流媒体传输实验

黄奔皓、朱鹏翔、李佳鑫、周晟洋

2023 春季学期

摘要

本实验项目的主题是流媒体传输协议 RTSP 和实时传输协议 RTP。其中，我们首先实现了基于 RTSP 和 RTP 协议的流媒体传输基本功能。此外，我们还实现了诸多拓展功能，如：帧率、丢包率等信息的计算和实时展示、PLAY 和 SETUP 按钮功能的合并。更进一步的，我们还做出了许多创新性的尝试。我们摒弃了模版代码提供的 UI 界面，自己搭建了美观的 UI 界面，并在其中加入了展示流媒体信息和设置端口号、文件名等窗口和菜单栏目。同时，我们还实现了对视频进度的实时展示，以及视频的快进和回退功能。最后，我们对相关数据进行了可视化操作，分析了在不同电脑下流媒体传输过程的异同。

目录

1 基于 RTP 与 RTSP 的基本功能实现

1.1 Real-time Transport Protocol

RTP（Real-time Transport Protocol）是一种用于实时传输音频和视频的协议。它定义了数据包的格式和传输规则，以确保在网络上传输音视频时能够满足实时性和同步性的要求。RTP 协议通常与 RTCP（RTP Control Protocol）一起使用，RTCP 用于监控 RTP 传输，并提供传输质量反馈和同步信息。

RTSP（Real-Time Streaming Protocol）是一种用于控制实时流媒体传输的协议。它允许客户端与服务器之间进行双向通信，以控制流媒体的播放、暂停、停止、快进等操作。RTSP 协议通常与 RTP 协议一起使用，RTP 负责传输音视频数据，而 RTSP 负责控制音视频的播放流程。

总的来说，RTP 和 RTSP 是在实时流媒体传输中常用的协议，它们分别负责传输数据和控制流程，协同工作以实现高质量的实时音视频传输。为了完成基础任务，我们先分析给出代码模版，定位我们所需要完成的操作。在本次实验中，我们用本机不同的端口号来模拟客户端与服务器端，并应用 RTP 与 RTSP 协议实现流媒体数据的传输。所有的操作同时建立在以`tkinter`为框架的 GUI 系统中，而我们的基础任务便是实现服务器与客户端的通信，数据的交互以及对于用户请求的响应。

- `Client`、`ClientLauncher`

  `ClientLauncher`启动`Client`和用户界面,我们将使用该界面发送 RTSP 命令并显示视频。在`Client`类中,我们首先需要实现按下诸如`Pause`、`Play`等按钮的操作。在基础操作中,不需要修改`ClientLauncher`模块。该模块作为客户端的启动器从外部接收服务器和客户端的端口号。

- `ServerWorker`、`Server`

  这两个模块实现了服务器，响应 RTSP 请求并将视频流回传。`RTSP`交互已经实现，`ServerWorker`调用`RtpPacket`类中的方法对视频数据进行分组。

- `RtpPacket`

  这个类用于处理`RTP`数据包。它有专门的方法用于处理客户端接收到的数据包，而客户端也会进行数据解包（解码）。对于这一文件，我们需要根据 RTP 协议的头部文件来完成视频数据`RTP`分组的实现（服务器使用）。

- VideoStream 这个类用于从磁盘上的文件中读取视频数据，作为内置函数并不需要进一步操作。

下面，我们从代码运行的角度来分析基本实现的过程。在此过程中，我们将逐步引入对于代码的必要修改。通过这种方式，我们将对 RTP 和 RTSP 有更加全面的认识。在实际代码运行中，我们首先指定服务器端口`server_port`（后续表述中我们将其设定为`6666`），并启动服务器来监听来自客户机的请求。之后，我们用类似的方式在`ClientLauncher.py`中启动客户机，它将创建一个`Client`对象，并开始与服务器进行交互。在后续的操作中，我们指定客户机为`5008`。

1.2 Client 端

在创建了`Client`类之后,首先该类初始化一系列核心参数,包括 RTSP 包序号`rtspSeq`、帧编号`frameNbr`等。在该类中，同样定义了客户端的基础 GUI，用于后续操作的输入。注意到模版给出的实现是将接收到的图片缓存到本地，进而从本地将图片读入 GUI 来进行显示。在该类中，还定义了客户所能进行的几个基本操作，以如下所示的函数`playMovie()`为例。

```
def playMovie(self):
    """Play button handler."""
    if self.state == self.READY:
```

```
        # Create a new thread to listen for RTP packets
        threading.Thread(target=self.listenRtp).start()
        self.playEvent = threading.Event()
        self.playEvent.clear()
        self.sendRtspRequest(self.PLAY)
```

为了实现这些功能，客户端需要向服务器发送 RTSP 请求，而面对这些请求所进行的具体操作还与客户机目前所处于的状态相关。这一关系可通过图1所示的状态转移机进行展示。基于不同的状态和不同的请求，客户机的下一状态也将对应地发生变化。

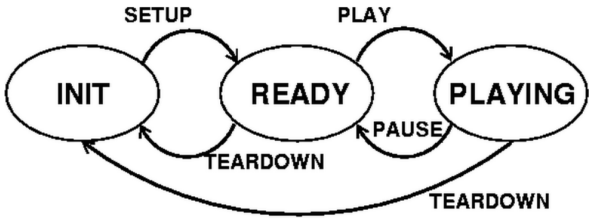

图 1: 客户机状态转移示意图

相应的代码实现同样遵循图1的转移过程，在这里我们给出两个代码片段以供分析，首先是考虑状态为INIT的情况。此时所发出的请求只能是SETUP，因而我们只需要更新SeqNum后向服务器发出请求，并维护该 RTSP 包请求的类型即可。

```
# Setup request
if requestCode == self.SETUP and self.state == self.INIT:
    threading.Thread(target=self.recvRtspReply).start()
    # Update RTSP sequence number.
    self.rtspSeq += 1

    # Write the RTSP request to be sent.
    request = "SETUP " + str(self.fileName) + "\nCseq: "+ str(self.rtspSeq) \
                + "\n" + "RTSP/1.0 RTP/UDP " + str(self.rtpPort)

    # Keep track of the sent request.
    self.requestSent = self.SETUP
```

而如果用户在播放视频（PLAYING状态下）进行暂停PAUSE操作，则需要通过类似的方式来发出对应的请求。

```
elif requestCode == self.PAUSE and self.state == self.PLAYING:
    # Update RTSP sequence number.
    self.rtspSeq = self.rtspSeq + 1

    # Write the RTSP request to be sent.
    request = "PAUSE " + "\nCseq: " + str(self.rtspSeq)

    # Keep track of the sent request.
    self.requestSent = self.PAUSE
```

注意在这里每一个分支所发送的请求字符串需要遵守较为严格的模版，这是为了与后续服务器端对于 RTSP 请求的译码模版进行对应。一个request命令主要包括三行，其格式如下：

```
[Request Type] <space> [File Name]
[RTSP Seq Num]
"RTSP/1.0 RTP/UDP" <space> [RTP Port]
```

在完成了上述设定后，便可以将正确对应用户输入和当前状态的 RTSP 报文进行发送

```
# Send the RTSP request using rtspSocket.
self.rtspSocket.sendto(str(request).encode(), (self.serverAddr, self.serverPort))
```

1.3 Server 端

在完成了发送之后，我们便可以关注服务器端的对应操作。在ServerWorker.py文件中，服务器将实现对 RTSP 请求的响应。具体而言，函数会根据请求类型和客户端状态，执行以下操作：

- 对于SETUP请求，如果客户端当前处于INIT状态，则更新客户端状态为READY，并尝试打开指定的媒体文件。如果文件不存在，则发送404响应，反之则生成一个随机的RTSP会话 ID，并发送200响应。
- 对于PLAY请求，当且仅当客户端当前处于READY状态时更新客户端状态为PLAYING，并创建一个新的 RTP/UDP 套接字。然后发送200 OK响应，并创建一个新的工作线程，用于发送 RTP 数据包。
- 对于PAUSE请求，如果客户端当前处于PLAYING状态，则更新客户端状态为READY，并设置一个 event 标志以暂停 RTP 数据包的发送，同时发送200 OK响应
- 对于TEARDOWN请求，无论客户端处于何种状态，都会更新客户端状态为INIT，并发送200响应。然后，该函数会等待工作线程结束，并类似地设置 event 标志以停止发送 RTP 数据包。最后，在这种情况下服务器将关闭 RTP/UDP 套接字。

在实现过程中，将控制信息封装至 RTP 包的操作非常重要。在给定模版的基础上，我们需要补充的函数为RTP包中的encode()函数。在给定了各种控制信息之后，我们根据图2所示的 RTP 报文格式，将相应的信息进行填充与分配，其代码实现如下所示。

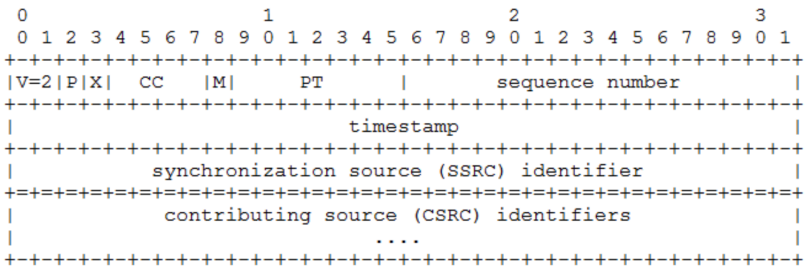

图 2: RTP 报文格式

```
def encode(self, version, padding, extension, cc, seqnum, marker, pt, ssrc, payload):
""" Encode the RTP packet with header fields and payload. """
    timestamp = int(time())
    header = bytearray(HEADER_SIZE)

    # Fill the header bytearray with RTP header fields
```

```
header[0] = (version << 6) | (padding << 5) | (extension << 4) | cc
header[1] = marker << 7 | pt
header[2:4] = seqnum.to_bytes(2, byteorder='big')  # big represents [big end] coding
header[4:8] = timestamp.to_bytes(4, byteorder='big')
header[8:12] = ssrc.to_bytes(4, byteorder='big')
self.header = header

# Get the payload from the argument
self.payload = payload
```

注意在这里我们需要大量进行位运算操作来处理Byte类型的数据，为了方便起见我们使用位运算符<<来进行处理。在编码时我们需要选择大端模式来保证填充信息的顺序正确。在RTP包的最后，还有诸如seqNum()、payloadType()等辅助函数用于从 RTP 包中提取相应的信息。

1.4 RTSP 响应模式总结

此时，所有需要我们修改的代码已经完成，但传输过程尚未结束。RTP包在被封装完成后会发送回客户端，而Client.py中的parseRtspReply()方法便用于解析服务器发送回来的数据。该函数用于解析 RTSP 服务器返回的响应。函数首先获取响应的序列号seqNum，并与客户端之前发送的请求的序列号进行比较。如果序列号相同，就说明该响应是对之前发送的请求的回应。接着，函数从响应中获取RTSP会话 ID 号session，并与客户端保存的会话 ID 号进行比较。如果 ID 号相同，说明该响应是对当前会话的回应。最后，函数根据响应的状态码和之前发送的请求类型，更新客户端的状态。具体而言，函数会根据请求类型和响应状态，执行以下操作：

- 对于SETUP请求的响应，函数将客户端的状态更新为READY，并打开 RTP 端口以接收视频数据。
- 对于PLAY请求的响应，函数将客户端的状态更新为PLAYING。
- 对于PAUSE请求的响应，函数将客户端的状态更新为READY，并停止视频播放线程。
- 对于TEARDOWN请求的响应，函数将客户端的状态更新为INIT，并关闭套接字。

在完成了以上过程后，我们便实现了客户与服务器之间的基本交互，通过合理的状态转移与控制信号传输来得到合理的结果。

2 附加功能的实现

2.1 UI 设计

为了提升交互体验，我们摒弃了原有的模板，并用 python 的`ttkbootstrap`库对 UI 进行了美化设计。

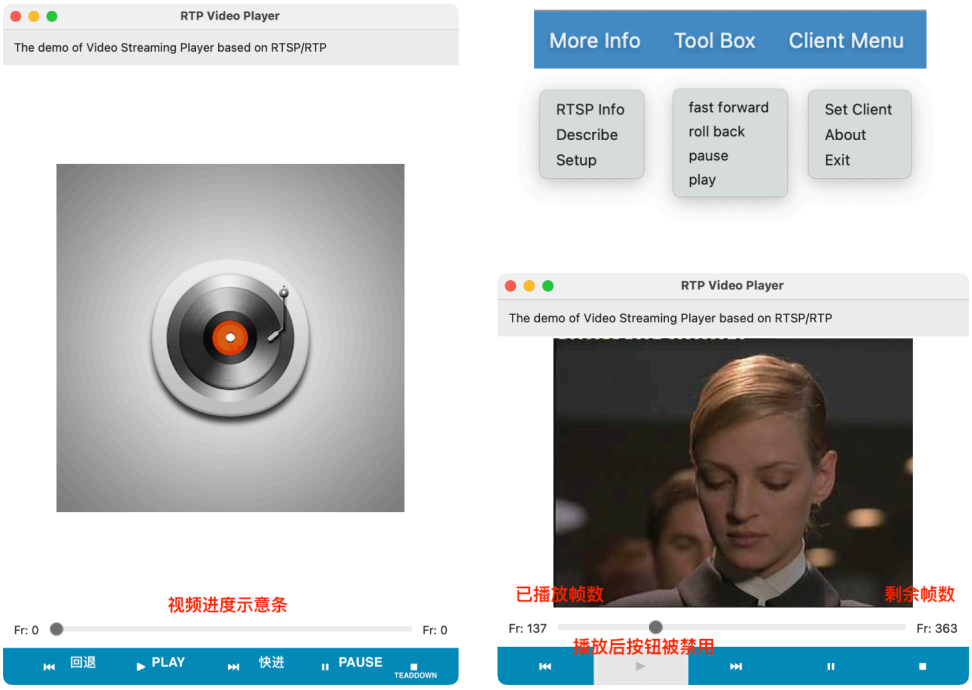

图 3: 主界面 UI 设计

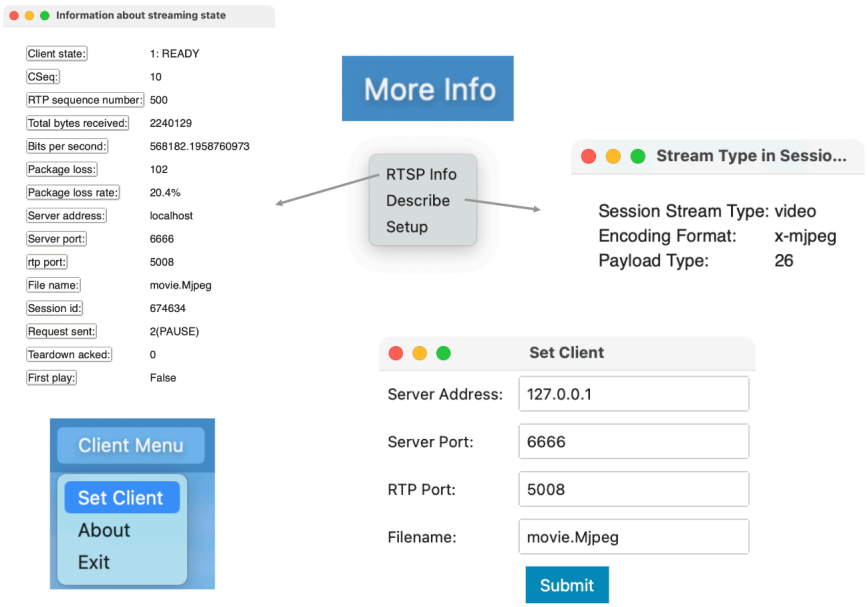

图 4: 菜单栏及功能模块展示

我们使用了更加美观的交互图标而不是文字的形式设计按钮，同时添加了视频的进度条展示，并为播放器添加了默认背景。此外，我们为播放器增加了三个层级菜单。`More Info`中展示了视频流的更多信息，其中、`RTSP info`命令展示了相关的详细信息，包括丢包率、bps、客户端状态等等；`Describe`命令中展示

了通过 DESCRIBE 请求获取的流媒体格式和相应的编码格式。在Client Menu中，Set Client中我们允许用户设置主机地址、服务器端口名和 RTP 端口，以及想要获取的文件名。About则展示开发人员的信息。Tool Box将播放器按键添加进了菜单栏。

以下函数是在 client 里创建 GUI 的代码片段，对应的函数创建了对应的组件。由于 UI 设计与课程的内容关联不大，我们不再此赘述细节。

```
def create_Widgets(self):
    self.create_header()  # header message
    self.create_media_window()  # media window
    self.create_progress_meter()  # progress meter
    self.create_buttonbox()  # media controls
    self.create_menu()  # menu bar
```

2.2 丢包率的计算

定义 2.1. *丢包率的计算公式为：*

$$\frac{\textit{Server 端发送的包的总数} - \textit{Client 端收到的包的数目}}{\textit{Client 端收到的包的数目}} \tag{1}$$

方便起见，我们定义一个包就是视频的一个帧。

Server 端发送的包的数量可以从 Server 发还 Client 端的 Reply 中获得，数值包裹在 RtpPacket 的 seqNum 字段中。

```
rtpPacket = RtpPacket()
rtpPacket.decode(data)
currFrameNbr = rtpPacket.seqNum()
```

得到 Server 端发送的包的总数后，我们与 Client 端自己收到的包的数目 (self.frameNbr) 进行比较后，其差值-1 就是丢失的包的数量，我们对此进行累加，并除以服务器当前发送的包的总数，即可得到丢包率。

```
self.packages_loss += currFrameNbr - self.frameNbr - 1
self.packages_loss_rate = self.packages_loss / currFrameNbr
```

需要注意的是，由于我们设计了快进和回退的功能。对于快进，我们在快进的同时也对 self.frameNbr 的设置进行了相应的增加，所以上述计算丢包率的方法仍然有效。对于回退 (self.rev = True)，我们则需要记录下第一次回退时 self.frameNbr 的数值 self.record_rev_frame，再将其减小，并暂停丢包率的计算。等 self.frameNbr 的值大于等于 self.record_rev_frame 时，我们再继续丢包率的计算，从而避免了丢包率出现负数的情况。

```
if not self.rev:
if currFrameNbr - self.frameNbr >= 1:
    self.packages_loss += currFrameNbr - self.frameNbr - 1
    packages_loss_list.append((curtime - start_time, self.packages_loss))
    self.packages_loss_rate = self.packages_loss / currFrameNbr
    packages_loss_rate_list.append((curtime - start_time, self.packages_loss_rate))

elif self.frameNbr >= self.record_rev_frame:
self.rev = False
if currFrameNbr - self.frameNbr >= 1:
```

```
        self.packages_loss += currFrameNbr - self.frameNbr - 1
        packages_loss_list.append((curtime - start_time, self.packages_loss))
        self.packages_loss_rate = self.packages_loss / currFrameNbr
        packages_loss_rate_list.append((curtime - start_time, self.packages_loss_rate))
```

2.3 帧率计算

该功能的实现主要在于 Client 端。

要计算每秒帧率，首先我们需要一个全局运行的计时单元，这里我们在初始化时引入 time 包，直接初始化时钟，同时也要计数接收到的比特数：

```
self.time = time.time()
self.totalBytes = 0
self.bits_per_second = 0
```

我们接收数据部分在 listenRTP() 函数中完成。首先当接收到的数据不为空包时，我们记下此时的时间间隔，并修改接收到的数据总长度：

```
if data:
    curtime = time.time()
    one_transimit_Bytes = len(data)
    self.totalBytes += len(data)
    total_bytes_list.append((curtime - start_time, self.totalBytes))
```

随后当时间每过 0.1s，我们便计算一次这段时间内数据的增量并输出：

```
if time.time() - self.time > 0.1:
    self.bits_per_second = one_transimit_Bytes * 8 / (time.time() - self.time)
    bits_per_second_list.append((curtime - start_time, self.bits_per_second))
```

2.4 DESCRIBE 的实现

在多媒体系统中，媒体流（media stream）是一种连续的数据流，用于传输音频、视频或其他类型的多媒体内容。常见的媒体流类型有：1. 音频流（Audio Stream）：音频流包含连续的音频数据，通常使用音频编解码器（如 AAC、MP3 等）进行编码和解码。2. 视频流（Video Stream）：视频流包含连续的视频数据，通常使用视频编解码器（如 H.264、VP8、VP9 等）对视频帧进行编码和解码。3. 文本流（Text Stream）：文本流包含字幕、字幕或其他同步文本数据。在某些情况下，文本流也可能包含元数据，例如实时事件信息或其他媒体相关数据。4. 数据流（Data Stream）：数据流用于传输与多媒体内容相关的其他数据，例如携带有关媒体内容的元信息，或者在实时应用中传输用户之间的交互信息。

在收到 DESCRIBE 请求时，服务器会返回一个会话描述文件，这个文件告诉客户端会话中包含哪些类型的媒体流，以及所使用的编码方式。这使得客户端能够正确地接收和解码所接收到的媒体流，从而实现多媒体内容的播放。

经过研究，我们发现这是通过一种叫做 SDP 的协议实现的。

2.4.1 Session Description Protocol

定义 2.2. *SDP（Session Description Protocol）是一种会话描述协议，用于描述多媒体会话的参数和信息。它通常用于 VoIP（Voice over IP）和视频会议等应用中，可以传输会话相关的音频、视频、文本等媒体类型的信息。*

- v：表示协议版本号，当前通常为 0。
- o：表示会话发起者的标识符和会话 ID。格式为 “username session id version network type address type address”。
- s：表示会话名称，通常为人类可读的字符串。
- i：表示会话信息，通常是一个非正式的会话说明。
- u：表示会话 URI（Uniform Resource Identifier），用于唯一标识会话。
- e：表示会话中的电子邮件地址。
- p：表示会话中的电话号码。
- c：表示媒体连接信息，包括网络类型、地址类型和连接地址等信息。
- b：表示带宽需求，包括总体带宽需求和每个媒体流的带宽需求。
- t：表示会话时间，包括开始时间和结束时间。
- r：表示会话重复规则，例如重复次数和间隔时间。
- z：表示会话中的时区调整规则。
- k：表示加密密钥信息，用于保护会话内容。
- a：表示会话属性，例如编解码器类型、流格式、媒体格式等信息。
- m：表示媒体流信息，包括媒体类型、传输协议、端口号、编解码器类型和格式等信息。

以上是 SDP 协议常用的格式，不同应用场景中可能会有一些自定义的格式。SDP 协议通过这些格式来描述会话相关的信息和参数，使得不同设备和系统之间可以相互识别和交流，从而实现多媒体会话的传输和展示。

2.4.2 Client 端

我们在 Client 端定义如下的 DESCRIBE 请求

```
# Describe request
elif requestCode == self.DESCRIBE and not self.state == self.INIT:
    # Update RTSP sequence number.
    self.rtspSeq = self.rtspSeq + 1

    # Write the RTSP request to be sent.
    request = "DESCRIBE " + str(self.fileName) + "\nCseq: " + str(self.rtspSeq) \
              + "\n" + "RTSP/1.0 RTP/UDP " + str(self.rtpPort)

    # Keep track of the sent request.
    self.requestSent = self.DESCRIBE
```

并添加专门用于解析 SDPdescription 报文的函数：

```
def processSdpDescription(self, sdpDescription):
    """Process the SDP description to get information about the video stream."""

    lines = sdpDescription.split('\r\n')

    for line in lines:
        parts = line.split('=')
        if len(parts) == 2:
            fieldType = parts[0].strip()
            fieldValue = parts[1].strip()

            if fieldType == 'm':
                # Extract the RTP/AVP payload type, session type
                self.sessionType = fieldValue.split(' ')[0] # session 的种类
                self.payloadType = int(fieldValue.split(' ')[-1]) # 媒体格式

            elif fieldType == 'a' and fieldValue.startswith('rtpmap'):
                # Extract the encoding format and clock rate
                rtpmap_parts = fieldValue.split(' ')[1].split('/')
                self.encodingFormat = rtpmap_parts[1]
```

2.4.3 Server 端

在 ServerWorker.py 文件中，我们根据 SDP 协议的格式要求，定义 SDPdescription

```
elif requestType == self.DESCRIBE:
    # SDP 格式
    sdpDescription = "v=0\r\n" \
                     "o=- 0 0 IN IP4 {serverAddress}\r\n" \
                     "s={streamName}\r\n" \
                     "c=IN IP4 {clientAddress}\r\n" \
                     "m=video {clientRtpPort} RTP/AVP 26\r\n" \
                     "a=rtpmap:26 {mjpegType}/90000\r\n"

    # Replace placeholders with actual values
    serverAddress = self.clientInfo['rtspSocket'][1][0]
    clientAddress = self.clientInfo['rtspSocket'][0].getpeername()[0]
    streamName = self.clientInfo['videoStream'].get_filename()
    clientRtpPort = self.clientInfo['rtpPort']
    mjpegType = self.MJPEG_TYPE

    sdpDescription = sdpDescription.format(
    serverAddress=serverAddress,
    streamName=streamName,
    clientAddress=clientAddress,
```

```
        clientRtpPort=clientRtpPort,
        mjpegType=mjpegType)

        # Send the DESCRIBE response with the SDP description
        print("Sending DESCRIBE response:\n" + sdpDescription)
        self.replyDescribe(sdpDescription, seq[1])
```

然后我们将它封装进 socket 套接字中发回 Client 端：

```
def replyDescribe(self, sdpDescription, cseq):
    # Send the RTSP response with the SDP description for the DESCRIBE request
    reply = 'RTSP/1.0 200 OK\nCSeq: {}\n'.format(cseq) + \
            'Session: ' + str(self.clientInfo['session']) + '\n' + \
            'Content-Type: application/sdp\r\nContent-Length: {}\r\n\r\n{}'.format(
            len(sdpDescription),
                sdpDescription)
    print(reply)
    self.clientInfo['rtspSocket'][0].send(reply.encode())
```

从而实现了 SDP 在 Client 端和 Server 端互送信息。下图中是 Client 端从 Server 端收到的相关信息展示：

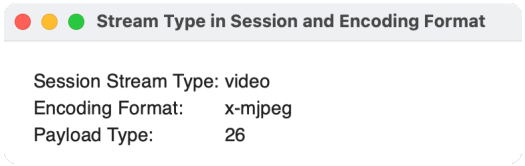

图 5: DESCRIBE 请求返回的信息

2.5 PLAY 与 SETUP 按钮合并

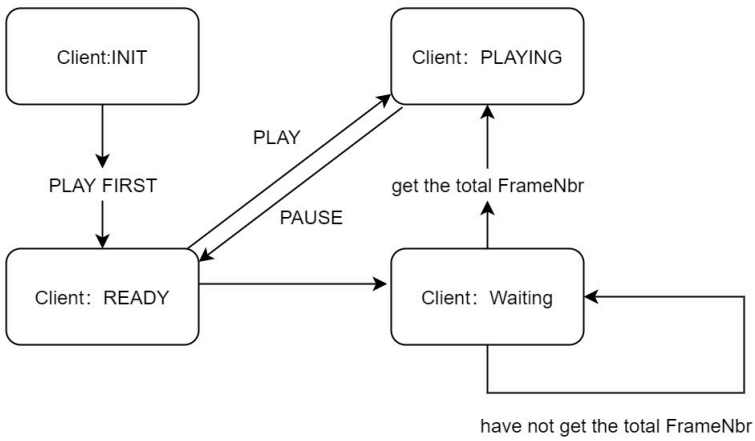

图 6: SETUP 与 PLAY 合并的有限状态机

该功能的实现主要涉及 Client 端的代码迭代：
首先，将 UI 界面的设计函数中将 SET UP 按钮删去；
为实现SETUP 与PLAY的功能合并，引入了一个判断是否是第一次触发 PLAY 按钮的标识符，

然后需要在PLAY的 command 触发函数中，增加一个关于是否是第一次触发 PLAY 按钮的判断分支，如果是则标志符改为 True，并执行原来SETUP的功能：即发送SETUP 请求。

为了保证 Clinet 在执行完SETUP功能之前，不会进入到 PLAYING 状态，需要设置一个 while 停顿，当SETUP功能执行完，再执行原来的 PLAY 功能：即建立 playEvent 的状态，然后发送PLAY 请求。

```
if self.first_play & self.state == self.INIT:
    self.first_play = False
    self.sendRtspRequest(self.SETUP)

while self.state == self.INIT:
    print("state not change")

if self.state == self.READY:
    # Create a new thread to listen for RTP packets
    threading.Thread(target=self.listenRtp).start()
    self.playEvent = threading.Event()
    self.playEvent.clear()
    self.sendRtspRequest(self.PLAY)
```

2.6 FORWAND 按钮：快进功能的实现

该功能的实现的基本过程：Client 端发送快进请求；Server 端接收解析请求，并指示 Vedio Steam 进行快进；vedio Stream 的快进功能实现：

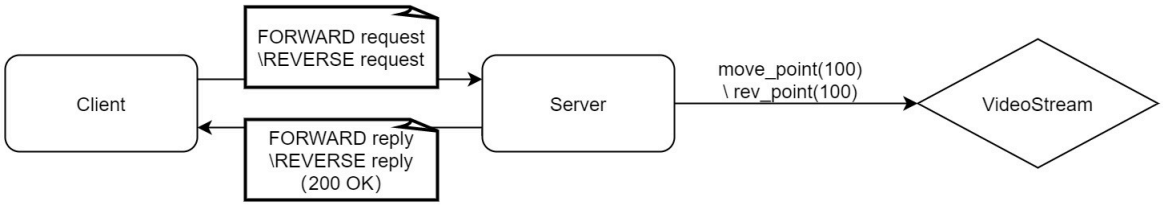

图 7: 快进/回退流程图

2.6.1 Client 端

将 UI 界面的设计函数设置 FORWARD 按钮. 同时设置 FORWARD 触发功能，当 FORWARD 被触发时，发送 FORWARD 请求。

同时当接收到 FORWARD 回复后，需要对 Client 的 FrameNbr 计数器进行调整。

```
def playfwd(self):
    """Fwd button handler."""
    if self.state == self.PLAYING:
        # send a request for fwd
        print("put the fwd")
        self.sendRtspRequest(self.FWD)
```

2.6.2 Server 端

当接收到 FORWARD 的请求时，Server 指示 vedio stream 进行读取文件指针向前跳转，总共跳转 100 帧。

```
elif requestType == self.FWD:
    print("processing FORWARD\n")
    self.clientInfo['videoStream'].movepoint(100)
```

2.6.3 VideoStream

帧数跳转功能的实现，关键在于理解.Mjpeg 文件度读取格式：

.Mjpeg 中图片的每一帧，都有一个 5bit 的整数表示下一帧图片对应的比特数作为前缀，后接实际的图片数据。

因此在读取时，需要先对前五个 bit 的数据进行读取，解析下一帧对应的比特数。然后据此进行跳转或读取。

```
def movepoint(self,n):
    for i in range(n):
        data = self.file.read(5)
        if data:
            framelength = int(data)
            # 存储 bit 长度到 stack，为了后续实现回退
            self.data_length_stack.append(framelength)
            data = self.file.seek(framelength,1)
            self.frameNum += 1
```

2.7 REVERSE 按钮：回退功能的实现

该功能的实现的基本过程：Client 端发送回退请求；Server 端接收解析请求，并指示 Vedio Steam 进行回退；vedio Stream 的回退功能实现：

2.7.1 Client 端

将 UI 界面的设计函数设置 REVERSE 按钮. 同时设置 REVERSE 触发功能，当 REVERSE 被触发时，发送 REVERSE 请求。

同时当接收到 REVERSE 回复后，需要对 Client 的 FrameNbr 计数器进行调整。

```
def playrev(self):
    """REV button handler."""
    if self.state == self.PLAYING:
        # send a request for rev
        print("put the rev")
        self.sendRtspRequest(self.REV)
```

2.7.2 Server 端

当接收到 REVERSE 的请求时，Server 指示 vedio stream 进行读取文件指针回退跳转，总共跳转 100 帧。

```
        elif requestType == self.REV:
            print("processing REVERSE\n")
            self.clientInfo['videoStream'].revpoint(100)
```

2.7.3 Video Stream

帧数回退跳转功能的实现，由于.Mjpeg 的格式限制，我们不能通过指针所在的位置，判断先前的视频帧对应的 bit 长度。

因此，我们设计了一个 stack，用于当视频往前读取或者跳转时，存储每一帧的 bit 长度。在进行回退时，在读取其中的 bit 长度，进行跳转。

```
def __init__(self, filename):
    self.filename = filename
    # stack store the length of frames
    self.data_length_stack = deque()

def revpoint(self,n):
    framelengths = 0
    for i in range(n):
        # 避免回退过度
        if self.frameNum == 3 :
            break
        framelength = self.data_length_stack.pop()
        framelengths += (framelength + 5)
        self.frameNum -= 1

    self.file.seek(-framelengths,1)
```

2.8 统计总帧数

为了判断视频的播放进度，在视频播放之前对视频总帧数的统计具有重要意义，基本过程：

Client 发送 SETUP 请求，Server 接收到 SETUP 请求，进行初始化操作，并指示 Video 统计总帧数，由 Video Stream 统计总帧数、最后通过 Sever 向 Client 发送视频总帧数的统计结构，再由 Client 解析来自 Server 的数据包，并存储结果。

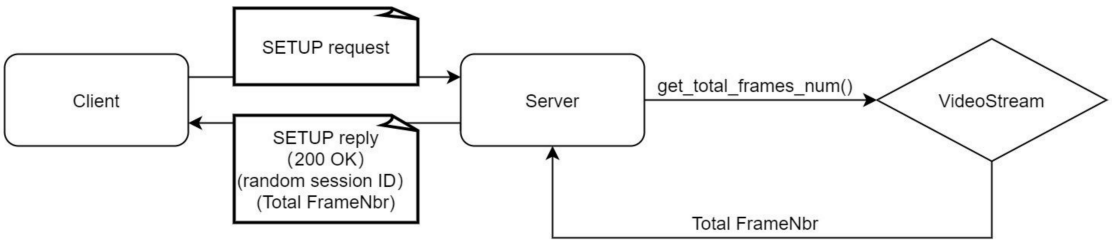

图 8: Client 获取总帧数流程图

2.8.1 Client 端

发送 SETUP 请求过后，需要稍作等待，获得总帧数再进行接下来的操作。

```
        if self.first_play & self.state == self.INIT:
            self.first_play = False
            self.sendRtspRequest(self.SETUP)

        # 稍作等待
        while self.total_frames == 0:
            print("have not get the total frames")
```

解析来自 Server 的含有总帧数数据包，并将它存储在 total_frames 的属性中。这里数据包的解析格式是参照 RTSP 的基本格式，自主设计的：以 Total 作为 TAG, 后接视频总帧数。

```
    if self.requestSent == self.SETUP:
        self.state = self.READY

        self.total_frames = int(lines[3].split(' ')[1])
        # Open RTP port.
        self.openRtpPort()
```

2.8.2 Server 端

Server 接收到 SETUP 请求后，指示 Vedio 统计总帧数，并发送视频总帧数数据包

```
# 收到 SETUP 请求后
if requestType == self.SETUP:
    if self.state == self.INIT:
        print("Sending Total number of Frames")
        # Update state
        print("processing SETUP\n")

        try:
            self.clientInfo['videoStream'] = VideoStream(filename)
            self.state = self.READY
        except IOError:
            self.replyRtsp(self.FILE_NOT_FOUND_404, seq[1])
        total_frames_num = self.clientInfo['videoStream'].get_total_frames_num()
        # Generate a randomized RTSP session ID
        self.clientInfo['session'] = randint(100000, 999999)

        # Send RTSP reply
        self.SendTotalFrame(total_frames_num,seq[1])

# 发送数据包
    # 用于发送总的帧数
    def SendTotalFrame(self,total_frames_num,cseq):
        reply = 'RTSP/1.0 200 OK\nCSeq: {}\n'.format(cseq) + \
                'Session: ' + str(self.clientInfo['session']) + '\n' + \
```

```
                'Total: {}'.format(total_frames_num )

        print(reply)
        self.clientInfo['rtspSocket'][0].send(reply.encode())
```

2.8.3 VideoStream

依据.Mjpeg 的文件格式，进行总帧数统计，统计后，将文件指针回复到文件开头。

```
        # 统计总的帧数
        self.total_frameNum = 0
        tmp_length = self.file.read(5)  # Get the framelength from the first 5 bits

        while tmp_length:
            framelength = int(tmp_length)
            # Read the current frame
            self.file.seek(framelength,1)

            self.total_frameNum += 1
            tmp_length = self.file.read(5)
        self.file.seek(0,0)
```

2.8.4 数据可视化

为了更直观地呈现相应信息的变化情况，我们对获取的视频流信息，如丢包率和总丢包数，接收的总 byte 数和 bps 进行了相应的可视化。首先，我们在 client 文件中用变量记录视频播放过程中的数据，并存储到本地。然后在visualize.py文件中进行了相应的可视化：

```
def visualize_data():
    with open('total_bytes_list.pickle', 'rb') as f:
        total_bytes = pickle.load(f)
    with open('packages_loss_list.pickle', 'rb') as f:
        packages_loss = pickle.load(f)
    with open('packages_loss_rate_list.pickle', 'rb') as f:
        packages_loss_rate = pickle.load(f)
    with open('bits_per_second_list.pickle', 'rb') as f:
        bits_per_second = pickle.load(f)

    plt.subplot(221)
    plt.plot([packages_loss_rate[i][0] for i in range(len(packages_loss_rate))],
             [packages_loss_rate[i][1] * 100 for i in range(len(packages_loss_rate))])
    plt.title("Packages Loss Rate")
    plt.ylabel("rate/%")
    plt.subplot(222)
    plt.plot([packages_loss[i][0] for i in range(len(packages_loss))],
```

```
        [packages_loss[i][1] for i in range(len(packages_loss))])
plt.title("Packages Loss")
plt.subplot(212)
plt.plot(
[total_bytes[i][0] for i in range(len(total_bytes))],
[total_bytes[i][1] for i in range(len(total_bytes))]
)
plt.plot([bits_per_second[i][0] for i in range(len(bits_per_second))],
         [bits_per_second[i][1] for i in range(len(bits_per_second))])
plt.xlabel("Time/s")
plt.title("Video Stream")
plt.ylabel("bytes")
plt.legend(["total_bytes", "bits_per_second"])

# 增加子图之间的垂直间距
plt.subplots_adjust(hspace=0.5)
plt.tight_layout()  # 自动排版
plt.savefig('result.png')
plt.show()
```

我们用组里四位同学的电脑进行实验，得到了如下数据：

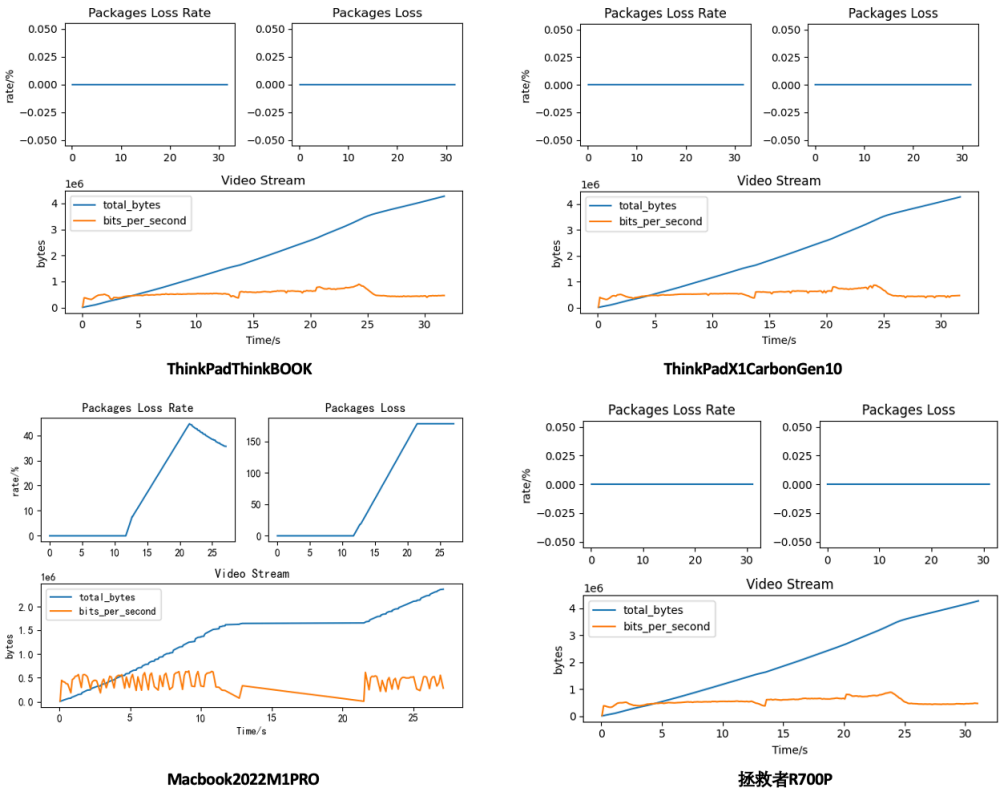

图 9: 不同电脑上视频传输信息的可视化

从图中可以看出，3 台 Windows 系统的电脑在丢包率和帧率方面的表现均十分优秀，基本没有出现丢包的情况，且帧率基本稳定在 5×10^5 bps。考虑到传输的文件并不庞大，以及本次实验的传输与接收是

在同一台电脑上实现的，这种结果是合理的。然而，在 mac 系统上，传输过程中出现了一段时间的丢包过程，经过多次尝试，这种情况一直存在，且帧率出现较大的波动。

我们猜想，这种现象可能与 mac 电脑的网络架构有关。不同操作系统之间网络处理的方式和机制可能会有所不同，导致了在 mac 电脑上出现了丢包和帧率波动的情况。此外，也有可能是 mac 电脑的硬件性能或者其他软件方面的因素导致的。因此，我们或许需要进行更加深入的分析和测试才能确定具体原因。

3 总结

在本次流媒体传输实验中，我们学习了 RTSP 和 RTP 协议，并在此基础上实现了流媒体的传输功能。此外，我们还完成了帧率、丢包率等信息的计算与实时展示，PLAY 和 SETUP 的合并，并且设计了新的 UI 界面。此外，我们还实现了对视频进度的实时展示，以及视频的快进和回退功能。最后，我们还对相关数据进行了可视化操作。

经过这次实验，我们对流媒体传输协议 RTSP 和实时传输协议 RTP 有了更深的认识。我们回顾和实现了帧率和丢包率的计算。而为了实现 DESCRIBE 功能，我们还研究了 SDP 协议的使用与实现。

此次大作业让我们学习到诸多新知识，锻炼了我们的自学能力、设计能力和编程能力，给予了我们许多宝贵的经验。

4 小组分工

组长 黄奔皓, 负责播放器 UI 设计与功能整合，实现丢包率的计算，`DESCRIBE`方法以及数据可视化的额外功能。负责报告中相应部分的写作、参与海报的制作。

组员 1 朱鹏翔, 负责本实验基础功能的实现（基于 RTP 与 RTSP 流媒体传输的基本实现），负责报告中相应部分的写作。

组员 2 李佳鑫, 负责`PLAY`按钮和`SETUP`按钮的合并，`FORWARD`和`REVERSE`功能的实现，以及视频总帧数的统计与传输。负责报告中相应部分的写作、参与海报的制作。

组员 3 周晟洋, 负责每秒帧率的计算及报告相应部分和海报制作。

5 代码

本实验的所有代码均可从：https://github.com/huskydoge/CS3611-videoStreaming-player 获取。