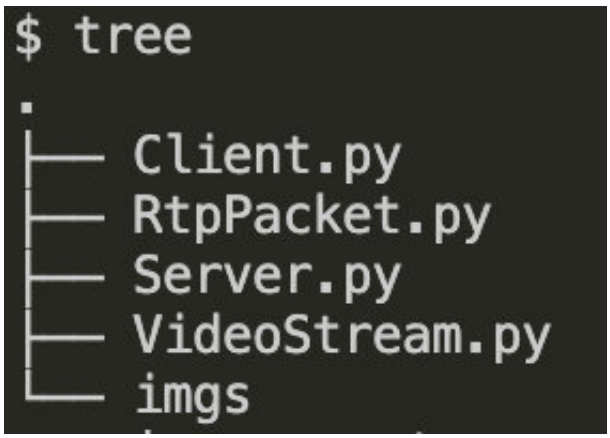
流媒体实验报告

1. TASK1简介

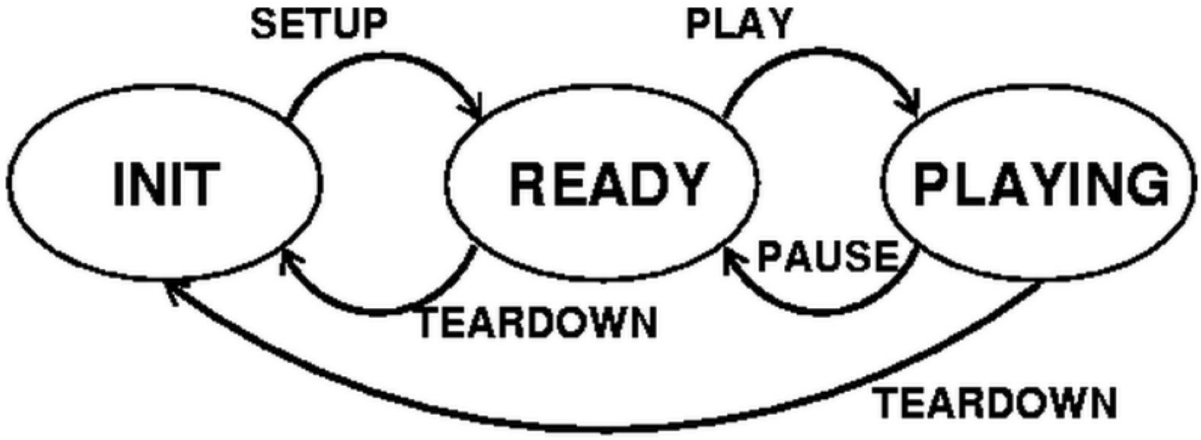
task1在给定代码基础上完善并完成RTP协议以及客户端与服务端，实现传输图片流并播放的功能，实现后界面如下：

task1代码结构如下：其中imgs文件夹中为测试图片文件夹。实现了播放/暂停视频的基本功能，SETUP，PLAY，PAUSE，TEARDOWN这四个RTSP命令。



下面介绍server的思路：

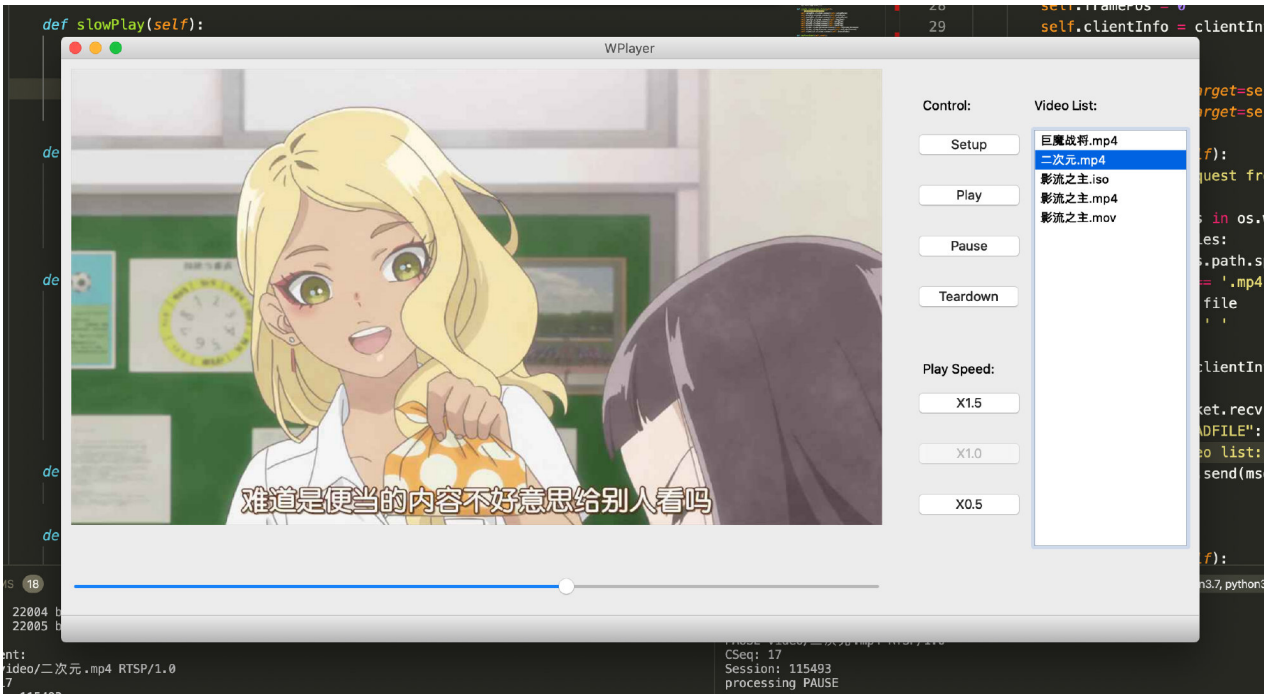
Server.py中定义了Server类，其具有INIT，READY，PLAYING这三种状态，当服务器运行时，将新创建一个进程recvRtspRequest来监听服务器的RTSP命令以防止阻塞主进程，当收到RTSP命令后解析并根据命令执行相关操作。服务器的状态转移图如下：



在setup后，服务器会建立起RTP套接字用于传输流数据，在任务一中我采用了每次传输一张图片作为视频一帧的方法，每隔0.05秒传输一次，client在接收到图片后立刻渲染，这样客户端可以播放FPS=20的视频。

2. TASK2简介

task2在task1的基础上实现了完整的流媒体播放器和服务器。实现了传输播放视频的功能，支持`.mp4`，`.mov(H.264)`，`.iso(DVD)`，`.avi`，`.flv`，`.mkv`等**多种格式**，并具有进度条，调整播放进度，变速，播放列表等功能，同时支持支持多客户端。运行效果如下：



下面介绍具体实现

任务2的核心功能相对于任务1，最大的区别就是要求播放完整的视频而不是图片，因此考虑在任务1的思路基础上对视频进行传输：对视频进行一帧一帧的解码并传输，这样就类似于任务1的效果。在本次实验中，我采用opencv2进行了视频格式的解码，并支持多种格式的视频。

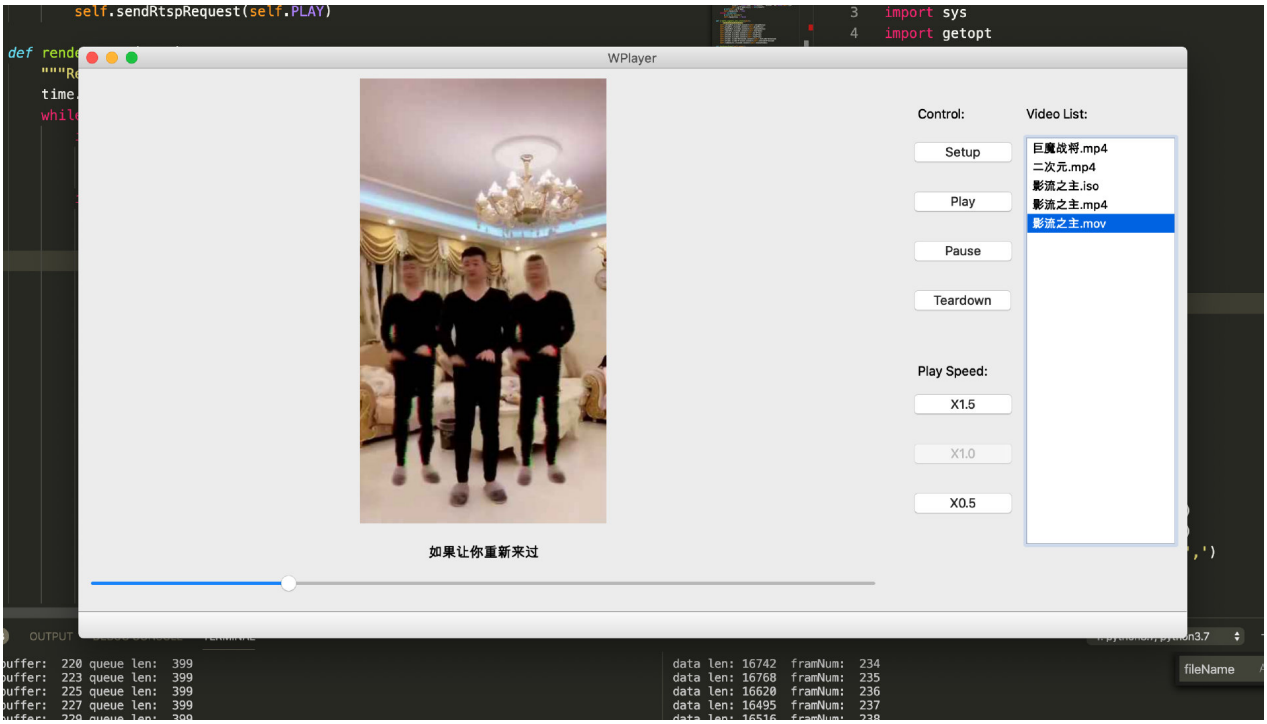
在task2中，相比task1的server与client有以下主要不同：

1. 由于在实验过程中发现tkinter的图形渲染性能较差，在处理较高质量的视频时会出现卡顿的情况，我采用pyqt5**重构了GUI客户端**，性能与美观度均有较大提升。
2. 在任务1的client中，采用的播放逻辑是：client一直接受RTP套接字，每当收到数据后立刻渲染显示图片，播放速度由服务端发送速度决定，并且接收数据与渲染图片在同一进程内，而这样有两个很大的问题：1.由于渲染和接收数据在同一进程内先后运行，**在渲染图片的时候监听被阻塞**，无法接收数据，导致大量的包丢失。 2.渲染的帧数完全由服务端发送帧之间的间隔决定，受网络质量影响较大，并且不易实现调速功能。

   因此我重构了client，将接收RTP数据和渲染图片**在两个进程中实现**，各自独立，并且**加入了视频缓存功能**，建立了一个缓存帧队列，将接收到的RTP数据缓存在队列中，客户端渲染时从缓存队列中获取帧即可，这样提高了速度，也更有逻辑性，提高了用户友好性。
3. 由于UDP的数据大小有限制，我在解码创建VideoStream时对图片帧进行了压缩以减小尺寸。
4. 为了实现更多的功能，我增加了一些相关RTSP功能和视频流属性，例如`DETAIL`命令获取视频的信息：尺寸，总帧数和帧率等；`SETPOS`命令来设置videoCature的帧定位位置用以实现视频进度调整。
5. 播放速度的调整：首先根据`DETAIL`命令获取视频的帧率，这是1倍速的帧率，而通过调整渲染两帧之间的时间间隔调整视频播放速度。
6. 支持多客户端并通过sessionid区分不同客户端同时运行。
7. 服务器上增加了一个socket用于传输服务器端文件列表，固定端口号为12345，客户端登录时通过该端口发送`READFILE`命令，服务端收到后读取video文件夹下面的视频文件列表并传回客户端。

3.实现的额外功能：

- 支持多种格式的视频，包括`.mp4`，`.mov(H.264)`，`.iso(DVD)`，`.avi`，`.flv`，`.mkv`等格式
- 支持缓存机制
- 对传输数据帧进行了压缩
- 视频尺寸的自适应，不会出现视频由于尺寸和宽高比的缘故被截断、比例异常的问题，提高用户友好性
- 快捷键操作方便用户：可以使用空格切换播放或暂停，使用左右方向键控制播放进度。
- 支持同一客户端切换多个视频播放，不用重启切换视频
- 可以加载远程服务器视频列表并选择播放
- 支持自动加载字幕并与播放进度同步显示



4.实验中遇到的主要难点和问题：

在本次实验中，我先后遇到了很多问题和难点，在此记录其中的一些典型问题：

1. 一开始在macOS上传送数据时，发现一直显示数据包过大传输失败，但是我的数据只有20KB左右大小，并没有超过UDP限制的60KB左右大小。经过查阅资料，我在stackoverflow的一个角落发现了这个问题的原因：不知道出于什么目的macOS默认将UDP包限制在了9KB的大小，需要手动调整系统参数才能增大阈值并且每次重启后会恢复9KB。

```
vincent@Mac: ~/Desktop/VideoStreaming-master
$ sudo sysctl -w net.inet.udp.maxdgram=65507
Password:
net.inet.udp.maxdgram: 9216 -> 65507
```

2. 一开始使用原始tkinter客户端渲染较高质量的图片时，发现卡顿极其明显，通过对服务端发送和客户端接收数据的分析，我发现问题不在数据传输上而是图片渲染上，因此为了提高性能我采用了pyqt重构代码。
3. 在代码中，由于我将渲染进程和数据接收进程分开了，每次点击play都会新建一个渲染进程进行图片渲染。而一开始我没有注意在每次暂停后释放当前渲染进程，导致每次暂停后开始播放都会多出一个进程进行图片渲染，变相的增加了播放速度。解决方案是每次暂停时及时释放进程。
4. 由于涉及多个进程，例如RTSP，RTP，视频渲染都是在不同进程中进行，进程间的通信和顺序非常重要。一个值得注意的问题是，不同进程间的代码执行顺序和代码顺序不一定一致，例如我希望在RTP进程中接受到某一帧后再播放这一帧，但是即使我将接受代码写在播放代码前也不能保证他们的执行顺序。解决这个问题的方法是使用回调函数或者全局的状态变量来控制运行流程。
5. 因为涉及到网络通信，很多地方需要使用`time.sleep`设置等待以确定顺序，保证正确执行。

6. 可能是pyqt的bug，在使用pyqt的slider控件时，为了实现进度条的更新，我在渲染进程中每次更新画面也会更新进度条。但是这样会导致偶发性的画面卡死。根据与同学的讨论与猜测，我们找到了几个解决方法：1.画面渲染与进度条更新采用不同进程，进度条定时刷新即可。 2.在同一进程内，先刷新进度条再渲染画面，这样可以解决卡死的问题。 3. 先渲染画面再sleep一段时间后再更新进度条。
7. 由于UDP是不可靠的，客户端必须考虑可能的异常情况，例如丢帧，网络延迟等情况并做出异常处理防止客户端崩溃。
8. 关于缓存。目前我的缓存方法是将缓存帧放在一个队列中，即储存在内存中，这种方法的好处是速度快，IO延迟低，缺点也显而易见，在播放较大视频的时候可能会造成内存占用过大。另一种方法是将缓存写出为文件，这样比较省内存但是由于密集的IO操作，会拖慢速度。为了解决内存缓存占用大的问题，可以使用控制缓存速度的方法：即一段一段缓存，当缓存超过播放进度太多时可以停止缓存等待播放进度，这和实际上很多流媒体播放器采用的策略相似。
9. macOS上pyqt的bug，在macOS的某些版本的pyqt5上(>5.10)，更新label的文字刷新不及时，会被遮挡，解决方法是采用pyqt 5.10版本或者更换系统。
10. macOS上pyqt的bug，在某些版本的pyqt5上，label的pixmap设置后不会立刻刷新，当失去焦点后会刷新，经测试在Windows版本下未出现该问题，是mac版pyqt自身的bug。

5.总结

本次作业历时三周，总体上任务量还是比较大的，并且感觉RTP协议不像FTP协议规定那么明确，参考资料那么多，因此在实现的过程中也碰了很多坑，查找了很多资料。同时还有很多功能可以拓展，例如可靠RTP，自动恢复客户端进度，声音播放等功能，但是限于时间精力我没能继续实现。不过通过这次作业我对于UDP和RTP理解更加深入了，编程水平也有了一定的提高，感觉还是收获较多。