



GIAC

全球互联网架构大会

GLOBAL INTERNET ARCHITECTURE CONFERENCE

七牛 WebRTC 连麦服务端架构实践

七牛云直播流媒体负责人 谢然

# 100

## 第七届 全球软件案例研究峰会

运营增长

组织发展

爆款架构

人工智能

测试实践

运维体系

产品创新

研发效能

数据平台

AI/数据驱动

测试体系

AIOps DevOps

体验设计

团队管理

工程实践

AI实践

大前端

区块链



扫描二维码

查看最值得学习的100+创新案例

2018年11月30日-12月3日

北京 | 国家会议中心

主办方 msup®



## 目录

- 使用 licode 的传输层实现连麦 SFU
  - licode 架构
  - 剥离传输层
  - ExternalOutput
- 使用 mediasoup 的传输层实现连麦 SFU
  - mediasoup 架构
  - mediasoup VS licode
- SFU 相关优化
  - 多 PeerConnection VS 单 PeerConnection
  - ICE 建立连接优化



## 目录

- 使用 WebRTC/ffmpeg 实现服务端合流
  - WebRTC 相关修改
  - GPU 加速
- 边缘透明加速
  - ICE 概述
  - 实现原理

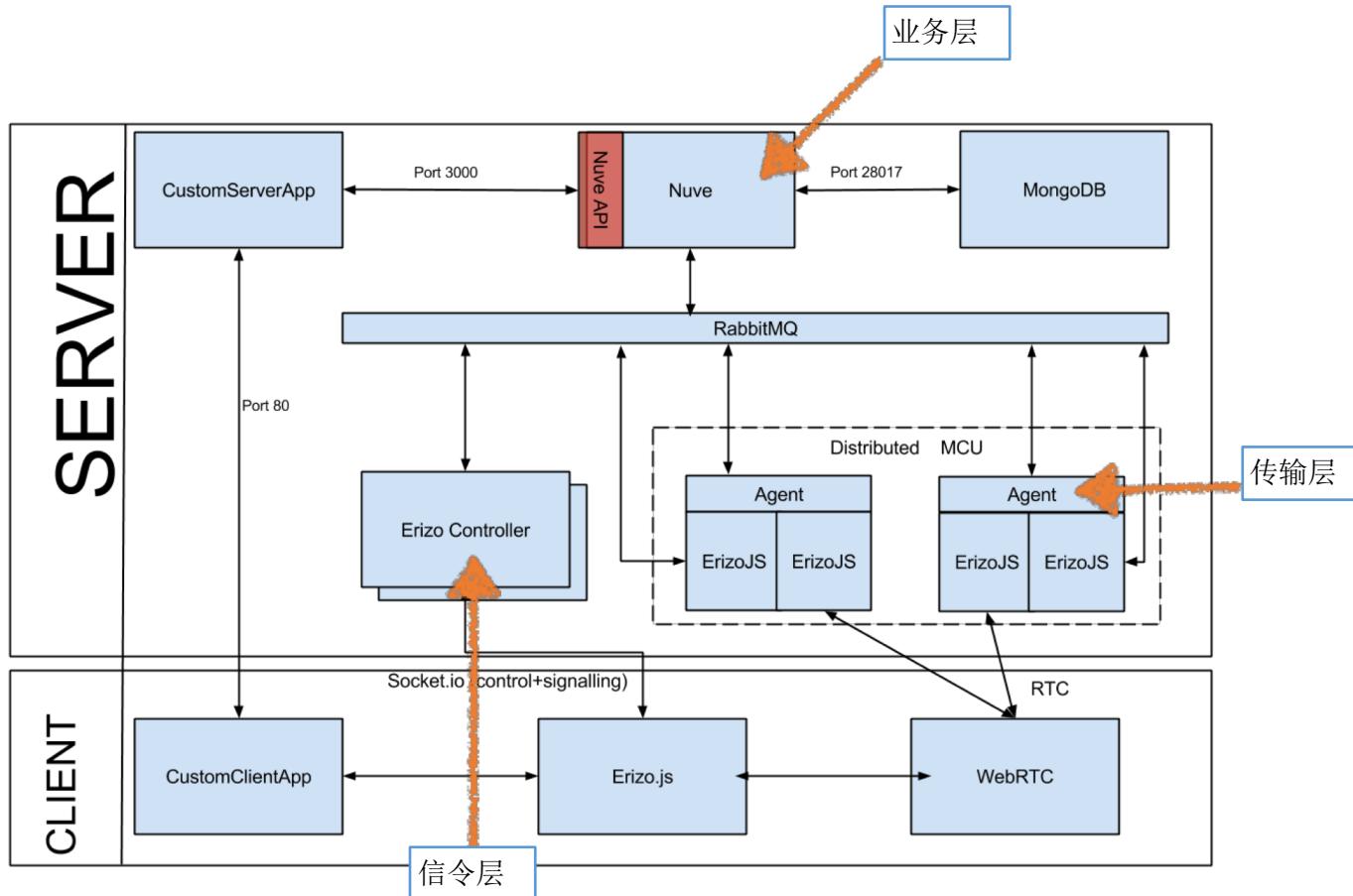


## 使用 licode 构建连麦服务端

- 连麦架构
  - 业务层 (用户管理, 房间管理)
  - 信令层 (房间内部消息, ICE)
  - 传输层 (RTP/RTCP)



# 使用 licode 构建连麦服务端





## 使用 licode 构建连麦服务端

- licode 传输层架构
  - libnice (IO)
  - openssl (DTLS)
  - libsrtplib (RTP)
  - Pipeline (RTP)
  - OneToManyProcessor (RTP)



## 使用 licode 构建连麦服务端

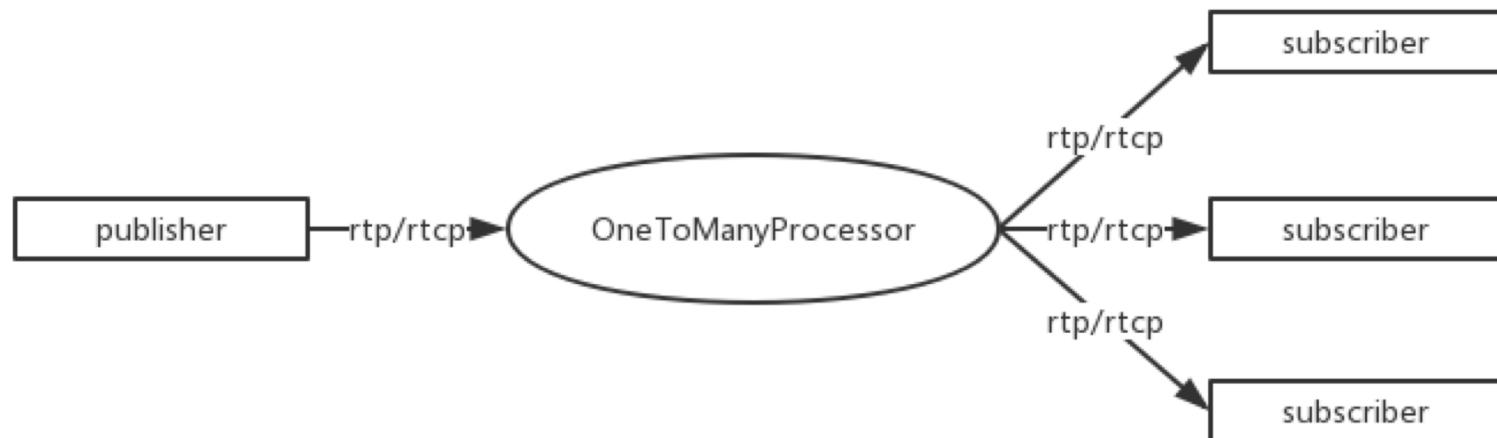
- Pipeline
  - 按一定顺序处理 RTP 包

```
pipeline_->addFront(PacketReader(this));
pipeline_->addFront(RtpDumpHandler(this, false));
pipeline_->addFront(LayerDetectorHandler());
pipeline_->addFront(RtcpProcessorHandler());
pipeline_->addFront(FecReceiverHandler());
pipeline_->addFront(LayerBitrateCalculationHandler());
pipeline_->addFront(QualityFilterHandler());
pipeline_->addFront(IncomingStatsHandler());
pipeline_->addFront(RtpTrackMuteHandler());
pipeline_->addFront(RtpSlideShowHandler());
pipeline_->addFront(RtpPaddingGeneratorHandler());
pipeline_->addFront(PliPacerHandler());
pipeline_->addFront(BandwidthEstimationHandler());
pipeline_->addFront(RtpPaddingRemovalHandler());
pipeline_->addFront(RtcpFeedbackGenerationHandler());
pipeline_->addFront(RtpRetransmissionHandler());
pipeline_->addFront(SRPacketHandler());
pipeline_->addFront(SenderBandwidthEstimationHandler());
pipeline_->addFront(OutgoingStatsHandler());
pipeline_->addFront(RtpDumpHandler(this, true));
pipeline_->addFront(PacketWriter(this));
```



## 使用 licode 构建连麦服务端

- OneToManyProcessor





## 使用 licode 构建连麦服务端

- 剥离传输层
  - erizo\_controller/erizoJS (去掉)
    - 上层业务逻辑
  - erizoAPI (保留 or 去掉)
    - JS/C++ 胶合层
  - erizo (保留)
    - C++ 核心部分

```
▷ .circleci
▷ .github
▷ .vscode
▷ build
▷ cert
▷ doc
▷ erizo
▷ erizo_controller
▷ erizoAPI
▷ erizod
▷ extras
▷ feature-review
▷ node_modules
▷ nuve
▷ rtpdump
▷ scripts
▷ spine
▷ test
▷ utils
```



## 使用 licode 构建连麦服务端

- 定制微服务接口
  - new-conn={ "ice\_servers": [ { "urls": [ "stun:..." ] }, ] }
  - set-remote-desc-create-answer={ "id": "xx", "sdp": "xx" }
  - on-ice-candidate={ "id": "xx", "candidate": "xx" }
  - add-ice-candidate={ "id": "xx", "candidate": "xx" }
  - licode-new-one-to-many-processor={ }
  - licode-one-to-many-processor-set-pub={ "id": "xx", "pub": "xx" }
  - licode-one-to-many-processor-add-sub={ "id": "xx", "sub": "xx" }
- 参考官方 Javascript API 以及结合 licode 自己的 API



## 使用 licode 构建连麦服务端

- ExternalOutput/ExternalInput
  - 集成 FFmpeg
    - RTP <-> ffmpeg AVPacket
  - 加在 OneToManyProcessor 下
  - 作为 Pub/Sub
  - 可以实现 RTC<->RTMP 互转



## 使用 mediasoup 构建连麦服务端

- mediasoup 架构
  - libuv (IO)
  - openssl (DTLS)
  - libsrtplib (RTP)
  - Producer/Consumer (RTP)



## 使用 mediasoup 构建连麦服务端

- mediasoup 架构
  - RtpSendStream/RtpRecvStream
    - NACK/RTCP 相关
  - WebRTCTransport
    - ICE 相关
    - 调用底层库完成传输
  - Router/Worker
    - C++/JS 胶合层
    - Pub/Sub 逻辑
  - NodeJS
    - 较简单的房间逻辑



## mediasoup VS licode

- licode
  - IO 库性能不佳
    - libnice 基于 gio
    - 为桌面系统设计，没有为高并发考虑
  - 上层业务逻辑完善
    - 会议，信令，分布式部署
  - 整合了 FFmpeg
    - ExternalOutput/Input
    - 便于实现 RTMP<->RTC 互转



## mediasoup VS licode

- mediasoup
  - IO 库性能好
    - libuv
    - 为高并发的 IO 库
  - 代码结构清晰
  - 没有上层业务逻辑
    - 不支持房间逻辑
  - 便于集成
    - C++/JS 协议
  - 发展较快



## SFU 相关优化

- 多 PeerConnection
  - 每路流都有一个 PeerConnection
- 单 PeerConnection
  - 多路流只有一个 PeerConnection
  - 和 Server 之间只有一个 PeerConnection
  - 减少 PeerConnection 创建交互
  - 增加 Track 过程
    - 把 track 信息放入 SDP 中
    - SetRemoteDescription({type:offer, sdp:sdp})



## SFU 相关优化

- 优化 candidate 交互过程
  - C<->S 模式只需要 host 类型的 candidate
    - 不需要 STUN Server
  - 把 candidate 填到 SDP 里
    - 不需要使用 addRemoteCandidate
    - a=candidate:udpcandidate ...



## 使用 WebRTC/ffmpeg 实现服务端合流

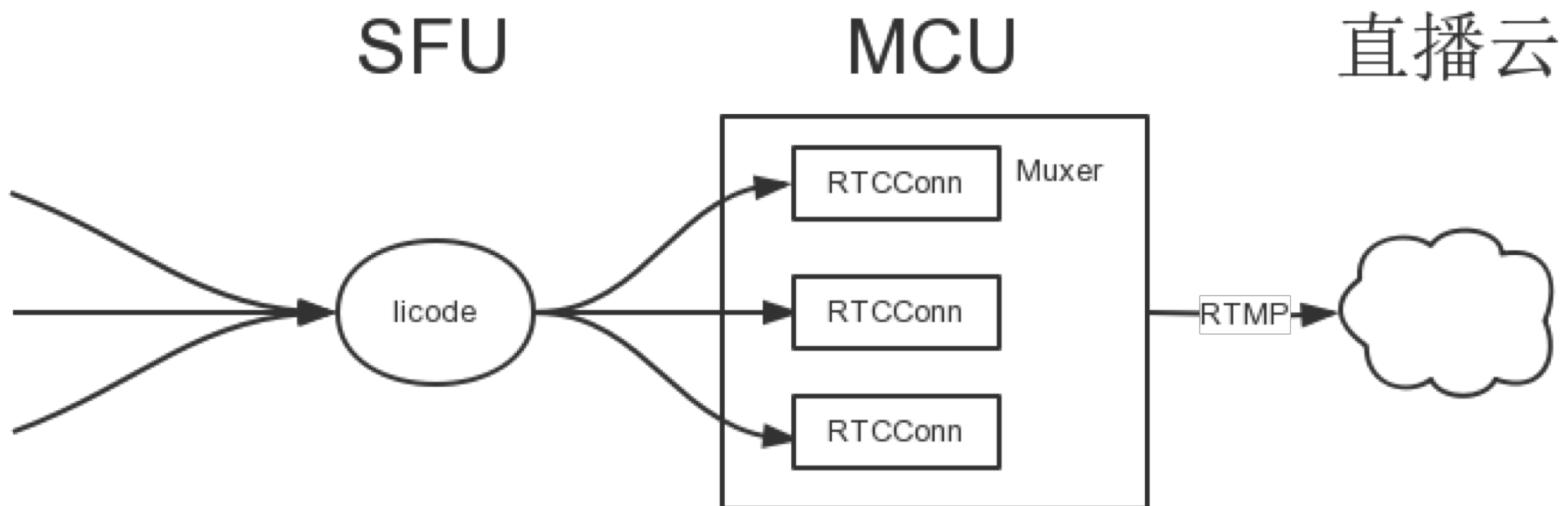
THE SFU      THE MCU





## 使用 WebRTC/ffmpeg 实现服务端合流

- 七牛服务端合流架构





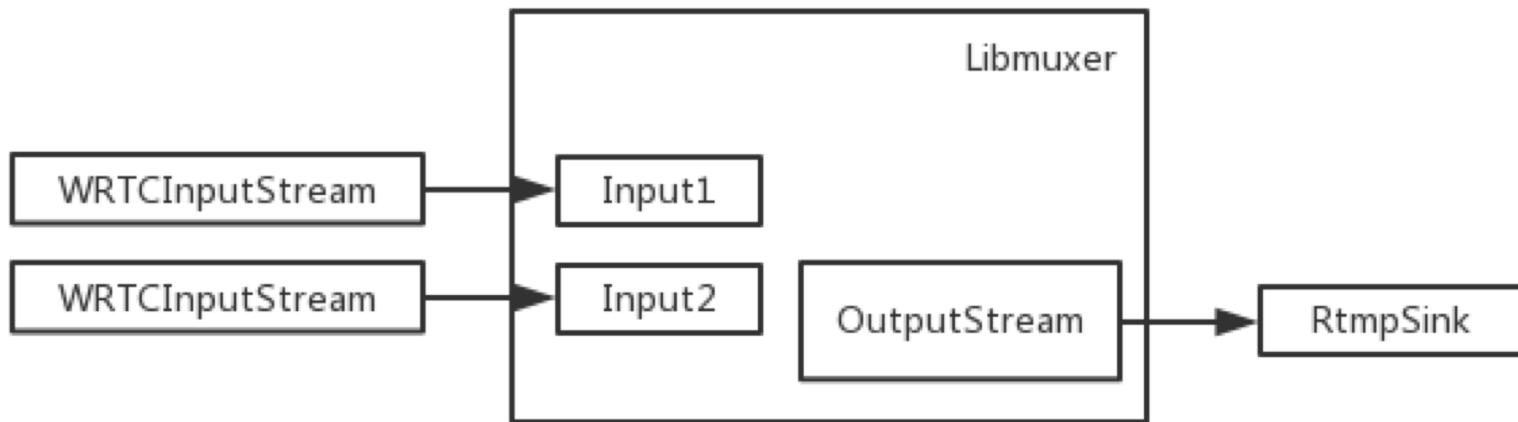
## 使用 WebRTC/ffmpeg 实现服务端合流

- 合流流媒体层设计
  - 组件化
    - 灵活
    - 方便写业务
    - 便于调试定位问题
    - 类 gstreamer 架构
  - Go/C++ 分层
    - Go 处理较为复杂的信令交互
    - C++ 处理流媒体相关



## 使用 WebRTC/ffmpeg 实现服务端合流

- 合流流媒体层设计





## 使用 WebRTC/ffmpeg 实现服务端合流

- 合流流媒体层设计
  - Stream -> Sink
    - 一个 Stream 可以添加多个 Sink
    - 传递 AVFrame, 裸数据
  - Stream
    - WRTCStream (WebRTC 输入)
    - FileInputStream (文件输入)
    - RtmpInputStream (RTMP 输入)
  - Sink
    - WRTCStream (WebRTC 输出)
    - RtmpSink (RTMP 输出)
    - FileSink (FLV 文件输出)



## 使用 WebRTC/ffmpeg 实现服务端合流

- 合流流媒体层设计
  - FileInputStream/RtmpInputStream
    - libRTMP
      - 增加对 FLV 文件的支持
    - FFmpeg h264 decoder
    - libx264
    - libfdk-aac



## 使用 WebRTC/ffmpeg 实现服务端合流

- 合流流媒体层设计
  - Libmuxer
    - 输入为多个 Stream
    - 输出为多个 Sink
    - 画面混合
      - libswscale 画面拉伸
      - AVFrame YUV 数据叠加
      - 支持 Alpha 通道
    - 水印
      - 解码为 AVFrame YUVA 格式
      - 同视频处理



## 使用 WebRTC/ffmpeg 实现服务端合流

- 合流流媒体层设计
  - Libmuxer
    - 声音混合
      - 先做 resample 统一规格
      - libavresample
      - 每个包 1024 samples
      - 每个 Stream 的队列最多缓存 8 个包
      - 队列为空则认为静音



## 使用 WebRTC/ffmpeg 实现服务端合流

- WebRTC 部分技术选型
  - WebRTC 官方 VS licode client
    - licode 支持的传输层特性不全
    - WebRTC 官方解码兼容性最好



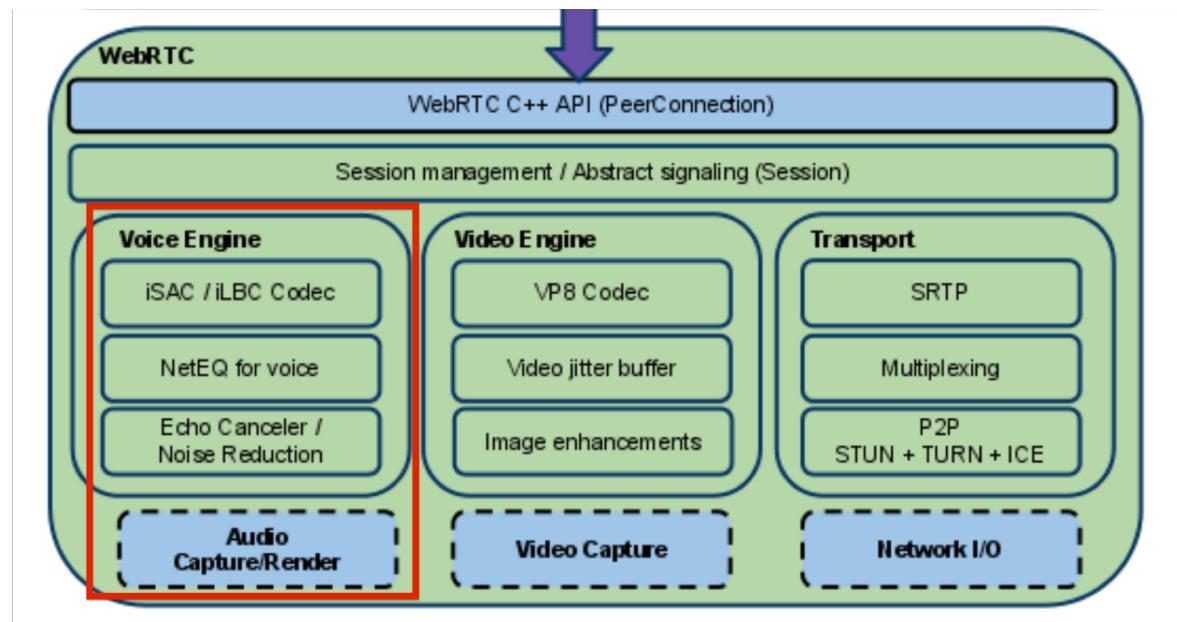
## 使用 WebRTC/ffmpeg 实现服务端合流

- WebRTC 相关代码修改
  - 对音频处理部分的修改
  - ffmpeg 编译选项的修改



## 使用 WebRTC/ffmpeg 实现服务端合流

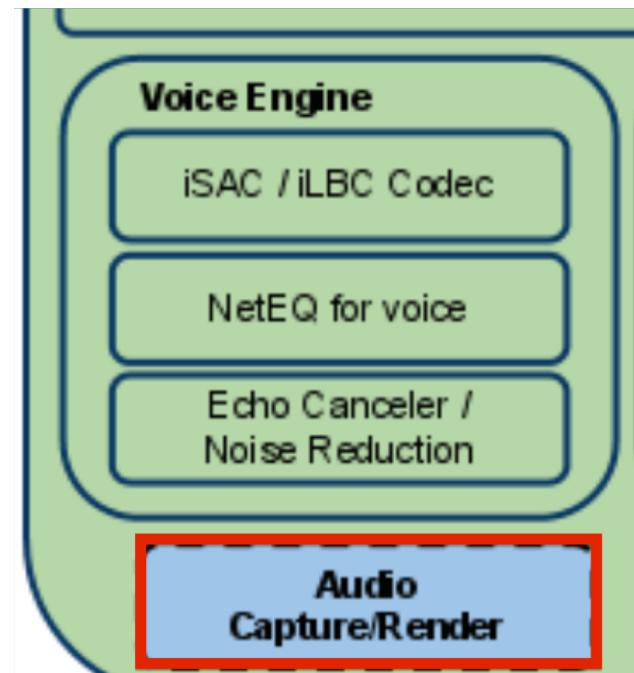
- 对音频处理部分的修改
  - WebRTC 官方仅针对客户端
  - 非服务端需要





## 使用 WebRTC/ffmpeg 实现服务端合流

- 对音频处理部分的修改
  - Audio Capture/Render
    - 被 Voice Engine 依赖
    - 依赖平台的音频驱动
    - 服务器无法使用音频驱动
    - 需要实现 FakeAudioDevice
  - Voice Engine
    - 单输入
    - 较难剥离





## 使用 WebRTC/ffmpeg 实现服务端合流

- 对音频处理部分的修改
  - 实现 FakeAudioDevice
    - modules/audio\_device/include/audio\_device.h
    - RegisterAudioCallback()
    - StartPlayout() / Playing()
    - StartRecording() / Recording()
  - 参考现有的 AudioDevice 来实现 FakeAudioDevice



## 使用 WebRTC/ffmpeg 实现服务端合流

- ffmpeg 编译选项的修改
  - 原生仅支持 openh264 decoder (LICENSE 原因)
  - 换成 ffmpeg 自带的 h264 decoder
  - 换成硬件加速的 h264 decoder
  - 增加 format/decoder/encoder 用于调试



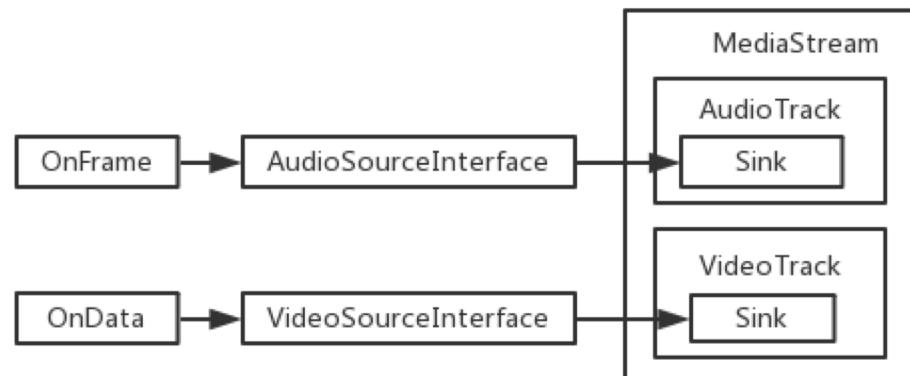
## 使用 WebRTC/ffmpeg 实现服务端合流

- ffmpeg 编译选项的修改
  - 特殊的编译方式
  - 生成 ffmpeg\_generated.gni
  - chromium/scripts/build\_ffmpeg.py
  - chromium/scripts/generate\_gn.py
  - chromium/scripts/copy\_config.sh



## 使用 WebRTC/ffmpeg 实现服务端合流

- WebRTC 如何推流
  - 创建 AudioSourceInterface / VideoSourceInterface
  - 创建 AudioTrackInterface / VideoTrackInterface
  - 创建 CreateLocalMediaStream
  - 调用 Source 的 OnFrame / OnData 填数据
  - OnData 音频 frame size 必须是 10ms





## 使用 WebRTC/ffmpeg 实现服务端合流

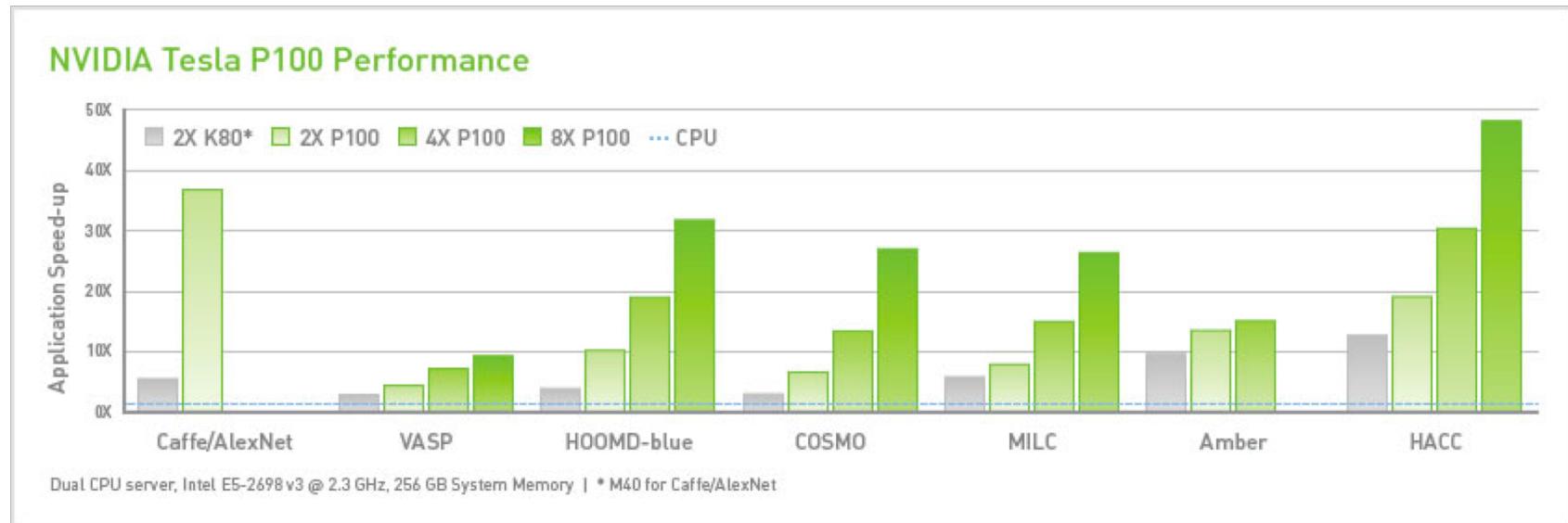
- 使用 GPU 加速 x264 编码
  - Nvidia Tesla 平台介绍





## 使用 WebRTC/ffmpeg 实现服务端合流

- 使用 GPU 加速 x264 编码
  - Nvidia Tesla 平台介绍





## 使用 WebRTC/ffmpeg 实现服务端合流

- 使用 GPU 加速 x264 编码
  - Nvidia Tesla 平台介绍





## 使用 WebRTC/ffmpeg 实现服务端合流

- 在 ffmpeg 中使用 Nvidia Tesla 方案
  - <https://developer.nvidia.com/ffmpeg>

FFmpeg GPU HW-Acceleration Support Table

	Fermi	Kepler	Maxwell (1st Gen)	Maxwell (2nd Gen)	Maxwell (GM206)	Pascal
H.264 encoding	N/A	FFmpeg v3.3	FFmpeg v3.3	FFmpeg v3.3	FFmpeg v3.3	FFmpeg v3.3
HEVC encoding	N/A	N/A	N/A	FFmpeg v3.3	FFmpeg v3.3	FFmpeg v3.3
MPEG2, MPEG-4, H.264 decoding	FFmpeg v3.3	FFmpeg v3.3	FFmpeg v3.3	FFmpeg v3.3	FFmpeg v3.3	FFmpeg v3.3
HEVC decoding	N/A	N/A	N/A	N/A	FFmpeg v3.3	FFmpeg v3.3
VP9 decoding	N/A	N/A	N/A	FFmpeg v3.3	FFmpeg v3.3	FFmpeg v3.3



## 使用 WebRTC/ffmpeg 实现服务端合流

- 在 ffmpeg 中使用 Nvidia Tesla 方案
  - ./configure --enable-cuda --enable-cuvid --enable-nvenc --enable-nonfree --enable-libnpp —extra-cflags=-I/usr/local/cuda/include —extra-ldflags=-L/usr/local/cuda/lib64
  - 以闭源库方式提供



## 使用 WebRTC/ffmpeg 实现服务端合流

- CPU vs GPU 结果对比 (CPU)
  - 高分辨率高画质占用 CPU 较多
  - CPU 占用率波动大
  - 内存占用率稳定

规格/输出	CPU	MEM
libx264_faster_480P	95%	106Mb
libx264_faster_720P	160%	122Mb
libx264_faster_1080P	250%	150Mb
libx264_veryfast_480P	80%	104Mb
libx264_veryfast_720P	115%	118Mb
libx264_veryfast_1080P	160%	142Mb
libx264_ultrafast_480P	58%	102Mb
libx264_ultrafast_720P	77%	112Mb
libx264_ultrafast_1080P	110%	128Mb



## 使用 WebRTC/ffmpeg 实现服务端合流

- CPU vs GPU 结果对比 (GPU)
  - GPU 占用率在 20%~35%
  - CPU 占用率稳定在 35%~55%
  - 内存 & 显存占用较多
  - 总显存为 8G

规格/输出	CPU	MEM	DISP MEM
nvenc_medium_1080P	55%	435Mb	279Mb
nvenc_medium_720P	42%	405Mb	230Mb
nvenc_medium_480P	36%	256Mb	170Mb
nvenc_fast_1080P	55%	430Mb	279Mb
nvenc_fast_720P	42%	375Mb	230Mb
nvenc_fast_480P	36%	256Mb	170Mb



## 使用 WebRTC/ffmpeg 实现服务端合流

- CPU vs GPU 结果对比（结论）
  - 越高分辨率高画质的编码 GPU 优势越明显



## 实现边缘透明加速

- 目的
  - 保障各地区边缘用户正常接入连麦
    - 加速传输层 (udp)
    - 加速信令层 (tcp)



## 实现边缘透明加速

- ICE 简介

- <https://tools.ietf.org/html/rfc5245>
- candidate 的作用
- 控制 candidate 下发机制以实现加速
  - 提前分配路径
  - 更改 candidate 端口

```
a=candidate:1467250027 1 udp 2122260223 192.168.0.196 46243 typ host generation 0
```

```
a=candidate:1467250027 2 udp 2122260222 192.168.0.196 56280 typ host generation 0
```



## 实现边缘透明加速

- 正常路径 vs 边缘加速路径



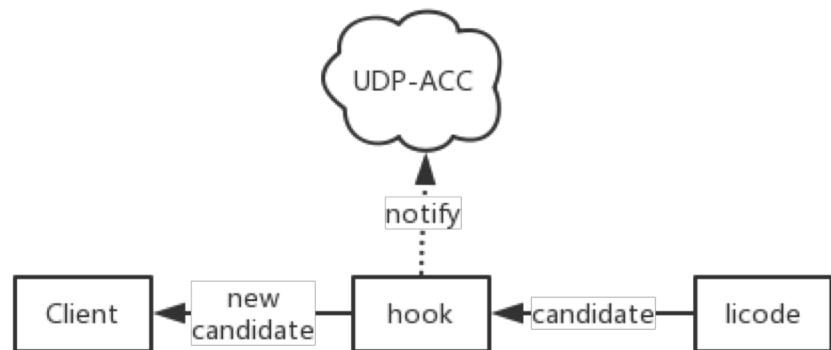


## 实现边缘透明加速

- candidate 下发：正常 vs 加速



正常

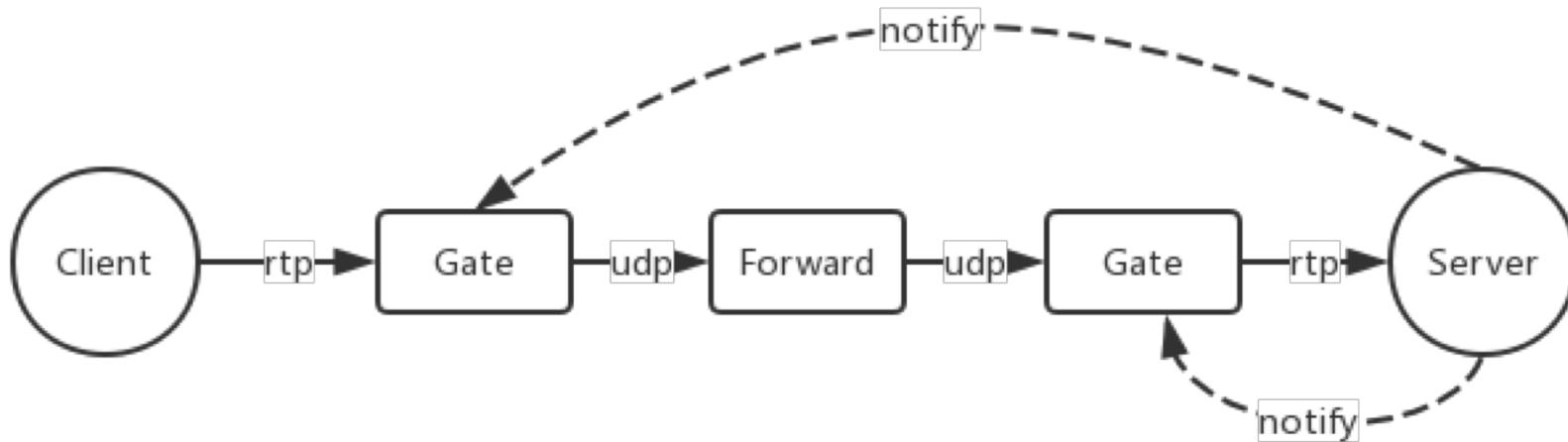


加速



## 实现边缘透明加速

- UDP-ACC 架构





## 实现边缘透明加速

- UDP-ACC 接入
  - Server (SFU) 收到 Client 请求
  - 根据 Client IP 选择合适的 Gate
  - 通知自己的 Gate 和 Client 的 Gate
  - 通知 Client 发送数据



## 实现边缘透明加速

- UDP-ACC 架构
  - Gate
    - WebRTC Client 接入点
    - RTP NACK
  - Forward
    - 无状态转发
    - 中心下发路由表



全球互联网架构大会  
GLOBAL INTERNET ARCHITECTURE CONFERENCE

Thank You !

- [github.com/nareix](https://github.com/nareix)
- [xieran@qiniu.com](mailto:xieran@qiniu.com)
- 18680581024



关注公众号获得  
更多案例实践