MobiStream: Live Multimedia Streaming in Mobile Devices

Chrysa Papadaki
Department of Informatics
Athens University of Economics and Business
Athens, Greece
chrpapa@intracom.gr

Vana Kalogeraki
Department of Informatics
Athens University of Economics and Business
Athens, Greece
vana@aueb.gr

Abstract— **In recent years, many techniques have been proposed so as to enable resource-constrained devices to consume or deliver live multimedia streams. The majority of the existing techniques use distributed multimedia services and powerful servers to handle streams on behalf of clients. This is due to the fact that, multimedia streaming, when smartphones act as both clients and servers, can generate many challenges due to the heterogeneity of the multimedia streaming protocols, the media formats and codecs supported by today's smartphones. In addition, multimedia processing is resource consuming and, in many cases, unsuitable for a plethora of resource-constrained devices. To overcome these challenges, we present MobiStream a device-to-device multimedia streaming system for resource-constrained devices that achieves efficient handling of live multimedia streams. The design of MobiStream architecture provides solutions to several issues including resource constraints, streaming among heterogeneous operating systems and platforms, generation, synchronization and presentation of multimedia streams. We have developed the MobiStream prototype system on Java 2 SE and Android platforms; this paper presents the implementation details and the experimental evaluation of our system.**

Keywords-live multimedia streaming; Android platform; streaming protocol; resource-constrained devices.

I. INTRODUCTION

In recent years, the demand for real-time multimedia services, including voice over IP (Internet Protocol), audio and video streaming, has been growing rapidly so that multimedia streaming applications have become dominant in present communications systems. Furthermore, the explosive development of mobile networks and the availability of mobile devices in the hands of the masses, have made real-time multimedia delivery popular on mobile devices, such as smartphones and tablets, which have now become a major part of everyday life. It is an indisputable fact that cellular traffic is growing tremendously, with a share of video traffic increasing from 50% now to an expected 66% by 2015 [2]. Consequently, the demand for innovative smartphone applications that allow users to receive and deliver live or on-demand rich content has increased dramatically.

Today's smartphones are equipped with significant processing, storage and sensing capabilities, as well as wireless connectivity through cellular, Wi-Fi and Bluetooth. They provide ubiquitous Internet access, primarily through their cellular connection and secondarily through Wi-Fi, and enable a plethora of distributed multimedia applications. However, the acquisition and transmission of large amounts of video data even on modern mobile devices create important challenges. Issues like resource allocation, energy consumption, CPU, memory and bandwidth constraints, as well as the software development platform must all be taken into consideration. It is, therefore, essential to address these challenges by efficiently managing the resources and employing effective streaming techniques.

Current solutions for mobile multimedia streaming assume a centralized architecture where a resource-powerful server can support heterogeneous sets of media encoders, decoders and streaming protocols and is able to adapt content on behalf of clients to provide multimedia streams to resource-constrained mobile devices [6][12]. On the other hand, solutions for multimedia streaming over ad hoc networks assume the existence of distributed multimedia services and require cooperation between mobile devices for content dissemination; however, these either do not consider the scenario of content adaptation [7] or are cross-layered [8]. Din and Bulterman [11] demonstrate the use of synchronization techniques for distributed multimedia, but without addressing the issue of energy reduction. Recently, lightweight middleware targeting mobile multimedia applications have been proposed to address the issues of heterogeneity on modern smartphones. One of the latest efforts is the Ambistream middleware [9], which provides an additional layer as an intermediate protocol and the associated container format for multimedia streaming among heterogeneous nodes. For the generation and presentation of the multimedia streams, PacketVideo OpenCore [13] and Stagefright [14] multimedia frameworks are used, respectively. Moreover, these multimedia frameworks are based on cross-platform solutions. One of them is FFmpeg (Fast Forward MPEG) [15], which is an Open Source lightweight multimedia framework that allows encoding, multiplexing and streaming of videos in different formats. However, FFmpeg has several limitations; it does not support a wide range of audio/video codecs, especially for Android devices and is better suited for streaming from a single source.

Multimedia streaming is a challenging problem when smartphones act as both clients and servers. This is due to

 ISBN: 978-1-61208-265-3

the fact that, the framework needs to be integrated into multiple mobile platforms to provide live streaming among multiple smartphones because of the variability of the supported media formats, codecs and streaming protocols. In addition, multimedia processing, especially in the case of handling streams of high-quality content, is resource-consuming and needs to be carefully handled in the case of mobile devices. To address the above challenges, in this paper, we present MobiStream, a mobile-to-mobile live multimedia streaming system that enables mobile devices to easily handle live multimedia streams leveraging the available multimedia software stack of the applied platform. We assume the scenario of a mobile device that requests to deliver a live multimedia stream to one or more peers. In fact, MobiStream enables mobile devices to act as both clients and servers and allows clients to process and deliver live multimedia streams to mobile devices or desktop servers, while considering resource constraints. An important feature of MobiStream is that it can also materialize the scenario of live multimedia streaming over an ad hoc network. For example, the Android Ice Cream Sandwich devices provide peer-to-peer (P2P) connectivity using WiFi Direct [10], so, either a laptop or an Android device can easily act as a virtual access point (AP). Thus, the system using nodes that act simultaneously as servers and clients can support this kind of scenarios. The streaming client in our approach does not act as a relay client for other phones. Taking all the above into account, we envision a system that provides sustainable solutions to a wide range of applications, such as streaming a live event directly to other devices reachable on the network, voice and video call applications, private audio-visual communication between peers without involving third party servers, sharing live multimedia content in cases of unavailable infrastructure, etc. We have implemented our prototype system that is running on both Android and Java 2 SE platforms to demonstrate the feasibility of our approach.

The rest of the paper is structured as follows. In Section II, we describe the system design in detail and discuss several design issues concerning the generation, transmission, synchronization and presentation of the live multimedia streams and the choices we made to address them. Section III demonstrates our approach on the synchronization of the streams. In Section IV, we present the prototype system we have implemented and discuss implementation details, including challenges specific to Android phones. In Section V, we present the system performance evaluation results of our testbed for a range of scenarios and conclude the paper in Section VI.

II. SYSTEM DESIGN

A. System Overview

MobiStream is structured in a client-server model, where devices are able to act as servers and clients simultaneously. These can communicate over cellular or WiFi. Each device can assume both roles, as it can be a client, when uploads content to a server, or a server, when it receives one or multiple media streams from the clients. The Client consists of the Dispatcher component, the Synchronization Module and the Media Recorders. The Dispatcher is responsible for communicating with the Server and packaging and transmitting the generated Media Units (MUs). The MUs are produced by the Audio and Video Recorders which are independent sub-applications of the Client. The Synchronization Module is responsible for synchronizing the generated media units before the final stage of transmission. The Server is designed to run on mobile devices as well as desktop computers. It comprises the Receiver component, the Sync Manager and the Media Players. The Receiver component is used to listen for incoming client requests, using a built-in TCP Server which is running independently in the background, and depackages and separates the received MU packets. The Sync Manager is responsible for the synchronization of the received MUs, while the Audio and Video Players are in charge of the presentation of the final synchronized multimedia stream. Both clients and servers are multithreaded so as to enable the server to receive multimedia streams from many clients and the client to transmit to multiple destinations. Fig. 1 illustrates the overall system architecture.

In the remaining of this section, we give an overview of the building blocks and the interaction between them.

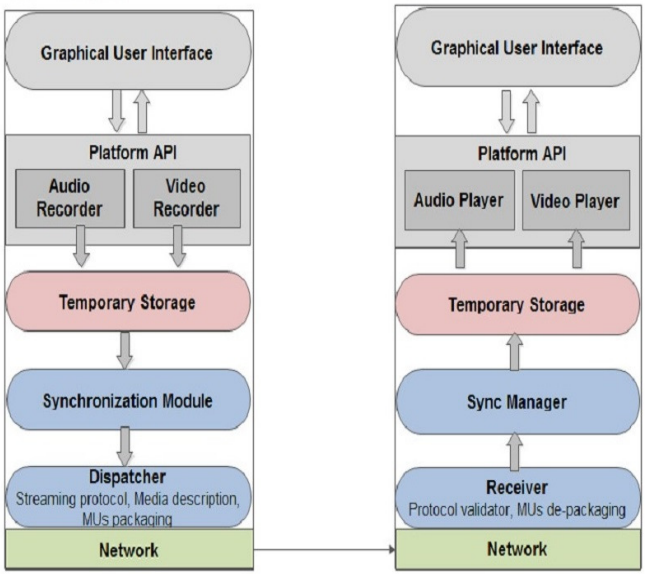

Figure 1. MobiStream Framework Architecture. Streaming Client (left) – Server (right)

B. Streaming Client

The Streaming Client is in charge of generating multimedia streams and transmitting them to the Server. More specifically, it comprises the following main components:

Media Recorders: one of the first design challenges we faced was the design of media components for Android devices that would enable the generation of live video streams. Currently, the available APIs (Application Programming Interface) of the latest Android SDK do not include specifications to allow developers to capture fragments of live video streams. To circumvent these problems, we designed and developed our media components, which are able to produce and consume media units of specific formats. Thus, for the audio recording, we designed the Audio Recorder, a component that records uncompressed PCM (Pulse-code modulation) frames of fixed size from the input hardware device and stores them in a concurrent data structure used in parallel with the Synchronization module (discussed below). For the video recording, we designed the Video Recorder, a component that obtains an instance of the hardware input camera, sets camera parameters, frame rate and preview resolution, starts updating the preview surface and simultaneously capture the preview frames and stores them. The module that captures preview surface frames actually captures a sampling of the video, consequently a lower-quality video than the expected is being produced and second, during the video recording, the FPS (frames per second) vary, and that would significantly affect the smoothness of the video play out.

Synchronization module: we designed the Synchronization Module in order to eliminate the variability of video capture rate and synchronize the audio and video streams. The Synchronization Module is responsible for monitoring video and audio in order to capture the rate, based on the formulas we discuss in the next section, and propagate the frames and the samples to the packaging stage at the Dispatcher application.

Dispatcher: the main responsibility for the Dispatcher is to establish a connection, setup a multimedia session and packetize the media units, that polls from the local buffers. The co-operation of Dispatcher and Synchronization module results in the transmission of the synchronized multimedia streams. The overall technique for the synchronization at the client side is described in details in Section III.

C. Server

A significant feature of our proposed Server design is that it is modular and platform-independent. The Server is multi-threaded in order to be able to present more than one multimedia streams from different sources. This component is responsible for handling client requests, configuring the requested multimedia sessions, receiving and reconstructing the multimedia streams, and displaying feedback during the experiments.

Receiver: the Receiver is in charge of handling incoming connection requests, de-packetizing the incoming RTP packets using a packet Validator module, and separating the streams by drawing information from the header. Then, the receiver provides Sync Manager with the received MUs in order to proceed to synchronization stage.

Sync Manager: the Sync Manager is one of the most significant components of our system as it is used to address several major problems related to synchronization of the media units and the presentation of the final stream. It consists of a multimodal functionality as described below. In case of an unreliable link for the uploading of the multimedia streams, the Receiver enables the entire functionality of the Sync Manager in order to execute the audio/video synchronization algorithm we discuss in Section III, so as to prevent the out-of-order presentation of the MUs and the lack of synchronization between audio and video. Given the video frame rate, the Sync Manager is able to compute the video and audio playback time in order to achieve the same temporal correlation of MUs as at the transmission point and synchronize them in order to be played by the Media Players without letting network delays affect the video presentation. In the case of a reliable connection, the Sync Manager assumes that the packets arrive in order, as the underlying protocol is TCP, so, it decides not to use the synchronization algorithm and only adopts a buffering technique in order to synchronize the media streams and provides them to Media Players in a constant rate which represents the playback rate of the multimedia stream at the origin. The proposed buffering technique is presented in the next section in detail.

Media Players: we designed these components in order to enable the presentation of multimedia streams of PCM and JPEG units at the receiver end. For both players we followed a producer-consumer design, using concurrent data structures. The Sync Manager is the producer that produces the MUs in order and the players are the consumers that consume the available media units. For video presentation, we created a user interface handler that updates the video screen when a new video frame is available. For audio play out, we designed an Audio Player that is able to play audio samples in a specific frame format and sampling frequency (discussed in details in the next section).

III. PROPOSED APPROACH

The system follows a client-server model of two independent audio and video decoders. Using multi-

threaded software, we managed to accelerate the process of video reconstruction by separating the multimedia streams, synchronizing them whenever required, at negligible CPU overhead, as we show in our experimental evaluation, and executing parallel decoding of each stream. This way, an application based on this system is able to run efficiently on resource-constrained devices minimizing the processing overhead and reduce processing delays, which are critical for real-time multimedia applications. Apart from software architecture and computer performance, another significant contributory factor to live multimedia streaming is the network availability. The Internet, like other packet networks, occasionally looses and reorders packets and delays them by variable amounts of time. To overcome these impairments, we designed a protocol for real-time communication following the Real-Time Transport Protocol (RTP) specifications [1] that provides end-to-end delivery services for data with real-time characteristics, such as interactive audio and video.

A. *Proposed Real Time Protocol*

One important feature of our real-time protocol was to provide a way to reconstruct audio and video streams with a controlled delay for play out. To achieve this goal, we use the RTP header to packetize MUs in order to provide the receiver with payload identification, timing information and a sequence number, the last two allow receivers to calculate packet losses and jitter as well. Although the proposed protocol follows the general design of RTP, it does vary in several major ways.

First, RTP does not provide any mechanism to ensure timely delivery or provide other Quality-of-Service guarantees i.e. prevention from out-of-order delivery. It actually uses underlying protocols, usually UDP, for transport and multiplexing functionality. In an audio/video session [3] as opposed to [5] where an algorithm is proposed for synchronizing of streams carried in separated sessions. This type of streaming is acceptable over low bandwidth communication channels. Thus, to begin live streaming, the establishment of one end-to-end connection over either TCP or UDP is required. In addition, each device is able to start multiple sessions to transmit video to different destinations. To achieve multimedia streaming in one session, we had to keep the payload type constant and allocate different values to the synchronization source identifier (SSRC) field regarding the media type of the payload. In comparison to RTP specifications where if both audio and video media are used in a conference, they are transmitted as separate RTP sessions, therefore SSRC identifier is a randomly chosen value meant to be globally unique within a particular RTP session. In Table I, we describe the attributes of the header we use to packetize the media units. Our goal in the streaming protocol is to support live multimedia services either over TCP or UDP.

TABLE I. PACKAGING ATTRIBUTES

Name	***Size***	***Description***
payload type	1 byte	This field identifies the format of the RTP payload and determines its interpretation by the application. It holds the same value for all packets regardless of the media payload type, because all packets represent one multimedia stream.
sequence number	2 bytes	The sequence number increments by one for each data packet sent, and may be used by the receiver to detect packet loss and to restore packet sequence
time stamp	4 bytes	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 allows synchronization and jitter calculations
SSRC	4 bytes	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
Payload	N bytes	Data

We implement a buffering technique that we discuss in the next section, consisting of two major parts. The first part refers to a dispatcher-side buffering in order to facilitate the synchronization of the generated MUs and the second part concerns the adoption of a receiver-side buffer to accommodate initial throughput variability and inter-packet jitter. The experimental results we conducted shown that the proposed buffering technique can be integrated into applications using TCP-Friendly transmission of multimedia streams, and benefit from TCP mechanisms as it is reliable and guarantees delivery of packets in order. However, using TCP as transport layer may induce long delays because of the TCP retransmission mechanism. This actually leads to long video pauses at the receiver-end, which highly degrade the real-time communication. To cope with this issue, we monitor the transmission delay between successive incoming packets and drop those that are late with respect to specific thresholds, we discuss later, related to the actual time user conceives. As far as the scenario of using a UDP-based streaming protocol is concerned, we adopt the proposed streaming protocol over UDP using the buffering technique, we discuss in the next session, and the time-oriented audio and video synchronization algorithm that we present in Section C.

B. *Buffering Technique*

One of our major design challenges was how to create a synchronized multimedia stream with a constant playback rate produced by two different media sources, as the capturing and coding rate on each source is different and induces variable delays. To address this problem, we first synchronized the camera and microphone capture rates by setting up our system's audio recorder appropriately so as to

 ISBN: 978-1-61208-265-3

capture audio samples depending on the capture frame rate of smartphone's camera. Moreover, we provide a client-side buffering so as to adjust multimedia stream capture rate by prefetching multimedia data into a buffer in a controlled rate, which represents the playback rate at the receiver. This assures the elimination of the variable delays induced by sources. Thus, media streams have well-defined temporal relations among themselves and can be sent synchronized to the server. More precisely, the relation among the audio samples, video frames and playback time is given by the following formulas:

$$VP_i = V_i / VR \quad (1)$$

where VP_i is the video playback time of the i^{th} video frame in seconds, V_i is the i^{th} video frame number which is an integer that increases by one representing the i^{th} generated video frame and VR represents the video frame rate (Frames per second) of the source. In practice, applying the (1), the system is able to accurately calculate the playback time of a particular video frame in seconds. To calculate the audio playback time, APj, of an audio frame, we use (2), where the num_samples represent the number of the encoded audio samples of 16-bit each of the produced PCM frame. In our approach, in stereo mode, a PCM audio frame contains 512 samples and, in mono mode, a frame contains 1024 mono samples, thus, it follows that each audio frame consists of 2048 bytes minimum. This size applies to all fragments of the audio stream. Note that using Android Media package, data should be read from the audio hardware in chunks of sizes subject to the total recording buffer size. In (2), A_j is the j^{th} audio frame number which is an integer that increases by one representing the j^{th} generated audio frame and sampling frequency corresponds to the produced samples per second (Hz).

$$AP_j = \text{num\_samples} \times (1 / \text{sampling frequency}) \times A_j \quad (2)$$

Taking the above-mentioned into account, we conclude to (3), which calculates the audio frame that must be presented in the VP_i^{th} second in order to achieve synchronization.

$$A_j = VP_i \times \text{sampling frequency} / \text{num\_samples} \quad (3)$$

Using the above formulas, the Synchronization module of the Client application is able to estimate the correlation among the produced MUs and provide the Dispatcher with a synchronized multimedia stream so as to transmit the MUs in the right order so as to be presented in sync at the Receiver, in case of transmission under ideal circumstances, no further processing would be required at the Server in order to present a synchronized multimedia stream. Nevertheless, a critical aspect lies in the lack of synchronization that may exist between audio and video streams at the receiver-end due to the fact that the characteristics of IP-based network, delay and jitter, affect the temporal relations present in multimedia streams. To circumvent these problems, we use a receiver buffer for the temporary storage of incoming media units comparing (1) to (2). In practice, the Sync Manager of the Server checks whether the playback time of a newer video frame is the same with the playback time of the corresponding audio frame. If this is the case, it follows the presentation of MUs at the proper time. The use of a MU buffer introduces some delay in the application, which is directly proportional to the size of this buffer. The objective of the process is to provide a presentation that resembles as much as possible the temporal relations that were created during the encoding and multiplexing process at the Client.

C. Audio/Video Synchronization Algorithm

In our system, the real-time delivery of the packets can be accomplished by using either TCP or UDP as the transport layer. Taking for granted that the media streams are synchronized at the origin, we need to achieve the same temporal correlation for playback at the receiver. This can be a quite difficult issue when the system performs transmission over UDP, which is unreliable and does not provide Quality-of-Service mechanisms, such as prevention from out-of-order delivery of packets. To cope with this challenge, we propose the following synchronization algorithm which imposes negligible CPU overhead, as shown in experimental results below, which is important as we have to deal with resource-constrained devices and real-time communication. In order to ensure a better quality of the reconstructed material, priority is given to audio information. We chose audio stream to be played regardless of the state of the video because human perception is more sensitive to degradation in audio quality than in video [4]. This means that audio would be played upon arrival as long as it is in order, regardless of the state of the video stream. In practice, if the audio stream anticipates the video stream, the receiver simply discards the video packets.

In the case of receiving a video packet, first, the audio/video synchronization algorithm checks the SSRC field of the packet header in order to determine whether the payload contains audio or video data. Then, it checks if the received video frame is newer than the displayed one by comparing the new timestamp with the old one. If this is the case, it calculates the video and audio playback times, using (1) and (2), respectively. If the audio is ahead of the video, the algorithm calculates the difference between their playback times, $AP_i - VP_i$. In the case of $AP_j - VP_i >$ threshold, where threshold is the maximum level at which humans detect frames as being in sync, the video is considered too old to be displayed and it is dropped, otherwise it is rendered. The threshold is tuned based on the application characteristics. In [4], a detailed study of the end user capability to detect harmful impacts of de-synchronization on QoE (Quality of Experience) is provided. The author indicates that an absolute skew smaller than 160 ms is harmless and greater than 320ms is harmful for QoE. The author identifies a double temporal area [-160,-80] and [80,160] called transient, in which the impact

 ISBN: 978-1-61208-265-3

of the skew heavily depends on the experimental conditions.

IV. IMPLEMENTATION

Our software architecture was motivated by the need to have a simple and platform-independent implementation. We chose Java as the development language. The object oriented features of Java and its simplicity enables our system to be simple and modular. Thus, MobiStream can run on any platform that supports Java and requires a real-time streaming protocol for multimedia services. The software for the smartphones is an Android application that enables the device to act simultaneously as client and server and runs efficiently on Android v2.3 or later versions. For the laptop server, we used in some experiments, the software runs on Java 2 SE. We have also developed a graphical user interface (GUI) and the code for the media components.

In this section, we describe the implementation details, the major challenges we faced specifically on Android phones, and the design choices we made to address them.

A. The Streaming Process

The phases required to complete the streaming process between two devices are media capture, media transmission and media presentation. In this section, we describe the implementation details of each phase and the technical problems we encountered.

1) Media Capture

Media content originates from hardware input devices, that is, camera and microphone. In most multimedia applications, the media capture phase is implemented using available APIs that provide access to built-in Multimedia Recorders that supports several media formats, encoders and streaming protocols in order to provide playable stream formats to Media Players. Developing on Android platform, we faced two major issues. First, the lack of hardware accelerated codec APIs when we implemented the prototype system and, secondly the fact that the exposed APIs do not provide the ability to stream live multimedia content from the built-in Media Recorder in a format playable from the built-in Media Player. To overcome these issues, we have implemented two independent Media Recorders. Each one is able to draw input from a different hardware device and use media formats and encoders supported by all platforms.

For the video recording, we used the Camera APIs to set image capture settings, start/stop preview and retrieve frames for encoding for video. An instance of the camera is actually a client for the Camera service, which manages the actual camera hardware. We install a callback to be invoked for every preview frame, using pre-allocated buffers, in addition to displaying them on the screen. The callback will be repeatedly called for as long as preview is active and buffers are available. The purpose of this method is to improve preview efficiency and frame rate by allowing preview frame memory reuse. The image format for preview pictures is either NV21 or YV12, since they are supported by all camera devices. To reduce the size of the video images, we use a JPEG encoder. The video frame size depends on the video resolution and the quality of the compressed data.

For the audio recording, we used the AudioRecord class of the Android SDK which manages the audio resources for Java applications to record audio from the audio input hardware of the platform. This is achieved by reading the data from the AudioRecord object. Upon creation, an AudioRecord object initializes its associated audio buffer that it will fill with the new audio data. The size of this buffer, specified during the construction, determines how long an AudioRecord can record before "over-running" data that has not been read yet. Data should be read from the audio hardware in chunks of sizes inferior to the total recording buffer size. Thus, the Audio Recorder captures uncompressed PCM samples of a specific sampling rate and size. In our prototype system, we set the sampling rate and the size of the recorded samples accordingly to the video frame rate in order to facilitate the synchronization process, as described previously. The captured MUs are stored in concurrent data structures so as to enable the co-operation of the modules involved in capture and transmission phases.

2) Media Transmission

At the end of the capture phase, since the MUs cannot be directly transmitted over IP-based networks, they are wrapped within media containers that provide the necessary meta-information to facilitate the decoding and correct presentation at the receiver end. This task is assigned to the module that packages the media units following the specifications of the real-time streaming protocol we discussed previously. At the server side, the receiver performs the de-multiplexing and de-packaging process and provides the separated media streams to the Sync Manager in order to synchronize them before the presentation phase. A contributory factor to the efficiency of the collaboration among the modules of the different phases is the use of Android Services, which are independent application components that host the main processes of our system and execute long-running operations while not interacting with the user.

3) Media Presentation

Using the above-mentioned Media Recorders, the proposed real-time streaming protocol and the synchronization algorithms we discussed previously, the system is able to reproduce the initial media streams and proceed to the presentation phase. In order to present the MUs, we developed two independent Media Players. For the video playback, first the decoding of the compressed data from the playback buffer takes place and then the User Interface Handler which extends the Handler class of Android SDK updates the video view. This process is executed as soon as there is a new video frame in the playback buffer. For the audio playback, we developed an Audio Player, using the AudioTrack class of the Android SDK which manages and plays a single audio resource for

 ISBN: 978-1-61208-265-3

Java applications. It actually allows streaming PCM audio buffers to the audio hardware for playback.

B. Streaming Protocol

We used the java.net library to implement a library that provides a streaming protocol for real-time applications, based on Real-time Transport Protocol, for multimedia services and can be ported to any platform supports Java and its network libraries. Using this library, the system is able to set up, start and handle multiple unicast sessions using UDP or TCP as the transport layer, and transmit multimedia data supporting a wide range of media formats for the packaging and de-packaging stages, even though in the prototype system we used specific formats in order to facilitate the porting of the live multimedia streaming process to different platforms.

V. EXPERIMENTAL RESULTS

A. Experimental Setup

We have conducted a set of experiments in order to evaluate the efficiency and performance of MobiStream. The testbed of the experiments is presented in Table II. Additionally, we provide screenshots of the android application in Fig. 2. This setup can be used in various scenarios, for example, in streaming video, in mobile video, e-health, assistive technologies. First, we assume a Streaming Client running on an Android-powered device that uploads live multimedia streams to a Server. We conducted a series of experiments using different levels of signal strength - weak, medium and strong using TCP, monitoring the five bar scale of the smartphones which basically measures radio signal levels maintained by the wireless network adapter, in decibels (dB) on a more linear scale. For each experiment, we report the averaged results of five runs. We also repeated the process using UDP and compared the results. To run this test we used the Xperia Neo V Android smartphone as Streaming Client and the Samsung Windows 7 laptop as Server. The second scenario concerns a Streaming Client delivering a live multimedia stream to multiple receivers of the network. The network comprises a wireless access point (*i.e.,* router), a streaming client (Xperia Neo V) and 5 to 20 receivers. We executed the experiment using a different number of receivers so as to record the end-to-end delay and the jitter, in order to investigate how these measurements affect the quality of the video at the receiver. In all cases, no external peers injected traffic in the network the server allows a few seconds (3s to 5s regarding the signal strength) startup delay, which is a common practice in commercial streaming products. All packets arriving earlier than their playback times are stored in the server's local buffer. In comparison to Ambistream in which a 30s start-up delay is introduced by the middleware layer to allow protocol translation. This aspect restricts the use of the middleware for real-time applications. The multimedia stream has a QCIF (176 by 144) frame size in 200kbs and 400kbs video bitrates, whereas in 600kbps, 800kbps and 1000kbps we apply a CIF (352 by 288). The stream duration is 180 seconds and the video capture rate varies accordingly to the video bitrate presented in the experimental results; in total, more than 12 hours of streaming required among the testing devices.

TABLE II. TEST DEVICES

Test Devices	*Sony Ericsson Xperia Neo V*	*HTC Explorer*	*Samsung NP300V5A-S05*
Role	Client/Server	Client/Server	Server
Platform	Android 4.0.4	Android 2.3.5	Windows 7
CPU	1 GHz	600MHz	I5-2450M 2.5GHz
Memory	420MB	256MB	4GB

D. Experimental Results

1) System Evaluation

We first present the experimental results of the mobile-to-server scenario. We focus on the following Quality of service metrics: end-to-end delay (i.e., the time taken for a packet to be transmitted from the client to the server), the jitter (i.e., packet delay variation measured at the server) and the download rate (i.e., the transmission bitrate measured at the server). In Fig. 3 and Fig. 4, we present the download rate of the desktop server using TCP and UDP respectively. We chose a high video bitrate of approximately 1100kbps, in order to evaluate network throughput. In case of using TCP, Fig. 3 clearly depicts the behavior of the transport protocol in the weak signal strength case, as it shows intense variability of the download rate induced by the retransmission mechanism of TCP. In the medium and weak signal strength cases, the download rates recorded were 4,96% and 17,97% lower than the rates observed in strong signal strength case. In case of using UDP, we observe from Fig. 4 that the download rates in medium and weak signal

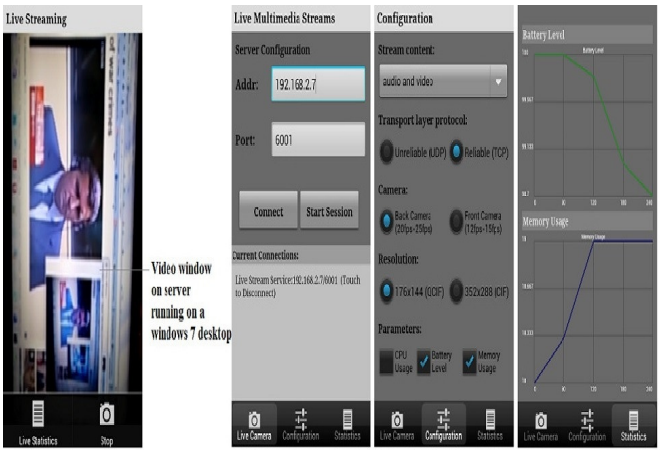

Figure 2. (a) The mobile screen while recording a live event and the video window of the server running on a desktop, (b) Server Configuration screen, (c) Session Configuration screen, (d) Statistics screen.

download rate is higher regardless of the signal state due to the client-side buffering employed in the framework. Fig. 5illustrates the jitter for different packets using TCP. In weak signal strength we recorded high values of jitter, e.g., 786ms, at 387th packet. This fact entails long pauses at the video presentation. Nevertheless, our proposed approach discussed in Section III accomplishes a good quality of the video stream without degrading the real-time communication. In medium signal strength, the highest absolute values of jitter are smaller than the values recorded in weak signal state. In strong signal strength, the highest positive value of jitter recorded was 40ms. Regarding the second scenario of the use of multiple server applications running on the network, we measured the end-to-end delay in case of 5, 10, 15, 20 receivers using TCP. Fig. 6 presents the mean end-to-end delay for different numbers of servers running in the network. The end to end delay remains within acceptable bounds in terms of video quality and Quality-of-Experience and increases proportionally to the number of receivers, approximately 28% from 5 to 10 receivers, 30% from 10 to 15 receivers and 48% from 15 to 20 receivers.

2) Evaluation of Memory and CPU usage

We also measured the resource usage of our approach. We run the experiments using the HTC Explorer smartphone described in Table II. Fig. 8 illustrates that the memory usage at the Server side remains constant. For higher data-rates, the memory usage may increase slightly because of the higher buffer sizes required. In the case of the Client application, the memory usage increases proportionally to video capture rate (including only JPEG data). In both applications, the framework re-uses the pre-allocated space in RAM in order for the multimedia application to be able to run under memory constraints, as in this scenario we run the experiments using a smartphone with 256MB RAM. Fig. 9 depicts the CPU overhead on both client and server mobile applications versus the video bitrate. In all experiments we observed slightly higher percentage of CPU overhead in Client application, this is due to the use of the hardware input camera and the YUV compression module. Nevertheless, in both applications when the video bitrate is greater than 700kbps the CPU overhead tends to be the same. In order to accurately estimate the CPU usage of the framework during the live streaming process, we divided the CPU monitoring into three phases; (I) initialization of media components, (II) streaming process, (III) media components finalization. In both client and server applications the CPU usage during the first phase were 67% and 55%, respectively. The second phase is illustrated by Fig. 9 and includes, from the client point of view, the recording, storing, packaging and transmission of the media units. Regarding the server application, the second phase includes the de-packaging, the synchronization, the storing and the presentation of the received media units. For the third phase, the server and client required approximately 55% and 68% CPU usage, respectively.

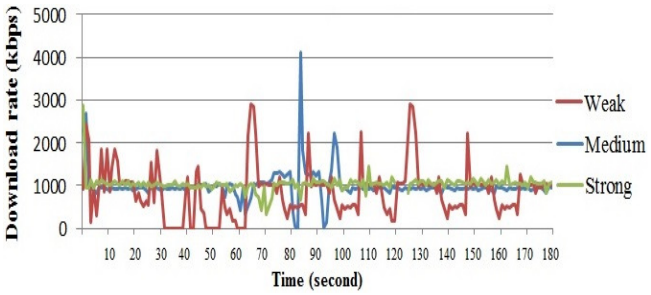

Figure 3. Download rate (kbps) - Signal Strength, using TCP.

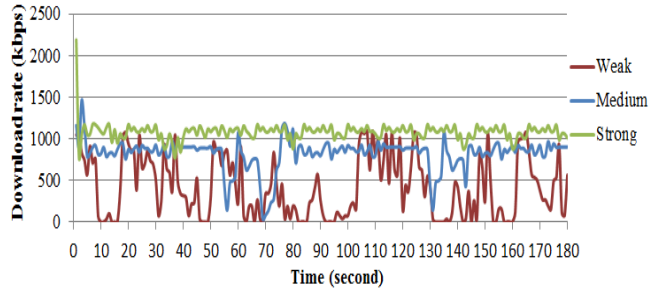

Figure 4. Download rate (kbps) - Signal Strength, using UDP.

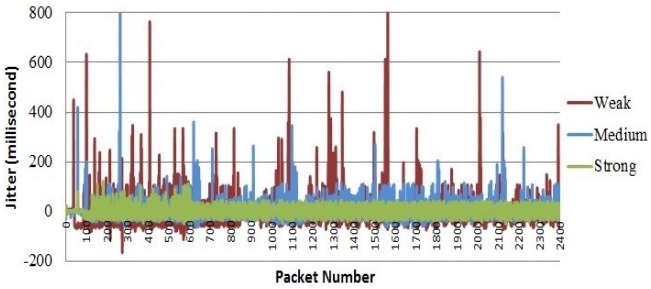

Figure 5. Jitter(ms) - Signal Strength, using TCP.

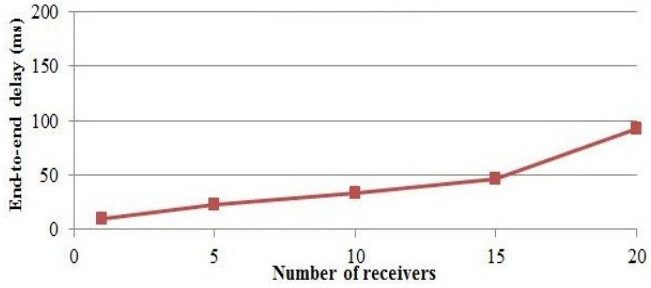

Figure 6. End to end delay (ms) – Number of Receivers, using UDP.

3) Evaluation of Energy Consumption

In the last set of experiments, we measured the energy consumption of our approach. We executed the scenario of mobile-to-mobile server running on smartphones and before the experiment both smartphones were fully charged. During the experiment, the battery states are recorded every 10 seconds. Fig. 7 presents the battery state as a function of time. The 100% percent corresponds to the fully charged battery. We chose a high video bitrate of 1100kbps and run

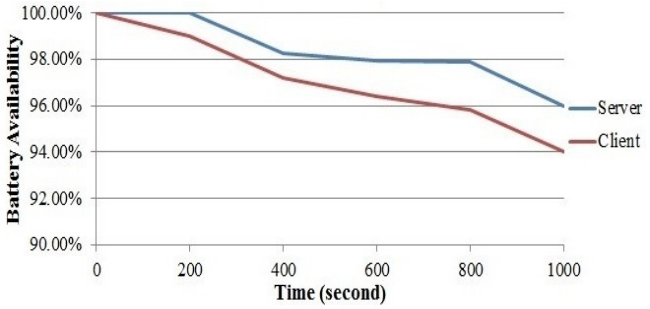

Figure 7. Battery Level (%) – Video bitrate (kbps)

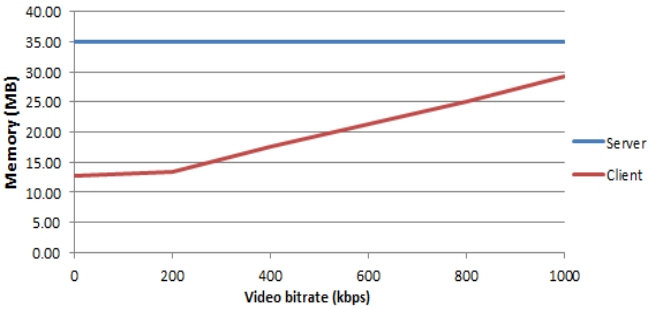

Figure 8. Memory (MB) – Video bitrate (kbps)

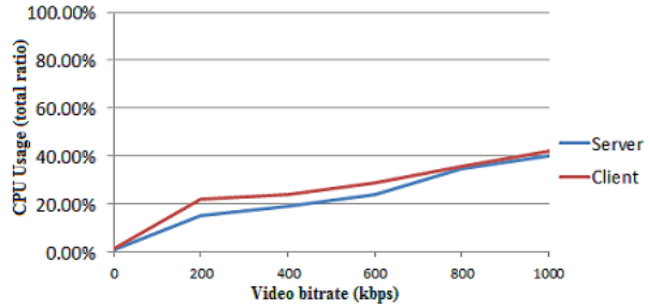

Figure 9. CPU Usage (total ratio) – Video bitrate (kbps)

each experiment for 16.6 minutes. Fig. 7 depicts that the Server hardware input Camera and framework's Audio Recorder compared to the Server application in which the main energy consuming component is the Audio Player.

II. Conclusion And Future Work

In this paper, we designed, implemented, and evaluated a mobile multimedia system, MobiStream that enables resource-constrained devices to handle real-time multimedia streams. We designed a platform-independent framework so that we can support live multimedia streaming among heterogeneous mobile devices. We present our approach on the synchronization of the media streams and the streaming process we employed. Our experimental results demonstrate significant performance benefits in terms of the usage of the mobile devices' resources and video quality. For our future work, we plan to evaluate the working of our approach using a larger number of heterogeneous mobile devices.

Acknowledgment

This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) – Research Funding Program: Thalis-DISFER, Investing in knowledge society through the European Social Fund.

References

[1] H. Schulzrinne, S.L. Casner, R. Frederick, and V. Jacobson. "/RTP: A Transport Protocol for Real-Time Applications", IETF Request for Comments: RFC 3550, Jul. 2003.

[2] Cisco Systems, "Cisco visual networking index: Global mobile data traffic forecast update", 2011-2016. http://www.cisco.com.

[3] M. Westerlund and C. Perkins "Multiple RTP Sessions over a single Transport flow", Ericsson, University of Glasgow, Nov. 2011.

[4] R. Steinmetz, "Human perception of jitter and media Synchronization", IEEE Journal on Selected Areas in Communications, vol. 14, no. 1, 1996, pp. 61–72.

[5] R. Bertoglio, R. Leonardi, and P. Migliorati, "Intermedia Synchronization for videoconference over IP", Signal Processing: Image Communication, Sept. 1999, pp. 149-164.

[6] K. Curran and G. Parr, "A Middleware Architecture for Streaming Media over IP Networks to Mobile Devices", IEEE Int. Conf. Wireless Communications and Networking, Mar. 2003.

[7] N.M. Do, C.H. Hsu, J.P. Singh, and N. Venkatasubramanian, "Massive live video distribution using hybrid cellular and ad hoc networks", IEEE International Symposium on World of Wireless, Mobile and Multimedia Networks, Jun. 2011.

[8] J. Domingo-Pascual, P. Manzoni, S. Palazzo, A. Pont, and C. Scoglio "On the Forwarding Capability of Mobile Handhelds for Video Streaming over MANETs", 10th International IFIP TC 6 Networking Conference, May 2011.

[9] E. Andriescu, R. Cardoso, and V. Issarny, "Ambistream: A Middleware for Multimedia Streaming on Heterogeneous Mobile Devices", Middleware, volume 7049 of Lecture Notes in Computer Science, pp. 249-268, 2011.

[10] C. Fragouli and E. Soljanin. "Network Coding Fundamentals." Now Publishers Inc, Delft, The Netherlands, Jun. 2007.

[11] S.U. Din and D. Bulterman, "Synchronization Techniques in Distributed Multimedia Presentation", IARIA MMEDIA 2012, Apr. 2012, pp. 1-9.

[12] T.E. Truman, T. Pering, R. Doering and R.W. Brodersen. "The InfoPad multimedia terminal: a portable device for wireless information access", IEEE transactions on computers, Oct. 1998, pp. 1073-1087.

[13] PacketVideo Coorporation, "PacketVideo OpenCORE Multimedia Framework", http://www.opencore.net/.

[14] D. Hobson-Garcia, K. Matsubara, T. Hayama and H. Munakata. "Integrating a Hardware Video Codec into Android Stagefright using OpenMAX IL", http://elinux.org/images/5/52/Elc2011_garcia.pdf.

[15] FFMPEG, "Developer Documentation", http://www.ffmpeg.org.

 ISBN: 978-1-61208-265-3