# Diagnosing Media Issues on the Fusion Platform

**Version V.1.0**

CaféX Communications
135 West 41st Street,
Suite 05-108,
New York,
NY 10036
www.cafex.com

## Version History

| Document Control | | |
|---|---|---|
| **Version** | **Author** | **Description/Change History** |
| 1.0<br>November 15th 2019 | CaféX Support (TH) | A guide to determine typical media issues that may be seen using CaféX Fusion Client SDK or Fusion Live Assist. |
| | | |
| | | |
| | | |

## Table of Contents

# Introduction

This document gives a very basic overview of how RTP traffic traverses a typical network and will describe how to:

- Analyze a typical SIP call flow through Fusion Client SDK
- Understand the media paths established between FCSDK and other network devices
- Understand common reasons for media establishment failure
- Quantify Packet Loss for a call from an iPAD Client
- Quantify Packet Loss for a call from a Chrome Browser Client
- Quantify Packet Loss at the Fusion Media Broker

## Use of Third Party Tools

This document uses a number of industry standard third party tools, such as Wireshark. It may be that the user interfaces in this document will change in these tools.
As a reader, it is more important you understand why the tools are being used and prepare for differences in the step by step process.

# Streaming Traffic Overview

Here will introduce common terminology for describing all forms of Audio and Video Streams.

Generally speaking a stream is a sequence of packetized media samples.  Typically, audio and video codecs have an associated sample rate.A codec also describes the clock rate; which is the rate the audio was sampled.

For example:

- The G711 audio codec suggests (but does not mandate) that 20ms of audio are transmitted per packet.
- G711 has a clock rate of 8000Hz; (eg 8000 samples per second).
- Each packet will contain 160 samples.

**Note:** G711 doesn't mandate 20ms of audio, so a packet may contain more of less than 160 samples; as a result the receiver must handle these variations to construct an audible stream.

The RTP packets contain information to help the far end reconstruct the stream, below is an example of an RTP packet:

```
⊟ Real-Time Transport Protocol
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 .... = Extension: False
    .... 0000 = Contributing source identifiers count: 0
    0... .... = Marker: False
    Payload type: ITU-T G.711 PCMU (0)
    Sequence number: 41853
    Timestamp: 1973995500
    Synchronization Source identifier: 0xabfbc9d8 (2885405144)
    Payload: b4c35e2998a047d3d52e5112c9102adac61609592d3fe0ab...
```

The *Payload* contains the data the was sent in this packet.

The *Sequence Number* identifies a packet's position in a stream.  The first packet of a stream will assign a random number and every subsequent packet will be incremented by one.  The Sequence Number's primary objective is to identify lost packets.

The *Timestamp* is used to construct the sample with the correct timing.  In this stream, the previous packet had a Timestamp of: 1973995340.  Thus, the receiver knows that this sample contains 160 samples(ie: 1973995500 - 1973995340 =160).    Multiple packets in a stream will have identical Timestamps if: they have been retransmitted or there are one of multiple packets used to create a sample (such as a video frame spread over many packets).

## Stream Metrics

Assuming a well behaved sender, generally three metrics are used to measure the quality of the stream at the receivers end.  This metrics can be used by receivers when reconstructing streams.  They are:

- ***Latency***:  The time it takes to get a packet from sender to receiver.
- ***Skew***: If the stream is a G711 steam, one typically expects samples to be arriving every 20ms.  Skew measures cumulative lateness of a given packet relative to the previous packet and since the start of the stream.
  - In theory, the 100th packet should arrive 2000ms from the start of the stream; if it arrives at 2020ms and the 99th packet arrived at 1980ms, the packet is considered 20ms late and the skew will be measured as -20ms.
  - Small amounts of skew fluctuation should be managed by a receiver.
- ***Jitter***: Generally it is a measure of the "Packet Delay Variation".
  - It is a good way of comparing one point of a given call with another.

# Video Traffic Overview

When a Video stream is established between two video enabled devices there are independant RTP/UDP streams which flow in either direction between the devices. The video stream is broken up into packets and these packets make up two fundamental components that a video devices needs in order to construct a coherent image:

- Keyframes
- δ-frames (delta-frames)

You should remember that video codecs and compression can drastically change how these frames are sent and received. For now, we will assume there are no Error Correction algorithms such as: PLI or NACK which can be used to trigger a new keyframe if required or resend a packet if it is missing. Similarly, Forward Error correction algorithms provide information in the stream to verify and correct a stream if something is lost.

## Keyframes

A key frame is used to construct an entire image which can be displayed. It contains all of the necessary information to render an image and is not dependant on any other parts of the stream.

A key frame will span multiple UDP packets (depending on the resolution of the video).

Keyframes are an important way of allowing a decoder to refresh and start again if things are going badly.

## Delta Frames

δ-frames are collections of packets which only contain parts of the previous image. They only contain information which has changed since the previous frame. A video device will interpret these delta-frames extrapolate an image. Below is an example showing 2-keyframes (numbered 1 and 5) and 3 δ-frames (numbered: 2,3,4); the bottom of the image shows what the video device renders.

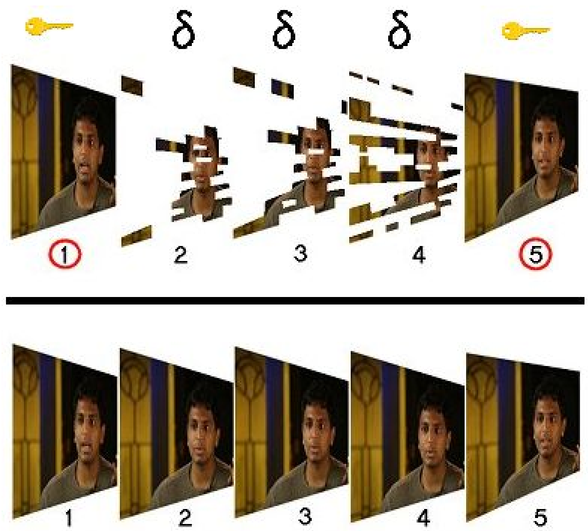



Image from: [http://nickyguides.digital-digest.com/keyframes.htm](http://nickyguides.digital-digest.com/keyframes.htm)

There are many reasons for packet-loss on a network.  This could be due to: weak wifi signals, high network contention, high network throughput and Quality of Service guarantees implemented by networks.  It must be understood that:

- If there are UDP packets lost for δ-frames then a video device will not be able to extrapolate a coherent image until a keyframe arrives.  This is displayed as a partially corrupted or pixelated image.  When a keyframe arrives it allows the video device to render a fresh image.
- Keyframes are constructed from multiple packets; If one of these packets is lost then the video device will not be able to render the new keyframe and will attempt to continue extrapolating images using δ-frames until the next suitable keyframe arrives.

**Note:** Video is more sensitive to lost or corrupt data compared to audio, this is for a number of reasons:

- Spoken audio is can be mostly silence, so you don't notice
- Our eyes are very good at detect subtle changes
- Corruption is often cumulative

# Diagnosing Media Issues

Diagnosing media issues has some absolute requirements which you cannot work without. There are also nice to haves which will make life easier but are not necessary in most situations. If you have the **Mandatory** items it may not be necessary to collect the **Additional Logs**.

## Mandatory

- Understanding of the call flow and Architecture
- A Media Broker packet capture of a complete single call displaying the issue:
  - Media Broker Logs (DEBUG by default) of a single call with the issue
  - Media Broker pcap of a single call with the issue
- Calls.log from FAS (if WebRTC to SIP)

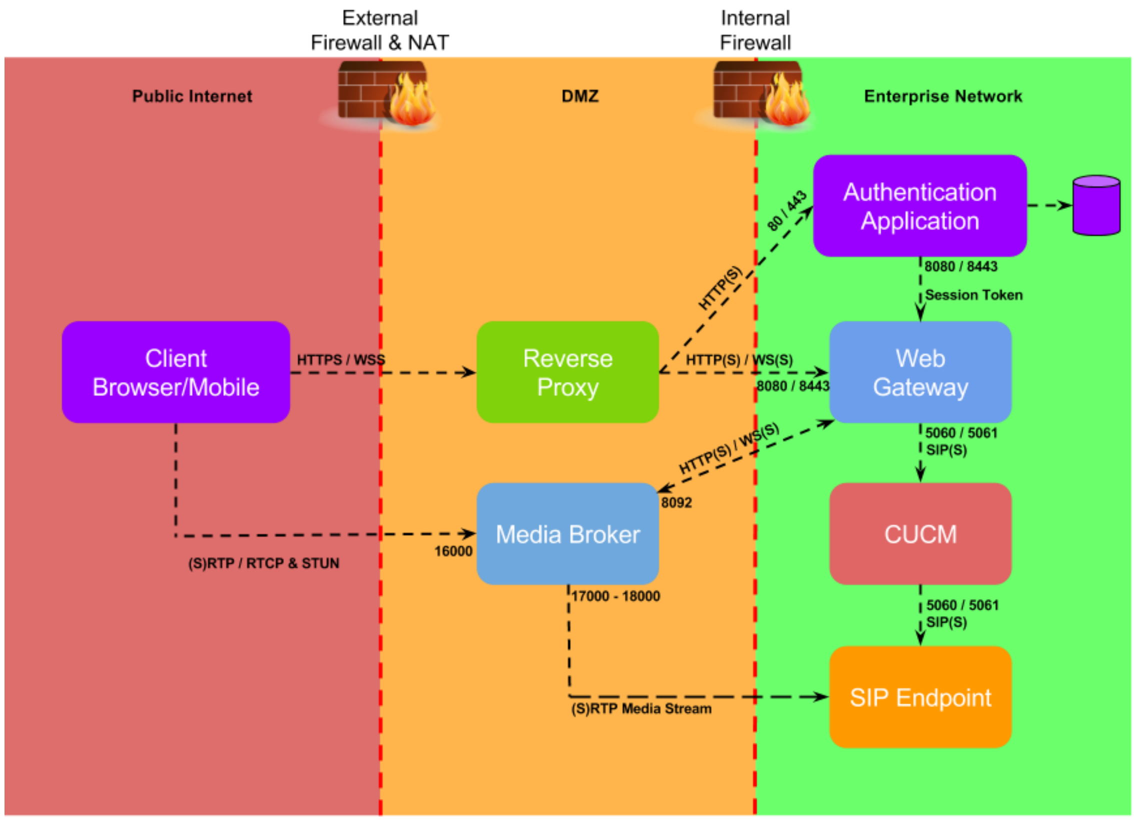
## Additional Logs

- Gateway Config XML or DEBUG server.log from Gateway
- Media Broker Logs (DEBUG by default) of a working call
- Media Broker pcap of a working call
- iOS/Android console logs
- Web Console Logs
- DEBUG FAS Logs

## Other Useful Information

Information such as version numbers etc, listed above, can normally be found from the Mandatory logs. Steps to collect logs can be found in the product troubleshooting guides found at: https://support.fusion.cafex.com.

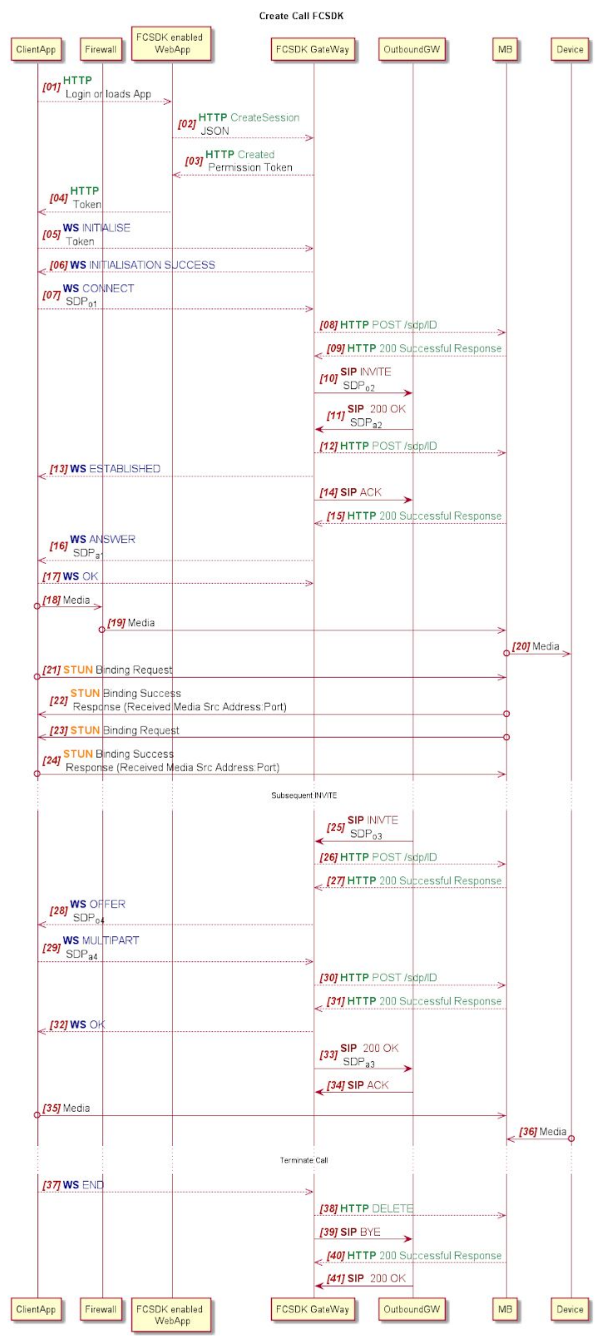
# Architecture Overview




All communications between WebRTC Clients and FCSDK are secure. The Media DTLS handshake will be covered in more detail, but this prevents packet inspection; however, some important details can be gleaned. By default SIP side transactions and media are not encrypted, so more inspection can be performed.

## Call Flow

A typical scenario for FCSDK is for the FCSDK client to callout to a SIP endpoint via the Fusion Gateway and through a PBX or contact center.

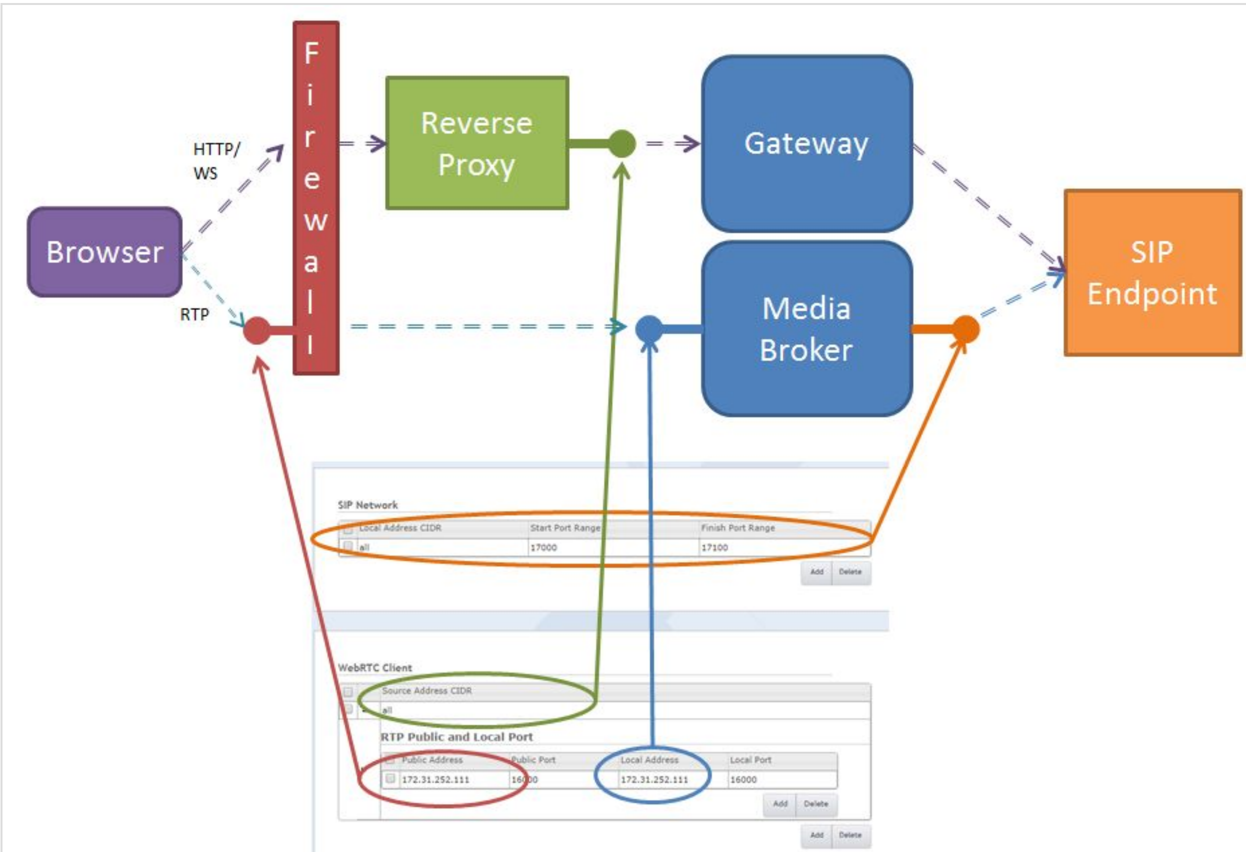


The FCSDK Gateway allocates a Media Broker to direct a call when a new request is received. Media Broker allocates a process and ports for the traversal of media. For a given call, codec and video resolutions are fixed. Initial information about the streams can be retrieved the transactional SDP at the start of the call.

Before initiating any detailed packet analysis, it is worth trying to understand the SDP negotiations, so you can hypothesis, what you expect each client and the Media Broker to be doing. This may simply be checking codecs and ports, but also understanding if any other transactions may result in a misunderstanding of each other's protocols.

## Understanding Fusion Media Broker Ports

This diagram explains what values should be specified when adding Media Broker configuration to a FCSDK installation:




- **SIP Network**
  - *Local Address CIDR* - is the address range the Media Broker will bind to for RTP communications on the SIP Network.

**Note**: If you have 2 network interfaces on the box don't use 'all' as the CIDR but target the internal interface only For example X.X.X.X/32

- **WebRTC Client**
  - ***Source CIDR Address*** - is the address range on which the Gateway will

receive WebRTC traffic from clients

**Note**: If all external traffic comes via a single ReverseProxy you can create a rule with the ReverseProxy internal address as the Source CIDR (X.X.X.X/32) with the Public address as the external firewall address. Then to allow internal clients to connect directly to the Gateway you can create a 2nd rule with 'all' as the Source CIDR with the Public address as the internal Media Broker address.

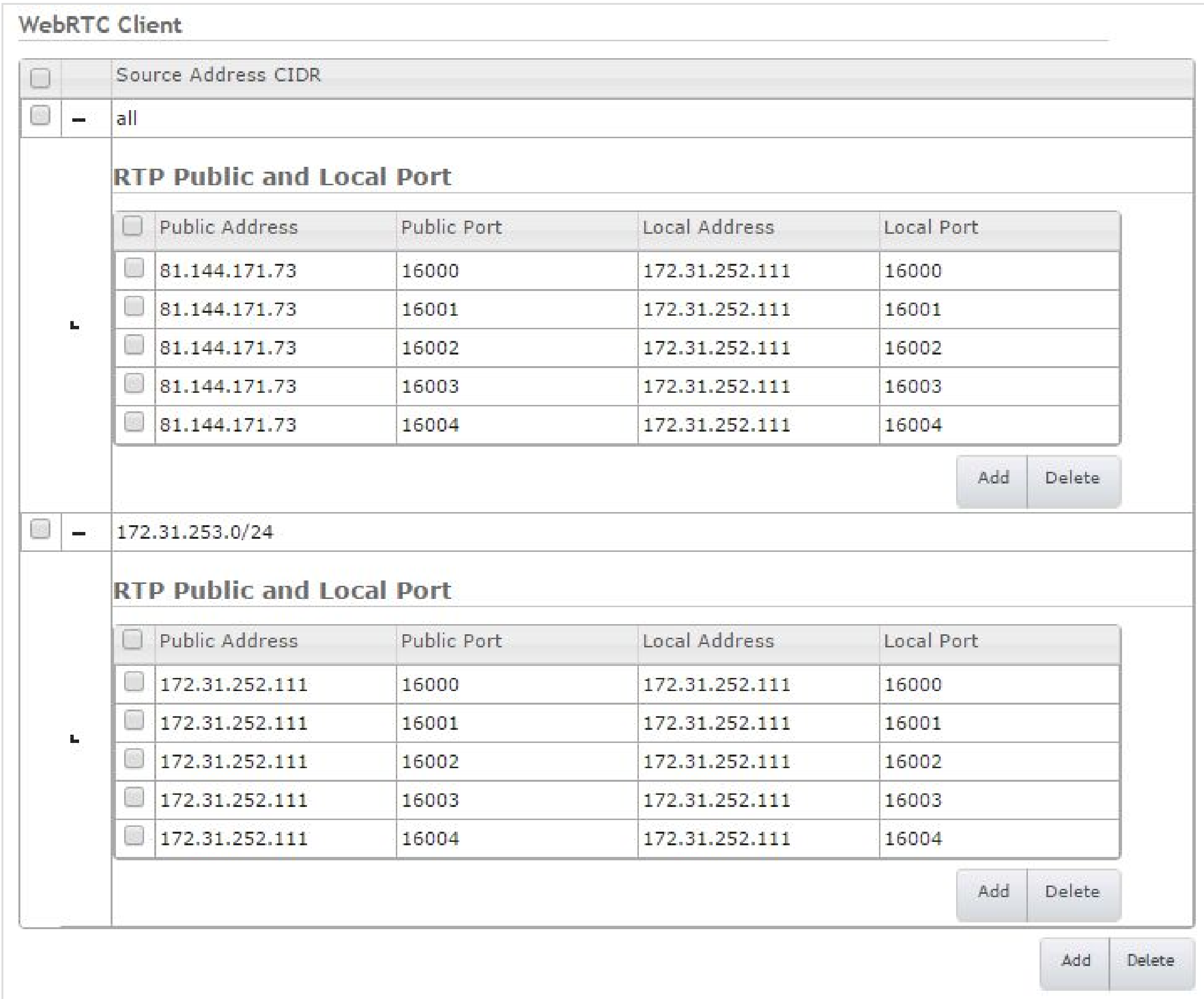
- ○ ***Public Address*** - is the address the client must send RTP traffic to; typically the front of a firewall.
- ○ ***Local Address*** - is the address the Media Broker will bind to in order to receive RTP traffic

## Configuring Multiple Media Broker Ports

Media Brokers of FCSDK 2.1.31 introduces simultaneous rtp-proxy processes for managing calls. This impacts how ports are allocated between these processes.

- SIP port Range - These ports are distributed across the rtp-proxy instances, in groups of 4.
  - Number of SIP-Ports to allocated = (4 ports for every WEB-RTC Client per call)x( Maximum Number of Concurrent calls on a Media Broker) + (a small contingency [eg: 10%]).
  - Ports are not reallocated immediately when a call is ended, so on smaller systems the contingency should be a larger percentage.
- WEB-RTC Port Range:
  - Number of WEB-RTC Ports to Allocated = (Number of rtp-proxy processes [default is 5]).
  - It is necessary to allocate the same number of ports to each *Source CIDR Address* to ensure that each rtp-proxy process can assign the correct interface/port pair to a call.

## Example Configuration



WebRTC Client

| | Source Address CIDR |
|---|---|
| – | all |

RTP Public and Local Port

| Public Address | Public Port | Local Address | Local Port |
|---|---|---|---|
| 81.144.171.73 | 16000 | 172.31.252.111 | 16000 |
| 81.144.171.73 | 16001 | 172.31.252.111 | 16001 |
| 81.144.171.73 | 16002 | 172.31.252.111 | 16002 |
| 81.144.171.73 | 16003 | 172.31.252.111 | 16003 |
| 81.144.171.73 | 16004 | 172.31.252.111 | 16004 |

Add Delete

| | Source Address CIDR |
|---|---|
| – | 172.31.253.0/24 |

RTP Public and Local Port

| Public Address | Public Port | Local Address | Local Port |
|---|---|---|---|
| 172.31.252.111 | 16000 | 172.31.252.111 | 16000 |
| 172.31.252.111 | 16001 | 172.31.252.111 | 16001 |
| 172.31.252.111 | 16002 | 172.31.252.111 | 16002 |
| 172.31.252.111 | 16003 | 172.31.252.111 | 16003 |
| 172.31.252.111 | 16004 | 172.31.252.111 | 16004 |

Add Delete

Add Delete

Above is an example configuration for 1 Media Broker with 5 rtp-proxy processes. It is intended to be used with Live Assist™; where a consumer's media is sent to a public IP address, but Agent media is sent to an internal interface.

It is possible to allocate the same local port and interface (eg: 172.31.252.111:16000) against each CIDR.

**Warning**

When modifications to the ports are made all rtp-proxy processes will restart to acquire their new config. This will terminate any calls in progress.

## Advanced Configuration

In case of a public Internet or ISP outage, some installations require that a Media Broker supports more than a single public Media Broker Address. This can be done by:

- Defining 2 (or more) Source Addresses for each ISP.
- Allocating disjoint sets of public addresses against each source address.

If an ISP becomes unavailable the Media Broker will stop receiving requests from the ISP; as a result the set of public addresses will never be allocated.

# Understanding Media Streams

Depending on your call setup numerous media streams will be established between clients, endpoints and the Media Broker.  If you are troubleshooting, it is often worth creating a diagram of your expectations so each stream can be identified in any packet captures.

While the above images and notes give a good indication of a normal call setup, each customer may have their own individual setups with small to significant differences.

The following are some common example scenarios:

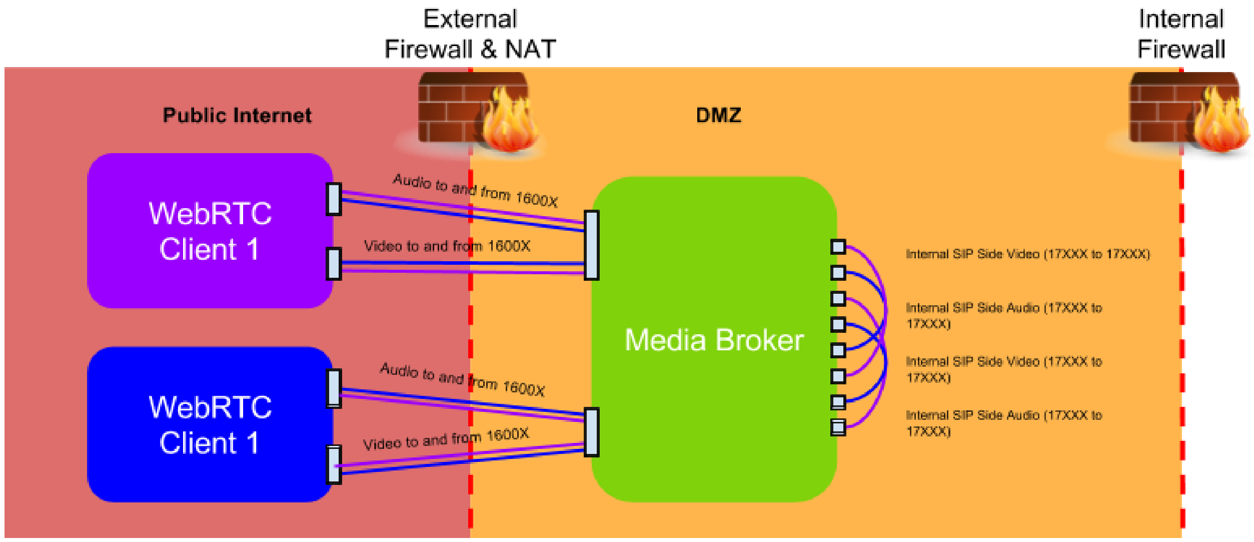
## WebRTC to WebRTC Calls

For **each Client** (WebRTC Side) you will see 8 Streams:

- One RTP stream for sending Video
- One RTP stream for receiving Video
- One RTP stream for sending Audio
- One RTP stream for receiving Audio

You will also see, on the sip side 4 Streams:

- Media Broker send itself one video stream per client (2 Streams total)
- Media Broker send itself one audio stream per client (2 Streams total)



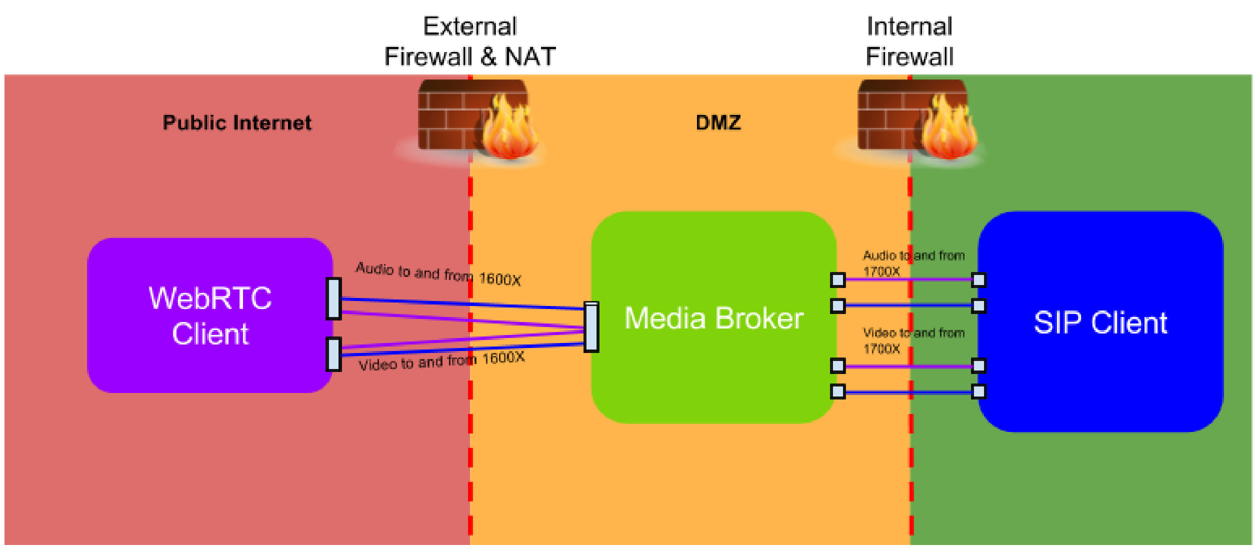
## WebRTC to SIP

For the WebRTC Client you will see 4 Streams:

- One RTP stream for sending Video
- One RTP stream for receiving Video
- One RTP stream for sending Audio
- One RTP stream for receiving Audio

For the SIP Client you will see 4 Streams:

- One RTP stream for sending Video
- One RTP stream for receiving Video
- One RTP stream for sending Audio
- One RTP stream for receiving Audio



# Calling an MCU

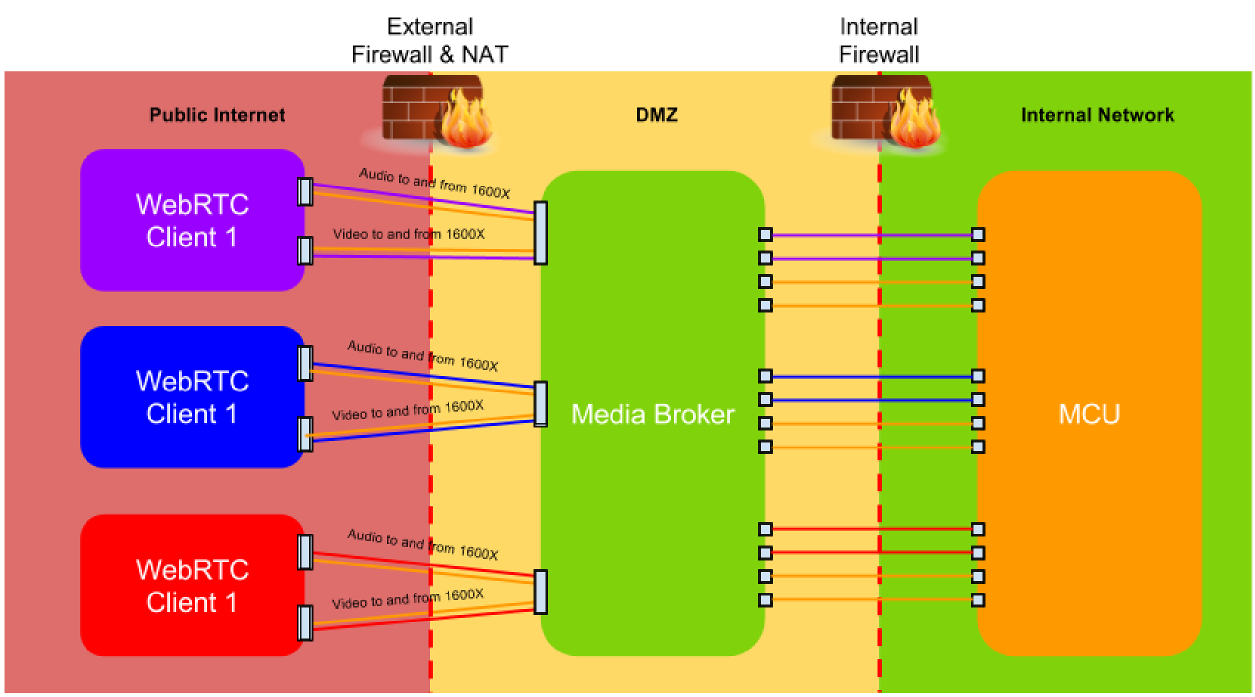
Three way calling is not overly common but does complicate the number of streams that will be seen.

For **each Client** (WebRTC Side) you will see 12 Streams:

- One RTP stream for sending Video
- One RTP stream for receiving Video
- One RTP stream for sending Audio
- One RTP stream for receiving Audio

You will also see, 12 streams on the sip side:

- Media Broker send the MCU one video stream per client (3 Streams total)
- Media Broker send the MCU one audio stream per client (3 Streams total)
- Media Broker receive from the MCU one video stream per client (3 Streams total)
- Media Broker receive from the MCU one audio stream per client (3 Streams total)

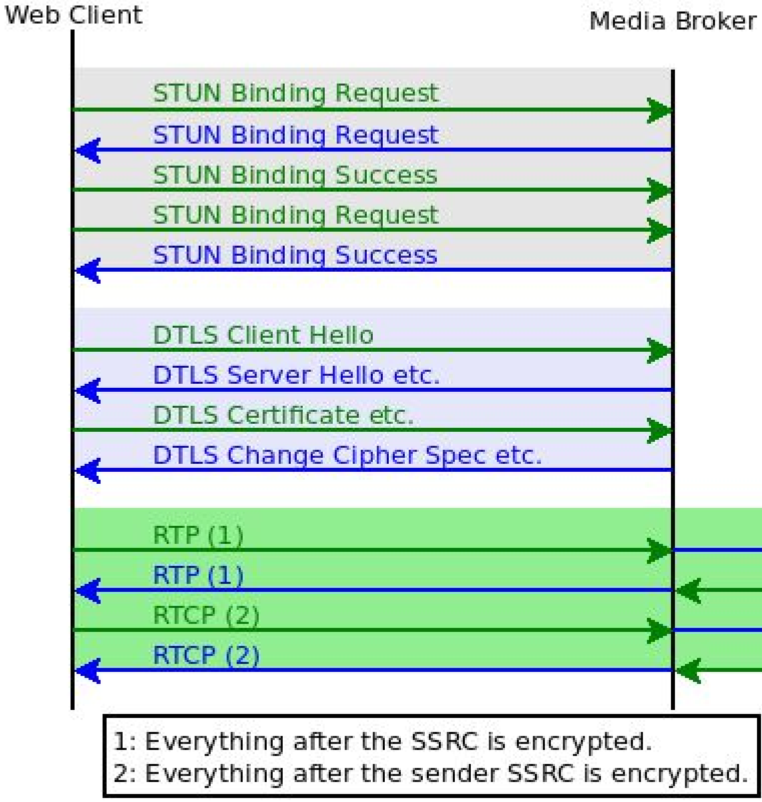


# Understanding Media Setup

The following diagram expands on the interaction between Web-RTC clients and the Media Broker during a call set up.

STUN Binding Requests are the first UDP messages sent between clients.

The Media Broker must be able to send UDP outbound towards all Web-RTC clients.




## STUN

STUN's primary purpose is to open paths through firewalls and authenticating source ports. Typically, there is a request and a corresponding success response from each client, resulting in a four packet exchange. If there is a failure at this stage, such a firewall block, no media will establish.

Once the WebRTC client has processed the answer SDP it will start sending STUN binding request packets to the public interface port (default: 16000) on MediaBroker. One exception is Firefox, where ICE is sent to the client for it to select a candidate.

Media Broker waits to receive a STUN request then generates a response, before sending its STUN requests back to the client. STUN will be seen throughout the call, it is used to check paths are media paths are still valid.  They are typically send every 0.5s from the majority of Google WebRTC clients.

Interactions with clients running the Google Chrome webRTC library behave a little differently.  Chrome clients only send a STUN request after it finishes sending a success response to a received request. As in the previous diagram, you may see 6 packets during the STUN setup.  If you’re analyzing the STUN, the second request and response pair from Chrome will take precedence and contain the media candidates.

Once STUN us successful the client and Media Broker can continue to establish the RTP stream.

## DTLS

Next DTLS exchanges happen in order to get keys for encrypting RTP. The client initiating the call takes on the role of DTLS client and sends a client hello to MB, which responds with a DTLS packet containing a server hello and a number of other details. The client then responds with a certificate and other information, to which MB then sends a change cipher specification and an encrypted handshake message. If there are packet losses then lost ones are automatically resent after delays specified in the DTLS RFC.

The diagram earlier shows the format you will see the DTLS packets, any deviation from that sequence (ignoring retransmissions) means something has gone wrong. A failure may result in an error packet, or it may fail silently. However a failure happens it is likely packets will continue to be resent without answer.

Often, there are two encrypted alerts sent at the end of the call, one from each side. They can be ignored, but can be a useful way of determining when one side thinks the call has ended.

Once STUN and DTLS are complete the media can begin to flow.

## RTP & RTCP

While the webRTC client is establishing a media path, the SIP client is sending RTP and RTCP to the MB to the internal ports (default: 17000+). These packets will be discarded by Media Broker because it could not send them to the client until the DTLS handshake completes. Once complete, the Media Broker can start to send received media in both directions.

For a given call leg, there are four ports allocated for SIP side media on Media Broker. The even numbered ports being used to receive RTP audio and video, and the port one higher for the corresponding RTCP. The Media Broker's webrtc side only uses one port and uses the SSRCs to distinguish the audio from the video media.

**Note**: The ports in the SDP of SIP clients are the ports a client wants to receive media on. It can send media from a different port, though this rarely happens.

Media on the web side is encrypted, without decrypting a PCAP, only the main RTP header is readable. For an RTCP packet, only the headers up to and including the first SSRC header is readable. The encryption also adds 10 (RTP) or 20 (RTCP) bytes at the end of the packet.

**Note**:Extended headers exist but you will probably never see them, but special codecs use similar extensions for various details which will be encrypted.

When MB receives packets from the web side they are held in the Media Broker before sending out on the SIP side in a jitter buffer. The jitter buffer's purpose is to reduce jitter. The buffer is initially ~300ms, but will increase if network conditions are poor, giving more time for packets to reorder before processing and sending downstream. In passthrough calls, packets received from the SIP have SSRCs, timestamps and sequence numbers changed in RTP packets so the client only ever sees one continuous stream when being transferred. The RTCP packets are effectively discarded and created within MB, although PLIs will result in immediate creation of a PLI to send to the client.

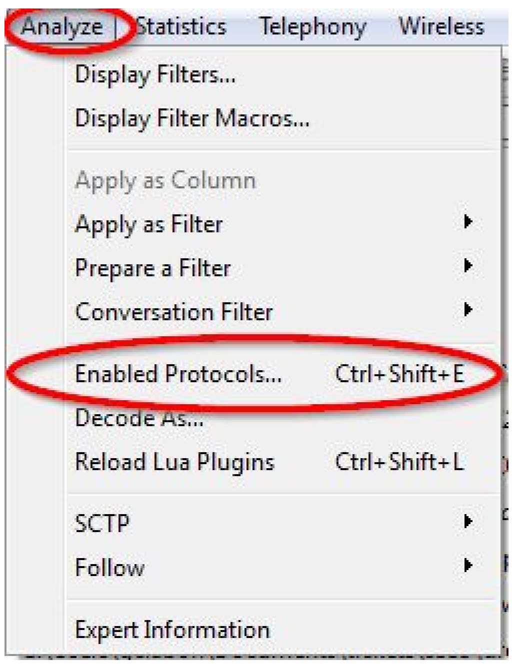
Some web client support the RED codec to send most RTP packets for the video stream which wraps the original packet. If you need to tell what packets are inside, to determine which codec was picked, or the content of ULPFEC, you will need to decrypt the pcap.

When either the webRTC client or the SIP side end the call, the Gateway sends an HTTP DELETE to Media Broker, which then tears the call down down. You may see some ICMP destination unreachable messages from one endpoint briefly, ignore these. Media broker will respond with details of the call's statistics. At the same time the Gateway sends a SIP BYE or an end message to the client as necessary.

# Setting Up Wireshark for Analyzing RTP Streams

## Enable Automatic Decoding of RTP Streams

To reduce the amount of streams you will have to manually decode, on the toolbar go to *Analyze | Enabled Protocols…*

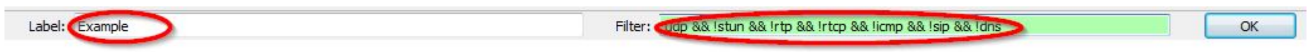



Scroll down the list to find *RTP* and check *rtp_udp* to automatically decode RTP over UDP

## Save Useful Filters

Saving filters can save you having to manually type in useful filters each time you are analyzing a pcap. Click the small **+** on the right hand side of the filter bar:


Enter a label describing the filter, and the filter required then click OK:




You will now have a button on the right hand side of the filter bar that will automatically apply the filter you have save

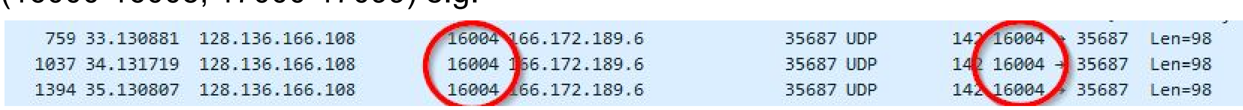

## Initial PCAP Analysis

**1.** First step is to decode the rtp streams for analysis. Open the pcap in wireshark, filter on:

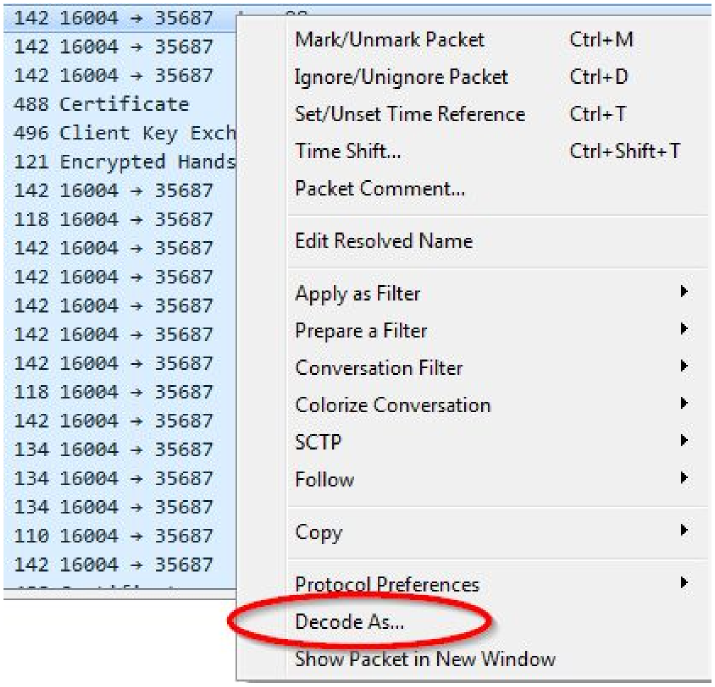
*udp && !stun && !rtp && !rtcp && !icmp && !sip && !dns*

This shows any UDP packets that are not stun, rtp, rtcp, icmp sip or dns.

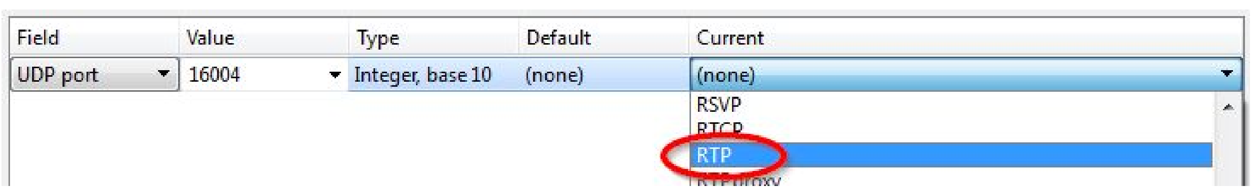
**2.** Now look for any packets going from or to a known Media Broker media port (16000-16005, 17000-17099) e.g.




**3.** Right click the packet and select *Decode As…*



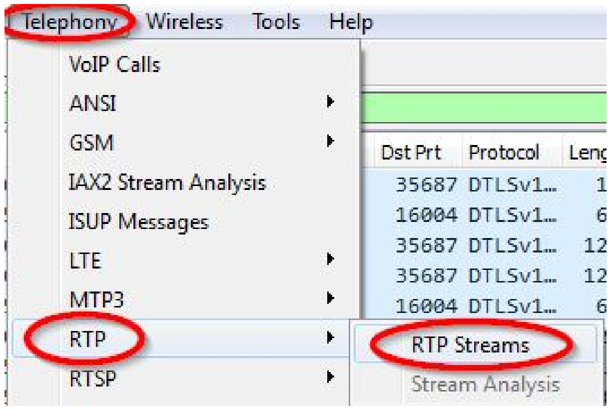

**4.** In the next window set Current to RTP and click ok




**5.** Repeat these steps until no more packets can be seen from/to the known Media Broker ports.

**6.** Clear your filters by clicking the cross:

**7.** Now go to the Telephony menu and select RTP > RTP Stream

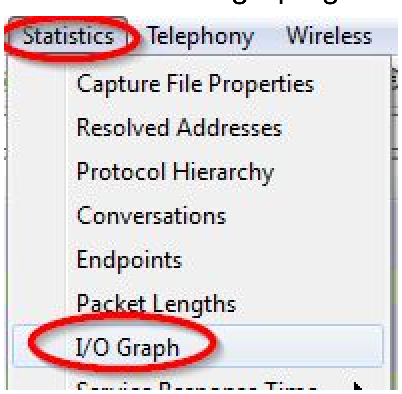



**8.** You will now see all of the RTP streams going to or from the Media Broker (it may be useful to order these by Payload as below).

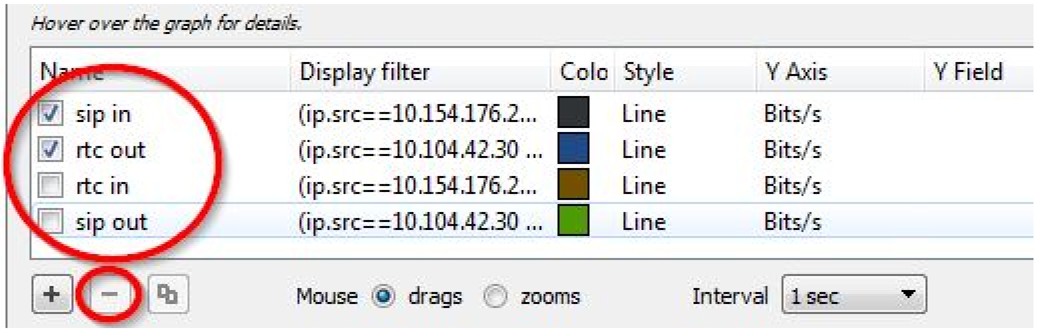
| Source Address | Source Port | Destination Address | Destination Port | SSRC | Payload | Packets | Lost | Max Delta (ms) | Max Jitter | Mean Jitter |
|---|---|---|---|---|---|---|---|---|---|---|
| 192.168.9.18 | 16001 | 192.168.30.84 | 58150 | 0x73461d14 | RTPType-97 | 146 | 0 (0.0%) | 0.000 | 0.000 | 0.000 |
| 192.168.9.18 | 17088 | 192.168.30.84 | 25426 | 0xdb93dd8c | H264 | 235 | 0 (0.0%) | 299.823 | 8.707 | 3.134 |
| 192.168.30.84 | 58150 | 192.168.9.18 | 16001 | 0xdb93dd8c | RTPType-97 | 250 | 0 (0.0%) | 0.000 | 0.000 | 0.000 |
| 192.168.30.84 | 25426 | 192.168.9.18 | 17088 | 0xfed43bcf | H264 | 154 | 0 (0.0%) | 30.418 | 3.149 | 0.990 |
| 192.168.9.18 | 16001 | 192.168.30.84 | 58150 | 0xee4aff77 | RTPType-109 | 288 | 0 (0.0%) | 0.000 | 0.000 | 0.000 |
| 192.168.30.84 | 20788 | 192.168.9.18 | 17040 | 0x1ffbec9c | opus | 357 | 0 (0.0%) | 23.902 | 0.850 | 0.419 |
| 192.168.30.84 | 58150 | 192.168.9.18 | 16001 | 0xc621d975 | RTPType-109 | 288 | 0 (0.0%) | 0.000 | 0.000 | 0.000 |
| 192.168.9.18 | 17040 | 192.168.30.84 | 20788 | 0xc621d975 | opus | 273 | 0 (0.0%) | 43.820 | 3.449 | 0.544 |

**9.** The Above RTP streams show a working call, with audio and video in both directions (4 video streams, 4 audio streams). These can be used to draw an I/O graph showing the bitrates.

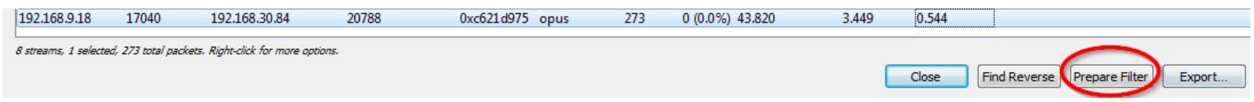
**10.** To draw the graph go to Statistics > I/O Graph



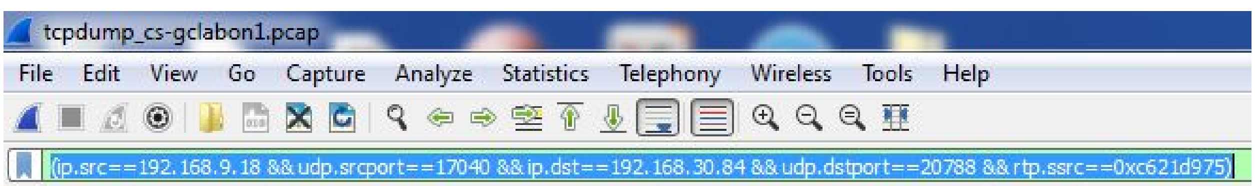
**11.** Remove any existing filters by selecting the filter and clicking the minus::



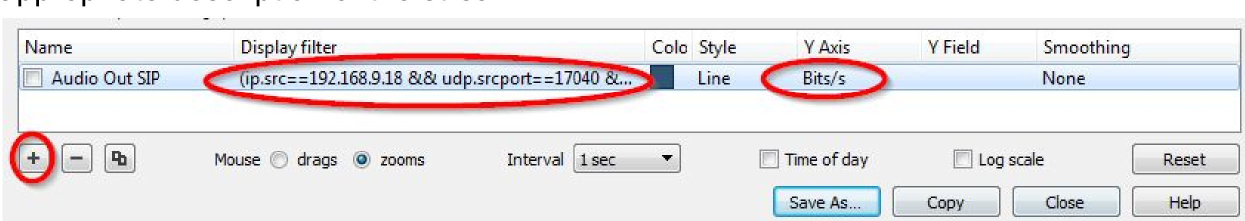

**12.** On the RTP Streams window highlight one stream (you can highlight more than one at a time so make sure!) and click prepare filter:




**13.** Copy the filter from the main WireShark window:

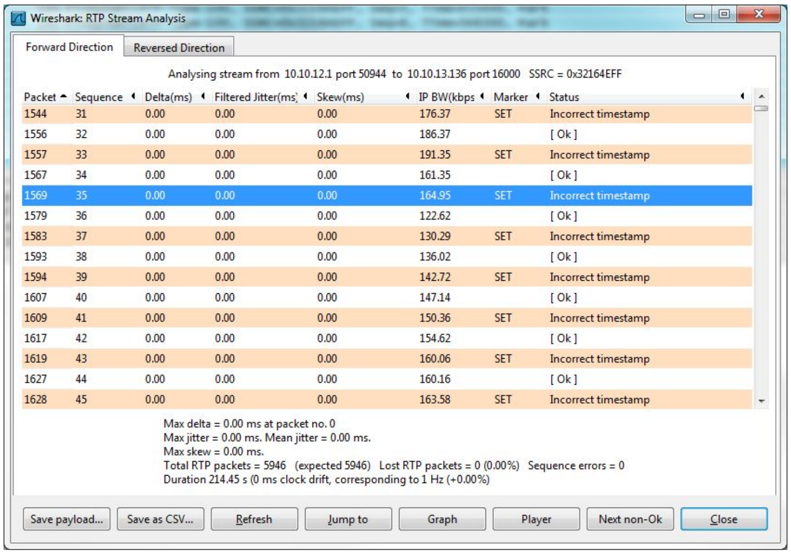



**14.** On the I/O Graph window add a new filter using the + button. Paste the copied filter into the display filter area and set the Y Axis to Bits/s. Change the Name to an appropriate description of the stream

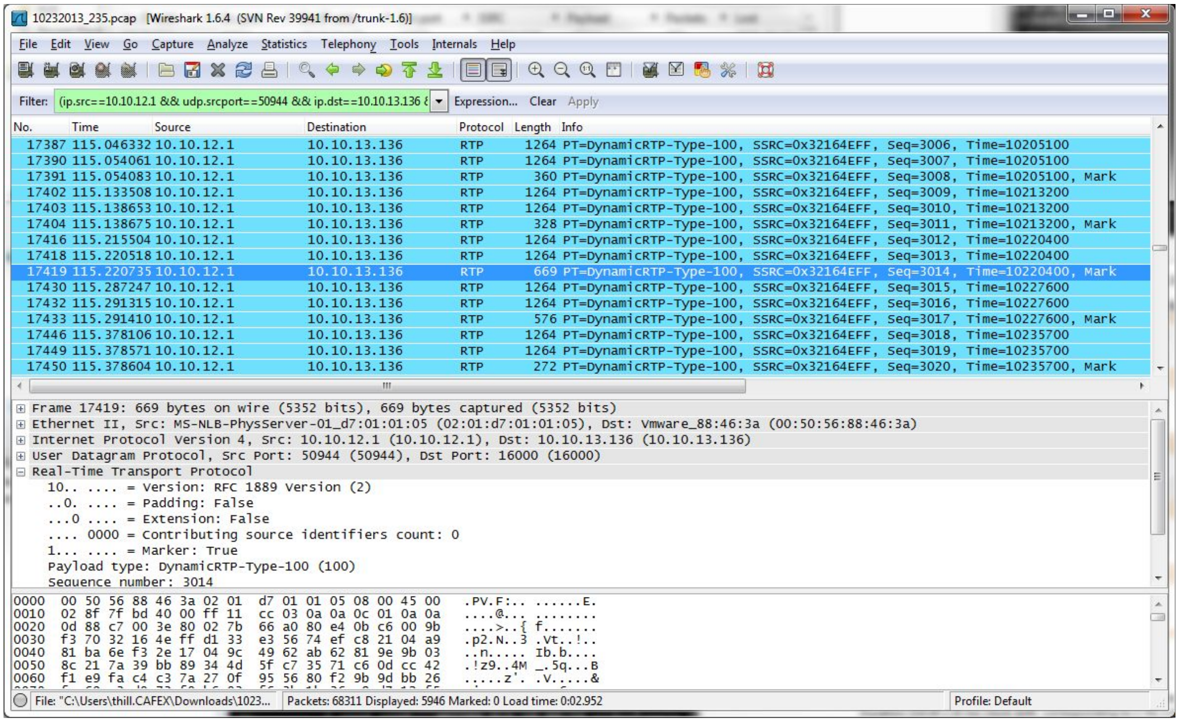


## Stream Analysis has "Timestamp incorrect"

It is possible to see many "Incorrect timestamp" messages when performing a Stream Analysis on a Video Stream:




Video Streams use the "Mark" attribute to indicate when a Frame ends. In a video stream, groups of packets with the same timestamp indicate that the packets belong to the same Frame. In the example below there is a sequence of 3 packets with a timestamp *10220400* and the last packet has the "Mark" attribute set. It is safe to assume that the three packets belong to the same frame.



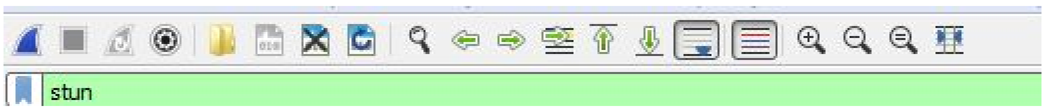

Typically, audio codecs indicate the start of a 'audio-burst' with a "Mark" attribute. The last packet of the video frame is "Marked" and wireshark has assumed that this packet is the beginning of an audio-burst. Wireshark assumes that the timestamp is incorrect because the last packet has the same timestamp; it is not possible to start a new audio-burst with a previously used timestamp.

Essentially, these warnings can be ignored on video streams.

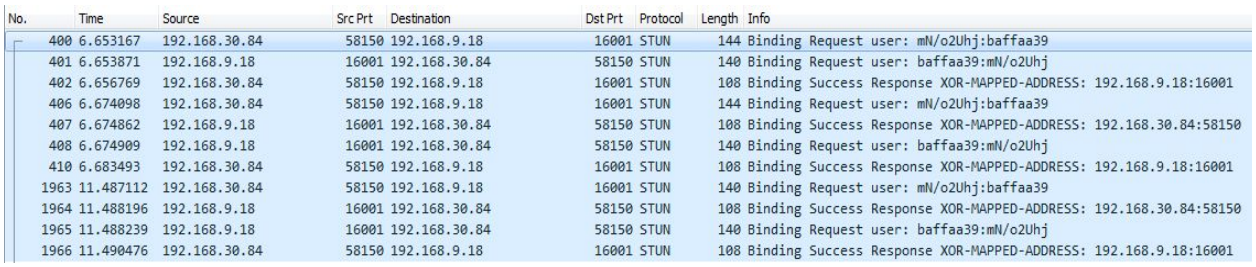
# Is STUN Working?

No STUN = No Media

If STUN is working correctly this rules out an issue with the media path on the WebRTC side. This can be checked inside the pcap. To start open the pcap file and set the filer to *stun*




For each WebRTC Client you should then see the following:



| No. | Time | Source | Src Prt | Destination | Dst Prt | Protocol | Length | Info |
|---|---|---|---|---|---|---|---|---|
| 400 | 6.653167 | 192.168.30.84 | 58150 | 192.168.9.18 | 16001 | STUN | 144 | Binding Request user: mN/o2Uhj:baffaa39 |
| 401 | 6.653871 | 192.168.9.18 | 16001 | 192.168.30.84 | 58150 | STUN | 140 | Binding Request user: baffaa39:mN/o2Uhj |
| 402 | 6.656769 | 192.168.30.84 | 58150 | 192.168.9.18 | 16001 | STUN | 108 | Binding Success Response XOR-MAPPED-ADDRESS: 192.168.9.18:16001 |
| 406 | 6.674098 | 192.168.30.84 | 58150 | 192.168.9.18 | 16001 | STUN | 144 | Binding Request user: mN/o2Uhj:baffaa39 |
| 407 | 6.674862 | 192.168.9.18 | 16001 | 192.168.30.84 | 58150 | STUN | 108 | Binding Success Response XOR-MAPPED-ADDRESS: 192.168.30.84:58150 |
| 408 | 6.674909 | 192.168.9.18 | 16001 | 192.168.30.84 | 58150 | STUN | 140 | Binding Request user: baffaa39:mN/o2Uhj |
| 410 | 6.683493 | 192.168.30.84 | 58150 | 192.168.9.18 | 16001 | STUN | 108 | Binding Success Response XOR-MAPPED-ADDRESS: 192.168.9.18:16001 |
| 1963 | 11.487112 | 192.168.30.84 | 58150 | 192.168.9.18 | 16001 | STUN | 140 | Binding Request user: mN/o2Uhj:baffaa39 |
| 1964 | 11.488196 | 192.168.9.18 | 16001 | 192.168.30.84 | 58150 | STUN | 108 | Binding Success Response XOR-MAPPED-ADDRESS: 192.168.30.84:58150 |
| 1965 | 11.488239 | 192.168.9.18 | 16001 | 192.168.30.84 | 58150 | STUN | 140 | Binding Request user: baffaa39:mN/o2Uhj |
| 1966 | 11.490476 | 192.168.30.84 | 58150 | 192.168.9.18 | 16001 | STUN | 108 | Binding Success Response XOR-MAPPED-ADDRESS: 192.168.9.18:16001 |

Where more than one WebRTC Client is being used you may need to be more specific with filtering, by setting the source and destination port specific to each client:

*stun && (udp.srcport == 58150 || udp.dstport == 58150)*

Where 51850 is the udp port of the current client

## No STUN in the pcap

A failed STUN setup may return no results, in this case either:

1. Media Broker has not received the STUN Request
2. Media Broker is not listening for STUN Requests

To resolve this you will need to ensure that

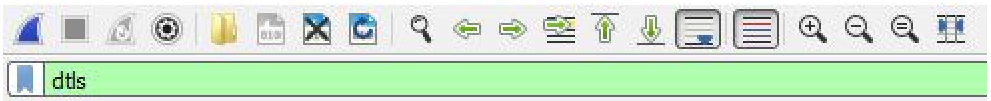
1. Media broker service is started and listening correctly
2. The Media Broker configuration is correct, specifically the Public and Local IPs/Ports
3. Firewalls on the media path have the correct ports opened and forwarding

4. A local firewall is not preventing communication (**See**: Local Firewall Configuration)
5. selinux is disabled

**Important:** Media Broker needs to send outbound STUN towards web clients.
See: **Testing the Local Firewall Ports**

# The DTLS Handshake

As part of setting up the media paths there is a DTLS handshake. If this DTLS Handshake fails, for whatever reason, media on the WebRTC side will not work. To check this open the pcap file and set the filer to *dtls*




Where more than one WebRTC Client is being used you may need to be more specific with filtering, by setting the source and destination port specific to each client, e.g.

*dtls && (udp.srcport == 58150 || udp.dstport == 58150)*

**Note:**Where 51850 is the udp port of the current client

## Working DTLS Handshake

An example of a working DTLS handshake is:

| No. | Time | Source | Src Prt | Destination | Dst Prt | Protocol | Length | Info |
|---|---|---|---|---|---|---|---|---|
| 403 | 6.658606 | 192.168.9.18 | 16001 | 192.168.30.84 | 58150 | DTLSv1… | 197 | Client Hello |
| 493 | 7.658398 | 192.168.9.18 | 16001 | 192.168.30.84 | 58150 | DTLSv1… | 197 | Client Hello |
| 495 | 7.664068 | 192.168.30.84 | 58150 | 192.168.9.18 | 16001 | DTLSv1… | 717 | Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello Done |
| 497 | 7.673989 | 192.168.9.18 | 16001 | 192.168.30.84 | 58150 | DTLSv1… | 614 | Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Handshake Me… |
| 499 | 7.686059 | 192.168.30.84 | 58150 | 192.168.9.18 | 16001 | DTLSv1… | 119 | Change Cipher Spec, Encrypted Handshake Message |
| 2757 | 13.464151 | 192.168.9.18 | 16001 | 192.168.30.84 | 58150 | DTLSv1… | 83 | Encrypted Alert |

## Failed DTLS Handshakes

Examples of a failed DTLS handshake are below. This particular failure was seen, when a very old version of FCSDK was used with a newer browsers (Chrome 56+, FF 51+).

| No. | Time | Source | Src Prt | Destination | Dst Prt | Protocol | Length | Info |
|---|---|---|---|---|---|---|---|---|
| 1308 | 6.258009 | 159.45.93.148 | 16000 | 50.150.80.192 | 59679 | DTLSv1… | 197 | Client Hello |
| 1362 | 6.378998 | 50.150.80.192 | 59679 | 159.45.93.148 | 16000 | DTLSv1… | 717 | Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello Done |
| 1364 | 6.381839 | 159.45.93.148 | 16000 | 50.150.80.192 | 59679 | DTLSv1… | 59 | Alert (Level: Fatal, Description: Internal Error) |
| 1385 | 6.429384 | 50.150.80.192 | 59679 | 159.45.93.148 | 16000 | DTLSv1… | 717 | Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello Done |

Or you may see the handshake enter a loop:

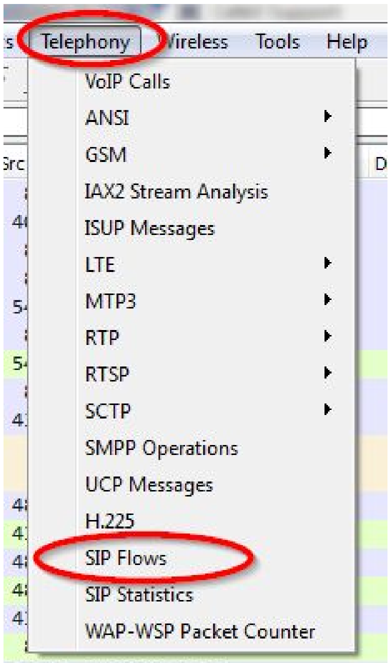
| No. | Time | Source | Port | Destination | Port | Protocol | Length | Info |
|---|---|---|---|---|---|---|---|---|
| 76788 | 264.349355 | 0.0.0.0 | 68 | 255.255.255.255 | 67 | DHCP | 344 | DHCP Discover - Transaction ID 0x15ea92e5 |
| 2068 | 4.723978 | 108.163.1.108 | 16003 | 66.87.149.0 | 11774 | DTLSv1.0 | 105 | Encrypted Alert |
| 2599 | 6.597147 | 108.163.1.108 | 16000 | 108.33.234.44 | 65265 | DTLSv1.0 | 105 | Encrypted Alert |
| 2880 | 12.325477 | 108.163.1.108 | 16004 | 66.87.149.0 | 26240 | DTLSv1.0 | 193 | Client Hello |
| 2888 | 12.412172 | 108.33.234.44 | 51440 | 108.163.1.108 | 16002 | DTLSv1.0 | 145 | Client Hello |
| 2890 | 12.412834 | 108.163.1.108 | 16002 | 108.33.234.44 | 51440 | DTLSv1.0 | 880 | Server Hello, Certificate, Certificate Request, Server Hello Done |
| 2909 | 12.458181 | 108.33.234.44 | 51440 | 108.163.1.108 | 16002 | DTLSv1.0 | 1018 | Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Client Hello |
| 2910 | 12.468321 | 108.163.1.108 | 16002 | 108.33.234.44 | 51440 | DTLSv1.0 | 135 | Change Cipher Spec, Encrypted Handshake Message |
| 3046 | 13.325625 | 108.163.1.108 | 16004 | 66.87.149.0 | 26240 | DTLSv1.0 | 193 | Client Hello |
| 3075 | 13.485284 | 66.87.149.0 | 26240 | 108.163.1.108 | 16004 | DTLSv1.0 | 656 | Server Hello, Certificate, Certificate Request, Server Hello Done |
| 3082 | 13.495554 | 108.163.1.108 | 16004 | 66.87.149.0 | 26240 | DTLSv1.0 | 1276 | Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Har |
| 3096 | 13.572709 | 66.87.149.0 | 26240 | 108.163.1.108 | 16004 | DTLSv1.0 | 135 | Change Cipher Spec, Encrypted Handshake Message |
| 33587 | 52.710787 | 108.163.1.108 | 16004 | 66.87.149.0 | 26240 | DTLSv1.0 | 105 | Encrypted Alert |
| 33594 | 52.713767 | 108.163.1.108 | 16002 | 108.33.234.44 | 51440 | DTLSv1.0 | 105 | Encrypted Alert |
| 34499 | 97.401060 | 108.33.234.44 | 61487 | 108.163.1.108 | 16002 | DTLSv1.0 | 145 | Client Hello |
| 34500 | 97.401599 | 108.163.1.108 | 16002 | 108.33.234.44 | 61487 | DTLSv1.0 | 880 | Server Hello, Certificate, Certificate Request, Server Hello Done |
| 34503 | 97.441023 | 108.33.234.44 | 61487 | 108.163.1.108 | 16002 | DTLSv1.0 | 1018 | Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Har |
| 34504 | 97.450781 | 108.163.1.108 | 16002 | 108.33.234.44 | 61487 | DTLSv1.0 | 135 | Change Cipher Spec, Encrypted Handshake Message |
| 34521 | 97.743235 | 108.163.1.108 | 16004 | 108.33.234.44 | 55099 | DTLSv1.0 | 193 | Client Hello |
| 34787 | 98.743364 | 108.163.1.108 | 16004 | 108.33.234.44 | 55099 | DTLSv1.0 | 193 | Client Hello |
| 34835 | 98.937842 | 108.33.234.44 | 55099 | 108.163.1.108 | 16004 | DTLSv1.0 | 655 | Server Hello, Certificate, Certificate Request, Server Hello Done |
| 34840 | 98.948719 | 108.163.1.108 | 16004 | 108.33.234.44 | 55099 | DTLSv1.0 | 1276 | Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Har |
| 34882 | 99.109257 | 108.33.234.44 | 55099 | 108.163.1.108 | 16004 | DTLSv1.0 | 135 | Change Cipher Spec, Encrypted Handshake Message |
| 58839 | 130.264264 | 108.163.1.108 | 16002 | 108.33.234.44 | 61487 | DTLSv1.0 | 105 | Encrypted Alert |
| 58850 | 130.272303 | 108.163.1.108 | 16004 | 108.33.234.44 | 55099 | DTLSv1.0 | 105 | Encrypted Alert |
| 60098 | 207.293847 | 47.199.169.238 | 55517 | 108.163.1.108 | 16002 | DTLSv1.0 | 145 | Client Hello |
| 60099 | 207.294401 | 108.163.1.108 | 16002 | 47.199.169.238 | 55517 | DTLSv1.0 | 880 | Server Hello, Certificate, Certificate Request, Server Hello Done |
| 60101 | 207.345540 | 47.199.169.238 | 55517 | 108.163.1.108 | 16002 | DTLSv1.0 | 1018 | Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Har |
| 60137 | 208.032061 | 108.163.1.108 | 16004 | 166.172.190.55 | 3025 | DTLSv1.0 | 193 | Client Hello |
| 60160 | 208.359154 | 47.199.169.238 | 55517 | 108.163.1.108 | 16002 | DTLSv1.0 | 1018 | Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Har |
| 60183 | 209.032194 | 108.163.1.108 | 16004 | 166.172.190.55 | 3025 | DTLSv1.0 | 193 | Client Hello |
| 60185 | 209.104911 | 166.172.190.55 | 3025 | 108.163.1.108 | 16004 | DTLSv1.0 | 655 | Server Hello, Certificate, Certificate Request, Server Hello Done |
| 60186 | 209.114987 | 108.163.1.108 | 16004 | 166.172.190.55 | 3025 | DTLSv1.0 | 1276 | Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Har |
| 60188 | 209.192218 | 166.172.190.55 | 3025 | 108.163.1.108 | 16004 | DTLSv1.0 | 135 | Change Cipher Spec, Encrypted Handshake Message |
| 60456 | 210.354276 | 47.199.169.238 | 55517 | 108.163.1.108 | 16002 | DTLSv1.0 | 1018 | Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Har |
| 61925 | 214.359127 | 47.199.169.238 | 55517 | 108.163.1.108 | 16002 | DTLSv1.0 | 489 | Certificate |
| 61926 | 214.359247 | 47.199.169.238 | 55517 | 108.163.1.108 | 16002 | DTLSv1.0 | 496 | Client Key Exchange, Certificate Verify, Change Cipher Spec |
| 61927 | 214.359299 | 47.199.169.238 | 55517 | 108.163.1.108 | 16002 | DTLSv1.0 | 121 | Encrypted Handshake Message |
| 65004 | 222.361756 | 47.199.169.238 | 55517 | 108.163.1.108 | 16002 | DTLSv1.0 | 489 | Certificate |
| 65005 | 222.361871 | 47.199.169.238 | 55517 | 108.163.1.108 | 16002 | DTLSv1.0 | 496 | Client Key Exchange, Certificate Verify, Change Cipher Spec |
| 65006 | 222.361878 | 47.199.169.238 | 55517 | 108.163.1.108 | 16002 | DTLSv1.0 | 121 | Encrypted Handshake Message |
| 71623 | 238.364556 | 47.199.169.238 | 55517 | 108.163.1.108 | 16002 | DTLSv1.0 | 489 | Certificate |
| 71624 | 238.364713 | 47.199.169.238 | 55517 | 108.163.1.108 | 16002 | DTLSv1.0 | 496 | Client Key Exchange, Certificate Verify, Change Cipher Spec |
| 71625 | 238.364734 | 47.199.169.238 | 55517 | 108.163.1.108 | 16002 | DTLSv1.0 | 121 | Encrypted Handshake Message |
| 76471 | 249.844952 | 108.163.1.108 | 16004 | 166.172.190.55 | 3025 | DTLSv1.0 | 105 | Encrypted Alert |
| 76479 | 249.851525 | 108.163.1.108 | 16002 | 47.199.169.238 | 55517 | DTLSv1.0 | 105 | Encrypted Alert |
| 51 | 0.119110 | 127.0.0.1 | 41120 | 127.0.0.1 | 37085 | HTTP | 190 | GET /statistics HTTP/1.1 |
| 52 | 0.119378 | 127.0.0.1 | 37085 | 127.0.0.1 | 41120 | HTTP | 794 | HTTP/1.1 200 OK (text/plain) |

# Drawing the Call Flow Sequence

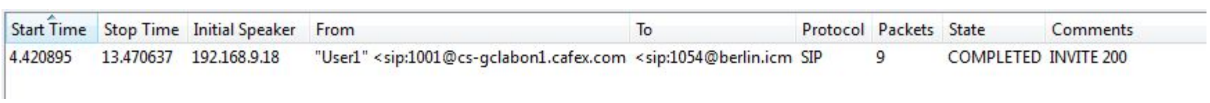
Drawing out the call flow sequence can be useful when analysing complicated calls. In the call flow you can record the relevant pieces of the SIP message and SDP.

## SIP Side with Wireshark (from FAS)

The SIP call flow must be taken from the FAS. The Media Broker pcaps do not contain SIP messages. However single box installs will contain both RTP and SIP dialogs.
Open the pcap in wireshark and decode all the rtp streams.
Then on the toolbar select *Telephony | Sip Flows:*




You will need to find the correct call. Complete calls that were ended will show a state of COMPLETED:



| Start Time | Stop Time | Initial Speaker | From | To | Protocol | Packets | State | Comments |
|---|---|---|---|---|---|---|---|---|
| 4.420895 | 13.470637 | 192.168.9.18 | "User1" <sip:1001@cs-gclabon1.cafex.com | <sip:1054@berlin.icm | SIP | 9 | COMPLETED | INVITE 200 |

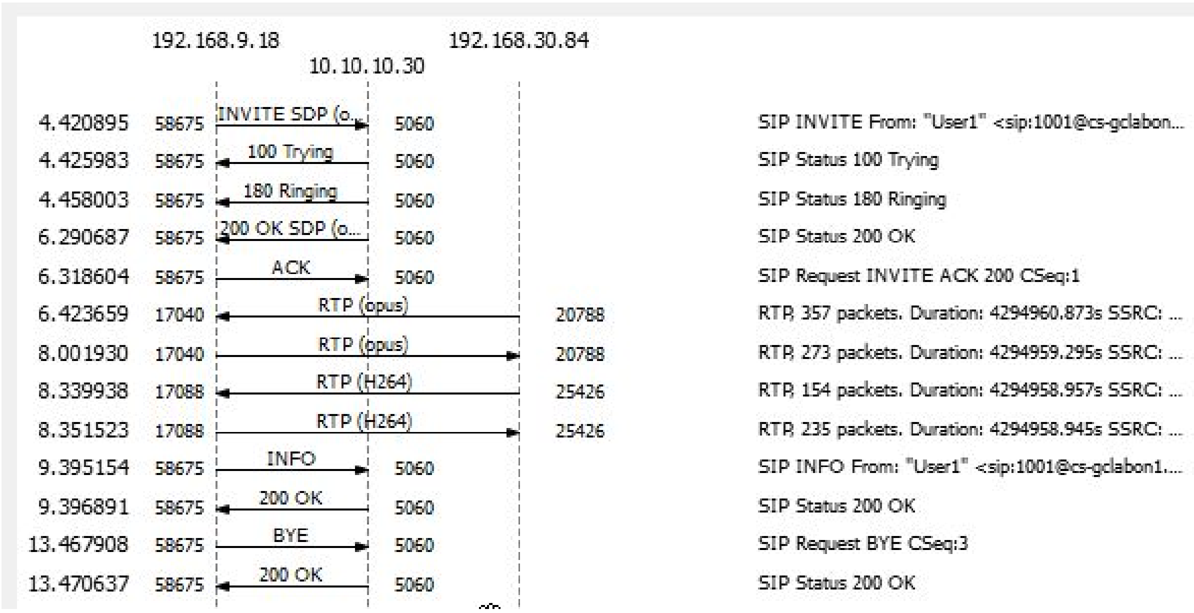
Highlight the appropriate call then click *Flow Sequence* in the bottom right corner:


This will draw the complete call flow as seen from FAS.

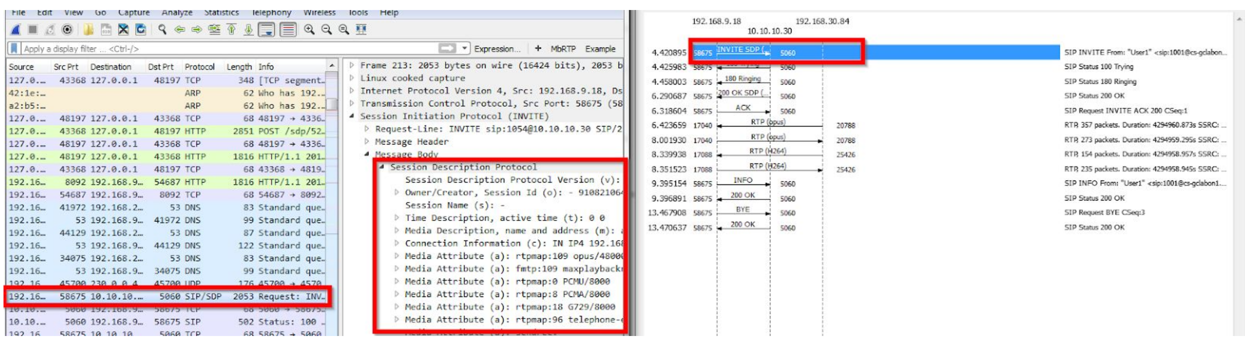
The image below is taken from a single box install so shows both SIP messages and the RTP streams, note that the SIP messages are between 192.168.9.18 (FAS) and 10.10.10.30 (a PBX).

**Note:**

- You cannot see what PBX has sent to a SIP client side from here.
- Nor can you see the SDP sent to the webRTC client




As most installs are not single box you will need to click the relevant SIP message, this will move wireshark's display to the correct packet, where you can view the SIP SDP (SIP INVITE in the case below:



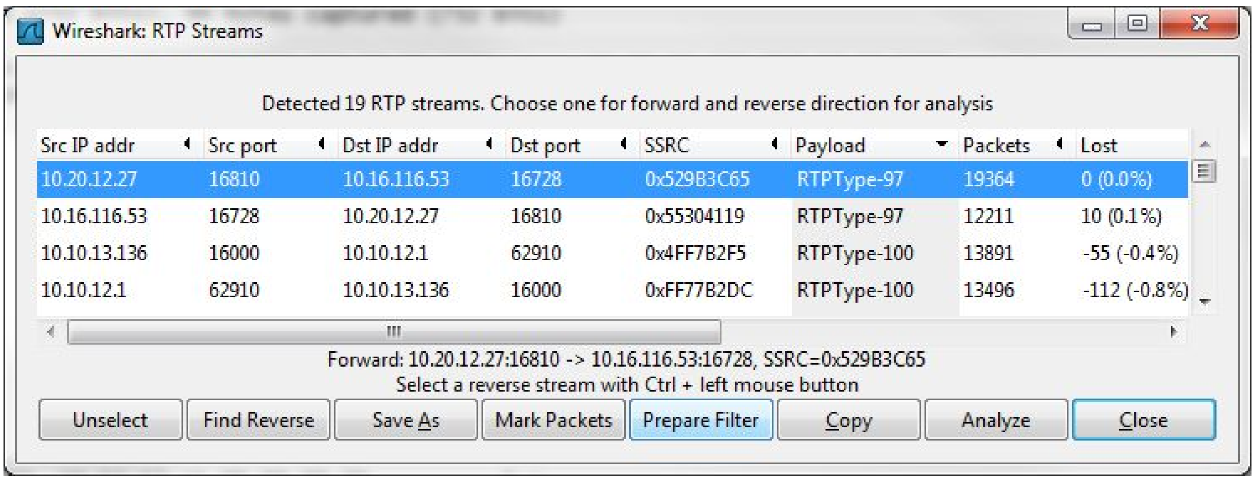
When you are more familiar with the call flow, you can read the inbound Offers and Answers reading the Media Broker sdp.log

# Advanced Packet Capture Analysis

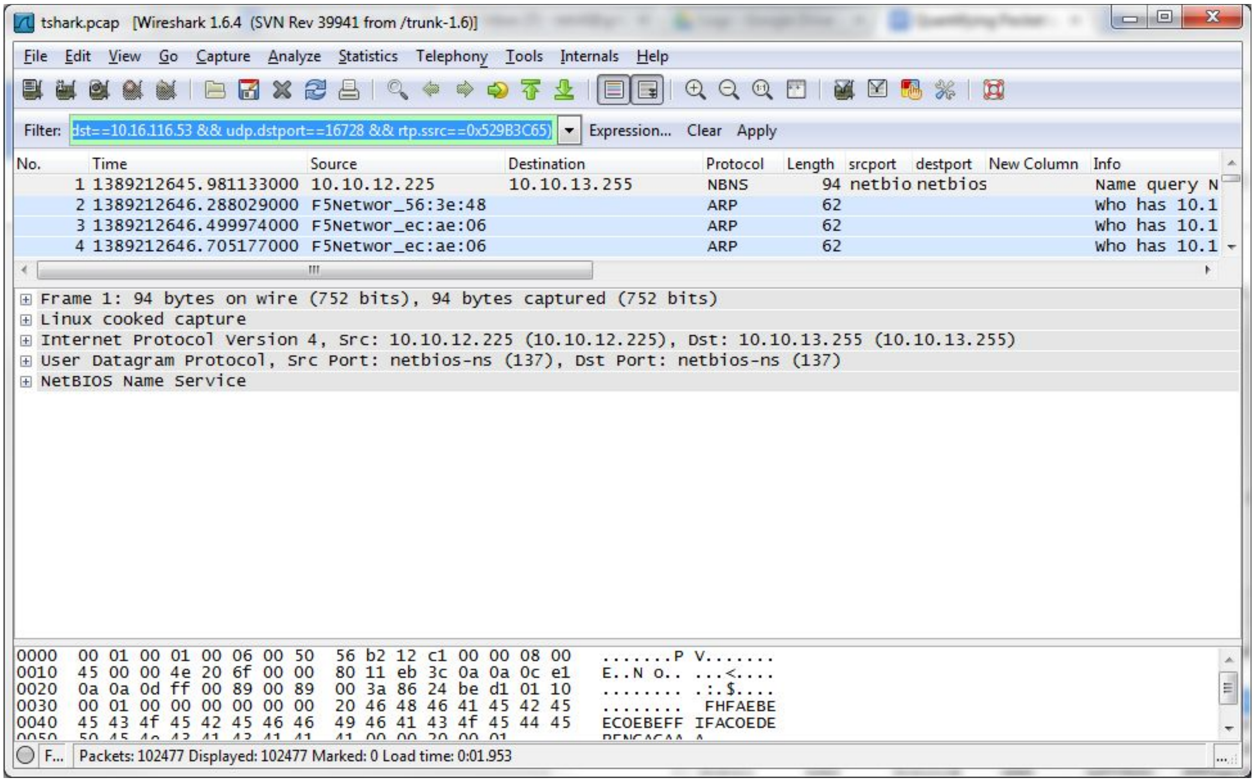
## Measuring Bandwidth

Currently, the easiest way to measure actual Bandwidth usage is to use Wireshark's **IO Graph** which can be found under *Statistics*.

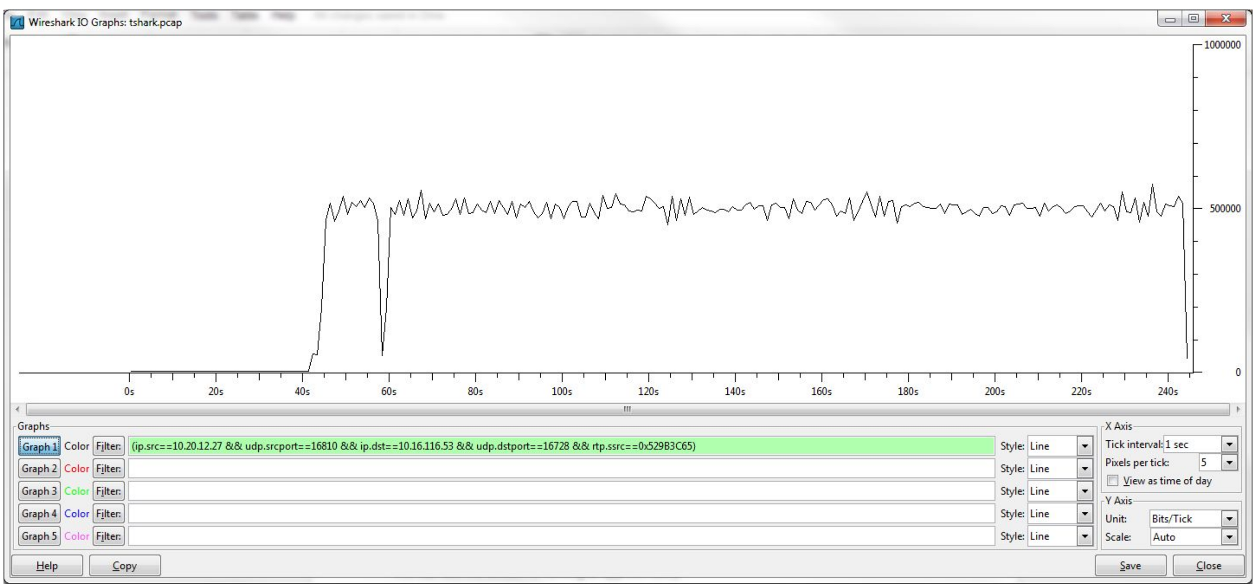
1) Find the appropriate Filter for a stream:




2) Copy the Filter from the Main Wireshark window:

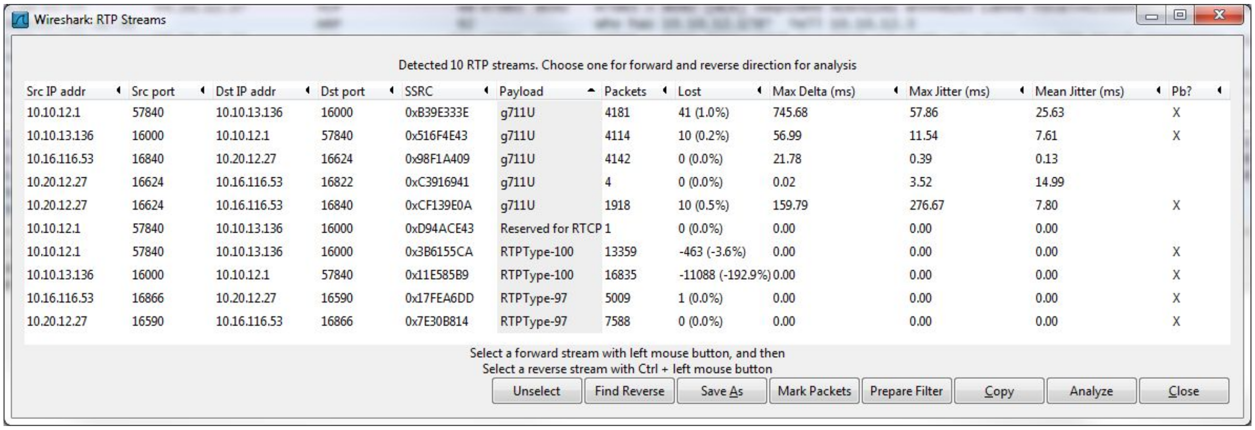



3) Paste the Filter into the IO Graph and set the Y Axis to *Bits per Second*




We can see the stream is running at approximately 500kbits$^{-1}$

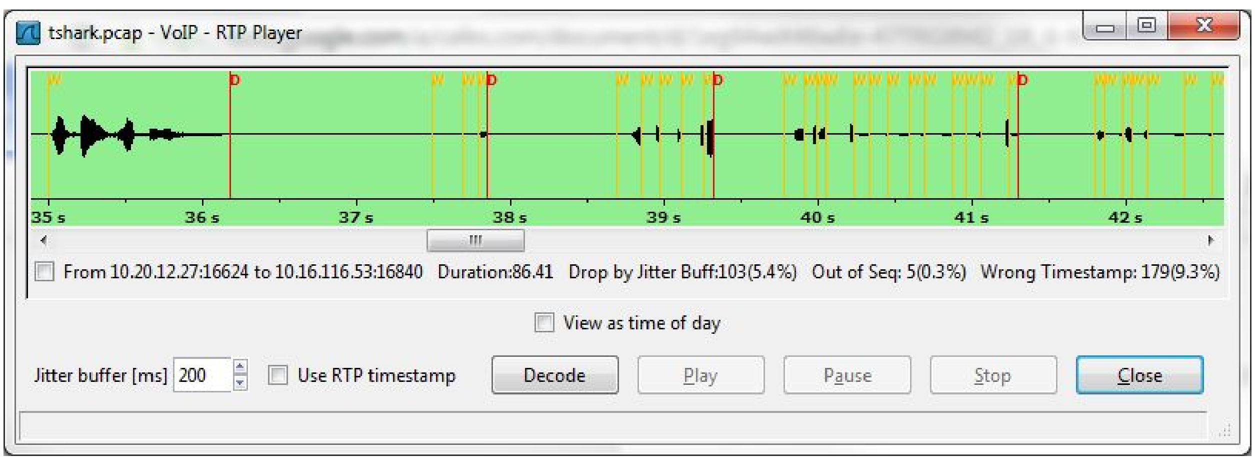
## Analysis of an SIP audio stream




| Src IP addr | Src port | Dst IP addr | Dst port | SSRC | Payload | Packets | Lost | Max Delta (ms) | Max Jitter (ms) | Mean Jitter (ms) | Pb? |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 10.10.12.1 | 57840 | 10.10.13.136 | 16000 | 0xB39E333E | g711U | 4181 | 41 (1.0%) | 745.68 | 57.86 | 25.63 | X |
| 10.10.13.136 | 16000 | 10.10.12.1 | 57840 | 0x516F4E43 | g711U | 4114 | 10 (0.2%) | 56.99 | 11.54 | 7.61 | X |
| 10.16.116.53 | 16840 | 10.20.12.27 | 16624 | 0x98F1A409 | g711U | 4142 | 0 (0.0%) | 21.78 | 0.39 | 0.13 | |
| 10.20.12.27 | 16624 | 10.16.116.53 | 16822 | 0xC3916941 | g711U | 4 | 0 (0.0%) | 0.02 | 3.52 | 14.99 | |
| 10.20.12.27 | 16624 | 10.16.116.53 | 16840 | 0xCF139E0A | g711U | 1918 | 10 (0.5%) | 159.79 | 276.67 | 7.80 | X |
| 10.10.12.1 | 57840 | 10.10.13.136 | 16000 | 0xD94ACE43 | Reserved for RTCP | 1 | 0 (0.0%) | 0.00 | 0.00 | 0.00 | |
| 10.10.12.1 | 57840 | 10.10.13.136 | 16000 | 0x3B6155CA | RTPType-100 | 13359 | -463 (-3.6%) | 0.00 | 0.00 | 0.00 | X |
| 10.10.13.136 | 16000 | 10.10.12.1 | 57840 | 0x11E585B9 | RTPType-100 | 16835 | -11088 (-192.9%) | 0.00 | 0.00 | 0.00 | X |
| 10.16.116.53 | 16866 | 10.20.12.27 | 16590 | 0x17FEA6DD | RTPType-97 | 5009 | 1 (0.0%) | 0.00 | 0.00 | 0.00 | X |
| 10.20.12.27 | 16590 | 10.16.116.53 | 16866 | 0x7E30B814 | RTPType-97 | 7588 | 0 (0.0%) | 0.00 | 0.00 | 0.00 | X |



The audio from Media Broker to the SIP phone is not encrypted and can be heard using wireshark's RTP Player, by Selecting the appropriate Stream and pressing *Analyze* then selecting *Player*. The stream needs to be decoded, for now select a large Jitter Buffer (200ms).

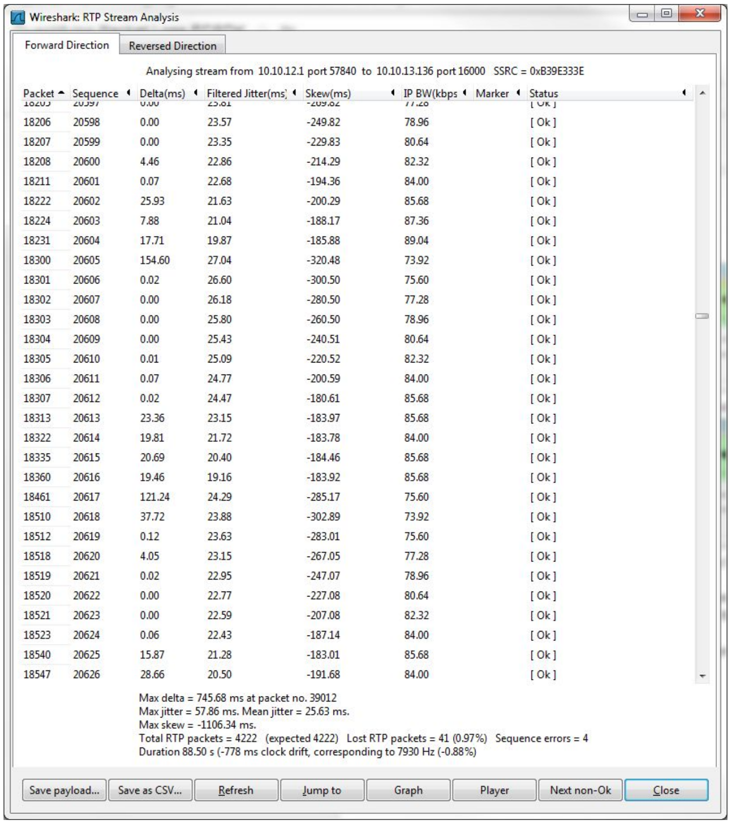



The Stream is clearly disrupted with sequence errors from 36s onwards.

## Stream Analysis

Wireshark provides some Stream Analysis, which is helpful for audio diagnosis.

In this example, wireshark shows clumping on the inbound g711 stream. Bursts of inbound packets arrive in large groups and later than expected:




| Packet | Sequence | Delta(ms) | Filtered Jitter(ms) | Skew(ms) | IP BW(kbps) | Marker | Status |
|---|---|---|---|---|---|---|---|
| 18206 | 20598 | 0.00 | 23.57 | -249.82 | 78.96 | | [ Ok ] |
| 18207 | 20599 | 0.00 | 23.35 | -229.83 | 80.64 | | [ Ok ] |
| 18208 | 20600 | 4.46 | 22.86 | -214.29 | 82.32 | | [ Ok ] |
| 18211 | 20601 | 0.07 | 22.68 | -194.36 | 84.00 | | [ Ok ] |
| 18222 | 20602 | 25.93 | 21.63 | -200.29 | 85.68 | | [ Ok ] |
| 18224 | 20603 | 7.88 | 21.04 | -188.17 | 87.36 | | [ Ok ] |
| 18231 | 20604 | 17.71 | 19.87 | -185.88 | 89.04 | | [ Ok ] |
| 18300 | 20605 | 154.60 | 27.04 | -320.48 | 73.92 | | [ Ok ] |
| 18301 | 20606 | 0.02 | 26.60 | -300.50 | 75.60 | | [ Ok ] |
| 18302 | 20607 | 0.00 | 26.18 | -280.50 | 77.28 | | [ Ok ] |
| 18303 | 20608 | 0.00 | 25.80 | -260.50 | 78.96 | | [ Ok ] |
| 18304 | 20609 | 0.00 | 25.43 | -240.51 | 80.64 | | [ Ok ] |
| 18305 | 20610 | 0.01 | 25.09 | -220.52 | 82.32 | | [ Ok ] |
| 18306 | 20611 | 0.07 | 24.77 | -200.59 | 84.00 | | [ Ok ] |
| 18307 | 20612 | 0.02 | 24.47 | -180.61 | 85.68 | | [ Ok ] |
| 18313 | 20613 | 23.36 | 23.15 | -183.97 | 85.68 | | [ Ok ] |
| 18322 | 20614 | 19.81 | 21.72 | -183.78 | 84.00 | | [ Ok ] |
| 18335 | 20615 | 20.69 | 20.40 | -184.46 | 85.68 | | [ Ok ] |
| 18360 | 20616 | 19.46 | 19.16 | -183.92 | 85.68 | | [ Ok ] |
| 18461 | 20617 | 121.24 | 24.29 | -285.17 | 75.60 | | [ Ok ] |
| 18510 | 20618 | 37.72 | 23.88 | -302.89 | 73.92 | | [ Ok ] |
| 18512 | 20619 | 0.12 | 23.63 | -283.01 | 75.60 | | [ Ok ] |
| 18518 | 20620 | 4.05 | 23.15 | -267.05 | 77.28 | | [ Ok ] |
| 18519 | 20621 | 0.02 | 22.95 | -247.07 | 78.96 | | [ Ok ] |
| 18520 | 20622 | 0.00 | 22.77 | -227.08 | 80.64 | | [ Ok ] |
| 18521 | 20623 | 0.00 | 22.59 | -207.08 | 82.32 | | [ Ok ] |
| 18523 | 20624 | 0.06 | 22.43 | -187.14 | 84.00 | | [ Ok ] |
| 18540 | 20625 | 15.87 | 21.28 | -183.01 | 85.68 | | [ Ok ] |
| 18547 | 20626 | 28.66 | 20.50 | -191.68 | 84.00 | | [ Ok ] |



G711 packets should arrive every 20ms. The Delta is the time between packets and the example above shows that 18301 to 18313 have all arrived at the same time at packet 18300 which recorded a Delta of 154ms. Also, the skew is large, this indicates that the audio packet has arrived ~180-290ms late, relative to their expected packet arrival time.

As a comparison, the inbound stream from the SIP phone; which is considered good; has a consistent 20ms Delta, a Jitter value of almost 0 and a Skew of only 3ms.

## H.264 Codec

The first keyframe contains will typically contain the Sequence Parameter Set which contains information common to all the pictures in the H264 stream.
This will contain information like the H.264 profile being used that should match the SDP negotiation:

### H.264 Profiles:

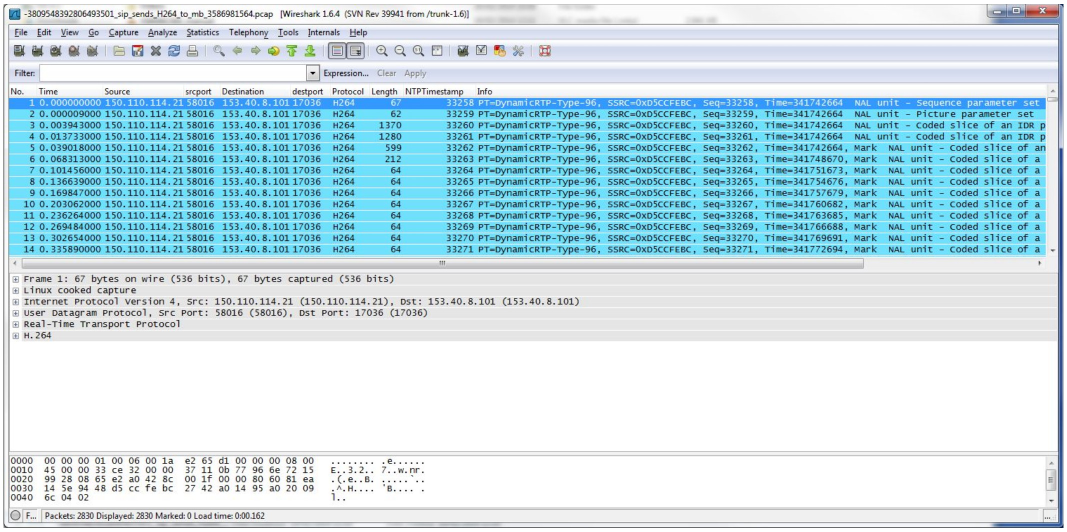
| |
|---|
| profile-level-id define the properties of the incoming H.264 stream. It indicates the profile level the decoder must comply to in order to decode the incoming NAL unit stream.<br><br>It is a Base16 representation of 3 bytes in the SPS of NAL unit.<br>1 byte - profile_idc<br>1 byte - profile_iop<br>1 byte - level_idc<br><br>for example, profile-level-id=42E015 imply<br>profile_idc = 42 imply Baseline profile<br>profile_iop = E0 imply only common subset of profile is supported<br>level_idc = 15 imply level 2.1<br><br>In general, profile-level-id and packetization-mode identify the media format configuration for H.264<br><br>See RFC 3984 Section 8.1 for details |

Check http://en.wikipedia.org/wiki/H.264/MPEG-4_AVC#Levels for more details on specific levels.
The SPS can be used to verify what an H264 stream contains:
https://cardinalpeak.com/blog/the-h-264-sequence-parameter-set/

### H264 Decoding

If you know an unencrypted/decrypted stream is H264, but wireshark isn't showing it as such, go to preferences->protocols->h264 and set the payload type to the one for the h264 stream. If multiple are needed then it can take a comma separated list (no spaces).

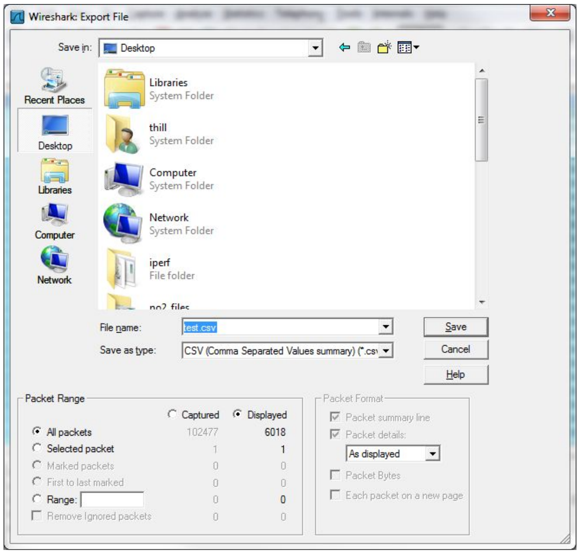
Finding the Sequence Parameter Set



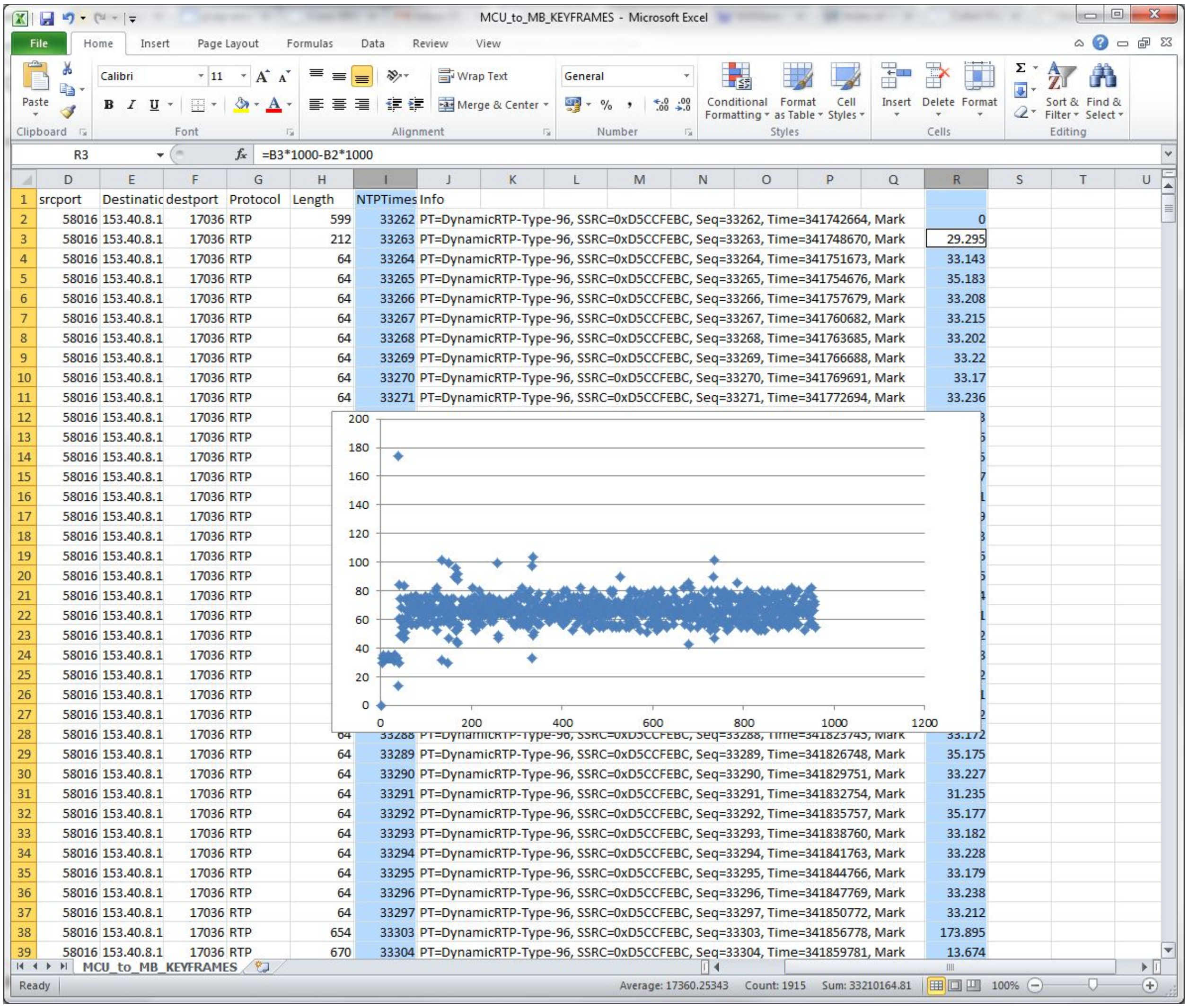
## Audio & Video Analysis

Typically all the packets that create a frame are sent around the same time, so the deltas in video streams are not comparable to those of audio streams.

1) Select the Stream to Analyze and Prepare a Filter
2) Alter the filter to only include Marked Packets - append: *&& rtp.marker==true*
3) Apply the filter
4) Save the Displayed Packets as a CSV and Open in Excel (or equivalent).




5) Add a New Column “Frame Deltas”
6) The Delta is equal to the Difference in Time*1000 from the previous packet and the current packet
7) This can be used to identify unusual ‘frame’ behaviour.




You can perform similar analysis with audio streams.

Generally, this helps us determine how well behaved a stream is; areas with high-deltas help indicate where in the call problems are occurring. Reasons for high-deltas may include:

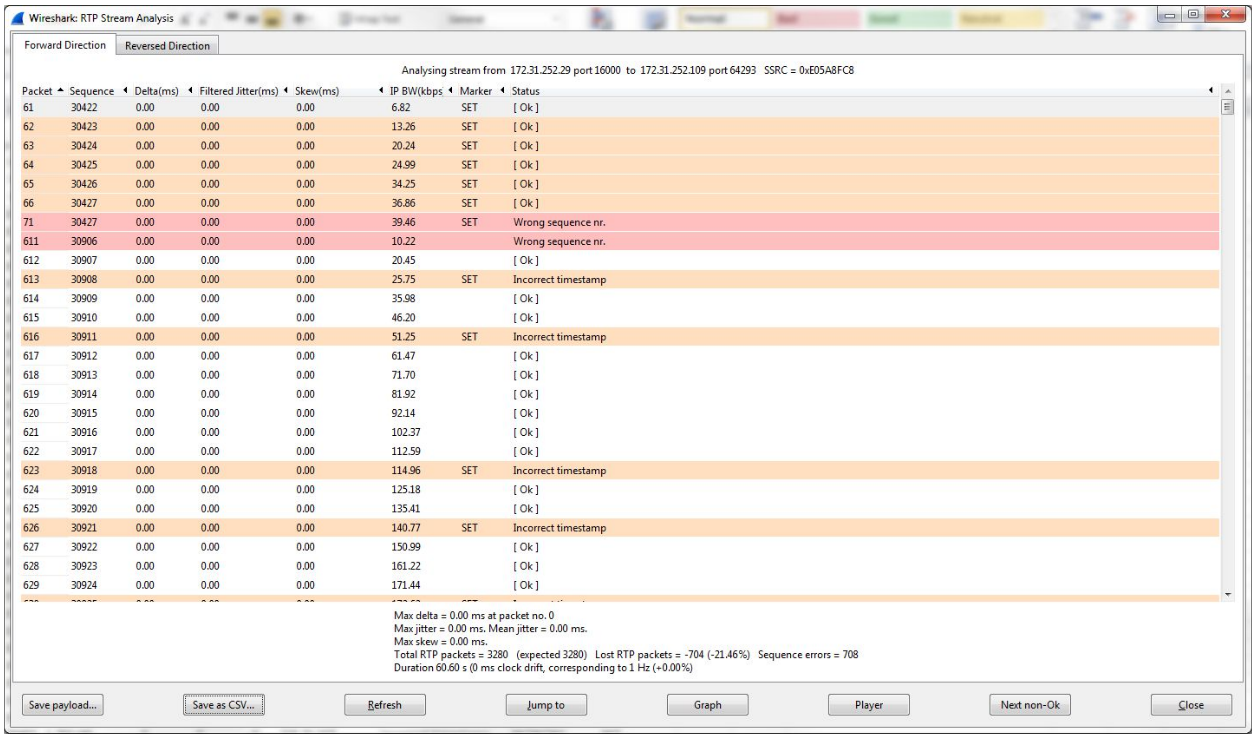
- Areas of High Packet Loss
- Queued traffic in the network
- Delayed Packets in the network

## Measuring Retransmissions

It is often useful to separate retransmissions from the original stream.

This is a many step process:

1. Take an appropriate capture - in this example we've taken it from Media Broker
2. Separate the Stream using an appropriate filter
3. Perform Stream Analysis:
4. Save the Stream as a CSV

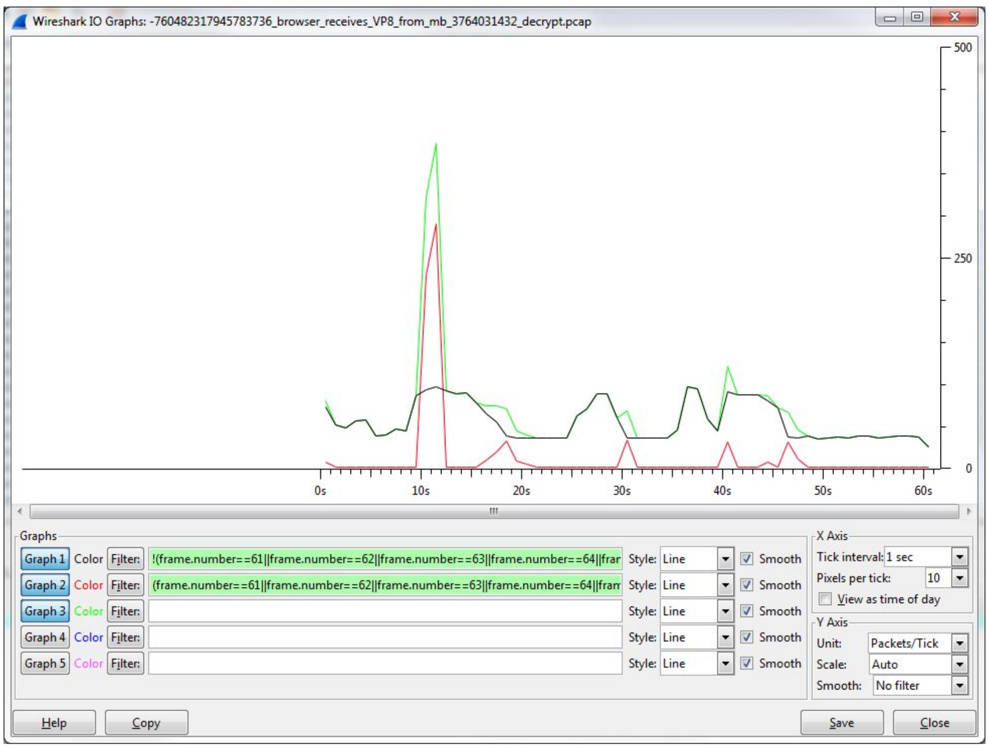



| Packet | Sequence | Delta(ms) | Filtered Jitter(ms) | Skew(ms) | IP BW(kbps | Marker | Status |
|---|---|---|---|---|---|---|---|
| 61 | 30422 | 0.00 | 0.00 | 0.00 | 6.82 | SET | [ Ok ] |
| 62 | 30423 | 0.00 | 0.00 | 0.00 | 13.26 | SET | [ Ok ] |
| 63 | 30424 | 0.00 | 0.00 | 0.00 | 20.24 | SET | [ Ok ] |
| 64 | 30425 | 0.00 | 0.00 | 0.00 | 24.99 | SET | [ Ok ] |
| 65 | 30426 | 0.00 | 0.00 | 0.00 | 34.25 | SET | [ Ok ] |
| 66 | 30427 | 0.00 | 0.00 | 0.00 | 36.86 | SET | [ Ok ] |
| 71 | 30427 | 0.00 | 0.00 | 0.00 | 39.46 | SET | Wrong sequence nr. |
| 611 | 30906 | 0.00 | 0.00 | 0.00 | 10.22 | | Wrong sequence nr. |
| 612 | 30907 | 0.00 | 0.00 | 0.00 | 20.45 | | [ Ok ] |
| 613 | 30908 | 0.00 | 0.00 | 0.00 | 25.75 | SET | Incorrect timestamp |
| 614 | 30909 | 0.00 | 0.00 | 0.00 | 35.98 | | [ Ok ] |
| 615 | 30910 | 0.00 | 0.00 | 0.00 | 46.20 | | [ Ok ] |
| 616 | 30911 | 0.00 | 0.00 | 0.00 | 51.25 | SET | Incorrect timestamp |
| 617 | 30912 | 0.00 | 0.00 | 0.00 | 61.47 | | [ Ok ] |
| 618 | 30913 | 0.00 | 0.00 | 0.00 | 71.70 | | [ Ok ] |
| 619 | 30914 | 0.00 | 0.00 | 0.00 | 81.92 | | [ Ok ] |
| 620 | 30915 | 0.00 | 0.00 | 0.00 | 92.14 | | [ Ok ] |
| 621 | 30916 | 0.00 | 0.00 | 0.00 | 102.37 | | [ Ok ] |
| 622 | 30917 | 0.00 | 0.00 | 0.00 | 112.59 | | [ Ok ] |
| 623 | 30918 | 0.00 | 0.00 | 0.00 | 114.96 | SET | Incorrect timestamp |
| 624 | 30919 | 0.00 | 0.00 | 0.00 | 125.18 | | [ Ok ] |
| 625 | 30920 | 0.00 | 0.00 | 0.00 | 135.41 | | [ Ok ] |
| 626 | 30921 | 0.00 | 0.00 | 0.00 | 140.77 | SET | Incorrect timestamp |
| 627 | 30922 | 0.00 | 0.00 | 0.00 | 150.99 | | [ Ok ] |
| 628 | 30923 | 0.00 | 0.00 | 0.00 | 161.22 | | [ Ok ] |
| 629 | 30924 | 0.00 | 0.00 | 0.00 | 171.44 | | [ Ok ] |

5. Open the Steam in Excel:
6. Create a New Column for Identifying Wrong Sequence Number:
   a. Something like this: *=IF(I3="Wrong sequence nr.",A3,"")*
   b. A Better way is to look for recurring sequence numbers: *=IF(ISNA(VLOOKUP(B3,$B$2:B2,1,FALSE)=B3),A3,"")*
7. Have a Column for creating the appropriate filter:
   a. Something like : *=IF(M3="","",CONCATENATE("frame.number==",M3))*

   b. or: use ISNA if #N/A comes up
8. Have a Column that concatenates a big filter:
   a. Something like: *=IF(N3="",O2,CONCATENATE(O2,"||",N3))*
9. Copy the Filter in Wireshark, it should look like:

   *||frame.number==61||frame.number==62||frame.number==63||frame.number==64 ……*
   a. You'll need to remove the first || and the last character
10. Now you can use wireshark's filter to differentiate between the original stream and retransmissions:




This graph shows the results from a wireshark of an isolated video stream:

- Original stream in black
- Transmissions in red
- Total in Green

## Double Packets

When a tcpdump is performed using the *-i any* option the capture is taken above the interface layer.  We have seen some capture taken at Media Broker that contain duplicates for packets being sent by Media Brother. These should have only been targeted at a single interface, but at the level the capture is taken the packet is presented to both interfaces.

Duplicate packets can disrupt analysis, but they can be removed using the following wireshark utility:

```
editcap -d orig.pcap noDups.pcap
```

## **Analyzing Streams on a Network Bridge**

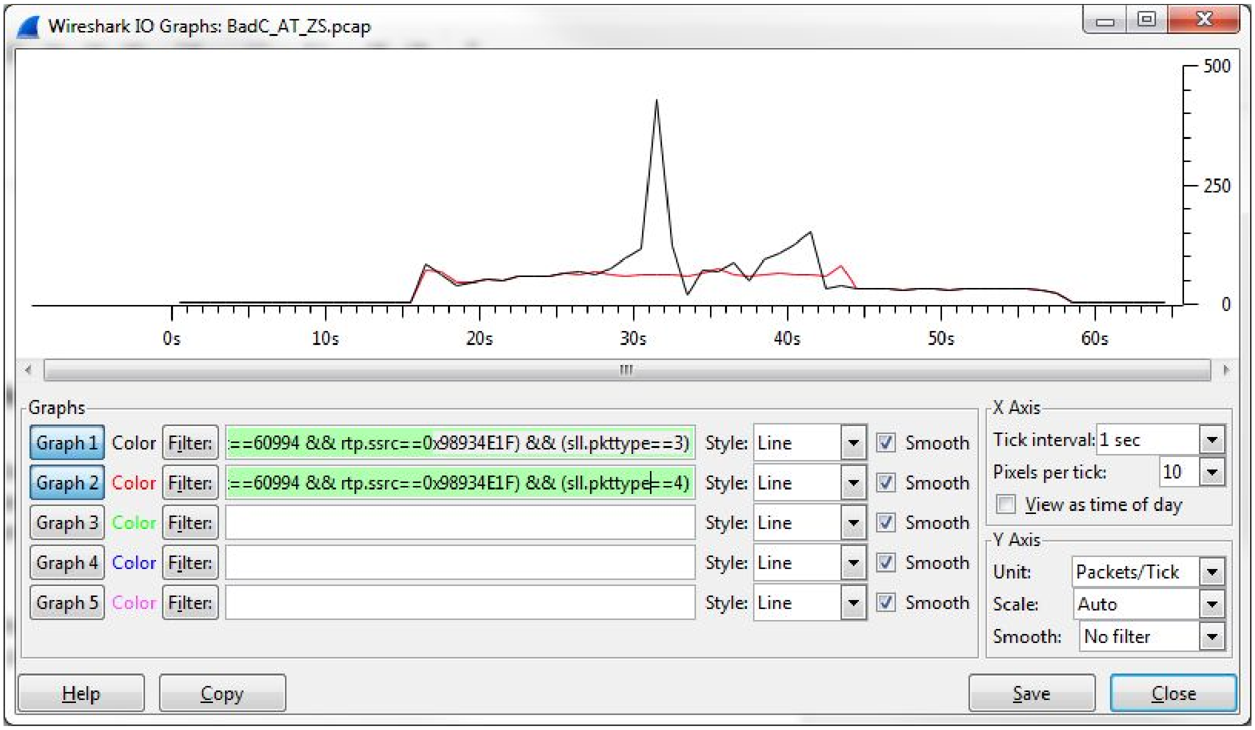
In this example zeroshell is being used to limit bandwidth across as a network bridge. This section is not a tutorial for setting up Zeroshell or capturing packets from the zeroshell machine.  Instead, it contains some useful wireshark filters for determining which interface packets are flowing through on the transparent network bridge.

On the bridge each packet is displayed twice: on the way in, and on the way out. Assuming that a capture on any-interface was performed; they can be distinguished by the “Linux cooked capture information”.

The following filters can be used:

| Filter | Packets |
|---|---|
| (sll.pkttype==3) | Unicast to another host |
| (sll.pkttype==4) | Sent by us |

These can be used to filter an srrc stream in the capture:




We can see above that Zero Shell has flattened or limited the bandwidth available.

# Picture Quality

## Picture Loss Recovery

When a video steam cannot be rendered a new keyframe is required.  These are requested by a client in a number of ways:

- PLIs,
- RFC2032 FIRs
- SIP INFO.

 PLIs are the only mechanism on the web side, so MB injects PLIs into a stream when it receives FIRs or INFOs. Media Broker can translate between PLIs and FIRs on the SIP side meaning any FIRs going to MB are sent as PLIs to the web client and any PLIs sent by the web client will be translated to FIRs on the SIP side.

PLIs can be found in the RTCP stream:

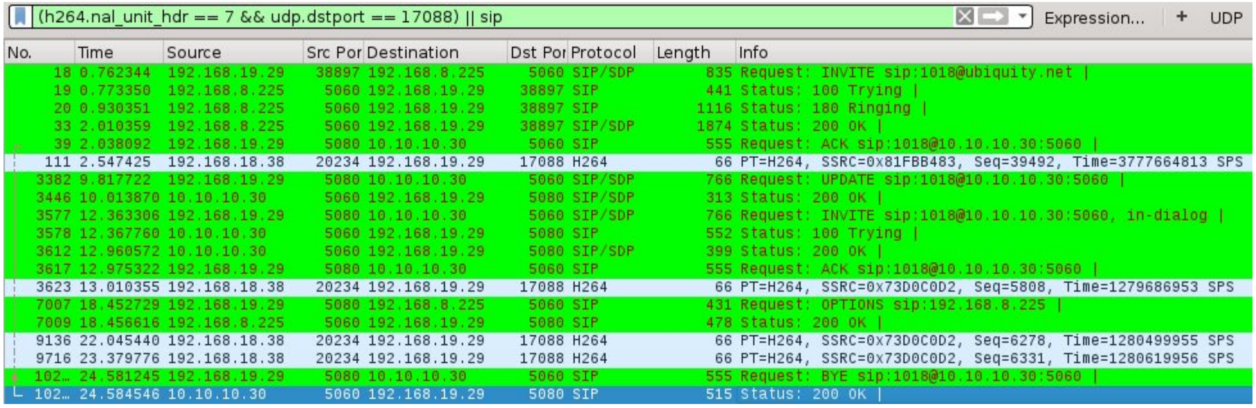
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 9105 | 21.983580 | 192.168.19.29 | 17089 | 192.168.18.38 | 20235 | RTCP | 76 | Receiver Report | |
| 9112 | 21.993169 | 192.168.19.29 | 17089 | 192.168.18.38 | 20235 | RTCP | 76 | Receiver Report | |
| 9113 | 22.003537 | 192.168.19.29 | 17089 | 192.168.18.38 | 20235 | RTCP | 88 | Receiver Report | Payload-specific Feedback PLI |
| 9132 | 22.033741 | 192.168.19.29 | 17089 | 192.168.18.38 | 20235 | RTCP | 88 | Receiver Report | Payload-specific Feedback PLI |
| 9165 | 22.073769 | 192.168.19.29 | 17089 | 192.168.18.38 | 20235 | RTCP | 88 | Receiver Report | Payload-specific Feedback PLI |
| 9172 | 22.084397 | 192.168.19.29 | 17089 | 192.168.18.38 | 20235 | RTCP | 100 | Sender Report | Source description |

If you want to filter, auto-completion can help to write a wireshark filter for sub-types:

*rtcp.psfb.fmt==1*

rtcp is the packet type, psfb for payload specific feedback which is the section type for PLIs, and fmt is the feedback message type).

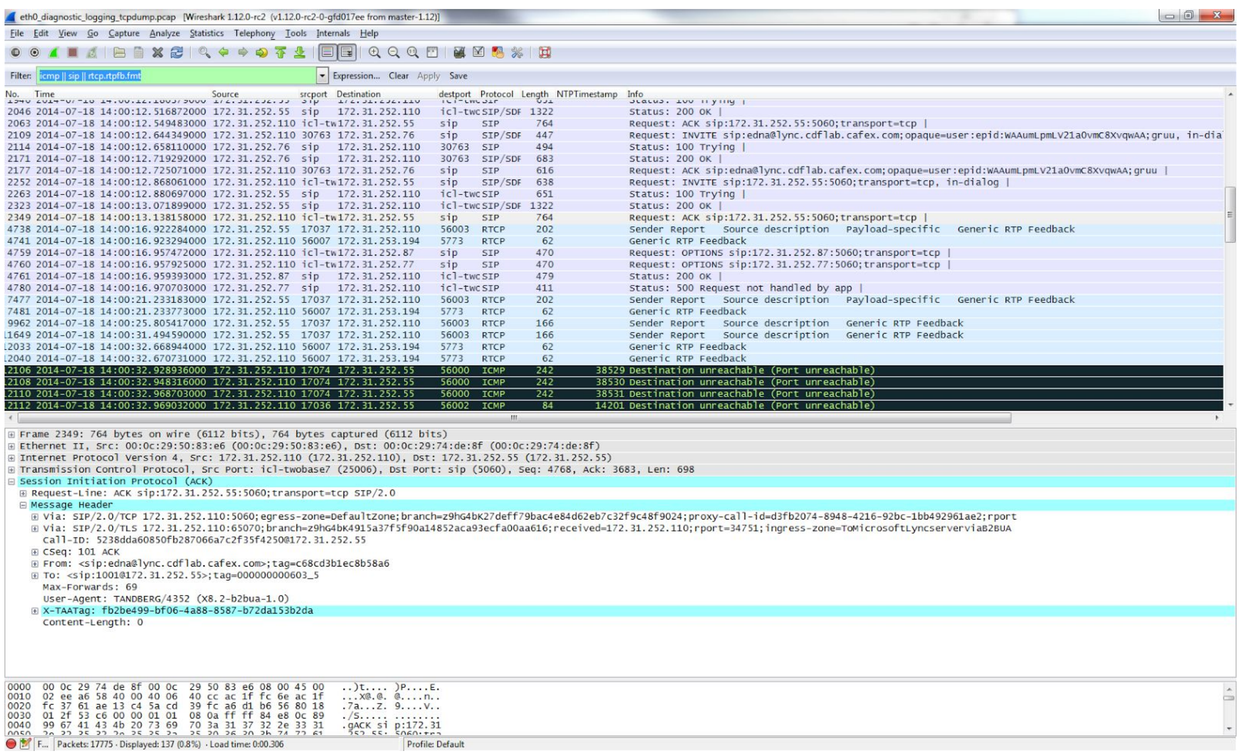
The corresponding RTP stream will hopefully contain a keyframe shortly afterwards. The filter:  *h264.nal_unit_hdr* can help you find them:




## Finding PLIs

This filter will help find PLIs and was useful in diagnosing an issue with ICMP errors:

*icmp || sip || rtcp.rtpfb.fmt*




The Screenshot shows:

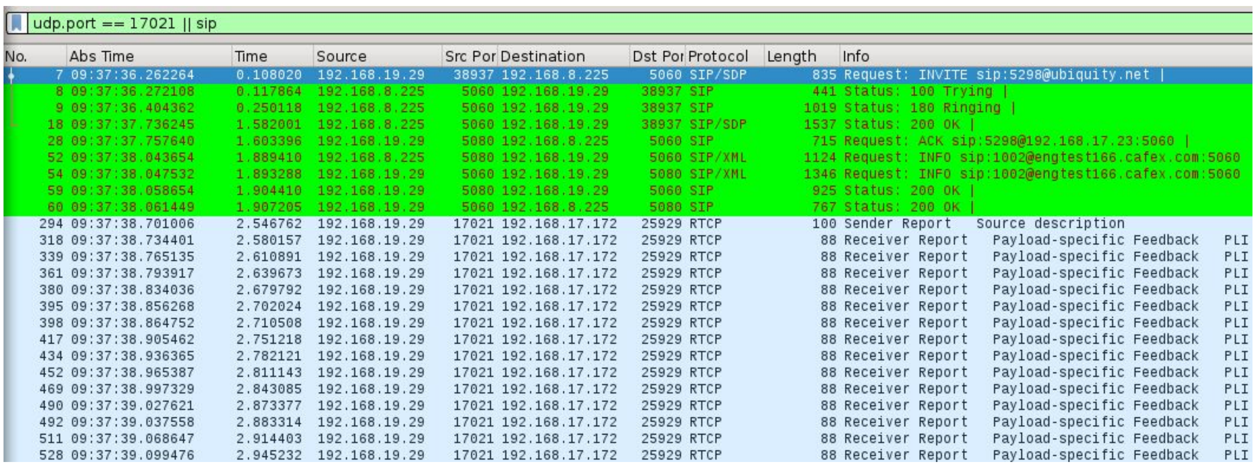
- The call finishes setting up at 14:00:13 with receipt on an ACK.

- There is a PLI at 14:00:21 - Marked with Payload Specific
- The first ICMP event happens at 14:00:32.

This was caused by a third party application crashing on receipt of a PLI.

## RFC2032 FIRs & SIP INFOs

If PLIs do not appear to be working and you can't find any PLIs coming from a SIP device it may be it uses RFC2032 FIRs and SIP INFOs (also known as PFUs or FPUs). SIP INFOs received by FAS will normally result in the web client sending a keyframe as MB translates them to PLIs in that direction, and FCSDK can be configured to have FAS send SIP INFOs when the web client sends a PLI.

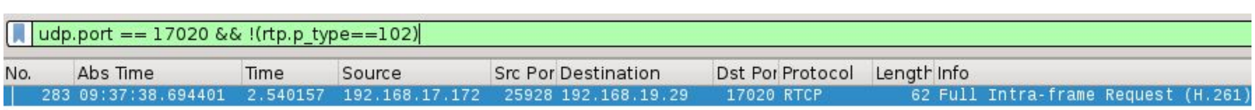


udp.port == 17021 || sip

| No. | Abs Time | Time | Source | Src Por | Destination | Dst Por | Protocol | Length | Info |
|---|---|---|---|---|---|---|---|---|---|
| 7 | 09:37:36.262264 | 0.108020 | 192.168.19.29 | 38937 | 192.168.8.225 | 5060 | SIP/SDP | 835 | Request: INVITE sip:5298@ubiquity.net \| |
| 8 | 09:37:36.272108 | 0.117864 | 192.168.8.225 | 5060 | 192.168.19.29 | 38937 | SIP | 441 | Status: 100 Trying \| |
| 9 | 09:37:36.404362 | 0.250118 | 192.168.8.225 | 5060 | 192.168.19.29 | 38937 | SIP | 1019 | Status: 180 Ringing \| |
| 18 | 09:37:37.736245 | 1.582001 | 192.168.8.225 | 5060 | 192.168.19.29 | 38937 | SIP/SDP | 1537 | Status: 200 OK \| |
| 28 | 09:37:37.757640 | 1.603396 | 192.168.19.29 | 5080 | 192.168.8.225 | 5060 | SIP | 715 | Request: ACK sip:5298@192.168.17.23:5060 \| |
| 52 | 09:37:38.043654 | 1.889410 | 192.168.8.225 | 5080 | 192.168.19.29 | 5060 | SIP/XML | 1124 | Request: INFO sip:1002@engtest166.cafex.com:5060 |
| 54 | 09:37:38.047532 | 1.893288 | 192.168.19.29 | 5060 | 192.168.19.29 | 5080 | SIP/XML | 1346 | Request: INFO sip:1002@engtest166.cafex.com:5060 |
| 59 | 09:37:38.058654 | 1.904410 | 192.168.19.29 | 5080 | 192.168.19.29 | 5060 | SIP | 925 | Status: 200 OK \| |
| 60 | 09:37:38.061449 | 1.907205 | 192.168.19.29 | 5060 | 192.168.8.225 | 5080 | SIP | 767 | Status: 200 OK \| |
| 294 | 09:37:38.701006 | 2.546762 | 192.168.19.29 | 17021 | 192.168.17.172 | 25929 | RTCP | 100 | Sender Report Source description |
| 318 | 09:37:38.734401 | 2.580157 | 192.168.19.29 | 17021 | 192.168.17.172 | 25929 | RTCP | 88 | Receiver Report Payload-specific Feedback PLI |
| 339 | 09:37:38.765135 | 2.610891 | 192.168.19.29 | 17021 | 192.168.17.172 | 25929 | RTCP | 88 | Receiver Report Payload-specific Feedback PLI |
| 361 | 09:37:38.793917 | 2.639673 | 192.168.19.29 | 17021 | 192.168.17.172 | 25929 | RTCP | 88 | Receiver Report Payload-specific Feedback PLI |
| 380 | 09:37:38.834036 | 2.679792 | 192.168.19.29 | 17021 | 192.168.17.172 | 25929 | RTCP | 88 | Receiver Report Payload-specific Feedback PLI |
| 395 | 09:37:38.856268 | 2.702024 | 192.168.19.29 | 17021 | 192.168.17.172 | 25929 | RTCP | 88 | Receiver Report Payload-specific Feedback PLI |
| 398 | 09:37:38.864752 | 2.710508 | 192.168.19.29 | 17021 | 192.168.17.172 | 25929 | RTCP | 88 | Receiver Report Payload-specific Feedback PLI |
| 417 | 09:37:38.905462 | 2.751218 | 192.168.19.29 | 17021 | 192.168.17.172 | 25929 | RTCP | 88 | Receiver Report Payload-specific Feedback PLI |
| 434 | 09:37:38.936365 | 2.782121 | 192.168.19.29 | 17021 | 192.168.17.172 | 25929 | RTCP | 88 | Receiver Report Payload-specific Feedback PLI |
| 452 | 09:37:38.965387 | 2.811143 | 192.168.19.29 | 17021 | 192.168.17.172 | 25929 | RTCP | 88 | Receiver Report Payload-specific Feedback PLI |
| 469 | 09:37:38.997329 | 2.843085 | 192.168.19.29 | 17021 | 192.168.17.172 | 25929 | RTCP | 88 | Receiver Report Payload-specific Feedback PLI |
| 490 | 09:37:39.027621 | 2.873377 | 192.168.19.29 | 17021 | 192.168.17.172 | 25929 | RTCP | 88 | Receiver Report Payload-specific Feedback PLI |
| 492 | 09:37:39.037558 | 2.883314 | 192.168.19.29 | 17021 | 192.168.17.172 | 25929 | RTCP | 88 | Receiver Report Payload-specific Feedback PLI |
| 511 | 09:37:39.068647 | 2.914403 | 192.168.19.29 | 17021 | 192.168.17.172 | 25929 | RTCP | 88 | Receiver Report Payload-specific Feedback PLI |
| 528 | 09:37:39.099476 | 2.945232 | 192.168.19.29 | 17021 | 192.168.17.172 | 25929 | RTCP | 88 | Receiver Report Payload-specific Feedback PLI |

In this example we can see lots of PLIs going from Media Broker to a SIP phone, but none in the other direction; INFOs are being received however. Often the INFOs will repeat until the phone gets a keyframe, but this device appears not to do that.

Looking at the DTLS, we can see the INFOs came in to the gateway, prior to the web client finishes establishing its media path. Unlike standard RTCP these are sent on the RTP ports and are effectively an RTCP packet containing just the FIR. With newer versions of Wireshark it seems mostly it will automatically decode them as RTCP, but it can't be relied upon, so there are two ways to filter for them.



udp.port == 17020 && !(rtp.p_type==102)

| No. | Abs Time | Time | Source | Src Por | Destination | Dst Por | Protocol | Lengtl | Info |
|---|---|---|---|---|---|---|---|---|---|
| 283 | 09:37:38.694401 | 2.540157 | 192.168.17.172 | 25928 | 192.168.19.29 | 17020 | RTCP | 62 | Full Intra-frame Request (H.261) |

**Note:**
If you want to check if PLIs or FIRs have raced in before the DTLS exchange has completed and not repeated then you can create a filter that shows dtls or video RTP/RTCP ports with rtcp packets.

*dtls || ((udp.port==17020 || udp.port=17021) && rtcp)*

## NACK

When packet loss occurs in the video stream on the WebRTC leg of a call the main recovery method used is Negative ACKnowledgement, a.k.a NACK. This is request for retransmission of some RTP packets sent via a section in an RTCP packet, which should result in the other end resending them if it still has a copy. If too much packet loss occurs, NACKs will not always be used, instead PLIs will be sent and old data given up on, or if the round trip time is high (over 50ms) the stream will switch on ULPFEC which will result in far fewer, if any, NACKs being sent.

As a result of encryption on the web side, any packet capture where you need to confirm which NACKs are present in RTCP will need to be decrypted. It is possible to tell if retransmissions of the requested RTP packets happens without decrypting, and if RTX is not used then you can simply look at the sequence numbers to see which; however, when RTX is used you will still need to decrypt to find out which are being resent.

## RTX

Modern webRTC clients, support RTX, where NACKs are not simple retransmissions. You can tell if this is in use as both the offer and answer on the web side will include the RTX codec, and have two SSRCs for the video media line which share the same MSID.

When in use responses to NACKs will be sent with the payload type for RTX and using a separate set of sequence number; to tell the sequence number it was sent for you need to look at the first two bytes of the payload. The extra sequence number is a special feature of RTX packets that is removed once processing of it is done, so you will not see it on the SIP side.

**Note**: RTX appears multiple times with different payload types in more recent versions of Chrome, you may see it in its FMTP attribute, but only one will be used and if there are multiple in both the offer and answer negotiations the RTX will be the one for RED.

## Fragmentation

UDP packets that are fragmented are not understood by any web clients, but if a packets comes from the SIP side that is at or above the maximum MTU of the network for the path between Media Broker and the FCSDK client then it will get fragmented as encryption adds a few bytes to the packet.
To check quickly if fragmentation is present in a capture use the filter:
*ip.flags.mf ==1 or ip.frag_offset gt 0.*
If this results in "Fragmented IP Protocol" frames being show for a stream then fragmentation will be a problem.

## Adaptive Bitrate

Video endpoints distributed over the internet cannot guarantee a stable bitrate required for real time communications.  It is important that the stream of packets which construct a video call arrive at their destination in a timely fashion; depending on the network pathway between an client and Media Broker, network buffering or QoS restrictions may limit or severely impact video performance.

Media Broker implements REMB (for WEB-RTC Clients) and TMMBR (for SIP Clients) specifications so that Media Broker can monitor the RTCP Reports and react to the ever changing environment clients may face.

These protocols will request more or less bandwidth from clients if the network conditions change.  If there is no change the protocol will maintain the current bitrate.

Media Broker allows the administrator to configure constraints on the bandwidth to maintain video quality parameters can be met by the business requirement.

In its current implementation Media Broker does not support Dynamic Video Resolution Changes; this means if a call is established at 720HD this resolution will be maintained throughout the call.

The *Maximum* and *Minimum Adaptive Bitrate* values, give bounds to the threshold that the Media Broker will go to when rendering video

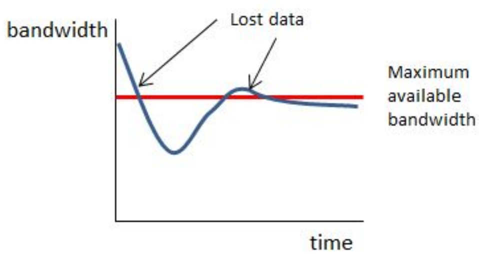
This value effects Media Broker in two ways:

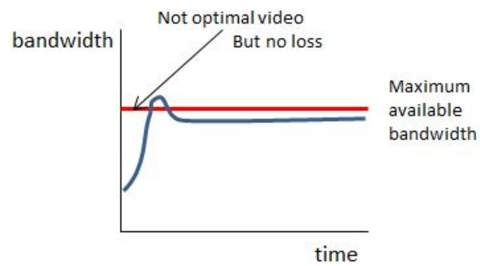
1. Media Broker will request no-more or less than these value from Clients.
2. Media Broker will not use a value outside of this range to encode video. Media Broker will ensure that the Video it generates is of an appropriate quality and not render a 720HD video with insufficient bitrate, nor render video with an inappropriately high bitrate with a diminished return.

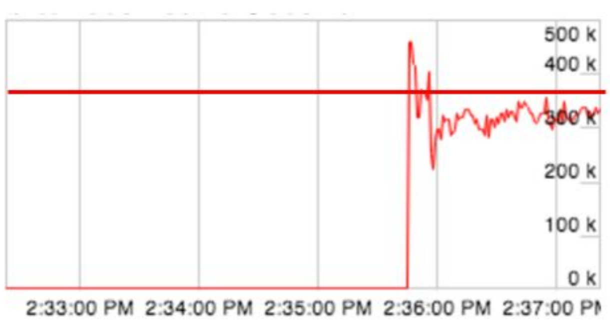
The *Initial Adaptive Bitrate* value is used by Media Broker when sending video at the beginning of a call before there is enough data collected from the RTCP to behave appropriately.

In most consumer cases it is appropriate to set this value equal to the *Minimum Adaptive Bitrate*; if the network is sufficient the bitrate of the call will improve shortly after the call starts; however, some video solutions may prefer the video starts at a higher bitrate, in which case clients on an insufficient networks will have a worse experience until the bitrate falls.

The following diagram demonstrates how data is received at a client with an actual physical bandwidth which is capped at 350kbps.  Such a consumer will never be able to receive more than 350kbps due to a limitation in their network capabilities.





The graphs on the left hand side show what the user's experience will be when the *Initial Value* is around 512kbps (more than the client's maximum). The graphs on the right hand side show what happens for this user when the Initial value is set to 300kbps (below the client's minimum).
The graph on the left shows the user will receive a poor video experience for the first few seconds of the call (and data may be lost); the user will see degraded video and audio may be effected.

The graph on the right shows the user's maximum bandwidth is reached very quickly and there is very little interruption to the user's experience.

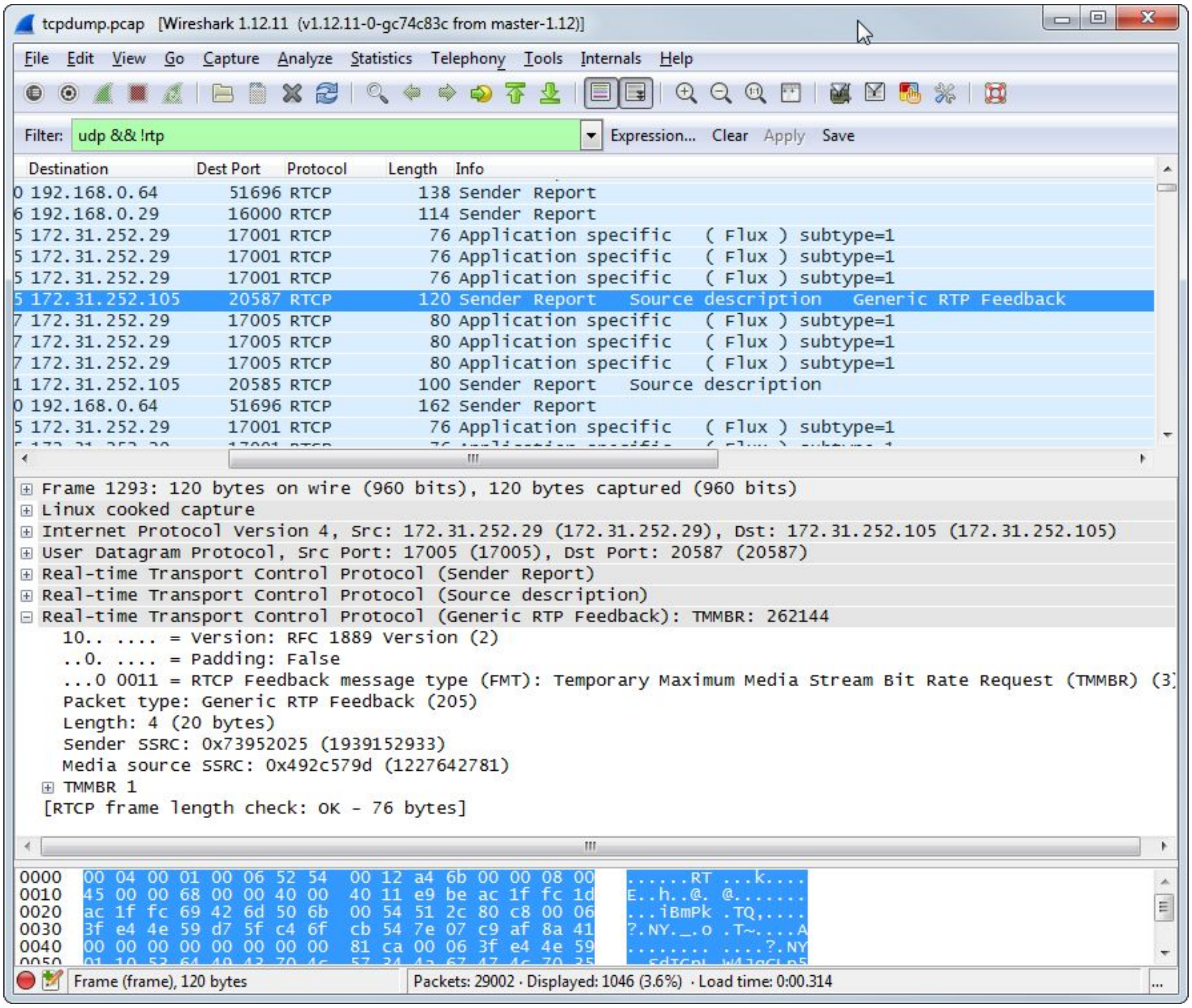
## SIP-side considerations

In solutions where clients are establishing video calls to traditional SIP-based video devices, such as deskphones, soft-UAs or MCUs the Media Broker can utilize the network architecture for a better video experience.

If SIP-endpoints are based in a dedicated network the Media Broker can apply a fixed-bitrate to a video negotiation. This is advantageous because typically dedicated video-networks can guarantee the network stability required for communications and thus both endpoints can agree an optimal bitrate when the call is established.
The value of the fixed bitrate can be found in the Media Broker's *proxy.properties* file. The property is *sip.bitrate.override.main=4000000*. This will set an AS parameter equal to 4mbps in the SDP sent in requests on the SIP side.

## Finding TMMBR

In wireshark TMMBR appears inside a *Generic RTP Feedback* block of the RTCP:




## Finding REMB

This is a lot harder to see in a wireshark, because the RTCP packets are encrypted. You can see a close approximation of what is being sent by Media broker when it is received at your client, if you are using Chrome. This will be covered in the Browser topic when looking at bandwidth estimates.

## Analyzing Lip Sync

Once the audio and video RTP streams leave the sender, they are independent of each other and it's the job of the decoder to cope for any fluctuations caused by network conditions when reconstituting the streams.

Lip Sync issues can be seen if the audio and video streams are vastly apart from each other and clients should be using the RTCP to keep packets in sync.

The RTCP sender report contains information that the decoder can use to keep packets in sync:

Example Audio RTCP Sender Report:

```
⊟ Real-time Transport Control Protocol (Sender Report)
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 0000 = Reception report count: 0
    Packet type: Sender Report (200)
    Length: 6 (28 bytes)
    Sender SSRC: 0x98171833 (2551650355)
    Timestamp, MSW: 3613377647 (0xd75fc46f)
    Timestamp, LSW: 3687079100 (0xdbc45cbc)
    [MSW and LSW as NTP timestamp: Jul  3, 2014 12:00:47.858465000 UTC]
    RTP timestamp: 1891280655
    Sender's packet count: 4
    Sender's octet count: 640
```

Example Video RTCP Sender Report:

```
⊟ Real-time Transport Control Protocol (Sender Report)
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 0001 = Reception report count: 1
    Packet type: Sender Report (200)
    Length: 12 (52 bytes)
    Sender SSRC: 0x3fe44e59 (1071926873)
    Timestamp, MSW: 3613377649 (0xd75fc471)
    Timestamp, LSW: 1719039185 (0x667674d1)
    [MSW and LSW as NTP timestamp: Jul  3, 2014 12:00:49.400244000 UTC]
    RTP timestamp: 3383869149
    Sender's packet count: 100
    Sender's octet count: 63585
```

In both cases the Sender Report contains an RTP timestamp and a ‘wall clock’ time. Each RTP packet contains its own RTP timestamp, with all this information the receiver can use this information to playback the audio and video streams relative to one another. It’s not so straightforward though, as the decoder will be making assumptions on the sender’s clock drift, clock skew to resynchronize the steams. Essentially, there are 3 ways of synchronizing the streams:

- Add silence
- Just ahead (remove excess)
- Alter playback feed of other stream to speed one up or slow one down

Neither of these things are instantaneous as the decoder needs to take time to sample enough data to detect a suitable bound.

Identifying a lip synchronization issue based on the packet arrival time is not trivial. Components like the Jitter Buffers may add delay allowing the encoder to keep the streams in synch.

You can calculate the relative ‘wall time’ for a given RTP packet by using its RTP timestamp and the RTP timestamp with the ‘Wall Time’ from a RTCP sender-report as a reference point. Comparing this time against the received time can provide latency for a given packet.
Performing these calculations for both the audio and video streams will allow you to compare how far apart the given streams may be arriving.

Codecs like G711 use a fixed sample rate, meaning each packet contains 160 samples at 8000MHz, or 20ms of audio. So it’s quite easy to determine if the RTP Time stamp is ahead or behind it’s expected arrival time.

Codecs with variable sample rates (like video) can be harder to estimate without the negotiated clock speeds; however, you can estimate it.

The follow table are values taken from RTCP packets

| wall time | rtp time | received time | delta wall time | delta rtp-time | delta received time |
|---|---|---|---|---|---|
| 00:47.8 | 3383724609 | 59:33.5 | | | |
| 00:49.0 | 3383836659 | 59:34.7 | 00:01.245 | 112050 | 00:01.245 |
| 00:49.1 | 3383838459 | 59:34.7 | 00:00.020 | 1800 | 00:00.019 |
| 00:49.1 | 3383842149 | 59:34.8 | 00:00.041 | 3690 | 00:00.041 |
| 00:49.1 | 3383845029 | 59:34.8 | 00:00.032 | 2880 | 00:00.032 |

We can see that a data of RTP timestamps equal to 1800 takes approximately 20ms and that's consistent, you can use this as a clock rate. You can estimate the expected arrival time for an RTP packet by:

1. calculating the difference between the RTP packet's timestamp and the last received RTCP packet's timestamp
2. Using the clock rate determine the number of ms since the RTCP packet arrived
3. Add this time to the wall clock time of the RTCP packet

With this information you can:

- Compare this value with the actual arrival time
- Compare the delay against another stream

# Captures taken from an iPAD

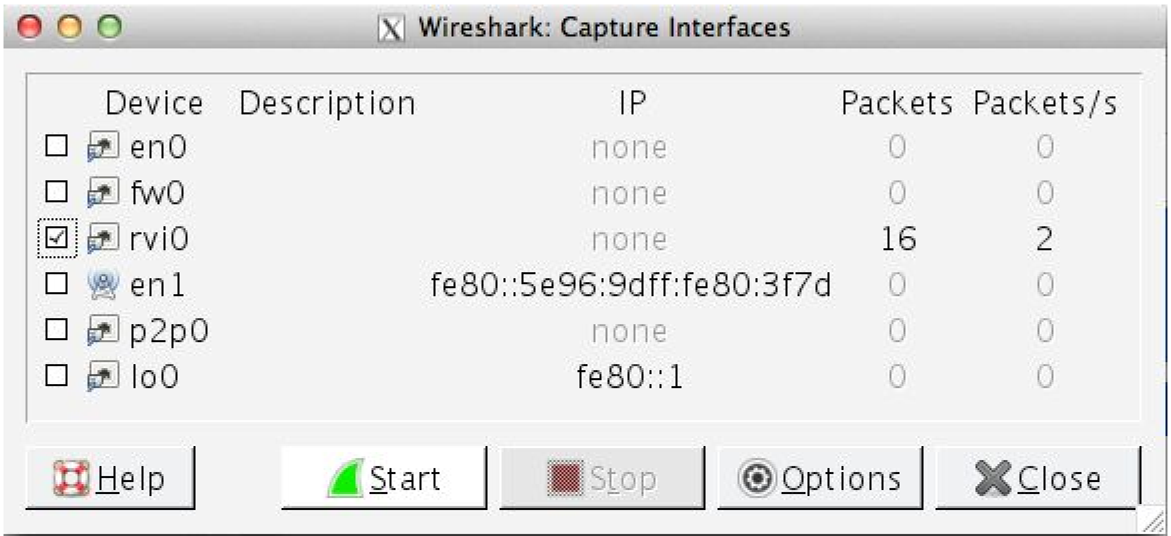
In order to measure the packet loss of the RTP stream from the Fusion Media Broker to a Fusion Enabled iOS Application the iPAD will need to be connected to a Macintosh with wireshark installed.  The following commands are taken from The Mac Developer Library (https://developer.apple.com/library/mac/qa/qa1176/_index.html#//apple_ref/doc/uid/DTS10001707-CH1-SECRVI) and they describe how the mac can be used used listen to wireless network traffic received by the ipad:

```
rvictl -s IPAD_UUID
```

Where IPAD_UUID is the *Identifier* for the IPAD, which can be obtained from XCode's Organizer view:


1. Taking a Capture using RVI
Using Wireshark there is now an additional Interface that can be monitored.  Make a call using the IPAD and allow the Call to establish and run for a few minutes, before stopping the capture and processing the results.

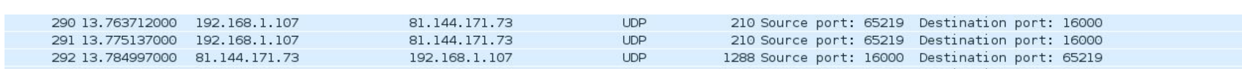
This capture will determine how much packet loss there is from the Fusion Media Broker to the Ipad.  This is because the UDP packets will either be missing or out of sequence.




2. Filtering Traffic

There will be four UDP streams in this capture: two inbound video and audio streams from the Fusion Media Broker to the ipad and another two outbound streams from the ipad to the Fusion Media Broker. It is necessary to hunt for a packet that is suspected to be from the video stream.

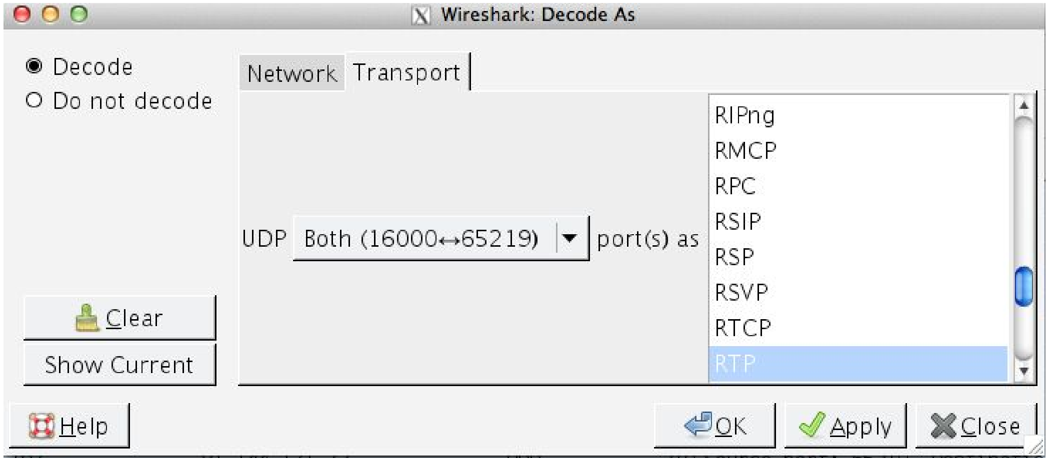
Apply the following filter: *udp && !rtp*



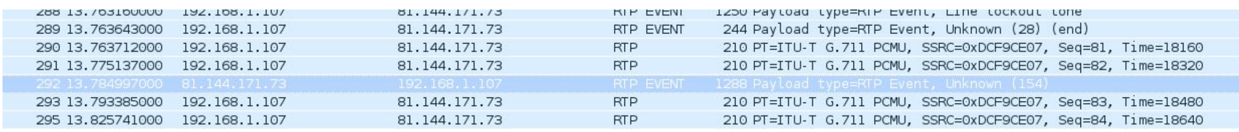

The Ipad will send to Destination Port: 16000, so it is likely that the first two packets belongs to the outbound stream. The third packet is likely to belong to the inbound stream.

3. Encoding the UDP Stream

Right Click a UDP packet from the Fusion Media Broker and Select: *Decode As*

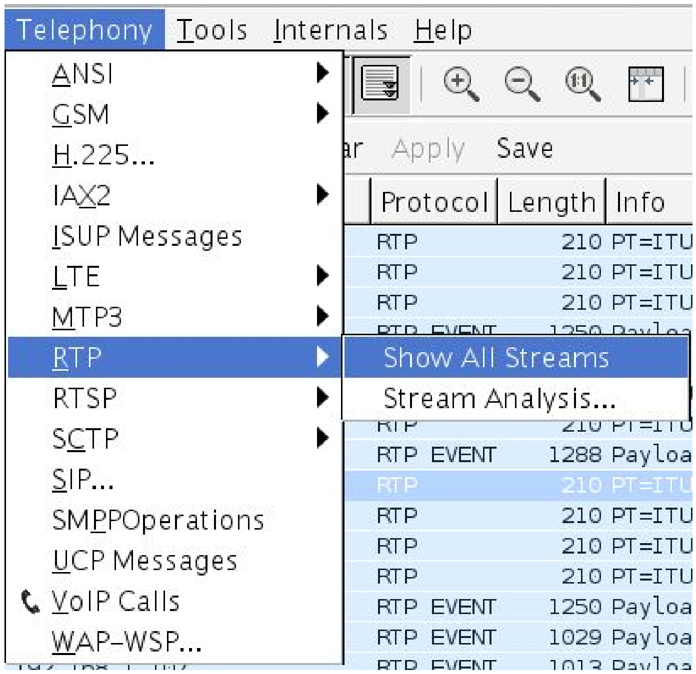



Select RTP and press Apply. Wireshark will now interpret the UDP stream as a full RTP stream. It should be noted that in this case a G.711 audio-packet was selected, but wireshark has also distinguished between inbound and outbound audio and video streams:

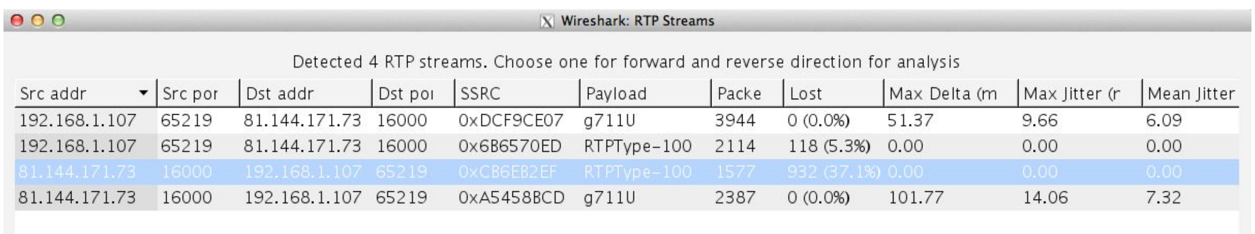



4. Analyze the Stream

Now that the streams have been formatted Select: *Telephony->RTP->Show All Streams*




The following will be displayed:




| Src addr | Src por | Dst addr | Dst poi | SSRC | Payload | Packe | Lost | Max Delta (m | Max Jitter (r | Mean Jitter |
|---|---|---|---|---|---|---|---|---|---|---|
| 192.168.1.107 | 65219 | 81.144.171.73 | 16000 | 0xDCF9CE07 | g711U | 3944 | 0 (0.0%) | 51.37 | 9.66 | 6.09 |
| 192.168.1.107 | 65219 | 81.144.171.73 | 16000 | 0x6B6570ED | RTPType-100 | 2114 | 118 (5.3%) | 0.00 | 0.00 | 0.00 |
| 81.144.171.73 | 16000 | 192.168.1.107 | 65219 | 0xCB6EB2EF | RTPType-100 | 1577 | 932 (37.1%) | 0.00 | 0.00 | 0.00 |
| 81.144.171.73 | 16000 | 192.168.1.107 | 65219 | 0xA5458BCD | g711U | 2387 | 0 (0.0%) | 101.77 | 14.06 | 7.32 |

Here it can be seen that the inbound and outbound video and audio streams. In this example 37% of packets were lost on the inbound stream to the ipad.

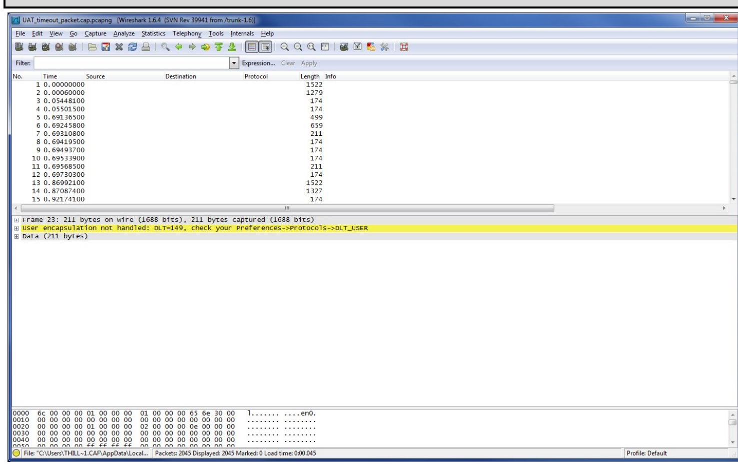
## RVI Captures Introduce Erroneous Skew Measurements

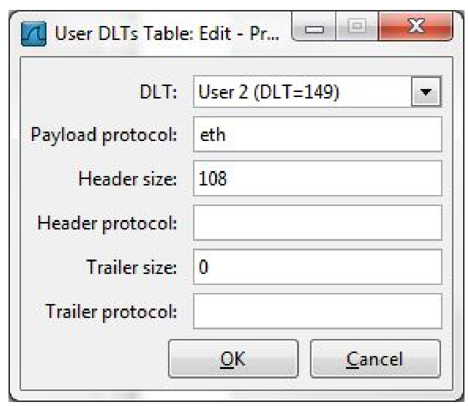
It must be remembered that the capture being taken using the RVI is not at the IPAD but the Mac which it is tethered to.  We have witnessed significant discrepancies in timings values taken using the RVI when compared to a network router.  Notably the skew values can be seconds out within a few minutes.  It is preferable to capture IPAD traces at a wifi access point or network router.

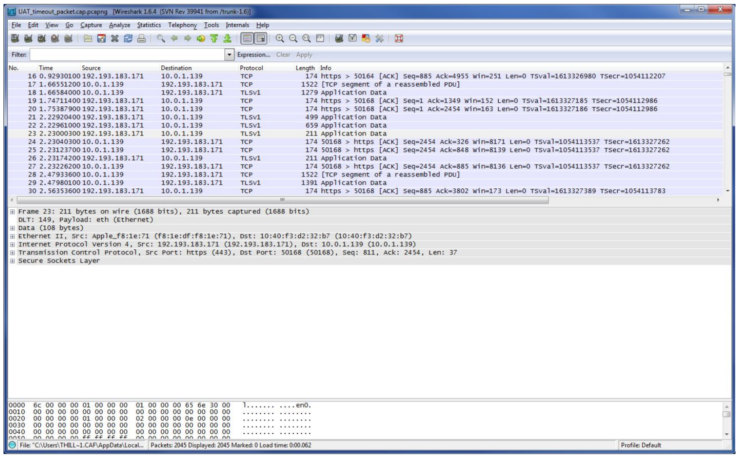
## Capture taken from a Mavericks Mac

Wireshark cannot understand the packets captured from Mavericks because Apple chose to use an unknown packet format.  This can be resolved by Opening *Edit→ Preferences* then *Protocols-->DLT_USER* and editing the Encapsulations Table to add the following Entry:

*DLT = User 2 (DLT=149)*
*Payload Protocol = eth*
*Header Size = 108*
*Header Protocol = <leave blank>*
*Trailer Size = 0*
*Trailer Protocol = <leave blank>*

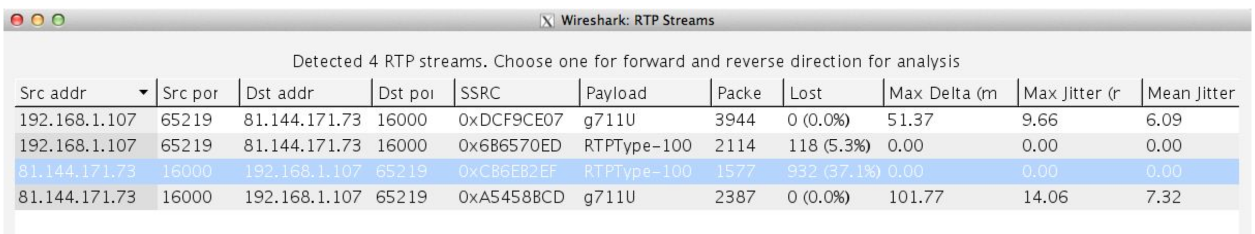






### iPAD capture shows Packet Loss on Outbound Stream

The capture used previously shows a 5.3% packet loss for the outbound stream.  It is not possible to measure outbound UDP packet loss (because the protocol is connectionless).



Wireshark: RTP Streams

Detected 4 RTP streams. Choose one for forward and reverse direction for analysis

| Src addr | Src por | Dst addr | Dst poi | SSRC | Payload | Packe | Lost | Max Delta (m | Max Jitter (r | Mean Jitter |
|---|---|---|---|---|---|---|---|---|---|---|
| 192.168.1.107 | 65219 | 81.144.171.73 | 16000 | 0xDCF9CE07 | g711U | 3944 | 0 (0.0%) | 51.37 | 9.66 | 6.09 |
| 192.168.1.107 | 65219 | 81.144.171.73 | 16000 | 0x6B6570ED | RTPType-100 | 2114 | 118 (5.3%) | 0.00 | 0.00 | 0.00 |
| 81.144.171.73 | 16000 | 192.168.1.107 | 65219 | 0xCB6EB2EF | RTPType-100 | 1577 | 932 (37.1%) | 0.00 | 0.00 | 0.00 |
| 81.144.171.73 | 16000 | 192.168.1.107 | 65219 | 0xA5458BCD | g711U | 2387 | 0 (0.0%) | 101.77 | 14.06 | 7.32 |

This implies that there was packet loss between the IPAD and the MAC performing the packet capture!  It is suspected this could be related to CPU load on the MAC but it implies that this could skew results because there could be similar levels of inbound packet loss.

Capturing at the local network's router has shown some packets appear as Comfort Noise initiating from the iPAD; this seems related to the wireshark decoding.

# Captures from a Browser

The analysis of the wireshark capture is identical to the previous section and will not be covered again. In addition, Chrome has some additional metrics which can be used:

When a call is active point the Chrome Browser to:

```
chrome://webrtc-internals/
```

When a call is active you can view the number of dropped packets:

**Statistics ssrc_447873670**

cname:rRl9FQ7Ylbid7KWN
msid:RUijbqmNT0lKkj9MicB1cb7nAlSTSKfilhaq a0
mslabel:RUijbqmNT0lKkj9MicB1cb7nAlSTSKfilhaq
label:RUijbqmNT0lKkj9MicB1cb7nAlSTSKfilhaqa0

| | |
|---|---|
| timestamp | Wed Oct 23 2013 13:40:06 GMT+0100 (BST) |
| ssrc | 447873670 |
| googTrackId | a0 |
| transportId | Channel-audio-1 |
| audioOutputLevel | 32 |
| bytesReceived | 671520 |
| googJitterReceived | 5 |
| packetsReceived | 4197 |
| packetsLost | 1 |

Web-rtc-internals-parameters from TestRTC, is a good tutorial for understanding the metrics in webrtc-internals:
https://testrtc.com/webrtc-internals-parameters/

# Bandwidth Estimates

From webrtc-internals the bandwidth estimates can tell you how the adaptive bitrate is behaving.


A video encoder has to make a lot of calculations based on how it perceives the network conditions and how the receiver is reporting theirs.

The following attributes are present:

- **googAvailableReceiveBandwidth**
    - the bandwidth that is available for receiving video data
- **googAvailableSendBandwidth**
    - the bandwidth that is available for sending video data
- **googTargetEncBitrate**
    - the target bitrate of the the video encoder, it will try and fill the available bandwidth

# Local Firewall Configuration

The local firewall configuration for Media broker must allow inbound UDP packets and initial outbound UDP packets for the media to establish.

The following is a sample setup for configuring the *firewalld* service to allow inbound traffic:

1. Install Firewalld:
   ```
   yum install firewalld
   ```
2. Add an interface to a public zone:
   ```
   sudo firewall-cmd --zone=public --permanent
   --change-interface=eno16777984
   ```
3. This can also be set in:
   ```
   vi /etc/sysconfig/network-scripts/ifcfg-eno16777984 ZONE=public
   ```
4. Configure the  Media Broker XML script:
   ```
   vi /etc/firewalld/services/csdk-mb.xml
   ```

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <service>
     <short>MB</short>
   <description>Service Description for Media Broker Service</description>
     <port protocol="udp" port="16000"/>
     <port protocol="udp" port="17000-17999"/>
     <port protocol="tcp" port="8092"/>
   </service>
   ```

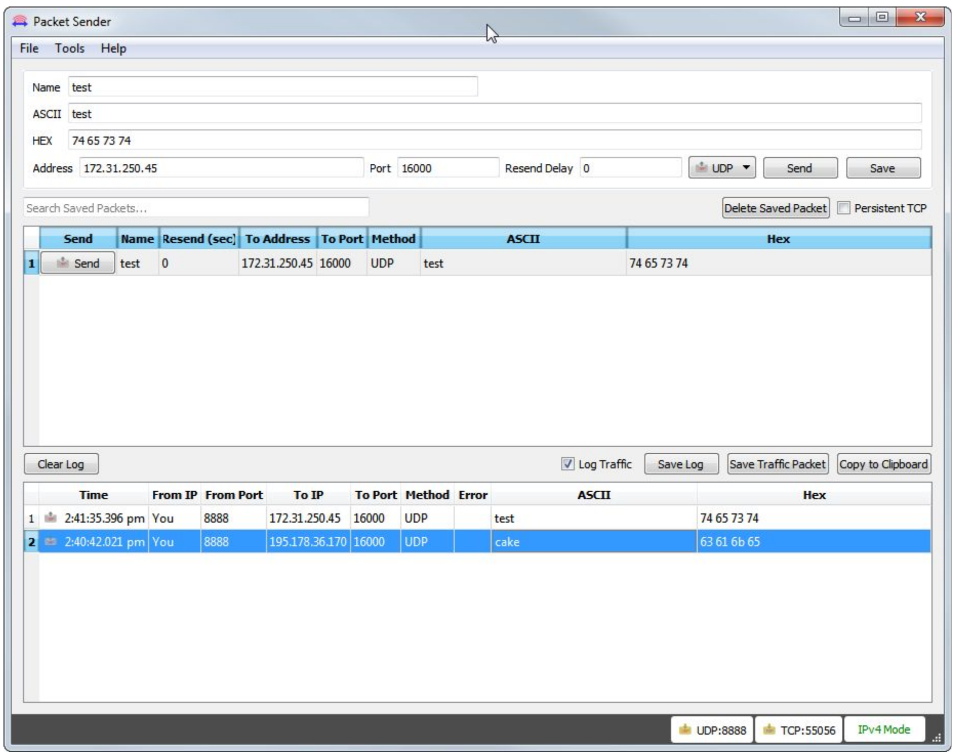
5. Reload to see new services:
   ```
   sudo firewall-cmd --reload
   ```
6. Apply Services to Zones:
   ```
   sudo firewall-cmd --zone=public --permanent --add-service=csdk-mb
   ```

# Testing the Local Firewall Ports

You can test if packets can be sent from a public network through to the Media Broker by using a packet sending tool.

Tools like *Packet Sender* ([https://packetsender.com](https://packetsender.com)) allow you to craft UDP packets and direct them to an IP and Port.

Using *tcpdump* on the Media Broker port you can listen for inbound traffic and determine if your sent packets arrive.

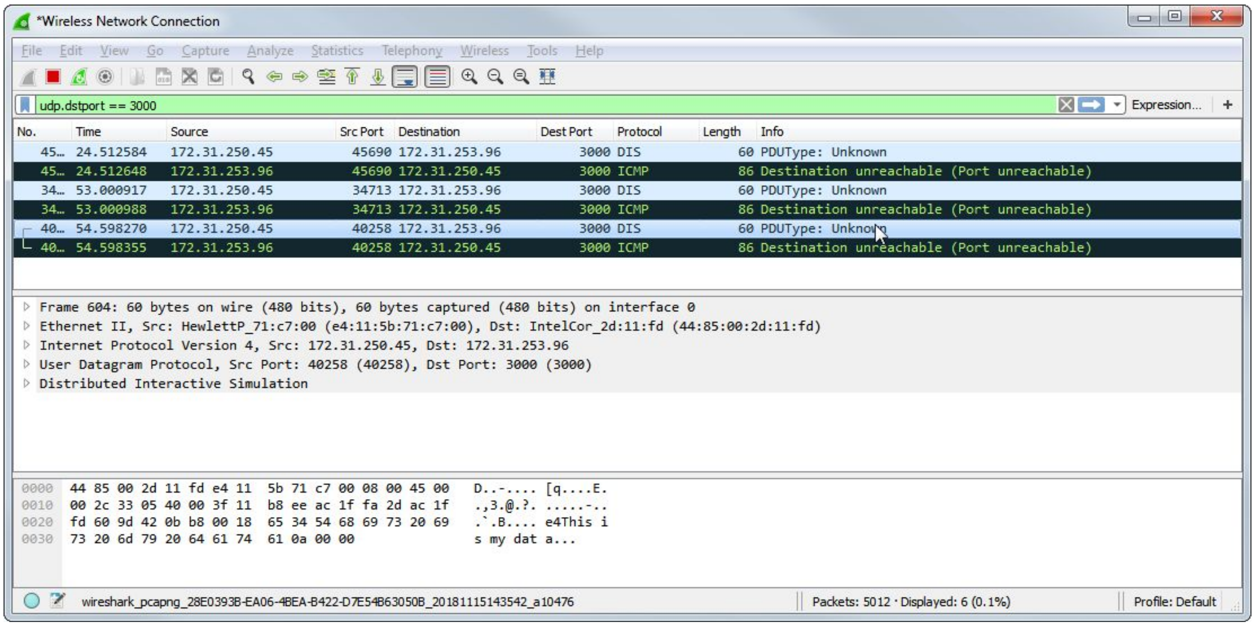



The following filter can be used to only show UDP packets arriving at port 16000:

**tcpdump -i any udp port 16000**

```
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked), capture size 262144 bytes
14:28:32.936669 IP 172.31.253.96.ddi-udp-1 > centos71.16000: UDP, length 4
```

A very simply way of sending UDP traffic from the Media Broker machine is to use the following command, specifying the destination address:
**echo "This is my data" > /dev/udp/{{DESTINATION ADDRESS}}/3000**




Obviously, the destination address will need to be publicly routable, so if your test machine is behind a NAT, this technique will not work. Suitable knowledge of your network's routing rules are required. It may be easier to verify simply the inbound and outbound STUN by establishing a call.