Design and Evaluation of Mobile-to-Mobile Multimedia Streaming using REST-based Mobile Services

AAMER SATTAR CHAUDRY

Master’s Degree Project
Stockholm, Sweden

XR-EE-LCN 2011:001
Chair of Communication Networks
RWTH Aachen University
Prof. Dr.-Ing. B. Walke

Master Thesis

Design and Evaluation of Mobile-to-Mobile Multimedia Streaming using REST-based Mobile Services
Entwurf und Bewertung von Mobile-to-Mobile Multimedia Streaming mit REST-basiert mobile Dienste

of

Aamer Sattar Chaudry
Matriculation Number: 300208

Aachen, January 26, 2011

Supervised by:
o. Prof. Dr.-Ing. B. Walke
Fahad Aijaz, M.Sc.

This publication is meant for internal use only. All rights reserved. No liabilities with respect to its content are accepted. No part of it may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.
I assure, that this work has been done solely by me without any further help from others except for the official attendance by the Chair of Communication Networks. The literature used is listed completely in the bibliography.

Aachen, January 26, 2011

(Aamer Sattar Chaudry)
First and foremost, I respectfully thank to Prof. Dr.-Ing. B. Walke for giving me an opportunity to conduct my thesis project in the world class environment at the ComNets Research Group. In the KTH University (Sweden), I would like to express my deep appreciation to Prof. Viktoria Fodor for agreeing to supervise my thesis. Special thanks to Muzzamil Aziz Chaudhary, Maciej Muehleisen and Ralf Jennen for their valuable suggestions for the accomplishment of my different project tasks. Additionally, I would like to express my gratitude to Fahad Aijaz for his support and technical guidance, which determines the success of this work.

I am also thankful to all my colleagues and seniors at ComNets, specially Benedikt Wolz, Afroditi Kyrligkitsi, Senay Kaynar and Adel Zalok for their support and encouragement during my stay at ComNets.

Finally, I am indebted my deepest thanks to my parents and family for their love, support and prayers during my studies far away from them.
ABSTRACT

Today, it has become conveniently possible to host Web Services (WS) on mobile node using a Mobile Web Server (MWS). To address a vast range of use cases, these hosted mobile WS may be implemented to offer synchronous and asynchronous execution styles depending upon the requirements of mobile application. Thus, the MWS actually provides the necessary architectural capabilities to handle and process incoming requests for each class of service. But, it is very vital for these servers to simplify the service access and creation mechanisms, so that, the processing overheads on the hosting node are reduced. Previously, research has shown promising optimizations in MWS processing by using the REST architecture style for service access and creation. However, the mobile WS offered by the existing MWS uses XML based payload for information exchange, which restricts the incorporation of the rich multimedia content, such as, audio and video data. As a consequence, the true potential of the REST-based server architecture is not utilized.

This thesis addresses architectural and transport layer issues to enable the exchange of rich multimedia content between mobile nodes using mobile WS over the live wireless data networks. The research work is focused on the implementation of multimedia streaming protocol standards, such as, the RTSP and the RTP, into the existing REST-based MWS architecture. Also, to enable the controlled Mobile-to-Mobile (M2M) media streaming capabilities, the thesis uses both TCP and UDP as transport layer protocols for signaling and data transmission, respectively. The control functions are implemented by mapping the synchronous and asynchronous mobile WS to the RTSP methods. The implementation extends the states of the asynchronous mobile WS to offer these multimedia control functions. The issues related to the firewalls and Network Address Translation (NAT) are addressed by the development of an Intermediate Access Gateway (IAG), which offers the functionality based on the STUN and TURN concepts. This work enables mobile WS based M2M media streaming either through the directly established connection with the peers, or via the IAG. Thus, the developed proof-of-concept prototype demonstrates the streaming capabilities of the extended MWS architecture over any wireless data networks.
<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>9</td>
</tr>
<tr>
<td>2 Streaming Protocols</td>
<td>11</td>
</tr>
<tr>
<td>2.1 Real Time Streaming Protocol (RTSP)</td>
<td>11</td>
</tr>
<tr>
<td>2.1.1 RTSP Operations</td>
<td>12</td>
</tr>
<tr>
<td>2.1.2 RTSP Methods</td>
<td>12</td>
</tr>
<tr>
<td>2.1.3 RTSP Session States</td>
<td>14</td>
</tr>
<tr>
<td>Client State Machine</td>
<td>14</td>
</tr>
<tr>
<td>Server State Machine</td>
<td>14</td>
</tr>
<tr>
<td>2.1.4 RTSP versus the HTTP</td>
<td>15</td>
</tr>
<tr>
<td>2.2 Real Time Transport Protocol (RTP)</td>
<td>15</td>
</tr>
<tr>
<td>2.2.1 RTP Frame Structure</td>
<td>16</td>
</tr>
<tr>
<td>2.3 Real Time Transport Control Protocol (RTCP)</td>
<td>18</td>
</tr>
<tr>
<td>2.4 Session Description Protocol (SDP)</td>
<td>19</td>
</tr>
<tr>
<td>2.4.1 SDP Specification</td>
<td>20</td>
</tr>
<tr>
<td>3 RESTful Mobile Web Services</td>
<td>25</td>
</tr>
<tr>
<td>3.1 Representational State Transfer (REST)</td>
<td>25</td>
</tr>
<tr>
<td>3.2 Mobile Web Services using REST</td>
<td>26</td>
</tr>
<tr>
<td>3.2.1 RESTful Synchronous Web Services</td>
<td>26</td>
</tr>
<tr>
<td>3.2.2 RESTful Asynchronous Web Services</td>
<td>27</td>
</tr>
<tr>
<td>3.2.2.1 Service Creation and Invocation</td>
<td>27</td>
</tr>
<tr>
<td>3.2.2.2 Service Monitoring</td>
<td>27</td>
</tr>
<tr>
<td>3.2.2.3 Service Control</td>
<td>28</td>
</tr>
<tr>
<td>4 Network Address Translation (NAT)</td>
<td>29</td>
</tr>
<tr>
<td>4.1 NAT Terminology</td>
<td>29</td>
</tr>
<tr>
<td>4.2 NAT Traversal for P2P operation</td>
<td>30</td>
</tr>
<tr>
<td>4.2.1 Simple Traversal of UDP through NATs (STUN)</td>
<td>30</td>
</tr>
<tr>
<td>4.2.1.1 Simple Traversal of UDP through NATs (STUN) Configuration</td>
<td>31</td>
</tr>
<tr>
<td>4.2.1.2 Discovery of NAT</td>
<td>31</td>
</tr>
<tr>
<td>4.2.1.3 Hole Punching</td>
<td>32</td>
</tr>
<tr>
<td>4.2.1.4 Types of NAT</td>
<td>32</td>
</tr>
<tr>
<td>4.2.1.5 Discovery of NAT types</td>
<td>34</td>
</tr>
<tr>
<td>4.2.1.6 STUN Reservations</td>
<td>36</td>
</tr>
<tr>
<td>4.2.1.7 P2P UDP sessions establishment using STUN</td>
<td>36</td>
</tr>
<tr>
<td>4.2.2 Traversal Using Relay NAT (TURN)</td>
<td>40</td>
</tr>
<tr>
<td>4.2.2.1 TURN Configuration and Operational Overview</td>
<td>41</td>
</tr>
<tr>
<td>4.2.3 Advantages of TURN over STUN</td>
<td>43</td>
</tr>
<tr>
<td>4.2.4 Overheads using TURN</td>
<td>43</td>
</tr>
<tr>
<td>5 Mobile-to-Mobile Streaming using Mobile Web Server</td>
<td>45</td>
</tr>
<tr>
<td>5.1 General Implementation Architecture for M2M Streaming</td>
<td>45</td>
</tr>
</tbody>
</table>
5.2 RTP Implementation ................................................. 46
  5.2.1 Implemented RTP Frame Structure ........................ 46
  5.2.2 Maximum RTP Payload Size ................................. 48
  5.2.3 Implemented classes ........................................ 49
  5.2.4 Debugger Snapshots for RTP Header ....................... 50
5.3 Implementation Scenarios for M2M Streaming .................. 53
  5.3.1 Network Investigation and Testing .......................... 54
    5.3.1.1 Scenario I: Direct M2M media streaming .............. 54
    5.3.1.2 Scenario II : M2M Media streaming using Intermediate Access Gateway (IAG) .................. 57
    5.3.1.3 Summary of the test results ........................... 63
5.4 Implementation Architecture for Scenario I & Scenario II (a/b) .... 65
5.5 RTSP Implementation over REST .................................. 68
  5.5.1 RTSP Methods using Synchronous Services ................. 68
    OPTIONS .................................................... 68
    DESCRIBE ................................................ 70
  5.5.2 RTSP Methods using Asynchronous Services .............. 74
    SETUP .................................................... 74
    PLAY .................................................... 76
    PAUSE .................................................. 77
    TEARDOWN ............................................... 78
  5.5.3 Integration of Streaming functionality over UDP .......... 79
5.5.4 Differences from the existing REST middleware ............. 80
5.5.5 Fully Working Implementation for Scenarios I and II (a/b) ... 84
  5.5.5.1 Media Server and Media Client Configuration .......... 84
  5.5.5.2 Intermediate Access Gateway (IAG) Configuration .... 85
  5.5.5.3 Streaming Media Server ................................ 85
  5.5.5.4 Streaming Media Client ................................ 86
  5.5.5.5 Multimedia Streaming in Scenario I .................... 91
    OPTIONS/DESCRIBE ..................................... 91
    SETUP .................................................... 92
    PLAY .................................................... 92
    PAUSE .................................................. 93
    TEARDOWN ............................................... 93
  5.5.5.6 Multimedia Streaming in Scenario IIa ................... 94
    OPTIONS/DESCRIBE ..................................... 94
    SETUP .................................................... 95
    PLAY .................................................... 95
    PAUSE .................................................. 96
    TEARDOWN ............................................... 96
  5.5.5.7 Multimedia Streaming in Scenario IIb ................... 97
    OPTIONS/DESCRIBE ..................................... 98
    SETUP .................................................... 98
    PLAY .................................................... 98
    PAUSE .................................................. 99
    TEARDOWN ............................................... 100
6 Evaluation and Performance of the M2M and IAG Scenarios ....... 101
  6.1 General Constraints of the Scenarios ......................... 101
    6.1.1 Constraints of Scenario I .............................. 101
6.1.2 Constraints of Scenario II .................................................. 101
6.2 Inter-scenario Evaluation ...................................................... 102
6.2.1 Comparison of Request-Response Delay .............................. 103
   6.2.1.1 Sequence Flow Diagram for Scenario I .......................... 103
   6.2.1.2 Sequence Flow Diagram for Scenario II ......................... 103
   6.2.1.3 Service Times for OPTIONS and DESCRIBE requests .......... 104
   6.2.1.4 OPTIONS Request-Response Delay ................................ 105
   6.2.1.5 DESCRIBE Request-Response Delay ............................. 107
   6.2.1.6 SETUP Request-Response Delay .................................. 107
   6.2.1.7 PLAY Request-Response Delay .................................... 108
6.2.2 Comparison of Streaming Packet Latency ........................... 109
   6.2.2.1 Testbed Configuration ............................................. 109
   6.2.2.2 Total File Buffering Time ....................................... 109
   6.2.2.3 Inter-Packet Delay Difference .................................. 111
6.3 Multimedia Streaming Performance over different networks ........ 112
   6.3.1 PDI for Uplink and Downlink ..................................... 112
   6.3.2 PDI vs RTP payload size for Uplink ............................... 113
   6.3.3 Requirements of RTCP and RSVP protocols ....................... 115
6.4 General Overheads in the Implemented Design Models .............. 116
   6.4.1 The TCP Keep Alive Messages .................................... 116
   6.4.2 UDP Keep Alive Messages ......................................... 117
6.5 Performance Conclusion .................................................... 117

7 Theses .................................................................................. 119

8 Conclusions and Outlook ...................................................... 121
   8.1 Conclusions ................................................................. 121
   8.2 Outlook ................................................................. 122

List of Figures ......................................................................... 123

List of Tables .......................................................................... 127

A Abbreviations ..................................................................... 129

Bibliography ........................................................................ 131
CHAPTER 1

Introduction

With the advent of new architectural styles, the hosting of Web Services (WS) is not only possible on the high-end servers, but also on resource-constrained mobile devices. These WS can be either short lived synchronous or long lived asynchronous in term of their execution styles, which depend upon the use case. Today, with the advancement of IP networks, and the intro of new light weight services with textual constructs, the demand of incorporating mobile multimedia capabilities to such services has become imminent. Usually, in the existing implementations, the multimedia content is hosted and published to the mobile clients only through high-end servers that reside at a fixed location. For mobile nodes, however, there is an increasing requirement that instead of being only the service consuming clients, the hosting of multimedia based content can also be made possible by the mobile servers.

This thesis focuses on the design and evaluation of Mobile-to-Mobile (M2M) multimedia streaming using Representational State Transfer (REST) based mobile services. As an experimental platform, these services are enabled for the Google Android smart phones to support the concept of both mobility and portability.

Today, the M2M streaming is not very common, especially through the mobile nodes being the multimedia hosts. Also, for such nodes, no mechanism to control the multimedia is developed such as, through the RTSP standard. Along with that, there is no API available for developers for M2M streaming applications. Moreover, in social networks the content is shared with the peers by storing it on the common third party servers. For this reason, this thesis provides a M2M streaming server where the streaming content is shared directly between the peers without involving the third party servers. Also, both the streaming server and the streaming client can be mobile where client is provided with the functionality to control the streaming through RTSP standard over any wireless data network. In addition to that, standardized API is provided with in the service provisioning framework for the development of multimedia streaming applications between mobile nodes (which may be run on different platforms). The thesis report is organized as follows:

Chapter 2: Discusses about the different standardized multimedia protocols that are involved in streaming process, such as, Real Time Streaming Protocol (RTSP), Real Time Transport Protocol (RTP), Real Time Transport Control Protocol (RTCP) and Session Description Protocol (SDP).

Chapter 3: Provides an overview on the Representational State Transfer (REST) architecture in terms of the synchronous and asynchronous servers that forms the basis for the thesis implementation.

Chapter 4: Gives the general overview of Network Address Translation (NAT). Then, the methods to find the existence and type of the NATs are discussed. Finally, the techniques and protocols that are involved for the NAT traversal like Simple Traversal of UDP through NATs (STUN) and Traversal Using Relay NAT (TURN) are presented.

Chapter 5: Describes all the required details regarding the design and development of this thesis. Firstly, it explains the implementation of RTSP protocol by using the REST architecture to provide the controlled streaming functionality. Secondly, it also explains the implemented RTP frame
structure for carrying the media data. Finally, it discusses about all the scenarios developed in this work to provide the M2M multimedia streaming functionality over the live wireless data networks like WLAN, EDGE and UMTS etc.

Chapter 6: Presents the evaluation of different scenarios that are designed and implemented in this thesis. Then, their behavior and performance aspects are discussed based on the experiments with the different live wireless and operator networks.

Chapter 7: Outlines the thesis’ results as concrete list of statements.

Chapter 8: Concludes the thesis work and suggests possible extensions.
CHAPTER 2

Streaming Protocols

Mainly there are 3 major protocols which are involved for control and transmission of real time multimedia data over the internet. The first one is the Real Time Streaming Protocol (RTSP), which is specified as an application layer protocol and works with lower layer protocols to enable controlled delivery of streamed data over IP network. The second one is the Real Time Transport Protocol (RTP), which provides support to User Datagram Protocol (UDP) for the transport of the real time multimedia data over IP network. The third one is the Real Time Transport Control Protocol (RTCP) that sends out-of-band control information for an RTP flow to provide feedback on the Quality of Service (QoS) being provided by the RTP. In conjunction to these three major protocols, the Session Description Protocol (SDP) provides an acceptable format for the initialization data which is required in setting up and control of both the signaling and data channels for real time multimedia transmission.

In this chapter, different streaming protocols like RTSP, RTP, RTCP and SDP will be discussed in the light of literature and other sources as explained by researchers and practitioners. First the RTSP will be discussed along with its major and optional methods. Then RTP protocol along with its header structure will be described. After that there will be brief overview on the RTCP protocol which is also one of the important streaming protocol, although it has not been implemented in this thesis. Finally, the SDP protocol along with its different parameters will be discussed in detail.

2.1 Real Time Streaming Protocol (RTSP)

RTSP is an application layer protocol which is designed to work with lower layer protocols like the RTP to provide streaming service over the internet. It is simply a client-server multimedia protocol to enable controlled delivery of streamed data over IP network [6]. It acts as “network remote control” for multimedia servers [4]. In RTSP, the client controls the media server by providing “VCR-style” remote control functionality like “play” and “pause” etc.

RTSP is more of a framework than a protocol [2]. It is meant to control multiple data delivery sessions, thus provide a way to choose delivery channels such as UDP, TCP and IP-multicast [2]. The delivery mechanisms are based solely on the RTP.

RTSP takes advantage of streaming by which multimedia data is usually sent across the network in streams, instead of storing large multimedia files first and then perform playback. By streaming, data is broken down into packets with size suitable for transmission between the servers and clients. This data then flows through the transmission, decompression and playback pipeline just like a water stream [6]. A client can play the first packet; decompress the second, while receiving the third [6]. Thus the user can start enjoying the multimedia without waiting to the end of transmission to get the entire media file. Both live data feeds and stored clips can be the sources of data [2].

There is no notion of an RTSP connection, instead, a server maintains a session labeled by an identifier [4]. An RTSP session is in no way tied to a transport-level connection such as a TCP
During an RTSP session, an RTSP client may open and close many reliable transport connections to the server to issue RTSP requests [4]. Alternatively, it may use a connectionless transport protocol such as the UDP [4].

2.1.1 RTSP Operations

RTSP supports the following operations:

Retrieval of media from media server [4]: The client can request a presentation description (via HTTP) or some other method. Then the media server can be requested to setup the stream and send the requested media data on it.

Invitation of a media server to a conference [4]: A media server can be “invited” to join an existing conference, either to playback the media into the presentation or to record all or a subset of the media in a presentation [4].

Addition of media to an existing presentation [4]: It is particularly for live presentations and used when the server tells the client about any additional media becoming available.

2.1.2 RTSP Methods

RTSP basically has 11 methods out of which 6 are major methods because they are either suggested as Required methods or Recommended methods and have been shown in the Figure 2.1. The remaining are considered as Optional methods. Both these major and optional methods are now discussed theoretically. The header structures for each of these major methods will be discussed later in section 5.5.

![Figure 2.1: RTSP Major Methods](image)

1. **OPTIONS:**
   In the OPTIONS request, the client can get the methods available at the server. In other and more generalized terms, by this request, the client or the server tells the other party the options it can accept like e.g. in Figure 2.1, by the OPTIONS request, server replies to the
client that it can have the options of **DESCRIBE, SETUP, PLAY, PAUSE and TEARDOWN requests** to send.

2. **DESCRIBE:**
   In the **DESCRIBE** request, client retrieves (low level) description of the media object from server. This server can be either any **web server** from where the initialization information can be received by the protocols like HTTP, email attachment, etc, or, directly the media server from where the description can be retrieved in the formats like SDP, XML etc. The **DESCRIBE** request-response pair constitutes the media initialization phase of RTSP [4].

   Media initialization is a requirement for any RTSP-based system [4], so the **DESCRIBE** response should contain all the media initialization information of the resource(s) that it describes.

3. **SETUP:**
   In the **SETUP** request, the client requests the media server to allocate resources for stream and starts an RTSP session. By using this request, the client also specifies the transport mechanism which it will use for the retrieval of the media data. In this request, the client also mentions the transport parameters such as delivery protocol and port number etc which are acceptable to the client during data transmission and retrieval. In the **SETUP** response, the transport parameters which will be used by the server will also be enclosed. The server also generates a valid **session identifier** which is delivered as a specific header field in the **SETUP** response. Thus the **SETUP** request-response will constitute the transport initialization phase.

4. **PLAY:**
   In the **PLAY** request, the client asks the server to start sending data on a stream allocated via the **SETUP**. Then, the media server starts to transmit the media data according to the mechanism that has been decided during the **SETUP** request. The client must not issue a **PLAY** request until any outstanding **SETUP** requests have been acknowledged as successful [4].

5. **PAUSE:**
   In the **PAUSE** request, the client temporarily halts the stream delivery without freeing the allocated server resources. If in the request, the name of any particular stream is provided, then only playback (or recording) of that particular stream is halted from the overall presentation. For example, if the presentation contains both audio and video streams and **PAUSE** request is sent only for audio, this is equivalent to muting. However if the request contains the name of a whole presentation or group of streams, then whole presentation (i.e. both audio and video streams) or that particular group is halted. Then, on a subsequent **PLAY** request, the delivery resumes from the point where it was paused.

6. **TEARDOWN:**
   In the **TEARDOWN** request, the client asks the server to stop the delivery of the specified stream and free the resources associated with it. After this request, any RTSP session identifier associated with the session (which has been issued via the **SETUP**) will no longer be valid. Unless, all transport parameters are defined by the session description, a new **SETUP** request has to be issued before the session can be played again [4].

Up to now the six major RTSP methods (which are considered either **required** or **recommended**) have been discussed. The remaining five RTSP methods that can be considered as **optional** will be discussed now.

1. **ANNOUNCE:**
   The **ANNOUNCE** can be sent either by client or server. When sent from client to server, the
2. Streaming Protocols

ANNOUNCE request posts the description of the media object (which is specified in the request) to a server. But when sent from server to client, ANNOUNCE request updates the session description of media object in real-time [6].

2. GET_PARAMETER:
By the GET_PARAMETER request, the value of a parameter of a presentation or stream (which is specified in the request) can be retrieved.

3. SET_PARAMETER:
By the SET_PARAMETER request, the value of a parameter of a presentation or stream (which is specified in the request) can be set.

4. REDIRECT:
By the REDIRECT request (which is sent from the server to client), the server informs the client that it must connect to another server location. In simple terms, the current server is redirecting the client to a new server.

5. RECORD:
By the RECORD request, the client requests the server to start recording of media data.

For more information on the RTSP and its different methods, [4] can be referred.

2.1.3 RTSP Session States

acsRTSP is a stateful protocol where server needs to maintain state by default in almost all cases. It is very important for RTSP server to maintain “session states” like INIT, READY and PLAYING in order to correlate the SETUP, PLAY, PAUSE and TEARDOWN RTSP requests with a stream.

The remaining requests like OPTIONS, ANNOUNCE, DESCRIBE, GET_PARAMETER, SET_PARAMETER do not have any effect on client or server states.

Client State Machine

The client assumes the following states [4]:

- **INIT**: The SETUP request has been sent, waiting for reply [4].
- **READY**: The SETUP reply has been received, or, while in the PLAYING state, the PAUSE reply has been received.
- **PLAYING**: The PLAY reply has been received.

In general, the client changes the state on receipt of replies to request.

Server State Machine

The server assumes the following states [4]:

- **INIT**: It is the initial state when no valid SETUP request has been received so far, or, while in the PLAYING state, the last TEARDOWN request has been received successfully.
- **READY**: Either the last SETUP request has been received successfully, or, while in the PLAYING state, the last PAUSE request has been received successfully.
2.2. Real Time Transport Protocol (RTP)

- **PLAYING**: The last PLAY request has been received successfully.

In general, the server changes state upon receiving the requests.

The flow of transition from one RTSP session state to the other on both the client and server is shown in Figure 2.2. The server changes its state on receiving the RTSP requests while client changes its states on receiving the responses of RTSP requests that it has issued previously.

![Figure 2.2: RTSP State transition diagram][9]

2.1.4 RTSP versus the HTTP

The RTSP is intentionally specified to be similar in syntax and operation to the HTTP/1.1 [2]. However, it still differs from HTTP in several aspects:

1. The RTSP has its own and different protocol identifier (i.e. rtsp://) which is not similar to that of HTTP that uses http:// or https://.

2. The HTTP is basically an “asymmetric protocol” where a client issues the requests and the server responds [6]. In contrast to that, RTSP is a “symmetric protocol” where both the client and the server can issue the request. For example, a media server can issue the REDIRECT request to direct its connection to some other media server in order to retrieve the remaining media data from that media server.

3. As mentioned above, the RTSP is a stateful protocol. In contrast to the RTSP, the HTTP is a stateless protocol.

4. In case of the RTSP, the data is carried out-of-band by a different protocol. The protocol and channel carrying the RTSP requests are independent and can be different from the data delivery channel and protocol. With the HTTP, mostly the requested data is delivered on the same channel that is used by request/response. So, it can be said that data is carried in-band in case of HTTP.

5. The RTSP is actually a transport independent protocol. It may use either an Unreliable Datagram Protocol (UDP), a Reliable Datagram Protocol (RDP) or a reliable stream protocol such as Transport Control Protocol (TCP). In contrast to that, the HTTP uses TCP as transport protocol.

For more differences between the RTSP and the HTTP, [4] can be referred.

2.2 Real Time Transport Protocol (RTP)

The RTP provides a standardized packet format for the transmission of real time media over IP network. It was primarily designed for multicast of real time data, but can also be used for unicast.
It is used in both one-way transports for providing on-demand services like Audio on Demand (AoD), Video on Demand (VoD) and also for interactive services like video conferencing.

Generally, over internet, the packets that are sent over the network experience unpredictable delay and jitter. In addition to this, the packets may follow different routes causing the packets to be received at the receiver out of sequence. This is not feasible for the multimedia applications that require appropriate timing and proper sequence for the playback. To handle these issues, the RTP contributes the Time Stamping and the Sequencing facilities. By time stamping, RTP provides the service of Timing Reconstruction to minimize the effect of variable delay/jitter and for the proper synchronization of different audio-video streams during the playback of the media content. With the sequencing, the RTP cares for making the received data in proper sequence and detects the loss of any packets at the receiver. In addition to this, the RTP also provide services for source and content identification.

However, it is important to know that RTP does not have any mechanism to ensure timely delivery by itself. It only provide hook-ups for applications to achieve that. Secondly, it does not have any delivery mechanism like multiplexing and port numbers of its own. It actually provide end-to-end delivery services for real time data by running over transport layer protocols like UDP or TCP. The RTP is mostly used over the UDP because the TCP does not support multicasting. Secondly, for real time data, a retransmission strategy for the lost or corrupted packets is not feasible in order to avoid congestion in the network by the retransmitted packets.

### 2.2.1 RTP Frame Structure

A typical RTP header contains a fixed header of 12 bytes followed by extension header. Variable size payload is inserted after the RTP header inside RTP packet.

The RTP generalized header structure along with Extension Header is shown in Figure 2.3.

![Figure 2.3: RTP Frame Structure](image)

Now all the different fields of RTP frame shown in Figure 2.3 will be discussed.

**Version (V):** This 2 bit field identifies the version of the RTP. The newest version is 2.

**Padding (P):** This is 1 bit padding field and when it is set, the RTP packet contains one or more additional padding octets at the end which are not part of the payload. In this case, the
value of the last octet (byte) contains a count of how many padding octets should be ignored, including itself. Padding may be needed by some encryption algorithms with fixed block sizes [3].

**Extension(X):** This is one bit *extension* bit and when it is set, the generalized header (after CSRC list) must be followed by exactly one variable header extension [3]. It contains 16-bit header field that counts the number of 32 bit words in the extension, excluding the four-octet extension header (therefore zero is a valid length) [3].

The *header extension* is profile-specific extension to the generalized header which is provided to allow individual implementations to experiment with new *format-independent payload* functions that require additional information to be carried in the RTP data packet header. So, if a particular class of applications under a specific profile needs additional functionality, that is independent of payload format, it should define additional fields in this header extension. Then, those applications will be able to quickly and directly access the additional fields while profile independent applications can still process the RTP packets by interpreting only generalized header fields. But, if the additional functionality is commonly needed across all profiles, then a new RTP version should be defined for making a permanent change to the generalized header.

It is to be noted that additional information which is required for a *particular payload format* should not use this header extension. It should either be carried in the payload section, or by a reserved value in data pattern of the RTP packet.

**Contributor Count (CC):** The *Contributor Count* (or CSRC count) is a 4 bits field which contains the number of CSRC identifiers (contributors) that follow the fixed header. If CC > 1, then the RTP payload contains data from more than 1 sources. In such case, *SSRC identifier* will be a *Mixer*. The Mixer actually combines several *flows* into a single new one. It then appears as a new source by using the new SSRC and puts the original SSRCs into the CSRC list.

**Synchronization Source (SSRC) identifier:** The SSRC identifier is a 32 bits field which identifies the synchronization source [3]. It is actually a randomly chosen number to distinguish between the synchronization sources within the same RTP session. Thus, no two synchronization sources within the same RTP session will have the same SSRC identifier. In case of more than one original source, the SSRC indicates where the data was combined (i.e. at Mixer), or the original source of the data if there is only one source.

Although the probability of multiple sources choosing the same identifier is low, all RTP implementations must be prepared to detect and resolve collisions [3].

**Contributing Source (CSRC) identifiers or CSRC List:** the CSRC identifiers is actually a list of 0-15 items where each item is actually a 32 bit CSRC identifier of that source which is contributing for the payload contained in an RTP packet. The number of identifiers is given by the CC field [3]. If there are more than 15 contributing sources, only 15 may be identified [3]. The CSRC identifiers are inserted by the mixers, using the SSRC identifiers of the contributing sources [3].

It must be noted that first twelve octets(bytes) of RTP header must be present in every RTP packet. But, the list of CSRC identifiers may be present only when inserted by a Mixer. In such case, CC value must be greater than 0 and the SSRC identifier must be used by the Mixer.

**Marker (M):** The interpretation of this 1 bit field is usually done by the profile. It is used by the application to indicate e.g. the end of its data or to allow significant events, such as, frame
boundaries to be marked in the packet stream when the packet contains multiple frames of multimedia (audio/video) data.

A profile may define additional marker bits or specify that there is no marker bit by changing the number of bits in the Payload Type field [3]. If there are any marker bits, they should be located as the most significant bits of the octet.

**Payload Type (PT):** This is a 7 bits field that identifies the format (e.g. Encoding) of the RTP payload [3]. It determines payload interpretation by the application. The receiver must ignore packets with payload types that it does not understand. An RTP source may change the payload type during the session, but this field is not intended for multiplexing separate media.

A profile may specify a default static mapping of payload type codes to payload formats. However, additional payload type codes may be defined dynamically [3]. As this a 7 bit field, so, mapping for the payload type codes ranges from 0127. IANA has categorized different codes both for static mapping of payload types for existing standard profiles/formats and for dynamic mapping of additional/future coming profiles or format types as shown [8]:

- 0 to 34 (static allocation)
- 35 to 71 (unassigned)
- 72 to 76 (reserved for RTCP conflict avoidance)
- 77 to 95 (unassigned)
- 96 to 127 (dynamic allocation)

**Sequence number:** The sequence number is a 16 bits field which increments by one for each RTP data packet sent, and may be used by the receiver to detect packet loss and to restore packet sequence [3]. The initial value of the sequence number should be random (unpredictable) to make known-plaintext attacks on encryption more difficult, even if the source itself does not encrypt [3].

**Timestamp:** The timestamp is also a 32 bits header field that reflects the sampling instant of the first octet in the RTP data packet [3]. The sampling instant must be derived from a clock that increments monotonically and linearly in time to allow synchronization and jitter calculations [3]. If the RTP packets are generated periodically, the nominal sampling clock is used but not the reading of the system clock [3]. Consider the case for fixed rate audio where timestamp clock will increment by one for each sampling period. If an audio application reads blocks from the source device by covering 200 sampling periods, then the timestamp would be increased also by 200 for each block, regardless of whether the block is transmitted in a packet or dropped as silent [3].

The initial value of the time stamp is also taken random just like in the case for the sequence number.

For more information on RTP protocol or its header fields, [3] can be referred.

### 2.3 Real Time Transport Control Protocol (RTCP)

The primary function of RTCP is to provide feedback on the Quality of Service (QoS) being provided by the RTP. It provides *out-of-band* control information for an RTP flow. It partners the
RTP in the delivery and packaging of multimedia data, but does not transport any data itself.

The RTCP is used to transmit periodic control packets to participants in a streaming multimedia session, using the same distribution mechanism as the data packets [3]. So, the underlying protocol must provide multiplexing of the data and control packets, for example, using separate port numbers with UDP [3]. Usually, the RTCP is assigned UDP port next to the port assigned for RTP in sequence. The control packets sent by the RTCP carry the messages to control the flow and quality of data and allows the recipient to send feedback to the source(s).

The RTCP gathers statistics on a media connection and information, such as, bytes sent, packets sent, lost packets, jitter, feedback and round trip delay.

For more information on RTCP protocol, [3] can be referred.

2.4 Session Description Protocol (SDP)

The SDP is a well defined format for conveying sufficient information for discovery and participation in a multimedia session. The purpose of the SDP is to convey information about media streams in multimedia sessions to allow the recipients of a session description to participate in the session [7]. It is used to describe multimedia sessions for the purposes of session announcement, session invitation, and other forms of multimedia session initiation [7]. A multimedia session is defined as a set of media streams that exist for some duration of time [7]. The times during which the session is active need not to be continuous [7].

The SDP is actually a general purpose protocol for using in a wider range of network environments and applications. It gives both the session and media related information.

For session, it describes the information like:

- Session name and purpose
- Time(s) the session is active
- The media comprising the session
- Information to receive those media (addresses, ports, formats etc)

In relation to media, it describes the information like:

- The type of media (video, audio, etc)
- The transport protocol (RTP/UDP/IP etc)
- The format of the media (H.261 video, MPEG video, etc)
- Remote address for media
- Transport port for media

The remote address and port are media and transport protocol dependent. They can be either that address or port to which data is sent, or where the data will be received. In contrast to this, they can also be used to establish a control channel for the actual media flow.
2.4.1 SDP Specification

An SDP session description consists of a number of lines of text of the form <type> = <value>[7]. <type> is always on left hand side of “=” with exactly one character and is case-significant. <value> is a structured text string whose format depends on <type>. It will be always on right hand side of “=” and will also be case-significant unless a specific field defines it otherwise.

A session description consists of a session-level description which is actually the details that apply to the whole session and all media streams. SDP can also include optionally several media-level descriptions where each media description provides details that apply onto a single media stream. The session-level section starts with a ‘v=’ line and continues to the first media-level section. The first media section starts with an ‘m=’ line and continues to the next media section or simply end up the whole session description. In general, session-level values are the default for all media unless overridden by an equivalent media-level value [7].

The required lines and some optional lines of session description in their proposed order of appearance have been discussed below. It is beneficial to follow the prescribed fixed order of lines as it enhances the chances for error detection and allows simple parsing of SDP.

1. **Protocol Version [v]**
   It is the first required field of SDP from where the session level description starts. It gives the version of SDP which by latest is “0” i.e. v=0

2. **Origin [o]**
   It is the second required field of SDP which tells about the originator of the session. It includes the below mentioned subfields:

   o= <username> <session id> <version> <network type> <address type> <address (of originator)>

   Here <username> is the user’s login-id (which should not contain any spaces) on the originating host or it is “-” if the originating host does not support the concept of user ids [7].
   <session id> is a numeric string such that the tuple of <username>, <session id>, <network type>, <address type> and <address> form a globally unique identifier for the session [7].
   Its allocation method is up to creating tool but normal suggestion is that a Network Time Protocol (NTP) timestamp should be used to ensure uniqueness. The next subfield <version> is the version number needed for proxy announcements to detect which of the several announcements for the same session is the most recent one. Again its usage is up to the creating tool, but it is again suggested that NTP time stamp can be used. <network type> is actually a textual string which shows the type of the network and “IN” has been defined initially to give the meaning of internet. The next <address type> textual string shows the type of the network that has been followed e.g. “IP4”. <address> is the globally identified unique address of the machine from which the session was created [7]. It must be noted that local IP address should not be used in any context related to SDP.

   In general, the “o=” field serves as a globally unique identifier for this version of session description, and the subfields excepting the <version> taken together identify the session irrespective of any modifications [7].

3. **Session Name [s]**
   This required field shows the session name. There should be one and only one “s=” field per session description [7].
4. Connection Data \([c]\]

This required field contains the connection data. There should be a session-level “c=” field and additionally “c=” field per media description can be contained in which case the per-media values override the session-level settings for the relevant media [7]. The subfields contained by this field are:

\[c=<\text{network type}> <\text{address type}> <\text{connection address}>\]

The first sub-field <network type> is the textual string which shows the type of network. Initially “IN” is defined to have the meaning of “Internet” [7]. The next field shows the address type and thus it allows SDP to be used for sessions that are not IP based. Currently only IP4 is defined [7].

The last subfield is the <connection address> and optional extra subfields may be added after the <connection address> depending on the value of the <address type> field [7]. For IP4 unicast address, the connection address contains the fully-qualified domain name or the unicast IP address of the expected data source or data relay or data sink as determined by additional attribute fields [7]. If a unicast data stream is to pass through a network address translator (NAT) [7], the Fully Qualified Domain Name (FQDN) should be used instead of unicast private IP address.

5. Time(s) \([t]\]

This required field shows the time for which the session is active or valid. Its two subfields are:

\[t=<\text{start time}> <\text{stop time}>\]

The first and second sub-fields give the start and stop times for the multimedia session (usually conference) respectively [7]. These values are the decimal representation of NTP time values in seconds [7]. With aggregate control, the server should indicate a stop time value for which it guarantees the description to be valid, and a start time that is equal to or before time at which the DESCRIBE request (of RTSP) was received [7].

If the stop-time is set to zero, then the session is not bounded, though it will not become active until after the start-time. If the start-time is also zero, the session is regarded as permanent [7]. User interfaces should strongly discourage the creation of unbounded and permanent sessions as they give no information about when the session is actually going to terminate, and so make scheduling difficult [7]. The general assumption may be made, when displaying unbounded sessions that have not timed out to the user, that an unbounded session will only be active until half an hour from the current time or the session start time, whichever is the later [7]. If behavior other than this is required, an end-time should be given and modified as appropriate when new information becomes available about when the session should really end [7].

Permanent sessions may be shown to the user as never being active. In general, permanent sessions should not be created for any session expected to have duration of less than 2 months, and should be discouraged for sessions expected to have duration of less than 6 months [7].

6. Attribute(s) \([a]\]

Attribute is an optional field whose primary means is to extend SDP. They can be defined to be used as “session-level” attributes, “media-level” attributes, or both [7].
A “session level” attribute before the media field is applicable to all media specifications rather than individual media. However, media description may have any number of attributes which are media specific and overrides the session level attribute.

Attribute fields are of two forms:

- **Property attributes**: A property attribute is of the form:

  \[ a=<attribute> \]

  These are binary attributes and their presence shows that they are property of session. An example can be “a=recvonly”.

- **Value attributes**: A property attribute is of the form:

  \[ a=<attribute>:<value> \]

  They can be either session-level or media-level attributes. An example can be “a=orient:landscape” to show the landscape orientation of white board.

Attributes that will be commonly used can be registered with IANA [7]. However there can be unregistered attributes and they should begin with “X” to prevent inadvertent collision with registered attributes. However if the receiver receives any attribute that it does not understand, it should simply ignore it.

7. **Media Description [m]**

This is the required field and session description may contain a number of media descriptions [7]. Each media description starts with an “m=” field, and is terminated by either the next m=field or by the end of the session description [7]. The subfields contained by this media field are:

\[ m=<media> <port> <transport> <media format> \]

The first field tells about the media type like “audio”, “video”, “application”, “data” and “control” etc. The second sub-field is the transport port to which the media stream will be sent [7]. The decision for this port depends on the network being used as specified in the relevant “c=” field and on the transport protocol defined in the next third field [7]. It should be noted that the port value should be taken in the range 1024 to 65535 inclusive for UDP based transports. To make it compliant with RTP, this port value should be an even number. The next subfield is the transport protocol whose value depends on the address- type subfield of the “c=” field. If it is selected as “IP4”, then it is normally expected that most media traffic will be carried as RTP over UDP [7]. For RTP media streams operating under the RTP Audio/Video Profile, the protocol field is “RTP/AVP” [7].

The fourth subfield is media format. For audio and video, these will normally be a media payload type as defined in the RTP Audio/Video Profile [7]. For media whose transport protocol is RTP, the SDP can be used to provide a dynamic binding of media encoding to RTP payload type [7]. The encoding names in the RTP AV profile do not specify unique audio encodings (in terms of clock rate and number of audio channels), and so they are not used directly in SDP format fields [7]. So to specify the format for static payload types, the payload type number should be used. For dynamically allocated payload types, the payload type number should be used along with additional encoding information. Normally, it is done by using the “rtpmap” attribute whose general form is:

\[ a=rtpmap:<payload type> <encoding name>/<clock rate>[/<encoding parameters>] \]
2.4. Session Description Protocol (SDP)

For audio streams, <encoding parameters> may specify the number of audio channels. If the number of channels is one provided no additional parameters are needed, then this parameter may be omitted. For video streams, no encoding parameters are currently specified [7].

RTP profiles that specify the use of dynamic payload types must define the set of valid encoding names and/or a means to register encoding names if that profile is to be used with SDP [7]. Experimental encoding formats can also be specified using rtpmap [7]. RTP formats that are not registered as standard format names must be preceded by “X” [7].

For further information on SDP, [7] can be referred.
2. Streaming Protocols
RESTful Mobile Web Services

W3C defines the Web Services (WS) as a software system which is actually designed to support interoperable machine-to-machine interaction in a network [11]. The WS can be considered as Web APIs which are accessed over a network, such as, the Internet and are executed on the remote system which is hosting the services [11].

In the today’s world the mobile devices are acting as multi-functional devices which are capable of providing a broad range of applications and services. These services include WS that are available both for business (commercial) and consumer use. The WS are hosted on mobile nodes using Mobile Web Server (MWS). These hosted mobile WS can be implemented to offer both synchronous and asynchronous execution styles depending upon the requirements of the mobile applications.

Thus, the MWS must provide necessary architectural capabilities to handle and process the incoming requests for each class (synchronous/asynchronous) of service. But it is very important for these MWS to simplify the service access and creation mechanisms such that the processing overhead on the hosting node is reduced. This is because mobile devices have low memory, low battery and processing capabilities as compared to any ordinary computer. The research work done by S.Z.Ali in [11] has supported REST architecture to be used in MWS.

In this chapter, the overview on the REST architecture will be provided. After that, the REST based mobile WS middleware available at ComNets will be presented. This will be followed by the discussion on both synchronous and asynchronous mobile WS architecture that are present in the middleware.

3.1 Representational State Transfer (REST)

REST is a software architecture style for distributed hypermedia systems such as the WWW [11]. REST strictly refers to a collection of network architecture principles which outline how resources are defined and addressed [11]. The system which follows the REST principles are often referred as “RESTful”.

It is important to note that REST is an architectural style, but not a standard. So, the WS can be designed analogous to the client-server architectural style. While REST is not a standard but it does use the following standards [11]:

- HTTP (Transferring between states)
- URL (for Resource addressing)
- XML/HTML/GIF/JPEG/etc (for Resource Representations)

REST middleware architecture developed at ComNets by [11] is simply an Hypertext Transport Protocol (HTTP) based request-response architecture. The requests are usually sent by establishing a mapping between the Uniform Resource Locator (URL) and the HTTP methods. If the
request is mapped to the HTTP POST method, then it includes payload. Usually, this payload may be in any encoding format, such as, XML, JSON etc. When the request is sent by using HTTP GET, then no payload is sent in the request. The response may or may not carry payload. In addition, the REST is light-weight architecture style for applications that do not require security beyond what is available in the HTTP infrastructure, and where the HTTP is the appropriate protocol [11].

3.2 Mobile Web Services using REST

The REST middleware makes the MWS capable of providing RESTful mobile WS. As the work presented in [11], the REST has shown promising optimization results for MWS processing in terms of service access and creation compared to the Simple Object Access Protocol (SOAP). It is shown that the services provided by mobile devices may be short-lived (synchronous) or long-lived (asynchronous) in terms of their execution. Figure 3.1 shows the existing architecture of the RESTful mobile web services middleware, whose behavior is discussed in [11].

![Figure 3.1: Architecture of the RESTful mobile Web Services middleware](image)

### 3.2.1 RESTful Synchronous Web Services

Synchronous services are short-lived services. When they are invoked, the client remains in a blocked state until the execution of the service is completed and the response is received. For these services, there is no mechanism for their control or monitoring at runtime.

As shown in the Figure 3.1, for the RESTful synchronous services, the HTTP Listener receives the request from the Observer (client). This request is then passed to the Request Handler for processing. The Request Handler uses the kXML API to parse the RESTful request, separates the header fields from the XML based payload and identifies the name and type of the target service. The parsed request is then transformed into the Request Object. From this object, the Request Handler identifies the type of the service. Then, in case the service is synchronous, it is forwarded to Deployment Interface. Depending upon the target service, the Deployment interface retrieves the respective service object from the service inventory and invokes the desired service method. Finally, after the completion of the service, the Request Handler sends the result to the Response
3.2. Mobile Web Services using REST

Handler. The Response Handler sends that result as RESTful response message to the Observer.

For more details on RESTful synchronous architecture, the study of [11] is advised.

3.2.2 RESTful Asynchronous Web Services

Basically, the asynchronous services are long-lived services. When they are invoked, client waits for the response but in the separate thread. Thus the client does not need to be in a blocked state and may continue its further processing while service is in execution. Usually, these services need a mechanism for their control and monitoring.

In reference to the Figure 3.1, for asynchronous, the request is transformed to Request Object in the same way as mentioned in the synchronous case. However, when the service type is identified by the Request Handler, it is forwarded to the ASAP Handler. Now, the ASAP Handler determines the intended recipient component (Factory or Instance) of the request. Subsequently, the recipient component performs the requested tasks and sends the response back to ASAP Handler, from where it is forwarded to the Response Handler. The Response Handler finally sends the response as a REST response message to the Observer (client). In addition to that, the Instance may also coordinate with the Request Handler, which then communicate with the Deployment Interface to invoke the web service.

In the following, the mechanisms that are developed for the asynchronous service creation, invocation, monitoring and control of these services are discussed.

3.2.2.1 Service Creation and Invocation

In the service creation, the request from Observer is delegated to the Factory component via the ASAP Handler. The component creates a new Instance for the target services and assigns it a unique Instance End Point Reference (EPR). If the service is requested to be started immediately by the Observer, then the Factory invokes the service instantly which is started in a separate thread. The Factory now creates response that carries the unique EPR of newly created service instance and passes it back to the ASAP Handler. ASAP Handler forwards it to Response Handler from where it is sent to Observer as REST response message. By using the unique enclosed EPR, the Observer may directly access the service instance via the ASAP Handler for services control and monitoring functions.

3.2.2.2 Service Monitoring

In the service monitoring, the Observer inquires Factory or Instance about information, such as, properties, status without disturbing the service execution.

When the service monitoring request is received by the ASAP Handler, it determines whether the request is for the Factory or Instance depending upon the Observer’s demand. Then the respective component processes the request and sends the desired information to the Response Handler via the ASAP Handler, from where, it is sent as a response message back to the Observer.
3.2.2.3  Service Control

In the service control, the request from the Observer is sent to the Instance via the ASAP Handler by using the unique Instance EPR. The corresponding service Instance changes the state of the service to the newly requested state. When the request is forwarded by the ASAP Handler to the specific service instance successfully, it sends the response to the Response Handler, from where, it is sent to the Observer as REST response message.

An asynchronous service can be in one of following states at a time [11]:

- openNotRunning
- opennotRunningSuspended
- openRunning
- closedCompleted
- closedAbnormalCompleted
- closedAbnormalCompletedTerminated

The state of an asynchronous web service may also be changed if any exception occurs during its execution. However, when the change occurs in the state due to any reason (request by the Observer, exception or service completed), a callback notification message is triggered to all of the subscribed Observers in order to notify them about current service state. If the state change occurs due to the service completion, then, the final result is also sent along with the changed state notification.

Network Address Translation (NAT)

With the massive growth of systems over internet, there becomes shortage of Internet Protocol (IP) addresses. To access the one system from the other over the internet usually needs global unique routable IP address. But due to this shortage of IP, it is not feasible to assign each system with its separate unique IP address. Although Internet Protocol version 6 (IPv6) gives great usable address space to overcome the shortage of IP problem in the future, but still most of the existing internet system is still based on Internet Protocol version 4 (IPv4).

Network Address Translation (NAT) has somehow solved this problem of IP shortage. With the induction of NAT, there is no need to assign public accessible unique IP address for every host. Instead, any host on the private network can access the internet by sharing a single public IP address with a number of other hosts on the same private network using the NAT as a multiplexer. The other advantage of the NAT is that it also provides security to the hosts on the private network as no external host residing on the external internet can access the internal hosts directly.

In this chapter, the brief overview on NAT will be given. Then different types of NAT and the methods to find its existence and types of the NAT will be presented. Along with that, different techniques for the traversal of NAT to enable the P2P communication will also be discussed.

4.1 NAT Terminology

Of particular importance is the notion of session, a session endpoint for TCP or UDP is an (IP address, port number) pair, and a particular session is uniquely identified by its two session endpoints [1]. So with reference to one of the host involved, a session is identified by the 4-tuple (local IP address, local port, remote IP address, remote port). The direction of session is normally determined by the flow direction of the first packet that initiates the session. In case for TCP, it is the initial SYNC packet while for UDP, it is the first user datagram.

Although NAT has various flavors, but the most common flavor (type) is traditional or outbound NAT. This type of NAT provides an asymmetric bridge between a private network and a public network because by default, it allows only outbound sessions to traverse the NAT. So all the incoming packets will be dropped unless the NAT identifies them as being part of an existing session that has been initiated from within the private network.

Outbound NAT has two sub-varieties [1]:

- **Basic NAT:**
  In this type of NAT, there is only translation from the private IP address in the IP header to public IP address. It does not involve port translation or mapping in the TCP/UDP header.

- **Network Address Port Translation (NAPT)/ PAT:**
  In this type of NAT, there is translation of both private IP address in the IP header and private port number in the TCP/UDP header to public IP address and public port number. In other words, NAPT translates the entire session end points [1]. Usually NAPT is the one which is
most commonly used because it enables the multiple hosts on the private network to share the single public IP address. Thus it helps out to overcome the IP shortage problem.

The operation of NAT\(^1\) can be seen in the Figure 4.1:

![Figure 4.1: NA(P)/T Operation](image)

Here NAT (e.g. configured with router) on the outgoing connection, translates the private source IP address 10.1.2.1 in the IP header to the public address assigned to it i.e. 88.3.4.3. Then it also translates the private source port 9090 in the TCP/UDP header to any public port number e.g. 65000 which is selected from the pool of public IP address configured with this NAT. Then the packet can be sent to any external host on the internet with the public IP address 210.1.1.2 and port 31000. Then the external host can send response back to the NAT address from where it is mapped back again to the private address.

4.2 NAT Traversal for P2P operation

Although NAT has somehow overcome the shortage problem of IPv4 addresses and provide security to internal hosts, but on the other hand it has caused hurdles for the Peer-to-Peer (P2P) communication between any hosts if one of them lies behind the NAT, unless the NAT is fully configured for the P2P explicitly. This is because NAT has no consistent permanent usable ports to which incoming TCP or UDP connections from the outside external host can be directed.

In addition to NAT, Firewall functionality is typically (but not always) bundled with NAT. These firewalls cause the similar problem because firewalls are generally designed as one way filters. So the sessions which are initiated inside the protected network to any host in the public network are allowed. However any sessions which are initiated from the external host on the internet to the host behind the firewall are blocked.

So there is a need of suitable techniques to traverse these NATs and firewalls in order to provide P2P functionality which are required by the applications e.g. Video Conferencing, Voice over IP (VoIP) and multiplayer online gaming etc.

4.2.1 Simple Traversal of UDP through NATs (STUN)

STUN is a light weight and simple client-server protocol. It allows the application to discover the presence of NAT (and types of NATs) and firewalls between them and the public internet [5].

\(^1\)From now onward, by saying NAT, we mean to say NAPT which involves both IP address & Port translation
addition to this, it can also help the applications to discover the public IP address and public port bindings done by NA(P)T.

### 4.2.1.1 STUN Configuration

STUN configuration mainly consists of two nodes as shown in the Figure 4.2:

![STUN Configuration](image)

**Figure 4.2:** STUN Configuration

**STUN Client** It is any network entity on Private Network (behind NA(P)T/Firewall). It generates the STUN binding request in order to know the public IP address and public port which is mapped by the NA(P)T.

**STUN Server** It is any network entity which is generally attached to Public Internet. It receives the STUN Binding Requests and sends STUN Responses which contains the information about public IP address and public mapped port of the NA(P)T behind which the STUN client lies.

### 4.2.1.2 Discovery of NAT

As it has been said earlier, STUN helps to find the discovery of the NA(P)T. Consider the Figure 4.2 again. Here STUN Server is attached to the public internet and has global accessible IP address (e.g. X) and public port (e.g. A). Now if the STUN Client wants to check whether it is behind any NAT, it will send the STUN binding request to the STUN Server at its public IP address and port. When the binding request reaches the NA(P)T, it will exchange the private IP address and port of client with its own public IP address and port and forwards the request towards the server. When the request reaches at the STUN server, it will check out the source IP address and port (which is actually the IP address and port of the last NA(P)T which is closest to the public internet, in case client is behind multiple levels of NAT). It will then copy that source IP address and port in the payload of the STUN binding response and will send the response back to the client. When the response is reached at the client, it will check the IP address and port contained in the payload with its own private IP address and port. If they are different, then client is behind NA(P)T otherwise if they are same, there is no NA(P)T which exists between the STUN client and the public internet.
4.2.1.3 Hole Punching

In the above Figure 4.2, it has been seen that STUN client sends the binding request and receive the binding response from the STUN Server in order to get to know about the public IP address of the possible NA(P)T and the transport level public mapped port number. But the question may arise if the client is behind the NA(P)T and possible firewall, then what is the thing which is causing the binding request to be sent to the publicly accessed STUN Server and how the response is received at the client behind the NA(P)T/Firewall gateway. It is actually because of Hole Punching as shown in Figure 4.3:

![Figure 4.3: UDP Hole Punching](image)

When the request is sent from the Client to the Server, a hole is punched inside the NA(P)T/Firewall on outgoing UDP connection for particular public destination IP address (X) and public destination port(A). Finally the request is forwarded up to the server. When the response is sent from the STUN server back to the Client on incoming UDP connection, the source IP address and source port will be checked at the punched hole. As the hole was punched against this specific request, i.e. request was initiated from inside the NA(P)T/firewall, so response will be received at the client successfully.

This technique of Hole Punching is widely used for UDP based applications. But essentially the same technique also work for TCP as well. As long as NATs involved meet certain behavioral requirements, hole punching works consistently and robustly [1]. Contrary to what its name may suggest, hole punching does not compromise the security of private network [1]. Instead, hole punching enables applications to function within the default security policy of most NATs, effectively signaling to NATs on the path that communication sessions are “solicited” and thus should be accepted [1].

4.2.1.4 Types of NAT

Although there are many types of the NAT, but based on the NAT implementation for treatment of UDP, STUN has classified NATs into four major types. Three of them belong to the Cone type NAT while the fourth one is Symmetric NAT which is different from others. Description of each type is mentioned as follows:
1. **Full Cone NAT:**

   *Full Cone* NAT is that where all the requests sent from the same internal IP address (e.g. C1) and same internal port (e.g. Z) are mapped to the same external IP address (e.g. N1) and same external port (e.g. X). Furthermore, any external host can send the response to the internal host by sending a packet to the mapped external IP address and port. Full Cone NAT is shown in Figure 4.4:

   ![Figure 4.4: Full Cone NAT](image)

2. **Restricted Cone NAT:**

   *Restricted Cone* NAT is again that where all the requests sent from the same internal IP address (e.g. C1) and same internal port (e.g. Z) are mapped to the same external IP address (e.g. N1) and same external port (e.g. X). But unlike the Full Cone NAT, only particular external host (e.g. Server 1) can send the response (by sending a packet to the mapped external IP address and port) to the internal host if the internal host had previously sent request on the IP address of that external host (i.e. S1). Restricted Cone NAT is shown in Figure 4.5:

   ![Figure 4.5: Restricted Cone NAT](image)

3. **Port Restricted Cone NAT:**

   *Port Restricted Cone* NAT is just like that of Restricted Cone NAT, but this NAT includes both the restriction of IP address and port number. So only particular external host (e.g. Server 1) can send the response from its particular port (e.g. A) to the internal host if the internal host had previously sent request on the IP address of that external host (i.e. S1) and its particular port A. Port Restricted Cone NAT is shown in Figure 4.6:

   ![Figure 4.6: Port Restricted Cone NAT](image)

4. **Symmetric NAT:**

   *Symmetric* NAT is different from the above mentioned three Cone Type NATs. In Symmetric NAT, any request which is sent from the same internal IP address (e.g. C1) and same internal
Network Address Translation (NAT)

Figure 4.6: Port Restricted Cone NAT

port (e.g. Z) to IP address of any particular external host (e.g. S1) at its particular port (e.g. A) will be mapped on a unique external IP address (e.g. N1) and on a unique external port (e.g. X). If another request will be sent from the same internal IP address and internal port to a different external IP address or external port (e.g. B), it will be mapped on a different public port (e.g. Y) at the NAT (or even different public IP address, in case of multiple NATs). Furthermore, only that particular external host (e.g. Server 1) with the IP address S1 can send the response from its particular port (e.g. A) to the internal host by sending the packet to that particular mapped external IP address (e.g. N1) and external port (e.g. X) from where it has received the request on its particular port number A. Symmetric NAT is shown in Figure 4.7:

Figure 4.7: Symmetric NAT

Symmetric NATs are used when high security in communication is required and usually this NAT is installed as routers in business enterprises. Also it is used for home purposes as high end routers.

4.2.1.5 Discovery of NAT types

As discussed above, STUN has categorized NAT implementation depending on their treatment on UDP. STUN also proposes some guidelines for the tests in order to identify the different types of NAT.

Test # 1: Cone vs Symmetric NAT: In the first test, it is checked that whether we have Cone type of NAT or Symmetric. To explain this test, lets refer to the Figure 4.8
4.2. NAT Traversal for P2P operation

Figure 4.8: Test # 1: Cone vs Symmetric NAT

In this test, the first UDP datagram based request is sent to port X of Server # 1 by some client (which is behind NAT). Its private port is mapped at some public port A and the response is received at the client via external port A. Then another UDP datagram request is sent by the client to port Y of Server # 2 which is mapped at external port B and its response is received at the client via external port B. Now if these mapped external ports A and B are not same, this shows that client is behind Symmetric NAT. But if they are same, this shows that client is behind Cone NAT.

If the identified NAT is of Cone type, still it is not sure whether we are behind Full Cone NAT or Restricted Cone NAT. So for that, another test is needed to conduct.

Test # 2: Full Cone vs Restricted Cone NAT: In this second test, it is checked that whether we have Full Cone NAT or Restricted Cone NAT. To explain this test, lets refer to the Figure 4.9

Figure 4.9: Test # 2: Full Cone vs Restricted Cone NAT

In this test, any client behind NAT will send the UDP request to port X of Server # 1. This request will be mapped at some external port A at NAT. Now Server # 1 will not send its response, rather it will forward the information to some other external Server # 2 about the mapped public IP address and port A of the NAT behind which the client resides. Now Server # 2 will send the response of previously sent request from its port Y on behalf of Server # 1 towards the public IP address and public mapped port A of NAT. If this response is received successfully at the client, this shows that client is behind Full Cone NAT. But if the response is not received at the client, this shows that client is behind Restricted Cone NAT.

If the identified NAT is of Restricted Cone, again still it is not sure whether we are behind simple Restricted Cone NAT only or we have Port Restricted Cone NAT. So for that, third test is needed to conduct.
Test # 3: Restricted Cone vs Port Restricted Cone NAT: In this third test, it is checked that whether we have simple Restricted Cone NAT or Port Restricted Cone NAT. To explain this test, lets refer to the Figure 4.10

![Figure 4.10: Test # 3: Restricted Cone vs Port Restricted Cone NAT](image)

In this test, any client behind NAT will send the UDP request to port X of Server # 1. This request will be mapped at some external port A at NAT. Now Server # 1 will not send the response from the same port X, rather it will use some different port Y for sending the response towards client. If the response is successfully received at the client, this shows that it is behind simple Restricted Cone NAT. But if the response is not received at the client, this shows that client is behind Port Restricted Cone NAT.

4.2.1.6 STUN Reservations

Although STUN has provided significant method to traverse most of the NATs which are existing today, but still it has reservations due to which it cannot be considered as generalized method which will work to traverse every type of NAT. Some main reservations concerning to P2P operation are mentioned below:

- STUN does not address TCP communications (either incoming or outgoing)
- STUN works mostly well for Cone Type NATs only
- STUN does not give usable address (i.e. does not work) for Symmetric NATs
- As the STUN server is in public domain, so can have Security Threats (as any body on the public internet can try to access it)

For more reservations and information about STUN, [5] can be referred.

4.2.1.7 P2P UDP sessions establishment using STUN

STUN provides the method for any two clients (which may be behind NATs) to establish UDP based P2P session between them using UDP Hole Punching technique. Suppose Client 1 is behind NAT which wants to establish P2P session with another Client 2 (which may also be behind NAT), then Client 1 will send the request UDP datagram towards the STUN Server by using the UDP Hole Punching. STUN Server will then send UDP response to Client 1 which contains Public and Private endpoints (IP address and port). Similarly STUN Server will send the UDP request towards Client 2 containing the Public and Private endpoints of Client 1. Then both
Client 1 and Client 2 can attempt to make the P2P UDP based session with each other by sending UDP datagrams to both Public and Private endpoints of each other until either Public or Private endpoint “locks in” the valid response from other.

The main concept of establishing P2P session between two clients is somehow similar to Connection Reversal. According to this concept, the two clients (one of which can be behind NAT) first makes the connection with the rendezvous STUN Server. Then if these two clients wants to establish session with each other, this STUN server will relay the request of one client to the other by telling its Public (and sometimes also Private) end points. Then the second client can make an attempt to make a “reverse connection” back to first client by sending the direct UDP request towards the “Public” (or Private) end points of first client.

But it should be noted that above method of making P2P session becomes only possible if both NATs of Client 1 and Client 2 exhibit the property of Consistent End Point Translation (i.e. private endpoints are always mapped to the same public endpoints for all outgoing UDP sessions). This property is present only in Cone type NATs but not in that of Symmetric NATs. So it is difficult or sometimes almost near to impossible to have the DIRECT P2P sessions with each other with the procedure described above.

The basic three scenarios by which UDP based P2P session is established using STUN are discussed as follows:

Peers behind a Common NAT In this simple scenario, the two clients that are wishing to have direct P2P sessions (probably unknowingly) are behind the same common NAT. To discuss this scenario, lets refer to Figure 4.11

![Diagram of Peer-to-Peer Communication Across Network Address Translators](http://www.brynosaurus.com/pub/net/p2pnat/)

**Figure 4.11:** P2P UDP Session for Peers behind a Common NAT [1]

In this figure, Client A and Client B are behind the common NAT with the private end points 10.0.0.1:4321 and 10.1.1.3:4321 respectively. Both these clients have their UDP sessions with the public STUN Server S with 18.181.0.31:1234 as its public end points. The clients have their public mappings at NAT as 155.99.25.11:62000 and 155.99.25.11:62005 respectively. Now when the two clients A and B want to have P2P session, Server S will forward the public and private end points of clients to each other. Now client A will send the UDP request messages directly both to public endpoints 155.99.25.11:62005 and private end points 10.1.1.3:4321 of client B. Similarly client B will send the UDP request messages directly both to public endpoints 155.99.25.11:62000 and private end points 10.0.0.1:4321 of client A. Now the messages directed towards the public end points will reach to destination...
clients only if the NAT supports the hair pin or loopback translation otherwise they will not pass as loop back packets. When the request datagram from client A reaches the NAT, NAT will check its destination address 155.99.25.11:62005 and comes to know that the destination address is one of its own translated public end points. So if the “hair pin” or “loopback” translation is allowed, NAT will translate the source end points of datagram as 155.99.25.11:62005 and destination endpoints as 10.1.1.3:4321 and loop back the datagram towards client B. Similarly, it will be done for the client A for the datagram which has been sent by client B on the public end points of client A.

Now whether the datagrams sent over “public” end points may or may not reach the respective clients. But the datagram which was directed by client A towards the private end point 10.1.1.3:4321 will do reach client B. Similarly the datagram from client B sent to the private end points 10.0.0.1:4321 will reach client A respectively. Thus the direct route is established over the private network between the two clients which is likely to be faster than an indirect route through the NAT.

**Peers behind Different NATs** Now consider the case where the two clients A and B are behind different NATs as shown in Figure 4.12

*Figure 4.12: P2P UDP Session for Peers behind Different NATs [1]*

Here the client A has already initiated its UDP session with the Server S through its respective NAT A which has mapped its public end points at 155.99.25.11:62000 for the connection between client A and Server S. Similarly client B already initiated its separate UDP session with the Server S through its respective NAT B which has mapped its public end points at 138.76.29.7:31000 for the connection between client B and Server S. If the two clients A and B want to have the direct P2P session with each other just like as discussed in the first scenario, server S will provide the two clients with both the public and private end points of one client to the other.

When the client A directs the datagram towards the private end points 10.1.1.3:4321 of client B, it will not reach to B as the client B lies on the separate private network and 10.1.1.3:4321 can not be accessed from outside of that private realm. Similarly is the case for Client B when it will direct the datagram towards the private end points 10.0.0.1:4321 of Client A as it lies in separate private network where 10.0.0.1:4321 cannot be accessed from the outside. So P2P session cannot be established by using the private endpoints.

But when the client A directs the datagram towards the public end point 138.76.29.7:31000 of client B before any datagram sent by client B towards A, it will be dropped at NAT.
**4.2. NAT Traversal for P2P operation**

B because NAT B will consider it as an attempt of incoming UDP connection which has not been initiated from inside of NAT B. However, it will punch a hole inside NAT A for the outgoing UDP session towards the public end point of client B as destination address. If the NAT A exhibit the property of **Consistent End Point Translation** (which is mostly essential for making direct P2P sessions), then this outgoing session will also mapped at the same public end points 155.99.25.11:62000 by NAT A as they were mapped for the session between client A and server S. Now when the client B directs the UDP datagram at the public end point 155.99.25.11:62000 of client A (after the drop out of datagram from Client A at NAT B) it will punch a hole inside NAT B for the public end points of client A as destination address. If the NAT B also exhibits the property of **Consistent End Point Translation**, then this session is also mapped at the same public end points where the previous UDP session between client B and server S was mapped by NAT B. When this datagram reaches NAT A with the source mapping 138.76.29.7:31000, it will consider it as the UDP response of that request against which a hole was punched inside NAT A. It will allow the datagram to pass the NAT A and forwards it to client A. Similarly, any subsequent packet from client A (via NAT A) directed towards the public end points 138.76.29.7:31000 will traverse the NAT B as the hole was already punched in NAT B against the public end pints of client A. After receiving of datagram at NAT B, it will be successfully forwarded to client B. Thus direct UDP based P2P session will be established between two clients A and B which are behind their specific different NATs.

**Peers Behind Multiple Levels of NAT** In this scenario, the two clients are behind multiple levels of NAT as shown in Figure 4.13

![Figure 4.13: P2P UDP Session for Peers Behind Multiple Levels of NAT](http://www.brynosaurus.com/pub/net/p2pnat/)

In this scenario, suppose NAT C is a large industrial NAT which has been deployed by any Internet Service Provider (ISP) and NATs A and B are small customer NAT routers deployed independently by two of the ISP’s customers for private home networks. Only Server S and NAT C have globally public routable IP address. The “public” IP addresses used by NAT A and B are actually private to the ISP’s private realm, while addresses of clients A and B in turn are private to the addressing realms of NAT A and B respectively [1]. Just like the above two cases, clients A and B have initiated their respective UDP sessions with server S via NAT A-C and via NAT B-C respectively.
Now if the clients A and B attempt to establish direct P2P UDP based session between them, one optimal way to achieve that client A can send a message towards the “semi public” end point 10.0.1.2:55000 of client B at NAT B in the ISP’s addressing realm. Similarly client B can send a message towards the “semi public” end point 10.0.1.1:45000 of client A at NAT A. But the problem here is that there is no way for clients A and B to learn these addresses because server S will only see the truly global public end points 155.99.25.11:62000 and 155.99.25.11:62005 of these two clients respectively. Another problem is that, even if these two clients get to know about these “semi public” addresses, there is a high chance that these “semi public” addresses (which are private to ISP’s address realm) may have a conflict with the unrelated address assignments in the client’s private realms. Therefore clients A and B can only use their global public addresses which is seen by the server S namely 155.99.25.11:62000 and 155.99.25.11:62005 respectively. Secondly they have to rely on NAT C to provide the hair pin or loopback translation in order to have P2P session establishment. So when the client A directs UDP datagram towards public end points 155.99.25.11:62005 of client B, first NAT A will translate the datagram source end point from 10.0.0.1:4321 to 10.0.1.1:45000 and then forwards the datagram towards NAT C. When the datagram reaches at NAT C, it will recognize that the datagram’s destination address is one of its own translated public end points. If the NAT C is “hairpin” or “loopback” translation capable, it will translate the source end point as 155.99.25.11:62000 and the destination end point as 10.0.1.2:55000 and “loops” the datagram back onto the private network towards NAT B. When the datagram reaches at NAT B, it will finally translate the destination address as 10.1.1.3:4321 and forwards the datagram towards its private network where it will be received at the client B successfully. The path back to client A from client B will work in the similar way.

From the above discussion of different UDP based P2P session establishment scenarios, it can be noted that two clients cannot establish an “optimal” P2P route between them unless they know the underlying network topology. Secondly, in the multilevel NAT scenarios, it is strongly needed that the higher most NAT which is closed to the public domain should have capability of “hairpin” or “loopback” translation in order to establish P2P sessions successfully. Although, UDP hole punching technique successfully helps out in traversing these multilevel NATs, but still STUN does not provide any generalized method to the higher applications for establishing P2P sessions because these applications do not know the underlying network topology according to which they can adopt the corresponding scenario to make the P2P session attempt successful. Additionally, STUN cannot provide any method to cater Symmetric type of NATs which do not exhibit consistent end point translation property. So still there is strong need for the alternative method which can provide a generalized way for these applications to successfully establish P2P sessions by traversing almost every type of NAT (independent of any specific NAT configuration) without any need to know the underlying network topology.

### 4.2.2 Traversal Using Relay NAT (TURN)

As it has already been discussed that STUN does not help out to traverse the Symmetric NATs. Much research has already been done and still it is in process to find out the most reliable ways by which direct P2P session can be successfully established between the clients which are behind Symmetric NATs. But most of the research which has been done so far like ([13],[14]) for Symmetric NAT traversal is based on the Prediction techniques. The bases for these techniques is that many Symmetric NATs allocate public port numbers for successive sessions in a fairly predictable way [1]. So by exploiting this fact, variants of hole punching algorithms can be developed to
work “most of the time” to traverse Symmetric NATs by first probing the NAT’s behavior using a protocol such as STUN [1]. Then by using the resulting information, the public port number that the NAT will assign to new session will be “predicted”. But it is not necessary that this prediction is correct all the time as the predicted port may be in use by some other client, resulting in failure of P2P session establishment. So the traversal of Symmetric NATs by these Prediction techniques does not represent a robust and long term solution. Secondly, if one prediction technique may become successful to traverse the NAT with specific network topology, it is not guaranteed that it will predict the port successfully when the underlying network topology is changed e.g. in case of Multi-level NATs configuration.

So it is still strongly needed to have such method which can tackle all these miscellaneous issues and provides fundamental basis to traverse the NATs independent of any specific configuration and network topology.

Traversal Using Relay NAT (TURN) is one of those fundamental methods which helps to find out these answers. It is the method which can provide one of generalized approach for the traversal of NATs independent of specific configuration and network topology. Secondly as per discussed by [12], TURN can be solution for the traversal of symmetric NAT as well. According to TURN specification, if a host is behind a NAT and due to certain situations, it is impossible for it to communicate directly with other hosts (peers), then it may need an intermediate node that acts as a communication relay. This relay node typically sits in the public internet and relays packets between two hosts that both sit behind NATs [10]. TURN allows the host to control the operation of the relay and to exchange packets with its peers using the relay [10].

4.2.2.1 TURN Configuration and Operational Overview

Typical TURN configuration is shown in Figure 4.14

![Figure 4.14: TURN Configuration](image)

In the above figure, a host (TURN Client) in the private network is connected through one or more NATs to the public internet. Then there exists another host on the public internet (TURN Server). Over the internet, there are one or more hosts (peers) with which client may wish to communicate. These peers may or may not be behind NATs. When the TURN Client wants to send packets to
4. Network Address Translation (NAT)

these peers or receive packets from these peers, it will use TURN server as relay for sending or receiving of packets.

To establish communication with either of peers, TURN client will send the TURN Allocation Request packet from its private Transport address (combination of IP address and port) i.e. 10.1.1.2:49721 towards the TURN Server Transport address 134.226.2.1:3478. This Server Transport address can be supposed to be already known by the TURN Client e.g. by “preconfiguration”. As the client is behind NAT, so the server will see as those packets are coming from the transport address on the NAT itself. This address is Client’s Server-Reflexive Transport address i.e. 88.0.2.1:7000. The TURN Allocation request which was sent by the client is actually request for the allocation of Relayed Transport Address so that if TURN Client wants to receive any packet from specific peer, that peer can send packet over that Relayed Transport address from where TURN Server will relay that packet to client. Similarly, if TURN Client wants to send packet to any peer, then it can send packet to the TURN Server from where they will be relayed to the respective peer from that Relayed Transport Address. It should be noted that each allocation belongs to a single TURN Client and has exactly one relayed transport address that is used only by that allocation [10].

Now when the allocation request has been sent by the client, Relayed Transport address 134.226.2.1:50000 has been allocated by the server. Now if the Peer A wants to send application data towards TURN Client, it will send that data packet with its Host Private Address 192.168.100.2:49582. When it reaches NAT, it will translate the source address from Host Private Address to its corresponding Server-Reflexive Transport address 89.0.1.1:32102 and the data packet will be sent to Relayed Transport Address 134.226.2.1:50000 of the TURN Server. As this was allocated for the TURN Client against the specific Client’s Server-Reflexive Transport Address 88.0.2.1:7000, so TURN Server will relay that application data packet towards that Reflexive Transport address (which is actually the public NAT address of TURN Client) from where it is forwarded to the TURN Client successfully. Similarly, if the TURN Client sends some application data towards Peer A, it will be sent towards TURN Server Transport Address 134.226.2.1:3478 from where TURN Server will relay that application data towards the Peer A Server-Reflexive Transport Address with the Relayed Transport Address 134.226.2.1:50000 as source address. When the data packet reaches at NAT of Peer A, it will be forwarded towards Peer A Host Transport Address 192.168.100.2:49582 and thus Peer A successfully receive it.

TURN can allow both TCP or UDP between TURN Client and TURN Server and between TURN Server and Peers. However once the Relayed Transport Address is allocated for TURN Client over TCP or UDP, a client must keep the allocation alive. To do this, client periodically sends a Refresh request (keep-alives) to the TURN Server.

Both the TURN Client and TURN Server keep track of value known as 5-tuple used in the Allocate request of Relayed Transport address. At the client, the 5-tuple consists of client’s host transport address, the server transport address and the transport protocol (UDP or TCP) used by the TURN Client to communicate with the TURN Server [10]. At the TURN Server, 5-tuple value is the same except that the Client’s Host Transport Address is replaced by Client’s Server-Reflexive address since that is the client’s address as seen by the server [10]. If the client wishes to have another second Relayed Transport Address at the TURN Server, it must create a second allocation using a different 5-tuple (e.g. by using a different client host address or port) [10].

For more information about TURN, [10] can be referred.
4.2.3 Advantages of TURN over STUN

- Can work both for TCP and UDP connections
- Provides the generalized NAT/Firewall traversal technique independent of any specific NAT configuration.
- Does not require the knowledge about underlying Network Topology for NAT traversal.

4.2.4 Overheads using TURN

- Additional resource in the form of STUN Server is required to have the connection established
- As the data is relayed through the STUN Server
  - Added latency in the communication
  - Data is not secure over the transmission path
  - Needs high bandwidth connection which comes with additional cost
  - Introduced additional point of failure in the network
4. Network Address Translation (NAT)
Mobile-to-Mobile Streaming using Mobile Web Server

This chapter mainly focuses on the Implementation of Mobile-to-Mobile (M2M) multimedia streaming using MWS. The main idea of this multimedia streaming is that as in any multimedia server, audio (or video) file on a mobile node is streamed packet by packet over the UDP to the other mobile node, who is an M2M client. When sufficient multimedia content is buffered at the client mobile, it will start its playback in parallel for the buffering of next packets. This process will continue until the complete file on the server mobile is successfully buffered (and then played back) at the client or any exception occurs during this transmission.

Through the RTSP standard, the client node is provided a functionality that is similar to remote control, by which it can START, PAUSE, RESUME and STOP the streaming of these multimedia data packets on the server mobile node.

In this chapter, all the design and implementation details are discussed by which the RTSP based controlled multimedia streaming model have been developed in this thesis. The implementation of RTP frame structure and RTSP protocol over REST architecture have been presented with all the necessary details.

5.1 General Implementation Architecture for M2M Streaming

The basic architecture for the implementation of M2M multimedia streaming from source mobile node (Server) to the recipient mobile node (Client) is shown in Figure 5.1.

According to this architecture, the RTSP based signaling to provide the remote control functionality like DESCRIBE, PLAY, PAUSE and TEARDOWN is implemented by establishing TCP based socket connections between the server and the client mobile nodes.

For the transmission of data packets containing the media content from the Server to Client, two Datagram Sockets will be created. One datagram socket will be created at the Server for sending of the datagrams containing the media data while other socket will be created at the client for receiving of the datagrams.

The main idea behind making of datagrams with media data is that the original media bytes will be read from the Audio/Video (A/V) source at the Server. Then these media content bytes are added as RTP Payload along with the RTP Header inside the RTP Packet. Then this RTP Packet will be encapsulated inside the UDP datagram. Finally, this UDP datagram will be sent by the Datagram Socket at the server. At the client, this UDP datagram will be received by its Datagram Socket. After that, the packet will be decapsulated from the UDP datagram. The media content is then separated as the RTP Payload from the RTP Header and finally the bytes of the media content will be available to the client. These bytes can now be played down using any Media Player functionality available at the client.

---

1Google Android smart phone in our case
2From now onward, Server means source mobile node
3Client means Recipient mobile node
5. Mobile-to-Mobile Streaming using Mobile Web Server

Figure 5.1: Implementation Architecture for M2M Streaming

5.2 RTP Implementation

RTP has already been discussed in detail in Section 2.2 and the generalized RTP frame structure has been shown in Figure 2.3. In this thesis, RTP frame structure has been implemented which can provide Application Programming Interface (API) for the multimedia streaming applications.

5.2.1 Implemented RTP Frame Structure

In this thesis, only Audio On Demand service has been provided between two mobile nodes. So some fields that were present in the generalized RTP frame structure shown in figure 2.3 may not be needed. The RTP Header fields that are required and those which can be excluded from the generalized frame structure are shown in Figure 5.2.

From the above Figure 5.2, the first header field that has been excluded is Contributing Source (CSRC) identifiers. This is because, only the “Audio On Demand” service has been implemented that is based only on single source at the server. So there is not any additional Contributing Sources for this on-demand service other than the single original audio source.

Second header field that has been excluded is the Extension Header. This is because for such audio based M2M streaming application, there is no extra information required that has to be transported with the help of Header Extension. The fields in the basic Header are enough to carry out the implemented streaming successfully.

The last field which is actually not the part of RTP Header is “padding”, which are the extra bytes that are added at the end of payload to keep the packet size constant for any encoding/decoding purposes. So far in this thesis, the original basic format of the data (which has been read from the source) has not been changed and there is no encoding or decoding involved. Based on that, the implemented prototype can be kept generalized, so that it is easily extensible for more multimedia
5.2. RTP Implementation

applications with different source formats in the future. Secondly, as this application has been developed on Google’s Android, which provides this functionality that it can separate out the original data bytes from the variable payload at the client. So, from the application point of view, no padding bits has been added explicitly and the resultant payload may still be of variable sizes.

Thus, the finalized RTP frame structure with 12 bytes of RTP fixed header that has been implemented in this thesis after the exclusion of unwanted fields is shown in Figure 5.3.

Now the way in which each header field has been implemented is described as follows.

- **Version (V)**
  First is the 2 bit Version field and has value “2”, as it is the newest RTP version available.

- **Padding (P)**
  Second 1 bit field is the padding. As this has already been discussed above that our implemented architecture will not put any padding at the end of payload, so, this bit has a value “0”.

figure 5.2: Excluded Fields from Generalized RTP Frame

figure 5.3: Implemented RTP Frame Structure
5. Mobile-to-Mobile Streaming using Mobile Web Server

- **(Header) Extension (X)**
  Third 1 bit field is Extension. As discussed above, our implemented architecture will not carry any Extension Header so this bit also contains a “0”.

- **CSRC count (CC)**
  Next 4 bit field is of CSRC count or CSRC List i.e. Contributing Source count. As discussed above, we have only one original single source at the server, so CSRC List does not exist and CC field carries a “0” value.

- **Marker (M)**
  Next 1 bit field is of Marker. As its implementation is based on application, so in our application, it actually helps out to identify the end of the streaming data i.e. when the last streaming packet is received, its Marker bit contains value “1”. For the remaining packets, its value is “0”.

- **Payload Type (PT)**
  This 7 bit field can have any value of Dynamic Payload Type that ranges from 96-127. This is because our source data will mostly be of Experimental format and will not come in the category of static payload types.

- **Sequence number**
  This field will contain 16 bits value showing the sequence number of the packets. Its starting value can be any random number (of 16 bit) in the first packet which will be then linearly incremented in the following packets.

- **Timestamp**
  This field contains 32 bits value. Its starting value is also taken as any 32 bit random value in the first packet which will be incremented linearly in the following packets with the offset equal to Sampling Period (which is actually equal to Transmission Delay of packet in our case).

- **SSRC Identifier**
  Next 32 bit field is the Synchronization Source Identifier. Its value is also taken as any 32 bit random value which remains the same in all packets as we have only single original multimedia source with us. Note that in our case, SSRC cannot be a Mixer since we do not have any additional contributing sources with us other than single original source.

### 5.2.2 Maximum RTP Payload Size

One of the motivations for this thesis is to develop a generalized prototype, which can be accommodated both for WLAN and for cellular operator’s WWAN e.g. EDGE. From our previous knowledge, we know that the Maximum Transmission Unit (MTU)

4MTU is determined by Layer 2 and is the maximum size of packet that can be transmitted over the network path without being fragmented
5.2. RTP Implementation

Figure 5.4: RTP Maximum Payload size

\[ \text{Header Size} = 12 \text{ (RTP)} + 8 \text{ (UDP)} + 20 \text{ (IP)} = 40 \text{ Bytes} \]
\[ \text{RTP maximum Payload Length} = 1500 \text{ (EDGE MTU)} - 40 = 1460 \text{ Bytes} \]

Thus the maximum size of the data that can be added in RTP packet as RTP payload is of \textit{1460 bytes}.

5.2.3 Implemented classes

The existing ComNets middleware has been extended by adding \texttt{cnMultiMedia} package to provide the RTP packetization API both at the server and the client. For that, two classes namely \texttt{RTPpacket} and \texttt{ReadSourceFile} has been implemented. \texttt{ReadSourceFile} actually reads the media file (of any format) present at the source as \textit{chunk-by-chunk} and gives it to the \texttt{RTPpacket} class. Then, the \texttt{RTPpacket} makes the corresponding RTP packets by adding that media chunks (provided by \texttt{ReadSourceFile} class) as RTP payloads at the server by setting it along with RTP Header fields. On the client side, \texttt{RTPpacket} also provides the functionality to separate out the RTP payload from RTP Header from the received RTP packet. Also, it picks out the corresponding RTP Header fields after separating it from the RTP payload.

The snippet of code 5.5 shows the \textit{Mounting} of the SD card. The next snippet 5.6 shows the selection of the media file at the source and setting of \texttt{FileInputStream} for specified file at the given \texttt{path} in \texttt{ReadSourceFile} class. These codes are implemented inside the \texttt{InitializeAll()} function of \texttt{ReadSourceFile} class.

```java
if (android.os.Environment.getExternalStorageState().equals(android.os.Environment.MEDIA_MOUNTED)) {
    Log.i("DatagramServer", "SDCard is mounted");
} else {
    Log.i("DatagramServer", "SDCard is NOT mounted");
}
```

Figure 5.5: Checking for SDCard Mounting

```java
path = new String("/sdcard/" + Folder_Name + "/" + filename);
stream = new FileInputStream(path);
```

Figure 5.6: Selection of File & Setting of FileInputStream

The next snippet 5.7 shows the function \texttt{getnextframe} for reading of the file contents \textit{chunk-by-chunk} as \textit{data frames} by the “input stream” created in above code with the help of built-in \texttt{read}
function, and put the read data inside “frame” byte array. It also returns the number of bytes that has been read by the frame.

```java
public int getnextframe(byte[] frame) throws Exception {
    int numread = 0;
    try {
        numread = stream.read(frame);
    } catch (IOException ex) {
        ....
    }
    return numread;
}
```

**Figure 5.7:** Reading File contents Frame by Frame

Now coming to the **RTPpacket** class, which has two constructors. Code snippet 5.8 shows the first constructor which will be used by server for setting up RTP Header fields and adds the read data from the source as RTP payload for making RTP packet so that it can be later encapsulated inside UDP datagram.

Now, the second RTP constructor which will be used by the client is shown in code snippet 5.9. It will be used for decapsulating the RTP packet from the received UDP datagram, and then, it can separate out the RTP Header and RTP payload inside the RTP packet. The assigned RTP Header fields will also be picked out by this constructor.

In addition to the above constructor, there are also some additional helping functions that have been implemented at the client. These include the functions to get the values of different RTP header fields like “Marker (M)” value by **getMarker** as shown by code snippet 5.10

Other than this, another function has been provided named as **getpayload** as shown in code 5.11 which will take any byte array as input and fill in with the RTP payload contents and also return the size of RTP payload in terms of **number of bytes**.

### 5.2.4 Debugger Snapshots for RTP Header

The snapshots of **Debugger (DDMS)** available in the Google’s Android emulator are shown in figures 5.12 and 5.13. The first figure 5.12 shows the RTP Header fields of the first received RTP Packet. The next figure 5.13 shows the RTP Header fields of the last received RTP Packet.

In these two figures, it can be seen that **V** has value of “2” while **P, X and CC** has “0” value which will remain the same throughout the RTP packets. Similarly, **SSRC Identifier** and **Payload number** is also the same throughout the RTP packets. **Payload Type (PT)** (which is marked with the yellow arrow) has also the same value of “96” showing the Dynamic Payload Type format of the source file. But the **Sequence Number** and **TimeStamp** will be increased linearly in the following RTP packets received at the client.

Now, first notable field between these two figures is **RTP Payload Length** which is highlighted by red rectangular shape. It can be seen that first RTP packet has RTP Payload Length of “1460” bytes which will remain the same in all the following packets except the last one. The last RTP packet that is received at the client has RTP Payload Length **equal to or less than** 1460 bytes as here it has “944” bytes. The second notable field is of **Marker (M)** (which is highlighted by the blue rectangular shape), whose value is “0” in the first received RTP packet and it will be same
5.2. RTP Implementation

```java
public RTPpacket(int PType, int Framenb, int ssrc, int mark, int Time, byte[] data, int data_length) {
    //fill by default header fields:
    Version = 2;
    Padding = 0; // Values will be "0" or "1"
    Extension = 0;// Values will be "0" or "1"
    CSRCcount = 0;// Values can range from "0" to "15" but set it
    // either "0" if no Mixer, otherwise "2" to "15"
    Marker = mark;// Values will be "0" or "1"
    SSRC = ssrc;
    SequenceNumber = Framenb;
    TimeStamp = Time;
    PayloadType = PType;

    //build the header of "12 bytes" and fill in the corresponding fields
    //--------------------------
    header = new byte[HEADER_SIZE];
    .....

    //Adding up the payload after the header
    //--------------------------
    payload_size = data_length;
    payload = new byte[data_length];

    for (int i = 0; i < payload_size; i++) {
        payload[i] = data[i];
    }
}
```

Figure 5.8: RTP Constructor for Server
public RTPpacket(byte[] packet, int packet_size) {

    // check if total packet size is greater than the header size
    // in order to have some valid payload
    if (packet_size >= HEADER_SIZE) {
        // get the RTP header
        header = new byte[HEADER_SIZE];
        for (int i = 0; i < HEADER_SIZE; i++) {
            header[i] = packet[i];
        }

        // separate out the RTP payload
        payload_size = packet_size - HEADER_SIZE;
        payload = new byte[payload_size];
        for (int i = HEADER_SIZE; i < packet_size; i++) {
            payload[i - HEADER_SIZE] = packet[i];
        }

        // interpret the different fields of the header
        PayloadType = header[1] & 127;
        ......
    }
}

Figure 5.9: RTP Constructor for Client

public int getMarker() {
    return (Marker);
}

Figure 5.10: Returning the value contained in Marker (M) field

public int getpayload(byte[] data) {
    for (int i = 0; i < payload_size; i++) {
        data[i] = payload[i];
    }
    return (payload_size);
}

Figure 5.11: Retrieving RTP Payload Contents
5.3 Implementation Scenarios for M2M Streaming

As it has been mentioned before that one part of the motivation of this thesis is to develop a prototype, by which, two mobile nodes can have M2M multimedia streaming functionality, which is available not only for WLAN, but also, for cellular operator’s WWAN networks like EDGE. So, three networks have been investigated and tested to see the possibility for the development of M2M multimedia streaming framework. These networks are:

- T-Mobile EDGE and HSPA networks
- O2 UMTS/HSPA network
- eduroam (WLAN) network at RWTH Aachen University

Now two scenarios have been planned that has to be implemented over these networks to provide the M2M multimedia streaming functionality.

- Scenario I : Direct Mobile-to-Mobile media streaming
  This scenario is simple and straightforward where the streaming is done directly from one mobile to the other.

throughout the following RTP packets except the last one. The last RTP packet has value “1” for this Marker (M) field showing the end of streaming data.
• Scenario II: M2M Media streaming using IAG
  In this scenario, Intermediate Access Gateway will be developed and installed between any two mobile nodes in order to take care for any NAT and firewall related issues and to make the multimedia streaming possible in M2M setting.

5.3.1 Network Investigation and Testing

The above mentioned WLAN and WWAN networks will now be tested to investigate the below mentioned issues:

1. Identify the existence and types of NATs
2. Identify a suitable method of the NAT traversal
3. Identify the influence of NAT on the mobile client and server
4. Identify the applicability of scenario I and scenario II for these networks

For the first issue, we will use STUN technique to identify the existence of any NAT, as mentioned in section 4.2.1.2, and to find the specific types of NAT as per the guidelines proposed in section 4.2.1.5.

For the second issue, we will use UDP Hole Punching for NAT traversal as explained in section 4.2.1.3.

To find out the influence of possible NATs (if exist) on the mobile client and server, the below mentioned test cases are conducted. Here, Test Cases I to IV belong to the scenario I, where the UDP datagrams containing media data are sent directly from server to client, whereas, the Test Cases (V-X) from IV onwards belong to the scenario II, where the UDP datagrams carrying the media data are sent from server to client using the help of IAG either for the NAT traversal to have direct streaming session or to relay UDP datagrams from the server to client if the direct session is not possible.

5.3.1.1 Scenario I: Direct M2M media streaming

Test Case I: Server on the T-Mobile EDGE network (Public IP+ Public Port), Client on the eduroam WLAN (Public IP+ Private Port)

In this case, the mobile node with the media server is connected to T-Mobile EDGE network while the client is connected with the eduroam WLAN, as shown in Figure 5.14.

![Figure 5.14: Scenario I: Server on T-Mobile EDGE, Client on eduroam WLAN](image)

When the IP addresses were checked for both nodes, server was having dynamic IP address of 88.x.x.x series which does not belong to the private address pool. So, this shows that
we have dynamic but public IP address. Then we put the UDP socket at server on the listening state at the fixed public port “9094”. Then, we checked the IP address of the client at the eduroam and it belonged to the 134.x.x.x group which also does not belong to the private IP address. So, this means that our client seems not to be behind any NAT/PAT but still that’s not sure. Now, by considering the mobile server here just like a STUN server, a request UDP packet (shown with the blue colored datagram) is sent from the client from its fixed port “9096”. When the packet is received at the server and the IP address and port numbers from the received UDP datagram were checked, we came to know that the IP address was same as we have seen before sending the packet, but the port was not “9096”, but any random different port e.g. Y. This means that our client at eduroam is not behind any Network Translation gateway, but it can be behind a gateway, which is only doing Port Translation. As we now have both the public IP address 134.x.x.x and mapped public port Y of the client from where the request packet was sent, so from the server as a response, the streaming packets (shown with the green colored datagrams) containing the media data will be sent on the public transport address that has been seen by the STUN server. At the client, these packets are successfully received and the client can play back the buffered packets successfully.

So, this shows that our Scenario I of direct M2M streaming becomes successful in this case. Another thing which is learned from this test case is that our server should have public IP address and public port so that server can be accessed globally.

Test Case II: Server on the T-Mobile EDGE network (Public IP+ Public Port), Client on the O2 UMTS/HSPA network (Private IP+ Private Port)
In this case, the media server mobile is connected on T-Mobile EDGE network while the client is connected with O2 UMTS/HSPA network as shown in Figure 5.15.

Just like the case I, the server has public IP and public port on T-Mobile network. But when we put the client on O2 UMTS/HSPA network and checked its IP address, it lied in the 10.x.x.x group, which belongs to the private IP address pool. This shows that client on O2 network may be behind some possible NAT, but still we are not sure. Again, the server is listening for UDP datagram at the public port “9094”. Then the UDP request packet (shown with the blue colored datagram) is sent by the client from the fixed UDP port “9096”. But when the packet is received at the server and the source transport address (source IP address, source port) of the packet is checked, we came to know that they were different from the private transport address. The source IP address was of 89.x.x.x series which belongs to the public IP address pool and the port was again some random port (lets say Y). This shows that client on O2 UMTS/HSPA network is behind some NAPT gateway. Now, as the public
Mobile-to-Mobile Streaming using Mobile Web Server

IP address and public port numbers of the client has been known to the server, so, as a response when the streaming packets (shown with the green colored datagrams) containing the media data have been sent to the public transport address of the client, they are received successfully at the client and we can play out the buffered packets.

So, this test shows that the Scenario I of direct M2M streaming again become successful here. The other noticeable thing is that our client (which was behind any possible NAPT/Firewall Gateway) successfully traversed the NAPT/Firewall by applying the UDP Hole Punching technique. By this UDP Hole punching, the client has made a UDP tunnel inside its NAPT/Firewall through which it can receive the streaming packets from the server. Thus a direct UDP based P2P session is established between the public server and the private client behind the NAPT/Firewall.

Test Case III: Server on the T-Mobile EDGE network (Public IP+ Public Port), Client on the T-Mobile HSPA network (Private IP+ Private Port)

In this case, the mobile media server is connected to the T-Mobile EDGE network and the client is also connected with T-Mobile but on HSPA network as shown in Figure 5.16.

In this case, just like above mentioned two cases, the server on T-Mobile EDGE network has public IP address of and public port. Now, when the client is connected on T-Mobile HSPA network and we checked its IP address, it was of private series like 10.x.x.x, just like in the O2 network discussed in the case II before. This means that client may be behind the possible NAT but again we have to check for that. So, we send the request packet (shown with the blue colored datagram) from some fixed UDP port “9096” at the client towards the server at its public IP address and port. But the different thing we have observed in this case is that packet does not reach successfully at the server, although, its IP address and port are public. The possible reason for this can be some operator’s UDP restriction, which caused this hindrance.

So, in this case, the Scenario I of direct M2M streaming becomes unsuccessful and we cannot consider it as a generalized solution.

Test Case IV: Server on the O2 UMTS/HSPA network (Private IP+ Private Port), Client on the eduroam WLAN (Public IP+ Private Port)

In this case, the mobile media server is connected to the O2 UMTS/HSPA network and the client is connected to the eduroam WLAN network as shown in Figure 5.17.

In this case, the server is connected with O2 UMTS/HSPA network and from the previous test cases, we came to know that in this case server will have some private IP address like
5.3. Implementation Scenarios for M2M Streaming

5.3.1.2 Scenario II: M2M Media streaming using IAG

In the scenario II, an Intermediate Access Gateway (IAG) will be used for the streaming of multimedia packets from the server to the client. This IAG is simply a machine placed at ComNets with the fixed public IP address 137.226.5.151 and with fixed public ports 9092 and 9093. So, the client (may or may not be behind NAT/Firewall) will make a UDP tunnel with this IAG at the port “9092” and server (which may also be behind NAT/Firewall) will make its UDP tunnel with this IAG at the port “9093”, respectively by using the UDP Hole Punching technique. Now the UDP datagrams (which are carrying the media data) can be sent directly from server to client, or via the IAG, depending on the two further scenarios discussed as follows.

- Scenario IIa: IAG as an Introducer Gateway
  Here IAG will act as an Introducer Gateway (and behaves similar to STUN Server discussed in section 4.2.1) because it will introduce the server with public transport address (by using the concept of Connection Reversal discussed inside section 4.2.1.7) of the client to which it can stream the multimedia packets directly. So, finally the media streaming will be M2M. But this scenario will only work with the special NAPT configuration of consistent port mapping. Although it seems to be a good solution but it is not generalized.
• Scenario IIb: IAG as a Relay Gateway

Here the IAG will act as a Relay Gateway (and behaves similar to TURN Server discussed in section 4.2.2) because it will relay the streaming packets (containing the media data) from the server to the client. So, finally media streaming will be carried out via IAG. This scenario will almost work with every NAPT, independent of any specific configuration. Although, it seems to be less efficient, but it is reliable and one of the most generalized solution.

Now we will again do the live network testing for the Scenarios IIa and IIb.

Test Case V: Server on the T-Mobile EDGE network (Public IP+ Public Port), IAG on CN LAN (Public IP+ Public Port), Client on the eduroam WLAN (Public IP+ Private Port)

In this case, the mobile media server is again connected to the T-Mobile EDGE network while the client is connected with the eduroam WLAN. The IAG will be connected with the ComNets(CN) LAN network as shown in Figure 5.18.

![Figure 5.18: Scenario II: Server on T-Mobile EDGE, IAG on CN LAN, Client on eduroam WLAN](image)

Here, the IAG and the server will act as two STUN servers. This is needed to check out the possible type of NAT (if exists) at the client. So, we will follow the test guidelines as mentioned in section 4.2.1.5 Test # 1. So, first it will be checked whether we have Symmetric or Cone type NAT. From the previous test cases, we know that T-Mobile EDGE network is giving the dynamic but public transport address, while the eduroam WLAN is only doing port translation. So, in this case, the server is configured at the public IP address like 88.x.x.x with the public port “9094” while the client is configured with the dynamic public IP like 134.x.x.x with the fixed private port “9096”. But the IAG is configured with the fixed public IP address 137.226.5.151 and the fixed public UDP ports “9092” and “9093”.

Now, to test the behavior of the Port Translation Gateway at the client, we will send two request packets towards the two (acting) STUN servers i.e. IAG and (media) server. The first request packet will be sent from the fixed port “9096” at the client towards the IAG at its public address 137.226.5.151 and public port 9092. So, IAG will record the public IP address and public port (e.g. Y) from this request packet. The second packet is sent by the client from the same fixed port “9096” to the server at its dynamic public address of type 88.x.x.x and public port “9094”. This server will also record the IP address and port number of the packet that has been received from the client. When the two addresses and port numbers recorded by IAG and media server are compared, we came to know that the both public addresses (and the dynamic ports) were same i.e. the public port Y has consistent values for the two request packets.

So, this test case shows that Port Translation gateway at the eduroam exhibits the property
of Cone NAT in terms of the port mapping i.e. it has consistent port mapping property.

**Test Case VI:** *Server on the O2 UMTS/HSPA network (Private IP+ Private Port), IAG on CN LAN (Public IP+ Public Port), Client on the eduroam WLAN (Public IP+ Private Port)*

In this case, the mobile media server is connected to the O2 UMTS/HSPA network while the client is connected with the eduroam WLAN. The IAG will be connected with the ComNets(CN) LAN network as shown in Figure 5.19.

![Figure 5.19: Scenario IIa: Server on O2 UMTS/HSPA, IAG on CN LAN, Client on eduroam WLAN](image)

From the previous Test Case 5, we came to know that port translation gateway at eduroam is exhibiting the property of Cone NAT in terms of port mapping. Now, we will further test it whether this port mapping property is similar to Full Cone NAT or Restricted Cone NAT for which this test case will be carried out under the guidelines discussed in section 4.2.1.5 Test # 2. From the previous tests, we know that O2 UMTS/HSPA network is behind NAPT/Firewall gateway. So, here the server will have private IP address of type 10.x.x.x and private port “9094”. Now the first request packet *(shown with orange colored datagram)* will be sent by the server from its fixed port “9094” to the IAG at its fixed public transport address 137.226.5.151:9093. It will punch a hole in the server NAPT/Firewall and will make a UDP Tunnel with the IAG. Then, the second request packet *(shown with blue colored datagram)* will be sent by the client from its fixed port “9092” towards the fixed IP address of IAG, but at different port i.e. “9092”. So now the IAG will record the public transport address of the packet received from the client. It will then put this information as payload and send a response packet *(shown with blue colored datagram inside orange colored tunnel)* towards the server via the UDP tunnel that has been established between the server and the IAG. Now, the server knows the public IP address and mapped public port Y of the client. The server will send the streaming packets *(shown with green colored datagrams)* containing the media data to the public transport address that is provided by the IAG. This will punch another hole inside the server’s NAPT/Firewall and server will try to make a direct UDP tunnel with the client. On the client side, the packets will be received successfully which can be played back after buffering of the packets.

So, this test case shows eduroam port translation gateway actually exhibits the property of Full Cone NAT in terms of the port mapping. The UDP tunnel is established successfully between the server and client. Here the IAG acts as an Introducer gateway because it has introduced the public IP address and public mapped UDP port of the client with the server, which is behind NAPT/Firewall and cannot be accessed directly. Thus, the multimedia streaming
is finally direct M2M, and thus the Scenario IIa becomes successful here.

**Test Case VII:** Server on the T-Mobile **HSPA** network (Private IP+ Private Port), IAG on CN LAN (Public IP+ Public Port), Client on the eduroam **WLAN** (Public IP+ Private Port)

In this case, the media server is connected with the T-Mobile HSPA network, while the client is again connected with the eduroam WLAN. The IAG will be connected with the ComNets (CN) LAN network, as shown in Figure 5.20.

This test case is similar to the Test Case VI conducted previously. The difference is that here media server will be on T-Mobile HSPA network and from the previous tests, we know that this T-Mobile HSPA network will be behind NAPT/Firewall due to which the server will have private transport address just like to that of server in case of the O2 UMTS/HSPA network discussed in Test Case VI. Remaining the whole testing is same as done in Test Case VI before and we have got the similar result.

So Scenario IIa again becomes successful.

**Test Case VIII:** Server on the T-Mobile **EDGE** network (Public IP+ Public Port), IAG on CN LAN (Public IP+ Public Port), Client on O2 UMTS/HSPA network (Private IP+ Private Port)

In this case, the media server is connected to the T-Mobile EDGE network while the client is connected with the O2 UMTS/HSPA network. The IAG will be connected with the ComNets (CN) LAN network as shown in Figure 5.21.

This test is conducted in the same way as the Test Case VI to identify the type of the NAT (if exists) behind which the client resides. So, here again the IAG and the media server will act like two STUN Servers and we will do the testing as done in Test Case VI for the client on the eduroam WLAN under the guidelines of Test # 1 in section 4.2.1.5. From the previous tests, we know that the O2 UMTS/HSPA network has NAPT/Firewall, so the client will have the private IP address of type 10.x.x.x and private fixed port number “9096”. Now, again the first request packet *(shown with the blue colored datagram)* will be sent to the public transport address of the IAG i.e. 137.226.5.151:9092 by the client. When the packet reaches the IAG, it will record the public mapped IP address and port from the received datagram. Now again the second request packet *(shown with the green colored datagram)* will be sent to the media server at its dynamic but public IP address like 88.x.x.x and the fixed public port “9094” which will also record the public mapped transport address against this
received packet. Now, when the two transport addresses recorded by the two STUN servers are compared, we have seen that the mapped public IP addresses like 89.x.x.x were same for both packets. But, the two dynamic public mapped ports were not same by information received from the IAG, it was some port Z for its received packet, and from the media server it was some other different mapped port like Y.

So, this test shows that NAPT/Firewall behind which the client resides does not exhibit the property of consistent port mapping and is of Symmetric Type NAT. Thus the Scenario IIa (which works only under consistent port mapping facility) seems to be unsuccessful. So the other Scenario IIb will be needed here.

**Test Case IX: Server on the T-Mobile HSPA network (Private IP+ Private Port), IAG on CN LAN (Public IP+ Public Port), Client on O2 UMTS/HSPA network (Private IP+ Private Port)**

In this case, the media server is connected to the T-Mobile HSPA network, while the client is connected with the O2 UMTS/HSPA network. The IAG will be connected with the ComNets (CN) LAN network as shown in Figure 5.22.

**Figure 5.22: Scenario IIb: Server on T-Mobile HSPA, IAG on CN LAN, Client on O2 UMTS/HSPA**

Now in this case, the client is connected to the O2 UMTS/HSPA network and from the Test Case VI, we came to know that the client will be behind Symmetric NAPT and firewall. The
media server is on the T-Mobile HSPA network and based on the previous test cases, it also shows that server will be behind NAPT/Firewall. So, in this case both the media server and clients are behind the NAPT/Firewalls. So here the IAG will be used as a relay node to make the multimedia streaming possible between the server and client.

In this case, the client will send the request packet (shown with orange colored datagram) to the IAG at its fixed transport address i.e. 137.226.5.151:9092. Similarly, the server will also send the request packet to the IAG with the same IP address 137.226.5.151 but at different port “9093”. These two packets will punch the holes inside the NAPT/Firewall at the client and server, respectively. Then, the two UDP tunnels will be established by both client and server with the IAG at their respective ports. Now the media server will send the streaming packet (shown with green colored datagram) containing the media content towards the IAG over the UDP tunnel which was established between the IAG (at its fixed port 9093) and the server (at its public mapped port A). When the streaming packet is received by IAG, it will simply relay that streaming packet towards the client over the tunnel which was established between IAG (at its other fixed port 9092) and the client (at its public mapped port B). The streaming packet will be successfully received by the client and then played back at the client.

So, this test case shows that IAG acts as a Relay gateway here. The streaming of multimedia packets between the server and the client is done via IAG. Thus, scenario IIb becomes successful here.

Now we will check whether this Scenario IIb can be a generalized solution and overcomes the UDP restriction that is imposed by the T-Mobile operator when the client and server are both on the T-Mobile network, as seen in the Test Case III. So, to check that, the next Test Case X is conducted.

**Test Case X:** Server on the T-Mobile EDGE network (Public IP+ Public Port), IAG on CN LAN (Public IP+ Public Port), Client on T-Mobile HSPA network (Private IP+ Private Port)

In this case, the media server is connected to the T-Mobile EDGE network while the client is also connected with the T-Mobile but at its HSPA network. The IAG will be connected with the ComNets (CN) LAN network as shown in Figure 5.23.

---

*Figure 5.23: Scenario IIb: Server on T-Mobile EDGE, IAG on CN LAN, Client on T-Mobile HSPA*
Now, in this test case, the client is on T-Mobile HSPA network and from the previous test cases, we know that client will be in private range behind NAPT/Firewall with a private transport address. Similarly, the server is also on T-Mobile but on its EDGE network and from the previous test cases, we know that server will not be behind any NAPT/Firewall and will have dynamic public IP address and public port. Now, just like in the Test Case IX, both the client and server will send the request packet towards the IAG at its fixed transport addresses. The client will send the request datagram (shown with orange colored datagram) to the IAG at its fixed public IP address 137.226.5.151 and public port “9092”. This will punch a hole in the client NAPT/Firewall and UDP Tunnel will be established between the IAG (at its fixed port 9092) and the client at its public mapped port B. Similarly the server will send the request packet (shown with blue colored datagram) from its public fixed port “9094” to the IAG at the same public IP address 137.226.5.151, but different port “9093”. As the server is not behind any NAT, so no UDP tunnel will be actually established, but still it will work as there is some tunnelled path between the server and the IAG. So, when the server will send the streaming packet (shown with green colored datagram) containing the media content towards the IAG at its public port “9093”, it will be successfully received by the IAG. Then the IAG will relay that streaming packet towards the client over the UDP tunnel that has already been established between the IAG and the client. The client will successfully receive that streaming packet through the UDP tunnel which can be played back later.

So, this shows that Scenario IIb even overcome the UDP restriction that is imposed by the T-Mobile operator and now both client and server can reside on the T-Mobile network. Thus the scenario IIb becomes successful and seems to be a generalized solution.

### 5.3.1.3 Summary of the test results

The summary of all the test cases that have been conducted before on the live operator networks to check the applicability of Scenario I and Scenario II (a/b) is shown in table 5.1.

From this table 5.1, it can be seen that Scenario I is only successful when the server is on T-Mobile EDGE network with public IP address and public port while the client is on the networks other than T-Mobile. But, this scenario fails when the client is on the T-Mobile HSPA network due to possible T-Mobile operator restriction over UDP. Even, the Scenario I fails when we put the server on the O2 UMTS/HSPA network because now server will come behind NAPT/Firewall due to which its IP address and port becomes private which cannot be accessed by any host from the external network. So this shows that the Scenario I cannot be considered as the generalized solution.

Now, coming to the Scenario IIa where we are using the IAG as the Introducer gateway (by which it is behaving like STUN Server and using Connection Reversal technique to work), but from the table, it can be seen that it is successful only in those cases when our client is on eduroam WLAN with the public IP address but private IP port. Here, the streaming is successful because eduroam WLAN exhibits the consistent port mapping property. But if the client is on the other networks than eduroam, this Scenario IIa becomes unsuccessful again.

Important information which we can get (but cannot check practically due to administrative issues) by the theoretical knowledge gained about the types of the NAT from the section 4.2.1.4, and about establishing the direct UDP based M2M sessions using the Connection Reversal in the section 4.2.1.7 is that if the server is on Symmetric NAT, while the client is on (IP) or Port Restricted Cone NAT, even then the Scenario IIa will be unsuccessful, although the client will have consistent port mapping property. So, for the Scenario IIa to become successful if the server is on Symmetric
<table>
<thead>
<tr>
<th>Client</th>
<th>Gateway (IAG)</th>
<th>Server</th>
<th>Scenario (I/II)</th>
<th>Result Observed</th>
</tr>
</thead>
<tbody>
<tr>
<td>eduroam WLAN</td>
<td>Public/Private</td>
<td>NA</td>
<td>T-Mobile EDGE</td>
<td>Public</td>
</tr>
<tr>
<td>O2 UMT-S/HSPA</td>
<td>Private</td>
<td>NA</td>
<td>T-Mobile EDGE</td>
<td>Public</td>
</tr>
<tr>
<td>T-Mobile HSPA</td>
<td>Private</td>
<td>NA</td>
<td>T-Mobile EDGE</td>
<td>Public</td>
</tr>
<tr>
<td>eduroam WLAN</td>
<td>Public/Private</td>
<td>Introducer</td>
<td>O2 UMT-S/HSPA</td>
<td>Private</td>
</tr>
<tr>
<td>eduroam WLAN</td>
<td>Public/Private</td>
<td>Introducer</td>
<td>T-Mobile HSPA</td>
<td>Private</td>
</tr>
<tr>
<td>O2 UMT-S/HSPA</td>
<td>Private</td>
<td>Relay</td>
<td>T-Mobile HSPA</td>
<td>Private</td>
</tr>
<tr>
<td>T-Mobile HSPA</td>
<td>Private</td>
<td>Relay</td>
<td>T-Mobile HSPA</td>
<td>Public</td>
</tr>
<tr>
<td>O2 UMT-S/HSPA</td>
<td>Private</td>
<td>Relay</td>
<td>T-Mobile EDGE</td>
<td>Public</td>
</tr>
<tr>
<td>eduroam WLAN</td>
<td>Public/Private</td>
<td>Relay</td>
<td>T-Mobile HSPA</td>
<td>Private</td>
</tr>
<tr>
<td>T-Mobile HSPA</td>
<td>Private</td>
<td>Relay</td>
<td>O2 UMT-S/HSPA</td>
<td>Private</td>
</tr>
<tr>
<td>eduroam WLAN</td>
<td>Public/Private</td>
<td>Relay</td>
<td>O2 UMT-S/HSPA</td>
<td>Private</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of Test Results
NAT, then the client must be on Full Cone NAT otherwise it will again not work. So, this shows that Scenario IIa also cannot be considered completely generalized.

Now, coming to the Scenario IIb where the IAG is used as the Relay Gateway (by which it is behaving like TURN Server). From the table 5.1 it can be seen that whatever is the network of either client or server, it is successful in all the cases. Even, we have checked for the case when the server and client are under the same NAT (not shown in the testing), and it is successful in those cases as well. Only there are two possibilities when this technique may fail. First is when the IAG is not accessible by either the client or server. Second, if the outgoing UDP connections are disabled or extra checks are configured for the incoming UDP connections at the firewall by the network administrator. But these two cases are very rare.

So, the Scenario IIb will hopefully be successful in almost every case. Thus, the Scenario IIb (where IAG is working as Relay Gateway) can be considered as a reliable and one of the most generalized solution for M2M streaming.

5.4 Implementation Architecture for Scenario I & Scenario II (a/b)

Based on the live testing results, now we have finally two main scenarios for the M2M multimedia streaming which we have implemented. One is of direct M2M where both signaling and streaming are carried out without the use of the IAG as an intermediate node, while the second is that where both the signaling and streaming are done with the help of IAG as an intermediate node (either for initializing or for complete streaming). Note that the signaling has been done by establishing the TCP (either HTTP or Socket) connection between server and client.

Scenario I: In the scenario I, which has been shown in Figure 5.24, both the signaling and streaming has been done directly between the server and client mobile nodes. The important thing to note here is that, while we were implementing this scenario on the live network, we have only found TCP port “8080” which can be accessed from the external network only on T-Mobile EDGE network. Secondly, only this particular EDGE network, we are getting the public (but dynamic) IP address and any public UDP port number. So for the implementation of Scenario I, the media server should be on T-Mobile EDGE network in this case. So in this scenario, the signaling has been done by initiating the TCP connection from the client (possibly lies behind NAPT/Firewall) to the server at its public dynamic IP address 88.x.x.x and fixed public TCP port “8080”. So, now as the TCP connection has been initiated from behind the NAPT/Firewall at the client, by the Hole Punching Technique (which is also valid for TCP), a signaling channel will be successfully established between the client and server over which the control messages can be sent. To initiate the streaming, UDP hole must be punched at the client’s NAPT/Firewall by initiating the outgoing UDP connection by the client from behind the NAPT/Firewall. So for that, the server will listen for UDP request packet at its any public UDP port e.g. “9094”. Then, the client will send the request data-gram to the server at its public transport address 88.x.x.x:9094. This request will punch a hole inside the client’s NAPT/Firewall and UDP tunnel will be established directly between the client and the server through the client’s NAPT/Firewall. Now, when the packet is arrived at the server, it will record the public mapped transport address from the UDP request packet and as a response, it will send the streaming packet (containing the media content) to the client at its public mapped transport address over the UDP tunnel which was established between client and server. When the streaming packets will arrive at the client’s NAPT/Firewall, they will successfully traverse NAPT/Firewall and will be received at the client. Thus, the complete direct M2M can be achieved.
It can be said that Scenario I (as a whole) is based on “STUN Server” and “Hole Punching Technique”. The only requirement is that server should be globally accessible both by TCP and UDP (which is achieved by putting the media server on T-Mobile EDGE network).

**Scenario IIa:** In the scenario IIa, the media server and the client both can be behind possible NAPT/Firewalls, so direct M2M connection will not be easily possible as shown in Figure 5.25. But for the implementation of this scenario, our work showed that client must be on that NAPT which will be either Full Cone Type or simply Restricted Cone Type having the property of consistent port mapping. As we are getting the Full Cone Type NAT public transport mapping on eduroam, so, to implement this scenario, client should be on the eduroam WLAN in this example.

Here the signaling has been done by establishing the two TCP connections with the IAG. One TCP connection (most likely HTTP originated from behind the NAPT/Firewall) will be established by the client with the IAG at its fixed public transport address 137.226.5.151:9090 (shown with green colored arrows). Similarly, another TCP connection (which will be most likely socket connection originated from behind the NAPT/Firewall) will be established by the server with the IAG at the same fixed IP address 137.226.5.151 but different public fixed port “9091” (shown with blue colored arrows). Now, the signaling messages can be relayed between the server and client by this IAG. For the streaming part, the UDP request packets will be sent by both server and client (shown with orange colored datagrams) at the public transport addresses 137.226.5.151:9092 and 137.226.5.151:9093, respectively. This will punch holes in their respective NAPT/Firewall gateways and two UDP based tunnels will be established both for server and client with IAG, respectively. Now when the request UDP packet (which was sent by the client) is received at the IAG, it will record the public mapped transport address from the packet and send it as a payload in the response packet over the UDP tunnel which has been established between the server and the IAG. When the server will receive the response, it will come to know about the public mapped transport address of the client. So, it will then send the streaming packets containing the media data towards that public mapped transport address. As, the NAPT/Firewall behind which the client lies exhibit the property of Full Cone NAT so it will allow the streaming packets to be received at the client. Thus the streaming is direct M2M.

Although, the Hole Punching has also worked for TCP when we have made the TCP sessions with the IAG, but the direct TCP based M2M sessions using the Hole Punching technique is very complex as compared to the establishment of direct UDP based M2M sessions. Secondly, the reliability is important for these signaling messages. That is why signaling has been done via IAG.
5.4. Implementation Architecture for Scenario I & Scenario II (a/b)

It can be said that in Scenario IIa, for signaling the IAG behaves like “TURN Server” and acts as Relay Gateway, and the signaling is based on simple TCP Hole Punching and relaying technique. For streaming, the IAG behaves like a “STUN Server” and acts as an Introducer Gateway and streaming is based on the UDP Hole Punching and Connection Reversal technique.

Scenario IIb: In the scenario IIb, the media server and the client both can again be behind possible NAPT/Firewalls just like the Scenario IIb, as shown in Figure 5.26. For the implementation of this scenario, no special NAPT/Firewall configuration is needed. Only the outgoing and incoming UDP connections should be allowed at the firewalls which is mostly the general case.

In this scenario IIb, the signaling will be done in the same way as implemented for Scenario IIa. Only the difference is that here streaming will also be carried out via IAG. Just like in Scenario IIa, the two UDP tunnels will be established by the server and the client with IAG, respectively. Then, the server will send the streaming packet containing the media data (shown with green colored datagrams) to the IAG over the UDP tunnel which has been established between the server and the IAG. When the streaming packet will be received by the IAG, it will simply relay the streaming packet towards the client over the UDP tunnel which has already been established between the client and the IAG and that packet will be successfully received at the client.

Figure 5.25: Scenario IIa: Direct M2M Streaming but Signaling via IAG

Figure 5.26: Scenario IIb: Both Signaling & Streaming via IAG
It can be said that in Scenario IIb, IAG will behave as a “TURN Server” and acts as Relay Gateway both for signaling and streaming. Both the signaling and streaming have been carried out by TCP Hole Punching and UDP Hole Punching, respectively, and by using the relaying technique.

5.5 RTSP Implementation over REST

It is said in the start of this chapter, RTSP based signaling between the server and client mobile nodes is implemented over the TCP connections i.e. by establishing HTTP or socket connections. For this purpose, REST architecture developed by [11] on ComNets middleware will be used which is capable of provisioning both synchronous and asynchronous services.

In this thesis, all 6 major (recommended and required) RTSP methods have been implemented to provide control functionality for M2M multimedia applications. These methods are OPTIONS, DESCRIBE, SETUP, PLAY, PAUSE and TEARDOWN. The OPTIONS method has been enhanced from the standard with some additional functionality that is required particularly for this thesis.

5.5.1 RTSP Methods using Synchronous Services

Two out of total six methods namely OPTIONS and DESCRIBE have been implemented using Synchronous services in the REST architecture. The reason for these methods to be implemented as Synchronous services is that they are only executed to perform short-lived operations, such as, to retrieve the options supported by the media server and to provide the initialization information.

OPTIONS

This RTSP request can be issued at any time, for example, if the client is about to try a nonstandard RTSP request (or any other request not supported by server). It is used by the client to know about all the options for the requests that it can send to the server. The generalized OPTIONS request message is shown in the listing 5.1.

```plaintext
1 OPTIONS ∗ RTSP/1.0
2 CSeq: 1
3 Require: implicit−play
```

Listing 5.1: Generalized OPTIONS Method Request

In this listing, Line 1 shows the request URI. After the line 1, line 2 contains the “CSeq” field which specifies the sequence number for an RTSP request-response pair i.e. every request should contain this field and there will be a corresponding response (even of any retransmitted request) having the same number. As it is the first request which will be generated from the client to the server, so CSeq value will be “1”. Line 3 contains the Require header field which is used by the client to query the server about the options that it may or may not support [4] and the server must respond to acknowledge positively to those options which are supported by it and negatively if it does not support that option. This is to make sure that the client-server interaction will proceed without delay when all options are understood by both sides [4] and the interaction will proceed quickly, saving a round-trip that is often required by negotiation mechanisms. Here the client has
sent the request to know about the options that are required for *implicit play* of media content, available by the media server.

In this thesis, OPTIONS method has been implemented with the additional functionality that is required by the application. Its purpose is to fetch all the collection of available media from the media server i.e. if the container file name is specified in the request, it will return with the names of all media (either Audio or Video) that is available in that container file for streaming. The implemented OPTIONS request message is shown in listing 5.2.

```
1 POST /OPTIONS RTSP/1.0
2 CSeq: 1
3 Require: implicit−play
4 Content-Type: text/xml
5 Content-Length: 119
6 ResourceMethod: POST
7
8 <xml version='1.0' ?>
9 <RESTReq>
10   <Rq>
11     <demo>TESTDATA</demo>
12     <ContextData>
13       <FName>ComNetMM</FName>
14     </ContextData>
15   </Rq>
16 </RESTReq>
```

**Listing 5.2: Implemented OPTIONS Method Request**

In the OPTIONS request message shown in listing 5.2, line 1-6 shows the request header while from 8-16 shows the XML based payload. Line 1 is actually the REST based Request URI in which the first field shows the Request Method field whose value can be either “POST” or “GET”. If this field contains “POST” (like here), it means the request header will be followed by payload otherwise its value will be “GET” in which there is no payload after the request header. The next field after the “/” shows the requested Synchronous service which is OPTIONS in this case. At the end is the RTSP version field which is “1.0” currently. After the line 1, line 2 contains the “CSeq” field and line 3 contains the “Require” fields which are same as discussed for generalized OPTIONS request message before. After this, line 4 contains the “Content-Type” field which shows the type of the contents contained as the payload. Here it can be either any text or XML data. Line 5 contains the “Content-Length” field which shows the length of the payload contents (in terms of number of bytes) after the request header fields. The last header field at line 6 is of “ResourceMethod” which is similar to the request method field discussed before and its value will also be either “POST” or “GET”.

From line 8, the XML based payload starts and this line shows the XML version which is “1.0” here. At line 9, there exists a body element which is “RESTReq” here and contains child element “Rq”. Rq contains further two child elements named as “demo” which can contain any demo text value while next element is “ContextData” which contains the child element named as “FName” at line 13. This FName field contains the actual input request data and in this request, it will carry the specific Folder Name that exists on the storage e.g. the SD Card at media server, and will contain all those media files which are available for streaming.

The generalized OPTIONS message response has been shown in the listing 5.3. In line 1, again the first field shows the RTSP version 1.0 followed by the Status Code and Reason Phrase. The status code is a 3-digit integer result code of the attempt to understand and satisfy the request [4] while the reason phrase is intended to give a short textual description of the status code. The status code
of “200” shows the success and with reason phrase of “OK” tells that the action was successfully received, understood and accepted [4]. The next header field “CSeq” contains the same sequence value “1” as was contained in the OPTIONS request. Line 3 contains the Cache-Control general header field e.g. “Public”, which is used to specify directives that must be obeyed by all caching mechanisms along the request/response chain [4]. Here the “Public” field at line 3 indicates that the media stream is cacheable by any cache [4] and contains all the options namely DESCRIBE, SETUP, PLAY, PAUSE and TEARDOWN that are required for implicit play and can be requested by the client to the media server.

The implemented OPTIONS method response is shown in listing 5.4. Again here the lines 1-5 show the header while lines 7-13 show the payload. In line 1, again the first field shows the RTSP version 1.0 followed by the Status Code of 200 and Reason Phrase OK. The next header fields “CSeq” at line 2 and “Public” at line 3 are same as shown for generalized OPTIONS response message. “Content-Type” and “Content-Length” at lines 4 and 5 are same as discussed for implemented OPTIONS request message before.

From line 7 onward, the payload contained in the response is started where exists the “Rs” body element. Rs further contains the child elements named as File0 till File3 where each of this child carries the name of the media that is available for streaming at the media server. Here it is showing that there are four wave format audio files namely sga.wav, sgnn.wav, mzrw.wav and mzr.wav which are available for streaming.

### DESCRIBE

This RTSP method retrieves the low level description i.e. initialization information of the media object. This information can be received from any web server over HTTP, but in this thesis, it is retrieved directly from the media server. One of the acceptable formats for this initialization data at the client is based on SDP.

The generalized DESCRIBE method request is shown in listing 5.5. From the listing, it can be seen that DESCRIBE request is based on the header fields only. In the line 1, there exists the URL for a
5.5. RTSP Implementation over REST

a particular presentation. e.g. here “rtsp://media.example.com:554/twister” identifies the presentation named “twister” (which can contain multiple audio/video streams) and is controlled by the mentioned RTSP request issued over TCP connection to port “554” of host “media.example.com”. If the port is not defined, then the port number 554 is assumed by default. In the next line 2 is again the CSeq field for sequence number. In the last line 3, the field “Accept” shows the acceptable format in which the initialization information from the media server can be received by the client. Here, the acceptable initialization format should be based on SDP so that it can be acceptable to the client.

Now, in our implementation, the rtsp:// URL is neither supported by the already developed HTTP connection based REST middleware available at ComNets nor the ordinary “URL connection” in the Android platform. So, these RTSP requests/responses are received by first establishing the socket connection with the media server over which these requests and responses are communicated between the client and server. The implemented DESCRIBE request is shown in listing 5.6.

It is different from the generalized DESCRIBE request because to extend it for REST based architecture, the presentation name cannot be included in the request URL, so we put it as an input parameter in the XML based payload. Just like the OPTIONS request discussed before, here the lines 1-6 contains the header fields which all have the same functionality that was discussed for OPTIONS request. Only the extra header field is “Accept” which is showing the acceptable format in which the media description will be accepted by the client as discussed in the generalized DESCRIBE request. From line 8-16 shows the XML based payload where line 12 contains the element “ContextData” whose child element “FName” is containing the name of the particular presentation file e.g. “mzr.wav” which has been selected by the user at the client from the list of available streaming media. Then, the media server will provide all the low level initialization information against this particular presentation. It is the same presentation which has been addressed by the RTSP request URL in the generalized DESCRIBE request shown in listing 5.5.
The generalized DESCRIBE method response is shown in listing 5.7, where the lines 1-4 contains the header and from lines 6-12 shows the initialization information for the presentation named as “twister” in the form of SDP. On the same guidelines, the DESCRIBE response has been implemented as shown in the listing 5.8.

Listing 5.7: Generalized DESCRIBE Method Response

Just like in the generalized DESCRIBE response, lines 1-4 in the implemented DESCRIBE response contains the header fields and lines 6-23 shows the SDP based description. One of the key difference from the generalized response here is that the initialization information that has been received at the client is implemented as XML based payload, otherwise, the fields are almost the same in both generalized and implemented DESCRIBE response.

The first element “v” at line 8 shows the version number which is “0”. The next line 9 contains the element “o” which shows the origin. Its starting value “-” shows that it does not support any user’s login system. The next value “2008227410” is session id. After that “2070171982” is the version. The next sub-field is for network type where “IN” denotes the Internet. After that, the sub-field shows the address type where “IP4” shows that it belongs to IP version 4. The next
sub-field shows the unique address of the originator of the request. Here it is “137.226.5.151” which shows that the request is generated by the IAG because we have generated this SDP for Scenario IIb where the IAG is acting as an intermediate node from where this DESCRIBE request was previously generated and the original client is not known to the media server. Otherwise, for scenario I, here we can have unique public IP address of the client which has originally issued the DESCRIBE request. In the next line 10, there exists element “s” showing the Session Name which is “RTSP Session” here. Line 11 contains the element “c” which denotes the Connection Data. Its first sub-field denotes the network type which is “IN” and shows that it is the Internet here. Next sub-field is about the address type which is of IP version 4 as shown by “IP4”. The last field is for connection address which denotes the destination address for the media stream. For on-demand services, the client generally specifies the destination address via SETUP request. As the destination address is not fixed and determined yet, so this field can be filled by “0.0.0.0”.

Upto here all the fields and their order is completely same in both generalized and implemented DESCRIBE response. In the generalized DESCRIBE response shown in listing 5.7, line 10 contains the session level attribute which contains the IANA registered attribute named as “control” and gives the rtsp based request URL “rtsp://media.example.com/twister/” for aggregate control of the presentation twister. As twister here is supposed to be of single audio stream presentation, so no additional media level attributes have been defined.

In contrast to this generalized DESCRIBE response, the implemented DESCRIBE response contains the element “Sa” at line 12 to denote the session level attribute. As we are unable to implement the rtsp based URL, so we implement our self defined attributes contained at line 13 and 14 namely “Xcontrol1” and “Xcontrol2”. As they are experimental attributes that we have developed for our implementation, so they are started with “X”. The reason that we are having here two control attributes is that first control attribute “Xcontrol1” will give the URL “POST /Request-Response/Factory/createInstanceRq/StreamMM/SETUP” which will be used by RTSP SETUP method by which the media stream channel is established. As this URL is actually for creation of the asynchronous service, thats why it is different from the other control attribute “Xcontrol2” having the control URL as “POST /Request-Response/Instance/changeStateRq/StreamMM/*”. This URL will be basically used for the service invocation and service control inside REST architecture. In RTSP, basically the SETUP is used for streaming service start (i.e. Service Invocation) and then the remaining RTSP methods PLAY, PAUSE and TEARDOWN are used for service control, so they will use this control URL. The “*” at the end denotes that it is sort of generalized URL which can be used by any of PLAY, PAUSE and TEARDOWN methods. The next experimental unregistered attribute at line 15 is “Xmtu” which denotes the maximum transmission unit in terms of number of bytes. Here its value is set as “1500” bytes because this is the maximum MTU which is available on ETHERNET and EDGE networks.

Then, the line 17 of the implemented DESCRIBE response contains the element “t” which contains the sub-fields for start and stop times for which the RTSP session is available and the generated SDP is valid. Here it contains both the start and end time values as “0” which shows the creation of the “permanent session”. Although as discussed in SDP literature section 4, it has been strongly discouraged to make the permanent session, but here we are making up the permanent sessions to support this point that in the RTSP implementation, once an RTSP session is established between the client and the server, it will be remain available until explicitly ended by client or server. Secondly, once the SDP is generated against the specific selected file, it will remain valid throughout the time for which the RTSP session is established.

At line 18, there exists the element “m” which denotes the media description. Its first sub-field shows the media type which is “audio” here. The next sub-field shows the transport port to which the streaming data will be sent at the client. Here “0” value shows that it has not been decided
yet and will be specified during the SETUP request. After that is the transport protocol sub-field which is “RTP/AVP” showing that it is RTP based and belongs to Audio-Video profile. The next sub-field shows the payload type and as its value is “98” which shows audio media “mzr.wav” is actually of dynamic type. So, the extra media level attribute is needed here which is denoted by “Ma” at line 19, to make it differentiable from the session level attribute Sa. It contains the registered media level attribute named as “rtpmap” at line 20. Its first field denotes the payload type which is “98” here. The next sub-field shows the encoding name/clock rate/encoding parameters which is “X-PCMwav8/11025/1” showing that it is experimental encoding PCM wave format at 8 bits/sample with the clock rate of 11025 Hz and the number of channels is “1” i.e. Mono. Line 21 contains another media level but unregistered attribute namely “Xfs” showing the file size in terms of numbers of bytes which are “1432256” here.

It is to be noted that in generalized DESCRIBE response message (as shown in listing 5.7), there does not exist any media level attribute because it contains the payload type “3” which shows the static payload type as per discussion in Payload Type subheading under section 2.2.1.

5.5.2 RTSP Methods using Asynchronous Services

Out of 6, the remaining 4 RTSP methods SETUP, PLAY, PAUSE and TEARDOWN have been implemented using Asynchronous services of the REST architecture. These methods have been implemented over Asynchronous services because SETUP method will be used for Service Creation and Invocation of asynchronous multimedia streaming service while PLAY, PAUSE and TEARDOWN will be used for Service Control by changing the State of the service.

SETUP

SETUP request and response contains all the transport related initialization information. It also specifies the transport mechanism over which the media content will be streamed. The generalized SETUP request has been shown in listing 5.9

```
1 SETUP rtp://media.example.com/twister RTSP/1.0
2 CSeq: 3
3 Transport: RTP/AVP; unicast; client_port=4588–4589
```

Listing 5.9: Generalized SETUP Method Request

In this listing, line 1 contains the name of the presentation (stream) “twister” for which this SETUP request is sent. As “twister” is only single stream audio file, so only single request is sent, but if “twister” is Audio/Video presentation, then SETUP request will be sent both for audio and video tracks separately. At line 3, there exists the header field Transport which contains all those transport initialization parameters that are acceptable to the client. Its first value “RTP/AVP” is showing that streaming will be done by using RTP over UDP (default is UDP, if UDP/TCP not mentioned specifically) using Audio-Video profile. It will be unicast transport with the client’s RTP port “4588” and corresponding RTCP port “4589” for receiving media data and control information respectively.

The SETUP method request which has been implemented in this thesis is shown in the listing 5.10. As discussed before, in this thesis the SETUP request is used for the Creation and Invocation of the service. In this listing, line 1 is showing that request is sent to RESTful Factory for the creation of
5.5. **RTSP Implementation over REST**

**Listing 5.10:** Implemented SETUP Method Request

Instance of the asynchronous service which is “StreamMM” here. The transport header field at line 3 is defined in the same way as it was discussed for the generalized case. However the implemented SETUP request is different from the generalized one because to follow the REST URL format, the name of the selected particular file or stream cannot be mentioned in the REST URL at the line 1. So the file or stream (track name) against which this SETUP request is sent, it is mentioned as value “mzr.wav” contained in the XML based payload field element “FName” at line 15. To invoke the service along with the same single service creation request, the “StartImmediately” element at line 9 will be set with the value “true” showing that service will be invoked just after its creation. By this, the service will default come to “rtsp.init” state which is first INIT RTSP state of the media server.

Other important thing is that the implemented SETUP request will also carry Observer’s unique EPR “http://192.168.0.109:9097” under the XML element “ObserverEPR” at line 10 so that it can be tracked at the server that which particular observer (client) has requested the particular service.

**Listing 5.11:** Generalized SETUP Method Response

The generalized SETUP method response has been shown in listing 5.11. It also contains the Transport header field containing the transport mechanism and client’s RTP/RTCP port numbers as shown in line 4, but it also shows the corresponding RTP and RTCP ports i.e. “6256” and “6257” that will be used by the server for receiving media data and control information (in case of two way communication). The important header field that is contained in the SETUP response is Session which actually carries the unique “session id” that is allocated for each stream/track (contained inside presentation file) against the single SETUP request. This is the unique id which will be used later by the PLAY, PAUSE and TEARDOWN requests for playing, pausing and stopping the particular track. Now the implemented SETUP method response has been shown in the listing 5.12.

Although the implemented SETUP response also contains the “Session” header field at line 3 with the 8 figured value “12345678”, but this is actually a hard coded entry and only added in
Mobile-to-Mobile Streaming using Mobile Web Server

Listing 5.12: Implemented SETUP Method Response

implementation to have a close realization with the generalized SETUP response. The actual entry by which the streaming functionality will be controlled is the unique instance EPR. This unique EPR “http://127.0.0.1:22981/instance0” contained under the element InstanceEPR at line 9 will act as the session id and by using this instance EPR, the streaming can be played, paused or stopped by the observer (client).

PLAY

By this method, the client tells the media server to start sending (streaming) data via the mechanism that has been specified in previous SETUP request. The generalized PLAY method request has been shown in listing 5.13

Listing 5.13: Generalized PLAY Method Request

In this request, it can be seen that the PLAY request has been sent for the particular stream by sending the specific session id “47112344” contained by the Session header field at line 3. It is the same session id which was received at the client by the previous SETUP response message. As the presentation “twister” is actually a single audio track (stream) presentation, so only single PLAY request will be enough. However, if the presentation contains more than one track (streams), then PLAY request will be sent for each track along with its particular session id that has been allocated to it during SETUP.

Listing 5.14: Implemented PLAY Method Request

Listing 5.14 shows the implemented PLAY method request. Just like for the generalized request,
Session field has also been added at line 3 in the implemented PLAY request but as mentioned in the implemented SETUP response before in listing 5.12, it is only hard coded and just added for the close realization with the generalized PLAY request format. The actual thing which is working like session id is “http://127.0.0.1:22981/instance0” at line 3 which is Instance EPR and by mentioning “rtsp.playing” in the XML element “State” as shown in line 9, the next RTSP state of server is requested. In the PLAY implementation, the PLAY request is sent with the instance EPR by which the server picks out the Instance of particular service. Then by changing the state of the service instance to “rtsp.playing” will bring the server to RTSP PLAYING state and the RTP packets containing the media data can be sent over UDP by the server towards client.

Note that the PLAY request has been sent without specifying any “Range” header field in both generalized and implemented requests. By this convention, stream will be played from the beginning unless it would be paused. Once it is resumed back, it will start to play again from that point where it was paused.

---

1 RTSP/1.0 200 OK
2 CSeq: 4
3 Date: 23 Jan 1997 15:35:06 GMT

Listing 5.15: Generalized PLAY Method Response

The generalized PLAY method response is shown in listing 5.15 while the implemented PLAY method response is shown in listing 5.16.

---

1 RTSP/1.0 200 OK
2 CSeq: 4
3 Date: Dec 22, 2010 12:09:30 AM
4 Content-Length: 139
5
6 <?xml version='1.0'?>
7 <changeStateRs>
8   <InstanceEPR>http://127.0.0.1:22981/instance0</InstanceEPR>
9   <State>rtsp.playing</State>
10 </changeStateRs>

Listing 5.16: Implemented PLAY Method Response

In the implemented PLAY method response, the current date and time has been shown in “UMT” format by the Date header field at line 3. Secondly, the instance EPR “http://127.0.0.1:22981/instance0” as shown in line 8 and the current state of the service instance “rtsp.playing” as shown in line 9 confirms that the RTSP server has changed its current state to PLAYING state successfully.

PAUSE

By the PAUSE request, the stream delivery can be temporarily interrupted or halted. However it should be noted that the resources which are allocated for streaming are kept allocated. The generalized PAUSE method request is shown in the listing 5.17. Just like mentioned for the previous PLAY request, the PAUSE request will also contain the Session header field which will carry the unique session id “47112344” by which the single track presentation “twister” will be paused. As the previous PLAY request shown in listing 5.13 did not contain any Range header field, so when the PAUSE request is received at the server, stream delivery is interrupted immediately and the
Mobile-to-Mobile Streaming using Mobile Web Server

PAUSE rtp://media.example.com/twister RTSP/1.0
CSeq: 5
Session: 47112344

Listing 5.17: Generalized PAUSE Method Request

The implemented PAUSE request is shown in listing 5.18. Again the implemented PAUSE request also contains the Session header field, but it will not doing any thing significant except to have closed resemblance with the generalized PAUSE request. In this PAUSE request implementation, the specific service instance will be selected by using Instance EPR “http://127.0.0.1:22981/instance0” contained at line 9 and then the instance is changed from its current PLAYING state to READY state by using the value “rtp.ready” contained by the State element at line 10.

Listing 5.18: Implemented PAUSE Method Request

The corresponding generalized PAUSE method response is shown in the listing 5.19 while the implemented PAUSE method response is shown in the listing 5.20. The current state of the service instance is shown by the element State at line 9 of the implemented PAUSE response where “rtp.ready” shows that the server has changed its current state to “READY” state successfully.

Listing 5.19: Generalized PAUSE Method Response

TEARDOWN

By issuing this request, the stream delivery will be stopped and all the resources which were allocated with it during SETUP request will be freed. The generalized TEARDOWN method request has been shown in the listing 5.21.

As this listing 5.21 contains the name of the presentation “twister” in the request URI at line 1, so the RTSP session identifier “47112344” contained by the Session header field at line 3 will no longer will be valid after the processing of this TEARDOWN request. The implemented TEARDOWN method request has been shown in listing 5.22.
5.5. **RTSP Implementation over REST**

---

1. RTSP/1.0 200 OK
2. CSeq: 5
3. Date: Dec 22, 2010 12:09:49 AM
4. Content-Length: 137
5. 
6. ```xml
5.5. RTSP Implementation over REST

Listing 5.20: Implemented PAUSE Method Response

```xml
<?xml version='1.0' ?>
<changeStateRs>
  <InstanceEPR>http://127.0.0.1:22981/instance0</InstanceEPR>
  <State>rtsp.ready</State>
</changeStateRs>
```

Listing 5.21: Generalized TEARDOWN Method Request

---

1. TEARDOWN rtsp://media.example.com/twister RTSP/1.0
2. CSeq: 6
3. Session: 47112344

---

In the implemented TEARDOWN request, there again exists the `Session` header field at line 3 which is again a dummy representational header field only. The actual stream delivery is stopped by first identifying the specific instance of asynchronous streaming service “StreamMM” with the help of instance EPR “http://127.0.0.1:22981/instance0” contained at line 9. Then the current state of the instance is changed to `INIT` by using “rtsp.init” contained in the `State` element at line 10.

The corresponding generalized TEARDOWN method response is shown in the listing 5.23 while the implemented TEARDOWN method response is shown in the listing 5.24. The current state of the service instance is shown by the element `State` at line 9 of the implemented TEARDOWN response where “rtsp.init” shows that the server has changed its current state to “INIT” state successfully. The other difference of the implemented TEARDOWN response from the generalized one is the additional `Date` header field at line 3 which does not cause as such significant difference.

5.5.3 Integration of Streaming functionality over UDP

As discussed before in this report that RTSP signaling is *out-of-band*. So the streaming channel over which the media data will be delivered is separate and can be different from the signaling channel. So in this thesis, the signaling is implemented by using the REST architecture over TCP sockets. For the streaming, the RTP packets are transported over UDP datagrams.

To implement this streaming functionality, two additional java classes named as `mmServer` and `mmClient` have been added in the `cnMultiMedia` package.

Here `mmServer` will be our original media streaming server which will make the asynchronous service `StreamMM` service capable to make chunks of media content file, put them as RTP payload inside the RTP packets and then send them over UDP datagrams. When the `StreamMM` service is invoked by its instance, `StreamMM invoke()` function is called which takes the StreamMM Instance and Restful Object as an input parameters, as shown in the code snippet 5.27

Here in this `invoke()` function, first the name of the audio file is retrieved from the “Context-Data” element contained in the `RESTfulRequestObject` by using the function `Xml2Data()` of `DealContextData` class. Then the current state of the `StreamMM` server is updated to “rtsp.ready” because the service instance is invoked just after its creation. When the audio file name is retrieved, then object of `mmServer` class is created in which the `StreamMM` will also pass its instance
as argument. Finally, the `ServerConnection()` function of `mmServer` class is called which will take the name of the selected audio file as input parameter. This function will actually create the **Datagram Sockets** at the server and depending on the state of the instance, it will send the UDP datagrams (containing the RTP packets with the media data) towards the client.

The media client is implemented by developing the `mmClient` class which will actually receive the UDP datagram from the media server, picks the RTP packet from the received datagram and then separate out the media contents contained as RTP payload from the RTP packet. Then this separated media content will be played by the media player on the client. At the client, the **RESTful Observer** will send the **SETUP** request and will receive its response inside the `CreateInstance()` function as shown in the code snippet 5.28. In this `CreateInstance()` function, first the protocol type is checked which will be “RTSP” protocol that is selected here. Then the current RTSP state of the client will be set to “rtsp.init”. Finally the SETUP request and response will be sent and received respectively inside the `call()` function. After receiving of the SETUP response, the object of `mmClient` class will be created which will also take the **RESTful Observer** object as input argument. Finally, the `ClientConnection()` function of the `mmClient` will be called which will take the streaming password (needed for streaming packet authentication by IAG in Scenario II implementation) as an input argument. This `ClientConnection()` function will then create the Datagram Socket by which it will first send the UDP datagram on the outgoing UDP connection to punch hole inside its NAT, then it will receive the UDP datagrams containing either the “UDPKeepAlives” to keep the NAT public transport allocations or the RTP packets that will contain the data contents as RTP payload.

### 5.5.4 Differences from the existing REST middleware

Although in this thesis, it has been tried to extend the functionality for the multimedia support by using the existing available REST middleware, however at some points, the existing functions are not used or may be replaced by the new extended functionality.

The first main difference from the existing middleware is that three more states have been added to `StateType` class namely “rtsp.init”, “rtsp.ready” and “rtsp.playing” states along with the existing states to show the three RTSP states “INIT”, “READY” and “PLAYING” at the media server.
5.5. **RTSP Implementation over REST**

```
RTSP/1.0  200 OK
CSeq: 6
Date: Dec 22, 2010 12:10:04 AM
Content-Length: 136

<?xml version='1.0'?>
<changeStateRs>
  <InstanceEPR>http://127.0.0.1:22981/instance0</InstanceEPR>
  <State>rtsp.init</State>
</changeStateRs>
```

Listing 5.24: Implemented TEARDOWN Method Response

```java
public void invoke(RESTfulInstance instance, RESTfulRequestObject restObj) {

    Element contextElement = restObj.getBodyElement().getElement("","ContextData");
    DealContextData audioFileName = new DealContextData();
    String audioFile = audioFileName.Xml2Data(contextElement);

    if (!instance.getState().getValue().equals(StateType._rtspREADY)) {
        instance.ChangeState(new ChangeStateRq(StateType._rtspREADY));
    } else if (instance.getState().getValue().equals(StateType._rtspREADY)) {
        Log.i("RTSP Server", "StreamMM::invoke() Successfully");
    }

    servMM = new mmServer(instance);
    try {
        servMM.ServerConnection(audioFile);
    } catch (IOException ex) {
        ....
    }
}
```

Figure 5.27: StreamMM Service Invocation and mmServer started

and the client respectively. From the six already existing states discussed under section 3.2.2.3, only three namely “closed.completed”, “closed.abnormalCompleted” and “closed.abnormalCompleted.aborted” will be used to make total of six states. “closed.completed” state is used to show the successful streaming of the complete file from the media server to the client. “closed.abnormalCompleted” state is occurred when the client issues the TEARDOWN request by which the streaming is stopped without completion. “closed.abnormalCompleted.aborted” is occurred when some exception occurred at the server during the streaming of media files. Note that for both TEARDOWN case by the client and exception case at the server, abnormalCompleted type of states are used because the media data that has already been streamed from the server to the client is still valid and is available at the client as the portion of the file to be played back.

The second important difference is that, in the existing architecture, RESTful Observer (client) initiates the HTTP Connection by using http URL with the media server over which the REST requests/responses are sent and received while in this implementation, as the Android platform does not support rtsp URL, so here its functionality is developed by first establishing the Socket Connection from the client to the media server and then RTSP requests and responses are sent and
public String CreateInstance(int TYPE, String fileName) {

    this.PROTOCOL_TYPE = TYPE; //Protocol Type=3, "RTSP" is selected
    try {
        Document RqXml = obsConfig.CreateInstance(fileName);

        //Setting "INIT" state while sending "SETUP" request
        previousState = StateType._rtspINIT;
        currentState = StateType._rtspINIT;

        //Sending & receiving of SETUP request/response
        restRespObj = call(RqXml, TYPE);

        previousState = currentState;

        //Starting the receiving of streaming packets
        strObj = new mmClient(this);
        strObj.ClientConnection(SetConfig.IAG_Streaming_Password);

        while (true) {
            if (strObj.getRTSPstate().equalsIgnoreCase(StateType._rtspREADY)) {
                break;
            }
        }
    } catch (Exception e) {
        
    }
}

Figure 5.28: Receiving of the SETUP response and starting mmClient
received respectively over that established socket connection.

The last but one of the most important difference is that in the existing middleware architecture, the partial results computed by service and the state changed updates are sent by the Service Instance to the client through an HTTP connection using the RESTful notification class at the server. But here in this implementation, although the RESTful Notification is still there but it has not been extended and used for this multimedia functionality. The results and the state changed updates to synchronize with the server have been implemented over UDP instead of HTTP. The reason for this is that as we are running our application over the live operator network in which the control messages are relayed over socket connections while the streaming of data packets are sent by UDP datagrams. As this network can be lossy, due to retransmission functionality of TCP, the responses of the RTSP requests may be delayed as compared to the UDP datagrams. So it can be possible that at some time, the media client sends the PLAY request which will change the state of the server from READY to PLAYING and start sending the UDP datagrams with the valid data. But due to lossy network, the PLAY response is delayed. So for the mean time, the client may remain in the READY state and cannot receive the UDP datagrams with media data, if it is synchronized with TCP based socket functionality. So to synchronize both UDP based media data and TCP based signaling data, the state change updates notifications (due to completion of the streaming data or occurrence of any exception) have been provided by sending the UDP datagrams. In the RESTful Instance class, the sendUDPNotificationPacket() function has been added to provide this functionality as shown in code snippet 5.29

```java
public void sendUDPNotificationPacket(DatagramSocket insDgc, DatagramPacket insDatagram, SocketAddress addr, String notifyMsg) {
    String notificationMsg = new String(notifyMsg); //"KeepAlive", "EndUDPConn",
    try {
        try {
            insDatagram = new DatagramPacket(notificationMsg.getBytes(),
                                             notificationMsg.getBytes().length, addr);
        } catch (SocketException ex) {
            ...
        }
        catch (Exception ex) {
            ...
        }
        try {
            insDgc.send(insDatagram);
        } catch (IOException ex) {
            ...
        }
    }
}
```

**Figure 5.29:** Sending of UDP based Notifications

Here sendUDPNotificationPacket function will take the Datagram Socket, Datagram Packet, Socket Address and Notification Message as an input arguments which will be provided to it by mmServer class. The notification messages can be either “KeepAlive” when the server comes in the READY state after receiving SETUP or PAUSE requests, “EndUDPConn” when the server
comes to INIT state after receiving the TEARDOWN request or “InterruptUDPConn” when any exception occurs at the server during streaming. “endRelayTransmission” will be used to stop the IAG for relaying the streaming packets when we are following the Scenario Iib. So this function will put one of these notification messages inside the Datagram Packet, set its destination address from the Socket Address and send the UDP datagram by using the Datagram Socket. Then these notifications packets will be received by the Datagram Socket created by mmClient class which will fetch out the particular UDP notification message. For the “InterruptUDPConn” notification message, mmClient will update the previous and current states of RESTful Observer by using the function setObserverState(String) which has been extended for RESTful Observer and has been shown in the code snippet 5.30.

```java
public void setObserverState(String currState) {
    try {
        previousState = currentState;
        currentState = currState;
    } catch (Exception e) {
    }
}
```

**Figure 5.30:** Setting of Observer State after receiving UDP Notification

For the remaining “KeepAlive” and “EndUDPConn” notification messages, mmClient will only update its own states. The RESTful Observer states are set by RTSP responses like PAUSE and TEARDOWN which are received over socket connections.

### 5.5.5 Fully Working Implementation for Scenarios I and II (a/b)

Upto now, all the basic implementation things have been discussed. In this subsection, the finalized working that has been implemented in this thesis will be discussed. As discussed before that there are 3 scenarios that have been developed. First is the Scenario I where both the Signaling and Streaming have been done directly from the server mobile to the client mobile. In the second and third Scenario IIA and Scenario IIB, the Signaling will be done via IAG while streaming will be done either by using IAG or via IAG.

#### 5.5.5.1 Media Server and Media Client Configuration

The media server and media client which are developed on android platform can be configured into 3 scenarios namely Scenario I, Scenario IIA and Scenario IIB. To set the particular scenario and some other streaming parameters for the streaming server and the streaming client, SetConfig class has been added in the cnMultiMedia package. The important configuration parameters provided in the SetConfig class are mentioned as follows:

- **selectScenario** for setting up the particular scenario at the media server and the client. If “selectScenario = 0”, then both media server and client will be configured to work with Scenario IIB (of Relay gateway). If “selectScenario = 1”, then Scenario I (of direct M2M) will be selected while if “selectScenario = 2”, then Scenario IIA (of Introducer gateway) will be selected.
5.5. RTSP Implementation over REST

- **IAG Streaming Password** acting as the Security Password (if Scenario IIb is selected) required for the authentication of media client with the IAG for relaying of the streaming media content by the IAG towards the client. By default, it is set as “cnhello”. In Scenario IIb, when the client sends the first UDP datagram towards the IAG to punch hole inside its NAT and to establish UDP tunnel with IAG, then if this password is authenticated by the IAG, only then UDP tunnel is established successfully between the client and IAG over which IAG will relay the streaming packets towards the client.

- **Directory_Name** to contain the name of the folder/directory on the storage like SD Card which will carry all Audio/Video files that are available for the streaming by the media server. This parameter will be used during the OPTIONS request/response where the particular directory name will be mentioned as request input, then at the media server, only that particular directory will be selected and all the multimedia data (audio or video), their file names will be provided in the OPTIONS response. By default, the “ComNetMM” directory is selected at the SD Card of media server.

- **Global_GW_IP** to carry the fixed public IP address of Intermediate Access Gateway (IAG) which is used as an intermediate node for Scenario IIa and Scenario IIb. Currently this IAG is present at the ComNets place with the fixed public IP address of “137.226.5.151”.

- **RTP_payload_inBytes** to provide the “RTP Payload Size” in number of bytes that can be contained inside the RTP packet other than 12 bytes of RTP header. As the Maximum Transmission Unit (MTU) is of 1500 bytes for the EDGE network, and we already know from the previous knowledge that IP header is of 20 bytes, UDP header is of 8 bytes and the RTP header (implemented in this thesis) is of 12 bytes, then the maximum value of **RTP_payload_inBytes** can be 1500- (20+8+12) = 1460 bytes.

- **transmissionDelay_in_msec** to provide the delay in msec between the transmission of any two packets. It is the minimum delay that must be provided by the streaming media server when it has transmitted one packet and going to transmit the next one. It is very important parameter and will be used to synchronize the streaming client with the streaming server as there is no common clock available between the client and server. Our testing has shown that when the server and client are on UMTS/HSPA network and eduroam WLAN respectively and vice versa, then **transmissionDelay_in_msec** of 200-300 msec gives the successful streaming synchronization between the client and the server.

5.5.5.2 Intermediate Access Gateway (IAG) Configuration

As per described earlier, IAG will act as the intermediate node between the client and the media server. It can either work as an “Introducer gateway” in Scenario IIa or as a “Relay gateway” in Scenario IIb. To select the configuration of the IAG, the configuration parameter **IAGmode** inside the rtspIAGMain class inside the IAG programming project named as “rtspIAG” developed in Java SE. If the “IAGmode = 0”, Scenario IIa will be selected where IAG will act as Introducer gateway for streaming channel while if the “IAGmode = 1”, then Scenario IIb will be selected where IAG will act as Relay gateway.

5.5.5.3 Streaming Media Server

As discussed before, in this thesis, the streaming media server functionality is provided through mmServer class. mmServer uses the functionality of other implemented classes like RTPpacket
and ReadSourceFile for reading any file byte by byte, add the data inside the RTP packet and send it towards the client. Good thing here for this media streaming server is that, with the help of functionality provided by ReadSourceFile class, it can stream file of any format. So by this, our server can be considered as generalized multimedia streaming server.

Inside the mmServer class, all the work has been done by the single ServerConnection function which is taking help from several other small functions. Inside this ServerConnection(String) function, it takes the file name as an input parameter by which it will locate the file on its SD card and open it in the read format by using its function readFile(String) as shown in the code snippet 5.31.

```java
public void readFile(String filename) {
    try {
        video = new ReadSourceFile(filename, this);
    } catch (Exception ex) {
        ...
    }
}
```

**Figure 5.31:** Opening up file for Reading

Inside this function, mmServer is actually making the object of ReadSourceFile class and inside the constructor of ReadSourceFile class, its InitializeAll() function is called which is actually locating the desired file on the SD card whose name is provided as an input argument to readFile(String) function and then open the input stream to read that file as shown in the code 5.6. After this in ServerConnection() function, the state of the RESTful Instance object named as myInstance of asynchronous service “StreamMM” will be checked that has been provided as an input argument to the mmServer constructor. If the state of myInstance object is “rtsp.ready”, then the file will be read chunk by chunk by using the function getNextFrame() of ReadSourceFile class as shown in the code 5.7. When one chunk of file is read and kept inside the byte array, ServerConnection() function will call another function sendRTPpacket() for sending this media data as an RTP packet towards the client. The code snippet for sendRTPpacket() function has been shown in the figure 5.32.

Here in this function, it will take marker value, sampling time, media content i.e. message, number of bytes of message and socket address of client destination as the input parameters. Then it will make the RTP packet by using the constructor of RTPPacket class to make the RTP packet as shown in figure 5.8. When the RTP packet is constructed, it will be encapsulated inside the datagram packet. Then the destination address for this datagram packet will be set according to the socket address that has been taken as an input argument and finally the datagram packet is sent by using the Datagram Socket dgc that has already been created inside the mmServer class.

### 5.5.5.4 Streaming Media Client

In this thesis, the streaming client functionality has been provided by the mmClient implemented class. Unlike the streaming server, the client in this thesis is implemented with the specific audio file format “.wav” only because of the android platform limitation. As this is implemented on Google Android platform, so the popular audio formats it supports for playback are mp3, ogg, wav and video formats supported are 3gp and mp4. But so far Android provides streaming playback
5.5. **RTSP Implementation over REST**

```java
public void sendRTPpacket(int mark, int sampleTime, byte[] msg, int numBytes,
SocketAddress addr){
    try {
        RTPacket rtp_packet = new RTPPacket(PAYLOAD_TYPE, Seqvalue, Ssrc,
mark, sampleTime, msg, numBytes);

        //get total length of the full rtp packet to send
        int packet_length = rtp_packet.getLength();

        //retrieve the packet bitstream and store it in an array of bytes
        byte[] packet_bits = new byte[packet_length];
        rtp_packet.getPacket(packet_bits);

        //making of UDP datagram packet & encapsulate RTP packet as byte array
        DatagramPacket datagram = new DatagramPacket(packet_bits, packet_bits.length, addr);

        //sending of UDP datagram packet towards client
        dgc.send(datagram);
    } catch (Exception ex) {
        ...
    }
}
```

*Figure 5.32: Sending of RTP Packet*

functionality only for *wav* audio format to play the buffered chunks of the file. The API provided for it is “AudioTrack”.

The main function implemented inside the `mmClient` class is `ClientConnection()` which is actually taking the help of other small functions for the receiving of the RTP packets and then play back the media content contained in those packets. Inside the `ClientConnection()`, there are actually two threads. In one thread, it is calling another function `recvRTPpacket()` which is actually receiving the UDP datagrams, separates out the RTP packet and provides the media content from the RTP packet for the playback. In the other thread, it is actually playing back the so far buffered RTP packets by using the `AudioTrack` API. So by this way, the new RTP packets are still received at the client while in parallel, the previously received RTP packets are being played out at the client. Thus the concept of multimedia streaming is supported i.e. *playing out one packet, processing the second one and receiving the third packet*. The `recvRTPpacket()` code snippet has been shown in the figure 5.33.

In this function, first the buffer size has been set at the Datagram Socket to receive the UDP datagram packet. As in the previous discussion, it was told that the maximum RTP payload size is of 1460 bytes. So in this function, `chunk_length` has the value of 1460 and in this thesis, the RTP Header of 12 bytes has been implemented. So the size required for the datagram packet to be received at the Datagram Socket of the client is `chunk_length + 12` i.e. `1460 + 12 = 1472`. Then the UDP datagrams are continuously received inside the `while` loop. For each packet, the content is checked that whether it is any *notification* datagram like “KeepAlive” or not. If the received datagram is found not for any notification, then it is considered that UDP datagram contains the valid RTP packet. So by using the constructor of implemented `RTPacket` class as per discussed in Figure 5.9, RTP packet will be decapsulated from the received UDP datagram. Then by using
public void recvRTPpacket() {
    ...
    while (true) {
        // Making buffer for UDP payload of size equal
        // to RTP Payload (chunk_length) + RTP Header (12)
        int size = chunk_length + 12;
        byte[] recv = new byte[size];
        datagram = new DatagramPacket(recv, size);
        while (true) {
            dgc.receive(datagram); // Receiving of UDP Datagram containing RTP packet
            ...
        }
        totalPacketsOfFile++; // Buffered number of RTP packets
        // Decapsulate RTP Packet from received UDP datagram
        RTPpacket rtp_packet = new RTPpacket(datagram.getData(), datagram.getLength());
        // Separating out the RTP payload from RTP packet
        int payload_length = rtp_packet.getpayload_length();
        final byte[] RTPpayload = new byte[payload_length];
        rtp_packet.getpayload(RTPpayload);
        // Checking out the Marker value
        chkMark = rtp_packet.getMarker();
        // Media content contained as RTP payload is added to Dynamic Buffer
        for (int j = 0; j < RTPpayload.length; j++) {
            dataBuffer.add(RTPpayload[j]);
        }
        packetsCollected++;
        if((packetsCollected >= (playbackBufferSize) && writeOnFile == false)|| (chkMark == 1)) {
            // Preparing Static buffer which will be shared between
            // RTP packets receiving thread and Playing back thread
            data2Write = new byte[dataBuffer.size()];
            // Transferring of so far stored data in the dynamic buffer to
            // static buffer for play back
            for (int i = 0; i < dataBuffer.size(); i++) {
                data2Write[i] = dataBuffer.get(i);
            }
            // Making dynamic buffer empty
            dataBuffer.clear();
            ...
            writeOnFile = true;
        }
        if (chkMark == 1) { // showing last RTP packet received successfully
            ...
        }
    }
}

Figure 5.33: Receiving of RTP Packets at Client
another function getpayload() of RTPpacket class as per shown in Figure 5.11, the RTP payload (which is actually the media data) that is filled up in the byte array buffer RTPpayload. After that, the value of the Marker field is retrieved from the RTP Header of the received RTP packet by using another RTPpacket class function getMarker(). Then the RTP payload of this specific received RTP packet is added to the dynamic buffer of bytes dataBuffer which is storing the data of all the RTP packets that has already been received before and not yet played back. packetsCollected tells the number of RTP packets whose data is currently buffered by dataBuffer. After that, it is checked that whether the packetsCollected is equal to or more than the constant playback buffer size playbackBufferSize and currently Audio player is not playing any buffered content by checking writeOnFile boolean value. Other than this, it is checked that whether the received RTP packet is the last one by comparing “1” with the value of chkMark which is containing the Marker value. If either of the condition becomes true, all the buffered data that has been stored by the dynamic buffer dataBuffer, it is transferred to another buffer of constant size data2Write and the dynamic buffer is made empty. Note that data2Write is the common buffer which has been shared by the two threads with the help of the boolean key writeOnFile. When the “writeOnFile=false”, then the thread which is receiving the media packets fill up the data2Write with the so far buffered media data that has been stored by dataBuffer and make “writeOnFile=true”. Then the AudioTrack player API (which is running in another thread) will play out all the media that is available in data2Write and when the play out is completed, it makes “writeOnFile=false” again. The code for playing back the media content is shown in the Figure 5.34.

Here in this thread, first intSize is configured to set AudioTrack buffer size according to the encoding format of the original source file which is streamed by the media server and now to be played back on the client. The three most important parameters which are required for its configuration are AT_samplingRate, AT_channelConfiguration and AT_PCM_encodingType to adjust the Sampling rate, Channel Configuration and Encoding Type. These three parameters are set by using the three functions which will adjust the values of these parameters according to the Media Attribute named as X-Ma (as discussed under heading of Media Description under section 2.4.1) which will be provided by SDP retrieved in the DESCRIBE request.

First function set_AT_samplingRate() shown in Figure 5.35 will set the AT_samplingRate according to the value of sampRate which is taken as input argument.

Second function set_AT_channelConfiguration() shown in Figure 5.36 will set the AT_channelConfiguration according to the value of channelConfig which is taken as input argument. If the “channelConfig=1”, it means that source file is of “Mono” type and will be adjusted as AudioFormat.CHANNEL_CONFIGURATION_MONO. But if the “channelConfig=2”, then the source file is of “Stereo” type and will be adjusted as AudioFormat.CHANNEL_CONFIGURATION_STEREO.

Third function set_AT_PCM_encodingType() shown in Figure 5.37 will set the AT_PCM_encodingType according to the value of encodeType which is taken as input argument. If the “encodeType=PCMwav8”, it means that source file is encoded with “8” bits per sample and will be adjusted as AudioFormat.ENCODING_PCM_8BIT. But if the “encodeType=PCMwav16”, then the source file is encoded with “16” bits per sample and will be adjusted as AudioFormat.ENCODING_PCM_16BIT.

Now coming back to the function shown in Figure 5.34, the buffer size for the AudioTrack has been set. Now AudioTrack object at is created in Streaming Mode shown by AudioTrack.MODE_STREAM and configured with buffer size intSize. Now this at is put in the “PLAY” state by using at.play(). Then by using at.write, the byte array contents can be written on the sound producing hardware to listen the media content. So as discussed above, if “writeOnFile == true” then all
Thread tt = new Thread() {

    public void run() {

        // Setting "AudioTrack" buffer size Configuration
        int intSize = android.media.AudioTrack.getMinBufferSize(AT_samplingRate,
        AT_channelConfiguration,AT_PCM_encodingType);
        // Configuring the "AudioTrack" & Set it into "Streaming Mode"
        at = new AudioTrack(AudioManager.STREAM_MUSIC, AT_samplingRate,
        AT_channelConfiguration,AT_PCM_encodingType, intSize, AudioTrack.MODE_STREAM);
        // Setting the "AudioTrack" to PLAY state
        at.play();
        while (true) {
            if (checkk == false) {
                if (writeOnFile == true) {
                    // Writing media contents on the audio hardware
                    at.write(data2Write, 0, data2Write.length);
                }
                if (fileWrittenComplete == true) { // Writing media of
                    // last received RTP packets
                    at.write(data2Write, 0, data2Write.length);
                    // Setting the "AudioTrack" to STOP state when streaming data finished
                    at.stop();
                    // Relasing all the play back resources of "AudioTrack"
                    at.release();
                    play = true;
                    break;
                } else {
                    ...
                    writeOnFile = false;
                    ...
                }
            }
        }
    }
};

    tt.start(); // Starting the Play Out thread

Figure 5.34: Playing Out of the media content by AudioTrack

static void set_AT_samplingRate(int sampRate) {
    AT_samplingRate = sampRate;
}

Figure 5.35: Setting of Sampling Rate
static void set_AT_channelConfiguration(int channelConfig) {
    if (channelConfig == 1) {
        AT_channelConfiguration = AudioFormat.CHANNEL_CONFIGURATION_MONO;
    } else if (channelConfig == 2) {
        AT_channelConfiguration = AudioFormat.CHANNEL_CONFIGURATION_STEREO;
    }
}

Figure 5.36: Setting of Channel Configuration

static void set_AT_PCM_encodingType(String encodeType) {
    if (encodeType.equalsIgnoreCase("PCMwav8")) {
        AT_PCM_encodingType = AudioFormat.ENCODING_PCM_8BIT;
    } else if (encodeType.equalsIgnoreCase("PCMwav16")) {
        AT_PCM_encodingType = AudioFormat.ENCODING_PCM_16BIT;
    }
}

Figure 5.37: Setting of PCM Encoding Type

the media data which is available in data2Write buffer can be written on the sound hardware by using the write function of the AudioTrack. If all the streaming media has been finished and last RTP packet is received as checked by the condition “fileWrittenComplete = true”, then AudioTrack can be put to “STOP” state by using at.stop() and all the play back resources reserved by the AudioTrack can be released by using at.release(). Otherwise, the play back will be continued by putting “writeOnFile = false” when all the buffered data available on data2Write has been played back by the AudioTrack successfully.

5.5.5 Multimedia Streaming in Scenario I

Now the finalized working of the implemented RTSP based multimedia streaming will be discussed against the different RTSP requests. In the scenario I, there will be direct socket based connection established between the client and the media server where the server is available with the public transport address. This socket connection will be established by each request and will be broken once the response of that request is received at the client. So by this, there is no need to keep the socket connection alive by the help of “Keep-Alive” messages. The streaming of UDP datagrams with the RTP packets containing the media data will also be delivered directly from the server to the client. In our implementation, the scenario I is successful on the “T-Mobile EDGE” network which is giving the public dynamic IP address and public dynamic UDP port, but for TCP connection, the public port is fixed at “8080”. The behavior against each RTSP request for Scenario I will be discussed as follows:

OPTIONS/DESCRIBE

For both OPTIONS/DESCRIBE requests, socket connection will be established with the media server having the public transport address i.e. 88.x.x.x:8080 as shown in Figure 5.38. The client may lie behind any NAPT/Firewall. As the socket connection is initiated from behind the NAPT/Firewall on the outgoing TCP connection, so by the TCP hole punching concept, the response for OPTIONS/DESCRIBE can be received successfully at the client. For the OPTIONS request,
the response will carry all the options available that are required for implicit-play and names of
the media files available for the streaming while for the DESCRIBE request, SDP can be received
inside the response. Note that this socket connection which was established with the server is
only available upto the receiving of the response. As soon as the response is received, the socket
connection will no longer be available between client and server.

\textbf{SETUP}

For SETUP request, again the socket connection will be established with the media server at the
public transport address i.e. 88.x.x.x:8080 as shown in Figure 5.39. When the SETUP request
will be received at the server, the server state is changed from \textit{INIT} to \textit{READY}. Just like discussed
for the DESCRIBE request, the response of the SETUP request will be received successfully at
the client and the socket connection will no longer be alive after that. However, SETUP request
also involves the establishment of streaming channel based over UDP. So to establish that, UDP
datagram containing the “cnhhello” will be sent towards the media server at the same public IP
address used for the TCP connection but on different public port i.e. 88.x.x.x:9094. This packet
will punch the hole inside its own NAPT by using the \textit{UDP Hole Punching} technique and make a
UDP tunnel with the server. Now the media server will record the public mapped transport address
of the client and then send the UDP response datagrams containing the “KeepAlive” periodically
to keep the UDP tunnel alive which has been established between the client and server. With the
help of punched hole, the datagrams are received successfully at the client.

\textbf{PLAY}

For the PLAY request, again the socket connection will be established with the media server at the
public transport address 88.x.x.x:8080 and the server state is changed from \textit{READY} to \textit{PLAYING}. 
5.5. **RTSP Implementation over REST**

Now the server will send the UDP datagrams (which are carrying the RTP packets with the media content) over the UDP tunnel that has already been established by the SETUP request before and was kept alive by periodic UDP keep-alive packets. Thus the multimedia content can be streamed towards client which can play out the buffered media content.

![Figure 5.40: PLAY request & response for Scenario I](image)

**PAUSE**

For the PAUSE request, its socket connection will be established again with the media server at public transport address over which the request will be sent to the server and the corresponding response will be received as shown in Figure 5.41. When the PAUSE request is received at the server, it will change its state from **PLAYING** back to **INIT** and again start sending periodic UDP “KeepAlive” packets over the UDP tunnel between the client and server to keep it alive. Thus the streaming of the media data has been paused but these periodic packets are important because once the UDP tunnel is broken, it is very difficult to retrieve the mapped public port of the NAPT/Firewall which was allocated during the SETUP request. When the successive PLAY request will be sent again, the server will change its state back to **PLAYING** and the server will again start sending UDP datagrams with the RTP packets over UDP tunnel and thus streaming is resumed back again.

![Figure 5.41: PAUSE request & response for Scenario I](image)

**TEARDOWN**

For the TEARDOWN request, again the socket connection will be established with the media server over which the TEARDOWN request will be sent and the corresponding response will be received as shown in Figure 5.42. When the TEARDOWN request will be received at the server, the server will change its current state to **INIT**. On the UDP tunnel, it will finally send the last UDP datagram towards the client containing “EndUDPConn” to show that streaming has been stopped in the middle. After that, there will be no more UDP packets that will be sent over UDP.
tunnel. Thus UDP tunnel becomes idle and after timeout (which is default configured on the NAT by administrator), the tunnel will be broken and the public mapped port that was allocated for this particular UDP session will no longer be available.

![Figure 5.42: TEARDOWN request & response for Scenario I](image)

### 5.5.5.6 Multimedia Streaming in Scenario IIa

For the Scenario IIa, the signaling will be done by establishing the two TCP based socket connections with the IAG. One permanent TCP tunnel will be established by the server with the IAG. However, at the client side, the socket connection will be established by each request and will be broken down once the response is received by the client against that request. So by this way, only one socket connection is needed to be kept alive by using the periodic TCP keep-alive messages. For the streaming channel, the direct UDP based channel will be established between the media server and the client by using the *UDP Hole Punching* and *Connection Reversal* techniques as discussed in Figure 5.25 before. In our implementation, Scenario IIa is successful when the client is behind *Full Cone NAT* and we have got Full Cone NAT type public port mapping property on “eduroam WLAN”. The behavior against each RTSP request for Scenario IIa will be discussed as follows:

**OPTIONS/DESCRIBE**

Here before sending of any request by the client, the TCP tunnel will be established by the server with the IAG at its fixed public IP address 137.226.5.151 and fixed public port 9091. Now the client will establish another socket connection with IAG at same public IP address 137.226.5.151 but different public port 9090 over which OPTIONS/DESCRIBE will be sent towards IAG as shown in Figure 5.43. As the TCP tunnel that was established by the server, it was initiated by the server from behind the NAPT/Firewall over the outgoing TCP connection, so when IAG will relay the OPTIONS/DESCRIBE request towards server, it will be successfully received by the server. Similarly when the server will send the response of this relayed OPTIONS/DESCRIBE towards IAG, it will be relayed successfully towards the client because of the socket connection (that was established by the client with IAG) was also initiated by the client from the inside of NAPT/Firewall on outgoing TCP connection. When the response is received by the client, then the socket connection between the client and IAG will be closed and will no longer be available. However, the TCP tunnel between the server and the IAG will remain active and available.
Figure 5.43: OPTIONS/DESCRIBE request & response for Scenario IIa

SETUP

For the SETUP request, the socket connection will again be established by the client with IAG at its fixed public IP address 137.226.5.151 and fixed public port 9090 as shown in Figure 5.44. Now the SETUP request will be successfully relayed towards the server over the TCP tunnel between the IAG and the server and corresponding response will be received after the change of the server state from INIT to READY state. The socket connection between the client and IAG will now be closed after receiving of the response at client. Now to establish the direct UDP channel between the client and the server for streaming of media content, connection reversal technique will be used. First the client will send the UDP datagram with “cnhello” towards IAG at fixed public transport address i.e. 137.226.5.151:9092. It will punch a hole inside the client’s NAT/Firewall and the UDP tunnel will be established between client and the IAG. IAG will record the public mapped transport address of the client from this received packet. Then another UDP datagram with “hello” will be sent by the server, which will punch a hole inside its NAPT/Firewall and another UDP tunnel will be established between the server and the IAG. Now over this tunnel, IAG will send the UDP datagram towards the server which is carrying the public mapped transport address of the client. Then the server will send the UDP datagram containing the “KeepAlive” towards the client and as the client’s NAPT has Full Cone consistent port mapping property, so this packet will be successfully received at the client causing another direct UDP tunnel establishment between the server and the client. As the server is in READY state, so it will send these “KeepAlive” UDP datagrams periodically to keep the UDP tunnel alive that was previously established between the server and the client. However the other two UDP tunnels that were established between client-IAG and between server-IAG, they are no longer active and will be broken down after timeout.

PLAY

For the PLAY request, the socket connection will be again established by the client with IAG at its fixed public IP address 137.226.5.151 and fixed public port 9090 as shown in Figure 5.45. When the request is relayed to the server, its state is changed from READY to PLAYING and the response is sent back to the client via IAG followed by closure of socket connection between the client and IAG. Now the server will start streaming the UDP datagrams with RTP packets (that are containing the media data) over the UDP tunnel that was established directly between the server and the client against the SETUP request.
PAUSE

For the PAUSE request, socket connection will be established again between the client and the IAG as described for SETUP and PLAY requests and shown in Figure 5.46. Now when the request is relayed to the server by the IAG, the server will change its state from PLAYING to READY and will start to send the periodic UDP datagrams containing the “KeepAlive” towards the client to keep the UDP tunnel alive which was established directly between the server and the client. Thus streaming of the media content via RTP packets (encapsulated inside UDP datagrams) has been paused. When the subsequent PLAY request is received by the server, its state will be changed back to PLAYING and then again the UDP datagrams with the RTP packets will be streamed by the server towards the client over the directly established UDP tunnel.

TEARDOWN

For the TEARDOWN request, socket connection will be established again by the client and the IAG as it was established for previous RTSP requests and shown in Figure 5.47. When the request is relayed to the server by the IAG, server will change its current state to INIT and response will be sent back to the client via IAG and the socket connection between client and IAG will be closed. Now the server will just send the last UDP datagram containing the “EndUDPConn” over the UDP tunnel available directly between the server and the client. This UDP datagram will tell the client...
5.5. RTSP Implementation over REST

5.5.5. Multimedia Streaming in Scenario IIa

For the Scenario IIb, the signaling will be done in the same way as described for the Scenario IIa in Section 5.5.5.6. There will be permanent TCP tunnel that has been established by the server with the IAG which will be kept alive by periodic TCP Keep Alive messages. However, client will establish socket connection separately for each RTSP request with IAG which will be closed after receiving of the corresponding response at the client. Unlike the Scenario IIa, in this scenario, the streaming will also be done via IAG. For that both client and server will establish their UDP tunnels with IAG over which the streaming UDP datagrams with RTP packets (containing the media data) will be sent by the server to the IAG over its UDP tunnel from where it is relayed to client over the UDP tunnel that has been established between the IAG and the client. In our implementation, the scenario is successful for almost every configuration of server and the client. The behavior against each RTSP request for Scenario IIb will be discussed as follows:
OPTIONS/DESCRIBE

Just like discussed for the Scenario IIa, before sending of OPTIONS/DESCRIBE request, constant TCP tunnel will be established by the server with the IAG at its public IP address 137.226.5.151 and fixed public port 9091 which will be kept alive by periodic Keep-Alive messages. For the OPTIONS/DESCRIBE request, the socket connection will be established by the client with the IAG at its same public IP address 137.226.5.151 but different TCP port 9090 as shown in Figure 5.48. Over this socket connection, OPTIONS/DESCRIBE request will be sent towards IAG from where it is relayed to the server over the TCP tunnel that has already been established between the IAG and the server. Then the response is sent to the client via IAG followed by the closure of socket connection between the IAG and client.

![Figure 5.48: OPTIONS/DESCRIBE request & response for Scenario IIb](image)

SETUP

For the SETUP request, the socket connection will be established again with the IAG at its public IP address 137.226.5.151 and fixed public port 9090 as shown in Figure 5.49. When the request is relayed towards the server via IAG, server will change its state from INIT to READY and the response will be sent back to the client via IAG which will be followed by closure of socket connection between the client and the IAG. Now for the streaming, first UDP datagram containing “cnhhello” will be sent by the client towards IAG at its public transport address 137.226.5.151:9092. It will punch a hole inside the client’s NAPT/Firewall and UDP tunnel will be established between the client and the IAG. Then server will send another UDP datagram containing “hello” at the same fixed public IP address 137.226.5.151 but different UDP port 9093. It will punch a hole inside the server NAPT/Firewall and another UDP tunnel will be established between the server and the IAG. Now as the server is in READY state, so it will send the UDP datagram with “KeepAlive” towards IAG over the tunnel between the server and the IAG which will be relayed to the client successfully over the UDP tunnel which has been established between the IAG and the client. Server will send these UDP keep-alive datagrams periodically towards client to keep the two UDP tunnels active until the successive PLAY request will be generated.

PLAY

For the PLAY request, again the socket connection will be established by the client with the IAG as it was established in the previous SETUP request and is shown in the Figure 5.50. When the
5.5. **RTSP Implementation over REST**

The **SETUP** request is relayed to the server via IAG, it will then change its state from **READY** to **PLAYING** and send the response to the client via IAG. Now as the server state is changed to **PLAYING**, it will then start sending the UDP datagrams with the RTP packets (which are containing the media data) towards the IAG over the UDP tunnel that is established between the server and the IAG. These UDP datagrams with RTP packets will then be relayed towards the client over the UDP tunnel that has been established between the client and the IAG. So these RTP packets will then be received successfully at the client where their media data can be played back.

![Figure 5.49: SETUP request & response for Scenario IIb](image)

The **PLAY** request, the socket connection will be established in the same way as it was established for the SETUP and PLAY requests previously and is shown in the Figure 5.51. When the request is received at the server, it will change its state back to **READY** and the response will be sent back to the client via IAG. Now the server will start sending the UDP datagrams with “KeepAlive” again periodically over the two UDP tunnels to keep them alive and thus multimedia streaming is being paused in the middle. When any subsequent PLAY request will be received at the server, it will change its state back to **PLAYING** and again start to send the UDP datagrams with RTP packets towards client via IAG and thus streaming of media data will be resumed again.

![Figure 5.50: PLAY request & response for Scenario IIb](image)

**PAUSE**

For the **PAUSE** request, the socket connection will be established in the same way as it was established for the SETUP and PLAY requests previously and is shown in the Figure 5.51. When the request is received at the server, it will change its state back to **READY** and the response will be sent back to the client via IAG. Now the server will start sending the UDP datagrams with “KeepAlive” again periodically over the two UDP tunnels to keep them alive and thus multimedia streaming is being paused in the middle. When any subsequent PLAY request will be received at the server, it will change its state back to **PLAYING** and again start to send the UDP datagrams with RTP packets towards client via IAG and thus streaming of media data will be resumed again.
TEARDOWN

For the TEARDOWN request, the socket connection will be established again in the same way as it was established for the previous SETUP, PLAY, PAUSE etc requests and is shown in the Figure 5.52. When the request is received at the server, it will change its current state to INIT again and the response will be sent back towards the client via IAG and the socket connection between IAG-client will be closed. Now the server will send the last UDP datagram containing “EndUDPConn” towards the client via IAG which will show that multimedia streaming has been stopped and there will be no more corresponding UDP datagrams that will be sent from the server towards the client. As there will be no further activity over the UDP tunnels (that were established between the client-IAG and between server-IAG), so they both will be broken after timeout.
Evaluation and Performance of the M2M and IAG Scenarios

This chapter focuses on the evaluation and performance of different multimedia streaming scenarios which are designed in this thesis, i.e. Scenario I of direct M2M streaming and Scenario II of streaming using the IAG. First, the scenarios will be evaluated on the basis of request-response delay in signaling channel, and the media data packet latency in streaming channel. Then, their performance over different live operator networks will be discussed. Finally, the recommendations will be provided for the possible extensions that are required for the improvement of the implemented functionality.

6.1 General Constraints of the Scenarios

In this thesis, there are two basic scenarios which have been implemented. In the Scenario I, there is a direct multimedia streaming between two mobile nodes. While in the Scenario II, the multimedia streaming has been carried out between two mobile nodes with the help of IAG. However, both the scenarios have some constraints, which make them significantly distinct from each other.

6.1.1 Constraints of Scenario I

In the scenario I, there is direct streaming of multimedia content from the media server towards the media client. There is no intermediate node which is involved in the middle for carrying out this M2M multimedia streaming. But, it has some constraints due to which it cannot be considered as an ideal solution. These constraints are mentioned as follows:

Publicly Accessible Media Server: For the Scenario I to work, the media server should be available on the public Internet. It should have globally unique and known IP address and port by which any client (which may reside behind any NAPT/Firewall) can access it. If the IP address and Port of the media server are not globally known, then clients cannot access the server and multimedia streaming cannot be carried out.

NAPT/Firewall Reservations: As discussed in the previous constraint that for the Scenario I to work, it should have globally unique IP address and port. This means that media server should not reside behind any NAPT/Firewall, otherwise, it cannot be accessible by client over the public internet and multimedia streaming will not be initiated between the media server and the client.

6.1.2 Constraints of Scenario II

In the scenario II, there is an IAG which is acting as an intermediate node between the server and the client. In the Scenario IIa, the streaming between two mobile nodes has been initiated with the help of IAG, while, in the Scenario IIb, the streaming has been done via IAG from the media server
to the media client. Although, the Scenario II can be a source towards the generalized solution, but it is obtained at the cost of additional overheads some of which are mentioned as follows:

**Maintenance of constant TCP tunnel:** For this Scenario II to work as a generalized solution and make any media server (which may be behind NAPT/Firewall) to be accessible, a constant TCP tunnel has to be established first between the media server and the client over which the signaling messages will be relayed. To bypass the effect of NAPT/Firewall, this tunnel has to be initiated by the media server from behind the NAPT/Firewall as the outgoing TCP connection. Secondly, to keep this tunnel alive, the periodic keep alive messages have to be sent by the media server towards the IAG over this TCP tunnel.

**Additional point of failure:** As, in the Scenario II, the IAG is an extra node in addition to the media server and the media client that is required as an intermediate helping node. So, for our implemented model to malfunction, its failure can be one of the significant reason. In other words, it can be an additional point of failure. So, if this IAG is failed, then signaling messages cannot be relayed between media server and the client. Secondly, streaming either cannot be initiated in the Scenario IIa or cannot be continued in the Scenario IIb.

### 6.2 Inter-scenario Evaluation

Upto now, we have evaluated the two scenarios on the basis of their generality. Now, these scenarios will be evaluated by the help of different tests that have been conducted on the live network. For these tests, two Android mobile phones are acting the media client and the media server, while, the IAG is the work station connected with the high speed Internet. One of the mobile phones is the original Google Android supporting handset, while, the other phone is actually the Android emulator that is running on the notebook. The configurations for these two mobile phones are given as follows:

**Configuration of the original Google phone handset:** The configuration of the Android handset is:

- **Manufacturer:** Sony Ericsson
- **Model:** Sony Ericsson Xperia X10i
- **CPU:** Qualcomm QSD8250 Snapdragon 1 GHz processor
- **Memory:** Internal 1 GB storage, 384 MB RAM microSD up to 32GB, 8GB card included
- **Operating System:** Android OS 2.1
- **DATA Network Support:** EDGE Class 10, 236.8 kbps 3G HSDPA, 7.2 Mbps; HSUPA, 2 Mbps

**Configuration of the Notebook for android emulator:** The configuration of the notebook PC on which the android emulator is running is:

- **System Manufacturer:** Lenovo IBM
- **Model:** ThinkPad SL510
- **Processor:** Intel Core 2 Duo T6570
6.2. Inter-scenario Evaluation

- **Speed**: 2.10 GHz
- **Memory**: 4.00 GB
- **Operating System**: Genuine Windows 7 Professional 64-bit

### 6.2.1 Comparison of Request-Response Delay

In this subsection, the delay for different requests and their received responses will be compared on the signaling channel among the three different Scenarios I, IIa and IIb. This delay includes all the times i.e.

- The time required for the generation and transmission of request at the client
- The time required for the propagation of request from client to media server (directly or via IAG) over the network
- The time required for the service of request and generation/transmission of response at the media server
- The time required for the propagation of response from media server to client (directly or via IAG) over the network.

To compute the request-response delays for different scenarios, the media server is run on the original Google android handset while the client is running in the emulator on the notebook. The media server is connected to the T-Mobile EDGE network because we are getting the dynamic public IP address and public port only on this network, which is required for the Scenario I to work. The client is connected on the eduroam WLAN.

#### 6.2.1.1 Sequence Flow Diagram for Scenario I

The sequence flow diagram for the Scenario I is shown in Figure 6.1. In this diagram, the Total Request-Response Delay $T_{Req-Res}$ at the client has been computed, which includes all the times of request-response transmission, propagation and service times at the media client and server respectively. The distribution of all the times that are involved in $T_{Req-Res}$ is shown in the Equation 6.1.

$$T_{Req-Res} = T_{trans1} + T_{prop1} + T_{service} + T_{trans2} + T_{prop2}$$  \hspace{1cm} (6.1)

In this Equation 6.1, the $T_{trans1}$ and $T_{trans2}$ are the transmission times taken by the request and the corresponding response at the client and server, respectively. Then, the $T_{prop1}$ and $T_{prop2}$ are the times taken by the request to reach the server and the corresponding response to reach client over the network, respectively. The $T_{service}$ is the actual time taken by the server to process the received request.

#### 6.2.1.2 Sequence Flow Diagram for Scenario II

The sequence flow diagram for Scenario II is shown in Figure 6.2. In this diagram, the Total Request-Response Delay $T_{Req-Res}$ at client is computed in the same way as it was for Scenario I before, but with the addition of the processing delays that has been introduced by the intermediate IAG during relaying of requests and corresponding responses between the client and server.
6. Evaluation and Performance of the M2M and IAG Scenarios

The distribution of all the times that are involved in the $T_{\text{Req-Res}}$ for Scenario II is shown in the Equation 6.2.

$$T_{\text{Req-Res}} = T_{\text{trans1}} + T_{\text{prop1}} + T_{\text{process1}} + T_{\text{trans2}} + T_{\text{prop2}} + T_{\text{service}} + T_{\text{trans3}} + T_{\text{prop3}} + T_{\text{process2}} + T_{\text{trans4}} + T_{\text{prop4}}$$

(6.2)

Here, in Equation 6.2, the $T_{\text{trans1}}$ is the transmission time taken by the request at the client and the $T_{\text{prop1}}$ is the prorogation time for the request to be received at the IAG over the network path. Then, the $T_{\text{process1}}$ is the processing time taken by the IAG to prepare the request that is received from the client for relaying it to the server. The $T_{\text{trans2}}$ and $T_{\text{prop2}}$ are times taken for the transmission and propagation of the relayed request sent by the IAG to the server, respectively. After that, the $T_{\text{service}}$ is the actual service time taken by the server to process the request received from the client via the IAG. Then, after processing of the request, the $T_{\text{trans3}}$ and $T_{\text{prop3}}$ are the transmission and propagation times of the response at the server to the IAG over the network path. Then, the $T_{\text{process2}}$ is actually the processing time taken by the IAG to prepare the response that is received from the server for relaying it towards the client over the signaling channel. Finally, the $T_{\text{trans4}}$ and $T_{\text{prop4}}$ are the times for transmission and propagation of relayed response sent by the IAG to the client.

6.2.1.3 Service Times for OPTIONS and DESCRIBE requests

As mentioned earlier in this thesis, the OPTIONS and DESCRIBE requests-responses have been implemented by using the synchronous REST architecture because these are the short-lived services that will run only once. Now, whenever this synchronous OPTIONS or DESCRIBE is sent by the client, it will remain in the blocked state until the response is received from the server at the client after completion of the synchronous service. So, the service times of these synchronous services will play a significant role in the request-response delay for these synchronous service requests.

The service times for the OPTIONS and DESCRIBE in both Scenarios I and II will remain the same and has been shown in Figure 6.3. Note that here OPTIONS average service time is $62 \text{ msec (approx)}$ which is almost the half as taken by the DESCRIBE synchronous service i.e. $123 \text{ msec.}$
But this will not be the case all the time. Here, the OPTIONS service is executed when only there are 4 available files for streaming at the SD card of the server and the generated XML payload (which is containing the names of files available for streaming) size was 121 bytes. In contrast to that, the XML payload generated by the DESCRIBE service for session description, its size was 395 bytes. So, the service times for both these services depend on the time taken by the generation of the required XML data. For example, if the number of files available for streaming is increased, or the file names are longer, then it may be possible that at sometime, XML data generated by the OPTIONS service becomes higher than that of the DESCRIBE service. One more important thing is that, this service time is independent of any specific scenario which has been implemented in this thesis, because it is actual the service time taken by the service and will depend on the processing speed of the original node on which the server is running.

**6.2.1.4 OPTIONS Request-Response Delay**

Now, the inter-scenario comparison for the OPTIONS request-response delay is measured for all the three Scenarios I, IIa and IIb, and has been shown in Figure 6.4. In this figure, it can be seen that request-response delay time for the Scenario IIa and IIb are almost same. This is because both these scenarios have same phenomena for the OPTIONS request-response. But, the strange thing which can be seen in this graph is that the Scenario I has the highest request-response delay time as compared to the other two Scenarios IIa and IIb. However, theoretically it should be less than Scenario II which involves extra processing and transmission times for both requests and responses to be relayed by IAG. The possible reason for this result can be that our IAG has actually a high speed processor and it is connected with the high speed Internet (Ethernet), so, the propagation times taken by both the OPTIONS request to reach the IAG from the client and the OPTIONS response to reach at the client from the IAG is so small that it has overcome all the processing and transmission time delays that are involved at the IAG. Our further testing shows that in Scenario II, propagation times taken by the OPTIONS relayed request-response i.e. $T_{prop2}$...
and $T_{prop3}$, over the cellular wireless channel between IAG and the server is almost twice/thrice to the propagation times taken by the OPTIONS request-relayed response i.e. $T_{prop1}$ and $T_{prop4}$ over the Internet channel between client and the IAG. The processing times i.e. $T_{process1}$ and $T_{process2}$, and the transmission times i.e. $T_{trans2}$ and $T_{trans4}$ at the IAG are to small to be negligible i.e. 1 or 2 msec each. Secondly, in Scenario II, the IAG is at much faster connection of wired LAN which makes socket connection with the media server in shorter time, than the wireless connection of eduroam WLAN in Scenario I. Thus, the resultant affect is that Scenario II has offered less total request-response delay $T_{Req−Res}$ as compared to that of Scenario I as shown in Equation 6.3.

$$T_{Req−Res(ScenarioII)} < T_{Req−Res(ScenarioI)}$$  \hspace{2cm} (6.3)

![OPTIONS vs DESCRIBE](image.png)

**Figure 6.3:** Service Times for OPTIONS and DESCRIBE

![OPTIONS Request-Response Time](image.png)

**Figure 6.4:** Request-Response Delay of OPTIONS
6.2. Inter-scenario Evaluation

6.2.1.5 DESCRIBE Request-Response Delay

Just like the OPTIONS, now the request-response delay for DESCRIBE is measured for inter-scenario comparison and is shown in the Figure 6.5. Just like we have seen for the OPTIONS, here Scenario IIa and Scenario IIb again almost have same delay time because this service is requested in the same way as was done for OPTIONS. However, the unexpected thing is again that the request-response delay for Scenario I is higher as compared to Scenario II. Its reason can be same as explained for the OPTIONS. However, DESCRIBE has higher values of request-response delays in all Scenarios as compared to those that we have seen for the OPTIONS. This is due to higher service time taken by the DESCRIBE than that required by the OPTIONS as discussed in Section 6.2.1.3 before.

![DESCRIBE Request-Response Time](image)

**Figure 6.5:** Request-Response Delay of DESCRIBE

6.2.1.6 SETUP Request-Response Delay

Now, the request-response delay time for SETUP is measured for the inter-scenario comparison and is shown in the Figure 6.6. It can be seen from the graph that here the request-response delays for SETUP in all three scenarios are in thousands of milliseconds, while, in the previous OPTIONS and DESCRIBE, these delays were actually in hundreds of milliseconds. The possible reason can be that both OPTIONS and DESCRIBE requests have been sent for synchronous services which are short lived services and have less service times. But, this SETUP request is sent for asynchronous service, which takes more time to start without keeping the client in the blocked state. Here, from the graph, it can be seen that highest time is taken by the Scenario IIa which is not same to that of Scenario IIb. Secondly, the Scenario I has taken the lowest time. The reason for this behavior is that the SETUP is the only request as compared to all other requests which involves the establishment of data channel as well in order to process its request. So the Scenario IIa has the highest request-response time because it takes the most time for the initialization of its data channel to be established between the server and the client. The Scenario I has the lowest, because its data channel initialization takes the least time. This can be the dominant factor which makes the request-response time lowest for Scenario I, as compared to the Scenario II and also the difference between Scenario IIa and Scenario IIb.
6. Evaluation and Performance of the M2M and IAG Scenarios

6.2.1.7 PLAY Request-Response Delay

The inter-scenario comparison for PLAY request-response delay is shown in the Figure 6.7. Again, here the values are in the thousands of milliseconds because it is also sent for the asynchronous service. But, the notable thing here is that its trend and behavior of graph is similar to what we have seen for the synchronous OPTIONS and DESCRIBE requests. Here again, the Scenario IIa and IIb have almost the equal values because this request is set in the same way for the two scenarios. But, for the Scenario I, it is again higher as compared to Scenario II with the possible same reason as explained for OPTIONS and DESCRIBE request.

![Figure 6.6: Request-Response Delay of SETUP](image)

**Figure 6.6:** Request-Response Delay of SETUP

The important thing to mention here is that in this evaluation of request-response delay, for the PLAY request, there is only involvement of change in the current state of service Instance. Similarly, the remaining PAUSE and TEARDOWN requests also involves the change in the state of the
service instance which will almost have the same effect as shown by the single PLAY request here. That's why they are not evaluated explicitly for this comparison, and their trend is assumed to be similar as seen for the PLAY request.

6.2.2 Comparison of Streaming Packet Latency

Just like the inter-scenario comparison for the request-response delays on the signaling channel, now the scenarios will be evaluated on the basis of the latency on the streaming channel. For that, we have mainly two cases to be considered for this evaluation. In the first case, there is direct M2M streaming of multimedia data between the server and the client, as implemented for Scenario I and Scenario IIa. In the second case, the streaming of multimedia data is done via IAG i.e. the data packets are relayed between the media server and the client, as implemented for Scenario IIb.

6.2.2.1 Testbed Configuration

To carry out this evaluation, the media server is connected to the O2 HSPA network while the client is connected to the eduroam WLAN. The remaining configuration details are as follows:

- File Name: mzr.wav
- File Configuration Parameters: 11025/8/1
- Total File Size $N_{fileSize}$: 1432256 Bytes
- MTU on Network Path: 1500 Bytes
- RTP Payload Size $N_{RTPpayloadSize}$: 1460 Bytes
- Packet Dispatch Interval (PDI) at Server: 200 msec
- Expected Number of RTP packets: 981

In this mentioned configuration parameters, file configuration parameters i.e. 11025/8/1 tells that the source wave format audio file is encoded at the sampling rate of 11025 Hz with 8 bits/sample and “1” shows that it is single channel file i.e. Mono. Then, the MTU size is taken as 1500 bytes by checking it with the help of Wire Shark for the network path. Thus, the RTP payload size is taken as 1460 bytes which is the maximum size of RTP payload as mentioned before in the section 5.2.2. After that, the next important parameter is PDI which is the minimum delay required at the server between sending the streaming packets. This time is required for the proper synchronization of the media client and the media server, and for the reception of all streaming packets successfully from the media server to the media client. Finally, the expected number of RTP packets $N_{RTPpackets}$ are calculated by using the formula shown in the Equation 6.4.

$$N_{RTPpackets} = \frac{N_{fileSize}}{N_{RTPpayloadSize}}$$  \hspace{1cm} (6.4)

6.2.2.2 Total File Buffering Time

For the comparison of the two cases i.e. direct M2M streaming and streaming via IAG, total file buffering time have been measured. It is actually the total time required for all the total 981 RTP packets of mzr.wav audio file to be received successfully at the client from the media server.
The total file buffering time that has been measured for the mentioned configuration are shown in Figure 6.8 in msec, and 6.9 in sec.

From these graphs shown in Figure 6.8 and 6.9, it can be seen that the file buffering time when streaming via IAG is much higher than that of the direct streaming, i.e., about approximately 24 seconds more in streaming via IAG, and this difference can be increased if the source file size or number of RTP packets is increased. The obvious reason for this difference can be due to the long network path and the processing delay caused by the IAG.

The performance difference for these two streaming cases according to the current configuration has been summarized in Figure 6.10. From the figure, it can be seen that streaming via IAG is 1.14 times slower to that of direct M2M streaming. Although, in this configuration, there is not that much high fractional difference between the two streaming cases, but if the streaming source file is of much large size, then it can show significant fractional difference between the two streaming cases.
6.2. Inter-scenario Evaluation

6.2.2.3 Inter-Packet Delay Difference

In the previous discussion, we have seen the difference on the overall file buffering time for the two different streaming cases. Now, we will see the effect of streaming via IAG for the receiving time of every data packet. For this, the inter-packet delay difference has been measured for the two cases and has been shown in Figure 6.11. From this figure, it can be seen that the average difference of the arrival time between the two packets is about 206 msec (approx) for streaming via IAG while it is of about 205 msec (approx) for the direct M2M streaming case.

Thus, the overall combined overhead caused by the network path delay and processing delay of IAG in the case of streaming via IAG is only of 1.12 msec, as shown in Figure 6.12. The reason for this overhead to be very small can be due to the high speed processor of PC machine over which the IAG is running due to which the processing delay of the IAG is very small. Secondly, as the IAG is having a high speed and large bandwidth Internet connection, the network path delay becomes very small. Thus, again the combined effect due to the two delays does not cause that much significant impact for the single data streaming packet that is arriving on the client, but this
difference can become considerable on the overall file buffering time when the file size is big and the number of RTP packets that are going to be buffered at client is large.

![IAG Delay Overhead](image)

**Figure 6.12:** IAG Delay Overhead

### 6.3 Multimedia Streaming Performance over different networks

So far, we have evaluated the implemented scenarios for their performance in terms of the request-response delay on the signaling channel and data packet latency over the streaming channel. Now, in this subsection, we will discuss about the different performance issues with respect to the live cellular data networks like EDGE and HSPA.

In this thesis, we have basically investigated the two cellular operators i.e. “T-Mobile” and “O2” for mainly their EDGE and HSPA networks. Along with that, there is another network of “eduroam” WLAN.

#### 6.3.1 PDI for Uplink and Downlink

As we know from the previous knowledge that uplink data (which is the data sent from the mobile phone towards the base station tower) is usually less than the downlink data (which is the data sent from the base station tower to mobile phone). So, these data networks like HSPA are asymmetric in nature i.e. more resources are allocated on the downlink than on the uplink.

So, to confirm this, and see their impact on the file buffering time, we have first connected the mobile server with the O2 HSPA network and the client is connected on eduroam WLAN as shown in Figure 6.13.

So now, here server will send data over the uplink of cellular network which has certain amount of the limited shared bandwidth but the client is connected on WLAN which has much higher available bandwidth. Then, we performed our test by buffering all the media packets at the client by setting the particular value of PDI at the server which is the minimum intentional delay required between sending of one packet and waiting before sending the next one. Similarly, then the client is connected on the O2 HSPA network and the server is connected on the eduroam WLAN (as
6.3. Multimedia Streaming Performance over different networks

Figure 6.13: RTP Packet Streaming over Uplink

shown in Figure 6.14) to see the effect of file buffering on the downlink by setting the different value of PDI.

Figure 6.14: RTP Packet Streaming over Downlink

The two values of PDI which has been set for the uplink and downlink channels over the O2 HSPA with the fixed RTP payload size of 1460 bytes are shown in Figure 6.15. These values are chosen after conducting various tests and at these selected values, we have received all the RTP packets successfully at the client from the media server without the loss of even single RTP packet.

From the Figure 6.15, it can be seen that for the downlink, the PDI value is very small i.e. only of 20 msec while for the uplink, the PDI value is much high i.e. of about 200 msec. The possible reason for this behavior is same as discussed before that on the uplink, we usually have less amount of limited shared bandwidth as compared to that available on the downlink.

6.3.2 PDI vs RTP payload size for Uplink

In the previous performance test, PDI values were selected for the uplink and downlink on O2 HSPA network and it has been seen that main constraint is caused on the uplink because of its
limited amount of shared bandwidth due to which the PDI was chosen with the high value of 200 msec.

Now, the same type of test is conducted for the T-Mobile EDGE network for the uplink channel only. So, the server is connected to T-Mobile EDGE network and the multimedia data packets are streamed towards client with PDI value kept at 200 msec and RTP payload size is fixed at 1460 bytes. When the test is conducted, severe packet loss is encountered. One possible reason for this high packet loss seems to be low bandwidth available on the EDGE access network (i.e. less than 384 Kbps). So, to confirm this presumption, the same test is conducted by putting the server on T-Mobile HSPA network (which usually has bandwidth in 2-6 Mbps) by keeping the PDI value of 200 msec with the RTP payload size fixed at 1460 bytes. The result of this test is summarized in the Figure 6.16.

In this Figure 6.16, it can be seen that we have faced large number of RTP packet loss. Out of 981 expected RTP packets, only 86 RTP packets are actually received successfully at the client. Even
the PDI has been increased to 1900 msec in this test (not shown in the figure), but still, all the RTP packets are not received successfully at the client at such high value of the PDI.

Then the intensive testing has been conducted by changing the values of the PDI and the RTP payload sizes other than 200 msec and 1460 bytes respectively. The results that are achieved after this thorough testing are summarized in Figure 6.17.

![Packet Dispatch Interval vs RTP Payload Size](image)

In this Figure 6.17, it can be seen that by keeping the RTP payload size with the maximum value of 1460 bytes, the PDI value of 2500 msec is needed to receive all the RTP packets successfully at the client. This high value of the delay is not acceptable for the real time multimedia services specially in the case of live streaming. So, in order to reduce the PDI value to the 800 msec or the 400msec mark, the RTP payload size is needed to be reduced to 400 bytes and 244 bytes, respectively.

From these results, the possible reason for this behavior is that T-Mobile may have applied “Traffic Policing” rules at the access point of their network, as shown in Figure 6.18, by controlling the Packets Arrival Rate or the Packet Sizes. Due to this traffic policing, the recipient of traffic to whom these policies apply, will observe packet loss throughout those periods when the incoming traffic exceeds certain limits that are defined by data network operator e.g. ISP. So, by changing these values of PDI and RTP payload sizes, we are doing the “Traffic Shaping” (also called packet shaping) at the media source which is bringing the amount of data traffic within the defined allowed limits set by the network operator.

6.3.3 Requirements of RTCP and RSVP protocols

Since, the real time multimedia applications are bandwidth hungry and delay sensitive, therefore their delivery has strict QoS requirement on the bandwidth and delay. From the results gained from the live tests, it seems that there is an increasing requirement of protocol, which can provide feedback for flow control of the multimedia streaming between the two hosts. From this feedback, the upper limits defined by the Traffic Policing and Traffic Shaping can be guessed for the data source to minimize the RTP streaming packet losses. The RTCP protocol that provides such feedback about the multimedia data delivery to the streaming source according to which the source can take certain precautions, like decreasing the RTP payload size or increasing the PDI value to
6. Evaluation and Performance of the M2M and IAG Scenarios

minimize the packet loss at the receiver. Thus, by setting the values of these two parameters i.e. $PDI$ and $RTP$ payload size under the feedback of RTCP, the server can be a “self-limiting source” which will produce traffic that will never exceed some allowed upper bound.

Another important thing is that as these multimedia services are “inelastic” and require a certain minimum level of bandwidth and a certain maximum latency to function properly. So, to achieve the RTP payload size of 1460 bytes with the $PDI$ of 2500 msec is not a proper solution in case of live streaming services, like Internet Telephony, which have the maximum acceptable latency of about 250 msec. Thus, there is a need of certain amount of resource reservation that can be guaranteed at the intermediate nodes over the network path to achieve the QoS that is required by a particular application. The RSVP protocol [6] is one possible answer for that. These desired resources can be reserved on need basis through the RSVP, for example, by having some concrete Service Level Agreement (SLA) with the data network operator.

6.4 General Overheads in the Implemented Design Models

In the design models that have been implemented in this thesis, there are basically two general overheads. First is the periodic TCP keep alive messages which are sent over the signaling path by the media server to the IAG in case of the Scenario II. Second is the periodic UDP keep alive datagrams that are sent from the media server to the client over the data channel in all three implemented scenarios.

6.4.1 The TCP Keep Alive Messages

In the Scenario II, the TCP keep alive messages are sent periodically as the string “TCPKeepAlive” after every 60 seconds by the server to the IAG to keep the TCP tunnel alive that has been established between the server and the IAG. Now, during the buffering of the data packets over the streaming channel between the streaming server and the client, if no RTSP based request is sent specifically over the signaling channel, these keep alive messages will still be sent over the signaling channel between server and the IAG. These message bytes do not belong to the significant control message bytes but are actually counted as an overhead.

Suppose, the total file buffering time $T_{buffer}$ of any media file is 200 seconds and these periodic
keep alive messages are sent with the periodic interval $T_{\text{interval}}$ of 60 seconds, then the minimum number of keep alive messages $N_{\text{keepAlive}}$ that will be needed to send over the signaling channel (when the media server is in its PLAYING state) without issuing any other significant control message bytes will be calculated by the formula shown in the Equation 6.5.

$$N_{\text{keepAlive}} = \frac{T_{\text{buffer}}}{T_{\text{interval}}} \quad (6.5)$$

So, the number of keep alive messages $N_{\text{keepAlive}}$ will be $200/60 = 3$ (approx). Now as the single “TCPKeepAlive” message $B_{\text{keepAlive}}$ is of 12 bytes, then the total number of bytes $N_{\text{totalBytes}}$ that will be sent over the signaling channel during this buffering period will be calculated by the formula shown in the Equation 6.6.

$$N_{\text{totalBytes}} = N_{\text{keepAlive}} \times B_{\text{keepAlive}} \quad (6.6)$$

As a result, the total number of bytes $N_{\text{totalBytes}}$ will be $3 \times 12 = 36$ bytes. These number of bytes will subject to change and will be increased if the $T_{\text{buffer}}$ is increased, either due to the buffering of source file with large size when the media server is in PLAYING state, or by putting the media server in the READY state due to sending of PAUSE request and temporarily halting the media streaming.

### 6.4.2 UDP Keep Alive Messages

The periodic UDP keep alive datagrams are sent over the data channel by the streaming server to the client to keep the UDP tunnel alive that has been established either directly between the media server and client, or between the server and IAG and the client and IAG. By sending these messages, the public port that has been allocated by the NAT (if exists) for the mapping of private port to the public port, are reserved.

The UDP keep alive datagram actually carries the “KeepAlive” message of 9 bytes as its payload. This is the wastage of the resource because the maximum payload that can be carried by UDP datagram is of 1472 bytes (12 Bytes RTP Header + 1460 Bytes RTP payload). Although, in this thesis, to keep the number of the keep-alive UDP datagrams low, their PDI is set to 2000 msec. However, this interval can still be needed to be minimized depending on the UDP timeout configured by the administrator of the NAT to release the reserved public port, causing the increase in the usage number of these undersized UDP datagrams.

### 6.5 Performance Conclusion

In this performance analysis, the different implemented Scenarios have been evaluated on the basis of the request-response delay on the signaling channel and the streaming packet delay on the data channel. Our tests show that the signaling is faster in case of Scenario II, whereas, the streaming is faster (about 1.14 times) when it is done directly between the server and the client instead of relaying it via IAG. So, under these results, Scenario IIa (where signaling is done via the IAG and streaming is done directly as M2M) is performing better as compared to the other Scenarios I and IIb. But, as Scenario IIa is dependent on the specific NAT configuration, so, it cannot be considered as a generalized solution. Therefore, in order to propose a generalized M2M streaming solution which is independent of any NAT specification, Scenario IIb is a better choice.
Regarding the two cellular wireless data network operators, the O2 HSPA performs better with the PDI value of 200 msec at the server (in case of uplink) with the maximum RTP payload size of 1460 bytes as compared to the T-Mobile EDGE/HSPA networks.

Along with the implemented multimedia streaming protocols like RTSP, RTP and SDP, there is a strong need to extend the implemented functionality with the RTCP and RSVP protocols to have the feedback of the data streaming with the resource reservation guarantees on the network path.
CHAPTER 7

Theses

- RTSP protocol has been implemented by using both synchronous and asynchronous architecture over existing REST-based MWS middleware.
- The mapping of the RTSP methods is established and implemented with the synchronous and asynchronous mobile WS.
- The RTSP-based multimedia control is implemented by extending the states of asynchronous mobile WS.
- The SDP protocol is implemented to retrieve the low level description (initialization information) of a media object from mobile media server.
- The RTP Frame structure has been implemented for carrying the single source multimedia data over UDP datagrams.
- Live wireless data networks like WLAN, EDGE and HSPA of eduroam, T-Mobile and O2 networks are investigated for their firewall and NAT issues.
- The Firewalls and NATs are traversed by using Hole Punching techniques based on the UDP and TCP.
- The Intermediate Access Gateway (IAG) is implemented based on the STUN and TURN standard methodologies to propose successful NAT traversal scenarios in M2M settings.
- Multiple scenarios are designed and implemented for M2M multimedia streaming to provide the proof-of-concept.
- Inter-scenario evaluations are conducted and their performance aspects are studied over different live wireless data networks like WLAN, EDGE and UMTS/HSPA.
- The streaming functionality is extended for the mp3 based audio, in addition to the wave format. (prototype is ready but not incorporated in final code)
- The IAG is extended to support the multicasting, which enables the streaming from a single media server to multiple media clients. (prototype implementation is in progress)
7. Theses
Conclusions and Outlook

8.1 Conclusions

The core purpose of the thesis is to extend the existing mobile Web Services (WS) to support the rich multimedia content, such as, audio and video data, along with the light weight textual data in real time. The idea is further extended by allowing both the multimedia server and the client to be any mobile nodes.

As a part of this work, the existing REST-based MWS architecture is extended with the multimedia streaming protocols standards, such as, RTSP and RTP. Here, the RTSP methods are mapped to the synchronous and asynchronous services using the REST architecture. To enable the Mobile-to-Mobile (M2M) controllable media streaming, the RTSP standard uses the TCP as the transport layer protocol, because the reliability is vital for the exchange of signaling to control the overall streaming functionality between the two mobile nodes. However, the RTP standard is implemented on UDP protocol, as the real time multimedia data is delay sensitive and the reliable TCP transport protocol due to its retransmission policy, if used over lossy wireless data networks, can interrupt the reception of the real time media at the client. Secondly, TCP does not support the multicasting which may be required for streaming of multimedia data to multiple clients. Additionally, the issues related to the firewalls and NAT are investigated for the live cellular data networks, like, EDGE, UMTS and HSPA of the T-Mobile and the O2 operators. Then, these issues are addressed by developing an IAG, based on the STUN and TURN concepts, to provide a general M2M multimedia streaming model that is independent of any specific underlying network configuration. Finally, the performance of the developed streaming model over the different wireless networks is evaluated through live network experiments. The best result is obtained by relaying both the control messages over signaling channel, and the multimedia data over streaming channel, via the IAG when the media sever is on O2 HSPA data network while the media client is on eduroam WLAN, with the maximum available RTP payload size of 1460 bytes and with PDI of 200 msec at the media server.

This thesis employs on the Google Android 2.1 as the implementation platform, There, some of the multimedia formats that are supported by the Android are the wave and the mp3 for audio, while the 3gp and the mp4 for video. So, both the mp3 and wave audio formats and the 3gp video format source files have been used to test the streaming between the mobile server and the client. But, it is the technical constraint of the Android that so far the audio file of only wave format can be streamed using the AudioTrack API. For the mp3 audio source file to be streamed at the client, it is first needed to write the bytes of the buffered chunks of the source file explicitly on some static file stored on the SD card of the client, and then played back with the MediaPlayer API. For the video source file, it is first needed to download the source file content completely and then write all the content on some static file stored on the SD card of the client, which can be played back with the VideoView API. These two streaming models are not efficient and that is why the streaming of audio source file of only the wave format has been used in the implementation of this thesis (using the AudioTrack API). For the source files of the mp3 and 3gp formats, another limited version of the prototype has also been implemented.
8.2 Outlook

The designed prototype can be extended for more source file formats for the audio and video data, with the availability of new APIs in the future releases of the Android. In this thesis, only the Audio on Demand (AoD) service has been provided, but it can be extended for the live audio streaming as well by getting the input directly from microphone.

In this thesis, the RTSP protocol has been implemented by using both synchronous and asynchronous REST architecture, which provides the controlled functionality for the streaming. The RTP protocol has been implemented for carrying the media data from one node to the other. As the real time multimedia content delivery has strict QoS requirement on bandwidth and delay. So, there is a strong need for the implementation of RTCP protocol which can provide feedback about the streaming of multimedia data, based on the underlying network specifications. Also, efforts are needed to implement RSVP protocol for the extension of functionality implemented in this thesis, which can reserve the necessary resources at the different nodes over the network path for the smooth real time M2M multimedia streaming.

The results produced in this thesis may be useful to extend the social networks on mobiles, to incorporate social interaction through multimedia sharing.
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>RTSP Major Methods</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>RTSP State transition diagram [9]</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>RTP Frame Structure</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>Architecture of the RESTful mobile Web Services middleware</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>NA(P)T Operation</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>STUN Configuration</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>UDP Hole Punching</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>Full Cone NAT</td>
<td>33</td>
</tr>
<tr>
<td>4.5</td>
<td>Restricted Cone NAT</td>
<td>33</td>
</tr>
<tr>
<td>4.6</td>
<td>Port Restricted Cone NAT</td>
<td>34</td>
</tr>
<tr>
<td>4.7</td>
<td>Symmetric NAT</td>
<td>34</td>
</tr>
<tr>
<td>4.8</td>
<td>Test # 1: Cone vs Symmetric NAT</td>
<td>35</td>
</tr>
<tr>
<td>4.9</td>
<td>Test # 2: Full Cone vs Restricted Cone NAT</td>
<td>35</td>
</tr>
<tr>
<td>4.10</td>
<td>Test # 3: Restricted Cone vs Port Restricted Cone NAT</td>
<td>36</td>
</tr>
<tr>
<td>4.11</td>
<td>P2P UDP Session for Peers behind a Common NAT [1]</td>
<td>37</td>
</tr>
<tr>
<td>4.12</td>
<td>P2P UDP Session for Peers behind Different NATs [1]</td>
<td>38</td>
</tr>
<tr>
<td>4.14</td>
<td>TURN Configuration [10]</td>
<td>41</td>
</tr>
<tr>
<td>5.1</td>
<td>Implementation Architecture for M2M Streaming</td>
<td>46</td>
</tr>
<tr>
<td>5.2</td>
<td>Excluded Fields from Generalized RTP Frame</td>
<td>47</td>
</tr>
<tr>
<td>5.3</td>
<td>Implemented RTP Frame Structure</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>RTP Maximum Payload size</td>
<td>49</td>
</tr>
<tr>
<td>5.5</td>
<td>Checking for SD Card Mounting</td>
<td>49</td>
</tr>
<tr>
<td>5.6</td>
<td>Selection of File &amp; Setting of FileInputStream</td>
<td>49</td>
</tr>
<tr>
<td>5.7</td>
<td>Reading File contents Frame by Frame</td>
<td>50</td>
</tr>
<tr>
<td>5.8</td>
<td>RTP Constructor for Server</td>
<td>51</td>
</tr>
<tr>
<td>5.9</td>
<td>RTP Constructor for Client</td>
<td>52</td>
</tr>
<tr>
<td>5.10</td>
<td>Returning the value contained in Marker (M) field</td>
<td>52</td>
</tr>
<tr>
<td>5.11</td>
<td>Retrieving RTP Payload Contents</td>
<td>52</td>
</tr>
<tr>
<td>5.12</td>
<td>RTP Header Fields for First received RTP Packet</td>
<td>53</td>
</tr>
<tr>
<td>5.13</td>
<td>RTP Header Fields for Last received RTP Packet</td>
<td>53</td>
</tr>
<tr>
<td>5.14</td>
<td>Scenario I: Server on T-Mobile EDGE, Client on eduroam WLAN</td>
<td>54</td>
</tr>
<tr>
<td>5.15</td>
<td>Scenario I: Server on T-Mobile EDGE, Client on O2 UMTS/HSPA</td>
<td>55</td>
</tr>
<tr>
<td>5.16</td>
<td>Scenario I: Server on T-Mobile EDGE, Client on T-Mobile HSPA</td>
<td>56</td>
</tr>
<tr>
<td>5.17</td>
<td>Scenario I: Server on O2 UMTS/HSPA, Client on eduroam WLAN</td>
<td>57</td>
</tr>
<tr>
<td>5.18</td>
<td>Scenario II: Server on T-Mobile EDGE, IAG on CN LAN, Client on eduroam WLAN</td>
<td>58</td>
</tr>
<tr>
<td>5.19</td>
<td>Scenario IIa: Server on O2 UMTS/HSPA, IAG on CN LAN, Client on eduroam WLAN</td>
<td>59</td>
</tr>
<tr>
<td>5.20</td>
<td>Scenario IIa: Server on T-Mobile HSPA, IAG on CN LAN, Client on eduroam WLAN</td>
<td>60</td>
</tr>
</tbody>
</table>
5.21 Scenario II: Server on T-Mobile EDGE, IAG on CN LAN, Client on O2 UMTS/HSPA .................................................. 61
5.22 Scenario IIb: Server on T-Mobile HSPA, IAG on CN LAN, Client on O2 UMTS/HSPA .................................................. 61
5.23 Scenario IIb: Server on T-Mobile EDGE, IAG on CN LAN, Client on T-Mobile HSPA .................................................. 62
5.24 Scenario I: Direct M2M Signaling & Streaming ......................... 66
5.25 Scenario IIa: Direct M2M Streaming but Signaling via IAG ........... 67
5.26 Scenario IIb: Both Signaling & Streaming via IAG .................. 67
5.27 StreamMM Service Invocation and mmServer started ............... 81
5.28 Receiving of the SETUP response and starting mmClient ........... 82
5.29 Sending of UDP based Notifications .................................... 83
5.30 Setting of Observer State after receiving UDP Notification ......... 84
5.31 Opening up file for Reading .............................................. 86
5.32 Sending of RTP Packet ...................................................... 87
5.33 Receiving of RTP Packets at Client ..................................... 88
5.34 Playing Out of the media content by AudioTrack ..................... 90
5.35 Setting of Sampling Rate ................................................... 90
5.36 Setting of Channel Configuration ....................................... 91
5.37 Setting of PCM Encoding Type .......................................... 91
5.38 OPTIONS/DESCRIBE request & response for Scenario I ........... 92
5.39 SETUP request & response for Scenario I ............................. 92
5.40 PLAY request & response for Scenario I ............................. 93
5.41 PAUSE request & response for Scenario I ............................ 93
5.42 TEARDOWN request & response for Scenario I ...................... 94
5.43 OPTIONS/DESCRIBE request & response for Scenario IIa .......... 95
5.44 SETUP request & response for Scenario IIa .......................... 96
5.45 PLAY request & response for Scenario IIa ........................... 96
5.46 PAUSE request & response for Scenario IIa .......................... 97
5.47 TEARDOWN request & response for Scenario IIa ................... 97
5.48 OPTIONS/DESCRIBE request & response for Scenario IIb .......... 98
5.49 SETUP request & response for Scenario IIb .......................... 99
5.50 PLAY request & response for Scenario IIb ........................... 99
5.51 PAUSE request & response for Scenario IIb .......................... 100
5.52 TEARDOWN request & response for Scenario IIb .................... 100
6.1 Sequence Flow Diagram for Scenario I ............................... 104
6.2 Sequence Flow Diagram for Scenario II ............................... 105
6.3 Service Times for OPTIONS and DESCRIBE ......................... 106
6.4 Request-Response Delay of OPTIONS ................................... 106
6.5 Request-Response Delay of DESCRIBE ................................ 107
6.6 Request-Response Delay of SETUP ..................................... 108
6.7 Request-Response Delay of PLAY ....................................... 108
6.8 Total File Buffering Time (msec) ...................................... 110
6.9 Total File Buffering Time (sec) ....................................... 110
6.10 Performance Difference between Streaming via M2M and via IAG .... 111
6.11 Inter-packet Delay Difference for Streaming via M2M and via IAG .... 111
6.12 IAG Delay Overhead ..................................................... 112
6.13 RTP Packet Streaming over Uplink .................................... 113
6.14 RTP Packet Streaming over Downlink ................................ 113
List of Figures

6.15  PDI at server for Uplink and Downlink ............................................ 114
6.16  Total Number of RTP Packet Loss ......................................................... 114
6.17  PDI vs RTP Payload Size ................................................................... 115
6.18  Components involved in Congested Network ........................................ 116
LIST OF TABLES

5.1 Summary of Test Results .................................................. 64
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTSP</td>
<td>Real Time Streaming Protocol</td>
</tr>
<tr>
<td>RTP</td>
<td>Real Time Transport Protocol</td>
</tr>
<tr>
<td>RTCP</td>
<td>Real Time Transport Control Protocol</td>
</tr>
<tr>
<td>SDP</td>
<td>Session Description Protocol</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>WS</td>
<td>Web Services</td>
</tr>
<tr>
<td>MWS</td>
<td>Mobile Web Server</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transport Protocol</td>
</tr>
<tr>
<td>IANA</td>
<td>Internet Assigned Numbers Authority</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>W3C</td>
<td>World Wide Web Consortium</td>
</tr>
<tr>
<td>WWW</td>
<td>World Wide Web</td>
</tr>
<tr>
<td>NAT</td>
<td>Network Address Translation</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IPv4</td>
<td>Internet Protocol version 4</td>
</tr>
<tr>
<td>IPv6</td>
<td>Internet Protocol version 6</td>
</tr>
<tr>
<td>NAPT</td>
<td>Network Address Port Translation</td>
</tr>
<tr>
<td>PAT</td>
<td>Port Address Translation</td>
</tr>
<tr>
<td>P2P</td>
<td>Peer-to-Peer</td>
</tr>
<tr>
<td>STUN</td>
<td>Simple Traversal of UDP through NATs</td>
</tr>
<tr>
<td>TURN</td>
<td>Traversal Using Relay NAT</td>
</tr>
<tr>
<td>ISP</td>
<td>Internet Service Provider</td>
</tr>
<tr>
<td>M2M</td>
<td>Mobile-to-Mobile</td>
</tr>
<tr>
<td>A/V</td>
<td>Audio/Video</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>WLAN</td>
<td>Wireless Local Area Network</td>
</tr>
<tr>
<td>WWAN</td>
<td>Wireless Wide Area Network</td>
</tr>
<tr>
<td>EDGE</td>
<td>Enhanced Data for GSM Evolution</td>
</tr>
<tr>
<td>HSPA</td>
<td>High Speed Packet Access</td>
</tr>
<tr>
<td>UMTS</td>
<td>Universal Mobile Telecommunications System</td>
</tr>
<tr>
<td>MTU</td>
<td>Maximum Transmission Unit</td>
</tr>
<tr>
<td>IAG</td>
<td>Intermediate Access Gateway</td>
</tr>
<tr>
<td>CN</td>
<td>ComNets</td>
</tr>
<tr>
<td>EPR</td>
<td>End Point Reference</td>
</tr>
<tr>
<td>NTP</td>
<td>Network Time Protocol</td>
</tr>
<tr>
<td>FQDN</td>
<td>Fully Qualified Domain Name</td>
</tr>
<tr>
<td>EPRD</td>
<td>End Point Reference</td>
</tr>
<tr>
<td>PDI</td>
<td>Packet Dispatch Interval</td>
</tr>
<tr>
<td>RSVP</td>
<td>Resource Reservation Protocol</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
</tbody>
</table>
A. Abbreviations
BIBLIOGRAPHY

[1] Dan Kegel Bryan Ford, Pyda Srisuresh. Peer-to-Peer Communication Across Network Address Translators. Published: www.brynosaurus.com/pub/net/p2pnat/, 17 February 2005. 4.1, 4.2.1.3, 4.11, 4.12, 4.13, 4.2.1.7, 4.2.2, 8.2


[7] V.Jacobson M.Handley. RFC 2327 - SDP: Session Description Protocol. Published: www.networksorcery.com/enp/rfc/rfc2327.txt, April 1998. 2.4, 2.4.1, 2, 3, 4, 5, 6, 7, 2.4.1


[14] Suguru Yoshida Shigeki Goto Yuan Wei, Daisuke Yamada. A New Method for Symmetric NAT Traversal in UDP and TCP, 4 August 2008. 4.2.2