2、WEBRTC DataChannel SCTP

In WebRTC, DataChannel API is used to send non-media data (sending arbitrary application data). To send the data between the peers, DataChannel uses SCTP (Stream Control Transmission Protocol) which runs over established DTLS tunnel over UDP. Multiple independent SCTP streams (incoming and outgoing) with the same application dependent identifier are carried in a single SCTP association. Each SCTP stream has a unique identifier.

Note: SCTP is a transport protocol (like TCP and UDP) and it can work directly over IP protocol. In WebRTC it works over DTLS tunnel over UDP. SCTP combines the best of TCP and UDP.

So to open a data channel between two peers, mainly we need these libraries :SCTP (for congestion, flow control,…), ICE (for NATing), DTLS (for security), and the datachannel itself.

原文：https://blog.csdn.net/doitsjz/article/details/51935231

--------------------

关于传输：

addtrack 默认创建send和recv

参见PeerConnection::AddTrackUnifiedPlan() transceivers\_

track\_id

sender\_id、receiver\_id都是uuid

## base中的类

factory\_

observer\_

event\_log\_

local\_streams\_

remote\_streams\_

port\_allocator\_

rtp\_data\_channels\_

sctp\_data\_channels\_

sctp\_data\_channels\_to\_free\_\_

**call\_ptr\_**

stats\_

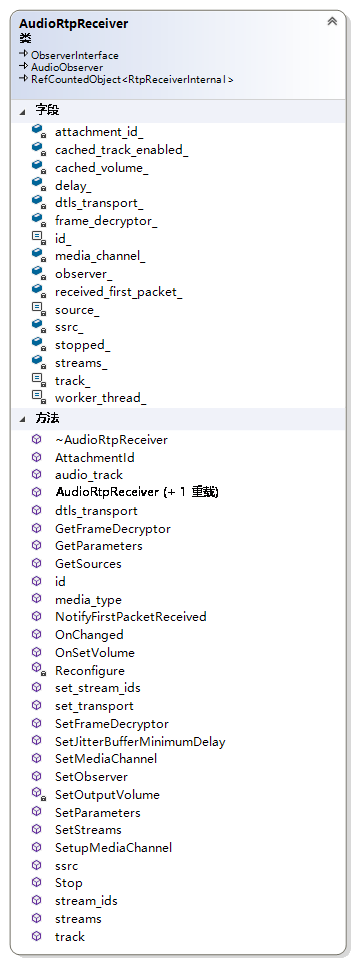
stats\_collector\_

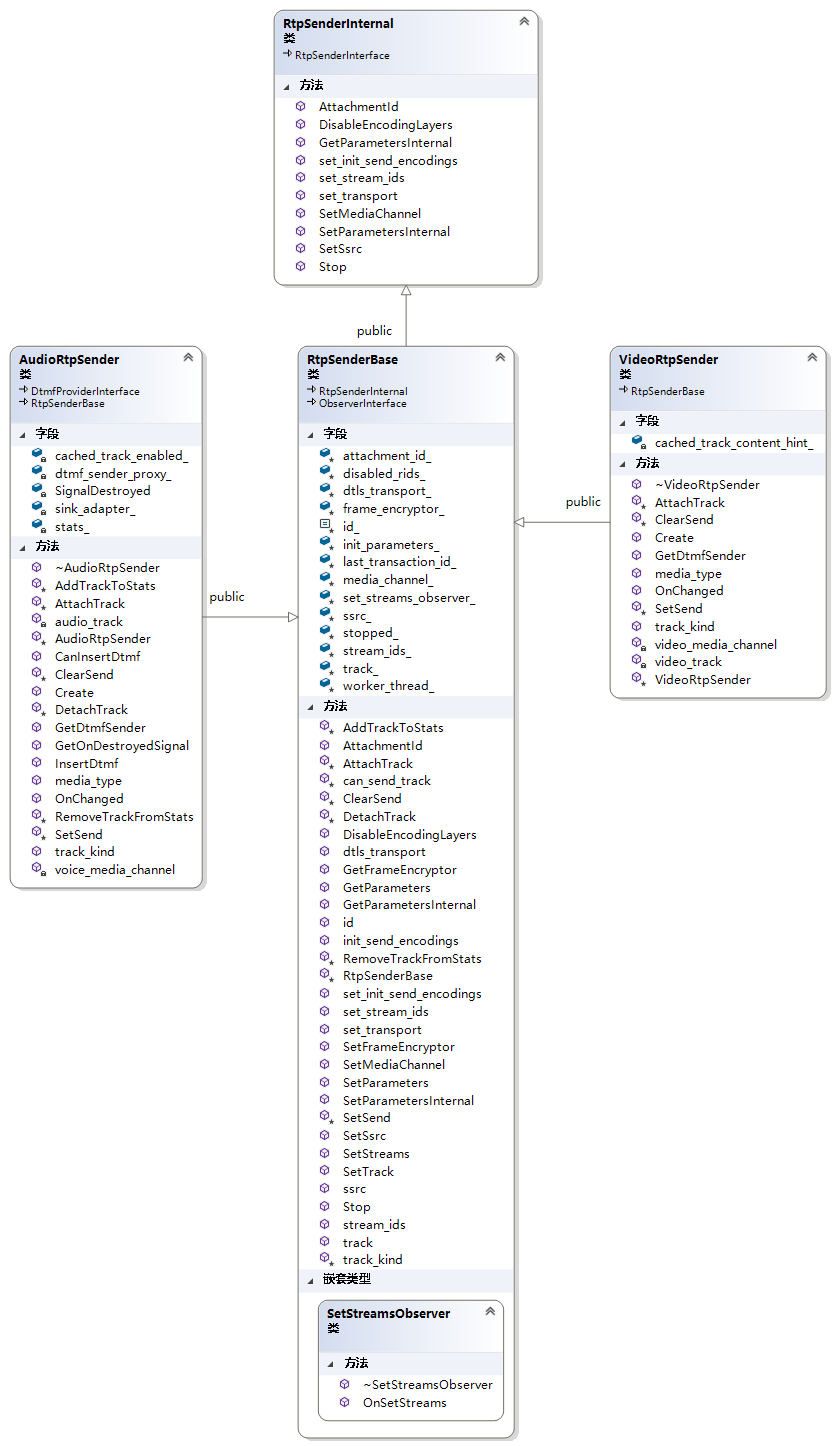
rtp\_data\_channel\_

transport\_controller\_

**media\_transport\_**

**sctp\_transport\_**





Peerconnection\_client与Peerconnection\_server的连接机制

1. 创建异步socket。在Posix平台下通过socketserver创建，在Windows平台下则是直接创建Win32Sokcet，socket的类型是SOCK\_STREAM，也就是tcp连接。





2、初始化socket。利用信号槽机制，将socket的变化信号关联到pc层处理。

3、连接服务器，成功建立连接。

在UI界面单击connet按钮其实是将socket的创建、初始化和连接服务器合并在DoConnect函数。用服务器的ip+port生成一个SocketAddress对象，然后调用步骤1的socket.connet()函数，因为是异步socket，所以connet调用后会立即返回，在返回前将查询的socket连接状态保存在state\_.



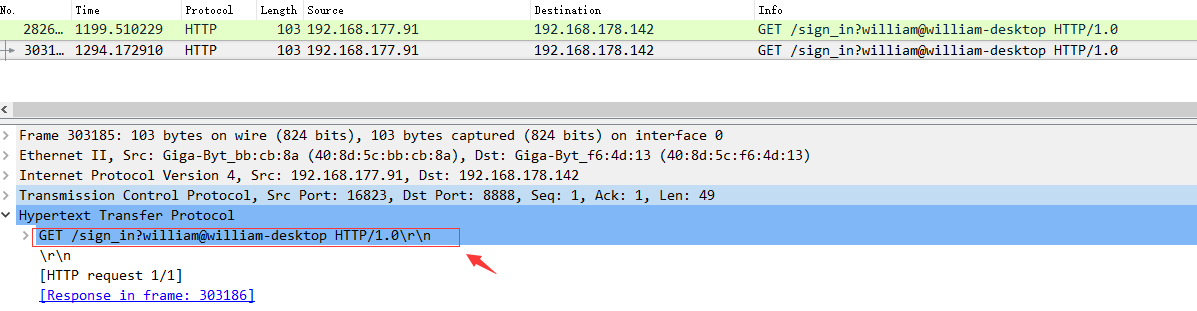
socket的连接状态可以有：



当与服务器建立tcp连接成功之后，Win32Socket::OnSocketNotify() 收到通知FD\_CONNECT（内部机制是怎么实现的），通过之前建立的信号和槽连接，pcc会收到SignalConnetEvent通知，触发PeerConnectionClient::OnConnect。

4、发送Http get登入报文，服务器返回客户端peer\_id和其他peer的列表。

既然连接建立成功，那么下一步就是调用socket的send()发送登入报文。



当服务器返回消息时收到通知FD\_Read，pcc::ReadIntoBuffer调用socket->Recv()读取数据。

服务器返回的http消息：

Win32SocketServer， 一下简称ss

只创建Win32Socket不管理。

成员变量：

message\_queue\_:运行时动态绑定到Win32Thread，这是个工作线程，在dispatch()接口把消息发送给handler。

wnd\_: 类型是MessageWindow，运行时动态绑定到Win32Window。（在其他平台下，猜测也是绑定到某个UI窗口，该UI窗口继承MessageWindow）当wnd\_接收到消息，会调用ss的Pump()接口，之后





Win32Socket能接收系统消息的原因。

成员变量sink\_的类型是EventSink，该类型继承了Win32Window，是注册给系统的主窗口类，所以会接收处理WM\_SOCKETNOTIFY消息，在收到消息后推送给parent处理，就是Win32Socket的OnSocketNotify()了。

Wndproc是Windows操作系统向应用程序发送一系列消息之一，每个窗口会有一个窗口过程的回调函数，分别是窗口句柄、消息ID、WPARAM、LPARAM。

# 编码基础

## C++部分

1、把构造函数声明为protected的目的？

1. enum class与enum
2. using指示符
3. static声明的函数，在文件范围

### STL库

1、std::map

find()

find\_if()

#### 智能指针

初始化

直接初始化

Make\_unique<>()

重置

Std::unique\_ptr<T>

Std::unique\_ptr<T> ::Reset()

#### String

构造新string对象

string substr (size\_t pos = 0, size\_t len = npos) const;

**Generate substring**

Returns a newly constructed [string](http://www.cplusplus.com/string) object with its value initialized to a copy of a substring of this object.

字符串比较

|  |  |
| --- | --- |
| *string (1)* | int compare (const string& str) const noexcept; |
| *substrings (2)* | int compare (size\_t pos, size\_t len, const string& str) const;  int compare (size\_t pos, size\_t len, const string& str,  size\_t subpos, size\_t sublen) const; |
| *c-string (3)* | int compare (const char\* s) const;  int compare (size\_t pos, size\_t len, const char\* s) const; |
| *buffer (4)* | int compare (size\_t pos, size\_t len, const char\* s, size\_t n) const; |

**Compare strings**

Compares the value of the [string](http://www.cplusplus.com/string) object (or a substring) to the sequence of characters specified by its arguments.

### 回车换行

回车”（Carriage Return）和“换行”（Line Feed）这两个概念的来历和区别。

符号        ASCII码        意义

\n               10          换行

\r                13            回车CR

在Windows中：

'\r' 回车，回到当前行的行首，而不会换到下一行，如果接着输出的话，本行以前的内容会被逐一覆盖；

'\n' 换行，换到当前位置的下一行，而不会回到行首；

Unix系统里，每行结尾只有“<换行>”

Linux中遇到换行符("\n")会进行回车+换行的操作，回车符反而只会作为控制字符("^M")显示，不发生回车的操作。而windows中要回车符+换行符("\r\n")才会回车+换行，缺少一个控制符或者顺序不对都不能正确的另起一行。

## 多线程

1. 关于线程
   1. PlatformThreadId 与PlatformThreadRef:

在Windows平台下这两个都指的是线程id，在Posix平台下分别返回的是pthread\_t和pid\_t。

DWORD WINAPI GetCurrentThreadId (void);

说明 获取当前线程一个唯一的线程标识符

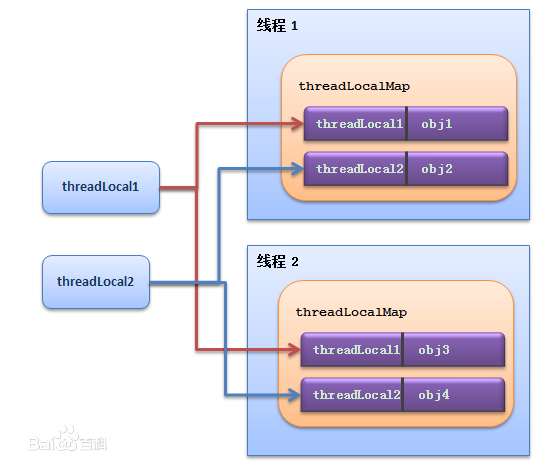
### TLS（线程局部存储Thread Local Storage）

**在一个线程修改的内存内容，对所有线程都生效**。这是一个优点也是一个缺点。说它是优点，线程的数据交换变得非常快捷。说它是缺点，一个线程死掉了，其它线程也性命不保; **多个线程访问共享数据，需要昂贵的同步开销**，也容易造成同步相关的BUG。

如果需要在一个线程内部的各个函数调用都能访问、但其它线程不能访问的变量（被称为static memory local to a thread 线程局部静态变量），就需要新的机制来实现。这就是TLS。

**线程局部存储在不同的平台有不同的实现，可移植性不太好**。幸好要实现线程局部存储并不难，最简单的办法就是建立一个全局表，通过当前线程ID去查询相应的数据，因为各个线程的ID不同，查到的数据自然也不同了。但Windows系统采用了每个线程建线程专享的索引表，表的条目为线程局部存储的地址。在线程执行的任何代码处，都可以查询本线程的这个索引表获得要访问的线程局部存储的地址。

**大多数平台都提供了线程局部存储的方法，无需要我们自己去实现**



#### Linux 实现

int pthread\_key\_create(pthread\_key\_t \*key, void (\*destructor)([void](https://baike.baidu.com/item/void/5126319)\*));

int [pthread\_key\_delete](https://baike.baidu.com/item/pthread_key_delete)(pthread\_key\_t key);

void \*pthread\_getspecific(pthread\_key\_t key);

int pthread\_setspecific(pthread\_key\_t key, const void \*[value](https://baike.baidu.com/item/value/2285610));

#### Windows实现

第一步，在主进程内调用TlsAlloc()函数，从将要启动的**每个**线程的TLS索引数组中预定一个条目(slot），并返回该条目的序号：

DWORD global\_dwTLS\_fvalue = [TLSAlloc()](https://baike.baidu.com/item/TLSAlloc%28%29);

注意，此步之后，[变量](https://baike.baidu.com/item/%E5%8F%98%E9%87%8F)( global\_dwTLS\_fvalue )保存的是分配得到的[TLS](https://baike.baidu.com/item/TLS/2979545)索引数组的某个条目的序号，例如值为3。编程者在写这个程序代码时规定了这个变量( global\_dwTLS\_fvalue )保存了线程局部存储fvalue在每个[线程](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/103101)的TLS索引[数组](https://baike.baidu.com/item/%E6%95%B0%E7%BB%84)的对应条目的序号。变量( global\_dwTLS\_fvalue )是普通的[全局变量](https://baike.baidu.com/item/%E5%85%A8%E5%B1%80%E5%8F%98%E9%87%8F)，各个线程随后只需要读取它的值。类似的，另外一个线程局部存储buf变量也需要定义一个变量( global\_dwTLS\_buf )并用TLSAlloc()初始化。

第二步，在每个进程执行的一开头，从堆中动态分配一块内存区域（使用[LocalAlloc](https://baike.baidu.com/item/LocalAlloc/4473086)()[函数调用](https://baike.baidu.com/item/%E5%87%BD%E6%95%B0%E8%B0%83%E7%94%A8)）

[void](https://baike.baidu.com/item/void/5126319)\* p\_fvalue = LocalAlloc(LPTR，sizeof([float](https://baike.baidu.com/item/float/1680868)));

然后使用TlsSetValue()函数调用，把这块内存区域的地址存入TLS索引[数组](https://baike.baidu.com/item/%E6%95%B0%E7%BB%84)相应的条目中：

TlsSetValue( global\_dwTLS\_fvalue, p\_fvalue);

第三步，在每个[线程](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/103101)的任意执行位置，都可以通过该线程私有的TLS索引数组的相应条目，使用TlsGetValue()函数得到上一步的那块内存区域的地址，然后就可以对该内存区域做读写操作了。这就实现了在一个线程内部处处可访问的线程局部存储。

LPVOID lpvData = TlsGetValue(global\_dwTLS\_fvalue);

\*lpvData = (float) 3.1416; //应用该线程局部存储

最后，如果不再需要上述[线程](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/103101)局部[静态变量](https://baike.baidu.com/item/%E9%9D%99%E6%80%81%E5%8F%98%E9%87%8F)，要动态释放掉这块内存区域(使用[LocalFree](https://baike.baidu.com/item/LocalFree/1455147)()函数)，这一般在线程即将结束时清理线程占用的各项资源时释放。然后，主进程从TLS索引[数组](https://baike.baidu.com/item/%E6%95%B0%E7%BB%84)中放弃对应的条目的占用(使用TlsFree()函数）。

LocalFree((HLOCAL) p\_fvalue );

TlsFree(global\_dwTLS\_fvalue);

### Webrtc线程模型

1. Thread

创建一个线程

1. ThreadManager

线程管理单例类

1. MessageQueueManager

消息队列管理单例类

问题：MessageQueueManager管理MessageQueue，也就是管理Thread，那还要ThreadManager何用？



1. MessageQueue

// Currently, MessageQueue holds a socket server, and is the base class for

// Thread. It seems like it makes more sense for Thread to hold the socket

// server, and provide it to the MessageQueue, since the Thread controls

// the I/O model, and MQ is agnostic to those details. Anyway, this causes

// messagequeue\_unittest to depend on network libraries... yuck.

翻译过来就是：目前MessageQueue拿着一个SocketServer，MQ是Thread的基类。不过让Thread拿SocketServer貌似更合理，因为Thread控制IO，MQ对这些细节应是无感知的。现在这种设计让MQ的单元测试要依赖网络库。。。吐槽

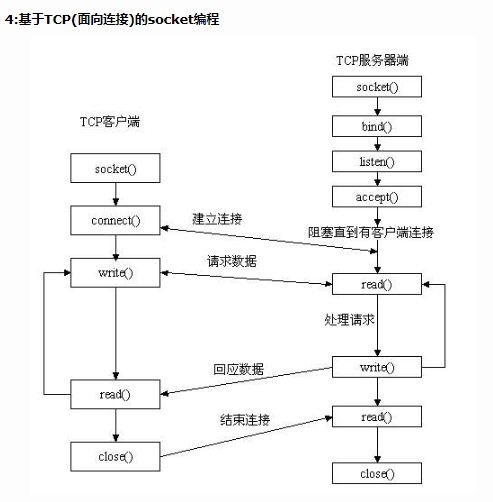


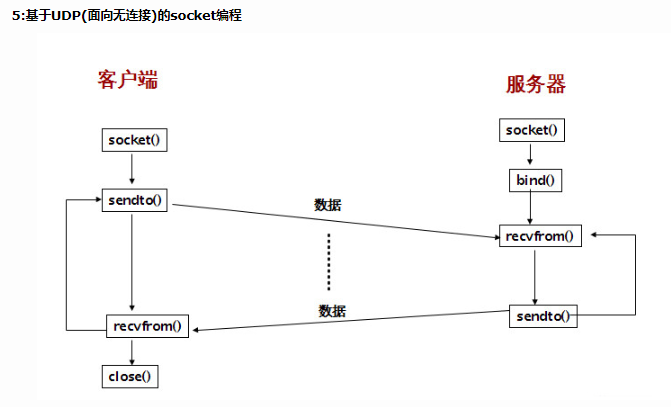
## 高级I/O

1. winsock编程WSAEventSelect模型

WSAEventSelect模型和WSAAsyncSelec模型类似，都是用调用WSAXXXXXSelec函数将socket和事件关联并注册到系统，并将socket设置成非阻塞模式。二者不同之处在于socket事件的通知方法：WSAAsyncSelec模型利用窗口句柄和消息映射函数通知网络事件，而WSAEventSelect模型利用WSAEVENT通知网络事件。

补充windows下TCP和UDP下使用socket的API和流程图：





## Git用法

1、http协议下的凭证存储问题。

使用http方式访问git仓库时需要输入用户名和密码，默认配置是在第一次输入后记住凭证，在windows下就是在凭证管理器中保存，mac下就是钥匙串中保存。

查看命令：git config --list | grep credential

https://www.cnblogs.com/volnet/p/git-credentials.html

## Linux常用命令

## 正则表达式

visual studio查找用到的正则表达式：

RTC\_RUN\_ON\(\w\*\(\)\)

## 网络

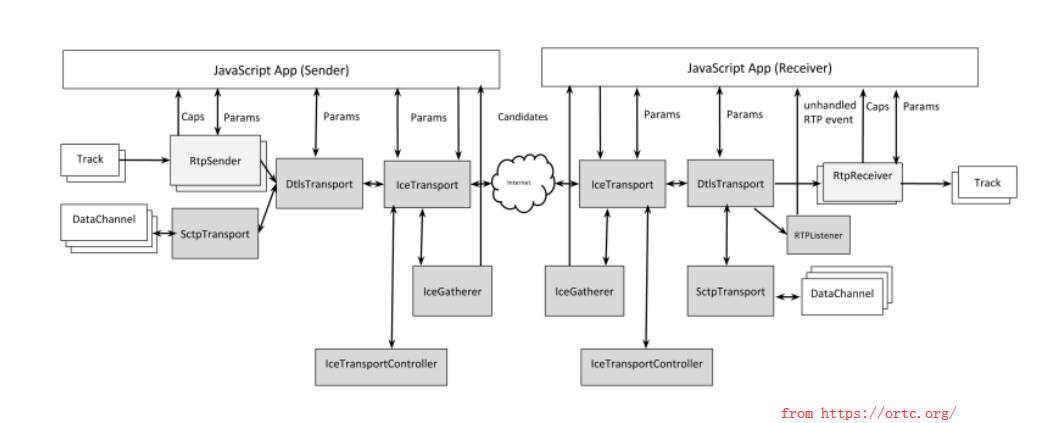
### 多播地址

在IPv4中，它历史上被叫做D类地址，一种类型的IP地址，它的范围从224.0.0.0到239.255.255.255，或，等同的，在224.0.0.0/4。在IPv6，多播地址都有前缀ff00::/8。但是注意，224.0.0.0被保留，不能赋给任何多播组。

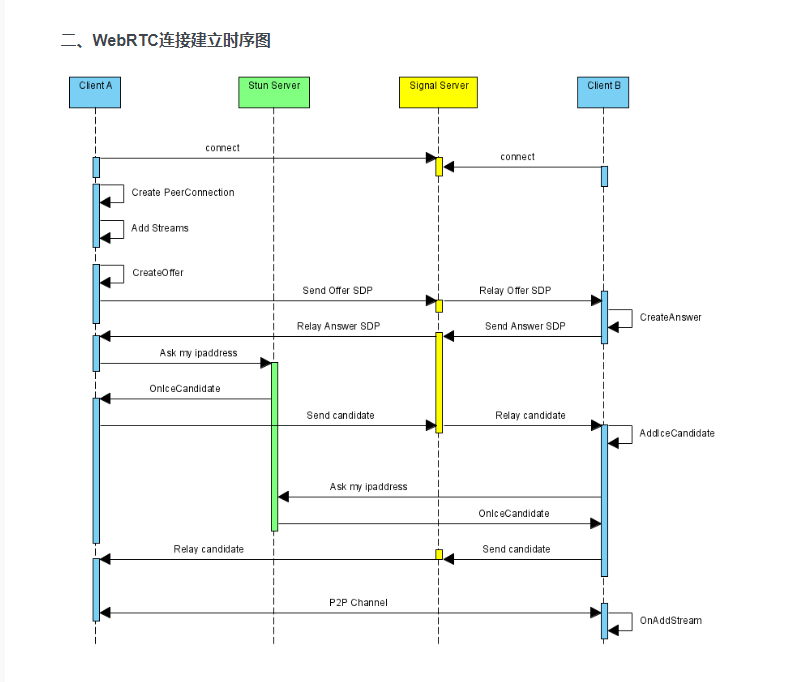
地址格式如下所示：1110××××.××××××××.××××××××.××××××××。第一组八位数的头四位用1110表示， 这代表该地址是一个多播地址， 其余28位比特标识具体的多播 地 址 。 因 此 ， 多 播 地 址 的 范 围 从 224.0.0.0到239.255.255.255。 其中， 后28比特的地址字段中不再有结构层次， 也不标出组的来源， 不会象 A、B、C类地址那样包含一个网络地址。多播地址的使用分配现在由互联 网 地 址 指 派 机 构 （IANA） 控 制[1]  。

# 协议

## JSEP

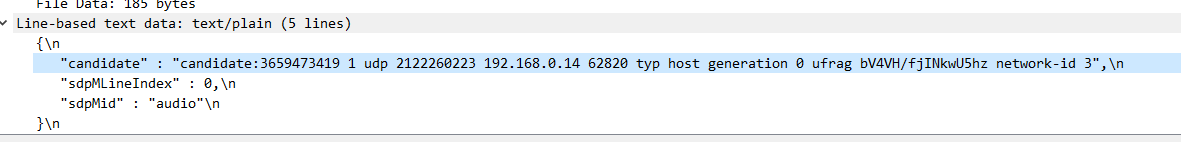


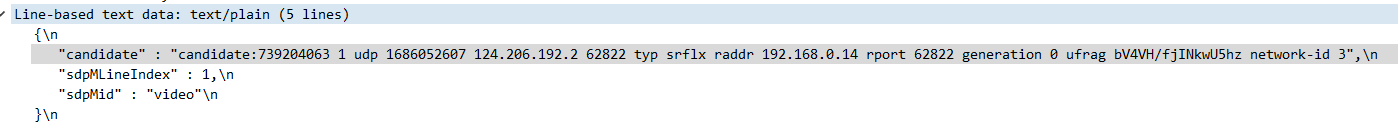
pc时序图来自<https://my.oschina.net/u/1863428/blog/868201>



### Candidate描述

相关文档： candidate-attribute syntax. http://tools.ietf.org/html/rfc5245#section-15.1

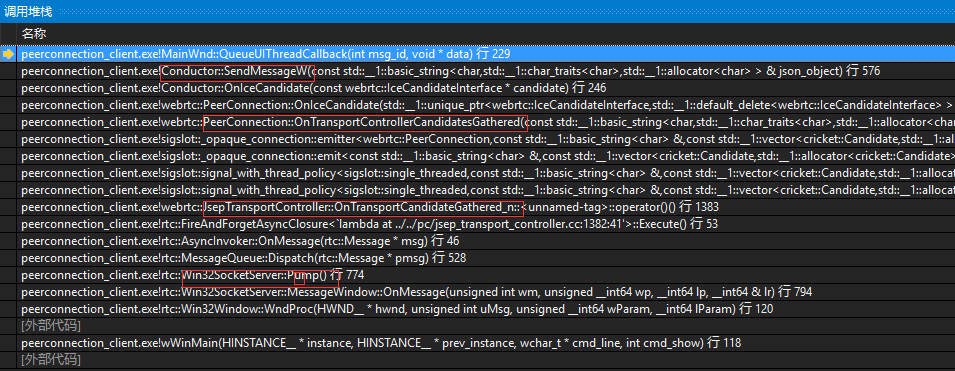






#### 发送candidate

现在只知道是从消息队列中获取了一条message，这个message包含已经获取的candidate信息，然后JsepTransportController -> PC -> Conductor，最后用PostThreadMessage放入线程队列（这个线程就是主线程，UI线程，在wMain中创建的Win32Thread线程，关联了socket）。



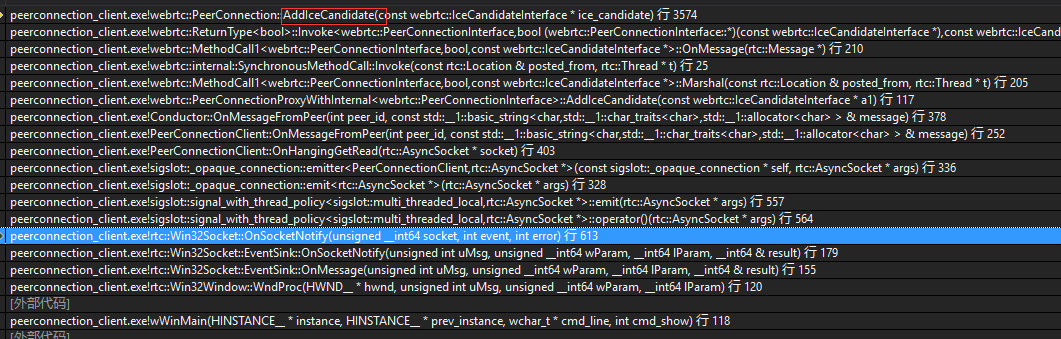
PostThreadMessage是一个Windows API函数。其功能是将一个队列消息放入（寄送）到指定[线程](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/103101)的[消息队列](https://baike.baidu.com/item/%E6%B6%88%E6%81%AF%E9%98%9F%E5%88%97/4751675)里，不等待线程处理消息就返回。

#### 接收candidate

hanging\_get\_类型是Win32Socket，hanging\_get\_是与服务器交互另一个peer建立连接的情况。与control\_socket\_一样，这也是一个异步socket，底层socket通知采用WSAAsyncSelec模型，利用窗口句柄和消息映射函数通知网络事件。



WinDemo下的调用堆栈：



### ICE

### SDP协议

sdp里面除了媒体描述的信息之外新增了几个选项：

* a=rtcp-mux  
  表示rtp和rtcp使用同一个端口进行发送和接收
* a=mid:video  
  标识这一路媒体的id名称，用于a=group
* a=group:BUNDLE video  
  现在只有一路视频流，当音视频都有的时候 a=group:BUNDLE video audio 标识音频和视频流复用同一个端口进行发送和接收，通过ssrc进行区分不同的流。所以当既有rtcp-mux，又有a=group把多路流打包到一起的时候，只要创建一个主机类型候选项即可。
* a=ice-ufrag 和 a=ice-pwd  
  用于ice进行stun协商时进行对端认证。

Purpose: This attribute is used with Interactive Connectivity Establishment (ICE), and provides the password used to protect STUN connectivity checks.

Purpose: This attribute is used with Interactive Connectivity Establishment (ICE), and provides the fragments used to construct the username in STUN connectivity checks.

The "ice-pwd" and "ice-ufrag" attributes can appear at either the session-level or media-level. When present in both, the value in the media-level takes precedence. Thus, the value at the session level is effectively a default that applies to all media streams, unless overriden by a media-level value. Whether present at the session or media level, there MUST be an ice-pwd and ice-ufrag attribute for each media stream. If two media streams have identical ice-ufrag's, they MUST have identical ice-pwd's.

* a=ice-options:trickle  
  通知对端支持trickle，即sdp里面描述媒体信息和ice候选项的信息可以分开传输。
* a=fingerprint  
  当双方进行dtls协商交互srtp的加密key时，对对端进行验证。
* a=candidate  
  ice的候选项 通知对端本地的ice连接的候选项，本文只使用了本地地址的主机候选项，没有使用stun服务器。

## DTLS协议

### 1.DTLS介绍

#### 1.1 DTLS的作用

互联网先驱们最开始在设计互联网协议时主要考虑的是可用性，安全性是没有考虑在其中的，所以传输层的TCP、UDP协议本身都不具备安全性。SSL/TLS协议是基于TCP socket，在传输层和应用层之间构建了一个端到端的安全通道，保证了传输数据的加密性。

但是SSL/TLS协议并不能用于UDP协议，而UDP也有安全传输的需求，于是产生了DTLS协议（Datagram TLS）。

即DTLS的作用为给UDP提供端到端的安全通道，就像SSL/TLS对TCP的作用一样。并且DTLS尽可能参考了SSL/TLS协议的安全机制，在具体实现上复用了70%的TLS代码。

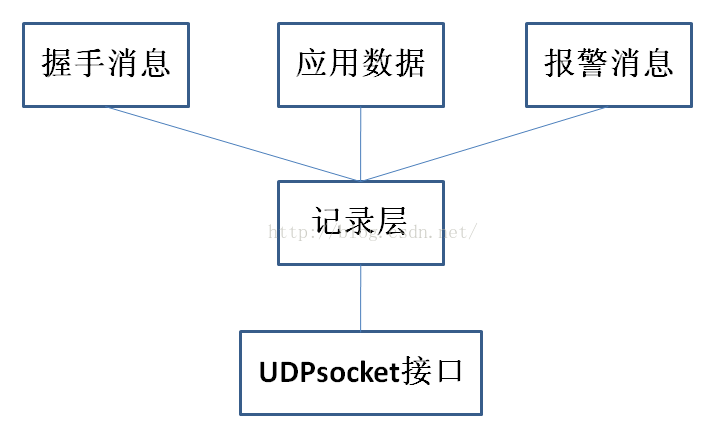
#### 1.2 DTLS的特点

UDP协议是不面向连接的不可靠协议，且没有对传输的报文段进行加密，不能保证通信双方的身份认证、消息传输过程中的按序接收、不丢失和加密传送。而DTLS协议在UDP提供的socket之上实现了客户机与服务器双方的握手连接，并且在握手过程中通过使用PSK或ECC实现了加密，并且利用cookie验证机制和证书实现了通信双方的身份认证，并且用在报文段头部加上序号，缓存乱序到达的报文段和重传机制实现了可靠传送。

在握手完成后，通信双方就可以实现应用数据的安全加密和可靠传输。

#### 1.3 DTLS协议层次

DLTS协议分为两层，下层为记录层（记录层），record包的内容分为头部和载荷两部分。记录包的载荷即为上层的内容。DTLS上层的包的类型分为三种，分别是握手消息，警告消息，应用数据；如图一所示。



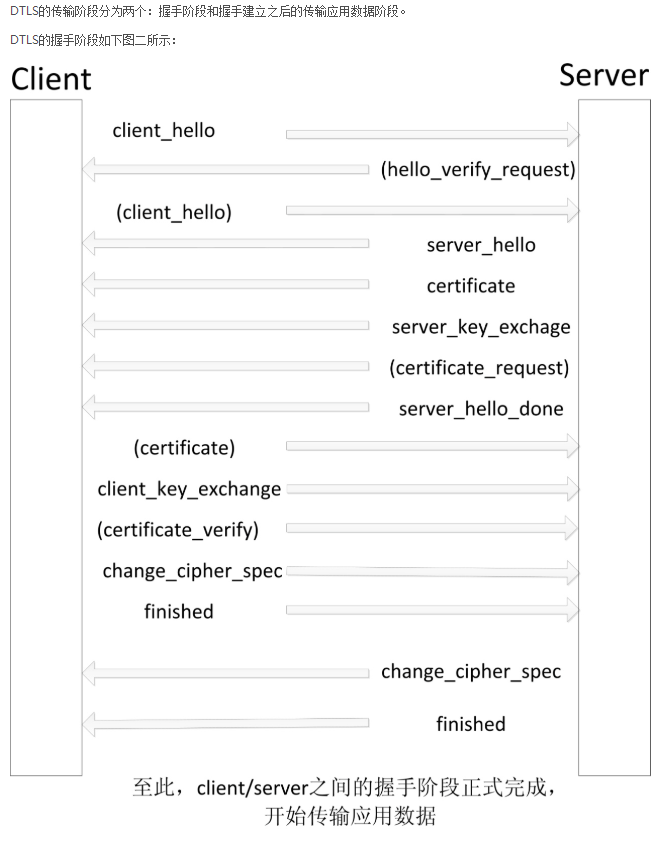
图一.DTLS协议的层次

在整个DTLS协议的通信过程中，通信双方构造报文段的过程都是先产生上层的载荷消息（如握手消息，应用数据，警告消息），然后添加头部，构成完整的上层消息。接着再以此作为记录层的载荷，最后添加记录层的头部，构成完整的记录报文段，最后调用UDP的socket接口，发送给另一方。

加密过程是只对记录层的载荷(即上层消息，此协议中被加密的消息是finished消息和应用数据两种)进行加密，所以接收方在收到记录消息后，首先要做的也是判断记录消息是否被发送方加密，若是，则应先解密才能读取出明文数据以进行后面的处理。

### 2.DTLS传输阶段

#### 2.1 整个握手阶段的交互过程

图二.DTLS协议客户机与服务器握手阶段的交互过程

客户机向服务器发起连接，服务器可以根据配置选择是否验证客户机的cookie和证书（即是否向客户机发送client\_hello\_verify和certificate\_request报文段）。

#### 2.2 DTLS的cookie验证机制

由于DTLS是基于UDP的，所以可能会遭受两种形式的拒绝服务攻击。一种是类似于对TCP的资源消耗攻击，另一种是放大攻击，即恶意攻击者仿造被攻击者的IP地址发通信初始化报文段给服务器，而服务器会返回一个体积大很多的证书给被攻击者，超大量证书有可能造成被攻击者的瘫痪。

cookie机制要求客户机重复发送服务器之前发送的cookie值来验证通信方的源IP地址确实可以通信，由此可以减少拒绝服务攻击的危害。

cookie验证身份的具体机制为：

协议规定客户机发送的第一个报文段client\_hello中含有cookie的值这一项（有可能为空）。服务器检验收到的该报文段中的cookie值，如果cookie为空，则说明之前没建立过连接，服务器根据客户机的源IP地址通过哈希方法随机生成一个cookie，并填入client\_hello\_verify中发送给客户机。

客户机再在第二次发送的client\_hello报文段中填入服务器之前发过来的cookie，服务器第二次收到该报文段之后便检验报文段里面的cookie值和服务器之前发给该主机的cookie值是否完全相同，若是，则通过cookie验证，继续进行握手连接；若不是，则拒绝建立连接。

#### 2.3 client\_hello报文段和server\_hello报文段的内容

client\_hello报文段的内容除cookie外，还有客户机产生的32字节的随机数，其中前4字节为时间戳，后28字节为系统产生的随机数。此外，该报文段的内容还有客户机支持的加密方式（PSK或者ECC）和压缩方式，供服务器进行选择。

在通过cookie校验后，服务器发送server\_hello报文段给客户机。该报文段包含有服务器产生的32字节的随机数，和服务器选中的用来进行之后的会话的加密方式和压缩方式。

#### 2.4 certificate报文段的内容

在服务器发给客户机的证书报文段中，包含有服务器证书的公钥；客户机接收到该报文段后，按照协议规定，从报文段的对应位置中读取出服务器证书的公钥存入相关变量中。

#### 2.5 基于ECC加密方式的ECDH秘钥交换协议和ECDSA数字签名算法

若协议所选加密方式为ECC（椭圆曲线加密），则在server\_key\_exchange报文段的构造过程中会使用ECDH（椭圆曲线秘钥交换协议）和ECDSA（椭圆曲线数字签名算法）。ECDH和ECDSA分别是ECC和DH（diffie-hellman）秘钥交换协议、DSA（数字签名算法）的结合。

在server\_key\_exchange报文段中，包含有所选用的椭圆曲线E，阶N和基点G的x,y坐标，客户机在收到这个报文段后，进行对应的格式检验，并读取数据，因此服务器和客户机共同获得约定好的用来进行ECDH秘钥协商交换协议的参数，从而可以共同协商出相同的对话秘钥用于加密之后的会话内容。

同时，为了防范中间人攻击，服务器还在server\_key\_exchange报文段的末尾对整个报文段进行了ECDSA数字签名。具体签名过程为先用client\_hello报文段和server\_hello报文段中的2个32字节的随机数作为函数参数，利用sha256哈希算法对server\_key\_exchange报文段本身的载荷产生摘要，然后再用服务器的私钥和sha256哈希算法进行ECDSA数字签名，得到签名结果r和s,并写入server\_key\_exchange报文段的末尾。

客户机在收到server\_key\_exchange报文段后，先进行各数值项格式的校验，然后提取出报文段末尾的签名值r和s。之后，用已经读取出的服务器的公钥的x,y坐标值来对server\_key\_exchange报文段进行ECDSA签名验证，若结果和报文段中的r和s值一致，则报文段通过验证。

#### 2.6 基于PSK加密方式的身份认证过程和会话秘钥产生过程

整个DTLS协议的加密方式可选用ECC或PSK（预共享秘钥，PreSharedKey）两种。若为ECC，则通过ECDH协议来进行通信双方的秘钥协商；若为PSK，则直接以通信双方事先就已经约定好了的秘钥为基础来进行加密通信。

对于PSK加密通信来说，验证对方的通信身份非常关键。所以通信双方会在本地存取对方的psk\_id（即身份标志）和psk\_id\_length（身份标志长度），通过比较收到的报文段中的psk\_id,psk\_id\_length和本地存储的是否完全一致来进行对方身份的验证。

在整个通信过程中，采用PSK与ECC的区别主要体现在server\_key\_exchange报文段、client\_key\_exchange报文段的内容不同和双方计算得到预主秘钥方式的不同。

当采用PSK加密时，server\_key\_exchange报文段和client\_key\_exchange报文段的内容分别是服务器与客户机各自的psk\_id和psk\_id\_length，由此双方可以互相知道对方的psk\_id和psk\_id\_length。

之后，双方都会对收到的报文段进行检验，只有psk\_id和psk\_id\_length与本地存储的完全一致才会进行后面的通信。

当双方都通过身份验证后，双方再各自用相同的函数产生预主秘钥，而函数的参数包括之前通信阶段中双方各自产生的32字节的随机数，由此可以保证虽然本地存储的psk秘钥不变，但每次临时通信时的会话秘钥还是会一直变化的，从而增强了抗攻击性。

双方产生预主秘钥后，再调用和使用ECC加密的相同方式来产生主秘钥，即用于之后会话通信的对称秘钥，该过程中依然会用到双方产生的32字节的随机数。

由此，通信双方使用PSK加密方式来实现了身份认证和会话秘钥的产生。

#### 2.7 server\_hello\_done报文段和client\_key\_exchange报文段的内容

服务器发送的server\_hello\_done报文段的载荷部分为空，只是发给客户机来作为标志，表示服务器当前阶段的报文段已经发送完毕。

客户机在收到server\_hello\_done报文段后，发送client\_key\_exchange报文段给服务器，里面包含了用于秘钥协商的基点的x,y坐标,并且不同于server\_key\_exchange报文段，客户机并没有在报文段的末尾进行ECDSA数字签名。

#### 2.8 客户机产生会话秘钥

之后，客户机再通过ecdh\_pre\_master\_secret函数来产生用于之后会话的预主秘钥。其中函数的参数包括客户机自己的私钥，和服务器共享的用于ECDH秘钥协商算法的基点的x,y坐标。

产生预主秘钥后，再根据之前阶段客户机和服务器分别产生的32字节的随机数产生主秘钥master\_secret，此时主秘钥为对称秘钥，用于之后会话的加解密。

#### 2.9 change\_cipher\_spec报文段和finished报文段的内容

客户机计算出会话秘钥后，发送change\_cipher\_spec报文段给服务器，这个报文段的有效载荷为空，用来作为标志通知服务器，表示客户机已经算出主秘钥，之后发送的报文段会采用主秘钥加密。

握手阶段中客户机发送的最后一个报文段为finished报文段，载荷内容为MAC值（消息验证码），用于给服务器做认证。并且值得注意的是，finished报文段作为记录层的载荷部分在发送时已经用上一步产生的会话秘钥进行加密编码。

#### 2.10 服务器产生会话秘钥

服务器在收到客户机发送过来的finished报文段后，也会和客户机用ECDH秘钥协商算法经过相同的流程，调用相同的函数先产生预主秘钥，再产生主秘钥。

#### 2.11 握手阶段的结束

最后，服务器产生经会话秘钥加密后的finished报文段给客户机，标志整个握手阶段的结束。

客户机收到服务器发过来的finished报文段后，便可发送应用数据。并且应用数据会一直用会话秘钥加密，从而实现了UDP所不具备的安全性。

## SRTP协议

### 一 概述

随着网络技术的发展和标准的制定,实时音频、视频的应用越来越广泛,这些应用反过来又促进了相关协议标准的发展。1996年IETF在RFC1889中定义了传输实时数据的Internet标准协议RTP,并在2003年制定了升级版本RFC3550。

**由于网络安全问题日益突出,2004年3月IETF在RFC3711中定义了RTP的一个扩展协议SRTP来提高RTP应用的安全性,SRTP定义了对RTP与RTCP流进行加密、认证以及抗重播攻击检测等的实现框架。**

### 二 简介SRTP & SRTCP

     安全实时传输协议（Secure Real-time Transport Protocol或SRTP）是在实时传输协议（Real-time Transport Protocol或RTP）基础上所定义的一个协议，旨在为单播和多播应用程序中的实时传输协议的数据提供加密、消息认证、完整性保证和重放保护。它是由 David Oran（思科）和Rolf Blom（爱立信）开发的，并最早由IETF于2004年3月作为RFC 3711发布。

     由于实时传输协议和可以被用来控制实时传输协议的会话的实时传输控制协议（RTP Control Protocol或RTCP）有着紧密的联系，安全实时传输协议同样也有一个伴生协议，它被称为安全实时传输控制协议（Secure RTCP或SRTCP）；安全实时传输控制协议为实时传输控制协议提供类似的与安全有关的特性，就像安全实时传输协议为实时传输协议提供的那些一样。

     在使用实时传输协议或实时传输控制协议时，使不使用安全实时传输协议或安全实时传输控制协议是可选的；但即使使用了安全实时传输协议或安全实时传输控制协议，所有它们提供的特性（如加密和认证）也都是可选的，这些特性可以被独立地使用或禁用。唯一的例外是在使用安全实时传输控制协议时，必须要用到其消息认证特性。

为了提供对数据流的保密，需要对数据流进行加密和解密。关于这一点，安全实时传输协议(结合安全实时传输控制协议)只为一种加密算法，即AES制定了使用标准。这种加密算法有两种加密模式，它们能将原始的AES块密文转换成流密文：分段整型计数器模式和f8模式。

除了AES加密算法，安全实时传输协议还允许彻底禁用加密，此时使用的是所谓的“零加密算法”。它可以被认为是安全实时传输协议支持的第二种加密算法，或者说是它所支持的第三种加密模式。事实上，零加密算法并不进行任何加密，也就是说，加密算法把密钥流想像成只包含“0”的流，并原封不动地将输入流复制到输出流。这种模式是所有与安全实时传输协议兼容的系统都必须实现的，因为它可以被用在不需要安全实时传输协议提供保密性保证而只要求它提供其它特性（如认证和消息完整性）的场合。

以上列举的加密算法本身并不能保护消息的完整性，攻击者仍然可以伪造数据——至少可以重放过去传输过的数据。因此，安全实时传输协议标准同时还提供了保护数据完整性以及防止重放的方法。

为了进行消息认证并保护消息的完整性，安全实时传输协议使用了HMAC-SHA1算法。这种算法使用的是默认160位长度的HMAC-SHA1认证密钥。但是它不能抵御重放攻击；重放保护方法建议接收方维护好先前接收到的消息的索引，将它们与每个新接收到的消息进行比对，并只接收那些过去没有被播放过的新消息。这种方法十分依赖于完整性保护的使用(以杜绝针对消息索引的欺骗技术)。

### 三 协议的数据报文结构

数据报文包括包头、数据主体以及认证信息三个部分。

（1）32bits：

V 2bits    版本号，现在已经是版本2

P 1bit 填充位，当负载的长度不够32bits的整数倍时，需要填充位

X 1bit 扩展位，若为1，则固定的包头后增加一个32bits的扩展（rtp extension)

CC 4bits CSRC的数目

M 1bit 允许在比特流中标记重要的事件

PT 7bits 负载类型

序列号16bits   每发送一个RTP数据包，序列号加1，根据此来判断序列号的顺序

（2）时间戳 32bits

（3）SSRC标识符 32bits synchronizating source identifier 识别同步源

（4）CSRC标识符 n个32bits contributing source identifiers 识别负载重的有效贡献源

（5）可选存在的RTP extension

（6）加密的payload（末尾可能包含RTP padding和RTP pad count）

（7）包尾是SRTP MKI（可选），master key identifier，是用来生成session加密密钥的随机位串标识符。

（8）认证标签（Authentication tag）

**与RTP包的主要区别是负载加密、SRTP MKI（主密钥标识符，由密钥管理协议决定）、认证标签。**

### 四、协议的工作流程

（1）协议中涉及哪些密钥和重要参数

**主密钥 master key**

主密钥是一个长度不定的随机位串，用来生成会话密钥

**会话密钥 session key**

会话密钥就是用于消息加密和认证的密钥

**master salt 不知道怎么翻译**

salt是指一种在生成会话密码时的输入参数，salt增加了字典式攻击的难度，每增加1bit的salt，就会使字典攻击的难度和时间翻倍。

**key\_derivation\_rate**

生成会话密钥的速率，必须是{1，2，4，……2^24}中的一个，必须为2的幂数

**<from,to>**

这是两个48bits的时间值，表示master key的有效时间。

**ROC（rollover counter)**

记录序列号的重置次数，用来计算SRTP包的索引

Index = 2^16 \* ROC + SEQ

（2）会话密钥的产生

 在协议中，密钥生成过程的描述是通过数学函数来说明的，如下：

key\_session = PRF (key\_master, x);

r = index/key\_derivation\_rate;;//label长度8位

key\_id = <label> || r;

x = key\_id XOR master salt;

PRF是一个AES-cm的对称加密函数。

用框图表示，如下：

其中，IV=[ <label>||(index/key\_derivation\_rate)XOR master\_salt ]\*2^16

index=2^16\*ROC+SEQ

对于生成的不同密钥，8位的label有不同的值：

session\_en\_key：消息加密密钥，<label>=0x00

session\_au\_key：消息认证密钥，<label>=0x01

session\_salt\_key：会话salt，<label>=0x02

（3）具体工作流程

得到会话密钥后，就可以对消息载荷进行安全操作了。

**发送方工作流程：**

1）确定加密上下文

2）根据ROC、加密上下文中的最高序列号以及RTP包中的序列号，确定SRTP包索引

index=2^16\*ROC+SEQ

3）根据index确定master\_key和master\_salt（需要密钥管理协议）

4）根据密钥管理协议中的各个参数，通过密钥生成器，得到会话密钥

5）利用会话密钥和传输参数设定，对载荷进行加密，作为数据包中的加密载荷

6）如果MKI字段为1，则加入MKI字段

7）计算认证标签，并加在包尾

8）更新ROC和数据包索引

**接收端流程**

1）确定加密上下文

2）计算包索引

index=2^16\*v+SEQ

v={ROC-1,ROC,ROC+1}/2^32

3）如果MKI字段为1，则根据MKI确定主密钥和主密钥salt，否则用index确定主密钥和salt

4）得到会话密钥

5）重播检查，利用index和重播列表来检查重播，如果为重播数据包，则丢弃，并记录log。

执行认证过程，如果不匹配，则丢弃，并记录log。

6）利用会话密钥解密数据载荷

7）根据index，更新ROC、最大序列号、加密上下文中的s\_l值，需要时更新重播列表。

8）从数据包中删除MKI和认证标签。

## DTLS-SRTP协议

DTLS-SRTP是DTLS的一个扩展，将SRTP加解密与DTLS的key交换和会话管理相结合。从SRTP的角度看，是为其提供一种新的key协商管理的方法；从DTLS的角度看，是为应用数据提供一个新的数据格式(SRTP/SRTCP)。

1，应用层数据加解密是由SRTP完成的，要求必须是RTP/RTCP的格式。  
2，DTLS的握手过程是为SRTP加解密过程协商使用哪种profile和密钥。  
3，除了应用数据加密为SRTP格式，其他record-layer的报文仍为普通的DTLS格式(比如TLS control message)  
4，当发送SRTP格式的应用层数据时，需要直接跳过DTLS加密层，将SRTP数据包透传到下层的数据传输层做发送。

由于密钥和加密参数是在DTLS握手过程中协商得到的，而此过程是保密的，因而相比常规的方式（比如在通过SDP消息交互来协商）更为安全。在发起DTLS握手之前，需要先设置use-srtp扩展。

接收端使用 DTLS-SRTP  
自DTLS下层的传输层收到报文之后，需要根据包头特征手动区分做demultiplexing，一般可以如下进行  
检查第一个报文的第一个字节  
1，是[0, 1]时，表示可能是STUN报文  
2，是[128, 191]时，表示可能时RTP(SRTP)报文  
3，是[20, 63]时，表示可能是DTLS record layer报文  
其他的类别请根据实际情况做区分处理

### SDES

DTLS-SRTP is a key exchange mechanism that is mandated for use in WebRTC.DTLS-SRTP uses [DTLS](http://webrtcglossary.com/dtls/) to exchange keys for the [SRTP](http://webrtcglossary.com/srtp/) media transport.

SRTP requires an external key exchange mechanism for sharing its session keys, and DTLS-SRTP does that by multiplexing the DTLS-SRTP protocol within the same session as the SRTP media itself.

This method is considered to be more secure than the [SDES](http://webrtcglossary.com/sdes/) mechanism that was first used in WebRTC but later on banned from use altogether.

DTLS:全称 Datagram transport layer security, 即udp + security，数据报层的安全，DTLS采用了TLS的安全机制，但是更轻量级，webrtc引入DTLS用于传输srtp数据包时的安全秘钥交换，dtls-srtp 在srtp基础上又提供了一层安全机制，比sdes更安全。

### 大致交互流程

webrtc使用sdp除了描述媒体类型，还有一些额外的字段来描述ice的连接候选项。

* chrome浏览器首先获取服务器提供的offer sdp，收到sdp之后，创建应答sdp和ice 候选项发送到服务器。
* 双方都收到sdp之后会首先进行ice连接（即一条udp链路）。
* 连接建立之后，发起dtls交互，得到远端和本地的srtp的key（分别用于解密远端到来的srtp和加密本地即将发出去的rtp数据包）。
* 然后就可以接收和发送rtp，rtcp数据了，发送之前要进行srtp加密，然后通过ice的连接发送出去。
* dtls 和 srtp 的数据包都是通过ice的udp连接进行传输的。

### Why Dtls-srtp

Q&A

<https://crypto.stackexchange.com/questions/50328/why-would-one-choose-dtls-srtp-versus-just-rtp-over-dtls>

为什么选择DTLS-SRTP而不选择RTP over DTLS？

DTLS是为了交换密钥，交换成功后双方开始发送SRTP加密码流。那为什么不直接用DTLS发送RTP，仅仅为了兼容SRTP协议栈吗？

原因是加密overhead的代价。DTLS头部13字节，。。。。，作为对比，SRTP是专门设计来减小overhead。这些代价对有线来说似乎无足轻重，但对移动设备和wifi网络连接来说却很关键，加密额外的字节不但耗费cpu，耗费电量，而且增加的字节会抢占有限的带宽。

[Why would one choose DTLS-SRTP versus just RTP over DTLS?](https://crypto.stackexchange.com/questions/50328/why-would-one-choose-dtls-srtp-versus-just-rtp-over-dtls)

If I understand DTLS-SRTP correctly, DTLS is used to exchange keys and then the endpoints switch to SRTP for encryption. What is the benefit of this setup versus just sending RTP over DTLS? Is it just about compatibility with existing SRTP stacks?

It's all about encryption overhead; how much the extra data the encryption method extends the packet by DTLS has a noticeable amount of overhead; the DTLS header alone is 13 bytes, and then you have the IV/nonce, and the tag; this overhead can be more than the actual VoIP payload. In contrast, SRTP was specifically designed to minimize this overhead; except for the tag (which is optional; IMHO, bad idea to omit it, but some people insisted), there is no overhead compared to RTP.

You might ask "what's the big deal about encryption overhead? Doesn't the internet not care that much about packet sizes?" Well, yes, if you're talking about wired internet connections, actually, this overhead might not be that significant. However, for wireless, yes, people do worry about it, because:

* Because of power; the more bytes you have, the more bytes need to be transmitted (and if you're on a battery, well, that's a concern)
* Because wireless is a shared media, so the more bytes you broadcast, that's less bandwidth everyone else connecting to the same AP gets.

### What’s SDES

## Webrtc SDP协议

Webrtc使用SDP和offer-answer模型进行peers之间的会话协商。

### 参数含义

// RFC 4566

v=(protocol version)

o=<username> <sess-id> <sess-version> <nettype> <addrtype> <unicast-address>

s=(session name)

i=\* (session information)

u=\* (URI of description)

e=\* (email address)

p=\* (phone number)

c=\* (connection information -- not required if included in all media)

b=\* (zero or more bandwidth information lines)

// RFC 5888 and draft-holmberg-mmusic-sdp-bundle-negotiation-00

// a=group:BUNDLE video voice

"a=group:BUNDLE";

Session\_description contentgroup : video voice

// Zero or more media descriptions

// RFC 4566

// m=<media> <port> <proto> <fmt>

// There are multiple variants of the RTP protocol stack, including

// UDP/TLS/RTP/SAVPF (WebRTC default), RTP/AVP, RTP/AVPF, RTP/SAVPF,

// TCP/DTLS/RTP/SAVPF and so on. We accept anything that has RTP/

// embedded in it somewhere as being an RTP protocol.

**a=extmap-allow-mixed**

When doing SDP Offer/Answer [[RFC3264](https://tools.ietf.org/html/rfc3264)], an offering client that wishes

to use both one-byte and two-byte extensions MUST include the

attribute "a=extmap-allow-mixed" in the SDP offer. If

"a=extmap-allow-mixed" is present in the SDP offer, the answerer that

supports this mode and wishes to use it SHALL include the

"a=extmap-allow-mixed" attribute in the answer. In the cases where

the attribute has been excluded, both clients SHALL NOT use mixed

one-byte and two-byte extensions in the same RTP stream but MAY use

the one-byte or two-byte form exclusively (see [Section 4.1.2](https://tools.ietf.org/html/rfc8285#section-4.1.2)).

#### mid

a=group:BUNDLE 0 1 2

a=mid:0

a=mid:1

a=mid:2

The read-only RTCRtpTransceiver interface's mid property specifies the negotiated media ID (mid) which the local and remote peers have agreed upon to uniquely identify the stream's pairing of sender and receiver.

#### msid

a=msid-semantic: WMS bESa4hDt2y4kZEizBkB4vbj7DQpC6hU7XTLD

This lines gives an unique identifier for the WebRTC Media Stream (WMS) during the PeerConnection’s life. This identifier that will be used in the a=msid attributes for each m-line belonging to a specific Media Stream (in our case both audio and video m-lines). This means that the RTP media stream (identified by the SSRC field present in every RTP packet) belongs to that media stream and that it is a track of that media stream. It is an explicit association of an individual RTP media stream to the MediaStream WebRTC object. For more info about this refer to draft-ietf-mmusic-msid

#### setup

a=setup:actpass

a=setup 主要是表示dtls的協商過程中角色的問題，誰是客戶端，誰是服務器

a=setup:actpass 既可以是客戶端，也可以是服務器

a=setup:active 客戶端

a=setup:passive 服務器

#### Unified plan 与 planB

unified plan 一个m行对应一个track，即ssrc和cname；老式的palnB 一个m可以对应多个track，即多个ssrc

https://docs.google.com/document/d/1PPHWV6108znP1tk\_rkCnyagH9FK205hHeE9k5mhUzOg/edit#

例子：https://ouchunrun.github.io/2018/10/23/%E2%80%9CUnified%20Plan%E2%80%9D%20%E8%BF%87%E6%B8%A1%E6%8C%87%E5%8D%97/

## Draft-ietf-mmusic-msid

From <https://datatracker.ietf.org/doc/draft-ietf-mmusic-msid/>

描述RTP媒体流的SDP分组机制和RTP媒体流的关系

MediaStream/MediaStreamTrack

## HTTP协议

## RTP协议

## ICE协议

voice activity dectctor

# 音视频基础

## 音频

1. vad与cn

vad：voice activity dectctor

## 视频

# The CODE（代码部分）

## 日志

打开日志

## 线程管理

ECMedia生成并管理worker\_thread, signaling\_thread和network\_thread

### 线程调用要求

- 在videoTrack上add sink或remove sink

AddOrUpdateSink and RemoveSink should be called on the worker thread.

- render线程是UI线程的子线程

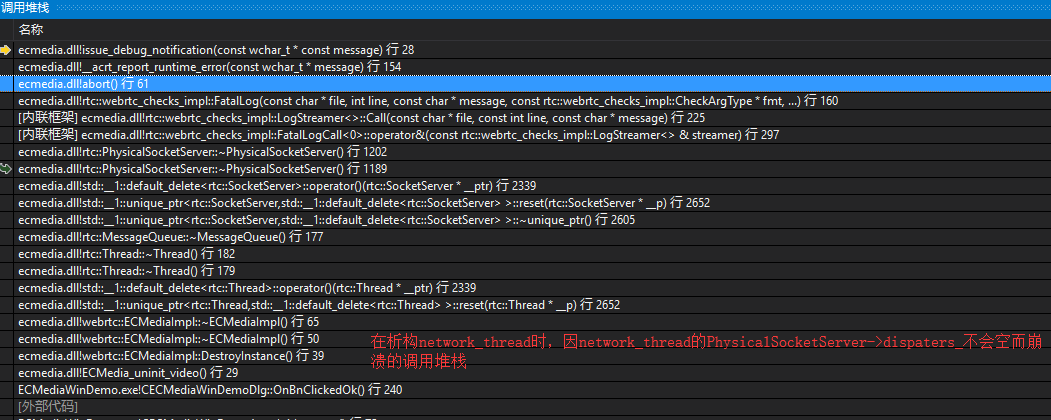
#### Signaling Thread

1. Remote\_render的创建和释放要在signaling\_thread线程上。

#### Worker Thread

#### Network Thread





### 创建线程

如何合理创建这3个线程，是否用到socketserver？mac demo下的创建方式可参考。

##### 视频发送线程

worker\_queue类型是rtc::TaskQueue \*,在VideoSendStream构造函数中指向RtpTransportControllerSendInterface的worker\_queue，也就是说没有独立的视频发送线程，所有的rtp发送都由RtpTransportControllerSendInterface来控制。

VideoSendStream的成员变量包括VideoStreamEncoder、VideoSendStreamImpl

在Call里创建VideoSendStream，与ssrc一一对应。



为什么不发送视频流？以下四个条件都具备才开始发送，目前的状态是本地sendrecv，远端inactive，enable是false，writable是false。所以要一一排查。







没有可用的connection，selected\_connection\_为空



##### 视频编码线程

encoder\_queue类型是rtc::TaskQueue,在VideoStreamEncoder中创建，VideoStreamEncoder代表一个视频编码器，输入raw Video frame，输出编码后的码流。

// VideoStreamEncoder represent a video encoder that accepts raw video frames as

// input and produces an encoded bit stream.

// Usage:

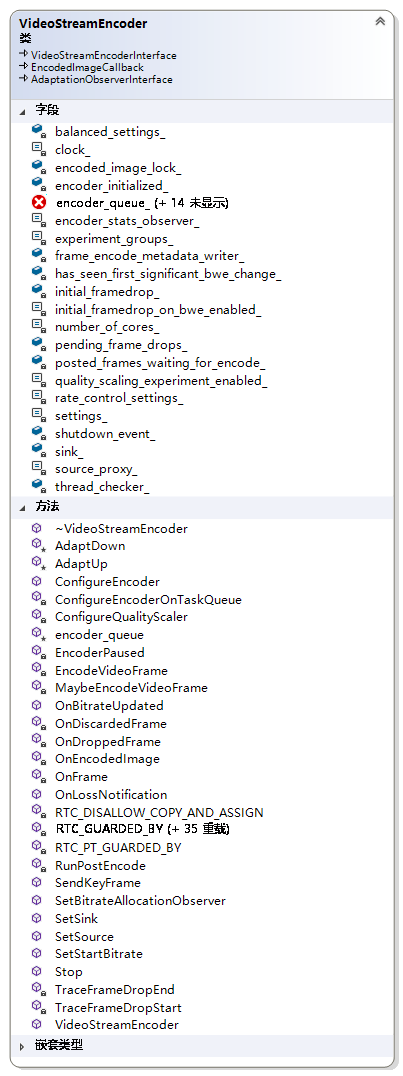
// Instantiate.

// Call SetSink.

// Call SetSource.

// Call ConfigureEncoder with the codec settings.

// Call Stop() when done.



## 视频采集

video\_capture是如何实现平台适配的？

### VideoCapturer类

VideoCapturer 是个中间类，既是source，又是sink。它继承了VideoSinkInterface，可以收到设备层采集的raw frame data，同时它又继承了VideoSourceInterface，可以把数据传递给注册进来的VideoSinkInterface，注册的VideoSinkInterface放在broadcaster\_成员变量。







### VideoCaptureModule类

跨平台设备接口，屏蔽底层硬件的差异。

## 视频显示

video\_render是如何实现平台适配的？

## PeerConnection

### UnifiedPlan与PlanB

With Plan B SDP, an RtpTransceiver can have any number of senders and receivers which map to a=ssrc lines in the m= section.

With Unified Plan SDP, an RtpTransceiver will have exactly one sender and one receiver which are encapsulated by the m= section.

在pc中为了兼容性两者并存。默认使用UnifiedPlan。注意不要看PlanB相关的接口。如：AddStream()是在PlanB下调用的，在UnifiedPlan下不会调用。

### AddTrack



函数目的：当pc调用AddTrack()时，创建（或使用已创建）与track对应的RtpSenderInternal和可能会用到的RtpReceiver，二者一起构造RtpTransceiver对象，放入transceivers\_变量。

其中RtpSendre与track关联，见下面RtpSender的类图。

生命周期：

AudioTrack和VideoTrack都是在pc外部创建，在AddTrack时通过指针方式传给transceiver\_。

标识参数：

track->kind()返回音频或视频。

track->id() 返回track id，在创建track时指定。

stream\_ids 在增加track时指定，音视频track使用相同的stream\_id

AddTrack() 为音频和视频分别创建一个RtpTransceiver对象，这两个对象都放入transceivers\_中，两个transceiver的stream\_ids相同，关联的track不同，media\_type也不同。

VideoTrackInterface的类图如下：



### RTPTransceiver

The RtpTransceiver is specialized to either audio or video according to the MediaType specified in the constructor. Audio RtpTransceivers will have AudioRtpSenders, AudioRtpReceivers, and a VoiceChannel. Video RtpTransceivers will have VideoRtpSenders, VideoRtpReceivers, and a VideoChannel.

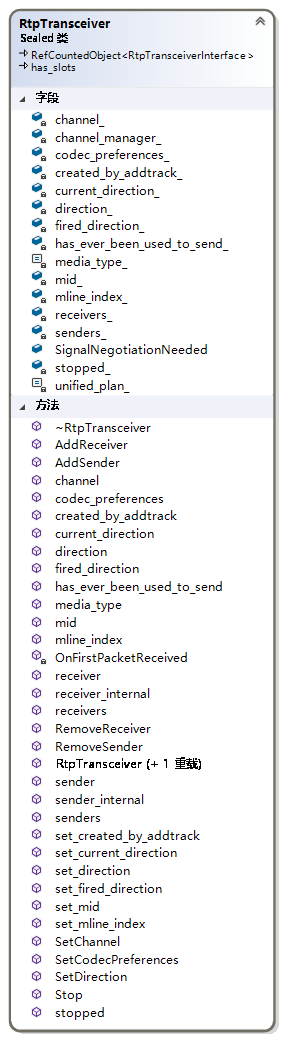
把m行的sender、receiver和channel组合在一起。

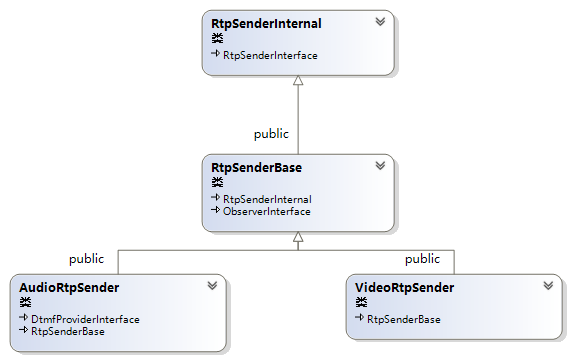
sender\_id

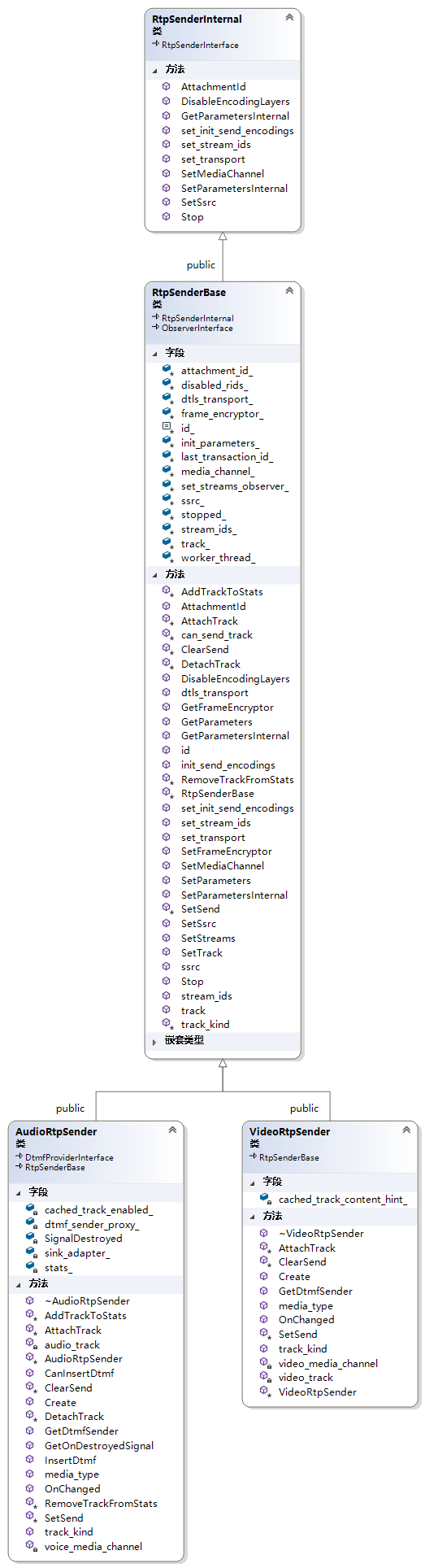
receiver\_id

media\_type() 取值：MEDIA\_TYPE\_AUDIO、MEDIA\_TYPE\_VIDEO

看RtpSenderInternal的类图，该图关联了Track、media\_channel和dtls\_transport









### ChannelManager

创建VoiceChannel、VideoChannel或RTPDataChannel

## SDP交互

相关的文件：session\_description.cc/h webrtc\_sdp.cc/h

类SessionDescription包含：ContentInfo、TransportInfo、MediaTransportSetting

"a=group:BUNDLE"

A=msid-semantic:WMS stream-id

在每个m行下有a=msid行，比如在m=audio下有：m=msid:stream-id audio-lable

说明此track属于stream-id标识的流，该track的id是audio-lable

一个track可以属于不同的stream

### 生成local SessionDescription

PC下包含一个调用RTPTransceiver对象transceivers\_，如果希望本地发送音视频数据，需要调用AddTrack() 分别增加一个音频Transceiver和视频Transceiver到pc中，这样，在生成本地SDP时就会包含对应的m行。

AddTrack() 的作用：增加对应track的RtpTransceiver接口，默认的方向是send and recv，既有发送又有接收，同时将track对应到RtpSendInternal接口。但此时并没有协商媒体编码和传输等信息。只是将本地可以提供的track（源）和发送接收Rtp通道准备好。注意这里只是Rtp通道，真正的底层发送由JsepTransport接口提供，是需要SDP来协商。

Transceiver包括send和receive。

AudioTrackSource和VideoTrackSource分别将本地的音频源和视频源与Track连接，作为RTPTransceiver的输入。

1. 增加音频track

调用pcf创建AudioTrackInterface，调用pc AddTrack()即可。



1. 增加视频track

与音频类似，不同的是在调用pcf创建VideoTrackInterface时需要传入VideoTrackSourceInterface参数。



1. CreateOffer

在增加音视频track后， ECBaseManager::GenerateLocalDescription()调用peer\_connection\_->CreateOffer()产生local\_description。CreateOffer异步执行，调用成功后在ECBaseManager::OnSuccess(webrtc::SessionDescriptionInterface\* desc\_ptr)返回SDP描述。









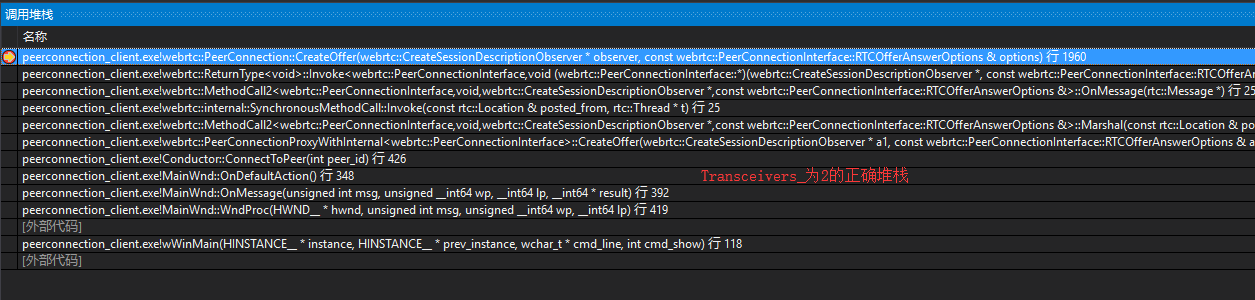
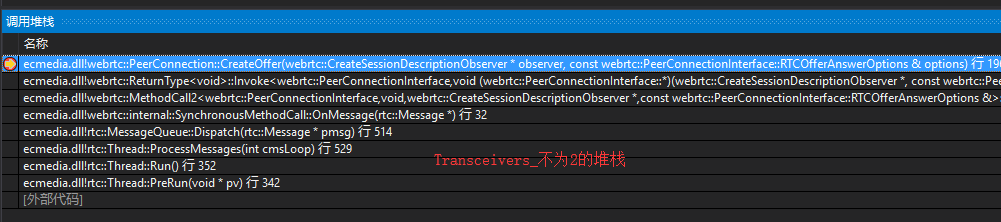
1. SetLocalDescription

在回调中设置本地sdp，会在底层创建传输等基础构件。具体包括：xxx

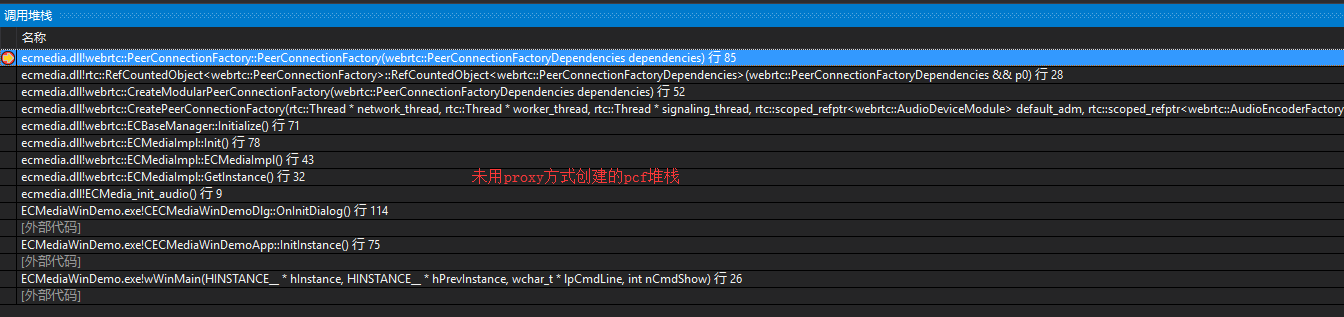
peer\_connection\_->SetLocalDescription()

==============调试过程 begin============================

Transceivers\_正常情况下为2,



~~跟踪到是pcfactory和pc的创建方式有问题：~~

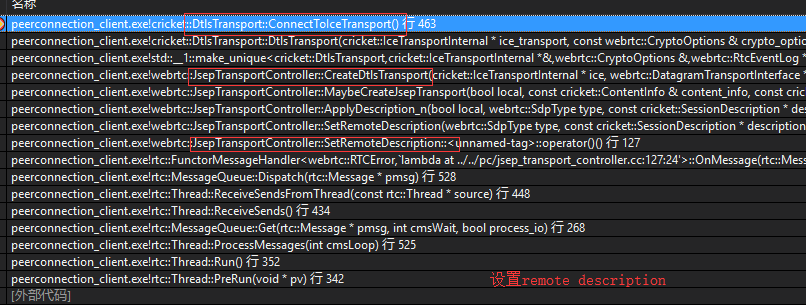


在WebRtcSessionDescriptionFactory::CreateOffer调用时判断证书状态，



调用void WebRtcSessionDescriptionFactory::CreateOffer创建offer

收到offer后创建传输层对象，由JsepTransportController创建DtlsTransport，DtlsTransport关联到IceTransport





================调试过程 end================================

### 生成remote\_SessionDescription

在生成local\_sdp之后用此sdp作为参数调用peer\_connection\_->SetRemoteDescription，在这里会添加

如果A发送Offer，A 调用 SetLocal



### 设置candidate

0x00000234eb1c3450 "candidate:4107283943 1 udp 2122260223 192.168.177.91 60707 typ host generation 0 ufrag +9/Q network-id 3"

0x00000188e7c36fd0 "candidate:3659473419 1 udp 2122260223 192.168.0.14 55251 typ host generation 0 ufrag CdV9YggnP4OX0CRY network-id 3"

0x000002369bc04000 "candidate:3126055191 1 tcp 1518280447 192.168.177.91 9 typ host tcptype active generation 0 ufrag sy/L network-id 3"

AddIceCandidate

0x0000014c45044710 "candidate:3659473419 1 udp 2122260223 192.168.0.14 49586 typ host generation 0 ufrag sPJhOxUVexeXc/Uz network-id 3"

0x0000014c44f47af0 "candidate:739204063 1 udp 1686052607 124.206.192.2 50493 typ srflx raddr 192.168.0.14 rport 50493 generation 0 ufrag gAzidTnsScMka9ig network-id 3"

调用AddIceCandidate之后

创建connection的打印信息：

(connection.cc:265): Conn[f6c06240:0:Net[Realtek:192.168.177.91/32:Ethernet:id=3]:2xTHG8/6:1:0:local:udp:192.168.177.91:9078->vWMvEl0D:1:2122194687:local:udp:192.168.177.61:9078|C--W|-|0|0|9114756780654476799|-]: Connection created

### 媒体传输

### 设置远端Jsep传输描述

transport->SetRemoteJsepTransportDescription

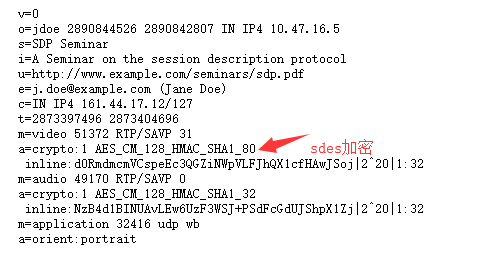
### crypto的验证

audio、video和data支持两种加密：dtls和sdes，默认使用dtls。

在支持group:bundle的情况下，每个m行都有对应的mid，crypto使用的是第一个m行的加密信息。

如果使用dtls：在m行下必须有

如果使用sdes：在m行下必须有a=crypto:



### RTP/RTCP包接收过程

1. 从底层socket收到加密的srtp包，其中，RTP包和RTCP包的处理流程是不一样的
   1. 如何判断收到的SRTP包是rtp还是rtcp。涉及到srtp协议，包头类似rtp， payload\_type占7bit。

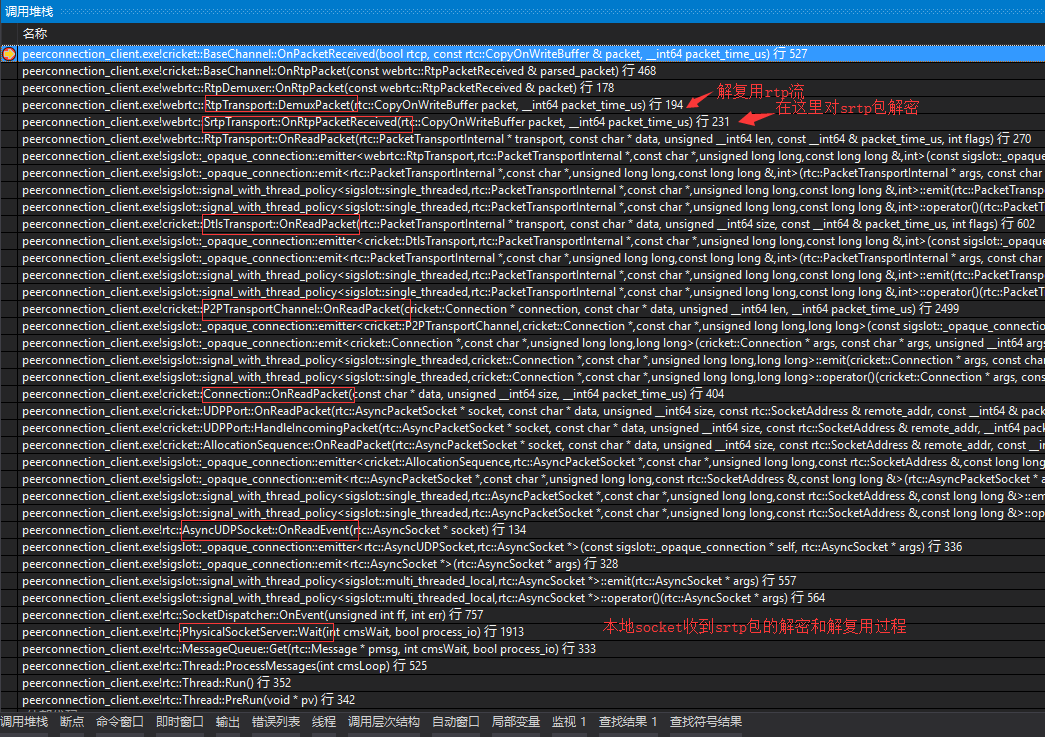


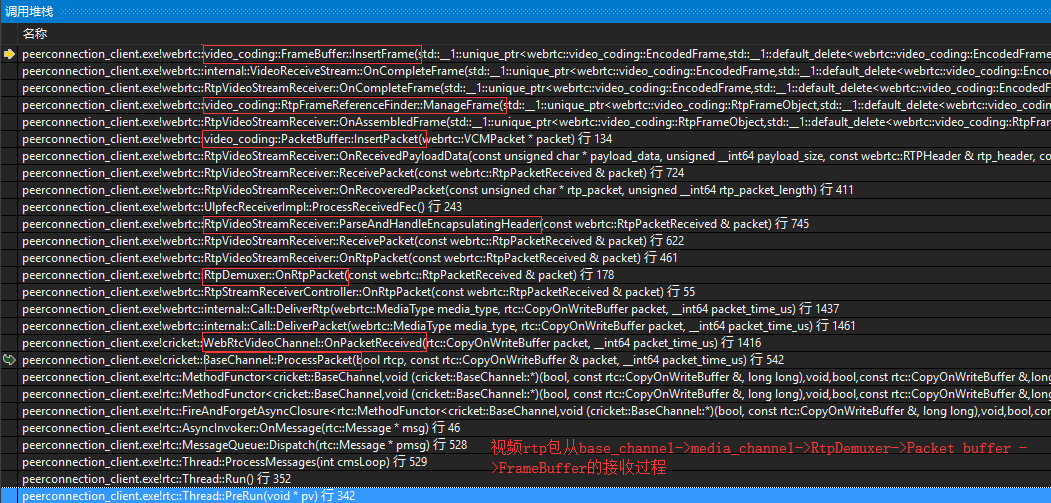
* 1. Rtcp包的处理流程

#### 音视频RTP包的接收

音视频RTP包的接收过程大致为：底层socket收包，…，DTLS判断是SRTP包，SRTP解密，RtpDemux解复用、传递给VoiceChannel或VideoChannel，以上是在network\_thread线程中处理；然后由basechannel传递给media\_channel处理，这里jitterbuffer换成了RtpFrameReferenceFinder，这部分是在worker\_thread线程完成。

以下两张堆栈图，分别代表在network\_thread线程和worker\_thread线程经过的类和调用关系。





解析RTP包头，depacketizer

void RtpVideoStreamReceiver::ReceivePacket(const RtpPacketReceived& packet) {}

插入packetBuffer对象

packet\_buffer\_->InsertPacket(&packet);

如果帧组装完整，通过回调返回完整帧

assembled\_frame\_callback\_->OnAssembledFrame(std::move(frame));

完整的帧由RtpFrameReferenceFinder管理，查找该帧的参考帧，对一帧的处理结果FrameDecision为则handoff，drop or stash，handoff的帧就是可以传递出去的帧

FrameDecision decision = ManageFrameInternal(frame.get());

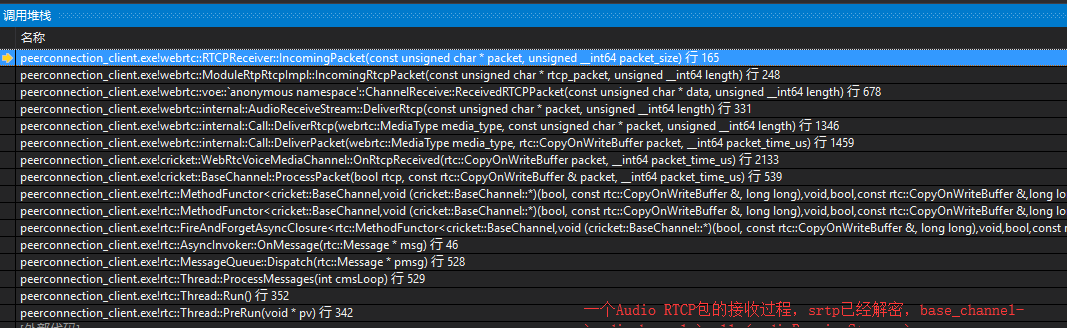
frame\_callback\_->OnCompleteFrame(std::move(frame));

将插入framebuffer

int64\_t last\_continuous\_pid = frame\_buffer\_->InsertFrame(std::move(frame));



1. 



# rtc\_base库

## 多线程

### rtc::PlatformThread

// Represents a simple worker thread. The implementation must be assumed

// to be single threaded, meaning that all methods of the class, must be

// called from the same thread, including instantiation.

该类代表一个简单的工作线程，对象的初始化和方法必须是由同一个线程调用

## 可独立使用的类

### Class CopyOnWriteBuffer

### rtc::CriticalSection

### rtc::Event

### rtc::StringBuilder

## 模板类

### rtc::scoped\_refptr<>

RAII方式的的引用计数对象，

### RefCountedObject<>

支持std::move语义

### absl::c\_any\_of



### absl::c\_none\_of



### absl::c\_all\_of

### absl::c\_remove\_copy\_if

### absl::c\_linear\_search

### rtc::ArrayView

相关RFC

RFC 3711 - SRTP

RFC 5705 - Keying Material Exporters for TLS

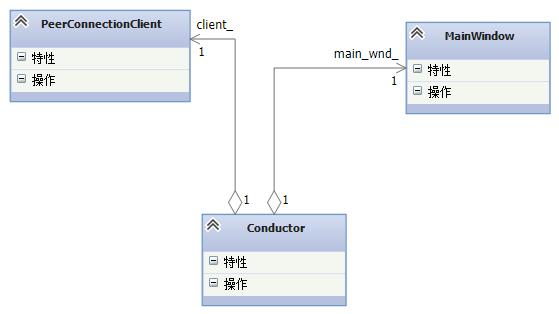
RFC 3264 - An Offer\_Answer Model with Session Description Protocol (SDP)

RFC 4566 - SDP\_ Session Description Protocol

# WindowsDemo

## 主要的类图

wWinMain()函数创建PeerConnectionClient对象、MainWindow对象，以此二者为参数创建Conductor对象，后者派生自PeerConnectionClientObserver和MainWndCallback，在Conductor的构造函数中分别把this注册回PeerConnectionClient对象和MainWindow对象。





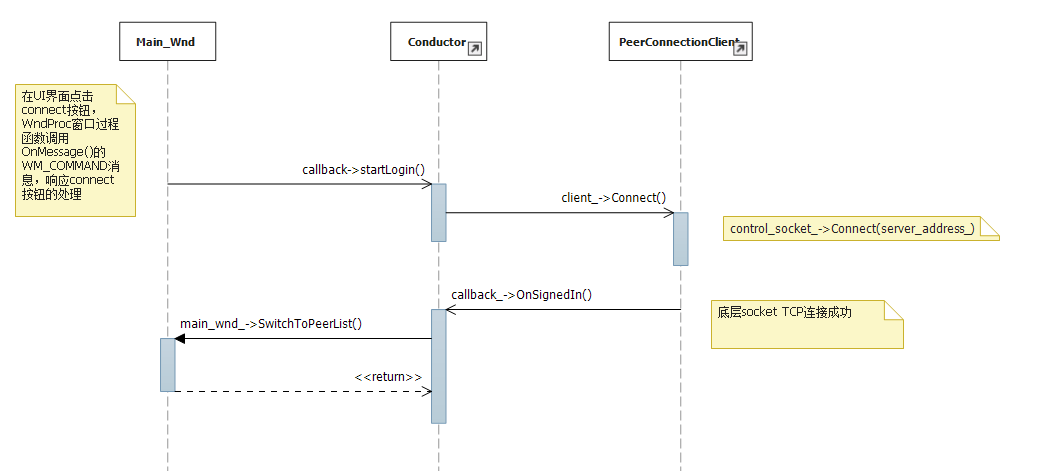
MainWindow是主窗口类，纯虚类，与平台无关。MainWnd派生自MainWindow，在其中添加了诸多控件对象，是windows平台下的主窗口实现。

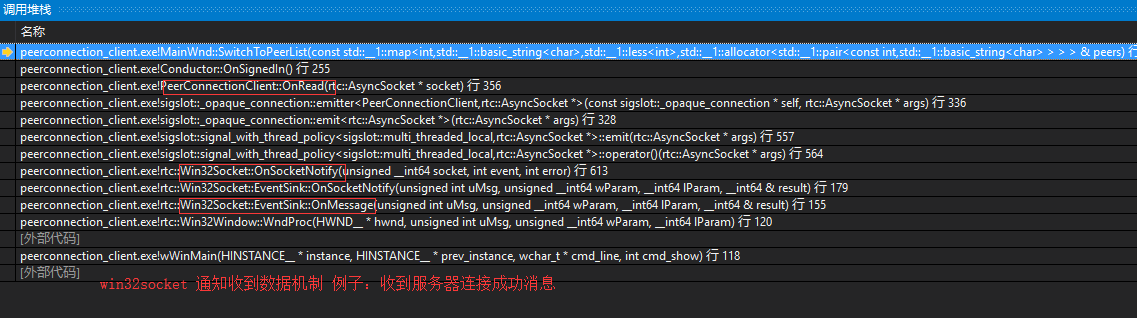


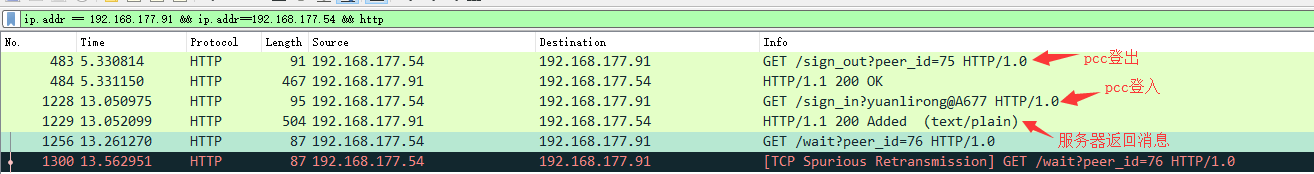
## 时序图

### pcc连接服务器

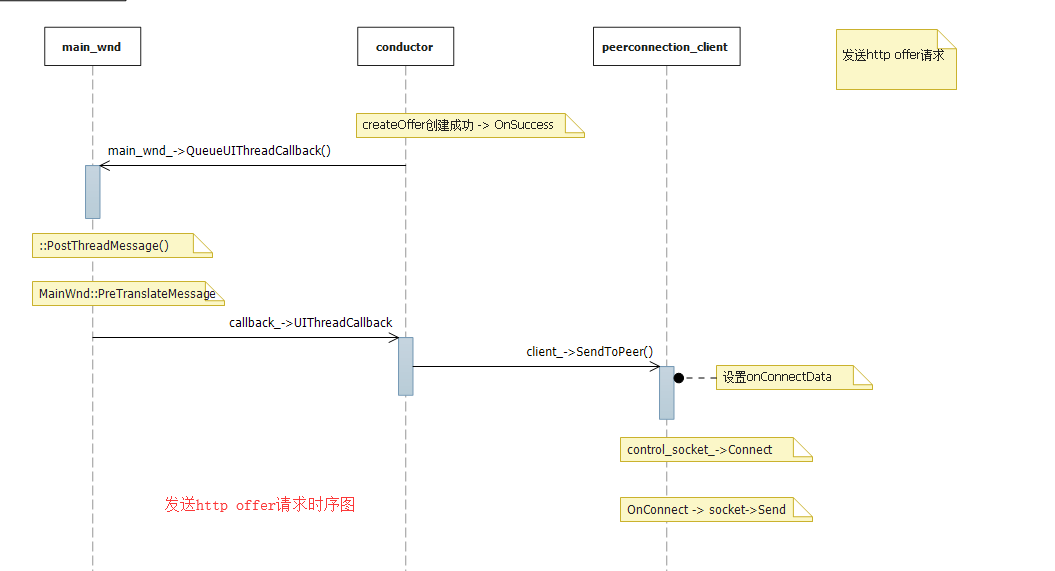
说明：Conductor在构造时把this注册到PeerConnectionClient和MainWindow中，在二者发生的事件都可以回调到conduct执行。







### 发送http offer请求



# 博客网站

讲p2p的建立过程：

<https://blog.piasy.com/2018/07/31/WebRTC-P2P-part2/index.html>