TECHNOLOGICAL EDUCATIONAL INSTITUTE OF CRETE

DEPARTMENT OF APPLIED INFORMATICS AND MULTIMEDIA

A SIP/IMS compatible PQoS-aware TAM for Android

GEORGE TSIOLIS

1548

DEC 2011

SUPERVISOR: GEORGE XILOURIS

Abstract

The convergence of multimedia (i.e. video, voice and data) services with mobile/fixed networks and broadcast-interactive applications is creating new demands for high quality and user- responsive service provision management. To face the challenges of defining and developing the next generation of ubiquitous and converged network, together with the respective service infrastructures for communication, computing and media, the industry has launched various initiatives in order to design the reference network architecture and standardize the various modules/interfaces that are necessary for delivering the expected services. Legacy standardization bodies drive some of these initiatives, with a particular focus on interoperability issues.

The predominant candidate for this convergence is the IP Multimedia Subsystem (IMS), which is a standard (originally defined by the 3GPP) for next-generation mobile networking applications based on Session Initiation Protocol (SIP) as the basic common signalling protocol. It is a specification framework that introduces the functional and network elements, as well as service platforms and the respective architecture, which enable real multimedia convergence on an IP–based infrastructure. Currently, IMS has been adopted as the standard control system in TISPAN NGN architectures.

This final year project describes the design, implementation and usage of a software developed for monitoring the quality of service perceived by the user of the terminal. The quality of service was calculated based on statistics available from existing terminals, which were compatible with the IMS (IP Multimedia Subsystem) environment. Each terminal was equipped with this software and communicated with the same server that collected statistics from all terminal devices regarding the quality of service. The software was developed using Android, an open platform for mobile development.

Keywords: PQoS, SIP, IMS, Android

Acknowledgments

I would like to thank my supervisor George Xilouris, Associate Researcher at Media Networks Laboratory at Informatics and Telecommunications Institute at National Centre for Scientific Research Demokritos for the precious guidance, valuable comments and prompt suggestions throughout this project.

Special thanks go to Dr. Anastasios Kourtis, Research Director at Media Networks Laboratory at Informatics and Telecommunications Institute of National Centre for Scientific Research Demokritos who gave me the opportunity and the ability to work and exploit the laboratory experimental infrastructure.

Also, I would like to thank everyone I have worked with, during my post-graduate studies at the Technological Educational Institute of Crete.

Contents

List of Figures

1 Introduction

Designing and implementing a SIP/IMS compatible client application for **Android** can be tricky. Someone must first get familiar with the **Session Initiation Protocol** (SIP), the signalling protocol that lies inside an **IP Multimedia Subsystem** (IMS). In other words, you will need to understand the structure of the SIP and experiment with SIP Messages such as Requests and Responses. Afterwards you will have to accumulate knowledge regarding the IMS in favor of deep understanding of this architectural framework and its inner workings. IMS was originally developed to address network convergence challenges and end-user requirements which make it a fairly complicated system. To expand on this, the IMS is the unified telecommunication industry approach toward an All-IP network architecture that merges the paradigms and technologies of the Internet with the cellular and fixed telecommunication worlds. Yet **Quality of Service** (QoS) is a key component in the original design and conception of IMS. The spectrum of this final year project includes the design, implementation and evaluation of software developed for monitoring the quality of service perceived by the user of the terminal. The **Perceived Quality of Service** (PQoS) concept implies that for the user what matters the most is the perception of the quality of an application rather than anything else. In other words, a large scale deployment of applications will only be successful if the perceived quality of these applications will be sufficiently high. Regarding the implementation of the software described in this project, a streaming server located in the IMS core network was used to stream an MPEG-4 H.264 video to a mobile device based on the Android Operating System. Additionally, the software should comply with the IMS including handling SIP messaging whilst being able to measure and evaluate the perceived quality of a video streamed to a mobile device over a wireless connection. On the following chapters you will be introduced to essential background information and knowledge someone should grasp to proceed on developing such software.

2 Background

This chapter describes the technologies that lie behind this project. Four of the core technologies utilized in this project include PQoS, SIP, IMS and Android. Apart from these, a fundamental experience with unices and networking is required to achieve valuable experimentation and analysis while implementation.

2.1 PQoS (Perceived Quality of Service)

The Perceived Quality of Service (PQoS) concept is supposed to introduce a new way of escalating the quality of service (QoS) by calculating QoS as it is finally perceived by the end-user. With the advent of 3G mobile communication networks, the classical boundaries between telecommunications, multimedia and information technology sectors are fading. The goal of this convergence is the creation of a single platform that will allow ubiquitous access to Internet, multimedia services, interactive audiovisual services, and in addition (and most important) offering the required/appropriate perceived quality level at the end user's premises. One of the 3G/4G visions is the provision of audiovisual content at various quality and price levels. There are many approaches to this issue, one being the Perceived Quality of Service (PQoS) concept.

2.1.1 QoS (Quality of Service)

The term quality of service (QoS) refers to resource reservation control mechanisms rather than the achieved service quality. Due to the exponential growth of the Internet, researchers have reached out for better mechanisms to control resource reservation.

A basic QoS architecture includes:

- **Identification and marking techniques** that help identifying traffic flows along with marking network packets, thus achieving classification of the traffic flows as well as resource reservation.
- **Queuing, scheduling, and traffic-shaping tools** that help network performance and efficiency.
- **Policy, management and control functions** that help setting and evaluating QoS policies and goals.

The most known technology which is broadly used for applying QoS at network level is Differential Services (DiffServ).

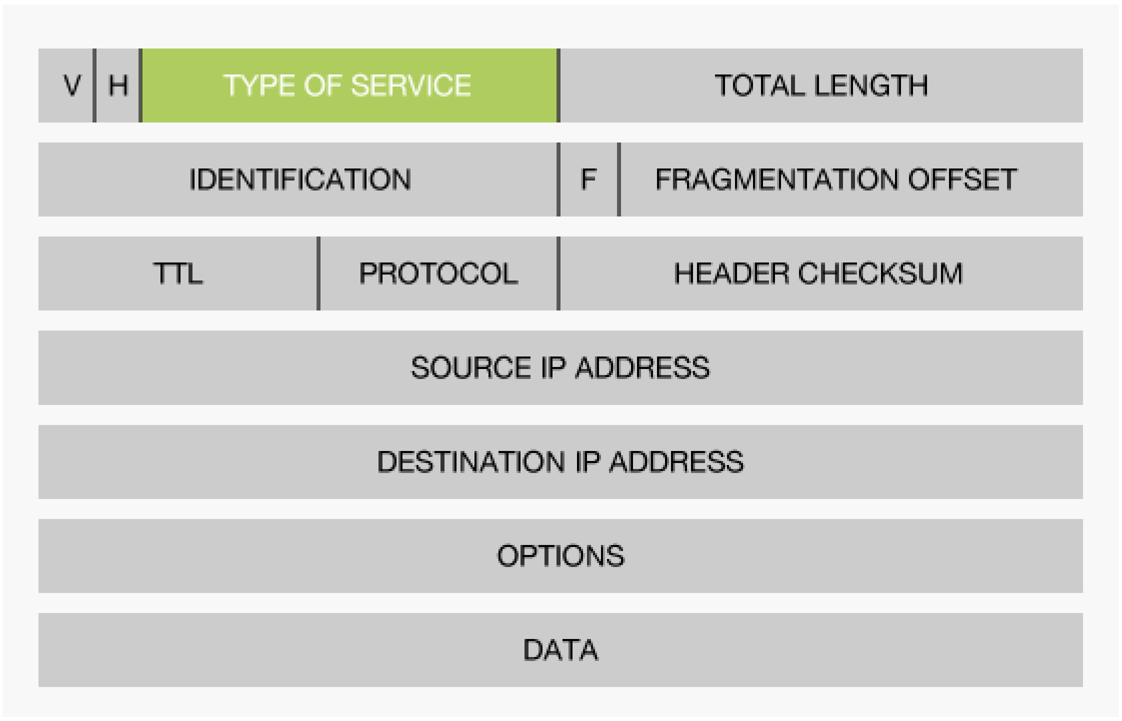

Figure 1 - TOS Field

The DiffServ architecture is based on a simple model. When traffic entering a network is classified and possibly conditioned at the boundaries of the network, and assigned to different Behaviour Aggregates (BAs), with each BA being identified by a single DiffServ Code-Point (DSCP). Within the core of the network, packets are forwarded according to a Per-Hop Behaviour (PHB) associated with the DSCP. The smallest autonomic unit of DiffServ is called a DiffServ domain, where services are assured by identical principles.

A domain consists of two types of nodes: Boundary (or Edge) routers and Core routers. Core nodes only forward packets, they do no signalling. Each router packets traverse is called a "hop" Packets, classified at the edge of the network, are forwarded according to a specific PHB throughout the core of the network. Packets may be forwarded across multiple networks on their way from source to destination. Each one of those networks is called a DiffServ Domain. More specific, a DiffServ Domain is a set of routers implementing the same set of PHBs.

The DiffServ PHB class selector offers three forwarding priorities:

Expedited Forwarding (EF) characterized by a minimum configurable service rate, independent of the other aggregates within the router, and oriented to low delay and low loss services.

Assured Forwarding (AF) group, recommended in [RFC 2597] for 4 independent classes (AF1, AF2, AF3, AF4) although a DiffServ domain can provide a different number of AF classes. Within each AF class, traffic differentiated into 3 "drop precedence" categories.
Best Effort (BE), which does not provide any performance guarantee and does not define any QoS level.

The next figure shows an example of how different kind of service flows classified to different behaviour aggregates with different priorities (highest, middle and lowest) and forwarded inside the DiffServ domain.

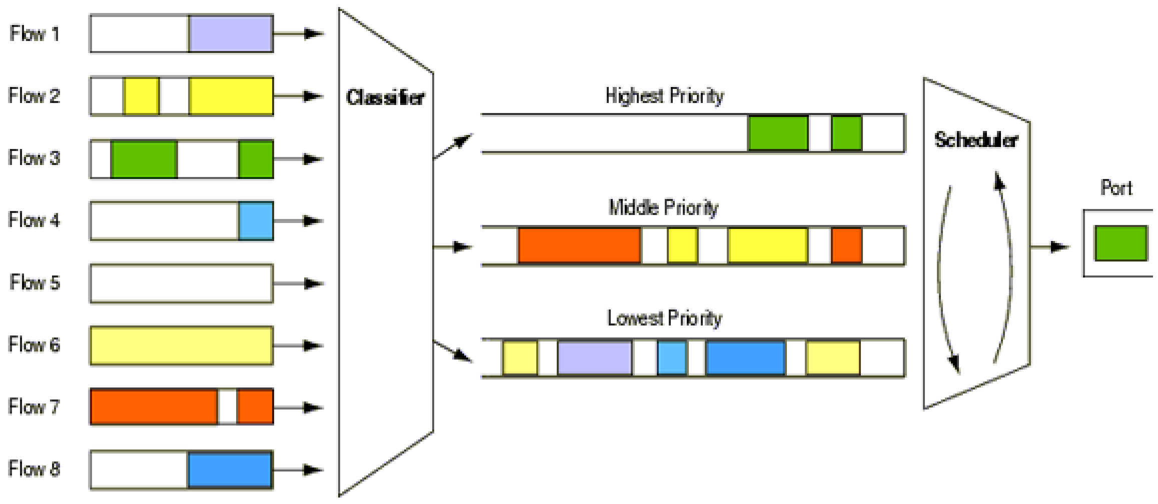

Figure 2 - DiffServ Mechanism

Achieving these tasks requires in-depth understanding of networking fundamentals as well as leveraging a user-centric approach to the quality of the service.

When deploying a QoS-aware network the key qualities of the traffic come down to availability, delay, reliability, scalability, etcetera. Thus packet losses will probably make communication less reliable, significant network latency will result to unavailability and so on.

2.1.2 QoS metrics

The term of Quality of Service usually comes down to the measuring of performance for a transmission system that reflects its transmission quality and service availability. To achieve these prerequisites you should be able to evaluate various network specific metrics. Dealing with network congestion is a challenging problem. As far as network metrics are considered, we could easily differentiate what we should be looking for depending on the kind of traffic that is mainly TCP or UDP. Regarding TCP, which is considered to be the reliable protocol inside the Transport Layer, someone should be able to keep track of the **round-trip time** (RTT) of the packets transmitted and the **retransmissions** required to make the connection reliable. **Throughput** of course is one main metric that you should take into consideration regardless of the kind of traffic. UDP on the other hand, provides non-reliable delivery of packets thus makes no sense to keep track of the RTT or retransmissions which are non-existent. When targeting for quality of service over a UDP connection you should be able to measure **one-way delay** of the packets transmitted, **packet losses** and **jitter** also known as average delay variation. The specific project relies mainly on UDP connections whilst TCP was a stepping stone for checking connections before experimenting.

A further step is to associate application to network layer QoS requirements. Concerning the IP environment, the QoS application metrics used are as follows:

- IPLR – IP Packet Loss Ratio
- IPTD – IP Packet Transfer Delay
- IPDV – IP Packet Delay Variation (known as Jitter)
- IPER – IP Packet Error Ratio.

IP Packet Loss Ratio is the performance measure. Even if, the problem needs to be solved for each specific application, it is also important to give a range of QoS requirements for traffic classes. Possible end-to-end performance-metric upper bounds are associated with QoS classes in [ITU-T-Y.1541]. This association is introduced in [ITU-T-Y.1541]. The nature of the objective performance parameter is defined in [ITU-T-Y.1541] as follows: IPTD – upper bound on the mean IPTD; IPDV – upper bound on the 1–10–3 quintile of IPTD minus the minimum IPTD; IPLR – upper bound on the packet loss probability; IPER – upper bound. “U” stands for “Unspecified” (actually meaning “Unbounded”, in this context).

The following table depicts the upper bound acceptable values for each application type:

Application Characteristics	IPTD	IPDV	IPLR	IPER
Real time, jitter sensitive, highly interactive (VoIP)	100ms	50ms	$1x10^{-3}$	$1x10^{-4}$
Real time, jitter sensitive, interactive	400ms	50ms	$1x10^{-3}$	$1x10^{-4}$
Transaction data, highly interactive	100ms	U	$1x10^{-3}$	$1x10^{-4}$
Transaction data, interactive	400ms	U	$1x10^{-3}$	$1x10^{-4}$
Low loss only (short transactions, bulk data, video streaming)	1s	U	$1x10^{-3}$	$1x10^{-4}$
Traditional applications of default IP networks	U	U	U	U

Figure 3 - IP QoS classes and objective performance-metric upper limits (U-Unspecified)

2.1.3 Estimation of Perceived Quality of Service

The evaluation of the PQoS for audiovisual content will provide a user with a range of potential choices, covering the possibilities of low, medium or high quality levels. Moreover the PQoS evaluation gives the service provider and network operator the capability to minimize the storage and network resources by allocating only the resources that are sufficient to maintain a specific level of user satisfaction and tune transmission parameters to obtain the best impact on users.

The challenge for the continued growth of VoIP and in general audiovisual services is the satisfaction of user quality expectations, especially their expectation of quality to be comparable to that provided by traditional telephony to which they are accustomed. Because the Internet is not always able to deliver user expectations of quality given its unreliability, adaptive mechanisms help deliver quality to as close as possible to what the users expect or to maximize quality for a given network state.

The evaluation of the PQoS for voice and video content provide a user with a range of potential choices, covering the possibilities of low, medium or high quality levels. Moreover the PQoS evaluation gives the service provider and network operator the capability to minimize the storage and network resources by allocating only the resources that are sufficient to maintain a specific level of user satisfaction and tune transmission parameters to obtain the best impact on users.

The PQoS models are able to predict voice/video quality from network/link parameters (e.g. packet loss rate, link bit error rate, delay and jitter) and terminal parameters (e.g. codec type, mode, bit rate, frame rate, packet size). One of the key design objectives is for PQoS model to be lightweight so that they can be easily implemented in user equipment (e.g. a mobile terminal). PQoS modeling for voice followed a similar approach to the popular ITU-T E-model, but supported adaptive 8 modes AMR codec used in 3G handset to allow adaptation to network/terminal changes automatically. PQoS modeling for video was similar to the current V-model, which is a hot research topic, given that there is no current standardised V-model yet for video over IP and IPTV applications.

As mentioned before, the **Perceived Quality of Service** (PQoS) concept implies that for the user what matters the most is the perception of the quality of an application rather than anything else. In other words, a large scale deployment of applications will only be successful if the perceived quality of these applications will be sufficiently high. Given that this project includes video streaming it is safe to mention that video delivery in IP networks is achieved via two main mechanisms: download-and-play and streaming.

Streaming is a more sophisticated mechanism. The Server-Streamer sends the video at a controlled rate, following specific timestamps within the stream, corresponding to the actual playback rate of the multimedia content. At the transport layer, UDP is more commonly used, combined with media-delivery-related protocols in the upper layers (such as RTP/RTSP). Unlike download-and-play, streaming is suitable for both unicast and multicast/broadcast services. Moreover it is suitable for both prerecorded and live content. Apart from the Internet and mobile applications, streaming is also almost exclusively used in large IPTV platforms (e.g. in a “triple-play” package) and also in broadcasting systems.

The video quality, as perceived by the user, can be affected by all the aforementioned elements. For example, a lot of research work has been carried out on the effect of the encoding procedure on the picture quality. In this project we will focus on the effect of the IP distribution network and the associated network-level impairments i.e. the Network-level QoS on the perceived video quality (PQoS). Such impairments could include e.g. congestion on a link-bottleneck in a wired system or a strong signal fading in a wireless platform, which are directly translated to increased delay, jitter and eventually packet loss.

2.1.4 Mapping of the QoS to PQoS

Regarding the mapping between the various discrete QoS layer (i.e. PQoS/AppQoS/NQoS), different metrics are used by each level in order to quantify the effects of each layer over the delivered media stream, even if it is VoIP or IPTV service. The following table provides the representative metrics of each layer:

Service QoS Level	Application QoS Level	Network QoS Level
User Satisfaction	Decoding Threshold	Packet Loss
PQoS Level	Encoding Parameters	Packet Loss Scheme
Terminal Specifications	Decodable Rate	Packet Size

Figure 4 - Metrics of each QoS Level

At the service QoS level, the critical metric is the user satisfaction (i.e. PQoS level). The evaluation of the PQoS for audiovisual content will provide a user with a range of potential choices, covering the possibilities of low, medium or high quality levels (i.e. gold, silver and bronze services). Moreover the PQoS evaluation gives the service provider and network operator the capability to minimize the storage and network resources by allocating only the resources that are sufficient to maintain a specific level of user satisfaction.

The evaluation of the PQoS is a matter of objective and subjective evaluation procedures, each time taking place after the encoding process (post-encoding evaluation). Subjective quality evaluation processing of video streams (PQoS evaluation) requires large amount of human resources, establishing it as a time-consuming process (e.g. large audiences evaluating video/audio sequences). Objective evaluation methods, on the other hand, can provide PQoS evaluation results faster, but require large amount of machine resources and sophisticated apparatus configurations. Towards this, objective evaluation methods are based and make use of multiple metrics, which are related to the content's artifacts (i.e. tilling, blurriness, error blocks, etc.) resulting from the quality degradation due to the encoding process.

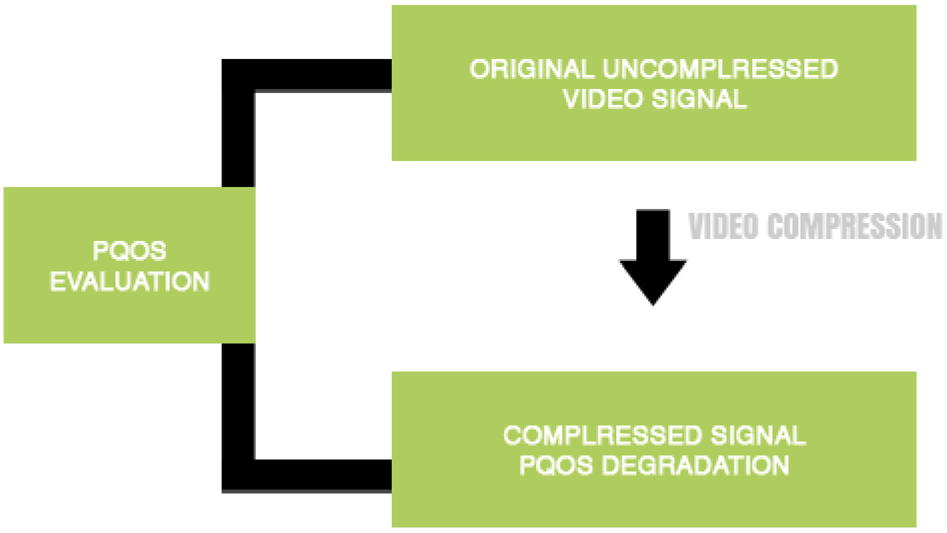
ORIGINAL UNCOMPLRESSED VIDEO SIGNAL
VIDEO COMPRESSION
PQOS EVALUATION
COMPLRESSED SIGNAL PQOS DEGRADATION

Figure 5 - Concept of the PQoS evaluation

At the AppQoS level, given that during the encoding process quality degradation of the initial video content (see Figure 5) is incurred, the values of the encoding parameters (i.e. bit rate, resolution) play a major role in the resulting PQoS. Thus, the various encoding parameters must be used as metrics in quantifying the deduced PQoS level. If we also consider additional degradation due to transmission problems (i.e. limited bandwidth, network congestion), which finally result in packet loss at the video packet receiver during the service transmission, then the Decodable Frame Rate can be considered as a metric for quantifying this phenomenon. The Decodable Frame Rate Q is an application-level metric, with values ranging from 0 to 1.0. The larger the value of Q, the higher the successful decoding rate at the end-user. Q is defined as the fraction of decodable frame rate, which is the number of decodable frames (i.e. frames that are theoretically able to be decoded without considering the post-filtering or error concealment abilities of each decoder) over the total number of frames sent by a video source. Since different codec and transmission techniques have different tolerance to packet loss, the theoretically expected decoding threshold will be also used as a metric in order to define the impact of the packet loss ratio on the frame loss ratio. A theoretical decoding threshold equal to 1.0 means that only one packet loss results in unsuccessful decoding of the corresponding frame, to which the missing packet is a part of.

Finally, at the NQoS level the metrics Packet Loss Ratio, Packet Loss scheme and Packet Size may be considered as key parameters. Although, it is obvious that other network statistics and phenomena may be present over a broadcasting network (e.g. jitter, delay), however all these parameters are quantified into the packet loss effect, since this is the final outcome of all these network QoS-sensitive parameters at the video packet receiver. Otherwise, if no packet loss occurs due to these phenomena, then sophisticated buffer techniques may eliminate their impact. Thus, with the appropriate approach the packet loss ratio can be considered as adequate parameter and used as a network metric to the PQoS-NQoS and NQoS-PQoS mapping. Regarding the various packet loss schemes (e.g. unified, bursty etc.), due to the stochastic nature of the PQoS degradation over an error-prone broadcasting channel, for reference purposes focus must be given on identifying the packet loss scheme, which provides the worst case scenario in terms of affecting the decodable frame rate (i.e. the delivered PQoS level) for specific packet loss ratio.

2.2 SIP (Sessions Initiation Protocol)

The **Session Initiation Protocol** (SIP) is an application-layer control protocol that can establish, modify and terminate multimedia sessions (conferences) such as Internet telephony calls. SIP can also invite participants to already existing sessions, such as multicast conferences. Media can be added to (and removed from) an existing session. SIP transparently supports name mapping and redirection services, which supports personal mobility - users can maintain a single externally visible identifier regardless of their network location.

2.2.1 Protocol Design

SIP is structured as a layered protocol, which means that its behavior is described in terms of a set of fairly independent processing stages with only a loose coupling between each stage. The protocol behavior is described as layers for the purpose of presentation, allowing the description of functions common across elements in a single section. It does not dictate an implementation in any way. When we say that an element "contains" a layer, we mean it is compliant to the set of rules defined by that layer. Not every element specified by the protocol contains every layer. Furthermore, the elements specified by SIP are logical elements, not physical ones. A physical realization can choose to act as different logical elements, perhaps even on a transaction-by-transaction basis.

The lowest layer of SIP is its syntax and encoding. Its encoding is specified using an augmented Backus-Naur Form grammar (BNF). The second layer is the transport layer. It defines how a client sends requests and receives responses and how a server receives requests and sends responses over the network. All SIP elements contain a transport layer. The third layer is the transaction layer. Transactions are a fundamental component of SIP. A transaction is a request sent by a client transaction (using the transport layer) to a server transaction, along with all responses to that request sent from the server transaction back to the client. The transaction layer handles application-layer retransmissions, matching of responses to requests and application-layer timeouts. Any task that a user agent client (UAC) accomplishes takes place using a series of transactions. User agents contain a transaction layer, as do stateful proxies. Stateless proxies do not contain a transaction layer. The transaction layer has a client component (referred to as a client transaction) and a server component (referred to as a server transaction), each of which are represented by a finite state machine that is constructed to process a particular request. The layer above the transaction layer is called the transaction user (TU). Each of the SIP entities, except the stateless proxy, is a transaction user. When a TU wishes to send a request, it creates a client transaction instance and passes it the request along with the destination IP address, port, and transport to which to send the request. A TU that creates a client transaction can also cancel it. When a client cancels a transaction, it requests that the server stop further processing, revert to the state that existed before the transaction was initiated, and generate a specific error response to that transaction. This is done with a CANCEL request, which constitutes its own transaction, but references the transaction to be cancelled.

The SIP elements, that is, user agent clients and servers, stateless and stateful proxies and registrars, contain a core that distinguishes them from each other. Cores, except for the stateless proxy, are transaction users. While the behavior of the UAC and UAS cores depends on the method, there are some common rules for all methods. For a UAC, these rules govern the construction of a request; for a UAS, they govern the processing of a request and generating a response. Since registrations play an important role in SIP, a UAS that handles a REGISTER is given the special name registrar. The most important method in SIP is the INVITE method, which is used to establish a session between participants. A session is a collection of participants, and streams of media between them, for the purposes of communication.

2.2.2 SIP Network Elements

A SIP user agent (UA) is a logical network end-point used to create or receive SIP messages and thereby manage a SIP session. A SIP UA can perform the role of a User Agent Client (UAC), which sends SIP requests, and the User Agent Server (UAS), which receives the requests and returns a SIP response. These roles of UAC and UAS only last for the duration of a SIP transaction.

2.2.3 SIP Messages

SIP is a text-based protocol and uses the UTF-8 charset. A SIP message is either a request from a client to a server, or a response from a server to a client.

SIP requests are distinguished by having a Request-Line for a start-line. A Request-Line contains a method name, a Request-URI, and the protocol version separated by a single space (SP) character. The Request-Line ends with CRLF. No CR or LF is allowed except in the end-of-line CRLF sequence. No linear whitespace (LWS) is allowed in any of the elements.

Request-Line = Method SP Request-URI SP SIP-Version CRLF

Method: This specification defines six methods: REGISTER for registering contact information, INVITE, ACK, and CANCEL for setting up sessions, BYE for terminating sessions, and OPTIONS for querying servers about their capabilities. SIP extensions, documented in standards track RFCs, may define additional methods.

Request-URI: The Request-URI is a SIP. It indicates the user or service to which this request is being addressed. The Request-URI MUST NOT contain unescaped spaces or control characters and MUST NOT be enclosed in "<>". SIP elements MAY support Request-URIs with schemes other than "sip" and "sips". SIP elements MAY translate non-SIP URIs using any mechanism at their disposal, resulting in SIP URI, SIPS URI, or some other scheme.

SIP-Version: Both request and response messages include the version of SIP in use. To be compliant with this specification, applications sending SIP messages MUST include a SIP-Version of "SIP/2.0". The SIP-Version string is case-insensitive, but implementations MUST send upper-case. Unlike HTTP/1.1, SIP treats the version number as a literal string. In practice, this should make no difference.

For SIP requests, [RFC 3261] defines the following methods:

REGISTER: Used by a UA to indicate its current IP address and the URLs for which it would like to receive calls.
INVITE: Used to establish a media session between user agents.

ACK: Confirms reliable message exchanges.

CANCEL: Terminates a pending request.

BYE: Terminates a session between two users in a conference.

OPTIONS: Requests information about the capabilities of a caller, without setting up a call.

Since the proposed standard in [RFC 3261], SIP introduced a few more methods that include:

PRACK: Provisional acknowledgement.

SUBSCRIBE: Subscribes for an Event of Notification from the Notifier.

NOTIFY: Notify the subscriber of a new Event.

PUBLISH: Publishes an event to the Server.

INFO: Sends mid-session information that does not modify the session state.

REFER: Asks recipient to issue SIP request (call transfer.)

MESSAGE: Transports instant messages using SIP.

UPDATE: Modifies the state of a session without changing the state of the dialog.

SIP responses are distinguished from requests by having a Status-Line as their start-line. A Status-Line consists of the protocol version followed by a numeric Status-Code and its associated textual phrase, with each element separated by a single SP character. No CR or LF is allowed except in the final CRLF sequence.

Status-Line = SIP-Version SP Status-Code SP Reason-Phrase CRLF

The Status-Code is a 3-digit integer result code that indicates the outcome of an attempt to understand and satisfy a request. The Reason-Phrase is intended to give a short textual description of the Status-Code. The Status-Code is intended for use by automata, whereas the Reason-Phrase is intended for the human user. A client is not required to examine or display the Reason-Phrase.
While this specification suggests specific wording for the reason phrase, implementations MAY choose other text, for example, in the language indicated in the Accept-Language header field of the request. The first digit of the Status-Code defines the class of response. The last two digits do not have any categorization role. For this reason, any response with a status code between 100 and 199 is referred to as a "1xx response", any response with a status code between 200 and 299 as a "2xx response", and so on. SIP/2.0 allows six values for the first digit:

1xx: Provisional -- request received, continuing to process the request.

2xx: Success -- the action was successfully received, understood and accepted.

3xx: Redirection -- further action needs to be taken in order to complete the request.

4xx: Client Error -- the request contains bad syntax or cannot be fulfilled at this server.

5xx: Server Error -- the server failed to fulfill an apparently valid request.

6xx: Global Failure -- the request cannot be fulfilled at any server.

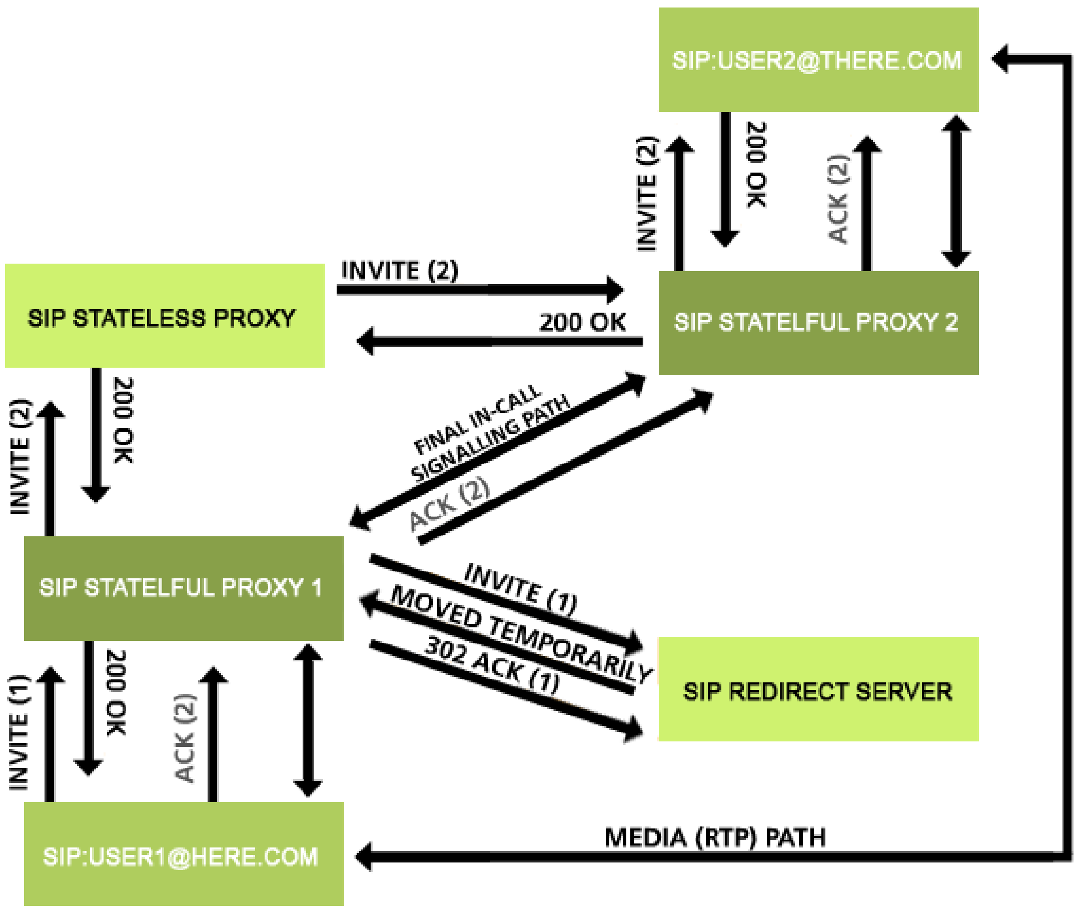

Figure 6 - Example use of SIP Signalling

2.3 IMS (IP Multimedia Subsystem)

The IP Multimedia Subsystem (IMS) environment is an architectural framework for delivering Internet Protocol (IP) multimedia services. To ease the integration with the Internet, IMS uses IETF protocols wherever possible, e.g. Session Initiation Protocol (SIP). According to the 3GPP, IMS is not intended to standardize applications but rather to aid the access of multimedia and voice applications from wireless and wireline terminals, i.e. create a form of fixed-mobile convergence

(FMC). This is done by having a horizontal control layer that isolates the access network from the service layer. From a logical architecture perspective, services need not have their own control functions, as the control layer is a common horizontal layer. However in implementation this does not necessarily map into greater reduced cost and complexity.

2.3.1 Architecture

Several roles of Session Initiation Protocol (SIP) servers or proxies, collectively called Call Session Control Function (CSCF), are used to process SIP signalling packets in the IMS.

The core IMS infrastructure consists of four components, namely: HSS, I/P/S-CSCF.

HSS (Home Subscriber Server)

HSS (Home Subscriber Server) is the master user database that supports the IMS network entities that are actually exploited by the CSCF modules for handling the calls/sessions. The main data stored in HSS include the subscription related information (user profiles), registration information, access parameters and service-triggering information. HSS contains IMS access parameters which include parameters like user authentication, roaming authorization and allocated S-CSCF names. Service-triggering information enables SIP service execution. HSS also provides user-specific requirements for S-CSCF capabilities. This information is used by the I-CSCF to select the most suitable S-CSCF for a user. Finally, HSS contains the subset of Home Location Register and Authentication Center functionality required by the Circuit-Switched and the Packet-Switched Domains.

Specific features of the Home Subscriber Server (HoSS) implementation are:

- Support for the 3GPP Cx Diameter application
- Support for the 3GPP Sh Diameter application
- Support for the 3GPP Zh Diameter application
- Integrated simple AuC functionality
- Java Diameter Stack implementation
- Web-based management console

Call Session Control Functions (CSCF)

The control layer of the IMS infrastructure consists of nodes for managing call establishment, management and release, which are called Call Session Control Functions (CSCF). The CSCF inspects each SIP/SDP message and determines if the signalling should traverse one or more application servers towards its final destination. More specifically, the CSCF is a distributed entity comprised of three different components, the Proxy CSCF, the Interrogating CSCF and the Serving CSCF, described below.

Proxy Call Session Control Function (P-CSCF)

P-CSCF handles signalling between users and the IMS. P-CSCF acts as the entry point for any service invocation within IMS and grants appropriate access rights after successful user authentication. It validates the request, forwards it to selected destinations and processes and forwards the response. The P-CSCF is also the functional entity in charge of IMS session signalling and network QoS control, relaying session and media related info through Diameter protocol to the Policy and Charging Rule Function (PCRF) when the operator wants to apply Policy and Charging Control (PCC). Based on the received information the PCRF is able to derive authorized NQoS information that will be passed to the PCEF within the GGSN. There can be one or more P-CSCFs within an operator's network. P-CSCF communicates with I-CSCF, S-CSCF and the user, so as to forward SIP requests and responses and handle session related information, like registration events, timers, policy issues, etc. OSIMS P-CSCF is also able to firewall the core network at the application level: only registered users are allowed to insert messages inside the IMS network and the P-CSCF asserts the identity of the users. For this, upon registration, the P-CSCF establishes secured channels individually for each User Endpoint (UE) that it services. To keep track of the registered users, it has an internal reversed-registrar that is updated by intercepting the registration process and later by subscribing in User Agent Client (UAC) mode to the registration package at the S-CSCF and receiving notifications. The actual data is kept in a hash-table to allow fast retrieval.

Specific features of the OSIMS P-CSCF implementation are:

- Signalling firewall and user identity assertion (P-Preferred-Identity, P-Asserted-Identity header support)
- Local registrar synchronization through "reg" event [RFC 3680]
- Path header support
- Service-Route verification/enforcement
- Dialog statefulness and Record-Route verification/enforcement

- IPSec set-up using CK and IK from AKA
- Integrity-protection for authentication
- Security-Client, Security-Server, Security-Verify header support
- Visited-Network-ID header support
- NAT support for signalling
- NAT support for media through RTPProxy

Serving Call Session Control Function (S-CSCF)

A Serving-CSCF (S-CSCF) is the central node of the signalling plane. It is a SIP server, but performs session control too. S-CSCF is located in the home network. It is responsible for key routing decisions as it receives all the User Equipment (UE)-originated and UE-terminated sessions and transactions. Therefore it is also responsible for handling registration processes, maintaining session states and storing the service profiles. During a session, S-CSCF maintains a session state and interacts with service platforms and charging functions. There may be multiple S-CSCFs with (possibly) different functionalities within a network. S-CSCF handles registration requests, gathers information about users and performs session control. One of the most important functionalities of S-CSCF is the execution of media-policing. The S-CSCF is able to check the content of the Session Description Protocol payload and check whether it fits to the user's profile.

The OSIMS S-CSCF implementation communicates with the HSS using Diameter (over the Cx interface) to retrieve authentication vectors, update registration information and download the user profiles as specified in 3GPP TS 29.228 . The S-CSCF can apply the user profile based initial Filter Criteria (iFC) to enforce specific SIP routing rules. For fast response times with minimal locking, the registrar of the OSIMS S-CSCF has a complex structure based on hash-tables. The information that is required to relate a user identity to a physical UE is stored here and used further on for call routing. It also accepts subscriptions to registration state events and notifies the subscribers about changes in the registrar. S-CSCF also communicates with P-CSCF and I-CSCF to route mobile-terminating and mobile-originated traffic.

Specific features of the OSIMS S-CSCF implementation are:

- Full Cx interface support
- Authentication through AKAv1-MD5, AKAv2-MD5 and MD5
- Service-Route header support
- Path header support

- P-Asserted-Identity header support
- Visited-Network-ID header support
- Download of Service-Profile from HSS
- Initial Filter Criteria triggering
- ISC interface routing towards Application Servers
- "reg" event server with access restrictions

Interrogating Call Session Control Function (I-CSCF)

An Interrogating-CSCF (I-CSCF) is another SIP function located at the edge of an administrative domain. Its IP address is published in the Domain Name System (DNS) of the domain, so that remote servers can find it, and use it as a forwarding point (e.g. registering) for SIP packets to this domain. I-CSCF acts as a topology-hiding gateway between the P-CSCF and the S-CSCF, by determining the S-CSCF or the AS (Application Server) to which an end-user should register. It has the role of a stateless proxy that, by using the indicated public identities of the caller or the called, queries the Home Subscriber Server (HSS) and based on responses routes the message to the correct S-CSCF. OSIMS S-CSCF implements the Cx interface 3GPP TS 29.228 of an I-CSCF to the HSS. Therefore it supports the required Diameter commands to locate the user-assigned S-CSCF or to select, based on capabilities, a new SCSCF and check identities, roaming authorizations as specified in 3GPP TS 29.228. I-CSCF is a contact point within an operator’s network for all connections destined to a subscriber of that network operator.

Specific features of the OSIMS I-CSCF implementation are:

- Full Cx interface support
- S-CSCF selection based on user capabilities
- Serial forking for forwarding to S-CSCF
- Visited-Network-ID header support and roaming permission verification
- Topology Hiding (THIG)
- Network Domain Security (NDS)
- Server Components (Application Servers)

SIP AS (SIP Application Server)

Application Servers are those components that operate on the media plane and are under the control of IMS Core functions, specifically Media Server (MS) and Media gateway (MGW).

The SIP Application Server is the service relevant part in the IMS. The SIP AS is triggered by the S-CSCF (Serving Call Session Control Function) which redirects certain sessions to the SIP AS based on filter information obtained from the HSS (Home Subscriber Server). Based on local filter rules, the SIP AS then decides which of the applications deployed on the server in which order should be selected for handling the session. During the execution of the service logic it is also possible for the SIP AS to communicate with the HSS to access additional subscriber related information. The Application Server may be located in the user's home network or in a third-party location (in a network or stand-alone AS). SIPSEE (SIP Servlet Execution Environment) is part of the IMS Application Layer and provides multimedia session control capabilities to SIP/HTTP/Diameter converged services in an IMS environment. SIPSEE can act as a SIP proxy, a SIP Redirect or a B2BUA (Back to Back User Agent). The API (Application Programming Interface) used by SIPSEE is compliant with the JSR116 - SIP Servlet API 1.0 which has been adopted from the successful HTTP Servlet API. SIPSEE is designed and implemented to support both SIP and HTTP protocols, so that an IMS Client combining SIP Client and HTTP browser can profit from such converged multi-protocol services, such as Click2Dial, See What I See and rich content IPTV service. It therefore merges the two most used protocols from the Internet and VoIP. An AS may be dedicated to a single service and there may be one or more ASs involved in a single session.

MS Service Control (ISC) Interface

The interface between a SIP AS and the S-CSCF is called ISC (IMS Service Control) interface. Depending on the role of an AS, the procedures of ISC can be divided into two main categories:

1. For incoming session initiating SIP messages, the S-CSCF analyzes them based on initial filter criteria from the user profile as part of the HSS subscriber data and then routes them to the appropriate SIP Application Server for further processing. The AS then can act as User Agent Server (UAS), SIP Proxy or a Redirect Server.
2. The Sip Application Server may initiate own SIP requests and acts as User Agent Client (UAC) or Back to Back User Agent (B2BUA).

Sh Interface

3GPP has defined the Sh interface as part of the IMS for Application Servers to access subscriber profile data in HSS. The protocol used for message exchange is Diameter. Subscriber related data from the HSS can be accessed from SIPSEE with help of a Diameter protocol adapter. Applications are also able to subscribe to special events in the HSS (e.g. change of a special item in user profile, user registration status etc.) and receives notification events sent from HSS.

Policy and Charging Control Function (PCF)

The Policy and Charging Control (PCC) Function is the component responsible for making policy decisions. PCRF decisions are based on session and media-related information taken from the P-CSCF.

Some of the supported functionalities are:

- To store session and media-related information
- To generate PCC rules to install at the PCEF, with session and authorized QoS information
- To provide an authorization decision on receiving a bearer authorization request
- To update the authorization decision at session modifications related to session and media-related information
- The capability to recall the authorization decision at any time
- The capability to enable/prevent the usage of an authorized bearer
- To inform the P-CSCF when the bearer is lost or modified
- To pass an IMS-charging identifier to the Gateway GPRS Support Node and to pass a GPRS-charging identifier to the P-CSCF

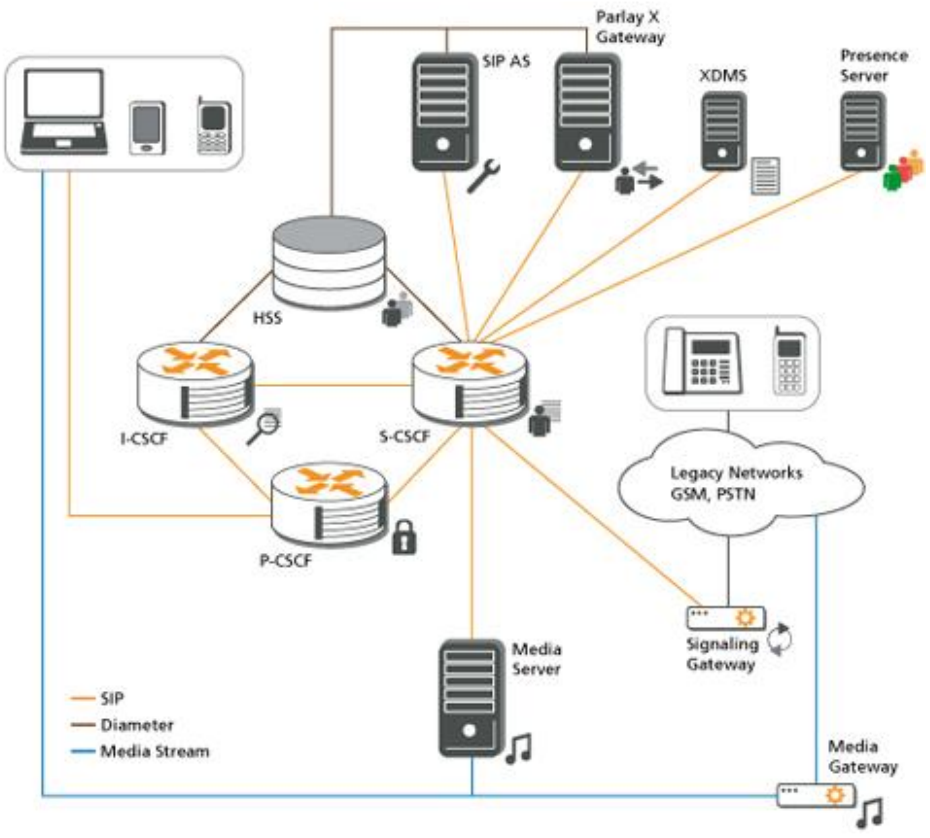

Figure 7 - IMS Topology

2.3.2 Access network

The user can connect to an IMS network in various ways, most of which use the standard Internet Protocol (IP). IMS terminals (such as mobile phones, personal digital assistants (PDAs) and computers) can register directly on an IMS network, even when they are roaming in another network or country (the visited network). The only requirement is that they can use IP and run Session Initiation Protocol (SIP) user agents. Fixed access (e.g., Digital Subscriber Line (DSL), cable modems, Ethernet), mobile access (e.g. W-CDMA, CDMA2000, GSM, GPRS) and wireless access (e.g. WLAN, WiMAX) are all supported. Other phone systems like plain old telephone service (POTS—the old analogue telephones), H.323 and non IMS-compatible VoIP systems, are supported through gateways.

2.4 Android

Repeating parts of the Android Developers Guide, Android is a software stack for mobile devices that includes an operating system, middleware and key applications. It is an open platform for mobile development. The Android SDK provides the tools and APIs necessary to begin developing applications on the Android platform using the Java programming language.

2.4.1 Architecture

The Android operating system includes the following components: **Applications**, **Application Framework**, **Libraries**, **Android Runtime** and **Linux Kernel**.

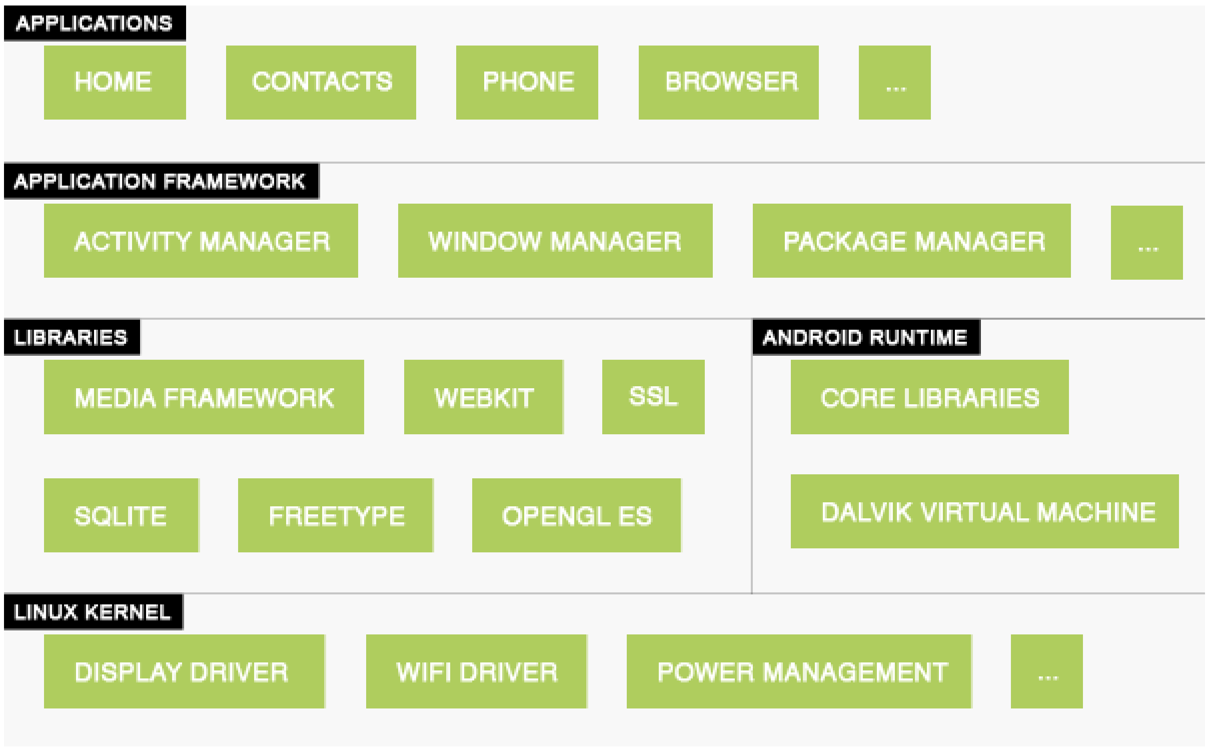

Figure 8 - Major components of the Android operating system

2.4.2 Developing for Android

All applications are written using the Java programming language. By providing an open development platform, Android offers developers the ability to build extremely rich and innovative applications.

Developers are free to take advantage of the device hardware, access location information, run background services, set alarms, add notifications to the status bar, and much, much more. Developers have full access to the same framework APIs used by the core applications. The application architecture is designed to simplify the reuse of components; any application can publish its capabilities and any other application may then make use of those capabilities (subject to security constraints enforced by the framework). This same mechanism allows components to be replaced by the user.

Underlying all applications is a set of services and systems, including:

- A rich and extensible set of **Views** that can be used to build an application, including lists, grids, text boxes, buttons, and even an embeddable web browser.
- **Content Providers** that enable applications to access data from other applications (such as Contacts), or to share their own data.
- A **Resource Manager**, providing access to non-code resources such as localized strings, graphics, and layout files.

- A **Notification Manager** that enables all applications to display custom alerts in the status bar.
- An **Activity Manager** that manages the lifecycle of applications and provides a common navigation backstack.

Android includes a set of C/C++ libraries used by various components of the Android system. These capabilities are exposed to developers through the Android application framework. Some of the core libraries are listed below:

- **System C library** - a BSD-derived implementation of the standard C system library (libc), tuned for embedded Linux-based devices.
- **Media Libraries** - based on PacketVideo's OpenCORE; the libraries support playback and recording of many popular audio and video formats, as well as static image files, including MPEG4, H.264, MP3, AAC, AMR, JPEG, and PNG.
- **Surface Manager** - manages access to the display subsystem and seamlessly composites 2D and 3D graphic layers from multiple applications.
- **LibWebCore** - a modern web browser engine which powers both the Android browser and an embeddable web view.
- **SGL** - the underlying 2D graphics engine.
- **3D libraries** - an implementation based on OpenGL ES 1.0 APIs; the libraries use either hardware 3D acceleration (where available) or the included, highly optimized 3D software rasterizer.
- **FreeType** - bitmap and vector font rendering.
- **SQLite** - a powerful and lightweight relational database engine available to all applications.

Android includes a set of core libraries that provides most of the functionality available in the core libraries of the Java programming language. Every Android application runs in its own process, with its own instance of the Dalvik virtual machine. Dalvik has been written so that a device can run multiple VMs efficiently. The Dalvik VM executes files in the Dalvik Executable (.dex) format which is optimized for minimal memory footprint. The VM is register-based, and runs classes compiled by a Java language compiler that have been transformed into the .dex format by the included "dx" tool.
The Dalvik VM relies on the Linux kernel for underlying functionality such as threading and low-level memory management.

Android relies on Linux version 2.6 for core system services such as security, memory management, process management, network stack, and driver model. The kernel also acts as an abstraction layer between the hardware and the rest of the software stack.

The Android SDK tools compile the code—along with any data and resource files—into an Android package, an archive file with an .apk suffix. All the code in a single .apk file is considered to be one application and is the file that Android-powered devices use to install the application.

Once installed on a device, each Android application lives in its own security sandbox:

- The Android operating system is a multi-user Linux system in which each application is a different user.
- By default, the system assigns each application a unique Linux user ID (the ID is used only by the system and is unknown to the application). The system sets permissions for all the files in an application so that only the user ID assigned to that application can access them.
- Each process has its own virtual machine (VM), so an application's code runs in isolation from other applications.
- By default, every application runs in its own Linux process. Android starts the process when any of the application's components need to be executed, then shuts down the process when it's no longer needed or when the system must recover memory for other applications.

In this way, the Android system implements the principle of least privilege. That is, each application, by default, has access only to the components that it requires to do its work and no more. This creates a very secure environment in which an application cannot access parts of the system for which it is not given permission. However, there are ways for an application to share data with other applications and for an application to access system services:

- It's possible to arrange for two applications to share the same Linux user ID, in which case they are able to access each other's files. To conserve system resources, applications with the same user ID can also arrange to run in the same Linux process and share the same VM (the applications must also be signed with the same certificate).
- An application can request permission to access device data such as the user's contacts, SMS messages, the mountable storage (SD card), camera, Bluetooth, and more. All application permissions must be granted by the user at install time.

That covers the basics regarding how an Android application exists within the system. The rest of this document introduces you to:

- The core framework components that define your application.
- The manifest file in which you declare components and required device features for your application.
- Resources that are separate from the application code and allow your application to gracefully optimize its behavior for a variety of device configurations.

Application components are the essential building blocks of an Android application. Each component is a different point through which the system can enter your application. Not all components are actual entry points for the user and some depend on each other, but each one exists as its own entity and plays a specific role—each one is a unique building block that helps define your application's overall behavior.

There are four different types of application components. Each type serves a distinct purpose and has a distinct lifecycle that defines how the component is created and destroyed. The four different types of application components include: **Activities**, **Services**, **Content providers** and **Broadcast receivers**.

Activities

An activity represents a single screen with a user interface. For example, an email application might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email application, each one is independent of the others. As such, a different application can start any one of these activities (if the email application allows it). For example, a camera application can start the activity in the email application that composes new mail, in order for the user to share a picture.

Services

A service is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface. For example, a service might play music in the background while the user is in a different application, or it might fetch data over the network without blocking user interaction with an activity. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it.

Content providers

A content provider manages a shared set of application data. You can store the data in the file system, an SQLite database, on the web, or any other persistent storage location your application can access. Through the content provider, other applications can query or even modify the data (if the content provider allows it). For example, the Android system provides a content provider that manages the user's contact information. As such, any application with the proper permissions can query part of the content provider (such as ContactsContract.Data) to read and write information about a particular person. Content providers are also useful for reading and writing data that is private to your application and not shared. For example, the Note Pad sample application uses a content provider to save notes.

Broadcast receivers

A broadcast receiver is a component that responds to system-wide broadcast announcements. Many broadcasts originate from the system—for example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured. Applications can also initiate broadcasts—for example, to let other applications know that some data has been downloaded to the device and is available for them to use. Although broadcast receivers don't display a user interface, they may create a status bar notification to alert the user when a broadcast event occurs. More commonly, though, a broadcast receiver is just a "gateway" to other components and is intended to do a very minimal amount of work. For instance, it might initiate a service to perform some work based on the event. A broadcast receiver is implemented as a subclass of BroadcastReceiver and each broadcast is delivered as an Intent object.

A unique aspect of the Android system design is that any application can start another application's component. For example, if you want the user to capture a photo with the device camera, there's probably another application that does that and your application can use it, instead of developing an activity to capture a photo yourself. You don't need to incorporate or even link to the code from the camera application. Instead, you can simply start the activity in the camera application that captures a photo. When complete, the photo is even returned to your application so you can use it. To the user, it seems as if the camera is actually a part of your application.

When the system starts a component, it starts the process for that application (if it is not already running) and instantiates the classes needed for the component. For example, if your application starts the activity in the camera application that captures a photo, that activity runs in the process that belongs to the camera application, not in your application's process. Therefore, unlike applications on most other systems, Android applications don't have a single entry point (there is no main() function, for example).

Because the system runs each application in a separate process with file permissions that restrict access to other applications, your application cannot directly activate a component from another application. The Android system, however, can. So, to activate a component in another application, you must deliver a message to the system that specifies your intent to start a particular component. The system then activates the component for you.

Three of the four component types—activities, services, and broadcast receivers—are activated by an asynchronous message called an intent. Intents bind individual components to each other at runtime, whether the component belongs to your application or another.

An intent is created with an Intent object, which defines a message to activate either a specific component or a specific type of component—an intent can be either explicit or implicit, respectively.

For activities and services, an intent defines the action to perform and may specify the URI of the data to act on. For example, an intent might convey a request for an activity to show an image or to open a web page. In some cases, you can start an activity to receive a result, in which case, the activity also returns the result in an Intent.

For broadcast receivers, the intent simply defines the announcement being broadcast.

The other component type, content provider, is not activated by intents. Rather, it is activated when targeted by a request from a ContentResolver. The content resolver handles all direct transactions with the content provider so that the component that is performing transactions with the provider doesn't need to and instead calls methods on the ContentResolver object. This leaves a layer of abstraction between the content provider and the component requesting information.

Before the Android system can start an application component, the system must know that the component exists by reading the application's **AndroidManifest.xml** file (the "manifest" file). Your application must declare all its components in this file, which must be at the root of the application project directory.

The manifest does a number of things in addition to declaring the application's components, such as:

- Identify any user permissions the application requires, such as Internet access or read-access to the user's contacts.
- Declare the minimum API Level required by the application, based on which APIs the application uses.
- Declare hardware and software features used or required by the application, such as a camera, bluetooth services, or a multitouch screen.
- API libraries the application needs to be linked against (other than the Android framework APIs), such as the Google Maps library.
- And more

The primary task of the manifest is to inform the system about the application's components. For example, a manifest file can declare an activity as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    <application android:icon="@drawable/app_icon.png" ... >
      <activity android:name="com.example.project.ExampleActivity"
                  android:label="@string/example_label" ... >
        </activity>
        ...
    </application>
</manifest>
```

In the `<application>` element, the `android:icon` attribute points to resources for an icon that identifies the application.

In the `<activity>` element, the `android:name` attribute specifies the fully qualified class name of the Activity subclass and the `android:label` attributes specifies a string to use as the user-visible label for the activity.

You must declare all application components this way:

`<activity>` elements for activities
`<service>` elements for services
`<receiver>` elements for broadcast receivers
`<provider>` elements for content providers

Activities, services, and content providers that you include in your source but do not declare in the manifest are not visible to the system and, consequently, can never run. However, broadcast receivers can be either declared in the manifest or created dynamically in code (as BroadcastReceiver objects) and registered with the system by calling `registerReceiver()`.

As discussed above, in Activating Components, you can use an Intent to start activities, services, and broadcast receivers. You can do so by explicitly naming the target component (using the component class name) in the intent. However, the real power of intents lies in the concept of intent actions. With intent actions, you simply describe the type of action you want to perform (and optionally, the data upon which you'd like to perform the action) and allow the system to find a component on the device that can perform the action and start it. If there are multiple components that can perform the action described by the intent, then the user selects which one to use.

The way the system identifies the components that can respond to an intent is by comparing the intent received to the intent filters provided in the manifest file of other applications on the device.

When you declare a component in your application's manifest, you can optionally include intent filters that declare the capabilities of the component so it can respond to intents from other applications. You can declare an intent filter for your component by adding an `<intent-filter>` element as a child of the component's declaration element.

For example, an email application with an activity for composing a new email might declare an intent filter in its manifest entry to respond to "send" intents (in order to send email). An activity in your application can then create an intent with the “send” action (`ACTION_SEND`), which the system matches to the email application’s “send” activity and launches it when you invoke the intent with `startActivity()`.

There are a variety of devices powered by Android and not all of them provide the same features and capabilities. In order to prevent your application from being installed on devices that lack features needed by your application, it's important that you clearly define a profile for the types of devices your application supports by declaring device and software requirements in your manifest file. Most of these declarations are informational only and the system does not read them, but external services such as Android Market do read them in order to provide filtering for users when they search for applications from their device.

For example, if your application requires a camera and uses APIs introduced in Android 2.1 (API Level 7), you should declare these as requirements in your manifest file. That way, devices that do not have a camera and have an Android version lower than 2.1 cannot install your application from Android Market.

However, you can also declare that your application uses the camera, but does not require it. In that case, your application must perform a check at runtime to determine if the device has a camera and disable any features that use the camera if one is not available.

Here are some of the important device characteristics that you should consider as you design and develop your application:

Screen size and density

In order to categorize devices by their screen type, Android defines two characteristics for each device: screen size (the physical dimensions of the screen) and screen density (the physical density of the pixels on the screen, or dpi—dots per inch). To simplify all the different types of screen configurations, the Android system generalizes them into select groups that make them easier to target.

The screen sizes are: small, normal, large, and extra-large.
The screen densities are: low density, medium density, high density, and extra high density.

By default, your application is compatible with all screen sizes and densities, because the Android system makes the appropriate adjustments to your UI layout and image resources. However, you should create specialized layouts for certain screen sizes and provide specialized images for certain densities, using alternative layout resources, and by declaring in your manifest exactly which screen sizes your application supports with the `<supports-screens>` element.

Input configurations

Many devices provide a different type of user input mechanism, such as a hardware keyboard, a trackball, or a five-way navigation pad. If your application requires a particular kind of input hardware, then you should declare it in your manifest with the `<uses-configuration>` element. However, it is rare that an application should require a certain input configuration.

Device features

There are many hardware and software features that may or may not exist on a given Android-powered device, such as a camera, a light sensor, bluetooth, a certain version of OpenGL, or the fidelity of the touchscreen. You should never assume that a certain feature is available on all Android-powered devices (other than the availability of the standard Android library), so you should declare any features used by your application with the `<uses-feature>` element.

Platform Version

Different Android-powered devices often run different versions of the Android platform, such as Android 1.6 or Android 2.3. Each successive version often includes additional APIs not available in the previous version. In order to indicate which set of APIs are available, each platform version specifies an API Level (for example, Android 1.0 is API Level 1 and Android 2.3 is API Level 9). If you use any APIs that were added to the platform after version 1.0, you should declare the minimum API Level in which those APIs were introduced using the <uses-sdk> element.

An Android application is composed of more than just code—it requires resources that are separate from the source code, such as images, audio files, and anything relating to the visual presentation of the application. For example, you should define animations, menus, styles, colors, and the layout of activity user interfaces with XML files. Using application resources makes it easy to update various characteristics of your application without modifying code and—by providing sets of alternative resources—enables you to optimize your application for a variety of device configurations (such as different languages and screen sizes).

For every resource that you include in your Android project, the SDK build tools define a unique integer ID, which you can use to reference the resource from your application code or from other resources defined in XML. For example, if your application contains an image file named `logo.png` (saved in the `res/drawable/` directory), the SDK tools generate a resource ID named `R.drawable.logo`, which you can use to reference the image and insert it in your user interface.

One of the most important aspects of providing resources separate from your source code is the ability for you to provide alternative resources for different device configurations. For example, by defining UI strings in XML, you can translate the strings into other languages and save those strings in separate files. Then, based on a language qualifier that you append to the resource directory's name (such as `res/values-fr/` for French string values) and the user's language setting, the Android system applies the appropriate language strings to your UI.

Android supports many different qualifiers for your alternative resources. The qualifier is a short string that you include in the name of your resource directories in order to define the device configuration for which those resources should be used. As another example, you should often create different layouts for your activities, depending on the device's screen orientation and size. For example, when the device screen is in portrait orientation (tall), you might want a layout with buttons to be vertical, but when the screen is in landscape orientation (wide), the buttons should be aligned horizontally. To change the layout depending on the orientation, you can define two different layouts and apply the appropriate qualifier to each layout's directory name. Then, the system automatically applies the appropriate layout depending on the current device orientation.

At this point it is not necessary to go further on developing for Android. If you are more interested in this particular subject, feel free to browse online for more documentation here.

3 Testbed

In this chapter you will be introduced to the techniques used to approach this implementation. As described before, four of the core technologies utilized in this project include PQoS, SIP, IMS and Android. Given that the main aspect of this project includes developing a SIP/IMS compatible PQoS-aware Terminal Application Module (TAM) for Android, the following shall present the analysis and development process of such software.

To start with, the core components that make up this project include the Android application along with its metering features and communication functionalities and the MSMM SIP Agent located inside the MCMS (Multimedia Content Management System).

3.1 Design

To be more specific regarding the design approach, an application was developed on top of the Android operating system. This particular application was able to handle incoming and outgoing SIP messages using jain-sip, a low level Java API specification for SIP Signalling. The Java APIs for Integrated Networks (JAIN) is a JCP work group managing telecommunication standards. At the time this is written, Android provides native access to Session Initiation Protocol (SIP) functionality, which is available since API level 9.

SIP signalling was used for communication between the application residing on the phone and the MSMM SIP Agent. Other than that, once the application opened the video stream it was constantly monitoring the packet loss of the transmission in favor of calculating the Perceived Quality of Service. Moreover, due to the experimentation and demonstrative aspect of this application the whole process was designed to simulate a step-by-step implementation. In practice this was done by breaking down the application to several activities to simply demonstrate step-by-step the entire process. Moreover, a 3-second-delay was added after each activity. However the whole process could be done by using just one activity.

In practice, when launching the application the first activity comes up with the message ***SIP ready*** in the center of the screen. In fact, nothing happens in the background while this message is on screen. The reason of this lies in psychological stimulation which aims to prepare the user for what follows.

The second activity involves the SIP messaging, a process which happens in the background while the user only sees the message ***SIP process*** on the screen.

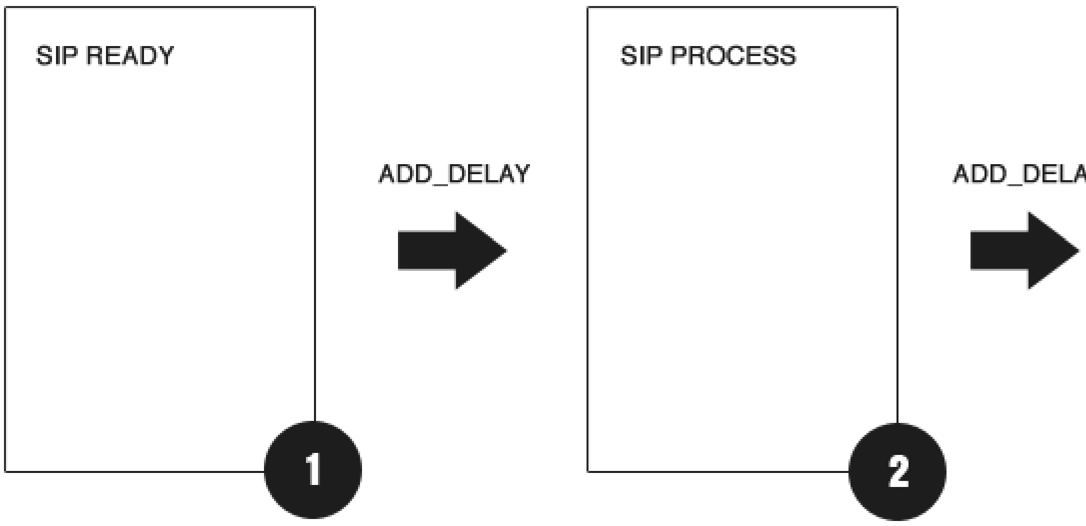

Figure 9 - Initializing & SIP signalling

While ***SIP process*** is on the screen, the process that takes place in the background is actually trying to communicate with the MSMM SIP Agent located inside the MCMS. This involves sending and receiving SIP messages via the mobile device. More specifically, this process is sending a SIP message to the IMS domain and after that it waits for a SIP response message which shall include an RTSP URL to the video to be streamed.

The video is hosted in a linux machine that runs a streaming service using the Darwin Streaming Server (DSS). Darwin Streaming Server is the open source version of the QuickTime Streaming Server allowing you to stream hinted QuickTime, MPEG-4, and 3GPP files over the Internet via the industry standard RTP and RTSP protocols.

If the SIP signalling is successful the application retrieves the RTSP URL and saves it on a local variable in order to use it in the next step, the third activity.

The third activity is the one that actually does something on screen apart from just showing a text message. This is where the application uses the RTSP URL, saved earlier, and starts playing the video streamed from the streaming server located in the IMS domain.

While this activity is on screen, there is a very important process that takes place in the background. The application, having stored the RTSP URL, it starts opening the video streamed while in the background it is using the DatagramSocket and the DatagramPacket classes to constantly parse header information of every incoming packet coming from the streaming server.

Using the RTP transport protocol for the video streaming has its advantages. RTP has important properties of a transport protocol: it runs on end systems, it provides demultiplexing. It differs from transport protocols like TCP in that it does not offer any form of reliability or a protocol-defined flow/congestion control. However, it provides the necessary hooks for adding reliability, where appropriate, and flow/congestion control. Some like to refer to this property as application-level framing. RTP so far has been mostly implemented within applications, but that has no bearing on its role. TCP is still a transport protocol even if it is implemented as part of an application rather than the operating system kernel.

No end-to-end protocol, including RTP, can ensure in-time delivery. This always requires the support of lower layers that actually have control over resources in switches and routers. RTP provides functionality suited for carrying real-time content, e.g., a timestamp and control mechanisms for synchronizing different streams with timing properties.

As currently defined, RTP does not define any mechanisms for recovering for packet loss. Such mechanisms are likely to be highly dependent on the packet content. For example, for audio, it has been suggested to add low-bit-rate redundancy, offset in time. For other applications, retransmission of lost packets may be appropriate. This requires no additions to RTP. RTP probably has the necessary header information (like sequence numbers) for some forms of error recovery by retransmission.

The timestamp is used to place the incoming audio and video packets in the correct timing order. The sequence number is mainly used to detect losses. Sequence numbers increase by one for each RTP packet transmitted, timestamps increase by the time "covered" by a packet. For video formats where a video frame is split across several RTP packets, several packets may have the same timestamp.

As obvious it is, this project was based on the assumption that you can calculate the lost packets of a transmission using the RTP transport protocol. Further research upon this topic would result in taking into consideration several more fields used in the RTP header.

While the application constantly tries to parse the header information of every incoming packet coming from the streaming server, it extracts the sequence number which is a two-byte number located after the first two octets of the header.

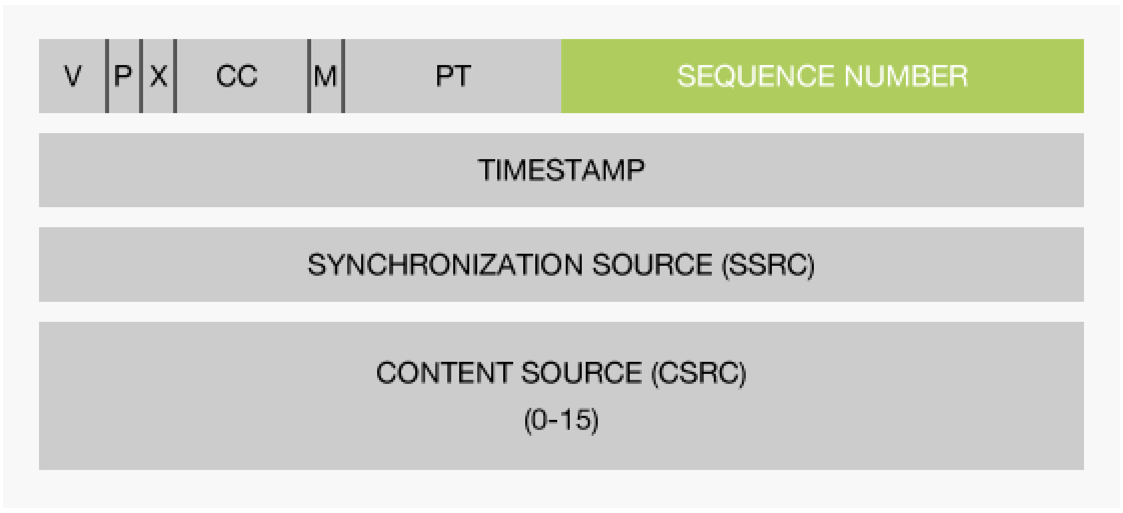

Figure 10 - RTP header

The first twelve octets are present in every RTP packet, while the list of CSRC identifiers is present only when inserted by a mixer. The fields have the following meaning:

Version (V): 2 bits

This field identifies the version of RTP. The version defined by this specification is two (2).

Padding (P): 1 bit

If the padding bit is set, the packet contains one or more additional padding octets at the end which are not part of the payload. The last octet of the padding 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 or for carrying several RTP packets in a lower-layer protocol data unit.

Extension (X): 1 bit

If the extension bit is set, the fixed header MUST be followed by exactly one header extension.

CSRC count (CC): 4 bits

The CSRC count contains the number of CSRC identifiers that follow the fixed header.

Marker (M): 1 bit

The interpretation of the marker is defined by a profile. It is intended to allow significant events such as frame boundaries to be marked in the packet stream. 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.

Payload type (PT): 7 bits

This field identifies the format of the RTP payload and determines its interpretation by the application. A profile MAY specify a default static mapping of payload type codes to payload formats. Additional payload type codes MAY be defined dynamically through non-RTP means. An RTP source MAY change the payload type during a session, but this field SHOULD NOT be used for multiplexing separate media streams. A receiver MUST ignore packets with payload types that it does not understand.

Sequence number: 16 bits

The sequence number 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. 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 according to the method in Section 9.1, because the packets may flow through a translator that does.

Timestamp: 32 bits

The timestamp reflects the sampling instant of the first octet in the RTP data packet. The sampling instant MUST be derived from a clock that increments monotonically and linearly in time to allow synchronization and jitter calculations. The resolution of the clock MUST be sufficient for the desired synchronization accuracy and for measuring packet arrival jitter (one tick per video frame is typically not sufficient). The clock frequency is dependent on the format of data carried as payload and is specified statically in the profile or payload format specification that defines the format, or MAY be specified dynamically for payload formats defined through non-RTP means. If RTP packets are generated periodically, the nominal sampling instant as determined from the sampling clock is to be used, not a reading of the system clock. As an example, for fixed-rate audio the timestamp clock would likely increment by one for each sampling period. If an audio application reads blocks covering 160 sampling periods from the input device, the timestamp would be increased by 160 for each such block, regardless of whether the block is transmitted in a packet or dropped as silent.

The initial value of the timestamp SHOULD be random, as for the sequence number. Several consecutive RTP packets will have equal timestamps if they are (logically) generated at once, e.g., belong to the same video frame. Consecutive RTP packets MAY contain timestamps that are not monotonic if the data is not transmitted in the order it was sampled, as in the case of MPEG interpolated video frames. (The sequence numbers of the packets as transmitted will still be monotonic.)

RTP timestamps from different media streams may advance at different rates and usually have independent, random offsets. Therefore, although these timestamps are sufficient to reconstruct the timing of a single stream, directly comparing RTP timestamps from different media is not effective for synchronization. Instead, for each medium the RTP timestamp is related to the sampling instant by pairing it with a timestamp from a reference clock (wallclock) that represents the time when the data corresponding to the RTP timestamp was sampled. The reference clock is shared by all media to be synchronized. The timestamp pairs are not transmitted in every data packet, but at a lower rate in RTCP SR packets.

The sampling instant is chosen as the point of reference for the RTP timestamp because it is known to the transmitting endpoint and has a common definition for all media, independent of encoding delays or other processing. The purpose is to allow synchronized presentation of all media sampled at the same time.

Applications transmitting stored data rather than data sampled in real time typically use a virtual presentation timeline derived from wallclock time to determine when the next frame or other unit of each medium in the stored data should be presented. In this case, the RTP timestamp would reflect the presentation time for each unit. That is, the RTP timestamp for each unit would be related to the wallclock time at which the unit becomes current on the virtual presentation timeline. Actual presentation occurs some time later as determined by the receiver.

An example describing live audio narration of prerecorded video illustrates the significance of choosing the sampling instant as the reference point. In this scenario, the video would be presented locally for the narrator to view and would be simultaneously transmitted using RTP. The "sampling instant" of a video frame transmitted in RTP would be established by referencing its timestamp to the wallclock time when that video frame was presented to the narrator. The sampling instant for the audio RTP packets containing the narrator's speech would be established by referencing the same wallclock time when the audio was sampled.

The audio and video may even be transmitted by different hosts if the reference clocks on the two hosts are synchronized by some means such as NTP. A receiver can then synchronize presentation of the audio and video packets by relating their RTP timestamps using the timestamp pairs in RTCP SR packets.

SSRC: 32 bits

The SSRC field identifies the synchronization source. This identifier SHOULD be chosen randomly, with the intent that no two synchronization sources within the same RTP session will have the same SSRC identifier. Although the probability of multiple sources choosing the same identifier is low, all RTP implementations must be prepared to detect and resolve collisions. If a source changes its source transport address, it must also choose a new SSRC identifier to avoid being interpreted as a looped source.

CSRC list: 0 to 15 items, 32 bits each

The CSRC list identifies the contributing sources for the payload contained in this packet. The number of identifiers is given by the CC field. If there are more than 15 contributing sources, only 15 can be identified. CSRC identifiers are inserted by mixers, using the SSRC identifiers of contributing sources. For example, for audio packets the SSRC identifiers of all sources, that were mixed together to create a packet, are listed, allowing correct talker indication at the receiver.

Now while this parsing takes place, the application also periodically, that is a time interval of 10 seconds, checks for the number of the lost packets by checking the differences in the sequence number field of the packets. Additionally, every 10 seconds, the application calculates the percentage of the lost packets during that time interval. Onwards, the application builds up a set of information bundle that includes the number of the lost packets, the number of the packets received and of course the percentage of the lost packets during that time interval. This set of information is locally stored in a file and holds data regarding the streaming session. Having these data records the application can calculate the Perceived Quality of Service (PQoS) and saves the outcome in a file according to the following:

```
if (PQoS >= 3)
      write _to_file(NONE + PQoS);
else if (2 < PQoS < 3)
      write _to_file(WARNING + PQoS);
else if (PQoS <= 2)
      write _to_file(RED + PQoS);
```

Given the PQoS value, the application should know when to send an alarm to the MSMM SIP Agent and what kind of alarm. Sending the alarms is also done via SIP signalling.

After the streaming session is over the application launches the last activity that serves for informing the user that the video playback finished.

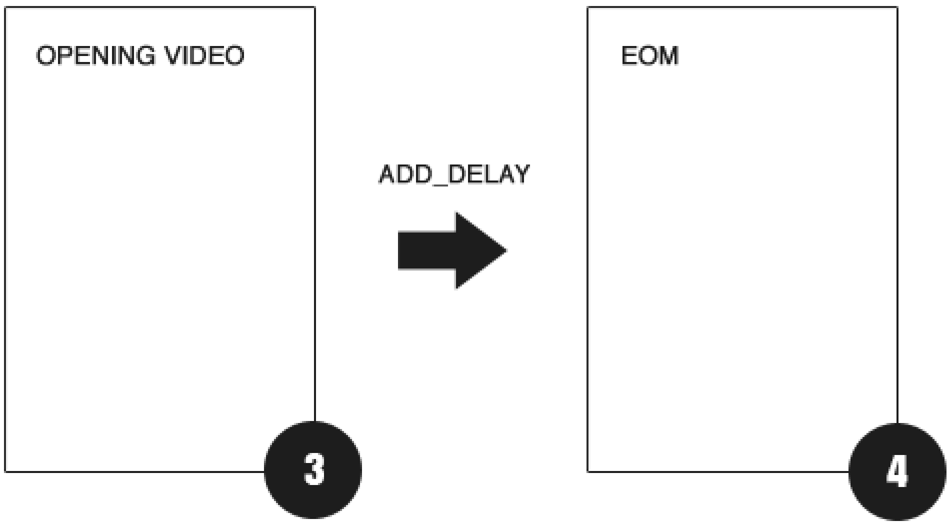

Figure 11 - Video playback & Ending screen

Regarding the MSMM SIP Agent located inside the MCMS (Multimedia Content Management System), this is the part which receives the SIP messages that are sent via the Android application. In other words, this is the part that handles all the communication between the mobile device and the IMS environment. To be more specific, the MSMM SIP Agent, except for receiving the SIP messages containing the alarms, it was also configured to respond with a SIP message containing the RTSP URL of the video to be streamed upon request.

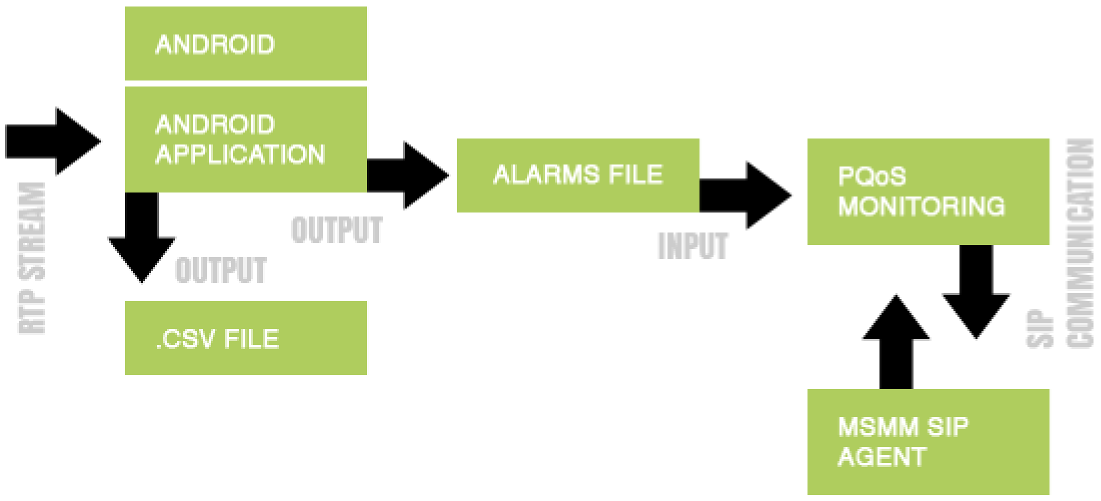

Figure 12 - PQoS monitoring

Due to the fact that this project mainly investigates the provision of IPTV services through the IMS environment, someone should get a grasp of how the IMS handles an IPTV scenario.

A user that would like to watch an IPTV streaming video sends an INVITE signal to the system which then sends an INVITE to the MSRF. The MSFR then sends an OK signal to the system which sends it to the session originator. The session is then setup and the user can watch the IPTV streaming video. In order for this to work the user should be registered to the IMS network and the MCMS. Given these, the user can open a media channel with the MSRF and display the IPTV streaming video.

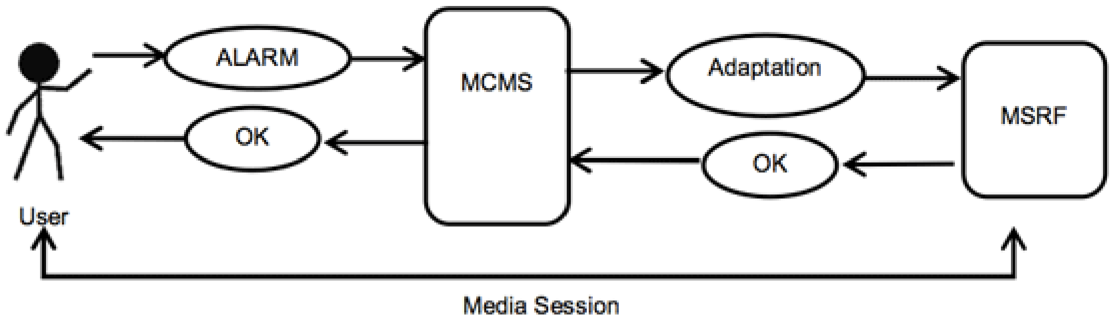

Figure 13 - Use case where a user wants to watch an IPTV streaming video

Now in the case where the user has an active multimedia session registered to the MCMS and is experiencing PQoS degradation, the system will try to notify the IMS environment about this. Onwards, the system will try to enhance the end user PQoS.

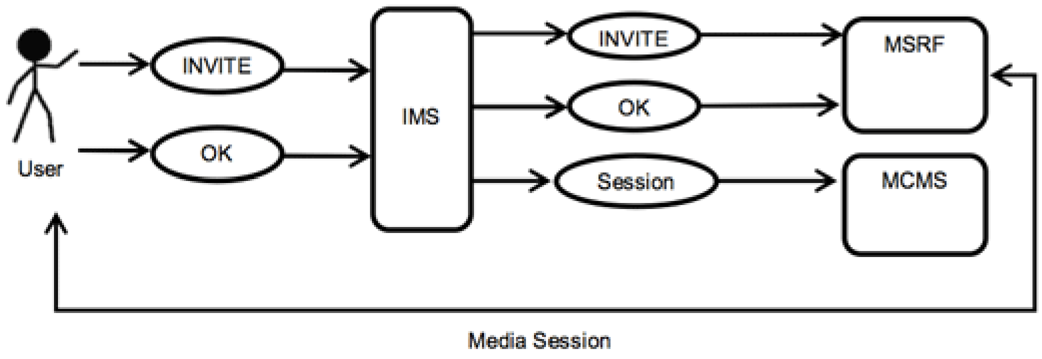

Figure 14 - Use case for unicast IPTV with problem at the server side

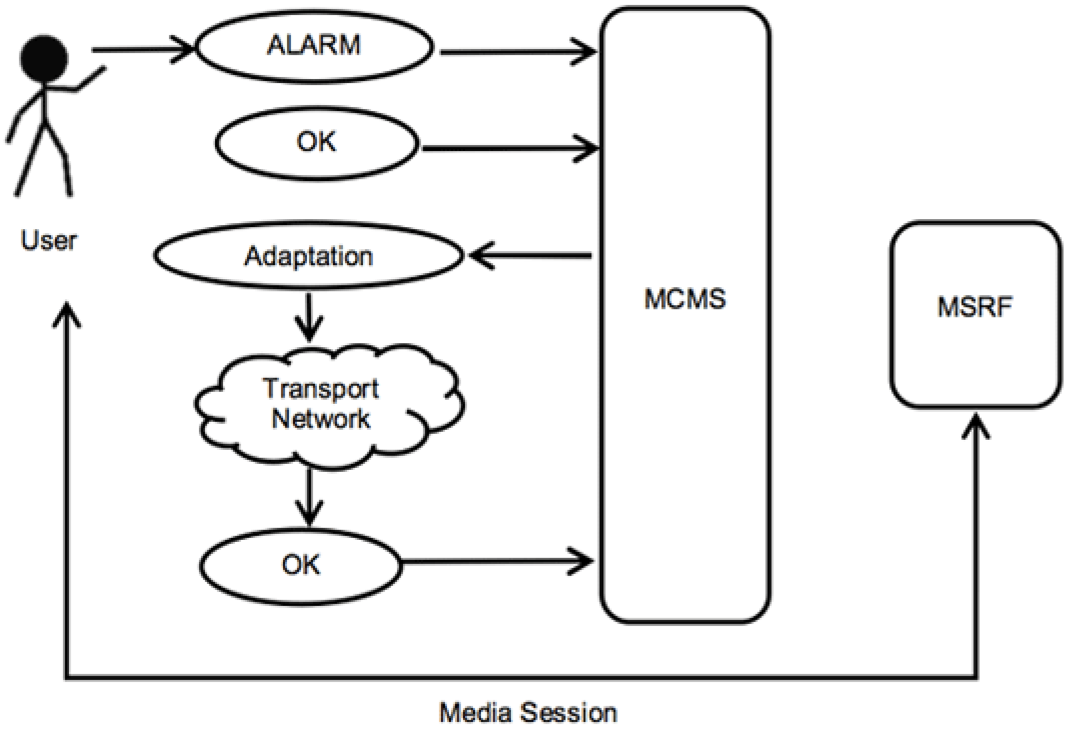

Figure 15 - Use case for unicast IPTV with problem at the Core Network

3.2 Implementation

As far as the MCMS modules are concerned, these modules are implemented by using modified UCT IMS Client which is designed to work with the Fraunhofer FOKUS Open IMS Core.

To make things work with the MSMM SIP Agent, it was necessary to edit the `ims_interface_event_handler.c` file. Inside this file there is a method called `ims_send_instant_message` where the following important line was added:

```
osip_message_set_body(message,
"rtsp://143.233.227.109/sample_h264_100kbit.mp4", 41);
```

As it it shown, a response message body was added including the RTSP URL of the video to be streamed. This particular video was included in the default playlist of the Darwin Streaming Server. The video codec of this MPEG-4 video was H.264 and the duration of the video was 1 minute and 10 seconds.

Moving forward to the implementation of the software, apart from the configuration needed at the end of the IMS environment, the most critical information includes the implementation of the SIP/IMS compatible client application for Android.

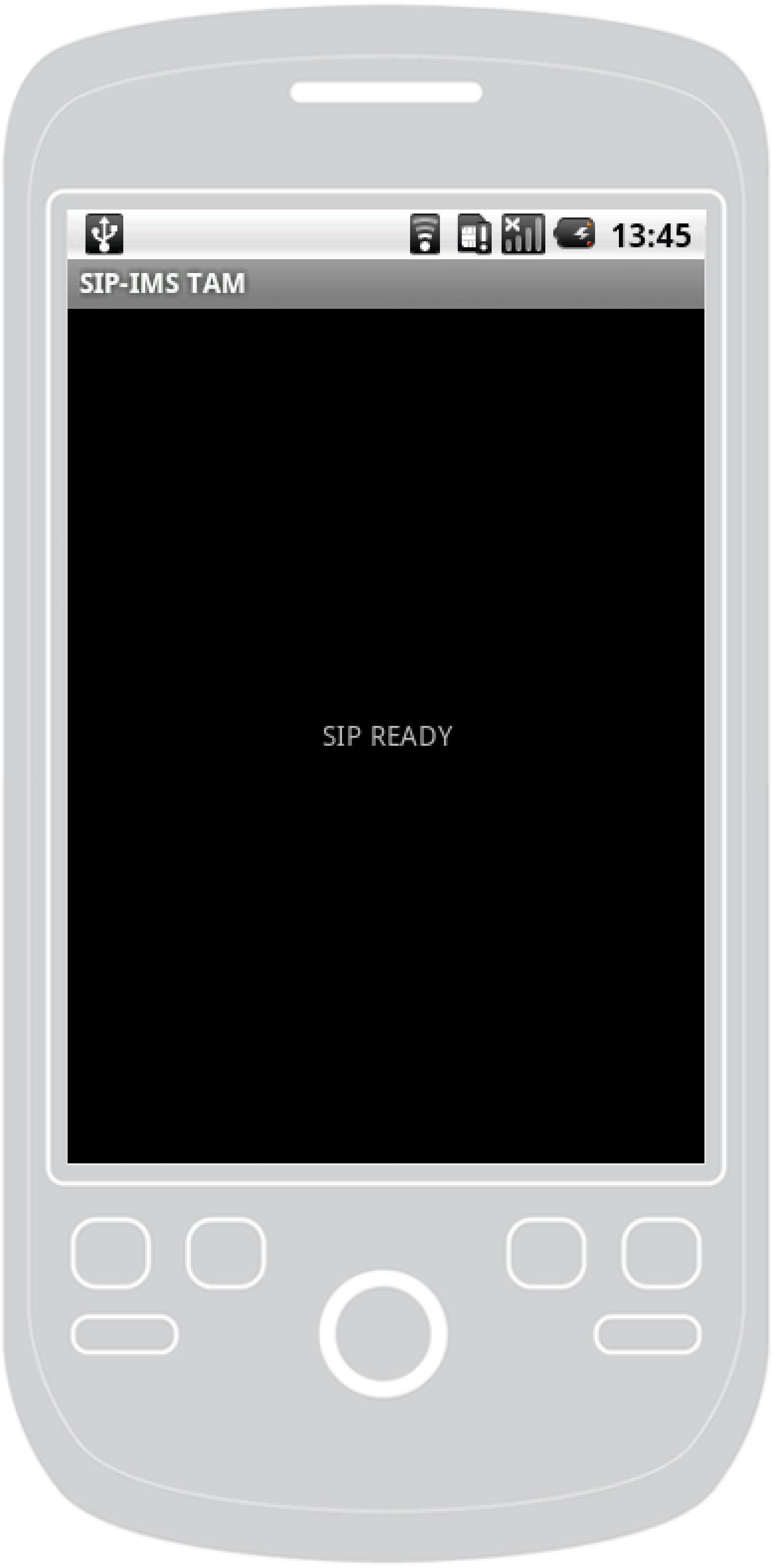

Figure 16 - Device screenshot while initializing the application

As mentioned before, this particular application was able to handle incoming and outgoing SIP messages using jain-sip, a low level Java API specification for SIP Signalling. For this reason, four external JARs were used in this project including:

1. `concurrent.jar`
2. `jain-sip-api-1.2.jar`
3. `jain-sip-ri-1.2.85.jar`
4. `log4j-1.2.8.jar`

As for the Android application, the example of Shootist and Shootme, included in the source code repository of jain-sip, were used as a testbed. Important modifications upon the Shootist were made in order for the application to successfully communicate with the IMS via SIP signalling. In particular, appropriate registration information was added to allow communication through the IMS realm.

To build up a SIP message the following are required:

- Create main elements
- Create message
- Complete message
- Send message

More specifically, the following main SIP elements are minimally needed to construct a message using the JAIN SIP API:

- Request URI
- Method
- Call-ID header
- CSeq header
- From header
- An array of Via headers
- Max-forwards header

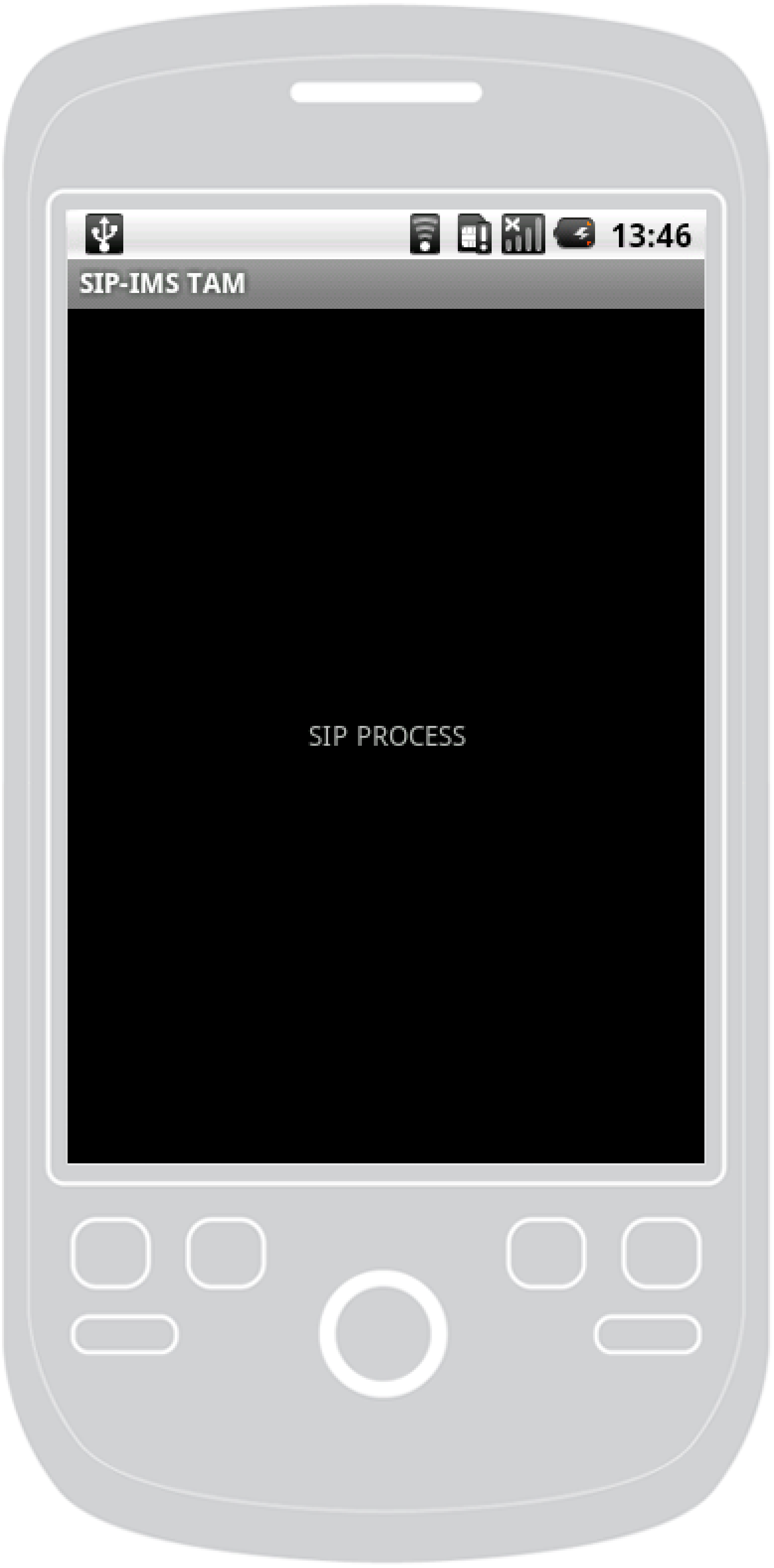

Figure 17 - Device screenshot while SIP signalling takes place in the background

The following code snippet creates all of these elements:

```
public void sendMessage(String to, String message) throws
            ParseException, InvalidArgumentException, SipException {
      SipURI from = addressFactory.createSipURI(getUsername(),
      getHost() + ":" + getPort());
      Address fromNameAddress = addressFactory.createAddress(from);
      fromNameAddress.setDisplayName(getUsername());
      FromHeader fromHeader =
            headerFactory.createFromHeader(fromNameAddress,
            "android-sip-client");

      String username = to.substring(to.indexOf(":")+1,
            to.indexOf("@"));
      String address = to.substring(to.indexOf("@")+1);

      SipURI toAddress =
            addressFactory.createSipURI(username, address);
      Address toNameAddress =
            addressFactory.createAddress(toAddress);
      toNameAddress.setDisplayName(username);
      ToHeader toHeader =
            headerFactory.createToHeader(toNameAddress, null);

      SipURI requestURI =
            addressFactory.createSipURI(username, address);
      requestURI.setTransportParam("udp");

      ArrayList viaHeaders = new ArrayList();
      ViaHeader viaHeader =
            headerFactory.createViaHeader(
                  getHost(),
                  getPort(),
                  "udp",
                  null);
      viaHeaders.add(viaHeader);

      CallIdHeader callIdHeader = sipProvider.getNewCallId();
```

```
	CSeqHeader cSeqHeader =
		headerFactory.createCSeqHeader(1, Request.MESSAGE);

	MaxForwardsHeader maxForwards =
		headerFactory.createMaxForwardsHeader(70);
	...
```

Here is specifically what was added inside the register() method of the Shootist class:

```
private void register() throws Exception {
	sipFactory = SipFactory.getInstance();
	sipFactory.setPathName("gov.nist");
	Properties properties = new Properties();
	properties.setProperty("javax.sip.STACK_NAME", "TestSIP");
	properties.setProperty("javax.sip.IP_ADDRESS", myAddress);
	properties.setProperty("javax.sip.OUTBOUND_PROXY",
peerHostPort + "/" + transport);

	sipStack = sipFactory.createSipStack(properties);
	headerFactory = sipFactory.createHeaderFactory();
	addressFactory = sipFactory.createAddressFactory();
	messageFactory = sipFactory.createMessageFactory();
	lp = sipStack.createListeningPoint(myAddress, myPort,
ListeningPoint.UDP);
	sipProvider = sipStack.createSipProvider(lp);
	sipProvider.addSipListener(this);

	String fromName = "jack";
	String fromSipAddress = "ims.adamantium.gr:5060";
	String fromDisplayName = "jack";
	String Password = "jack";

	String toUser = "kate";
	String toSipAddress = "ims.adamantium.gr:4060";
	String toDisplayName = "kate";
```

```
      // Create From Header
      SipURI fromAddress = addressFactory.createSipURI(fromName,
fromSipAddress);
      Address fromNameAddress =
addressFactory.createAddress(fromAddress);
      fromNameAddress.setDisplayName(fromDisplayName);
      FromHeader fromHeader =
headerFactory.createFromHeader(fromNameAddress, "TestSIP");

      // Create To Header
      SipURI toAddress = addressFactory.createSipURI(toUser,
toSipAddress);
      Address toNameAddress =
addressFactory.createAddress(toAddress);
      toNameAddress.setDisplayName(toDisplayName);
      ToHeader toHeader =
headerFactory.createToHeader(toNameAddress, null);

      // Create Request URI
      SipURI requestURI = addressFactory.createSipURI(toUser,
toSipAddress);
      requestURI.setTransportParam("udp");

      // Create ViaHeaders
      ArrayList viaHeaders = new ArrayList();
      ViaHeader viaHeader =
            headerFactory.createViaHeader(
                  lp.getIPAddress(),
                  lp.getPort(),
                  lp.getTransport(),
                  null);

      // add via headers
      viaHeaders.add(viaHeader);
      // Create ContentTypeHeader
      contentTypeHeader =
headerFactory.createContentTypeHeader("application", "sdp");

      // Create a new CallId header
      CallIdHeader callIdHeader = sipProvider.getNewCallId();
      CSeqHeader cSeqHeader = headerFactory.createCSeqHeader(1,
Request.MESSAGE);

      // Create a new MaxForwardsHeader
      MaxForwardsHeader maxForwards =
headerFactory.createMaxForwardsHeader(70);

      // Create the request.
      Request request =
            messageFactory.createRequest(
                  requestURI,
                  Request.MESSAGE,
                  callIdHeader,
                  cSeqHeader,
                  fromHeader,
                  toHeader,
                  viaHeaders,
                  maxForwards);

      // Create the contact name address.
      //SipURI contactUrl = addressFactory.createSipURI(fromName,
ipAddress);
      SipURI contactURI = addressFactory.createSipURI(fromName,
myAddress);
      contactURI.setPort(lp.getPort());



      Address contactAddress =
addressFactory.createAddress(contactURI);
      contactAddress.setDisplayName(fromName);
      contactHeader =
headerFactory.createContactHeader(contactAddress);
      request.addHeader(contactHeader);

      ContentTypeHeader contentTypeHeader =
headerFactory.createContentTypeHeader("text", "plain");
      String message = "Request RTSP";
      request.setContent(message, contentTypeHeader);

      sipProvider.sendRequest(request);
}
```

Getting the RTSP URL from the SIP response was done modifying the `processRequest` method, with the following:

```
ServerTransaction serverTransactionId =
requestEvent.getServerTransaction();
Request request = requestEvent.getRequest();
String content = new String(request.getRawContent());
```

Emulating the 3-second delay was done using Handlers. A Handler allows you to send and process `Message` and Runnable objects associated with a thread's `MessageQueue`. Each Handler instance is associated with a single thread and that thread's message queue. When you create a new Handler, it is bound to the thread / message queue of the thread that is creating it -- from that point on, it will deliver messages and runnables to that message queue and execute them as they come out of the message queue.

There are two main uses for a Handler: (1) to schedule messages and runnables to be executed as some point in the future; and (2) to enqueue an action to be performed on a different thread than your own.

Scheduling messages is accomplished with the `post(Runnable)`, `postAtTime(Runnable, long)`, `postDelayed(Runnable, long)`, `sendEmptyMessage(int)`, `sendMessage(Message)`, `sendMessageAtTime(Message, long)`, and `sendMessageDelayed(Message, long)` methods. The post versions allow you to enqueue

Runnable objects to be called by the message queue when they are received; the sendMessage versions allow you to enqueue a `Message` object containing a bundle of data that will be processed by the Handler's `handleMessage(Message)` method (requiring that you implement a subclass of Handler). When posting or sending to a Handler, you can either allow the item to be processed as soon as the message queue is ready to do so, or specify a delay before it gets processed or absolute time for it to be processed. The latter two allow you to implement timeouts, ticks, and other timing-based behavior. When a process is created for your application, its main thread is dedicated to running a message queue that takes care of managing the top-level application objects (activities, broadcast receivers, etc) and any windows they create. You can create your own threads, and communicate back with the main application thread through a Handler. This is done by calling the same post or sendMessage methods as before, but from your new thread. The given Runnable or Message will then be scheduled in the Handler's message queue and processed when appropriate.

Here is the piece of the code used to emulate the delay.

```
private void threeSecondDelay() {
      handler.post(new Runnable() {
            public void run() {
                  handler.postDelayed(this, 3000);
                  Log.v(TAG, "3-second delay");
            }
      });
}
```

Figure 18 - Device screenshot during video playback

During the video playback on the mobile device, network losses were added to the network using `tc`.

```
echo  ./loss.sh add loss_percentage rate_in_mbps
tc qdisc del dev lo root 2>/dev/null>/dev/null
tc qdisc $1 dev lo root handle 1: prio

echo BE
tc qdisc $1 dev lo parent 1:1 handle 20: netem loss $2%
tc qdisc $1 dev lo parent 20:1 tbf rate $3Mbit buffer 1600 limit
3000
tc filter $1 dev lo protocol ip parent 1: prio 2 u32 match ip tos
0x00 0xff flowid 1:1
```

For the video playback a simple code snippet was used to get the MediaPlayer stared. MediaPlayer class can be used to control playback of audio/video files and streams. Playback control of audio/video files and streams is managed as a state machine.

```
String LINK = "rtsp://143.233.27.121/sample_h264_100kbit.mp4";
setContentView(R.layout.mediaplayer);
VideoView videoView = (VideoView) findViewById(R.id.video);
MediaController mc = new MediaController(this);
mc.setAnchorView(videoView);
mc.setMediaPlayer(videoView);
Uri video = Uri.parse(LINK);
videoView.setMediaController(mc);
videoView.setVideoURI(video);
videoView.start();
```

Setting the `LINK` variable was done dynamically as soon as the application received the SIP response from the MSMM SIP Agent containing the RTSP URL of the video to be streamed.

And here is the screenshot after finishing the video streaming.

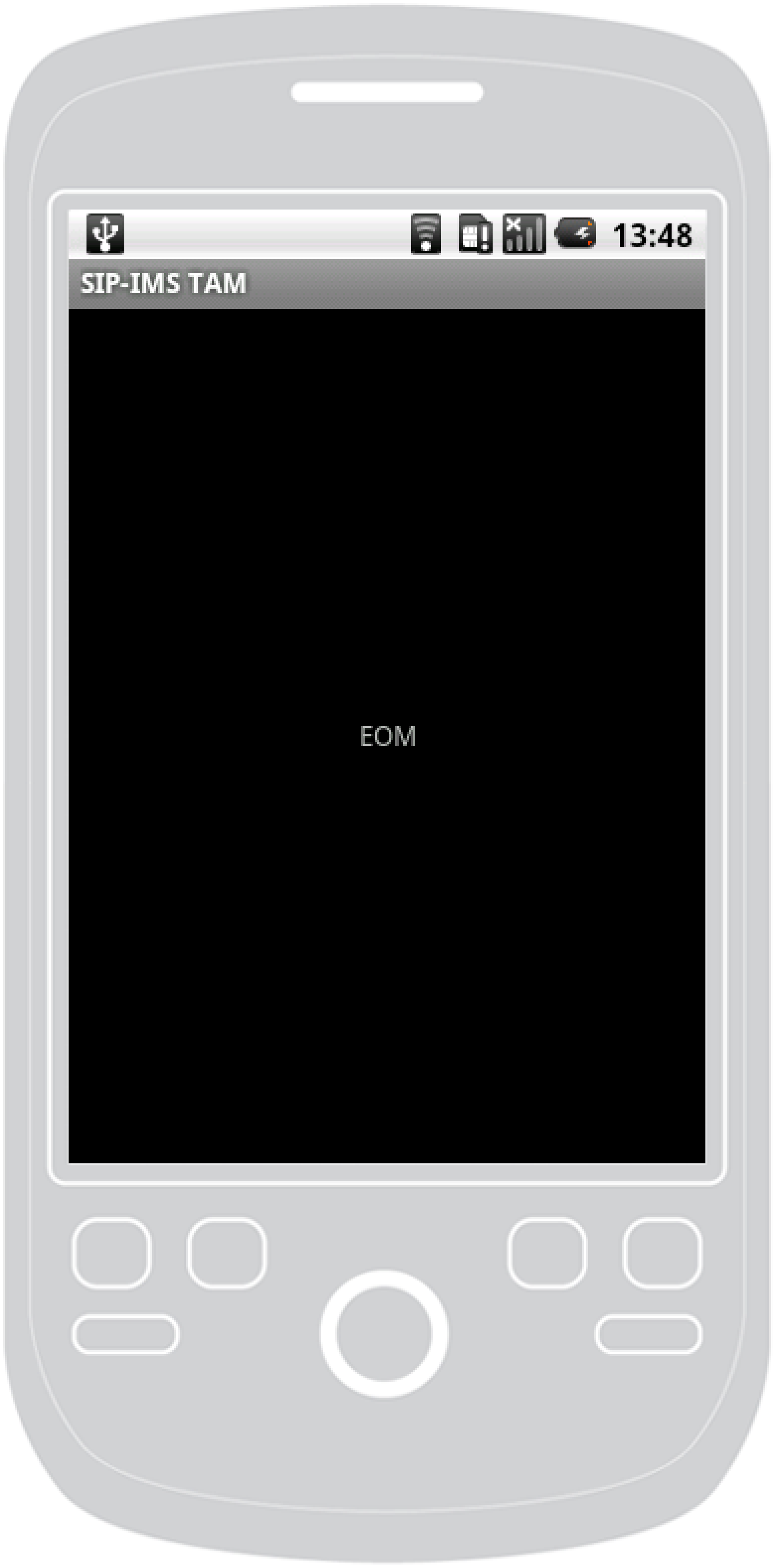

Figure 19 - Device screenshot after finishing the video streaming

3.3 Evaluation

Evaluating PQoS at the end device required a few recursive network measurements. In particular, for losses varying from 0% to 20% with a 2Mbps bandwidth and a 2.5% step, there was a recording of the sent, received and lost packets during the video playback which was, according to the video duration, 1 minute and 10 seconds. Afterwards, the packet loss ratio was calculated and the average value of it was extracted each time the video was streamed and then compared to the network losses ratio added to the network. Onwards, comparing PQoS with packet loss ratio was done with the same intervals and steps.

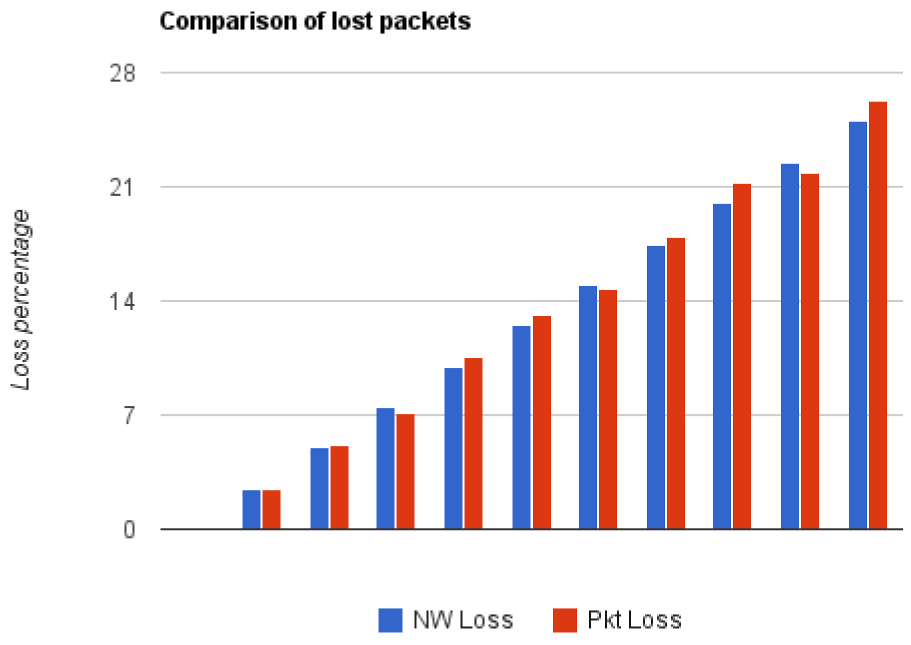

Figure 20 - Comparison of lost packets

NW Loss	Pkt Loss	Loss diff (%)
0	0	0
2.5	2.40	4.16
5.0	5.19	3.80
7.5	7.13	5.18
10.0	10.60	6.00
12.5	13.10	4.80
15.0	14.77	1.55
17.5	17.90	2.28
20.0	21.23	6.15
22.5	21.90	5.00
25.0	26.30	5.20

Figure 21 - Measurements results

So comparing real packet losses with the calculated losses the difference is really small varying from 1.55% to 6.15%. And this will result in more accurate comparison of the Perceived Quality of Service with the loss percentage as shown in Figure 22.

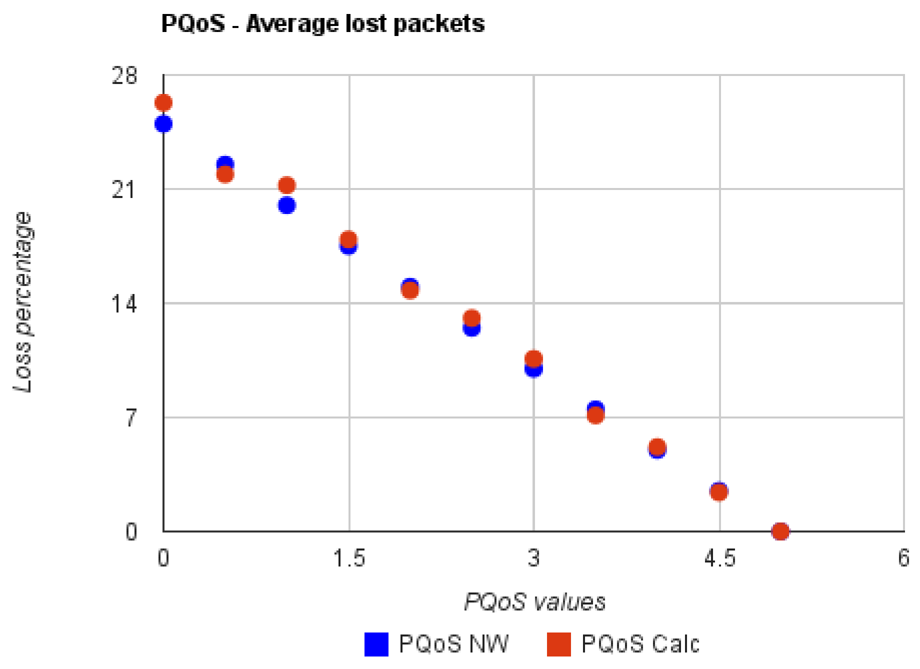

Figure 22 - PQoS v. average lost packets

Taking into consideration Figure 22 it is clearly shown that:

- NONE ALARM (PQoS > 3) => Loss = 10%
- WARNING ALARM (2 < PQoS < 3) => 10% < Loss < 15%
- RED ALARM (PQoS < 2) => Loss > 15%

4 Conclusion

This final year project described the design, implementation and usage of a software developed for monitoring the Quality of Service perceived by the user of the terminal. The software was developed using Android, an open platform for mobile development. Based on extracting the sequence number from the RTP header while streaming a video, the application was able to calculate the level of Perceived Quality of Service as a critical metric for the user satisfaction, after a repeated series of measurements. The testbed was able to communicate via SIP signalling through an IMS realm while being able to handle alarm notifications given the value of PQoS.

Future work could include implementing SIP signalling using Android's native access to Session Initiation Protocol (SIP) functionality, which is available since API level 9. An alternative version of this application could support VoIP functionality. And finally, someone could use a different algorithm for calculating and/or predicting the Perceived Quality of Service.

5 References

5.1 Acronyms

3GPP 3rd Generation Partnership Project

ACC Advanced Audio Coding

AF Assured Forwarding

AMR Adaptive Multi-Rate

API Application Programming Interface

AppQoS Application QoS

AS Application Servers

B2BUA Back to Back User Agent

BA Behaviour Aggregate

BE Best Effort

BNF Backus-Naur Form

BSD Berkeley Software Distribution

CDMA Code Division Multiple Access

CRLF Carriage-return Line-feed

CSCF Call Session Control Functions

DiffServ Differentiated Services

DSCP DiffServ Code-Point

DSL Digital Subscriber Line

DSS Darwin Streaming Server

EF Expedited Forwarding

GPRS General Packet Radio Service

GSM Global System for Mobile Communications

HOSS Home Subscriber Server

HSS Home Subscriber Server

HTTP Hypertext Transfer Protocol

ICSCF Interrogating Call Session Control Function

IMS IP Multimedia Subsystem

IP Internet Protocol

IPDV	IP Packet Delay Variation
IPER	IP Packet Error Ratio
IPLR	IP Packet Loss Ratio
IPTD	IP Packet Transfer Delay
IPTV	IP Television
IPSec	Internet Protocol Security
ISC	IMS Service Control
JAIN	Java APIs for Integrated Networks
JCP	Java Community Process
JSIP	JAIN SIP
JPEG	Joint Photographic Experts Group
LWS	Linear Whitespace
MSMM	Media Service Monitoring Module
MCMS	Multimedia Content Management System
MPEG	Moving Picture Experts Group
MSRF	Media Service Resource Function
MGW	Media Gateway
MS	Media Server
NAT	Network Address Translation
NDS	Network Domain Security
NQOS	Network QoS
OpenGL	Open Graphics Library
OSIMS	Open Source IMS
PCC	Policy and Charging Control
PCEF	Policy and Charging Enforcement Function
PCF	Policy and Charging Control Function
PCSCF	Proxy Call Session Control Function
PDA	Personal Digital Assistant
PHB	Per-Hop Behavior
PNG	Portable Network Graphics
POTS	Plain Old Telephone Service
PQoS	Perceived Quality of Service
QoS	Quality of Service
RTP	Real-time Transport Protocol

RTSP Real Time Streaming Protocol

RTT Round-trip Time

SCSCF Service Call Session Control Function

SD Secure Digital

SDK Software Development Kit

SIP Session Initiation Protocol

SIPSEE SIP Servlet Execution Environment

TAM Terminal Application Module

TC Traffic Control

TCP Transmission Control Protocol

THIG Topology Hiding Internetwork Gateway

TOS Type of Service

TU Transaction User

UAC User Agent Client

UAS User Agent Server

UI User Interface

UDP User Datagram Protocol

URI Uniform Resource Identifier

URL Uniform Resource Locator

VM Virtual Machine

VOIP Voice over IP

W-CDMA Wideband Code Division Multiple Access

WLAN Wireless Local Area Network

5.2 Bibliography

[1] Android Developers, developer.android.com

[2] Stack Overflow, stackoverflow.com

[3] Differentiated Service on Linux HOWTO, opalsoft.net/qos/DS.htm

[4] SIP: Session Initiation Protocol, tools.ietf.org/html/rfc3261

[5] Session Initiation Protocol, Wikipedia

[6] RTP: A Transport Protocol for Real-Time Applications, tools.ietf.org/html/rfc3550

[7] Henning Schulzrinne, cs.columbia.edu/~hgs/rtp/

[8] JAVA API for SIP Signalling, java.net/projects/jsip

[9] An Introduction to the JAIN SIP API, oracle.com/technetwork/articles/entarch/introduction-jain-sip-090386.html

[10] Jean Deruelle, jeanderuelle.blogspot.com/2008/10/jain-sip-is-working-on-top-of-android.html

[11] Open IMS Core, openimscore.org

[12] FOKUS Open IMS Playground, fokus.fraunhofer.de/en/fokus_testbeds/open_ims_playground

[13] IP Multimedia Subsystem, Wikipedia

[14] Georgios Gardikis, George Xilouris, Evangelos Pallis, Anastasios Kourtis, "Joint assessment of Network- and Perceived-QoS in video delivery networks", Springer Science+Business Media, 2010

[15] Xiaoming Zhou, Henk Uijterwaal, Rob E. Kooij, Piet Van Mieghem, "Estimation of Perceived Quality of Service for Applications on IPv6 Networks", PM2HW2N, 2006

[16] Lemonia Boula, Harilaos Koumaras, Anastasios Kourtis, "An Enhanced IMS Architecture Featuring Cross-Layer Monitoring and Adaptation Mechanisms", ICAS, 2009

[17] Harilaos Koumaras, Nikolaos Zotos, Lemonia Boula, Anastasios Kourtis, "A QoE-aware IMS Infrastruture for Multimedia Services", ICUMT, 2011

[18] ICT-ADAMANTIUM D2.1, "Overall System Architecture and Specifications"

[19] ICT-ADAMANTIUM D4.1, "Voice and Video Quality Perceptual Model"

[20] ICT-ADAMANTIUM D4.2, "Mapping PQoS to Transport and Access Traffic Classes"