Kaleidoscope: A Cloud-Based Platform for Real-Time Video-based Interaction

by

Hu Zhang

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

© Hu Zhang, 2016

Abstract

Mobile video streaming becomes increasingly useful in a variety of contexts (social interaction, education and entertainment) and increasingly feasible with the rapid development of wireless networks and mobile technologies. In this thesis, we develop a platform for multimedia streaming on mobile devices, enhanced with textual and touch-display interactions for a rich user experience. Users (Senders) can use our Kaleidoscope mobile application to set up a streaming channel on the platform server, and invite their contacts (Receivers) to share their real-time video recordings. At the Sender site, the Kaleidoscope app captures the video and shares it with the streaming server. The streaming server saves the multimedia streams into files. At the Receiver site, the Kaleidoscope app replays the video. At both sites, users can send text messages to the connected peers and touch the display to point out interesting video scenes; the Kaleidoscope app shares these interactions with all the peers. The data (audio/video, text, touch events) is stored on the cloud server with timestamps to support feature extraction and analytic services on the cloud. We evaluated the Kaleidoscope system on the SAVI cloud at multiple locations, testing the CPU and memory usage of Kaleidoscope streaming server with different numbers of clients, in different locations.

“MAT VICTORIA CURAM”

— Latin Proverbs

VICTORY LOVES PREPARATIONS

Acknowledgements

With two and half years of my graduate studies experience, there are so many words to say and so many things to be thankful.

Firstly, I'd like to thank my supervisor Dr. Eleni Stroulia for her patient instructions and advice, sincerely. She is a so nice professor. She gave me a great help in my graduate study.

I also want to thank my colleagues in SSRG lab and my friends in Canada.

Thanks to my family and especially my Chang Chen. Thanks for their unselfish love and emotional support.

Finally, I wish all the best to all of you, love you.

Table of Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Contributions 4
1.3 Outline 4

2 Background **6**
2.1 Media Streaming 6
2.1.1 Media Streaming Categories 6
2.1.2 Media Streaming Protocols 7
2.2 Instant Messaging 10
2.2.1 Extensible Messaging and Presence Protocol 10
2.2.2 Authentication and Security 13
2.3 Compression and Codec 14
2.4 Literature Review 16

3 The Kaleidoscope System **19**
3.1 System Overview 19
3.1.1 Architecture 20
3.1.2 The Overall Kaleidoscope Process 25
3.2 Communication and Behavior 27
3.2.1 Clients and Social Interaction Service 27
3.2.2 Clients and Streaming Service 33
3.2.3 Touch-display Feature 38
3.3 Storyboard 40

4 Evaluation **44**
4.1 Experiment Setup 44
4.2 Results 46

5 Conclusions **51**

Bibliography **53**

List of Tables

2.1 An audio-only SDP example. 10

3.1 An example of presence broadcast and response between clients and server. 29
3.2 An XMPP example of retrieving friend list from server. 29
3.3 The chat room configuration request and response between clients and server. The value of 0 means false and 1 means true. . . . 30
3.4 The Sender sends invitation to a Receiver. 31
3.5 The Receiver gets the invitation from the Sender. 31
3.6 Server response containing the Receiver's role in the chat room. 32
3.7 The clients communicate in a chat room. 32
3.8 The Receiver leaves chat room. 33
3.9 The Sender destroys the room. 33

4.1 Hardware configuration of the devices used in our experiment. 45
4.2 Results of Edmonton-Edmonton-Edmonton experiment. 47
4.3 Results of Edmonton-Calgary-Edmonton experiment 47
4.4 Results of Edmonton-Toronto-Edmonton experiment. 47
4.5 Results of Edmonton-Calgary-Calgary experiment. 47
4.6 Results of Edmonton-Edmonton-Edmonton with higher video quality experiment. 47
4.7 Results of number of Receivers in multi-locations. 48

List of Figures

1.1 The SAVI Infrastructure (The figure is from `http://www.savinetwork.ca/news-events/poster-booklet-of-savi-agm-workshop-2015/`). 3

3.1 Kaleidoscope system interaction diagram. On the *Sender* side a user can create a chat room, invite friends to join and share a video streaming with friends in that chat room. On the *Receiver* side a user can accept the invitation, join the chat room and play the video streaming in real-time. On both Sender and Receiver sides users can communicate by sending text messages or touching the video screen to annotate video content to share with each other. 20
3.2 The libVLC library view. 21
3.3 XMPP Client Library aSmack liabrary view. 22
3.4 ESS project view. 23
3.5 The libstreaming library view. 24
3.6 Communication between Sender, social interaction server and Receiver. When explaining these messages below, we use *Si* to refer to Sender side's step i, and *Ri* to refer to the Receiver side's step i. 27
3.7 Streaming communication between Sender, streaming server and Receiver. When explaining these messages below, we use *Si* to refer to Sender side's step i, and *Ri* to refer to the Receiver side's step i. 34
3.8 The touch-display feature on a Sender and Receiver side. . . . 39
3.9 Configuration interface. 41
3.10 Friend list interface. 41
3.11 The Receiver gets an invitation from Sender. 42
3.12 The Receiver touches the screen to add some tags. 42
3.13 The Sender receives touch-display annotation. 43
3.14 The Receiver gets a destroy-room notification from Sender. . . 43

Chapter 1

Introduction

In this thesis, we deal with the problem of video streaming on mobile devices. Our contribution is a platform that supports video streaming and sharing through a mobile client, enhanced with social interaction. In this chapter we introduce the motivation and objectives of this work, we summarize the contributions, and give an outline of the thesis.

1.1 Motivation

In recent years, the proliferation of social-networking sites and content sharing applications have enabled a sudden increase in video traffic. According to Cisco's forecast [1], consumer Internet video traffic will increase up to 80 percent of all consumer Internet traffic by 2019. The increasing video traffic is accompanied by a fast growth of the mobile industry, producing widespread adoption and providing capabilities for new kinds of content and interactions, especially high-quality video streaming. The Pew Research Center estimated that around 68 percent of American adults have a smartphone in 2015. In contrast, only 33 percent of the same population owned a similar smart device in 2011 [2]. In fact, mobile video traffic exceeded 50 percent of the total mobile data traffic in 2014 [3]; and this number is predicted to grow to 72 percent by

[1] http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper.html

[2] http://www.pewinternet.org/2015/04/01/us-smartphone-use-in-2015/

[3] http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html

the end of 2019. This growth momentum will greatly reinforce the widespread of mobile video applications.

Many well-known video-sharing applications emerged. Likely the most successful and well known among these applications is YouTube [4]. YouTube allows users to upload, view, and share videos. The available content on YouTube includes video clips, TV clips, and live events. Vevo and Hulu [5] offer similar kinds of content. All the aforementioned applications serve on-demand videos, whose content has already been uploaded to the Internet. Furthermore, user interaction with the content is limited to asynchronous social-network-agnostic actions, such as adding comments.

More recently, the high adoption of mobile smart devices and broadband gave rise to a new breed of applications, with rich media, such as video, audio, or graphics elements. These applications encourage users to interact and engage with the content, frequently based on the user's social context. For example, Snapchat [6] is a mobile application, where users can share personal photos, short videos, text and drawings with a controlled list of recipients. Although Snapchat is a context-aware application, its design is limited to only on-demand videos of about 10 seconds duration.

Very recently, Periscope [7] offered live video streaming, in which users can follow other users, called broadcasters, and view their live videos. In Periscope, users can interact by sending messages or "hearts" as a form of appreciation to broadcasters during a chat. Although the above applications are popular and widely used, the interactions are limited to monolithic pieces of content, and ignore the temporal dynamics of such content. These messages are synchronous with the video streaming but are not directly related to video content itself.

Under these premises, in this thesis we introduce Kaleidoscope, a scalable context-aware mobile video streaming and sharing platform, enhanced with textual and touch-display interactions for a rich user experience.

[4] YouTube: `https://www.youtube.com/`
[5] Vevo: `http://www.vevo.com/`, Hulu: `http://www.hulu.com`
[6] Snapchat: `https://www.snapchat.com/`
[7] Periscope: `https://www.periscope.tv/`

Our platform can be deployed in a hierarchical cloud, such as the "Smart Applications on Virtual Infrastructures" (SAVI) cloud [8] which consists of multiple smart edges, geographically close to end users, and a (set of) core node(s) (Figure 1.1). A smart edge is a computing infrastructure that provide heterogeneous virtualized resources and services close to end-users. In contrast, the core node provide resources at a massive scale and services available to all end-users, agnostic from their location.

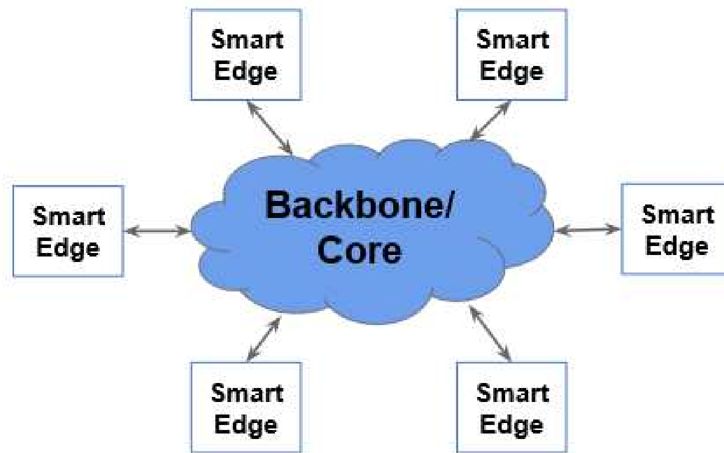

Figure 1.1: The SAVI Infrastructure (The figure is from `http://www.savinetwork.ca/news-events/poster-booklet-of-savi-agm-workshop-2015/`).

With the SAVI infrastructure, our platform can potentially offer low latency by assigning broadcasters to the smart edge that is closest to viewers. In addition, the platform can automatically optimize the entire life cycle of the the applications, such as the topology of the servers and devices in the infrastructure, based on available resources such as CPU or memory. Finally, we would be able to use the core, with its powerful computation and ample storage resources, in order to provide persistence for the user-generated content, and perform content analysis such as extracting knowledge from videos, text and annotations in parallel.

[8] SAVI: `http://www.savinetwork.ca/`

1.2 Contributions

Developing a stable real-time video sharing platform is very challenging. First, streaming videos to allow real-time playing on multiple mobile devices is challenging because different mobile devices support different audio/video encoding. Second, these systems are architecturally complex, consisting of senders (users who share their video), servers (streaming server and interaction server), and receivers (users who receive the shared videos); an effective system should implement these components and also the communications between them.

In this thesis, we developed a platform (Kaleidoscope) for multimedia streaming sharing on Android devices, enhanced with textual and touch-display interactions. With the Kaleidoscope, users can sign in and create"chat rooms" to share videos in real time. A novel feature of Kaleidoscope is that during a video sharing process, users can touch the screen and annotate the video content and they can discuss the video through text messaging. We have implemented three services. The streaming service is responsible for receiving audio/video streaming data from senders, and transmitting these data to the receivers. The social-interaction service takes care of user management, video-chat room management and instant messaging. The storage service saves audios/videos into files with timestamps to support extended services such as video content analysis and automatic feature extraction. We evaluate the Kaleidoscope platform on SAVI and Cybera clouds. Our empirical results show that the success connection rate is very high whether the clients and servers are in different locations such as Edmonton, Calgary and Toronto.

1.3 Outline

In Chapter 2, we present a background of media streaming, social activity interactions and related works. In Chapter 3, we begin with a system overview of Kaleidoscope. We describe the components and services as well as the flow of video sharing with textual and touch-display social interactions. We also give details of communication between clients (Sender and Receiver) and

servers (social interaction server and video streaming server). We evaluate our platform by testing it on the SAVI and Cybera clouds with different number of requesting loads in Chapter 4. Finally, in Chapter 5, we discuss our contributions and lay out our plans for future work.

Chapter 2

Background

In this chapter, we review media streaming, including on-demand streaming and live streaming. Then we introduce related protocols and technologies of media streaming in Section 2.1. Next we describe the social-interactions technologies used in our system (Section 2.2). In Section 2.3 we introduce two series of audio/video compression and codec algorithms. Finally, we discuss related works in Section 2.4.

2.1 Media Streaming

2.1.1 Media Streaming Categories

Media Streaming is a technology that presents data to end-users over the internet continuously at the same time of data transmission [1, 16, 25]. During the transmission, the media information is packed into data packets and sent to the client. The media information can be text, audio or video.

There are two types of media streaming [20], including on-demand streaming and live streaming. *On-demand streaming* is a traditional way which users cannot watch the video until the whole resource is downloaded or enough data is cached. With on-demand streaming, users can pause or stop the stream, playback or fast forward it, even jump to a specific time point. Conversely, with *live streaming* the media is sent in a continuous way to clients and it is played as it arrives in real-time. Based on the fact that live streaming transmits data in real time, there are more limitations than on-demand streaming. For example, live streaming allows users to go back to a history point, but

they can not fast forward to a future point.

Our work is focused on live streaming. When we use “video streaming” or “media streaming” in the remainder of this thesis, it refers to live streaming.

2.1.2 Media Streaming Protocols

In past decades, many real-time video streaming protocols and mechanisms for mobile devices have been implemented, such as the Real Time Streaming Protocol [32] (RTSP) and Real Time Messaging Protocol [26] (RTMP). These protocols make it possible for users to share audio and video streams over the network and play them in real-time.

In this thesis, we choose RTSP to implement video streaming. For real-time media streaming on mobile devices, RTMP has a few disadvantages. RTMP is a TCP-based protocol, which supports retransmission for lossless data communication. That means system designers need to come up with a good missing-data retransmission strategy to avoid audio and video delay, jitter and asynchronization because of the data-packet loss. Furthermore, RTMP uses different protocols/ports from HTTP [7], which makes it vulnerable to getting blocked by firewalls and it only works in Flash [18].

Real-time Transport Streaming Protocol (RTSP)

RTSP is a text-based application-layer protocol. It plays the role of “network remote control” in multimedia services such as audio and video in real-time. In the media streaming transmission, RTSP involves a few basic protocols for data transmission, media control, and media content description. Most RTSP servers use Real-time Transport Protocol (RTP) as a delivery mechanism to transmit the data stream by building a Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) connection as a media streaming delivery channel. RTSP defines a series of commands for communication of clients and streaming servers including OPTIONS, ANNOUNCE, DESCRIBE, SETUP, PLAY, PAUSE, REDIRECT, RECORD AND TEARDOWN. When a client starts a media streaming session to a streaming server through RTSP commands, they use the Session Description Protocol (SDP) to exchange media details, transport addresses, and other session description

metadata.

Transmission Control Protocol (TCP)

TCP is a connection-oriented and reliable byte-stream protocol [27]. The sender and receiver should build a TCP connection before data transmission. TCP is a transport layer protocol. It has a series mechanisms to ensure the transmission reliability. Three-way handshake, also referred to as "SYN-SYN-ACK", is needed before transmitting data. During data transmission, each data segment should be acknowledged by the receiver. If the sender does not get the acknowledged information from the receiver in a predetermined time, TCP will use a retransmission timeout mechanism to qualify non-loss data and use sequence acknowledgement numbers to make sure the data segment is in the right order. In order to control the flow of data and to improve transmission efficiency, TCP has a sliding window mechanism and uses slow-start algorithm to avoid network congestion.

User Datagram Protocol (UDP)

UDP is a connectionless and unreliable protocol [28]. Similar to TCP, UDP is also a transport layer protocol. UDP is only responsible for sending and receiving the datagram, but it does not guarantee the datagram is received by the destination after sending the datagram. As a result, the data can be received out of order, or even lost. Compared to TCP, UDP is faster because UDP has no flow control, no error checking and no datagram acknowledged mechanisms. Therefore, UDP is often used by multimedia applications for transmitting the data stream, such as audio and video streaming data because these applications are demanding on real-time response and cannot be delayed.

Real-time Transport Protocol (RTP)

RTP is an Internet transport protocol that manages real-time multimedia data streams transmission [33]. It is defined as working in the transport layer, built on the top of UDP. The purpose of RTP is to provide time information and to synchronize multiple streams. RTP only guarantees real-time data transmission but it does not support a reliable transport mechanism for the transmission of data packets in order. Furthermore, it does not provide flow control and congestion control, which rely on Real-time Transport Con-

trol Protocol (RTCP). RTP provides a timestamp, serial number, and other structures to manage the real-time streaming data. After receiving data packets, the clients recover these data packets in an original order according to the RTP header information which tells the clients how to recover the data packets and how the codec bit streams are unpacked. RTP header information contains timing information, sequence number, payload identification, frame indication, source identification, intramedia synchronization, etc.

Real-time Transport Control Protocol (RTCP)

RTCP is a RTP control protocol [24]. RTCP is responsible for managing transmission quality between applications to exchange control information on large networks, mainly for streaming media, telephone and video conferencing. During an RTP session, an application uses two adjacent ports: one for RTP, the next one for RTCP. RTCP packets are sent periodically to monitor the quality of service and transfer of users' session information and other functions. RTCP packet contains the number of packets sent, the number of lost packets and other statistics through receiver report (RR), sender report (SR), source description items (SDES), indicates end of participation (BYE) and application specific functions (APP) packet formats. Therefore, the server can use these information to dynamically change the transmission rate, and even change the payload type. RTP and RTCP work together to minimize transmission overhead and optimize efficiency.

Session Description Protocol (SDP)

SDP is used for describing multimedia sessions [10]. It serves for session announcement, invitation and what other forms. SDP does not support the negotiation operation of SDP session content or media encoding.

When initiating audio/video streaming, video conference, or other sessions, there is a requirement to convey media details, transport addresses, and other session description metadata to the participants. SDP provides a standard representation for such information such as session name and objectives, session time, media session and so on.

Table 2.1 shows an audio-only SDP example.

Table 2.1: An audio-only SDP example.

```
v=0
o=- 0 0 IN IP4 127.0.0.0
s=Unnamed
i=N/A
c=IN IP4 0.0.0.0
t=0 0
a=recvonly
a=control:*
m=audio 0 RTP/AVP 96
a=3GPP-Adaptation-Support:1
a=rtpmap:96 AMR/8000
a=fmtp:96 octet-align=1;
a=control:trackID=0
```

In Table 2.1, the parameter v means protocol version; o is originator and session identifier; s is the session name; i is the session information (optional); c is the connection information (optional); t is the time description and time the session is active; a means attribute lines (optional). m means media name and transport address. If a media includes video stream, the m tag looks like "m=video 0 RTPAVP 96", optionally followed by some attribute lines.

There are some other optional parameters. For more details, please refer to [10].

2.2 Instant Messaging

In this section, we give a description of technologies we used in instant messaging (IM) with touch-display interactions.

2.2.1 Extensible Messaging and Presence Protocol

Extensible Messaging and Presence Protocol (XMPP) [30, 31], also known as Jabber protocol, developed in 1999 that is intended for IM and online presence detection. XMPP is based on Extensible Markup Language (XML) streaming technology. XMPP makes messaging over the internet possible, independent of operating systems and browsers. In addition, XMPP is designed to support IM tasks such as authentication, access control, end-to-end encryption, and compatibility with other protocols.

Furthermore, the XMPP Standards Foundation (XSF) develops many ex-

tensions (XEPs) [1] which make XMPP more powerful such as *roster*, *client* and *server* elements or attributes whose extended namespaces are "jabber:iq:roster", "jabber:client" and "jabber:server", respectively.

Each XMPP entity needs an address to identify itself, called Jabber ID (JID). JID format is, "node@domain/resource" where"node" can be a user name or chat room, "domain" is server name and "resource" is the entity's device identifier. A complete example: "sender@myria/Smack" where "sender" is the username, "myria" is the server name and "Smack" is the mobile identifier. There are three core elements or stanzas: <presence/>, <message/> and <iq/> in XMPP. In each of them, there are five common attributes: *from, to, id, type*, and *xml:lang*:

- *from* means the initiator of the XMPP entity sending the stream element;
- *to* indicates the JID of the intended recipient which the initiator knows;
- *id* is a unique identifier of the stream;
- *type* means the purpose or context of stanzas;

The definition of <presence/>, <message/> and <iq/> stanzas and example are below.

- The <presence/> element represents status of entities. It is a broadcast notification mechanism to notify whenever a user is online or not in IM applications. A <presence/> stanza format example is,

```
<presence id="ikRMO-3" from="sender@myria/Smack"
  to="receiver@myria/Smack">
  <status>online</status>
</presence>
```

  In this example, an *online* notification is sent from a user to another user. The *id* identifies this interaction.

- The <message/> element is used when users send messages to each other in real-time. It also supports store-and-forward asynchronous messages.

[1] XEPs: http://xmpp.org/xmpp-protocols/xmpp-extensions/

A <message/> stanza example when the owner of a chat room "`room@conference.myria`" sends an invitation to a user "`receiver@myria`" is,

```
<message from="room@conference.myria" to="receiver@myria" >
  <x xmlns="http://jabber.org/protocol/muc#user">
    <invite from="sender@myria">
      <reason>Join us receiver@myria </reason>
    </invite>
  </x>
</message >
```

In this example, an invitation is sent from a room to a user with a reason. The *xmlns* is XML schema such as `"http://jabber.org/protocol/muc#user"`. This XML schema defines a series of elements or tags such as $x, invite, item$, etc. Different XML schemas have different elements.

- The <iq/> element is used when users request from and respond to each other. A <iq/> stanza format example is,

```
<iq type="result" id="ikRMO-2" to="sender@myria/Smack">
  <query xmlns="jabber.iq:roster">
    <item jid="receiver@myria" name="Receiver"
    subscription="both">
      <group>Friends </group>
    </item>
  </query>
</iq>
```

In this example, this is friends query response from server to user. The *type="result"* which means this message is a response from server; the *to="sender@myria/Smack"* means the destination; the *query* element means this is a query result; the *xmlns* attribute indicates that this tag and the following tag are defined in "jabber.iq:roster" schema. For each friend of this user, it returns an *item* element with this friend's JID, name and the status of subscription. If the friend belongs to some group, a *group* tag with group name is returned.

XMPP starts from <stream/> stanza, then followed with these three core stanzas: <presence/>, <message/> and <iq/>. Including *from, to,*

id, xml:lang attributes, <stream/> also has *version* attribute which means the version of protocols.

2.2.2 Authentication and Security

XMPP uses Simple Authentication and Security Layer (SASL) framework to replace its original authentication and data security mechanism. SASL is a framework for adding authentication support to application protocols [23]. SASL was designed by Myers, John G. in 1997. SASL has many authentication approaches that can be easily used in XMPP, including DIGEST-MD5, EXTERNAL, SKEY and PLAIN. When XMPP uses SASL to connect a connection, it uses DIGEST-MD5 approach to authenticate a session.

A summary of the process of authentication between clients and an XMPP server is below.

- step 1: Clients initiates a stream to the XMPP server with <stream/> tag:

  ```
  <stream:stream to="server Name"
    xmlns="jabber:client"
    xmlns:stream="http://etherx.jabber.org/streams"
    version="1.0">
  ```

- step2: The server responds clients with available authentication mechanisms such as DIGEST-MD5, PLAIN, ANONYMOUS and CRAM-MD5 using <features/> tag:

  ```
  <stream:features >
    <starttls xmlns="urn:ietf:params:xml:ns:xmpp-tls"/>
      <mechanisms xmlns="urn:ietf:params:xml:ns:xmpp-sasl">
        <mechanism>DIGEST-MD5</mechanism>
        <mechanism>PLAIN </mechanism>
        <mechanism>ANONYMOUS </mechanism>
        <mechanism>CRAM-MD5 </mechanism>
      </mechanism>
    <auth xmlns="http://jabber.org/features/iq-auth"/>
  </stream:features>
  ```

- step 3: Then clients select an authentication mechanism and return the choice to the server;

- step 4: The server communicates with clients and validate their authentication by sending an encoded challenge and clients need to pass it. If a client passes, the server will return to the client a response with <success/> tag:

```
<success xmlns="urn:ietf:params:xml:ns:xmpp-sasl"/>
```

  or the authentication is failed, the server will return a response with <failure/> tag:

```
<failure xmlns="urn:ietf:params:xml:ns:xmpp-sasl"/>
```

- step 5: Once a client receives <success/> tag, it initiates a new stream to the server and binds itself to a specific resource with JID by sending a <bind/> tag. For detailed examples of SASL authentication, please refer to [30].

2.3 Compression and Codec

Before transmitting the audio and video data to streaming server, the data should be compressed or encoded because of the limited network bandwidth. During the past decades, there are many kinds of audio/video compression and codec algorithms. For instance, the video coding standards have been developed into two categories. One is the Moving Pictures Expert Group (MPEG) with video coding standards by ISO/IEC (an ISO organization). The other is the Video Compression Expert Group (VCEG) with video coding standards by ITU-T (an organization). In the following, we briefly summarize these media coding standards.

MPEG series standards

- MPEG-1 is designed for CD-ROM digital media and compressed at bit rate up to 1.5Mbps in 1990 [5]. It became video coding standard in 1992 and is the first video compression standard made by MPEG. It consists of five parts (systems, video, audio, conformance testing and reference software) and is widely used, such as MP3 audio format.

- MPEG-2, also known as H.262 [13, 9], was published in 1994 [11, 3, 38]. It targets television broadcast such as cable and satellite television. MPEG-2 is backward compatible and is a core technology of DVD-Video.

- MPEG-4 was introduced in 1998 [29, 37]. It keeps many good features of MPEG-1 , MPEG-2 and other related standards. It is aimed at web streaming media, voice and broadcast television applications, etc. MPEG-4 enables encoding and transmitting mixed media data robustly and efficiently.

H Series standards

- H.261 is another kind of video compression standard which was published by VCEG in 1996 [12]. It mainly targets on ISDN video conferencing with a 64kbps bit-rate. H.261 is a milestone in the video compression development history.

- H.263 is a widely used video compression standard in many applications [14]. It has a big improvement based on previous video compression standards such as H.262. In addition, it added many features such as supporting streaming media and IP-based videoconferencing. It is extended by H.263+ and H.263++ to improve coding performance.

- H.264/AVC, also known as MPEG-4 part 10 [37] Advanced Video Coding (MPEG-4 AVC), it was completed in 2003 by VCEG and MPEG and is one of the most commonly used video compression standards [15]. H.264 is designed for a variety of applications and is more efficient than MPEG-2 and H.263.

- H.265, also known as HEVC (High Efficiency Video Coding) [35], is jointly developed by MPEG and VCEG. The first version was published in 2013 [8]. It targets to provide high quality video with low bit-rate and high coding efficiency.

In our system, we use the commonly used media compression algorithms H.263 and H.264 (MPEG-4 part 10). Users can choose one of the algorithms

to set it up.

2.4 Literature Review

Due to the rapid development of wireless networks and smart mobile technologies, mobile video streaming and social networks have become an essential part in people's live in many kinds of areas (social interaction, education, entertainment, surveillance, etc.). There have been lots of well-known technologies of video streaming on mobile devices over the internet, from client end to streaming server side, from the video compression to streaming protocols. A majority of research and applications on video streaming are focused on video-clip sharing and on-demand streaming Some of them are focused on video-clip or real-time annotation

For example, in video sharing, Cheng and Liu [4] presented a NetTube, a peer-to-peer for short video sharing application. They designed a model to reduce the server workload to improve the playback quality and scalability.

Jia [17] and Ma [21] proposed MoviShare (movie video share), a video sharing platform that can provide video browsing and publishing services for mobile users. MoviShare targets a seamless combination of location-based mobile social networking and mobile multimedia sharing. It can also generate a GIF (Graphics Interchange Format) file of available video clips as video abstraction to solve bandwidth and energy limitation problems. However, MoviShare is not a live streaming system but an on-demand streaming. Users can only share their videos once they have finished recording and uploading their videos to a server.

Silva et al. [34] described a method for annotating objects on a live video streaming on Tablets. They designed two approaches to allow users to add annotations when the objects are moving in the video. They use object tracking methods such as Kinect sensor to create anchors on those annotations to avoid the annotations lost when overlaid.

Yamamoto et al. [43] proposed a way to generate annotations based on social activities associating with video clips, such as user comments and weblog.

They developed a system called Synvie to extract valuable information from those social and community activities as videos annotations.

El-Saban et al. [6] presented a system for real-time video annotation of captured videos on mobile phones to facilitate browsing and searching. A user can use this system to capture a video with his mobile phone. The video is sent, in real-time, to a centralized server which analyzes video keyframes to generate annotations by using MSERs (maximally stable extremal regions) detector with SIFT (scale-invariant feature transform) features. Then these annotations are returned to the user's mobile phone. Their work is focused on generating annotations from real-time videos.

H. Sun et al. [36] presented an overview of scalable video streaming techniques which include scalable video coding, video transcoding, network protocols and streaming methods that have been developed in recent years.

Walker et al. [39] provided an architecture of mobile video streaming under 3G and GPRS networks and some standard techniques for audio/video streaming such as audio/video compression and streaming protocols. In their system, they used client buffering for packet loss and stream switch-down decision for lower bit rate. However, their system has a constant delay of tens of seconds when streaming live events. They tested it with only one mobile client.

Meyer [22] addressed a research work on mobile video streaming with a Client/Server architecture. He extended the SpyDroid [2] project and used the VLC Player [3] as a client streaming player. However, his focus is optimizing encoding standards according to the quality of connection for transmission and minimizing consumption of power. It has no social activities or touch-display interactions.

We developed a platform called Kaleidoscope for live media streaming sharing with social-activities on mobile devices. Users can build a private chatting room to invite their friends to join and share real-time video streaming. Besides sending instant messages to the whole set of peers, all the users can

[2]SpyDroid: `https://code.google.com/p/spydroid-ipcamera/`

[3]VLC: `http://www.videolan.org`

also communicate through touching the screen to annotate interesting things. The Kaleidoscope system supports multi-audio/video encoders. Therefore, the Kaleidoscope supports different screen resolutions and frame rates.

Chapter 3

The Kaleidoscope System

In this chapter, we present a synchronous multi-modal interaction system (Kaleidoscope) that supports streaming video sharing, instant messaging, and UI interaction sharing. Kaleidoscope enables friends to share live videos and interact with textual messages and touch-display annotations. Firstly, we give a system overview of Kaleidoscope. Then we describe the flow of Kaleidoscope (Section 3.1). The Kaleidoscope system has two major components. One component is Kaleidoscope client with Sender, Receiver and functions. The other component is Kaleidoscope server which is responsible for room management, video streaming, instant messaging, touch-display annotations and storage. Then, we describe the communications and behaviors between clients and servers of Kaleidoscope in Section 3.2. Finally, we provide a storyboard of the Kaleidoscope in Section 3.3.

3.1 System Overview

The purpose of Kaleidoscope system is to provide a real-time video sharing platform with social-activity interactions on mobile devices. Figure 3.1 shows a high level description of the interaction of the Kaleidoscope system. The Kaleidoscope system uses Client/Server (C/S) model. On the client site, it consists of two components — the Sender and the Receiver. On the server site, it has three components which are video streaming service, social interaction service and storage service.

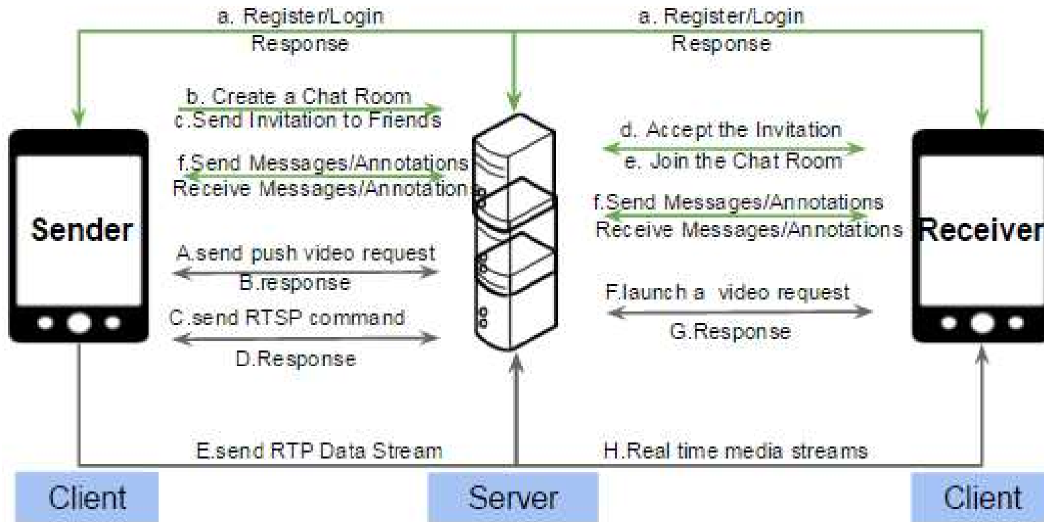

Figure 3.1: Kaleidoscope system interaction diagram. On the *Sender* side a user can create a chat room, invite friends to join and share a video streaming with friends in that chat room. On the *Receiver* side a user can accept the invitation, join the chat room and play the video streaming in real-time. On both Sender and Receiver sides users can communicate by sending text messages or touching the video screen to annotate video content to share with each other.

3.1.1 Architecture

In Figure 3.1, there are three services on the server. The social interaction service process starts from step (a) to step (f) by green lines. The video streaming service process is described from step (A) to step (H) labeled by black lines. The storage service is responsible for saving social interaction information and media streams into database. In this section, we describe all major components in client side and server side with the libraries used in developing Kaleidoscope system. In Section 3.1.2, we give a detail description of social interaction and video streaming service.

Kaleidoscope Client

We implement Sender and Receiver clients based on VLC player library — (libVLC) [1] on Android mobile devices. VLC player is a cross-platform multimedia player which is open source and supports many kinds of audio and

[1]libVLC: `http://www.videolan.org/vlc/libvlc.html`

video codecs. The libVLC is the core engine of VLC. To build libVLC and embed it in our system, we need to download VLC source code and compile it. The libVLC is a C library. Since Kaleidoscope client is an Android project, we compile the libVLC library into a dynamic library according to the Android system requirement and load it into the project. See AndroidCompile [42] and LibVLC Android Sample [40] for details about compiling VLC and embedding it into Android applications. Figure 3.2 shows the libVLC library view.

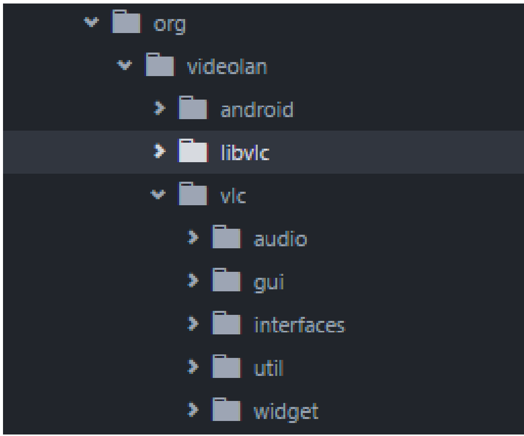

Figure 3.2: The libVLC library view.

In the *org.videolan.libvlc* package, it provides decoding library functions and basic functions of playing audios and videos. The *org.videolan.vlc.interface* defines playing and controlling audio interfaces. The *org.videolan.vlc.audio* and *org.videolan.org.gui.video* implement audio and video decoding playing service. It also provides audio and video player widgets in *org.videolan.vlc.widget* package.

Kaleidoscope Server

Social Interaction Service. In Kaleidoscope, the social interaction service is responsible for user management, chat room management and instant messaging (IM). We use Openfire [2] as the social activity interaction server. Openfire is an open-source cross-platform IM server using XMPP protocol. Openfire provides a plugin interface for further development. With the plugin interface, developers can extend their own functions or plugins according to their system requirements. Openfire has many nice features such as providing a web based administration panel to manage users, chats (peer-to-peer chat and multi-user chat), chat rooms and database connectivity for storing messages and user details.

[2]Openfire: `http://www.igniterealtime.org/projects/openfire/`

We use the aSmack [3] library to implement social interactions client side. The aSmack is an open source XMPP client library written in Java language. It allows us to implement user registration, login, chat rooms, friend inviting and messaging. Figure 3.3 shows the API package view. The latest version has been replaced with the Smack library [4]. All the XMPP API functions we used are in *jivesoftwore.smack* and *jivesoftware.smackx* packages. The *jivesoftwore.smack* package includes important classes such as, SASLAuthentication, XMPPConnection, ConnectionConfiguration, Chat and ChatManager, etc. The SASLAuthentication class is responsible for SASL authentication; XMPPConnection and ConnectionConfiguration are responsible for XMMP connection and connection configuration; Chat and ChatManager are responsible for peer-to-peer messaging and messages management.

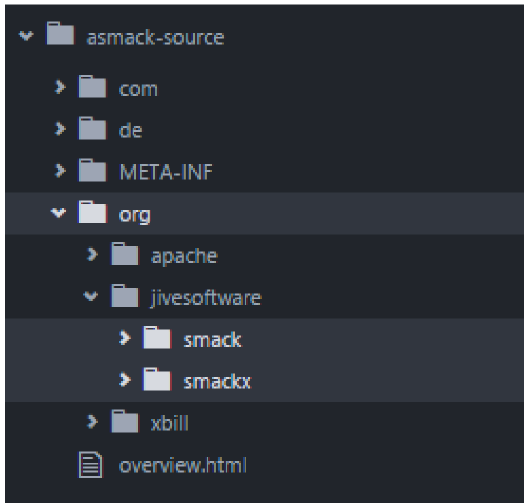

Figure 3.3: XMPP Client Library aSmack liabrary view.

The *jivesoftware.smackx* is a plugin extension package which contains very useful classes such as MultiUserChat, Roster, RosterEntry and GroupChatInvitation, etc. The MultiUserChat class is used to create chat rooms and multi-user chatting; GroupChatInvitation is responsible for inviting friends to a group chat; Roster and RosterEntry classes are used to retrieve all friends of a user and add friends to a group.

Video Streaming Service. The video streaming service is responsible for receiving the multimedia streaming data from Senders and forwarding them to Receivers. We use EasyDarwin [5] (ESS) as the streaming server. EasyDarwin is an open-source streaming server project based on Apple's Dar-

[3] aSmack: `http://asmack.org`
[4] Smack: `https://github.com/igniterealtime/Smack`
[5] EasyDarwin: `https://github.com/EasyDarwin/EasyDarwin`

win Streaming Server [6]. It allows users to send video streaming to clients through the Internet using RTP/RTSP protocols. Besides live streaming, ESS also supports on-demand streaming. Furthermore, ESS uses modular design and provides an interface for further development. We can develop our own modules according to the requirement of Kaleidoscope system such as storage module which is used to save audio and video streaming into files. Figure 3.4 shows the ESS project view.

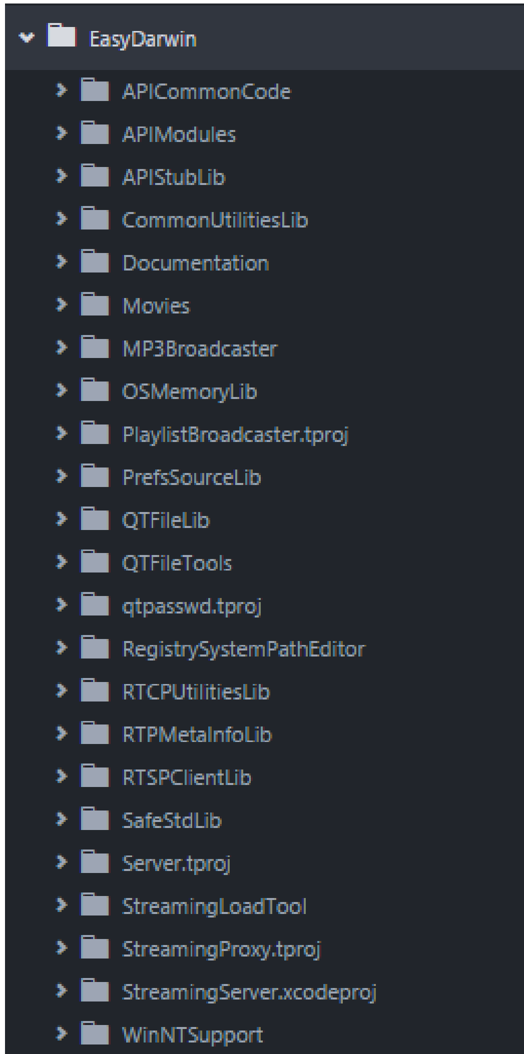

Figure 3.4: ESS project view.

The *Server.tproj* is the main project of ESS which contains the core server code and RTP subsystem and RTSP subsystem which are responsible for management of all modules, RTP/RTSP streams and sessions. There are two important libraries *CommonUtilitiesLib* and *QTFileLib* in ESS project. These two libraries cover almost all the basic functions and tools of ESS such as thread management, data structure, networking and files parsing utilities. The *RTCPUtilitiesLib* has functions for processing RTCP requests and *RTSPClientLib* is responsible for implementing server's RTSP client. The *APIModules* contains streaming server modules of ESS such as *QTSSFileModule*, *QTSSReflectorModule*, *QTSSAccessLogModule* and *QTSSFlowControlModule*. More detailed programming guides are in [2].

[6] http://dss.macosforge.org/

On the video streaming client, we use the libstreaming [7] library to implement the interactions and data transmission between clients and ESS. The libstreaming is a video streaming API which provides a strategy for streaming multiple audio and video encoders by using RTP/RTSP. Figure 3.5 shows the libstreaming library package view. The functions of the packages are as follows.

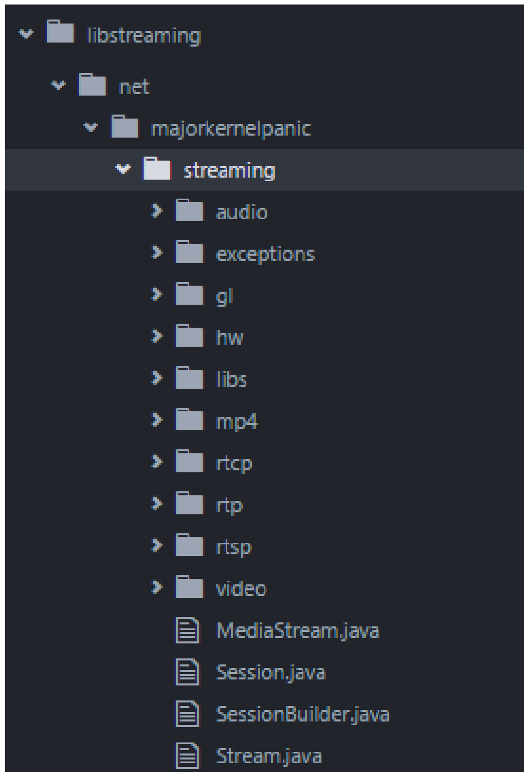

Figure 3.5: The libstreaming library view.

The *net.majorkernelpanic.rtsp* package implements RTSP commands and is responsible for sending RTSP requests and processing interactions with streaming server on client side such as SendRequestAnnounce() and SendRequestSetup() methods. The *net.majorkernelpanic.audio* and *net.majorkernelpanic.video* packages are responsible for encoding audio and video into AAC, AMR, H.263 or H.264 format. The *net.majorkernelpanic.rtp* package defines audio and video "packetizers" which are responsible for packing the audio and video streaming data into RTP packet, and transmitting them to streaming server. The *net.majorkernelpanic.rtcp* package is responsible for processing RTCP requests and sending reports to server. The *net.majorkernelpanic.streaming* package is responsible for controlling RTP/RTSP session connections. The *net.majorkernelpanic.mp4* package implements adding MP4 header information to audio and video RTP packets such as PPS and SPS information on the Sender side. On the Receiver side, the header information is used to decode videos into MP4. The other packages provide video surface control and

[7] libstreaming: `https://github.com/fyhertz/libstreaming`

exception handling.

Storage Service. In Kaleidoscope, the storage service is responsible for storing the content (text messages, touching annotations and audio/video streams) with timestamps on a cloud. We implement the storage service based on ESS, Openfire and LAMP (Linux-Apache-MySQL-PHP) stack [41]. LAMP is an open source software bundle including Linux Operating system, Apache HTTP server, MySQL relational database management system and PHP script language. Text messages and touching annotations are stored in database directly. Audio and video streams are saved into files. All the relevant information are connected with timestamps, chat rooms and stream names to support extended services such as video content analysis and feature extraction.

3.1.2 The Overall Kaleidoscope Process

The whole running process of the Kaleidoscope system is shown in Figure 3.1, where green lines stand for social interaction process and black lines stand for video streaming process. The process starts with Step 1 and terminates at Step 15.

Registration.

Step 1: When a user (Sender) wants to start a video streaming channel, he should register an account first. When the user registers an account in the Kaleidoscope system, it needs a *username* and *email.*

Login.

Step 2: Then the Sender can sign in the Kaleidoscope system with the registered account. Once logged in, an online notification is sent to all his friends.

Initiating a session.

Step 3: The Sender creates a chat room which is used to share content (text messages, audio, video and touch annotations of videos) with friends. The content is only visible to friends who joined the room.

Step 4: The Sender sends invitations to selected friends (Receivers). The invitation information includes the Sender's name and the subject of video.

Sender: push video streaming

Step 5: Then the Sender starts a video streaming channel and sends pushing video streaming request to the streaming server through RTSP commands.

Step 6: With the response of streaming server, if the response is OK, the Sender begins to capture the audio and video streaming data. The Sender client encodes the streaming data and packetizes them into RTP packets, then uploads them to a streaming server.

Media streaming storage.

Step 7: When the streaming server receives the streaming data from the Sender, it saves it into an MP4 file with metadata such as the room name and Sender name on storage server.

Receiver: accept the invitation.

Step 8: On the Receiver side, when Receivers get the invitation, they can accept or reject the invitation.

Step 9: Once a Receiver accepts the invitation, he joins the chat room; and

Receiver: play video streaming.

Step 10: A watching video streaming request is sent to the streaming server through RTSP commands.

Step 11: After receiving the request from the Receivers, the streaming server transponders the streaming data to them.

Chatting and touch-display interactions.

Step 12: Then the Sender can talk about the video content with the Receiver by sending messages and the Receiver can also send messages back to the Sender.

Step 13: Furthermore, the Sender and the Receiver can communicate by touching the screen to annotate the video content to share with each other.

Message and annotation storage.

Step 14: All text messages and touching annotations are stored on MySQL database with timestamps.

Session teardown.

Step 15: The Sender and Receivers can stop playing the video and leave the room. When the Sender stops streaming and leaves the room, the room will be destroyed and an notification is sent to all Receivers that the Sender

destroyed the room. Then all Receivers leave room automatically.

3.2 Communication and Behavior

In this section, we discuss communications and behaviors of social interactions and video streaming. We also explain the information exchange between clients (including Sender and Receiver) and servers (including interaction server and video streaming server).

3.2.1 Clients and Social Interaction Service

In this section, we describe the communication between clients and the social interaction server.

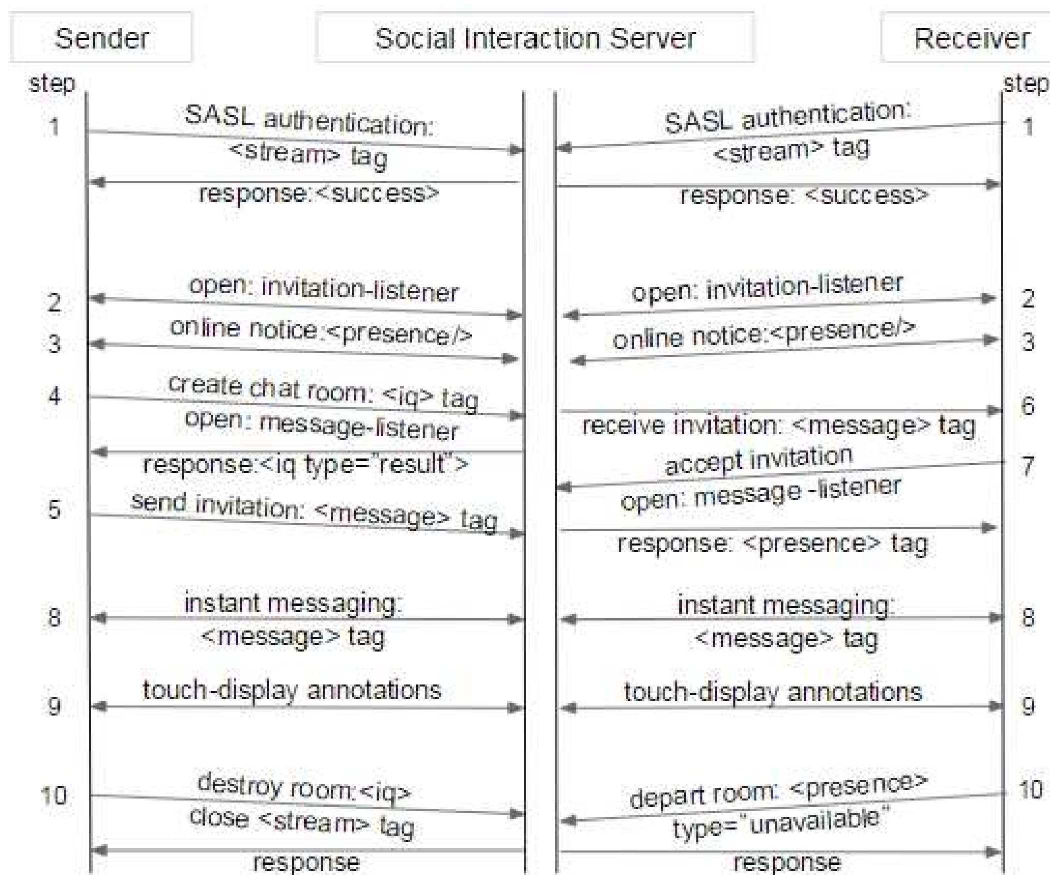

Figure 3.6: Communication between Sender, social interaction server and Receiver. When explaining these messages below, we use Si to refer to Sender side's step i, and Ri to refer to the Receiver side's step i.

In Figure 3.6, the Sender invites a Receiver to watch a video. Detailed description of the communications between Sender and social interaction server and between Receiver and social interaction server are as follows. Corresponding to the step numbers in Figure 3.6, we use Si to refer to Sender side's step i, and Ri to refer to the Receiver side's step i.

S1 and R1: All the clients need to be validated by SASL authentication mechanism to build an XMPP connection when login.

In order to build an XMPP connection with the social interaction server, the client should first create an object of ConnectionConfiguration class with three parameters—server IP address, server port and server name; and then set the security mechanism as SASL authentication mechanism with setSASLAuthenticationEnabled(true) method. An XMPP connection is built between the client and the server by connect() method. A response with <success/> tag is returned to the clients. Then the login() method is called with two parameters: username and password. The client will use DIGEST-MD5 authentication method to validate the credentials with the interaction server. Once authenticated, the user can login the social interaction server.

S2 and R2: After logging in successfully, the client opens an invitation-listener which listens to the invitations from friends through the InvitationListener() method.

S3 and R3: Once logging into the server, the clients send presence broadcast (online notification) to their friends with <presence/> stanza and the server returns their friends' online notification to the clients. A request - response example is shown in Table 3.1:

Table 3.1: An example of presence broadcast and response between clients and server.

<table>
<tr><td>request sent from S3 to R3:

```
<presence id="HklE3-27">
  <c xmlns="http://jabber.org/protocol/caps" hash="sha-1"
  node="http://www.igniterealtime.org/projects/smack/"
  ver="VFSF0KBWvgtPDuY7gJPzp1m5j4E="/>
</presence>
```
</td></tr>
<tr><td>response:

```
<presence id="QPEQ8-3" from="receiver@myria/Smack"
  to="sender@myria">
  <status>online</status>
</presence>
```
</td></tr>
</table>

Then the clients will retrieve all their friends from the server through sending <iq/> element with *type="get"* to the server. After receiving the <iq/> element with *type="get"* request, the server will perform the query and return a list of their friends. An example is shown in Table 3.2.

Table 3.2: An XMPP example of retrieving friend list from server.

<table>
<tr><td>request:

```
<iq id="HklE3-26" from="sender@myria/Smack" type="get">
  <query xmlns="jabber:iq:roster">
  </query>
</iq>
```
</td></tr>
<tr><td>response:

```
<iq type="result" id="HklE3-26" to="sender@myria/Smack"
  type="get">
  <query xmlns="jabber:iq:roster">
    <item jid="receiver@myria" name="receiver"
        subscription="both"/>
    <item jid="user3@myria" name="user3"
        subscription="both"/>
  </query>
</iq>
```
</td></tr>
</table>

To get the friend list, we first call XMPPConnection.getRoster() method which returns a Roster object, and then use Roster.getEntries() method.

S4: The Sender creates a chat room with createMultiUserRoom() method

and opens two listeners, room-message-listener (RoomMsgListenerConnection()), and touch-annotation-listener (PAINTViewRoomMsgListener()). The former listener is responsible for listening to the messages from the chat room and the latter is responsible for listening to the screen touching annotation messages. The Sender configures the chat room settings such as the room name, the number of members and room persistence by sending a <iq/> element with type="set" to the server. If the chat room is created successfully, the server returns the result with chat room information to the Sender. The example is shown in Table 3.3:

Table 3.3: The chat room configuration request and response between clients and server. The value of 0 means false and 1 means true.

request:

```
<iq id="HklE3-32" to="roomname@conference.myria" type="set">
  <query xmlns="http://jabber.org/protocol/muc#owner">
    <x xmlns="jabber:x:data" type="submit" >
      <field var="muc#roomconfig_roomname" type="text-single"
        label="Room Name">
        <value>roomname</value>
      </field>
      <field var="muc#roomconfig_maxusers" type="list-single"
        label="Maximum Room Occupants">
        <value>30</value>
      </field>
      <field var="muc#roomconfig_persistentroom" type="boolean"
        label="Room is Persistent">
        <value>0</value>
      </field>
      <field var="muc#roomconfig_moderatedroom" type="boolean"
        label="Room is Moderated">
        <value>1</value>
      </field>
      <field> ... </field>
    </x>
  </query>
</iq>
```

response:

```
<iq type="result" id="HklE3-32" from="roomname@conference.myria"
    to="sender@myria/Smack"/>
```

S5: The Sender sends joining room invitations to selected friends with media streaming link by inviteToChatRoom() method, and a <message/> stanza is sent to the server. The example is shown in Table 3.4:

Table 3.4: The Sender sends invitation to a Receiver.

```
<message id="HklE3-35"
  to="receiver@myria/Smack" type="groupchat"
  from="roomname@conference.myria/sender@myria/Smack" >
  <body>rtsp://162.246.156.33:554/live.sdp ></body>
</message>
```

The invitation information includes message type, chat room name, Sender's JID and the resource URL of media streaming.

Then the Sender starts pushing media streaming to streaming server at the same time. The communication process of Sender and steaming server is introduced in Section 3.2.2.

R6: With the invitation-listener, once invited, the Receiver will receive the invitation with <message/> tag (Table 3.5):

Table 3.5: The Receiver gets the invitation from the Sender.

```
<message from="roomname@conference.myria" to="receiver@myria" >
  <x xmlns="http://jabber.org/protocol/muc#user" >
    <invite from="sender@myria" >
      <body>rtsp://162.246.156.33:554/live.sdp</body>
    </invite>
  </x>
</message>
```

R7: The Receiver can accept or reject. Upon accepting, the Receiver opens the room-message-listener and touch-annotation-listener, and joins the chat room. The server sends a message to the Receiver to notify Receiver's role in this chat room. A message about the role being a *participant* example is as follows (Table 3.6).

Table 3.6: Server response containing the Receiver's role in the chat room.

```
<presence id="eE58-6" to="receiver@myria/Smack"
  from="roomname@conference.myria/sender@myria/Smack">
  <x xmlns="http://jabber.org/protocol/muc#user">
    <item jid="receiver@myria/Smack" affiliation="none"
    role="participant"/>
  </x>
</presence>
```

Then the Receiver sends a streaming request to the streaming server with the URL received from the Sender. The behavior of requesting media streaming between Receiver and streaming server is described in Section 3.2.2.

S8 and R8: The Sender and Receiver can communicate freely in this room through sending instant messaging with SendMessage(). An example of chatting in the chat room is shown in Table 3.7.

Table 3.7: The clients communicate in a chat room.

Sender sends message in a chat room:

```
<message id="deE58-14" to="roomname@conference.myria"
  type="groupchat">
  <body>Hello everyone!</body>
</message>
```

Receiver gets message in the chat room:

```
<message id="deE58-14"
  to="receiver@myria/Smack" type="groupchat"
  from="roomname@conference.myria/sender@myria/Smack">
  <body>Hello everyone!</body>
</message>
```

The Sender sends message "Hello everyone!" in the chat room and all members will receive this message.

S9 and R9: The Sender and Receiver can also communicate by touching the screen and annotating the video content. This is novel feature of social interactions. A detailed description is described in Section 3.2.3.

R10: The Receivers can depart the chat room through departChatRoom() method by sending "unavailable" <presence/> to friends. Table 3.8 shows the example:

Table 3.8: The Receiver leaves chat room.

```
<presence id="HklE3-22" type="unavailable">
  <c xmlns='http://jabber.org/protocol/caps" hash='sha-1"
    node="http://www.igniterealtime.org/projects/smack/"
    ver="VFSF0KBWvgtPDuY7gJPzp1m5j4E="/>
</presence>
```

S10: The Sender can destroy the chat room to stop playing the media streaming and instant messaging with stopStream() and destroyChatRoom() methods. A close <stream/> tag will be sent to the server: </stream:stream>. If the Sender destroys the room, a notification is sent to all Receivers and the Receivers leave the chat room, e.g.,

Table 3.9: The Sender destroys the room.

```
<iq id="HklE3-45" to="roomname@conference.myria" type="set">
  <query xmlns="http://jabber.org/protocol/muc#owner">
    <destroy jid="roomname@conference.myria">
      <reason>destroy reason</reason>
    </destroy>
  </query>
</iq>
```

All the contents such as Sender name, Receiver name, room name, instant messages and touch-display annotations are saved into MySQL database with timestamps.

3.2.2 Clients and Streaming Service

After creating a chat room and sending invitations to selected friends, the Sender starts capturing the audio and video data. The Sender client will encode the streaming data and packetize them into RTP packets, then push these packets data to the streaming server by sending RTSP push commands.

On the other side, after receiving and accepting the invitation, Receivers send a video streaming request to the streaming server by sending RTSP request commands. The communication mechanism between Senders, the streaming server and Receivers is shown in Figure 3.7. Corresponding to the

sequence numbers in Figure 3.6, we use Si to refer to Sender side's step i, and Ri to refer to the Receiver side's step i.

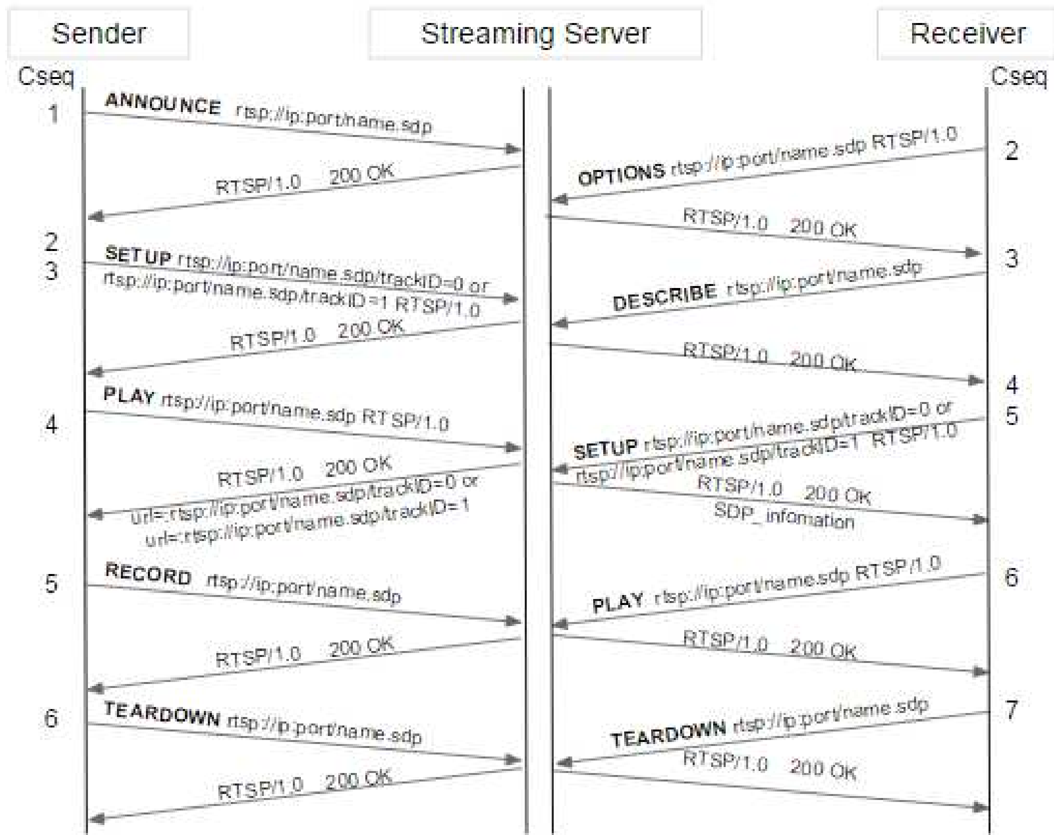

Figure 3.7: Streaming communication between Sender, streaming server and Receiver. When explaining these messages below, we use Si to refer to Sender side's step i, and Ri to refer to the Receiver side's step i.

When a Sender starts a video streaming channel, the Sender needs to initialize some configuration of the streaming process. Firstly, the Sender calls PLAYVideoStreaming() function to start a video streaming. After configuring audio and video settings such as audio/video encoders and video resolution, PLAYVideoStreaming() will call startStream() to begin to send RTSP commands to the streaming server to start video streaming. The whole process of communication between Sender and streaming server can be described in a series of commands: ANNOUNCE — SETUP — PLAY — TEARDOWN.

When a Receiver get the invitation from a Sender, the Receiver parses the invitation information to get the media URL resource and call load()

function first. This method uses audio as a default player. In this method, it sends RTSP requesting commands to the streaming server according the stream URL. A streaming requesting process of communication from Receiver to streaming server through (OPTIONS) — DESCRIBE — SETUP — PLAY — TEARDOWN commands. Detail interaction of description is explained below.

S1: *ANNOUNCE.* When starting a video streaming, the Sender sends RTSP ANNOUNCE command. Its format is ***rtsp://server-ip:port/streaming-name.sdp*** *RTSP/version* where the *server-ip* is the IP address of the streaming server, *port* is the server port, *streaming-name.sdp* is the unique identifier of the video streaming, and *RTSP/version* is the version of RTSP. This function will return a sequence number and use it to send the next command.

An example of ANNOUNCE command. *ANNOUNCE* ***rtsp://162.246.156.33:554/live.sdp*** *RTSP/1.0* is sent to the streaming server. In this example, the Sender sends SDP information of local media to the streaming server, where the sequence number *CSeq* starts from 1.

In the streaming server, ANNOUNCE command is processed by DoAnnounce() function of QTSSReflectorModule module. It sets the media streaming in a broadcast mode and parses the header of the request of SDP and return a status code of "200 OK" *RTSP/1.0 200 OK* to the Sender.

S2 and S3: *SETUP.* Next the Sender sends SETUP command through SendRequestSetup() method to the streaming server. Taking video-only streaming for example, a SETUP command is *SETUP* ***rtsp://162.246.156.33:554/live.sdp/trackID=0*** *RTSP/1.0*. In the streaming server, the SETUP command is processed by DoSetup(). The streaming server creates a session using ReflectionSession class with a session ID, and then parses the media streaming tracks, such as audio track and video track. For each media stream, the Sender should send SETUP command independently and the sequence number increases automatically. For example, if the media mixes audio and video together, the Sender will send another SETUP command to the streaming server: *SETUP* ***rtsp://162.246.156.33:554/live.sdp/trackID=***

1 RTSP/1.0 and sequence number *CSeq* is 3.

S4: *PLAY.* After the streaming server parses SDP information from the Sender, AddRTPStream class will create RTP streams according to the media streaming tracks and returns "200 OK" to the Sender. Then the Sender sends PLAY command with a resource link to the streaming server (SendRequest-Play() method) (*PLAY* ***rtsp://162.246.156.33:554/live.sdp*** *RTSP/1.0*) with *CSeq:4*. The streaming server calls DoPlay() function to redirect ReflectionSession to RTPSession. Accordingly, ProcessRTPData() method is called to process the RTP data according to the track's ID

The streaming server returns a response which has a request address of media streaming data. The request address is a URL, for example, ***rtsp://server_ip:port/streamName.sdp***. The *server_ip* is the IP address of the streaming server, the *streamName.sdp* is the file location of the media streaming. The stream name must end with *.sdp*, because it is a file identifier to indicate the stream with SDP value.

S5: *RECORD.* Then the Sender sends RECORD command to the streaming server. The RECORD command is *RECORD* ***rtsp://162.246.156.33:554/live.sdp*** *RTSP/1.0*. After receiving the RECORD command, the streaming server will call openRTSP program to output the media streams into a MP4 file.

S6: *TEARDOWN.* The Sender can stop the media stream by sending a TEARDOWN command (*TEARDOWN* ***rtsp://162.246.156.33:554/live.sdp*** *RTSP/1.0*) with stopStream() function to the streaming server. After receiving the TEARDOWN command, the streaming server stops receiving the streaming data and forwarding the streaming data to connected Receivers. The Sender destroys the chat room and the social interaction server closes all the connections with the Sender and the Receivers.

When a Receiver get the invitation from a Sender, the Receiver parses the invitation information to get the media URL resource and call load() function first. This method uses audio as a default player. In this method, it sends RTSP requesting commands to the streaming server according the stream URL. A streaming requesting process of communication from Receiver

to streaming server through (OPTIONS) — DESCRIBE — SETUP — PLAY — TEARDOWN commands. Detailed process is as follows.

R2: ***OPTIONS.*** The OPTIONS command is optional. Either the OPTIONS and DESCRIBE commands together in sequence, or the DESCRIBE command only can be sent to the streaming server. The OPTIONS command is responsible for showing the protocol version and available commands from the server to Receivers.

After receiving OPTIONS command *OPTIONS* ***rtsp://162.246.156.33:554/live.sdp*** *RTSP/1.0*, the streaming server returns a "200 OK" status code and a series of possible commands to the Receiver.

R3: ***DESCRIBE.*** The DESCRIBE command retrieves the description of the request URL from the server. The server returns a description of the request resource to Receivers. A DESCRIBE command is *DESCRIBE* ***rtsp://162.246.156.33:554/live.sdp*** *RTSP/1.0.* In this example, a request address is used which is returned from streaming server before. A streaming server will call DoDescribe() function which parses the URL address to get the address of the streaming server and looks for the streaming media resource in the server.

Then it returns a status code of "200 OK" to the Receiver with the streaming media SDP information. The SDP includes the stream tracks information such as audio track ID, audio frequency and video encoder.

R4 and R5: ***SETUP.*** After receiving SDP information of the media streaming, the Receiver sends SETUP and PLAY commands to the streaming server. Note that the Receiver sends a SETUP command to the streaming server independently for each stream track.

An example of the SETUP command is *SETUP* ***rtsp://162.246.156.33:554/live.sdp/trackID=0*** *RTSP/1.0*, which means the Receiver requests the content from the server with track ID. After receiving the request, the server calls DoSetup() function in a similar way to Sender communicating with the streaming server.

R6: ***PLAY.*** Then the Receiver sends a PLAY command to the streaming server, e.g., *PLAY* ***rtsp://162.246.156.33:554/live.sdp*** *RTSP/1.0.*

The server calls DoPlay() to transponder the stream to the Receiver.

R7: ***TEARDOWN.*** The Receiver can stop the media stream through sending a TEARDOWN command: *TEARDOWN* ***rtsp://162.246.156.33:554/live.sdp*** *RTSP/1.0* to the streaming server. After receiving the command, the streaming server stops forwarding the streaming data to the Receiver. Then the social interaction server closes the connections with the Receiver. Then the Receiver leaves the chat room.

3.2.3 Touch-display Feature

In this section, we discuss the novel feature — *touch-display feature* which is when a user touches the video screen to tag something in the video, the same annotation should be displayed in the same position on the screen of the other side. The problem is challenging. As we know, different mobile devices have different screen sizes. Normally the video screen size is different on Sender and Receiver sides when playing videos. Additionally, there may be 90° anti-clock rotation of video between Sender and Receiver sides depending on the angle that the Sender shoots the video.

For example, Figure 3.8 shows the problem of touch-display. Assume on Sender side (X_1, Y_1) is the coordinate of Sender tagged, what is the position displayed on screen of Receiver side, (X_R, Y_R)? On the other hand, assume a Receiver tags (X_3, Y_3) position, what is the position should be displayed on screen of Sender side, (X_S, Y_S)?

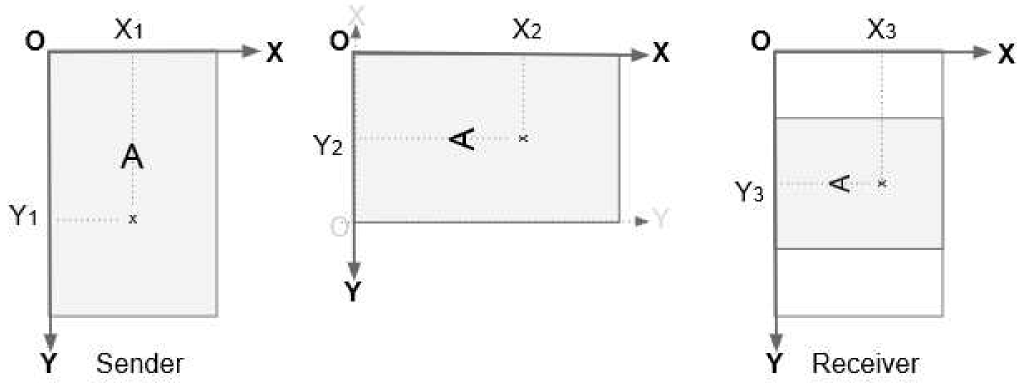

Figure 3.8: The touch-display feature on a Sender and Receiver side.

Let X_{width}, Y_{length} denote the mobile device screen width and length respectively. For drawing a circle, the radius is R. The position the Sender touched is (X_1, Y_1) and the radius is R_S, (X_3, Y_3) is the position the Receiver touched, radius is R_R .

$$\begin{aligned} X_2 &= Y_1 \\ Y_2 &= X_{width} - X_1 \\ X_3 &= X_2 * X_{width}/Y_{length} \\ &= Y_1 * X_{width}/Y_{length} \\ Y_3 &= Y_2 * X_{width}/Y_{length} + (Y_{length}^2 - X_{width}^2)/(2 * Y_{length}) \\ &= (X_{width}^2 + Y_{length}^2 - 2 * X_{width} * X_1)/(2 * Y_{length}) \end{aligned}$$

For radius, if the $R_S = r$ in the Sender side, the radius in Receiver side R_R should be: $R_R = r * X_{width}/Y_{length}$.

So, if the Sender touches (X_1, Y_1) and the radius is $R_S = r$, the position on Receiver side (X_R, Y_R) is

$$\left(\frac{Y_1 * X_{width}}{Y_{length}}, \frac{X_{width}^2 + Y_{length}^2 - 2 * X_{width} * X_1}{2 * Y_{length}}\right)$$

and the radius is $R_R = r * X_{width}/Y_{length}$.

Similarly, if the Receiver touches (X_3, Y_3) and the radius is $R_R = r$, the position on Sender side (X_S, Y_S) is

$$\left(\frac{X_{width}^2 + Y_{length}^2 - 2 * Y_3 * Y_{length}}{2 * X_{width}}, \frac{X_3 * Y_{length}}{X_{width}}\right)$$

and the radius is $R_S = \sqrt{r * Y_{length}}$.

A detailed process of touch-display feature is below. In the step "S9 and R9" of Figure 3.6, as we discussed before, when explaining these messages below, we use *S9-i* to refer to Sender side's step i, and *R9-i* to refer to the Receiver side's step i. We start from on a Sender side.

S9-1: The Sender touches the screen to annotate some content of the video. With paintViewTouchListener() touch-listener, the touch position coordinate (X, Y) is obtained.

S9-2: A circle is drawn on the screen at the touch position with drawCircle() method. The circle is fifty pixels with the center being at the point where touched.

S9-3: Then an annotation box popes up to let the Sender add some tags with touchAnnotation() method.

S9-4: Through SendMessage() method, a message with the coordinate of the circle and annotations is sent to the chat room.

R9-1: Receivers can get this message with PAINTViewRoomMsgListener() room-annotation-listener.

R9-2: Then the Receiver parses the message to get the coordinate and radius of the Sender's touch-point. According the above rotation rules we discussed above, the Receiver will get a new coordinate and radius.

R9-3: A circle is drawn on the screen on the Receiver side with drawCircle().

R9-4: The Receiver can also touch the screen to annotate some content of video and send them to the Sender in a similar way.

3.3 Storyboard

In this section, we provide a few main screenshots (Figure 3.9 — Figure 3.14) of our Kaleidoscope system with descriptions.

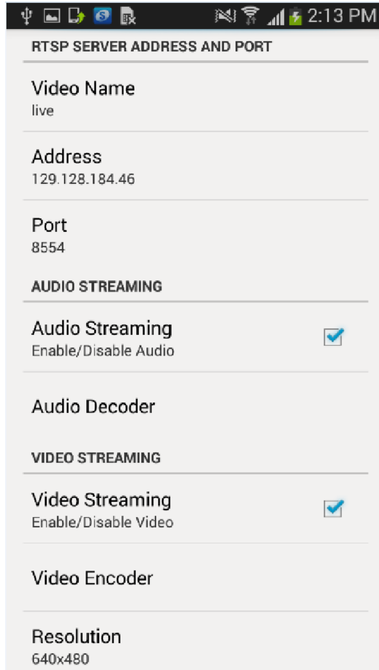

Figure 3.9: Configuration interface.

In Figure 3.9, after login, the users can do some configuration such as setting audio/video-only, setting the audio/video encoder, setting the resolution and frame rate of the video. Then the user can back to the main interface with the default "back" button of Android devices.

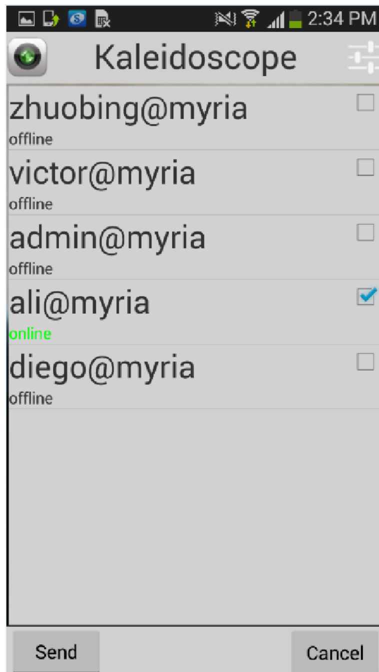

Figure 3.10: Friend list interface.

After configuration, the Sender can invite friends (Receivers) to join a chat room and share a video. Figure 3.10 shows the friend list. If a friend is online, there is green text to indicate the user is online. Only online users can receive the invitation.

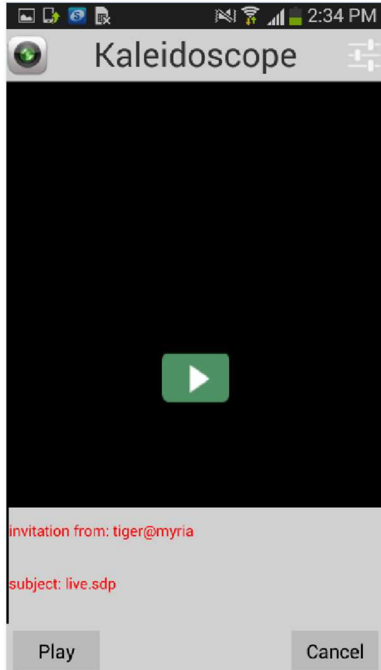

Figure 3.11: The Receiver gets an invitation from Sender.

Figure 3.11 shows the interface of the Receiver receiving the invitation from the Sender. If the Receiver accepts the invitation, he joins the chat room and sends video request to the server. Then the Sender and Receiver can communicate though sending text messages or touch-display feature.

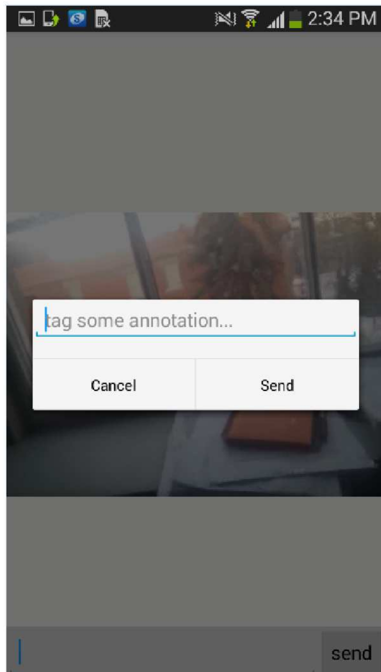

Figure 3.12: The Receiver touches the screen to add some tags.

Figure 3.12 shows the example that a Receiver touches the screen to annotate an object and sends it to friends.

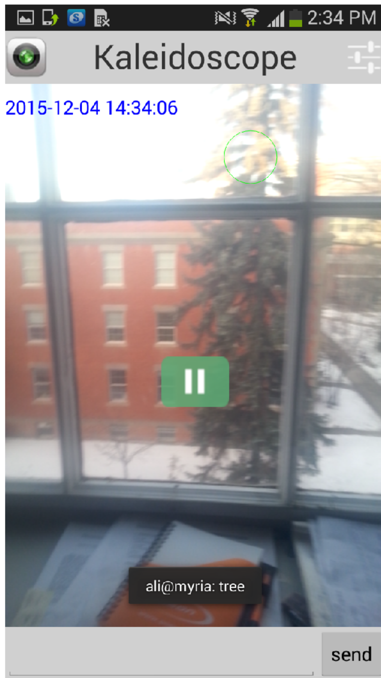

Figure 3.13: The Sender receives touch-display annotation.

Figure 3.13 shows that a Sender receives the annotation message and displays the circle on the screen.

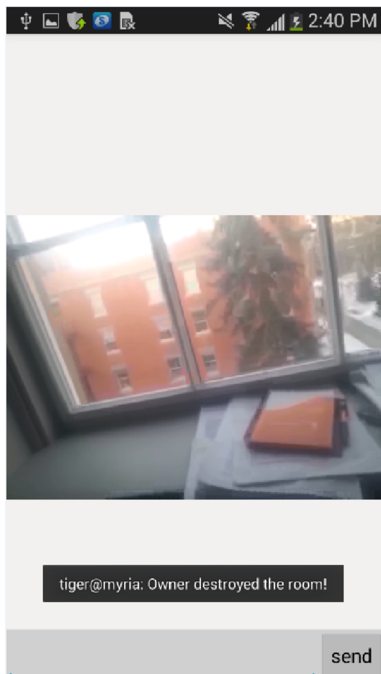

Figure 3.14: The Receiver gets a destroy-room notification from Sender.

The Sender and Receiver can stop the stream independently. When the Sender stops the stream, he will send a destroy-room notification to Receivers and destroy the room. If Receiver receives the notification, he will leave the room. Figure 3.14 shows the Receiver receives the destroy-room notification from the Sender.

Chapter 4

Evaluation

In this chapter, we present an evaluation of the Kaleidoscope platform. We aim to test the system in various settings. Section 4.1 describes the experiments. In the experiments, we test the system with several situations with different locations of steaming server and different number of Receivers. For each situation, we analyze the the usage of CPU and memory, number of success requests and the video frames lost. We give a summary of experiment results in Section 4.2.

4.1 Experiment Setup

In the experiment, we evaluate on our platform by gathering statistics of video streaming under different traffic and locations. A typical session in Kaleidoscope starts by a user creating a streaming channel and inviting friends to participate in a chat room.

The resources used in the experiment are as follows.

- Sender: An Android mobile device in Edmonton.
- Streaming server: A desktop computer in Edmonton, two large cloud instances (one is on Cybera cloud in Calgary, and the other one is on SAVI cloud in Toronto).
- Receiver: We also have another android mobile device in Edmonton to serve as a Receiver in the real world.

- Simulated Receivers: A desktop in Edmonton and two medium instances which are on Cybera cloud in Calgary. For the Simulated Receivers, to some extent, it is a streaming player which is used to play the live streaming. So we create a number of Receiver clients on these devices with VLC player to send video streaming requests to a streaming server. Because of the limited computational resources, we can only run up to 30 to 50 connections on each simulated receiver machine.

- Network environment: On the client side, we use University of Alberta University Wireless Service (UWS) network which speed is 72 Mbps.On the server side, the network speed is 1000 Mbps of the Server in Edmonton. On the Cybera cloud (Calgary), the wireless network is CANARIE which speed is 100 Gigabit Ethernet and above 100 GbE on SAVI cloud.

The Sender and the Receiver have Android OS v4.1.1 and pre-installed with our Kaleidoscope App. We also used a server in Edmonton for running ancillary services, such as authentication, persistence and chatting. These services are not tested as their use of resources is moderate, compared to the streaming service. The detailed experiment environment and hardware configuration of these devices is shown in Table 4.1.

Table 4.1: Hardware configuration of the devices used in our experiment.

Device Type	Location	Memory	CPU	Disk
Desktop Computer	Edmonton	4.7GB RAM	2 CPU	256GB
Large Instance(Cybera)	Calgary	8GB RAM	4 VCPU	80.0GB
Large Instance(SAVI)	Toronto	8GB RAM	4 VCPU	80.0GB
Medium Instance(Cybera)	Calgary	2GB RAM	2 VCPU	20.0GB
Android Mobile	Edmonton	2GB RAM	ARM	16GB

In the servers, we collect three metrics from ESS logs: the number of RTP Connections, CPU and memory consumption of the servers.

In the clients, we use VLC to establish the streaming connections and play the video streaming. We extract two metrics: buffering time and frames lost.

In each experiment, the Sender sends a video sharing request to the Receiver and pushes streaming data to the streaming server, the Receivers request

streaming data from the streaming server. The time duration of the live video is about 60 seconds for each experiment. Then we increase the number of Receivers to send streaming requests to the server with a desktop and two medium instances.

4.2 Results

We performed five test cases in the experiment. In particular, each experiment is labeled by "Sender location—Server location—Receiver location". In the first four tests, the video resolution is 320×240 and frame rate is 24. In the last test, the video resolution 640×480 and frame rate is 24. The last test is meant to test the platform in a higher video quality.

Edmonton-Edmonton-Edmonton. In this test case, the Sender, the streaming server and the Receivers are all in Edmonton. Table 4.2 shows the result of the test. In this test, we simulate the number of Receivers ranging from five to thirty in the desktop. The number of frame loss per second is lower than 15%. (Recall that the number of frames per second is 24.) Thus the chances of a key frame being lost is low. Keep in mind that frame loss is largely dependent on the quality of networks.

Edmonton-Calgary-Edmonton. In this case, the Sender and the Receivers are in Edmonton. The streaming server is in Calgary. Table 4.3 show the result of this test. We simulate five to fifty Receivers from the desktop.

Edmonton-Toronto-Edmonton. In this test, the Sender and Receivers are in Edmonton. We move the streaming server to a farther place. Table 4.4 shows the result of this test. We simulate the number of Receivers ranging from two to thirty in the desktop.

Edmonton-Calgary-Calgary. Table 4.5 shows the result of the test. In this test, we simulate the number of Receivers ranging from two to fifty in the two medium instances.

Edmonton-Edmonton-Edmonton with Higher Video Resolution. In this test, the locations are the same with the first experiment, but the video quality is higher with video resolution 640×480. Table 4.6 show the result of

this test. We simulate five to thirty Receivers in the desktop.

Table 4.2: Results of Edmonton-Edmonton-Edmonton experiment.

#connections	success connection rate	average # of lost frames per second
5	100%	2.9
10	100%	3.0
20	100%	3.2
30	95%	2.7

Table 4.3: Results of Edmonton-Calgary-Edmonton experiment

#connections	success connection rate	average # of lost frames per second
5	100%	1.9
10	100%	3.6
20	97%	2.1
30	100%	2.7
40	100%	2.4
50	100%	4.0

Table 4.4: Results of Edmonton-Toronto-Edmonton experiment.

#connections	success connection rate	average # of lost frames per second
5	100%	3.5
10	100%	3.6
20	100%	3.1
30	100%	1.4
40	100%	4.2
50	100%	4.0

Table 4.5: Results of Edmonton-Calgary-Calgary experiment.

#connections	success connection rate	average # of lost frames per second
10	100%	4.8
20	100%	6.1
30	100%	5.6
40	100%	1.9
50	100%	1.0

Table 4.6: Results of Edmonton-Edmonton-Edmonton with higher video quality experiment.

#connections	success connection rate	average # of lost frames per second
10	97%	1.6
20	100%	1.7
30	96%	2.1

Table 4.7: Results of number of Receivers in multi-locations.

#connections	success connection rate	average # of lost frames per second
5	100%	2.1
25	100%	2.2
50	100%	3.9
100	100%	11.8
150	100%	14.9

Finally, we did a complex experiment with number of Receivers in multiple locations. The Sender is in Edmonton, the streaming server locates on Cybera cloud in Calgary and number of Receivers are simulated in SAVI and Cybera clouds. On Cybera and SAVI clouds, we setup two medium instances on each cloud. With one desktop in Edmonton, there are 5 Receiver simulations. Each of them simulates 0-30 connections. The total number of connections is 150. The video streaming lasted 3 minutes based on Receiver side. The result is shown in Table 4.7.

In the experiments, for each test case (with different number of Receivers), we repeated three times. In the experiment results, the first column means the number of total request connections of Receivers. The second column is the success connection rate, which is the number of success connections divide the total number of connections of three times. The success connection means that Receivers should get the streaming data successfully in a timeout threshold (the default value is 10 seconds). The last column is the average number of frame lost per second on Receiver side, which is calculated by

$$\text{frame lost/second} = \frac{\text{\# lost frames of all success connections}}{\text{\# success connections} \times \text{time duration} \times \text{repeated times}}.$$

One thing should be noted that the bandwidth of network has a severe effect on video frame loss. In the first five experiments, the time duration of streaming is about 60 seconds based on Sender side. The results of frame loss per second seem to decrease as more connections added. The reason is that When we increased the number of Receivers with desktop, the computer took a while to open many VLC streaming players (For 30 Receivers, it takes about 10 seconds to start all of them). When we calculated the duration time (60

seconds), we were based on the recording time on the Sender side. on the Receiver side, the playing time maybe just 40$\tilde{5}$0 seconds. The last experiment is more complex with 150 connections in multiple locations at the same time. And the streaming lasts 3 minutes based On Receiver side.

In all the experiments, the number of Receivers is between 5-50. The *CPU consumption* of the streaming server is lower than 1% and the *memory consumption* is lower than 8%. We also see the following results.

- When the server and clients are at the same location, the success connection rate is very high. The success connection rate is above 95% with up 30 connections at the same.
- When the server and clients are at the same location or different locations, the success connection rate is very high. For example, when the server is in a far location – Toronto and all the clients are in Edmonton, the success connection rate is 100% with up to 50 connections at the same.
- When the receivers are at same location but the server is at the varied locations, the success connection rate is very high. For example, when the receivers are in Edmonton and the server is in Calgary and the server is in Toronto, the rate is above 95% with up to 50 connections.
- Even with a higher video quality, the success connection rate still remains above 96% with up to 30 connections. The average number of lost frames is lower than 3 frames.
- The frame loss per second is acceptable with our experimental network environment, which means users can watch the video normally if the key frame is not lost (only one key frame in 24 frames per second).
- Including the bandwidth factor, the frame loss can also be affected by the number of connections. The results of frame loss per second will increase as more connections added

- Under a high wireless network environment, the success connection rate is very high even with a far distance between the clients and server.

In conclusion, we did some tests on cloud with different situations such as different number of Receivers in the same and different location(s), low/high video resolutions and number of Receivers in multiple locations. We evaluated the CPU and memory consumption on server side and the frame lost on Receiver side. The results show that the Kaleidoscope is a stable on both client and server sides.

Chapter 5

Conclusions

In this thesis, we present a real-time video streaming sharing platform on mobile devices with instant messaging and touch-display interaction. The contributions are as follows.

- We developed an Android App (the Kaleidoscope client). In order for the client to be used on multiple types of Android devices, we implemented multiple video-encoding (H.263 and H.264) and audio-encoding algorithms (AAC and AMR). H.263 and AMR is used for Android devices which version is Android 2.3. Android 4.1 supports these two video-encoding and audio-encoding algorithms.

- With the Kaleidoscope app, users can create “chat rooms” to share videos in real time. Moreover, users in the same chat room can interact with each other with textual messages. A novel feature of the Kaleidoscope is that, during a video sharing process, users can touch the screen and annotate the content in the video to talk about the subjects through text messaging.

- We have implemented two components to support these services. The streaming server receives audio/video streaming data from senders, and transmits these data to the receivers. The social-interaction server takes care of user management, video-chat room management, and instant messaging.

- The Kaleidoscope platform also includes a storage system to allow saving various data such as audio/video streaming, text messages and touching annotations to cloud servers. Later the data can be used to support extended future services, such as content analysis and feature extraction.

- We evaluated the Kaleidoscope platform on the SAVI and Cybera clouds. We tested several real-world usage cases of our platform. Results show that the success connection rate is very high whether the clients and servers are in Edmonton, Calgary or Toronto.

The Kaleidoscope platform can be improved in the following aspects.

Firstly, we can make the Kaleidoscope system more robust through using adaptive wireless bandwidth and optimizing video coding technologies [19] to match the bandwidth of network, network condition and mobile devices automatically. The quality of video can become better because of lower data-packet loss.

Secondly, we can add more administrative features to support the Kaleidoscope system, such as reallocating resources according to CPU and memory usage by monitoring the load balance of all servers on cloud, and monitoring and managing all user streaming requests, such as access control and data control.

Thirdly, we can make a user in a multiple rooms at the same time through split the screen in multiple parts. For example, we have a user who is a Sender and he is recording the video and sharing the live video with his friends. Now he receives an invitation from his friends. The Sender can accept that invitation and join his friend's chat room and enjoy the video streaming. The video screen is split into two parts, e.g. up part and down part. The up-part is used to recording the video and the down-part is used to play the video streaming.

Finally, we can add video content analysis and feature extraction in real-time. We can also bind the video frame with touch-display interaction together.

Bibliography

[1] Apostolopoulos, J. G., Tan, W.-t., and Wee, S. J. (2002). Video streaming: Concepts, algorithms, and systems. *HP Laboratories, report HPL-2002-260.*

[2] Apple Computer, I. (2002). Quicktime streaming server modules programming guide @2002,2005. "http://www.apple.com/quicktime/pdf/QTSS_Modules.pdf".

[3] Bosi, M., Brandenburg, K., Quackenbush, S., Fielder, L., Akagiri, K., Fuchs, H., and Dietz, M. (1997). Iso/iec mpeg-2 advanced audio coding. *Journal of the Audio engineering society*, 45(10):789–814.

[4] Cheng, X. and Liu, J. (2009). Nettube: Exploring social networks for peer-to-peer short video sharing. In *INFOCOM 2009, IEEE*, pages 1152–1160. IEEE.

[5] Chiariglione, L. (1996). Mpeg-1-coding of moving pictures and associated audio for digitial storage media at up to about 1.5 mbit/s. *International Organisation for Standardisation, Technical Report, ISO/IEC JTC1/SC29/WG11.*

[6] El-Saban, M., Wang, X.-J., Hasan, N., Bassiouny, M., and Refaat, M. (2011). Seamless annotation and enrichment of mobile captured video streams in real-time. In *Multimedia and Expo (ICME), 2011 IEEE International Conference on*, pages 1–4. IEEE.

[7] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext transfer protocol–http/1.1. Technical report.

[8] Group, I.-T. V. C. E. H.265 : High efficiency video coding - recommendation h.265. "http://www.itu.int/rec/T-REC-H.265", Visited on 2014/11/27.

[9] Group, T. M. P. E. Mpeg-2 video. "http://mpeg.chiariglione.org/standards/mpeg-2/video", Visited on 2014/11/27.

[10] Handley, M., Perkins, C., and Jacobson, V. (2006). Sdp: session description protocol.

[11] Haskell, B. G., Puri, A., and Netravali, A. N. (1997). *Digital Video: An Introduction to MPEG-2: An Introduction to MPEG-2*. Springer Science & Business Media.

[12] ITU-T. H.261 : Video codec for audiovisual services at p x 64 kbit/s - recommendation h.261 (03/93). "http://www.itu.int/rec/T-REC-H.261-199303-I/en", Visited on 2014/11/27.

[13] ITU-T. H.262 : Information technology - generic coding of moving pictures and associated audio information: Video - recommendation h.262. "http://www.itu.int/rec/T-REC-H.262", Visited on 2014/11/27.

[14] ITU-T. H.263 : Video coding for low bit rate communication - recommendation h.263. "http://www.itu.int/rec/T-REC-H.263", Visited on 2014/11/27.

[15] ITU-T. H.264 : Advanced video coding for generic audiovisual services - recommendation h.264. "http://www.itu.int/rec/T-REC-H.264", Visited on 2014/11/27.

[16] ITU-T. Introduction to streaming media. "http://www.cod.edu/it/streamingmedia/intro.htm", Visited on 2014/11/27.

[17] Jia, Z. M. (2009). *MoViShare: building location-aware mobile social networks for video sharing*. PhD thesis, School of Computing Science-Simon Fraser University.

[18] JWPlayer (2014-11-30). About rtmp streaming. "http://support.jwplayer.com/customer/portal/articles/1430349-about-rtmp-streaming" visited on 2014/11/30.

[19] Lindeberg, M., Kristiansen, S., Plagemann, T., and Goebel, V. (2011). Challenges and techniques for video streaming over mobile ad hoc networks. *Multimedia Systems*, 17(1):51–82.

[20] Lu, J. (2000). Signal processing for internet video streaming: A review. In *Electronic Imaging*, pages 246–259. International Society for Optics and Photonics.

[21] Ma, L. (2011). *Location-aware mobile social networking for video sharing*. PhD thesis, Applied Science: School of Computing Science.

[22] Meyer, R. (2013). *Adaptation mechanism for streaming server applications optimized for the use on mobile devices with limited resources*. PhD thesis, Technische Universität Dresden.

[23] Myers, J. G. (1997). Simple authentication and security layer (sasl).

[24] Ott, J., Wenger, S., Sato, N., Burmeister, C., and Rey, J. (2006). Extended rtp profile for real-time transport control protocol (rtcp)-based feedback (rtp/avpf). *Request for Comments*, 4585.

[25] Padmanabhan, V. N., Wang, H. J., Chou, P. A., and Sripanidkulchai, K. (2002). Distributing streaming media content using cooperative networking. In *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 177–186. ACM.

[26] Parmar, H. and Thornburgh, M. (2014). Adobes real time messaging protocol.

[27] Postel, J. (1981). Transmission control protocol.

[28] Protocol, U. D. (1980). Rfc 768 j. postel isi 28 august 1980. *Isi*.

[29] Richardson, I. E. (2004). *H. 264 and MPEG-4 video compression: video coding for next-generation multimedia*. John Wiley & Sons.

[30] Saint-Andre, P. (2011a). Extensible messaging and presence protocol (xmpp): Core.

[31] Saint-Andre, P. (2011b). Extensible messaging and presence protocol (xmpp): Instant messaging and presence.

[32] Schulzrinne, H. (1998). Real time streaming protocol (rtsp).

[33] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V. (2003). Rtp: A transport protocol for real-time applications. Technical report.

[34] Silva, J., Cabral, D., Fernandes, C., and Correia, N. (2012). Real-time annotation of video objects on tablet computers. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*, page 19. ACM.

[35] Sullivan, G. J., Ohm, J.-R., Han, W.-J., and Wiegand, T. (2012). Overview of the high efficiency video coding (hevc) standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(12):1649–1668.

[36] Sun, H., Vetro, A., and Xin, J. (2007). An overview of scalable video streaming. *Wireless Communications and Mobile Computing*, 7(2):159–172.

[37] Team, J. V. (2002). Coding of audio-visual objectspart 10: Advanced video coding. In *Awaji MPEG Meeting, Japan*.

[38] Tudor, P. (1995). Mpeg-2 video compression. *Electronics & communication engineering journal*, 7(6):257–264.

[39] Walker, M., Nilsson, M., Jebb, T., and Turnbull, R. (2003). Mobile video-streaming. *BT Technology Journal*, 21(3):192–202.

[40] Wang, E. (2015-10-14). Libvlc android sample. "https://bitbucket.org/edwardcw/libvlc-android-sample".

[41] Ware, B. et al. (2002). *Open Source Development with LAMP: Using Linux, Apache, MySQL and PHP*. Addison-Wesley Longman Publishing Co., Inc.

[42] Wiki, V. (2010-09-09). Androidcompile. "https://wiki.videolan.org/AndroidCompile".

[43] Yamamoto, D., Masuda, T., Ohira, S., and Nagao, K. (2008). Video scene annotation based on web social activities. *IEEE multimedia*, (3):22–32.