对Janus的优化设计

Janus Gateway是一个webrtc的server，一种插件式架构，基于这个架构，开发很丰富的插件，比如streaming，SIP，videoroom, audiobridge等，甚至支持lua，javascrip等脚本性语言去处理会话和音视频内容等等，毫无疑问是个集大成者。

那我们原生的Janus的设计存在那些问题呢？能支撑多少并发呢？

我们目前的业务主要用到了videoroom和audiobridge及lua三个插件，在8核16G并且开启loop=8（ice loop的feature，及RTP/RTCP/Data的收发开启了线程池的模式）的环境配置下：

1. audiobridge支持的并发是300路64K码率的音频，这个跟他们的专业测评是一样的，请参考https://core.ac.uk/download/pdf/74316352.pdf，且延迟在300路时达到了500毫秒左右；
2. 测试videoroom的时候，我们设置的脚本是每隔60毫秒加入一路视频，到1000路之前，每次都会core dump，无法继续压测下去；

以上是性能压测的一个结果，实际上我们在继续研究它的设计时，也有一些发现，好的方面就不详述了，大致描述下不足，或者说我个人认为的不足：

1. 函数代码过长，甚至有多个函数代码超过2000行，这让二次开发和维护的困难较大，第一次开发个静音通知的业务需求时，足足看了3天的代码，最后还存在一个BUG，就是重复通知导致APP重复显示联系人，可以这么说在国内很多公司都不太可能出现这样的状况；
2. 插件间存在大量冗余的代码，甚至有些业务的实现还不一致，比如token的业务，audiobridge和textroom、videoroom的实现不一致，且三个插件里重复代码；
3. 基于libnice和janus都是基于jlib开发的，jlib的对象对内存是有缓存的，这就给内存方面的BUG定位带来了困难，比如内存泄漏；
4. libnice和janus的事件循环是基于jlib的GMainContext和GMainLoop的，在它们之上实现事件源的接口就可以收发事件了：

```
struct GSourceFuncs {
  gboolean (*prepare)  (GSource    *source,
                        gint       *timeout_);
  gboolean (*check)    (GSource    *source);
  gboolean (*dispatch) (GSource    *source,
                        GSourceFunc callback,
                        gpointer    user_data);
  void     (*finalize) (GSource    *source); /* Can be NULL */
};
```

这样的一个实现意味着你如果想要对事件源进行操作就必须挨个对每个事件调用prepare函数，这显然是低效的，而且该事件循环基于poll实现，poll的性能明显不如epoll。

5. 并发模型采用线程的模式，线程间使用mutex同步原语的GAsyncQueue；janus core使用线程池，ice loop使用线程池，每个插件创建handle线程去处理接口，audiobridge的每个房间的创建一个mixer线程去混音并且每个participant都创建一个participant线程去opus编码混音后的数据，更不可接受的是mixer线程调用sleep去间隔性取音频数据解码并混音；所有的这些线程关系导致几乎每个对象每个消息处理都需要使用mutex或者原子操作去同步，这并不是一个好的并发模型。
6. 在处理SDP的offer和answer时使用sleep这个野蛮的方式等待收集地址完成的callback，并发量不大的情况下还可接受，但是如果较大并发性接入服务器，会出现部分客户端接入时因ICE过程而失败。

```
3458:     if(!updating && !janus_ice_is_full_trickle_enabled()) {
3459:         /* Wait for candidates-done callback */
3460:         while(ice_handle->cdone < 1) {
3461:             if(ice_handle == NULL || janus_flags_is_set(&ice_handle->webrtc_flags, JANUS_ICE_HANDLE_WEBRTC_STOP)
3462:                     || janus_flags_is_set(&ice_handle->webrtc_flags, JANUS_ICE_HANDLE_WEBRTC_ALERT)) {
3463:                 JANUS_LOG(LOG_WARN, "[%"SCNu64"] Handle detached or PC closed, giving up...!\n", ice_handle ? ice_handle->handle_id : 0);
3464:                 janus_sdp_destroy(parsed_sdp);
3465:                 return NULL;
3466:             }
3467:             JANUS_LOG(LOG_VERB, "[%"SCNu64"] Waiting for candidates-done callback...\n", ice_handle->handle_id);
3468:             g_usleep(10000);
3469:             if(ice_handle->cdone < 0) {
3470:                 JANUS_LOG(LOG_ERR, "[%"SCNu64"] Error gathering candidates!\n", ice_handle->handle_id);
3471:                 janus_sdp_destroy(parsed_sdp);
3472:                 return NULL;
3473:             }
3474:         }
3475:     }
```

而且这段代码还很不好重构。

以上种种问题导致Janus的author Lorenzo Miniero 也会感叹"I'm getting older but, unlike whisky, I'm not getting any better Author of the Janus WebRTC Server, and the only viking with no muscles"，当然他很谦卑，也在自嘲，但是的确我们在优化的时候碰到很多的困难，甚至有些莫名其妙。

针对以上问题，我们是如何做优化的？

针对事件循环的问题，我们在libnice封装基于epoll进行了封装，替换了poll，并且基于epoll封装了定时器，定时器像nginx一样是基于红黑树的：