一种保证初始数据完整性的实时视频流分发插件①

张 凯

(苏州科达科技股份有限公司 上海研发中心, 上海 201103)

摘 要: 在视频监控联网系统中进行实时监控时, 如果监控客户端收到的初始视频数据不完整会出现初始播放画面花屏的问题. 基于开源多媒体框架 GStreamer, 针对 H.264 编码标准, 设计并实现了一种实时视频流分发插件. 该插件使用随机创建的请求型 source 衬垫用于视频数据的分发, 并通过缓存当前 IDR 帧组的方法确保发送初始视频数据的完整性. 该插件可应用于流媒体服务器当中, 解决实时监控时初始画面花屏的问题. 实验结果表明, 应用该插件的流媒体服务器能够高效分发实时视频数据并保证初始播放画面完整, 对提升实时监控效果有着明显的作用.

关键词: 视频监控; 流媒体服务器; GStreamer 框架; H.264 标准; IDR 帧; RTP 协议

Real Time Video Stream Dispatch Plugin for Ensuring the Integrity of Initial Data

ZHANG Kai

(Suzhou Keda Technology Co. Ltd., Shanghai R & D Center, Shanghai 201103, China)

Abstract: In the real time video surveillance network system, if the initial video data received by the monitoring client is not complete, the screen will show blurred screen. For H.264 coding standard, this paper designs and implements real time video stream dispatch plugin based on the open source multimedia GStreamer framework. The plugin uses a randomly requested source pad to dispatch video data. It also caches current IDR frames group to ensure the integrity of initial IDR frame to be sent. The plugin can be used in streaming media server to solve the problem of the blurred screen of initial screen effect in video surveillance. Experiment shows that the streaming media server with this plugin can effectively distribute the real time video data and ensure the integrity of the initial playback screen, which has a significant effect on improving the real-time monitoring effect.

Key words: video surveillance; streaming media server; GStreamer framework; H.264 standard; IDR frame; RTP protocol

视频监控联网系统中使用 IPC(IP Camera 网络摄像机)作为监控前端采集实时视频流, 通过流媒体服务器分发到多个监控客户端(以下简称客户端)进行实时视频监控. 流媒体服务器的这种工作模式呈现“一进多出”的拓扑形态.

高清视频编码标准(如 H.264[1])使用帧间图像数据压缩技术. 由于实时视频流访问的随机性, 客户端获取到的初始视频帧(第一个视频帧)可能需要参考之前的帧进行解码, 这会导致客户端由于接收的初始视频数据不完整出现初始播放画面花屏现象, 影响了实时监控效果.

为解决这一问题, 需要在视频流分发侧保证发送到客户端的每一路视频流的初始视频数据是完整的.

在视频流分发侧使用缓存数据保证所发送数据完整性是目前业内常用方式. 内容分发网络 CDN (Content Delivery Network)是使用数据缓存方式的代表[2]. 在 CDN 中, 使用缓存代理服务器缓存流媒体服务器的内容, 客户端从缓存代理服务器上获取流媒体内容.

CDN 在提升流媒体服务响应速度, 媒体流分发容量的同时, 也对网络传输服务质量提出了很高的要求. 在 CDN 中传输实时视频流时, 缓存代理服务器和流媒

① 收稿时间:2016-08-28;收到修改稿时间:2016-09-27 [doi:10.15888/j.cnki.csa.005747]

体服务器之间需要实时同步缓存数据. 若缓存代理服务器和流媒体服务器之间的网络质量不佳, 实时监控效果会受到较大影响.

本文提出了在流媒体服务器上通过实时视频流分发插件缓存当前视频数据以保证初始视频数据完整性的方法, 针对 H.264 编码标准, 基于 GStreamer 框架[3,4]设计并实现了一个应用于流媒体服务器中的实时视频流分发插件. 相比较 CDN 而言, 本文描述的流媒体服务器不需要缓存代理服务器就能通过分发插件保证所分发初始视频数据的完整性, 不再产生流媒体服务器与缓存代理服务器之间的通讯开销.

本文描述了分发插件的设计与实现方法并给出相关实验结果.

1 相关定义

1.1 GStreamer 中相关定义

GStreamer 框架基于插件(plugins). GStreamer 中的插件是一段可以加载的代码, 常常以动态链接库的形式存在. 一个 GStreamer 插件中可包含一个或多个元件(Element)[5].

元件是 GStreamer 中最重要的概念, 见定义 1.

定义 1. 元件: GStreamer 中, 元件是构建一个媒体管道的基本块, GStreamer 中的元件都继承自 GstElement 对象.

多媒体应用程序可以创建若干个元件(elements)连接在一起, 从而构成一个管道(pipeline)来完成一个媒体处理任务. GStreamer 中的管道见定义 2.

定义 2. 管道: 管道是一种容器元件, 可以向管道中添加元件从而将一组存在链接的元件组合成一个大的逻辑元件. 管道可以操作包含在其自身内部的所有元件.

管道中的元件需要建立链接关系, 衬垫(Pads)在 GStreamer 中被用于元件之间的链接,从而让数据流能在管道中流动. 衬垫的定义见定义 3.

定义 3. 衬垫: 衬垫是元件之间的接口, 元件之间的数据交互依靠衬垫来完成. 衬垫能够限制数据流的通过, 只有两个元件链接在一起的衬垫允许通过的数据类型一致时, 这两个元件才可交换数据.

一个衬垫很像一个物理设备上的接口, 如电视机上的 HDMI 输入接口和机顶盒上的 HDMI 输出接口. 正因为电视机和机顶盒都有相同类型的接口, 才可以使用数据线连接起来完成媒体流的播放.

根据数据流向, 衬垫被分为两种类型: source 衬垫与 sink 衬垫. 数据向元件之外流出可以通过一个或多个 source 衬垫, 元件接受数据通过一个或多个 sink 衬垫来完成. 在一个管道中, 数据流从一个元件的 source 衬垫流到另一个元件的 sink 衬垫.

根据生命周期, 衬垫分为永久型(always)、随机型(sometimes)、请求型(request)三种类型. 永久型衬垫在元件创建时就存在, 并一直存在直到所属元件销毁. 随机型衬垫在某种特定条件触发后才创建并存在. 请求型衬垫只在应用程序明确发出请求时才创建并存在.

使用元件, 管道和衬垫这些 GStreamer 基本元素可以构建丰富的多媒体应用程序.

1.2 IDR 帧组

视频编码中, 每一个图像帧代表一幅静止的图像. 实际压缩时, 会采取各种算法减少数据的容量[6].

I 帧表示关键帧, 可理解为这一帧画面的完整保留, 解码时只需要本帧数据就可以完成. P 帧表示的是这一帧跟之前一个关键帧(或 P 帧)的差别, 解码时需要用之前缓存的关键帧(或 P 帧)叠加上本帧定义的差别, 生成最终画面. B 帧是双向差别帧, B 帧记录本帧与前后帧的差别. 要解码 B 帧, 不仅要取得之前的缓存画面, 还要解码之后的画面, 通过前后画面与本帧数据的叠加获得最终画面.

在 H.264 编码中, I 帧不用参考任何帧, 但是之后的 P 帧和 B 帧有可能参考这个 I 帧之前的帧. 为区分首个 I 帧和其他 I 帧, 把首个 I 帧称为 IDR(Instantaneous Decoding Refresh 即时解码刷新)帧[1]. 由 IDR 帧开始, 重新开始编解码一个新的帧序列. IDR 帧之后的所有帧都不会引用该 IDR 帧之前帧的内容. 由此可知, IDR 帧阻断了误差的积累, 提供了随机访问性, 播放器可以从视频流中任意一个 IDR 帧播放而不会导致花屏.

考察帧序列(下标表示帧的显示顺序): I_1 P_4 B_2 B_3 P_7 B_5 B_6 I_8 P_{11} B_9 B_{10} P_{14} B_{11} B_{12}. 这里 I_1 和 I_8 就是 IDR 帧, 因为其之后的帧不需要跨过该帧去参考之前的帧.

定义 4 定义了 IDR 帧组的概念.

定义 4 IDR 帧组 视频流中一个 IDR 帧和其之后, 下一个 IDR 帧之前的帧组成的集合称为 IDR 帧组.

参考帧序列: I_1 P_4 B_2 B_3 P_7 B_5 B_6 I_8 P_{11} B_9 B_{10} P_{14} B_{11} B_{12}. $\{I_1, P_4, B_2, B_3\}$, $\{I_1, P_4, B_2, B_3, P_7, B_5, B_6\}$, $\{I_8, P_{11}, B_9, B_{10}\}$都可被称为 IDR 帧组.

2 基于GStreamer的流媒体服务器架构

流媒体服务器是视频监控联网系统的核心部分. 流媒体服务器获取所请求 IPC 的实时视频流, 并将这一路视频流分发到各个请求的客户端[2].

基于 GStreamer 框架, 仅用于实时视频浏览的流媒体服务器架构如图 1 所示. 本文约定 IPC 生成的视频流格式为 H.264 编码格式, 客户端与流媒体服务器之间, 流媒体服务器与 IPC 之间都使用 RTSP[7]作为控制协议, 使用 RTP[8]作为视频数据传输协议. 为方便描述, 本文约定一个 IPC 只提供一路实时视频流, 并使用 ES 流(Elementary Stream, 基本流)传输.

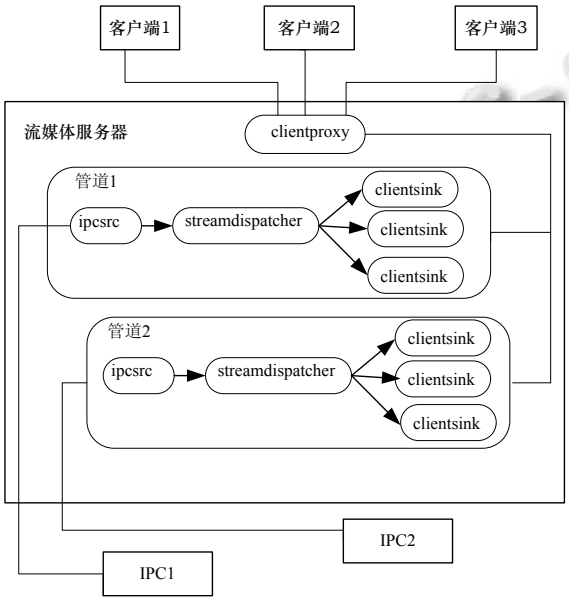


图 1　基于 GStreamer 框架的流媒体服务器架构

图 1 的流媒体服务器中, clientproxy 元件接收来自客户端的实时浏览请求建立 RTSP 会话, 并对 RTSP 会话中所请求的 IPC 建立对应的数据分发管道. 该数据分发管道为定义 2 所定义的 GStreamer 管道, 包含一个 ipcsrc 元件, 一个 streamdispatcher 元件和若干个 clientsink 元件. Clientsink 元件是随机产生的, 每建立一个对该 IPC 请求的 RTSP 会话, clientproxy 元件就在该数据分发管道中创建一个 clientsink 元件与其对应.

数据分发管道中的 ipcsrc 元件使用 RTSP 协议向所请求 IPC 获取实时视频流, 获取的实时视频流使用 RTP 协议由 IPC 发送到 ipcsrc 元件. Ipcsrc 元件收到承载实时视频流的 RTP 包后, 传递到 streamdispatcher 元件进行分发. Streamdispatcher 元件将收到的 RTP 包分别传递给管道中每个 clientsink 元件.

管道中每个 clientsink 元件包含了其对应 RTSP 会话的会话参数(传输协议, 地址端口等信息), clientsink 元件将 streamdispatcher 元件分发的 RTP 包根据会话参数发送到客户端.

以图 1 为例, clientproxy 元件收到客户端 1 对 IPC1 的实时视频流会话请求后建立 RTSP 会话 1, 并建立名为管道 1 的数据分发管道. 管道 1 中包含一个 ipcsrc 元件 ipcsrc1, 一个 streamdispatcher 元件 streamdispatcher1 和一个 clientsink 元件 clientsink1(包含 RTSP 会话 1 的会话参数). ipcsrc1 使用 RTSP 协议向 IPC1 请求实时视频流后, 承载视频流的 RTP 包经“ipcsrc1→streamdispatcher1→clientsink1”的路径传递后发送到客户端 1. 此后若 clientproxy 元件又建立客户端 2 对 IPC1 请求的 RTSP 会话 2, 则在管道 1 中再创建 clientsink 元件 clientsink2(包含 RTSP 会话 2 的会话参数), 承载视频流的 RTP 包经“ipcsrc1→streamdispatcher1→clientsink2” 的路径传递后发送到客户端 2.

图 1 中管道的生命周期依赖于请求其对应 IPC 的客户端 RTSP 会话数目: 当会话数目大于 0 时, 管道被创建 clientproxy 元件; 当会话数目等于 0 时, 管道被 clientproxy 元件销毁.

streamdispatcher 元件是数据分发管道中的分发控制中枢, 不仅将数据分发给 clientsink 元件, 还保证发送给 clientsink 元件的初始帧为 IDR 帧. streamdispatcher 元件的设计与实现是本文的重点, 下一节将详细介绍 streamdispatcher 元件的设计与实现.

3 实时视频流分发插件设计与实现

3.1 设计思想

将 IPC 发送的实时视频流视为一个帧序列. 为确保发送到客户端的初始帧为 IDR 帧, streamdispatcher 元件需缓存当前帧参考的 IDR 帧组(当前帧参考的 IDR 帧及当前帧与 IDR 帧之间的其它帧). Streamdispatcher 元件在分发数据时总是先发送缓存的当前 IDR 帧组, 再发送当前帧. 这种工作方式可由图 2 示意.

图 2 中 t_1 时刻, 请求 IPC 实时视频流的 RTSP 会话 1 建立, 与之对应的 clientsink 元件也随之创建. streamdispatcher 元件此时收到 IPC 发送的当前帧为 $SLICE_{1m}$(非 IDR 帧, 且 IDR1 与 $SLICE_{1m}$ 之间无 IDR 帧). 因为在 t_1 时刻之前, streamdispatcher 元件中已经缓存了 IDR1 及 IDR1 至 $SLICE_{1m}$ 之间所有帧的 IDR

帧组, 所以在 t_1 时刻 streamdispatcher 元件首先发送缓存的 IDR 帧组, 再发送 $SLICE_{1m}$ 帧给对应 RTSP 会话 1 的 clientsink 元件. 对应 RTSP 会话 1 的 clientsink 元件收到初始帧为 IDR1. 同理在 t_2 时刻, streamdispatcher 元件中缓存的是 IDR2 及 IDR2 至 $SLICE_{2n}$ 之间(IDR2 与 $SLICE_{2n}$ 之间无 IDR 帧)所有帧的 IDR 帧组, 对应 RTSP 会话 2 的 clientsink 元件收到的初始帧为 IDR2.

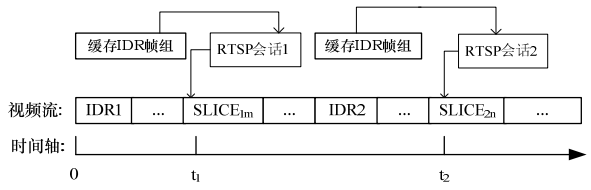


图 2 streamdispatcher 元件工作原理

Clientproxy 元件建立 RTSP 会话时会创建对应的 clientsink 元件, 然后向 streamdispatcher 元件申请创建一个请求型 source 衬垫, 并将此衬垫与 clientsink 元件的 sink 衬垫进行链接. 当 streamdispatcher 元件在接收视频流数据后, 通过已创建的请求型 source 衬垫将数据推送到 clientsink 元件. Ipcsrc、clientproxy 和 clientsink 之间的衬垫链接关系如图 3 所示.

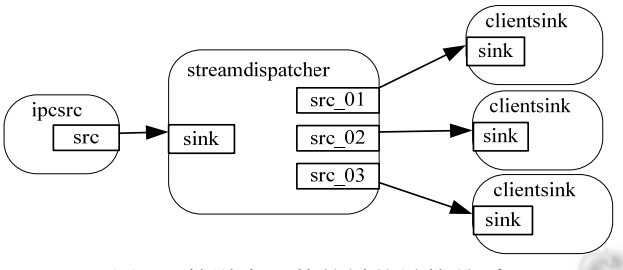


图 3 管道中元件的衬垫链接关系

3.2 IDR 帧识别方法

ES 流中, IPC 将 H.264 视频编码后的 NALU (Network Abstract Layer Unit)封装在 RTP 包中. NALU 的首字节定义如图 4 所示.

0	1	2	3	4	5	6	7
F	NRI			Type			

图 4 NALU 的首字节定义

其中 F 为 1 个比特, 在规范中规定为 0. NRI 为 2 个比特, 用于表示 NALU 的优先级. Type 为 5 个比特, 表示 NALU 的类型. 若 NALU 类型为 5, 则表示该 NALU 为 IDR 图像, 其对应的帧为 IDR 帧.

RTP 在对 NALU 封包时, 会使用单一 NAL 单元模式, 组合封包模式, 分片封包模式这三种模式中的一种. 在实际运行中, 应根据视频流的封包模式判断 RTP 载荷的 NALU 类型是否为 IDR 帧类型.

以分片封包模式为例, 若 RTP 包载荷的首字节中 Type 为 28, 表示为 FU-A 类型的分片单元. 此时应判断该 RTP 包是否为第一个分片单元, 若为第一个分片单元, 再对 FU 头字节(载荷的第二个字节)的后 5 个比特判断 NALU 类型. 只有包含第一个分片单元且 FU 头字节中类型为 5 的 RTP 包才被判断为 IDR 帧类型.

3.3 视频流分发算法

如图 3 所示, 管道中每一个 clientsink 元件创建时, streamdispatcher 元件也会创建一个请求型 source 衬垫与该 clientsink 元件链接, 承载实时视频流的 RTP 包经该 source 衬垫传递到与其连接的 clientsink 元件. 创建请求型 source 衬垫的过程如算法 1 所示.

当承载实时视频流的 RTP 包从 streamdispatcher 元件的 sink 衬垫流入时, streamdispatcher 元件将流入的 RTP 包分发到使用算法 1 创建的 source 衬垫中, 同时缓存当前帧的 IDR 帧组数据. 这个过程使用算法 2 描述. 算法 2 在 streamdispatcher 元件每接收到一个 RTP 数据包时都会被调用一次.

算法 1 描述如下:

算法 1: streamdispatcher 元件的 source 衬垫创建算法

输入: *sessionId*(客户端请求的 RTSP 会话 ID)
index(创建 source 衬垫的当前索引)
srcPadList(已创建 source 衬垫的列表)
输出: *srcPad*(新创建的 source 衬垫)

1) *padContext* = newContext();
2) *padContext.sessionid* = *sessionId*;
3) *padContext. initialized* = FALSE;
4) *srcPad* = newSourcePad();
5) *srcPad*.setPadName(*index*);
6) *srcPad*.setContext(*padContext*);
7) *srcPadList*.append(*srcPad*);
8) return *srcPad*.

算法 1 中 *padContext* 为 *srcPad* 的上下文对象, 包含两个成员变量: *sessionid* 和 *initialized*. *sessionid* 为整型, 用于存放所创建衬垫对应的 RTSP 会话 ID. *initialized* 为布尔型, 表示该衬垫是否已经接收到了初始 IDR 帧, 初始值为 FALSE. 算法 1 的第 6 行表示

srcPad 设置上下文对象.

streamdispatcher 元件使用列表变量 *srcPadList* 存放所有被请求创建的 source 衬垫, 并用 *index* 变量记录索引. 算法 1 的第 7 行表示新创建的 *srcPad* 被加入到 *srcPadList* 列表中.

算法 2 描述如下:

算法 2: streamdispatcher 元件的视频流分发算法

输入: *rtpPkg* (来自 sink 衬垫的 RTP 数据包)
输出: 无

```
1)  isIdrType = parseFrameType(rtpPkg);
2)  foreach(pad  in  srcPadList)
3)    {  context = pad.getContext();
4)       if (!context. initialized)
5)              {if  ((idrGroupList.length>0)
&&(!isIdrType))
6)              { pushData(pad, idrGroupList);
7)                pushData(pad, rtpPkg);
8)                context. initialized = TRUE;
9)              }
10)          }
11)       else
12)          { pushData(pad, rtpPkg);
13)          }
14)   }
15)  if (idrGroupList.length>0)
16)    { if (isIdrType)
17)        {idrGroupList.empty();
18)        }
19)     idrGroupList.append(rtpPkg);
20)    }
21)  else
22)    { if (isIdrType)
23)        { idrGroupList.append(rtpPkg);
24)        }
25)    }
```

算法 2 中, *idrGroupList* 是 streamdispatcher 元件用于缓存当前 IDR 帧组的列表变量, 在 streamdispatcher 元件初始化时创建且设初始值为空. *rtpPkg* 是来自 sink 衬垫由 IPC 发送的 RTP 数据包, 被 parseFrameType 函数用来判断载荷中 NALU 类型是否为 IDR 帧类型 (判断方法参考 3.2 节, 本文中只考虑单一 NAL 单元模式和分片封包模式).

算法 2 的第 1 至 14 行描述了向每个已创建 source 衬垫(包含在 *srcPadList* 中)推送数据的过程. 算法 2 先根据每个 source 衬垫的上下文信息了解该衬垫是否已经接收过初始 IDR 帧: 如果该衬垫还未接收过初始帧: 在缓存当前 IDR 帧组的 *idrGroupList* 不为空且当前 NALU 类型不为 IDR 帧类型的条件下, 先推送 *idrGroupList*, 再推送 *rtpPkg* 到该衬垫, 并将上下文中的初始帧接收标示置为 TRUE; 如果该衬垫已经接收过初始 IDR 帧: 只需要将当前 *rtpPkg* 数据推送至该衬垫即可.

向 source 衬垫推送数据后, 需要判断当前 *rtpPkg* 是否需要加入到 *idrGroupList* 中, 算法 2 的第 15 至 25 行描述了这一个过程. 如果当前 *rtpPkg* 的 NALU 类型为 IDR 帧类型则表示 *idrGroupList* 已缓存的 IDR 帧组不再成为当前 IDR 帧组, 这时算法 2 清空 *idrGroupList* 后添加当前 *rtpPkg* 到 *idrGroupList* 中.

算法 2 利用缓存的当前 IDR 帧组保证了发送到各 source 衬垫的初始帧为 IDR 帧.

4　实验结果

应用第 2 节介绍的架构, 基于 GStreamer 1.6 版本开发了一个简单的流媒体服务器, 仅实现实时浏览 IPC 的功能. 其中 clientproxy 元件和 streamdispatcher 元件都实现为独立插件, ipcsrc 元件和 clientsink 元件实现在一个插件当中. 基于 GStreamer 1.6 版本自带的 rtspsrc, decodebin, autovideosink 等插件开发了流媒体服务器的实时浏览客户端, 可用于实时浏览 IPC 监控的视频流.

服务器与 IPC 之间, 客户端与服务器之间都通过 RTSP 协议建立媒体会话并且使用 RTP 协议进行视频流传输. 实验使用一个 IPC 作为视频监控前端, 客户端建立多个 RTSP 会话来测试 streamdispatcher 插件的分发效率. IPC 视频流为 ES 流, 使用 H.264 编码, 分辨率为 704×400, 帧率为 25fps, 平均每 3 秒发送一个 IDR 帧.

实验所使用服务器为 PC 机, 操作系统为 windows 7, CPU 为英特尔酷睿双核处理器(2.5GHZ),内存为 4GB. 实验中针对不同的 RTSP 会话数目重点考察了 streamdispatcher 插件分发一个 RTP 包(执行一次算法 2)的时间, 并和不使用 streamdispatcher 插件直接分发 RTP 包的时间做了比较. 实验结果如图 5 所示.

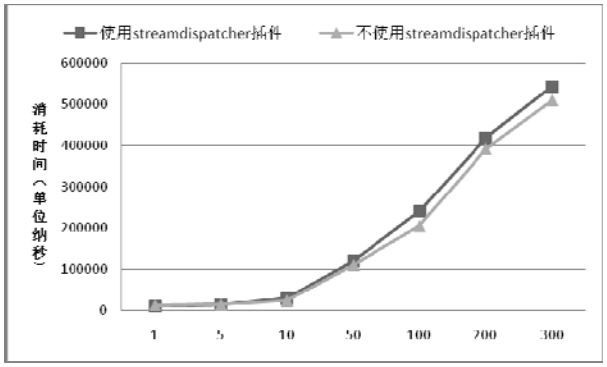


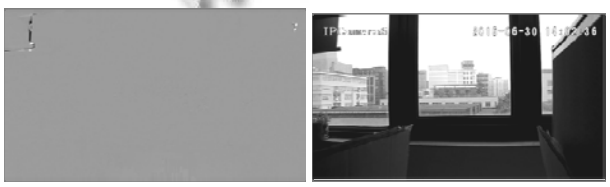
图 5 RTP 包分发消耗时间及比较

图 5 中的消耗时间是分发一个 RTP 包到所有 clientsink 插件所消耗的时间(采集头 1000 个 RTP 包的分发时间作为样本, 计算出平均时间). 每个 clientsink 插件对应一个 RTSP 会话, 随着会话数目的增加, 分发时间也随之增加. 分发时间随会话连接数目线性增加, 这与算法 2 的逻辑拟合.

图 5 对使用 streamdispatcher 插件分发和不使用 streamdispatcher 插件直接分发所消耗的时间进行了比较. 可以看到两者在消耗时间上基本无差别, 这是因为 streamdispatcher 插件只在直接分发流程上增加了一些条件判别, 所增加的开销极小.

我们的实验目标设置为 300 路实时视频流并发输出能力. 在会话连接数目达到 300 时, 一个 RTP 包发送到所有 clientsink 插件的消耗时间为 542206 纳秒. 使用抓包工具得知 IPC 平均一秒内大约发送 60 个 RTP 包, 因而 streamdispatcher 插件的转发能力完全能够满足实验目标.

Streamdispatcher 插件保证了发送到客户端的初始帧为 IDR 帧, 这也使得视频监控的效果得到了提升, 图 6 展示了使用 streamdispatcher 插件和未使用 streamdispatcher 插件的视频监控效果.



(a)未使用 streamdispatcher 插 (b)使用 streamdispatcher 插件

图 6 streamdispatcher 插件使用与否的效果比对

图 6 中, 未使用 streamdispatcher 插件的流媒体服务器发送到客户端的初始画面出现花屏现象, 而使用 streamdispatcher 插件的流媒体服务器保证了客户端初始画面的完整性, 提升了视频监控效果.

5 结语

本文基于 GStreamer 框架, 针对 H.264 编码规范, 设计并实现了一种实时视频流分发插件: 使用随机创建的请求型 source 衬垫完成数据的分发, 并通过缓存当前 IDR 帧组的方法保证了发送的初始帧为 IDR 帧, 解决了视频监控初始画面花屏的问题. 本文所描述的插件可应用于流媒体服务器当中, 实验证明这种保证了初始数据完整性的实时视频流分发插件在高效转发的同时, 也大大提升了实时视频监控效果.

虽然本文讨论的插件针对 H.264 编码规范, 但这种在视频分发端缓存当前 IDR 帧组以保证初始视频数据完整性的思路仍可扩展到兼容多种编码格式的流媒体服务器设计当中.

参考文献

1 Wiegand T, Sullivan G J, Bjontegaard G, Luthra A. Overview of the H.264/AVC video coding standard. IEEE Trans. on Circuits & Systems for Video Technology, 2003, 13(7): 560–576.

2 杨戈,廖建新,朱晓民,樊秀梅.流媒体分发系统关键技术综述.电子学报,2009,37(1):137–145.

3 Taymans W, Baker S, Wingo A, Bultje RS, Kost S. GStreamer Application Development Manual(1.6.0). 2013. https://gstreamer.freedesktop.org/documentation/.

4 刘兴民,赵连军.基于 GStreamer 的远程视频监控系统的关键技术研究.计算机应用与软件,2011,28(5):243–246.

5 Boulton RJ, Walthinsen E, Baker S, Johnson L, Bultje RS, Kost S, Müller TP, Taymans W. GStreamer Plugin Writer's Guide(1.6.0). 2013. https://gstreamer.freedesktop.org/documentation/.

6 计文平,郭宝龙,丁贵广.新一代视频编码国际标准的研究.计算机应用与软件,2004,21(2):60–62.

7 Schulzrinne H, Rao A, Lanphier R. Real Time Streaming Protocol (RTSP). RFC 2326, Internet Engineering Task Force, 1998,4.

8 Schulzrinne H, Casner S, Frederick R, Jacobson V. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, Internet Engineering Task Force, 2003,7.