AmbiStream: A Middleware for Multimedia Streaming on Heterogeneous Mobile Devices

Emil Andriescu[1], Roberto Speicys Cardoso[2], and Valérie Issarny[1]

[1] ARLES Project-Team, INRIA Paris-Rocquencourt,
Domaine de Voluceau, Rocquencourt, Le Chesnay 78153, France
{emil.andriescu,valerie.issarny}@inria.fr
[2] Ambientic
Domaine de Voluceau, Rocquencourt, Le Chesnay 78153, France
roberto.speicys_cardoso@ambientic.com

Abstract. Multimedia streaming when smartphones act as both clients and servers is difficult. Indeed, multimedia streaming protocols and associated data formats supported by today's smartphones are highly heterogeneous. At the same time, multimedia processing is resource consuming while smartphones are resource-constrained devices. To overcome this complexity, we present AmbiStream, a lightweight middleware layer solution, which enables applications that run on smartphones to easily handle multimedia streams. Contrarily to existing multimedia-oriented middleware that propose a complete stack for multimedia streaming, our solution leverages the available highly-optimized multimedia software stack of the smartphones' platforms and complements them with additional, yet resource-efficient, layers to enable interoperability. We introduce the challenges, present our approach and discuss the experimental results obtained when executing AmbiStream on both Android and iOS smartphones. Our results show that it is possible to perform adaptation at run time and still obtain streams with satisfactory quality.

Keywords: multimedia streaming, mobile, smartphone, middleware

1 Introduction

The present generation of smartphones enables a number of applications that were not supported by previous generation cellular phones. Particularly, the greater processing power, better network connectivity and superior display quality of these devices allow users to consume rich content such as audio and video streams while moving. Not surprisingly, radios[3] and television channels[4] today provide mobile applications that allow access to their live media streams. Even video rental services[5] provide mobile applications that support movie streaming to smartphones.

[3] www.npr.org/services/mobile
[4] www.nasa.gov/connect/apps.html
[5] itunes.apple.com/us/app/netflix/id363590051

All those applications, however, assume a centralized architecture where a powerful server (or a farm of servers) provide streams to lightweight mobile devices. Node heterogeneity also remains an issue: most of those applications are available for a single smartphone platform. Indeed, to support multiple phone platforms, developers must (i) modify the mobile application to support different sets of decoders, streaming protocols and data formats and (ii) generate multiple data streams on the server side to be consumed by each mobile platform. Hence, when a resourceful server is not available, as in the case of a smartphone to smartphone streaming scenario, this approach is impractical.

In this paper, we introduce AmbiStream, a middleware-layer solution to enable multimedia streaming among heterogeneous smartphones. Such a solution can be beneficial to a large number of applications. Examples of such applications include:

- Streaming a live event directly to other devices reachable on the network;
- Sharing media on the fly between different devices (phone to tablet/TV);
- Voice call applications;
- Distributed processing of a video stream;
- Mixing augmented reality with live remote user interaction for, e.g., a networked game;
- Multimedia-rich collaboration among mobile users;
- Audio/video sharing in crisis situations when infrastructure is unavailable;
- Private communication of multimedia data between peers (without involving a third party server);

Today, to create such applications, developers must overcome a number of constraints. First, smartphones run different mobile operating systems, each supporting a different set of media encoders, decoders and streaming protocols. Second, communication is performed over wireless networks that are unstable and that do not support resource reservation, and thus streaming quality is managed by the protocol without cooperation from the network layer. Finally, the multimedia streaming software stack of each platform is highly optimized to deliver high quality audio and video while reducing resource usage.

Existing system support for multimedia streaming is unsuitable to face the smartphone challenges described above. Indeed, architectures for multimedia streaming on the Internet such as [10, 23] suppose the existence of powerful servers that can adapt content on behalf of clients, which is infeasible when the streaming server is a resource-constrained smartphone. Solutions for multimedia streaming on ad hoc networks either do not consider the problem of content adaptation [2, 12, 24] or are cross-layered, such as those surveyed in [14]. Cross-layered solutions require cooperation between application layers and networking layers, e.g., integration between the video codec and the routing protocol to optimize streaming quality.

To enable multimedia streaming among heterogeneous devices, two main challenges must be solved. First, multiple incompatible protocols for multimedia streaming exist today, and each platform supports one or a small subset of

them. As a result, smartphones must overcome the streaming protocol heterogeneity problem to be able to exchange multimedia streams with heterogeneous devices. Second, each smartphone platform generates and stores multimedia data using some specific container format, usually depending on the streaming protocols it supports. These data cannot be directly transmitted through a different streaming protocol because the media container format is specific to the protocol. Smartphones, then, must also adapt the media container format to enable translation from the native streaming protocol to non-native protocols supported by other peers.

To address the above challenges, we propose a lightweight middleware layer that complements existing software stack for multimedia streaming on smartphones with components that enable interoperability. The proposed layer defines an intermediate protocol and the associated container format for multimedia streaming among heterogeneous nodes. This layer also mediates the native media container formats and protocols to/from the intermediate streaming protocol.

The remainder of the paper is organized as follows. In the next section we review existing work on multimedia streaming in mobile environments, as well as research related to automated protocol adaptation. In Section 3 we detail the challenges involved in creating a layer to adapt multimedia streams in mobile heterogeneous environments. Section 4 presents the architecture of AmbiStream layer and explains how the main components operate: the format adapter, the protocol translator and the local media server. Section 5 discusses our initial experimental results on Android and iOS devices, which show that it is possible to adapt data and protocols at run time and also obtain streams with satisfactory quality. Finally, in Section 6, we draw our conclusions and discuss future work.

2 Related Work

Many multimedia-oriented middleware have been proposed in the literature. One of the earliest efforts in this direction was proposed in [9], which provided applications with mechanisms for late binding based on QoS constraints. The proposed platform was later extended in [8] to leverage CORBA's mechanisms for inspection and adaptation and enable applications to adapt the stream quality based on information obtained by inspecting middleware components. However, as predicted in [4], the lack of mature multimedia support at the middleware level led the industry to develop platform-specific solutions to handle multimedia streaming quality. As a result, today, most existing streaming protocols integrate mechanisms to adapt video quality to network conditions. Other middleware solutions have been proposed to provide multimedia streaming services. Chameleon [11] is a middleware for multimedia streaming in mobile heterogeneous environments. It is implemented using pure Java Core APIs in order to be portable to all Java and JavaME handsets. In Chameleon, servers send streams with different levels of quality to different multicast groups, so that clients can select the best quality according to their available resources and also adapt to changes on resource availability by selecting a multicast group providing a

stream with lower quality. This approach imposes a heavy burden on the server side, which has to keep multiple streams in parallel regardless of the number of clients. Furthermore, Chameleon implements the whole software stack required for streaming, which has a negative impact on performance.

Fewer works take into account the capabilities of current smartphones and their impact on mobile multimedia streaming. The evaluation of streaming mechanisms in [18] for Android 1.6 and iOS 3.0 tries to identify which design is better suited for mobile devices. Traditional metrics such as bandwidth overhead, start-up delay and packet-loss are used to evaluate the quality of multimedia streaming in various test situations. They observe that high network delays can result in non-continuous playback when using the HTTP Live protocol from iOS, while RTP streaming remains unaffected on Android.

Our approach to solve heterogeneity issues and enable multimedia streaming between heterogeneous mobile devices specifically stems from research on protocol translation and mediation. The work in [21] proposes a framework to formalize the process of synthesizing connectors that mediate two incompatible protocols, and suggests that data mediation can be solved through ontology integration. However, it falls short from addressing the specifics of multimedia streaming protocols, where messages are dependent on time and where message data must also be adapted during mediation.

Nakazawa et. al. [15] propose a taxonomy of bridging solutions among communications middleware platforms and present uMiddle, a system for universal interoperability, which supports mediation (entities and protocols are translated to an intermediate common representation) and is deployed as an infrastructure-provided service. This design choice is appropriate for bridging communications middleware, since it requires communication through different transport technologies that may not be available on all nodes. However, in our scenario, we want to enable peer-to-peer streaming between smartphones without using an untrusted third party server. As such, it is desirable that clients and servers are able to perform mediation independently from the infrastructure.

Another approach for the automatic translation of protocols is z2z [7], which combines a language for specification of protocols and messages, a compiler that automatically generates protocol gateways using C code, and a runtime that executes and manages protocol gateways. Z2z can translate a large number of protocols, but it does not take into account timing requirements typical from real-time streaming protocols. In such protocols, state transitions are not defined by a fixed set of message exchanges but rather by the time deadlines that the protocol must meet. With regard to message contents, z2z protocol gateways can adapt messages by rearranging data from an input message to an output message. This is also not sufficient to overcome the complexity of multimedia streaming protocols, where timing limits may require that messages are processed and regenerated when adapting protocols. Z2z evolved to Starlink [6] which enables protocol translation dynamically at run time, a particularly important feature in systems where existing protocols are unknown at compile time. Our approach

adopts an intermediate protocol and requires only clients to adapt their native protocols to the intermediate protocol, which can be done at compile time.

3 Challenges for Mobile Interoperable Media Streaming

As we mentioned in Section 1, two challenges must be solved to enable peer-to-peer streaming of multimedia data between heterogeneous smartphones: (a) how to enable interoperability among incompatible streaming protocols, and (b) how to adapt media containers to consume multimedia data transmitted through an incompatible streaming protocol.

Here, we detail the challenges introduced above. Specifically, Section 3.1 reviews the process of streaming multimedia data from a server to heterogeneous clients. Then, based on this general schema, Section 3.2 details the challenges involved when translating multimedia streaming protocols, while Section 3.3 explains the issues caused by the different media container formats available on current smartphones.

3.1 The streaming process

Streaming to heterogeneous devices is classically done by servers supporting a set of audio/video **codecs**, **media container formats** and **streaming protocols**, and comprises three phases: **media capture**, **media transmission** and **media presentation**. The steps commonly required to stream multimedia between two devices are depicted in Figure 1 and are detailed below.

Considering the sequence of steps in Fig. 1, media container formats are used in multiple cases. At Step 1 the demuxing (or demultiplexing) phase refers either to the unwrapping from a disk container if the media source is a file, or to a streamable container if it is a media server or a camera. The elementary stream obtained from Step 1 can be transcoded to a different video/audio compression format and it is then re-multiplexed to a streamable container format in Step 3. The format of multiplexing used is dependent on the streaming protocol, since in most of the cases streaming protocols support a single format.

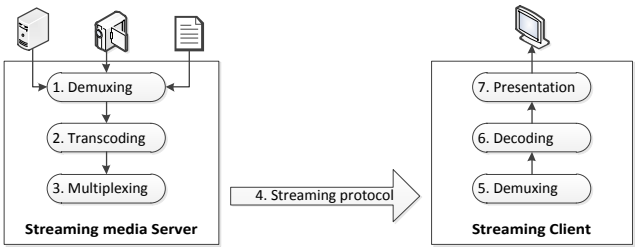


Fig. 1: Multimedia streaming process

Media Capture : Media content can originate either from a camera, stored data or from a remote source via a streaming protocol. The input can be already wrapped inside a media container (for instance, MPEG-TS or RTP) by the source hardware, so the first (Demuxing) step is optional. Possibly the most important characteristic of multimedia content is its audio/video encoding. Indeed, being a highly resource demanding operation, multimedia encoding is subject to software and hardware optimizations on both personal computers and embedded devices. The availability of encoders and decoders therefore varies depending on the mobile operating system, platform and device.

If a client does not support a decoder compatible with the server's encoder, the client cannot consume the media. When a server supports multiple encoders, multimedia data can be re-encoded on a format compatible with the client supported decoders (Step 2 in Fig. 1), but this process is resource consuming and can affect performance, especially when streaming live content.

Media Transmission : Since video and audio frames cannot be directly transferred over an IP network, they are wrapped within media containers that provide the necessary meta-information to facilitate the decoding and correct presentation at the receiver (i.e., client) side. The process of wrapping and unwrapping audio/video frames from a media container is also referred to as multiplexing and demuxing, respectively. This is related to the fact that in some container formats, frames (or frame fragments) from multiple audio and/or video tracks are interleaved. The media transmission also requires control and signalling. This task is assured by means of a communication protocol specifically designed to transport multimedia content. Streaming protocols can be divided in two subgroups:

Real-time streaming protocols are best suited for conversational content such as video conferences where user interaction with the streamed content is important.

Video on-demand protocols are designed to offer better scalability and connectivity; are usually based on the higher level Hypertext Transport Protocol (HTTP) and introduces acceptable delays.

Media Presentation : In order to correctly reproduce an audio/video stream on a mobile phone, it is required that the platform supports the given streaming protocol, media container format, the audio/video codecs and the codec profile used by the encoder. Being a resource consuming activity, multimedia decoding is usually managed by the mobile platform through hardware decoders or by efficient native code implementations. To offer a satisfactory multimedia user experience on resource-constrained devices, mobile platforms provide a default **media player** that applications can access through a standard API. This approach has the advantage of providing a uniform multimedia experience regardless of applications. However, it limits the possibilities to improve audio/video handling in mobile devices since the exposed API is generally limited. For instance, existing decoders used by the player to display multimedia content might be inaccessible for use or extension by applications.

3.2 Streaming protocol heterogeneity

Most smartphone platforms support at least one streaming protocol client. The most well known protocols used in mobile phones today are: Real Time Streaming Protocol (RTSP) [19], Apple HTTP Live Streaming (HLS) [16], Microsoft Smooth Streaming[6] and Adobe HTTP Dynamic Streaming (HDS)[7] (provided that the mobile platform supports Adobe Flash). The most commonly found on mobile platforms is RTSP, but because it uses UDP as transport protocol on unprivileged ports it is inappropriate for use in restricted networks such as 3G and public WiFi hotspots. A standard extension defined in [20] enables interleaving messages over the TCP control connection, but is not supported by most implementations. Protocols designed for video-on-demand scenarios, such as HLS and HDS, are almost equivalent in terms of functionality and concept, but differ in message formats and media containers.

Decoder / Platform	iOS	Android	BlackBerry OS	Windows Phone 7
H.263	–	+	+	–
H.264	+	+	+	+
MPEG-4	+	+	+	+
AAC-LC, AAC+, eAAC+	+	+	+	+
AMR-NB	–	+	+	+
MP3	+	+	–	+

Table 1: Audio/video decoders supported for streaming on smartphones

Still, even if the streaming protocols are incompatible by default, the encoded video and audio elementary streams may be compatible with multiple devices. For example, HLS uses H.264 codec for video, but the same codec is also largely used to stream video over RTSP to Android devices. As it can be seen in Table 1, there exists a common set of video and audio decoders available on multiple mobile phone platforms. In contrast, streaming protocol support is increasingly heterogeneous on mobile platforms, with the arrival of new proprietary protocols such as HTTP Live Streaming and Microsoft Smooth Streaming. The currently supported streaming protocols on mobile phone platforms are presented in Table 2. From both tables, we conclude that multimedia data can be exchanged between heterogeneous smartphones without the need to perform costly transcoding operations. However, it is still necessary to adapt streaming protocols to enable streaming between heterogeneous devices.

Protocol / Platform	iOS	Android	BlackBerry OS	Windows Phone 7
RTSP	–	+	+	–
RTSP interleaved	–	–	–	–
RTSP - SRTP	–	–	–	–
HTTP Live Streaming	+	+	–	–
HLS with SSL	+	–	–	–
MS Smooth Streaming	–	–	–	+
MSS with SSL	–	–	–	+

Table 2: Streaming protocols supported on smartphones

[6] http://www.microsoft.com/silverlight/smoothstreaming/
[7] http://www.macromediastudio.biz/products/httpdynamicstreaming/

3.3 Media container adaptation

The conversion between different media container formats is a critical requirement for assuring interoperability between heterogeneous streaming protocols. Supporting both real-time and video-on-demand protocols makes this task more complex due to the mismatching of properties of the protocol groups.

Encoded elementary multimedia data is stored on disk using a media container format (e.g., 3GPP, MP4, AVI). Such containers are designed to be used only in random access scenarios and therefore are not suited for streaming over a network connection. Another type of containers are streamable media containers (e.g., MPEG-TS, ASF, PIFF). They are designed to be transported over IP packet networks, provide methods for fragmenting audio and video streams and may also offer synchronization and recovery mechanisms to cope with network delays or packet losses. The wrapped media packets can contain multiplexed audio/video tracks (e.g., MPEG-TS, PIFF) or single tracks (e.g., RTP). Depending on the streaming protocol type (real-time/on-demand), multimedia fragments differ in size and structure. In general, real-time protocols use lightweight headers and small packet sizes, usually less than the MTU[8] in order to reduce the transfer delay by avoiding packet fragmentation. Video-on-demand protocols regularly use large video fragments composing 10-30 seconds of audio/video each. Such formats commonly rely on the ISO base media file format[9] structure which supports storing of multiple interleaved frames inside a single fragment, [5, 1]. Larger fragments reduce the need of receiver buffers but also introduce a start-up delay which is at least equal to the duration of the first fragment.

Real-time streaming protocols are generally designed over the UDP transport protocol because timeliness is much more important than the reliability offered by TCP. Consequently, simple reliability features, such as sequence numbers, sequence identification, synchronization codes, continuity counters, flags and timestamps are integrated in the media container layer to cope with the unreliable nature of the transport. Such features are not necessarily found in the same configuration in all formats. As a result, transforming a real-time stream to a video-on-demand fragment requires complex buffering and efficient transformation of real-time data. Such requirements impose strict temporal constraints for the transformation. It is true that real-time to on-demand protocol translation is less desirable, but interoperability should still remain possible.

4 AmbiStream Architecture

The aim of the AmbiStream middleware is to allow smartphones supporting different streaming protocols to directly connect to, and receive live multimedia content from, other smartphones without using an untrusted server for adaptation. AmbiStream consists of a set of portable server and client components as well as a plug-in interface, designed to reduce the effort of adding support for

[8] Maximum transmission unit (less than 1500 bytes for Ethernet)
[9] ISO/IEC 14496-12:2008

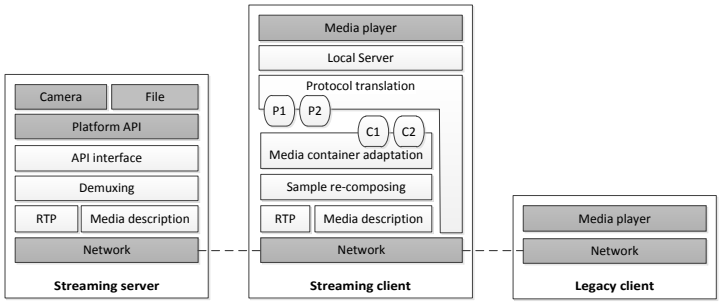


Fig. 2: The AmbiStream middleware architecture

new protocols. The structure of the middleware is presented in Fig. 2, where the greyed components are not part of AmbiStream, but are elements of the mobile platform architecture or external components.

Our work extends the approach proposed by Starlink [6] in two directions. First, our approach enables the translation between real-time and on-demand streaming protocols, which requires buffering, dropping and combination of messages to deliver time-sensitive data at the right moment. Second, we support the translation of container formats, which in the case of multimedia is dependent on the streaming protocol.

The communication is realised in a client-server mode. The streaming server, as well as the client, reside entirely on mobile devices. The current server implementation is designed to support a single streaming protocol. This protocol is translated by the client device to another protocol depending on its native protocol support. On the server side, a platform specific *API interface* has to be designed to access the *Camera* data stream. On the other hand, *File* access for streaming from pre-recorded content can be designed in a portable fashion. We assume that any input data is already multiplexed in the platform's native format (e.g., RIMM proprietary video format for BlackBerry). It is true that demuxing might not be needed if the platform's API gives access to elementary frame buffers. That is why there is an initial demuxing step in Fig. 2. Once the data is *Demuxed* (unwrapped) from its container we obtain the elementary stream tracks (e.g., the audio track) and the necessary meta-data such as sample sizes and frame durations. Considering that the middleware should enable applications to stream both in real-time and on-demand, we use *RTP* [19] as an intermediate streaming protocol. As a consequence, the middleware translates from the intermediate protocol to each existing streaming protocol, thus considerably reducing the total number of bridges required. However, this does not imply that any of the mobile platforms or devices should support this intermediate protocol natively. RTP alone is not sufficient to describe the payload characteristics such as audio/video encoders, sampling frequency, packet fragmenting method, and other media information. AmbiStream thus introduces a negotiation phase where the server sends a XML-formatted description message

to clients for the middleware to correctly instantiate the client protocol bridge and the media container format adapter.

The client receives the *Media description*, and instantiates the appropriate protocol translator (e.g., *P1* or *P2*) and media container adapter. Streaming protocol translators and media container adapters (e.g., *C1* or *C2*) are used as pluggable components created at compile time. To simplify support for a large array of protocols, these components are generated automatically from descriptions of messages and behaviour given in the form of DSL (domain specific language), as detailed in Section 4.1. The plug-ins could as well be generated at run-time, but since the required plugin of the platform is known at compile-time the only use would be to support more legacy devices. Received RTP packets are *Re-composed* into elementary streams and, once a sample is complete, they are passed to the *Media container adapter* (which is detailed in Section 4.2). Depending on the adapted protocol, the samples might be buffered at this point. A *Local server* is managed by the protocol translator that composes the necessary control messages for establishing a streaming session.

The adaptation server running on the client device can be also used as a mediator agent to solve interoperability for streaming enabled legacy devices. This is done using at least three nodes: a server running AmbiStream, a mediator also relying on our middleware and a legacy client (i.e., without AmbiStream or any other additional software installed). The mediator smartphone translates the server streaming protocol to the one supported by the legacy device.

The AmbiStream architecture enables smartphones to stream multimedia between each other without involving a third party server, since all the adaptation is performed on the client side. In terms of privacy, this solution is superior to other architectures that require the stream to pass through an untrusted server for adaptation and/or distribution. So, even though data passes through probably untrusted peers, the authenticity of the stream can still be established using an efficient security protocol such as TESLA [17]. Even legacy clients, that receive the streaming from an intermediate node instead of directly from the server can select a trusted peer based on any trust establishment protocol. The diversity of existing legacy devices such as TVs, tablets, and mobile phones motivate the use of distributed translation nodes instead of a centralised server.

4.1 Streaming protocol translation

Because writing protocol adapters for each existing streaming protocol implies a high development effort for a large number of platforms, we introduce an automated protocol translation solution, to enable easier integration of additional protocols. To achieve this, we base our solution on existing research in the domain of automated protocol translation. However, while advanced solutions for interoperability between heterogeneous protocols exist [7, 22, 6], streaming protocols tend to be more complex because of the constant data flow, time constraints and multimedia wrapper formats.

Our approach is inspired by Starlink [6], a run-time solution for protocol interoperability. Although run-time adaptation of the media format and protocol is

more flexible and enables adapting protocols that are unknown at compile-time, in our case the availability is only subject to the support of mobile platforms, thus making possible to know in advance the adaptation requirements of each mobile device. Also, the adaptation only concerns the client-side since, at the server side we use an intermediate protocol. We thus propose a simpler compile-time interoperability solution based on Starlink.

Streaming protocols are a mix of control and complex data messages. We discuss the translation of the control part of streaming protocols below, while dealing with multimedia data adaptation in Section 4.2. To create a new protocol translator, the developer must provide a high level description in the form of two DSL-based models. One describes the format and structure of messages and the other outlines the protocol states, transitions and the sequence of actions performed at each protocol state. The model is expressive enough for generating message parsers and composers for multiple existing streaming protocols. The model obtained in this form is passed on to a compiler (which is part of the currently presented solution) that produces multi-language (Java, C and C#) protocol bridges in the form of plug-ins (e.g., *P1* and *P2* in Fig. 2) for our middleware.

An schematic example of a message description for HLS protocol is shown in Fig. 3. The description is divided in *Input* and *Output* to differentiate between incoming messages that should be parsed into structured data types and outgoing messages that are composed. This distinction is more important with text protocols, where messages have loose requirements in terms of line order, optional parameters, delimiters, spacing characters and so on. The DSL proposed here supports protocols that use either binary, text or XML message formats. To assure a sufficiently expressive message description, we extract the required fields using value capture patterns defined using Posix regular expressions for text protocols, XPath for XML and based on field size and location for binary protocols. The choice of Posix regular expression for text protocols was driven by its availability on most of the platforms, most notably that it is part of the GNU C library and is compatible with the regular expressions integrated in Java standard library (java.util.regex).

Real time protocols do not usually follow a request-response messaging pattern, as implemented by on-demand ones, but rather a one-way pattern. The problem here is that a protocol translator can not produce a response by calling the real-time inner protocol. In fact, the translator must buffer the messages of the real-time protocol and, upon a request of the video-on-demand client, generate the corresponding message.

4.2 Media container format adaptation

Translating the control part of streaming protocols is not sufficient to distribute multimedia between incompatible protocols. The format in which audio/video content is wrapped also differs depending on the protocol. To achieve a complete solution, the translation between media container formats must also be taken into account. The most important factors that led to the decision to separate

```
<Protocol type="text">
  <Input>
    <Header name="http_head">
        <Var name="Url" type="String"/>
        <Rule test="capture_order(Url)">1</Rule>
        <Capture var="Method"> [RegEx] </Capture>
        <Finish test="empty_line"/>
    </Header>

    <Message name="GET_IDX">
        <Insert>http_head</Insert>
        ...
    </Message>
    ...
  </Input>
  <Output>
    <Message name="IDX">
       <Var name="$TargetDuration" type="Integer"/>
       <Line>#EXTM3U</Line>
       ...
    </Message>
    ...
  </Output>
</Protocol>
```

Fig. 3: DSL describing message formats for the HLS protocol

this part from the protocol translation model are: the much higher complexity of multimedia packets, the dependence relation between messages (order, timing, fragmenting), the buffering requirements, and the multiplexer logic required to interleave multiple media tracks inside one packet/message.

We further divide the media container adaptation in four distinct steps: sample fragmenting, fragment packaging, multiplexing and final adjustment. The process of adapting a stream composed of two tracks (one audio and one video) is presented in Fig. 4. Each of the four phases is defined by the developer using a DSL to describe multimedia containers, different from the ones used for protocol description. Similarly to the generation of protocol translation plug-ins, the description of the multimedia container adaptation is compiled to be deployed to designated platforms. To simplify the description, a number of media packet-related parameters are exposed through the DSL. Parameters include: the length of the media payload, media encoding, fragmentation flag, sampling frequency, sequence number, inner frame sequence number and first/last fragment flag. The components for protocol description and container adaptation are considered to be independent, thus allowing, for example, a protocol to choose between multiple supported data formats. The *Sample Re-composing* middleware component (see Fig. 2) provides real-time input to the container adapter in the form of elementary stream samples for audio and frames for video.

Because we use a real-time protocol (i.e., RTP) for transporting multimedia data, the problem of timing should also be taken into account. We thus add a time-stamp reference to each packet resulting from any of the four phases of media format adaptation. Fragments of one frame share the same time-stamp information, while messages composing multiple frames contain the time-stamp of the first frame and their duration. The time required for a frame to pass through

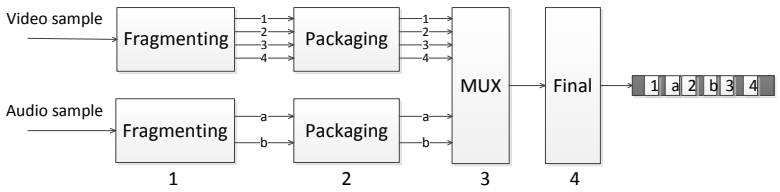


Fig. 4: Adapting the media container format

all of the phases required by the format should not exceed the sampling interval of the content. Failing to assure this property can cause the client to run out of buffered data, resulting in playback stalls. In order to prevent such behaviour, frames should be explicitly dropped such that the output of the conversion is completed at the right time to assure a fluent playback. At this moment, no QoS related limits of packet drops are considered.

The fragmenting step defines the way large audio or video samples are divided into smaller segments according to the limits imposed by the streaming protocol, by the media container or by the network configuration. For example, in the case of MPEG-TS, the samples are split into fragments which are inferior in size to 184 bytes, such that they can be correctly wrapped inside the standard 188 byte packets. For RTP, fragmentation follows the standard RTP Payload Format depending on the codec used (for instance, the one described in [25]). We note that a simplified description of the packet format is very useful in the case of RTP, where there are multiple payload formats depending on the media encoder used. In the case where media content is composed of multiple tracks (i.e., one video and one audio track), two separate fragmenting units are used. The number of fragments created from single frames is variable. Each fragment contains a reference to the time-stamp of its originating frame. The time required for fragmenting one frame should never exceed the sampling interval of the content.

The packaging stage adds individual packet headers. This transformation conforms to [19] for RTP packets and [13] for MPEG-TS. Depending on the protocol, the resulting packets are passed to the multiplexer or sent directly to the protocol translator.

The multiplexing phase assures time-division multiplexing for a set of given fragments or frames of multiple data tracks. Depending on the format, the multiplexing is done at a frame level or at a frame-fragment level. In order to achieve multiplexing at frame level, phase one of the adaptation should be skipped. This phase outputs only at a given time or data limit. Such a limit is necessary to be able to produce media fragments of specified duration or size. The split is always done at random access points of the stream, such that no reference between frames is lost.

The final transformation adds extra headers or packets, such that the resulting fragment is recognised as valid by standard client protocol implementations.

Many existing media container formats also contain a number of specific fields which are particularly hard to model. One example is the MPEG2 Transport Stream [13], which requires a 32-bit cyclic redundancy check value to be added to the Program Association Table package. In such a case we offer the possibility to add function"hooks" inside the DSL media container description. The compiler uses these to generate function templates, that developers can later implement.

5 Experimental Results

In order to evaluate the presented solution, we have implemented AmbiStream in Java and Objective-C and used it on Android and iOS. The goal of the experiments presented here is to evaluate the overall performance of the middleware and the achievable stream quality. The experiments were performed on both Android and iPhone smartphones.

Device	Samsung GT-I9000	Google Nexus One	iPhone 3G
Role	Server	Client	Client
Platform	Android 2.2.1	Android 2.3.4	iOS 4.2.1
CPU	1 GHz (S5PC110)	1 GHz (QSD8250)	412 MHz
Memory	512 MB	512 MB	128 MB
Media framework	PV OpenCORE	Stagefright	AV Foundation
Stream support	RTSP	RTSP/HLS	HLS

Table 3: Test smartphones used

In both of the experiments presented below, the same set of source media files was used. The test files have a duration of 210 seconds, are encoded with a single (H.264-avc video) track, have a CIF frame-size (352 by 288), and a frame-rate of 30 fps. The test is conducted for 16 different bit-rates between 50kbps and 1500kbps using the mentioned file format and content. Each set of tests is repeated at least three times, so each of the metrics presented is characterized by 168 minutes of video streaming to each client device. In total, more than 16 hours of streaming between smartphones were necessary. The mobile phones used are mentioned in Table 3. The first two (Samsung GT-I9000 and Google Nexus One) are used in the first experiment, and all three in the second one.

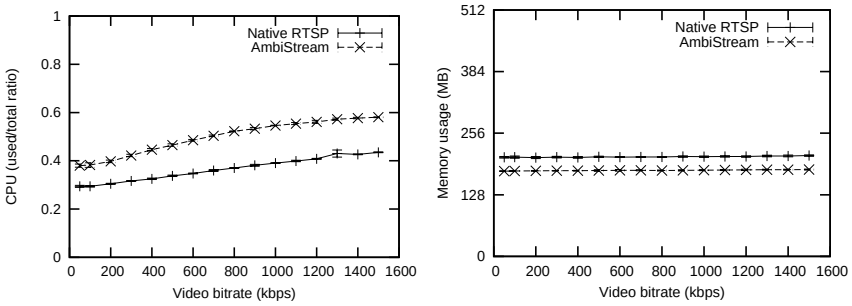
5.1 Collecting mobile device performance data

Although RTSP provides out-of-band feedback of stream quality through RTCP, we have decided not to use this feature to obtain information related to the quality of service. This is due to the fact that in the case of the media framework Pocket Video OpenCore (used by Android platform in versions preceding 2.3) the information provided is not sufficiently precise. For example, the interval jitter value reported, used to observe the effect of network packet delays, is usually ten times higher than what we found at network level or on the client device. Furthermore, on the newer Stagefright media framework the feedback always reports no packet loss and inter-arrival jitter equal to zero. Android also provides an information callback from the media player service. Unfortunately,

this information is limited to a small set of event codes and does not include any metric.

We have chosen to favour system-wide metrics to more specific ones (i.e., metrics of the application process) because we also make use of native system services and because mobile platforms do not frequently provide equivalent metrics. We use as metrics for device performance: the total CPU utilization and the system-wide used RAM memory. Quality of service metrics considered are the packet delay variation (also referred to as inter-arrival jitter, described in [19]) and packet loss ratio. The quality metrics are only provided for the case where the protocol is adapted. The values are obtained at the middleware level and should indicate the maximum bit-rate achievable while still providing satisfactory quality. The reference test cases, used to compare the overall performance, make use of system media services directly.

On Android mobile phones, the CPU and memory information is obtained by accessing the *proc* filesystem, used as an interface to the operating system kernel on most Linux based distributions. The logs are stored in the internal memory of both Android phones. To avoid that the access to the filesystem and data parsing are influencing the final results, the access to the */proc/stat* and */proc/meminfo* is done every five seconds, and the same file-descriptors are reused multiple times until the end of the test. On the iOS platform, system performance information was collected using the tools integrated with the development kit.



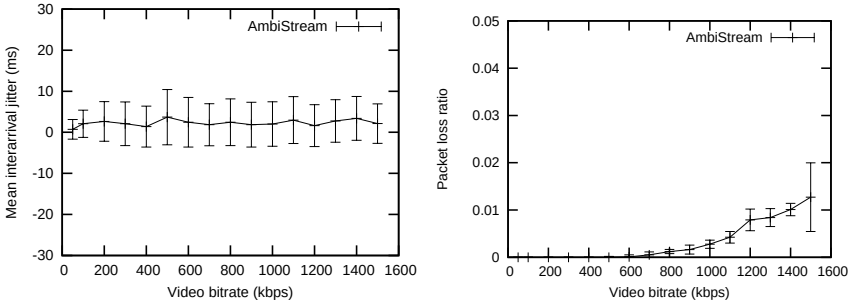
(a) CPU usage on Nexus One (b) RAM usage on Nexus One

Fig. 5: AmbiStream performance on Nexus One (RTSP)

5.2 Translating to RTSP between Android smartphones

In this first experiment, we show that adaptation from the middleware intermediate protocol to RTSP/RTP/UDP is sufficiently efficient to be used in mobile multimedia-enabled applications. Since in this case the message format used by the client protocol is equivalent to the middleware transport protocol, the wrapping and unwrapping of messages is simpler than in other cases. Nevertheless, this client protocol is the only real-time streaming protocol currently available on mobile phones, and is thus interesting to analyze the feasibility of streaming real-

time multimedia data through the middleware. Another experiment involving a more complex media format adaptation is presented at the end of this section. In this test we use two server implementations: one using the AmbiStream intermediate protocol and the other using RTSP. The RTSP server is not part of the solution but it is used in this experiment to determine the overhead of the adaptation (on the client-side) with reference to the native RTSP support.



(a) Inter-arrival jitter (b) Packet loss

Fig. 6: Adapted stream quality (RTSP)

In the case of protocol translation to RTSP, the performance of the client device (realising the content and protocol adaptation) is not badly affected, with a processing overhead of less than 20% compared to a native RTSP session (see Fig. 5a). As with all of the experiments conducted, the memory usage remains constant, or increases slightly because of buffers required for higher data-rates (Fig. 5b). The fact that our solution uses slightly less memory than the reference one is due to the way jitter buffers are managed internally by the RTSP client, most probably being influenced by the different transport protocols (UDP and TCP). The quality of the stream remains within acceptable limits in terms of inter-arrival jitter (see Fig. 6a) and packet loss (Fig. 6b), for all the test cases (from 50 to 1500kbps) considered.

5.3 Translating to HLS between Android and iOS smartphones

The second experiment consists of translating the intermediate middleware protocol to HTTP Live Streaming, using two different client platforms: Android 2.3.4 and iOS 4.2.1. The choice of the smartphones is motivated by their native support of HLS. This way we can reason about the overhead introduced by our middleware layer with two different devices. Contrary to the first experiment, this one requires data conversion between RTP and MPEG-TS. MPEG-TS is one of the most used multimedia formats, most notably for digital television. The conversion from RTP to MPEG-TS requires a large number of transformations, thus providing a good impression of achievable on-the-fly conversion limits of media formats on current generation smartphones.

Because HLS protocol requires the existence of a cached amount of content on the server-side before a client can connect (and begin playback), while the

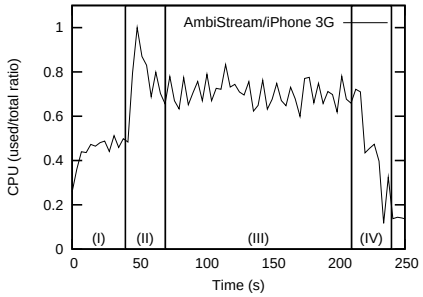


Fig. 7: Data capture (HLS)

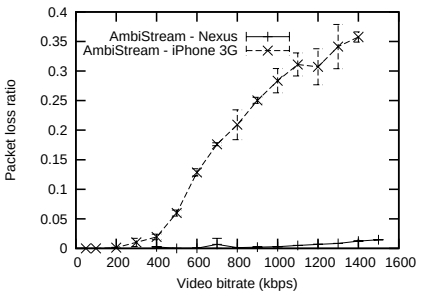
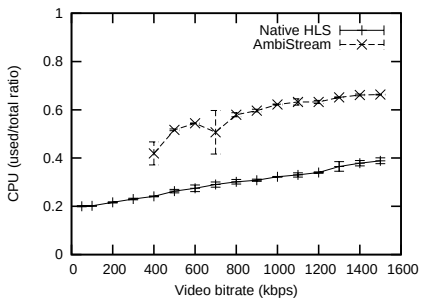
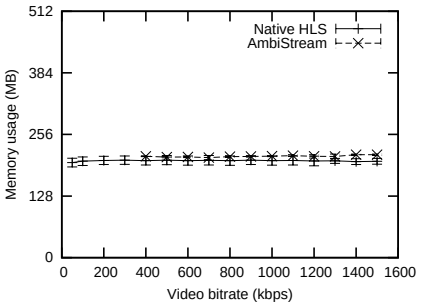


Fig. 8: Packet loss (HLS)



(a) CPU usage

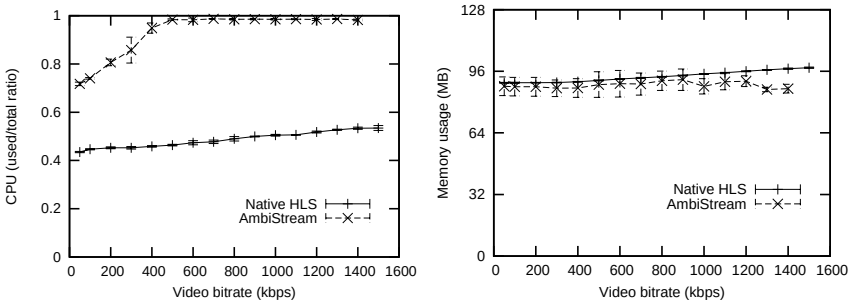


(b) RAM usage

Fig. 9: AmbiStream performance on Nexus One (HLS)

intermediate AmbiStream protocol does not, 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 in this situation. This is not the case when the device supports a real-time protocol. During this delay period, less memory and CPU are used. To better evaluate the performance of the devices, we divide the experiment run in four periods (e.g., as shown in Fig. 7 for CPU utilisation): *(I) the buffering period* (only multimedia data adaptation is performed), *(II) the media-player start-up* (causes a short increase in CPU usage), *(III) the streaming period* (both data adaptation and playback are performed) and *(IV) the stream-end* (the source has finished streaming, but the playback is continued until buffer depletion). Thus, only the part *(III)* of the observation was used to produce the results presented in Figures 9 and 10.

As expected, the difference in container formats (RTP and MPEG-TS), increases the overhead of AmbiStream. For Android platform, the tests for bit-rates inferior to 400kbps (in Figures 9a and 9b) were discarded due to the existence of a minimal caching size, requiring a longer start-up delay. While on the Nexus One, the overhead introduced does not reach a quality limit for bit-rates below 1500kbps, the iPhone 3G is only able to adapt streams of up to 400kbps. Above this limit, the packet loss (see Fig. 8) becomes noticeable and the media-



(a) CPU usage (b) RAM usage

Fig. 10: AmbiStream performance on iPhone 3G (HLS)

player suffers playback stalls. The results on the iPhone are worse due to the significantly lower processing power and memory (see Fig. 3). Nevertheless, according to the mobile platform providers, a 400kbps video bit-rate is considered to be medium/high quality for smartphones[10] [11]. Considering the results in Figure 10b, we see that the memory usage is decreasing (in the case of AmbiStream) for higher video bit-rates. This behaviour is normal considering the packet loss (see Figure 8).

6 Conclusions and Future Work

In this paper we have identified the challenges raised by the heterogeneity of the streaming protocols of existing mobile phone platforms. Further, we have introduced the AmbiStream multimedia-oriented middleware architecture, designed to enable the multi-platform and multi-protocol interoperability of streaming services. We have also shown the applicability of the presented solution with an experiment on two different platforms and two different streaming protocols.

AmbiStream was modelled taking into consideration the architecture of modern smartphone platforms, such that resource critical operations (e.g., multimedia decoding) are managed by each platform internally. We prove that automated streaming protocol adaptation can be done locally on mobile phone platforms without sacrificing performance or extensibility. Furthermore, we enable legacy devices to employ unsupported streaming protocols by using an AmbiStream-enabled device as mediator intermediary.

We intend to continue this work by extending the current model, taking into account challenges such as routing over different networks and multi-peer collaboration. We will then integrate AmbiStream with iBICOOP [3], a middleware designed to enrich the user collaboration and provide seamless access across different networks and devices. Such an integration will complement the existing solution with features such as discovery, distributed storage and partnership

[10] http://developer.apple.com/library/ios/# technotes/tn2224/_ index.html
[11] http://developer.android.com/guide/appendix/media-formats.html

management, enabling the development of rich cross-platform and multimedia-enabled applications. We will also port the solution to Blackberry and Windows Phone to evaluate the approach on a greater number of mobile platforms.

Acknowledgement. *This work is partially supported by the FP7 ICT FET IP Project CONNECT.*

References

1. Adobe Flash Video File Format Specification, Version 10.1 (Aug 2010), `http://download.macromedia.com/f4v/video_file_format_spec_v10_1.pdf`
2. Andronache, A., Brust, M.R., Rothkugel, S.: Multimedia content distribution in hybrid wireless networks using weighted clustering. In: Proceedings of the 2nd ACM international workshop on Wireless multimedia networking and performance modeling. WMuNeP '06, ACM (Oct 2006)
3. Bennaceur, A., Pushpendra, S., Raverdy, P.G., Issarny, V.: The iBICOOP middleware: Enablers and services for emerging pervasive computing environments. In: PerWare 2009 IEEE Middleware Support for Pervasive Computing Workshop (Oct 2009)
4. Blair, G.: On the failure of middleware to support multimedia applications. In: Interactive Distributed Multimedia Systems and Telecommunication Services (Oct 2000)
5. Bocharov, J., Burns, Q., Folta, F., Hughes, K., Murching, A., Olson, L., Schnell, P., Simmons, J.: The Protected Interoperable File Format (PIFF) (Mar 2010)
6. Bromberg, Y.D., Grace, P., Réveillère, L.: Starlink: runtime interoperability between heterogeneous middleware protocols. In: Proceedings of 31th International Conference on Distributed Computing Systems, ICDCS (IEEE). (Jun 2011)
7. Bromberg, Y.D., Réveillère, L., Lawall, J.L., Muller, G.: Automatic generation of network protocol gateways. In: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware. Middleware '09 (Dec 2009)
8. Coulson, G., Blair, G., Davies, N., Robin, P., Fitzpatrick, T.: Supporting mobile multimedia applications through adaptive middleware. IEEE Journal on Selected Areas in Communications (Sep 1999)
9. Coulson, G.: A configurable multimedia middleware platform. IEEE MultiMedia (Jan 1999)
10. Cruz, R.S., Nunes, M.S., Gonalves, J.E.: A personalized HTTP adaptive streaming WebTV. In: User Centric Media. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering (Dec 2010)
11. Curran, K., Parr, G.: A middleware architecture for streaming media over IP networks to mobile devices. In: Wireless Communications and Networking (Mar 2003)
12. Do, N.M., Hsu, C.H., Singh, J.P., Venkatasubramanian, N.: Massive live video distribution using hybrid cellular and ad hoc networks. In: Proceedings of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2011) (Jun 2011)
13. ITU-T Rec. H.222.0 — ISO/IEC 13818-1, Generic coding of moving pictures and associated audio information, `http://www.iso.org/iso/catalogue_detail?csnumber=44169`
14. Lindeberg, M., Kristiansen, S., Plagemann, T., Goebel, V.: Challenges and techniques for video streaming over mobile ad hoc networks. Multimedia Systems 17(1) (Feb 2011)

15. Nakazawa, J., Tokuda, H., Edwards, W.K., Ramachandran, U.: A bridging framework for universal interoperability in pervasive systems. International Conference on Distributed Computing Systems (Jul 2006)
16. Pantos, R., May, W.: HTTP Live Streaming. (Internet-Draft) (Mar 2011), http://tools.ietf.org/html/draft-pantos-http-live-streaming-06
17. Perrig, A., Song, D., Canetti, R., Tygar, J.D., Briscoe, B.: Timed Efficient Stream Loss-Tolerant Authentication. RFC 4082 (Proposed Standard) (Jun 2005), http://tools.ietf.org/html/rfc4082
18. Ransburg, M., Jonke, M., Hellwagner, H.: An evaluation of mobile end devices in multimedia streaming scenarios. In: Mobile Wireless Middleware, Operating Systems, and Applications (Jul 2010)
19. Schulzrinne, H., Casner, S., Frederick, R., Jacobson, V.: RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard) (Jul 2003), http://tools.ietf.org/html/rfc3550, updated by RFCs 5506, 5761, 6051, 6222
20. Schulzrinne, H., Rao, A., Lanphier, R.: Real Time Streaming Protocol (RTSP). RFC 2326 (Proposed Standard) (Apr 1998), http://tools.ietf.org/html/rfc2326
21. Spalazzese, R., Inverardi, P., Issarny, V.: Towards a formalization of mediating connectors for on the fly interoperability. In: Joint Working IEEE/IFIP Conference on Software Architecture 2009 & European Conference on Software Architecture 2009 (Sep 2009)
22. Tegawendé, B., Réveillère, L., Bromberg, Y.D., Lawall, J., Muller, G.: Bridging the gap between legacy services and web services. In: Middleware 2010. Lecture Notes in Computer Science (Dec 2010)
23. Van Lancker, W., Van Deursen, D., Mannens, E., Van de Walle, R.: Implementation strategies for efficient media fragment retrieval. Multimedia Tools and Applications (Mar 2011)
24. Vu, L., Nahrstedt, K., Rimac, I., Hilt, V., Hofmann, M.: ishare: Exploiting opportunistic ad hoc connections for improving data download of cellular users. In: GLOBECOM Workshops, 2010 IEEE (Dec 2010)
25. Wang, Y.K., Even, R., Kristensen, T., Jesup, R.: RTP Payload Format for H.264 Video. RFC 6184 (Proposed Standard) (May 2011), http://tools.ietf.org/html/rfc6184