采用的是 dshow 技术，来实现枚举视频的设备信息和视频数据的采集，这意味着可以支持大多数的视频采集设备；对那些需要单独驱动程序的视频采集卡（比如海康高清卡）就无能为力了。

视频采集支持多种媒体类型，比如 I420、YUY2、RGB、UYUY 等，并可以进行帧大小和帧率控制。

视频编解码---video_coding

源代码在 webrtc/modules/video_coding 目录下。

WebRTC 采用 I420/VP8 编解码技术。VP8 是 google 收购 ON2 后的开源实现，并且也用在 WebM 项目中。VP8 能以更少的数据提供更高质量的视频，特别适合视频会议这样的需求。

视频加密--video_engine_encryption

视频加密是 WebRTC 的 video_engine 一部分，相当于视频应用层面的功能，给点对点的视频双方提供了数据上的安全保证，可以防止在 Web 上视频数据的泄漏。

视频加密在发送端和接收端进行加解密视频数据，密钥由视频双方协商，代价是会影响视频数据处理的性能；也可以不使用视频加密功能，这样在性能上会好些。

视频加密的数据源可能是原始的数据流，也可能是编码后的数据流。估计是编码后的数据流，这样加密代价会小一些，需要进一步研究。

视频媒体文件--media_file

源代码在 webrtc/modules/media_file 目录下。

该功能是可以本地文件作为视频源，有点类似虚拟摄像头的功能；支持的格式有 Avi。

另外，WebRTC 还可以录制音视频到本地文件，比较实用的功能。

视频图像处理--video_processing

源代码在 webrtc/modules/video_processing 目录下。

视频图像处理针对每一帧的图像进行处理，包括明暗度检测、颜色增强、降噪处理等功能，用来提升视频质量。

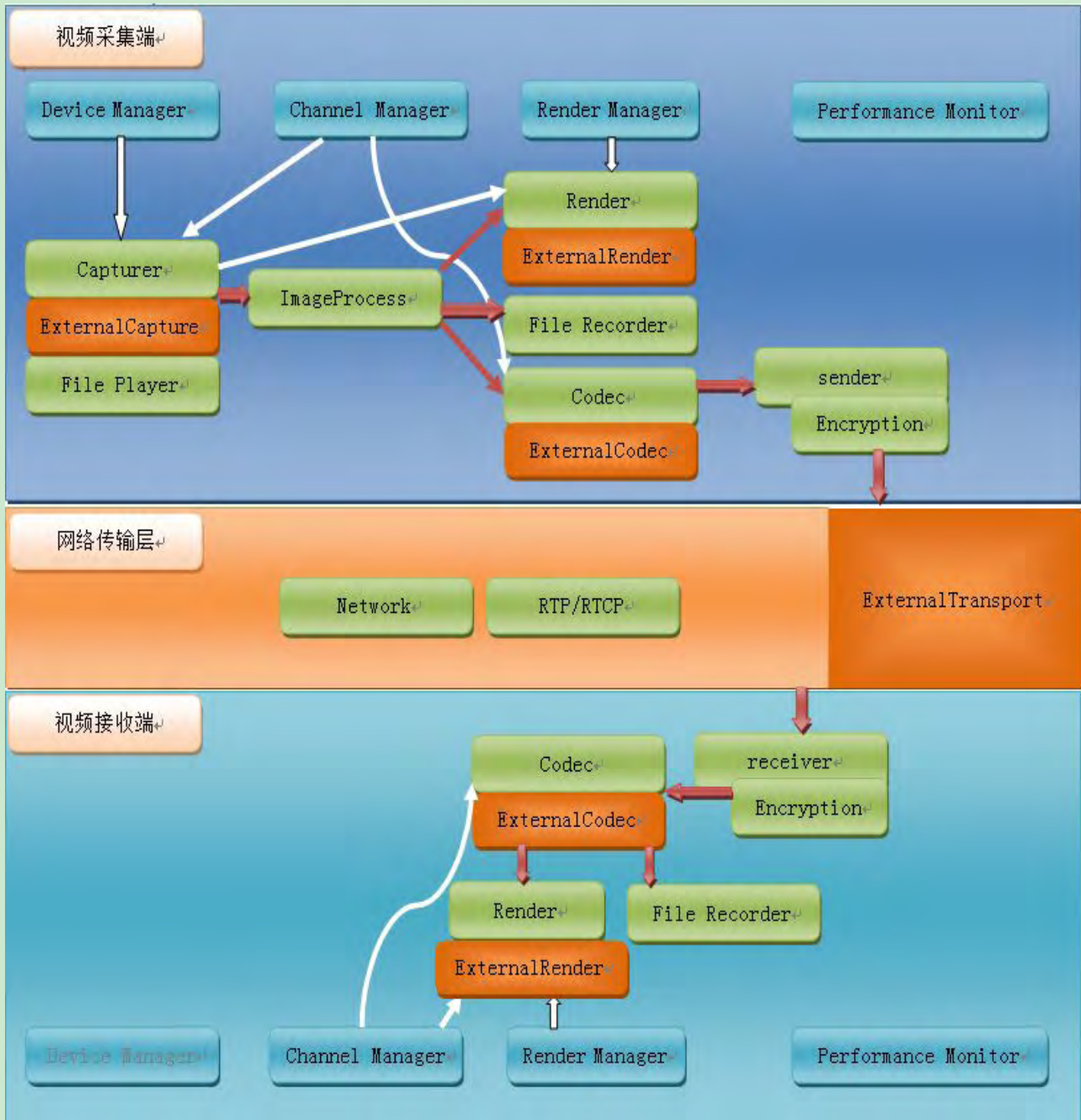
视频显示--video_render

源代码在 webrtc/modules/video_render 目录下。

在 windows 平台，WebRTC 采用 direct3d9 和 directdraw 的方式来显示视频，只能这样，必须这样。

网络传输与流控

对于网络视频来讲，数据的传输与控制是核心价值。WebRTC 采用的是成熟的 RTP/RTCP 技术。



7.4 webrtc 代码相关基础知识

<http://blog.csdn.net/chenyufei1013/article/category/1248211>

libjingle 源码分析之一：Signal 机制

分类： libjingle 2012-10-18 17:18 1133 人阅读 评论(3) 收藏 举报

[signalclass](#) [文档](#) [parametersfunctionstring](#)

• 摘要

本文主要分析了 libjingle 中的 Signal（信号）机制，它实际上是基于 sigslot 开源库。本文开始描述了 Signal 机制是什么；然后，给出一个 libjingle 文档中的例子，来描述它是如何使用的。最后，介绍了 Signal 机制的具体实现。

• 概述

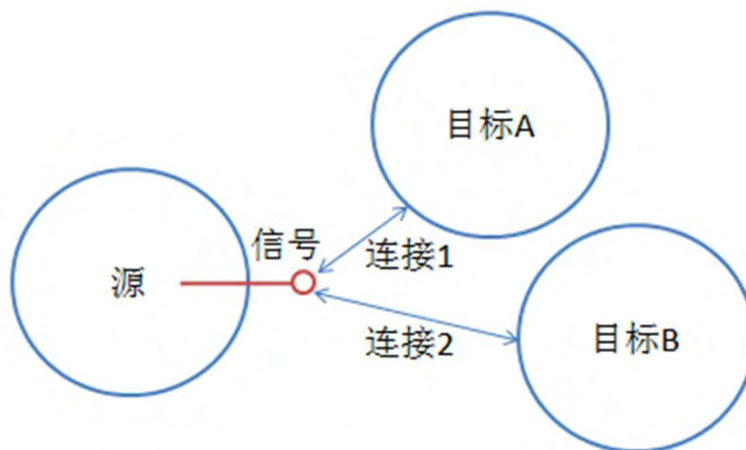
按照 libjingle 文档关于 Signal

（https://developers.google.com/talk/libjingle/important_concepts#signals）的介绍，Signal 机制实际上采用的是 sigslot 开源库

（<http://sourceforge.net/projects/sigslot/?source=directory>）。sigslot 是一个开源的回调框架，它可以使得类之间的回调使用的简单化，下面是 libjingle 文档中对 sigslot 的描述。

sigslot is a generic framework that enables you to connect a calling member to a receiving function in any class (including the same class) very simply.

Signal 机制的工作方式参见下图的描述。源中设置一个或多个信号，目标为了在源的信号触发时获获得通知，需要连接到信号上。可以有多个目标发起连接，也可以同一个目标发起多个连接。连接创建好之后，源触发信号时，目标 A 和目标 B 就可以收到信号触发的消息了。



7.5 STUN 和 TURN 技术浅析

STUN 和 TURN 技术浅析

原文:

http://www.h3c.com.cn/MiniSite/Technology_Circle/Net_Reptile/The_Five/Home/Catalog/201206/747038_97665_0.htm

在现实 Internet 网络环境中,大多数计算机主机都位于防火墙或 NAT 之后,只有少部分主机能够直接接入 Internet。很多时候,我们希望网络中的两台主机能够直接进行通信,即所谓的 P2P 通信,而不需要其他公共服务器的中转。由于主机可能位于防火墙或 NAT 之后,在进行 P2P 通信之前,我们需要进行检测以确认它们之间能否进行 P2P 通信以及如何通信。这种技术通常称为 NAT 穿透 (NAT Traversal)。最常见的 NAT 穿透是基于 UDP 的技术,如 RFC3489 中定义的 STUN 协议。

STUN,首先在 RFC3489 中定义,作为一个完整的 NAT 穿透解决方案,英文全称是 Simple Traversal of UDP Through NATs,即简单的用 UDP 穿透 NAT。

在新的 RFC5389 修订中把 STUN 协议定位于为穿透 NAT 提供工具,而不是一个完整的解决方案,英文全称是 Session Traversal Utilities for NAT,即 NAT 会话穿透效用。RFC5389 与 RFC3489 除了名称变化外,最大的区别是支持 TCP 穿透。

TURN,首先在 RFC5766 中定义,英文全称是 Traversal Using Relays around NAT:Relay Extensions to Session Traversal Utilities for NAT,即使用中继穿透 NAT:STUN 的扩展。简单的说,TURN 与 STUN 的共同点都是通过修改应用层中的私网地址达到 NAT 穿透的效果,异同点是 TURN 是通过两方通讯的“中间人”方式实现穿透。

1 STUN

了解 STUN 之前,我们需要了解 NAT 的种类。

NAT 对待 UDP 的实现方式有 4 种,分别如下:

1. Full Cone NAT

完全锥形 NAT,所有从同一个内网 IP 和端口号发送过来的请求都会被映射成同一个外网 IP 和端口号,并且任何一个外网主机都可以通过这个映射的外网 IP 和端口号向这台内网主机发送包。

2. Restricted Cone NAT

7.6 基于 ICE 的 VoIP 穿越 NAT 改进方案

1 引言

近年来,随着数据网络通信逐渐融入传统的话音业务领域,VoIP 技术越来越成为当前商业考虑的对象,并正在向一种正式的商业电话模式演进,而会话初始协议(SIP, Session Initiation Protocol)就是用来确保这种演进能够实现而需要的 NGN(下一代网络)系列协议中重要的一员。SIP 是一个用于建立、更改和终止多媒体会话的应用层控制协议。SIP 因其简单、灵活、可扩展性强的特点,已经成为实现 VoIP 系统的热点技术。

随着计算机网络技术的不断发展,互联网规模飞速膨胀,大量企业和驻地网采用了私有网络通过 NAT/防火墙出口来接入公共网络。而由于 SIP 包头中含有很多对于路由、接续 SIP 信令和建立呼叫连接必不可少的地址信息,这样引发了业界对于 SIP2 穿越 NAT/防火墙问题的研究。

目前, IETF 已经对该问题提出了多种解决方案。例如: ALGs(Application Layer Gateways)、Middlebox Control Protocol、STUN(Simple Traversal of UDP through NAT)、TURN(Traversal Using Relay NAT)、RSIP(Realm Specific IP)、Symmetric RTP 等。然而,当这些技术应用于不同的网络拓扑时都有着显著的利弊,以至于只能根据不同的接入方式来应用不同的方案,所以,未能很好地解决 A11-NAT 的问题,同时还会给系统引入许多复杂性和脆弱性因素。此外,由于 NAT/防火墙已经大量应用, SIP 设备也已经比较成熟,对它们进行升级来支持多媒体通信穿越 NAT/防火墙的代价将相当的大。因此,一种不需要升级任何现有网络设备,能够穿越各种 NAT/防火墙并且方便在现有网络中实施的解决方案成为迫切的需要。

本文试图寻找一种能够穿越各种类型的 NAT/防火墙,无需对现有 NAT/防火墙设备做任何改动的解决方案——ICE 解决方案,这种方式比以前的解决方案更加灵活,具有广阔的应用前景。

2 现有 NAT 解决方案的比较分析

主流的 NAT 穿越解决方案包括 STUN、TURN、Proxy 及隧道穿越等,这几种方式各具优缺点,比较如下:

(1) STUN(simple traversal of UDP over NAT)的原理是通过某种机制预先得内部私有 IP 地址对应在出口 NAT 上的对外公网 IP 地址,然后在报文负载中所描述的地址信息就直接填写出口 NAT 上的对外 IP 地址。其最大的优点是无需对现有 NAT/防火墙设备做任何改动。局限性在于需要应用程序支持 STUN CLIENT 的功能,同时 STUN 并不适合支持 TCP 连接的穿越。

(2) TURN 即通过 Relay 方式穿越 NAT,也是私网中的 SIP 终端通过某种机制预先得 TURN Server 上的公网地址,私网终端发出的报文都要经过 TURN Server 进行 Relay 转发。这种方式除了具有 STUN 方式的优点外,还解决了 STUN 应用无法穿透对称 NAT(Symmetric NAT)以及类似的 Firewall 设备的缺陷,局限性在于需要 SIP 终端支持 TURN Client,并增大了包的延迟和丢包的可能性。

(3) Proxy 方式是指通过对私网内用户呼叫的信令和媒体流做 Relay 来实现对 NAT/防火墙的穿越。由于不用对运营商和客户端的现有网络设备进行任何改造,具有很强的适应性,组网灵活,可满足 NGN 初期多样化的组网和用户接入。

(4) 隧道穿越技术的基本思想是通过把需要穿越的数据流封装在某种隧道中,从而绕过 NAT/防火墙。它在很大程度上解决了对于不同应用协议需要开发不同穿越策略的办法,但是必须多媒体终端和服务器能够支持隧道,这是一个比较大的限制条件。

3 穿越 NAT/防火墙方案的实现

3.1 ICE 方式

交互式连通建立方式 ICE(Interactive Connectivity Establishment)并非一种新的协议,它不需要对 STUN, TURN 或 RSIP 进行扩展就可适用于各种 NAT。ICE 是通过综合运用上面某几种协议,使之在最合适的情况下工作,以弥补单独使用其中任何一种所带来的固有缺陷。对于 SIP 来说,ICE 只需要定义一些 SDP(Session Description Protocol)附加属性即可,对于别的多媒体信令协议也需要制定一些相应的机制来实现。本文是针对 SIP 呼叫流程实现 ICE 的功能。

这种方式的优点是可以根据通讯双方所处的网络环境,选取适合穿越 NAT/防火墙的方式。首先,获取用户所征网络中 NAT 的类型,如果用户没有设置使用何种方式连接,那么默认首先使用 UDP 连接,如果一定时间内没有连接成功,接着使用 TCP 连接,同样如果没有在一定时间内连接成功,那么将采用其他方式如 Upnp、Http tunnel。如果所有穿越方案都失败后,将结果返回给用户,由用户决定是否重试。

3.2 ICE 算法流程

ICE 算法流程分为以下几个过程:

(1) 收集本地传输地址

会话者从服务器上获得主机上一个物理(或虚拟)接口绑定一个端口的本地传输地址。

(2) 启动 STUN

与传统的 STUN 不同,ICE 用户名和密码可以通过信令协议进行交换。

(3) 确定传输地址的优先级

优先级反映了 UA 在该地址上接收媒体流的优先级别,取值范围 0 到 1 之间,按照被传输媒体流量来确定。

(4) 构建初始化信息(Initiate Message)

初始化消息由一系列媒体流组成,每个媒体流的任意 Peer 之间实现最人连通可能性的传输地址是由公网 L 转发服务器(如 TURN)提供的地址。

(5) 响应处理

连通性检查和执行 ICE 算法中描述的地址收集过程。

(6) 生成接受信息(Accept Message)

若接受则发送 Accept 消息,其构造过程与 InitiateMessage 类似。

(7) 接受信息处理

接受过程需要发起者使用 Send 命令,由服务器转发至响应者。

(8) 附加 ICE 过程

Initiate 或 Accept 消息交换过程结束后,双方可能仍将继续收集传输地址。


3.3 ICE 算法实现

当将 ICE 算法集成到 SIP 呼叫过程的时候, 流程应该是: (1)SIP 终端注册, 并且访问 STUN(STUNT) 服务器, 判断 NAT/防火墙类型, 以及 TCP 时三种序列的包的过滤情况。(2)当发起呼叫信息(INVITE)或接收到呼叫信息回应(200 OK)之前根据 NAT/防火墙类型进行对 RTP 进行地址收集(任非对称性 NAT/防火墙后需要收集 NAT 映射地址, 在对称性 NAT/防火墙后还需要收集 TURN 地址)。(3)在 RTP 的地址端口启动接收线程 RSTUN 服务程序。(4)发送 SIP 消息, 收集的地址放列 SDP 消息中的 alt 属性中。(5)SIP 终端得到通讯双方地址后进行地址配对(将双方地址进行组合), 并且根据双方网络情况去掉无效的地址对。(6)根据地址对发

送 STUN check 的包, 其中 STUN 消息的用户名, 密码从 alt 属性中得到, 标识该地址对。(7)当检测到有效的地址对时(可以发送 RTP 媒体流的地址), 停止接收线程 STUN 服务), 开始传输 RTP 流。

本文实现采用 Winpcap API 首先捕获 TCP 连接的 SYN--out 包, 修改 IP 包头的 TTL 的值, 用 pcap—sendpacket()。然后使该 socket 调用 listen 函数。实现过程中对应于 ICE 收集地址的算法描述为:

```
class ICECandidate
{
public:
    ICECandidate();
    ICECandidate & operator =(const ICECandidate
&other);
public:
    int m_nCandidateID;    //candidate ID
    int m_nPriority;
    unsigned char m_slid[64];
    bool m_NewCandi; //when check this can be
true,
    ICEAddress m_CandidateAddr;
private:
    bool m_bTurnCandidate;
    static int m_num;
    char m_sAddr[50];
};
```



类中 m_nCandidateID 对应地址序号, m_nPriority 表示地址优先级, m_CandidateAddr 表示地址(IP 地址, 端口)。实现 ICE 算法的实体算法描述为:

```
class CTraICE
{
public:
    CTraICE(),
    CTraICE(char*sRtpaddress, int nPort),
    virtual ~CTraICE(),
private:
    int m_nLRtpPort,
    CString m_sLRtpaddr, // 本地 rtp 地址
    int m_nRRtpPort,
    CString m_sRRtpaddr, // 返回可以连接的 rtp 地址
    int m_nLocalltype, // 本地网络类型
    int m_nRemoteType, // 远程网络类型
    int m_bCaller, // 1 为呼叫方, 2 为被呼叫方
    int m_bStopFlag, // 线程是否终止

    SOCKET m_rtpsocket, //local rtp socket!!
    ///for the check conectivity!!!
    vector<ICEPair> pairs,
    vector<ICECandidate> LocalCandidates,
    vector<ICECandidate> RemoteCandidates,
public:
    void Init(),
    bool StartStunLisener(),
    void StopStunLisener(),
    void WaitStunLisener(),
    void SendICECheck(),
    bool InitPair(), // 构造 pairs 变量!!
```

实现 ICE 中会话发起者和接收者的步骤基本一样, 仅任处理流程中先后次序稍微有些不同, 本文中实现的会话流程如图 1 所示。

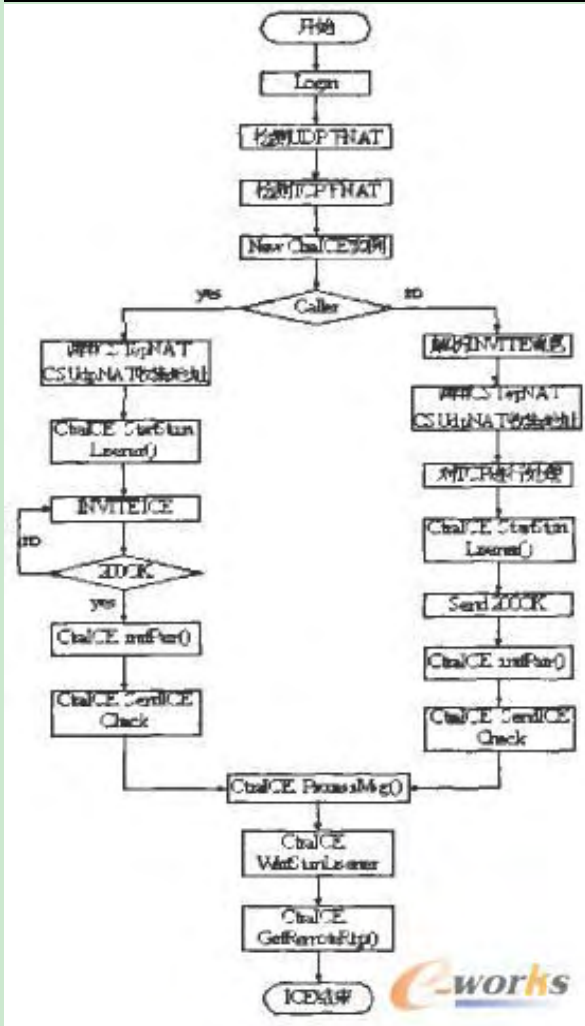


图 1 会话流程

4 测试

以安装了 SIP 软终端的双方都在 Full Cone NAT/防火墙后为例，进行实例测试。测试过程：

(1) 将两台 PC 的 IP 的配置分别为公网 59. 64. 148. 187/22 和私网 10. 0. 0. 5/8

(2) 从公网中的用户代理向私网内的用户代理呼叫，结果能够建立会话，无明显的延时，话音质量好；

(3) 从私网内的用户代理向公网中的用户代理呼叫，结果能够建立会话，且话音质量好；

通过抓包分析可以确定，使用该算法可以成功地穿越 NAT/防火墙设备。

5 结论

ICE 方式的优势是显而易见的，它消除了现有的机制的许多脆弱性。例如，传统的 STUN 有几个脆弱点，其中一个就是发现过程需要客户端自己去判断所在 NAT 类型，这实际上不是一个可取的做法。而应用 ICE 之后，这个发现过程已经不需要了。另一点脆弱性在于 STUN，TURN 等机制都完全依赖于一个附加 的服务器，而 ICE 利用服务器分配单边地址的同时，还允许客户端直接相连，因此即使 STUN 或 TURN 服务器中有任何一个失败了，ICE 方式仍能让呼叫过程继续下去。此外，传统的 STUN 最大的缺陷在于，它不能保证在所有网络拓扑结构中都正常工作，对于 TURN 或类似转发方式工作的协议来说，由于服务器的

负担过重，很容易出现丢包或者延迟情况。而 ICE 方式正好提供了一种负载均衡的解决方案，它将转发服务作为优先级最低的服务，从而在最大程度上保证了服务的可靠性和灵活性。此外，ICE 的优势还在于对 IPv6 的支持。由于广泛的适应能力以及对未来网络的支持，ICE 作为一种综合的解决方案将有着非常广阔的应用前景。

7.7 ubuntu 安装使用 stuntman

1. 官网地址（要翻墙）：STUNTMAN: <http://www.stunprotocol.org/>

从 <http://www.stunprotocol.org/stunserver-1.2.3.tgz> 下载源码

2. 编译依赖：

```
sudo apt-get install g++
sudo apt-get install make
sudo apt-get install libboost-dev # For Boost
sudo apt-get install libssl-dev # For OpenSSL
```

3. 编译 stunserver

```
cd stunserver
sudo make
```

4. 在 stunserver 目录下生成下面三个程序

stunclient, stunserver, stunttestcode

5. run the unit test. Should HAVE NO "FAIL" in the end of any line

```
./stunttestcode
Result of CTestDataStream: PASS
Result of CTestReader: PASS
Result of CTestBuilder: PASS
Result of CTestIntegrity: PASS
Result of CTestMessageHandler: PASS
Result of CTestCmdLineParser: PASS
Testing detection for DirectMapping
Testing detection for EndpointIndependent mapping
Testing detection for AddressDependentMapping
Testing detection for AddressAndPortDependentMapping
Testing detection for EndpointIndependentFiltering
Testing detection for AddressDependentFiltering
Testing detection for AddressAndPortDependentFiltering
Result of CTestClientLogic: PASS
Result of CTestRecvFromEx(IPV4): PASS
Result of CTestRecvFromEx(IPV6): PASS
Result of CTestFastHash: PASS
Result of CTestPolling: PASS
Result of CTestAtomicHelpers: PASS
```

6. start stun server.....

```
./stunserver --help # 使用说明。
nohup ./stunserver --mode full --primaryinterface eth0 --altinterface eth1 &
```

7. stunclient 检测地址端口映射及 NAT 类型

用法：./stunclient --mode full --localport 7777 stun.sipgate.net

NOTE: stuntman 只具有 stun 功能，没有转发功能。支持 UDP，TCP。兼容 RFC3489。

7.8 一个开源的 ICE 库——libnice 介绍

原文地址：<http://blog.csdn.net/kl222/article/details/19336179>

libnice 是一个 ICE 实现库。它实现了 Interactive Connectivity Establishment (ICE) standard (RFC 5245) 和 the Session Traversal Utilities for NAT (STUN) standard (RFC 5389)。

官网地址: <http://nice.freedesktop.org/wiki/>

源码 git 库地址: <http://cgit.collabora.com/git/libnice.git>

1、下载源码:

```
git clone git://git.collabora.co.uk/git/libnice.git
```

2、编译:

2.1、linux 平台下:

它依赖:

```
glib >= 2.10
pkg-config
gupnp-igd >= 0.1.2 (optional)
gstreamer-0.10 >= 0.10.0 (optional)
gtk-doc-tools      #autogen.sh 需要
```

2.2、编译

```
k@k-C410:/home/libnice$ ./autogen.sh
```

```
k@k-C410:/home/libnice$ ./configure
```

```
k@k-C410:/home/libnice$ make
```

3、生成的程序和库

在 nice/.libs 目录下生成静态库 libnice.a、动态库 libnice.so
在 example 目录下生成三个例子程序。

4、例子程序的使用

```
k@k-C410:/home/libnice/examples$ ./simple-example 0 stunserver.org
Copy this line to remote client:
```

Tyyp 33oInvKVEN1Lo6LkVVy6P5 1, 2013266431, 192. 168. 10. 17, 47748, host

```
Enter remote data (single line, no wrapping):
>
```

红色部分表示提供给对等端协商时的验证用户名、密码、外网地址，以空格分隔。

启动二个实例，就可以开始 IM 对话了：

第一个控制台：

```
k@k-C410:/home/libnice/examples$ ./simple-example 0 stunserver.org
Copy this line to remote client:
```

```
Tyyp 33oInvKVENlLo6LkVVy6P5 1,2013266431,192.168.10.17,47748,host
```

```
Enter remote data (single line, no wrapping):
```

```
> h4p1 7M8uL1928RzeRv6cWRDqG8 1,2013266431,192.168.10.17,47758,host
```

```
Negotiation complete: ([192.168.10.17]:47748, [192.168.10.17]:47758)
```

```
Send lines to remote (Ctrl-D to quit):
```

```
> a
>
```

第二个控制台：

```
k@k-C410:/home/libnice/examples$ ./simple-example 0 stunserver.org
Copy this line to remote client:
```

```
h4p1 7M8uL1928RzeRv6cWRDqG8 1,2013266431,192.168.10.17,47758,host
```

```
Enter remote data (single line, no wrapping):
```

```
> Tyyp 33oInvKVENlLo6LkVVy6P5 1,2013266431,192.168.10.17,47748,host
```

```
Negotiation complete: ([192.168.10.17]:47758, [192.168.10.17]:47748)
```

```
Send lines to remote (Ctrl-D to quit):
```

```
> a
```

存在的问题：在 linux 下，stun 服务器地址不能通过域名解析到 IP 地址。解决方法是，直接用 stun 服务器的 IP 地址。本人已向项目提交了补丁包。

7.9 4 种利用 TURN 穿越对称型 NAT 方案的设计与实现

华中科技大学

硕士学位论文

一种利用TURN穿越对称型NAT方案的设计与实现

姓名: 闵江

申请学位级别: 硕士

专业: 通信与信息系统

指导教师: 杜旭

20080528

7.10 基于 ICE 方式 SIP 信令穿透 Symmetric_NAT 技术研究

[基于ICE方式SIP信令穿透Symmetric NAT技术研究【转】](#)

基于ICE方式SIP信令穿透Symmetric NAT技术研究

曾立 吴平 高万林 武文娟

1 (中国农业大学 计算机科学与技术系, 北京 100083) 2 (中国人民大学信息学院, 北京 100872)

摘 要 基于IP的语音、数据、视频等业务在NGN网络中所面临的一个实际困难就是如何有效地穿透各种NAT/FW的问题。对此, 会话初始化协议SIP以往的解决方法有ALGs、STUN、TURN等方式。本文探讨了一种新的媒体会话信令穿透NAT/FW的解决方案—交互式连通建立方式(ICE)。它通过综合利用现有协议, 以一种更有效的方式来组织会话建立过程, 使之在不增加任何延迟同时比STUN等单一协议更具有健壮性、灵活性。本文详细介绍了ICE算法, 并设计一个实例针对SIP信令协议穿透Symmetric NAT流程进行了描述, 最后总结了ICE的优势及应用前景。

关键词 ICE; Symmetric NAT; STUN; TURN; SIP

1 问题背景

多媒体会话信令协议是在准备建立媒体流传输的代理之间交换信息的协议, 例如SIP、RTSP、H.323等。媒体流与信令流截然不同, 它们所采用的网络通道也不一致。由于协议自身设计上的原因, 使得媒体流无法直接穿透网络地址转换/防火墙(NAT/FW)。因为它们生存期的目标只是为了建立一个在信息中携带IP地址的分组流, 这在遇到NAT/FW时会带来许多问题。而且这些协议的目标是通过建立P2P(Peer to Peer)媒体流以减小时延, 而协议本身很多方面却与NAT存在兼容性问题, 这也是穿透 NAT/FW的困难所在。

而NAT仍是解决当前公用IP地址紧缺和网络安全问题的最有力手段, 它主要有四种类型: 完全圆锥型NAT(Full Cone NAT)、地址限制圆锥型NAT(Address Restricted Cone NAT)、端口限制圆锥型NAT(Port Restricted Cone NAT)、对称型NAT(Symmetric NAT)。前三种NAT, 映射与目的地址无关, 只要源地址相同, 映射就相同, 而对称型NAT的映射则同时关联源地址和目的地址, 所以穿透问题最为复杂。

不少方案已经被应用于解决穿透NAT问题, 例如: ALGs(Application Layer Gateway s)、Middlebox Control Protocol、STUN(Simple Traversal of UDP through NAT)、TURN(Traversal Using Relay NAT)、RSIP(Realm Specific IP)、symmetric RTP等。然而, 当这些技术应用于不同的网络拓扑时都有着显著的利弊, 以至于我们只能根据不同的接入方式来应用不同的方案, 所以未能很好地解决All-NAT与Efficiency的问题, 同时还会给系统引入了许多复杂性和脆弱性因素。所以我们目前需要一种综合的足够灵活的方法, 使之能在各种情况下对NAT/FW的信令穿透问题提供最优解。事实上, ICE正是符合这样要求的一种良好的解决方案。

2 ICE技术

2.1 ICE简介

交互式连通建立方式ICE(Interactive Connectivity Establishment)并非一种新的协议, 它不需要对STUN、TURN或RSIP进行扩展就可适用于各种NAT。ICE是通过综合运用上面某几种协议, 使之在最适合的情况下工作, 以弥补单独使用其中任何一种所带来的固有缺陷。对于SIP来说, ICE只需要定义一些SDP(Session Description Protocol)附加属性即可, 对于别的多媒体信令协议也需要制定一些相应的机制来实现。本文仅就SIP问题展开讨论。

<http://www.blogcn.com/user35/leeshun/index.html>

2006-7-14