Master Thesis

Live Audio Streaming in Mobile Peer-to-Peer Systems

Marc Bruggmann

October 18, 2011

Supervisor: R. Meier

Prof. Dr. R. Wattenhofer
Distributed Computing Group
ETH Zürich

Abstract

Mobile devices are becoming part of our daily life. One of the most important uses for them is consumption of multimedia content, which is often streamed from the internet through cloud music and video services.

The Pulsar project provides a new, peer-to-peer based content distribution system for rich media content. The goal of this thesis is to make this technology usable on mobile devices. Mobile devices have specific requirements like limited computational capacity and battery life. We present an adapted and optimized version of Pulsar, tailored for use on mobile devices.

During the course of the thesis, it became apparent that the original Java implementation of Pulsar is not efficient enough for use on smartphones. Therefore, a major part of the thesis was the development of a Java to C++ converter, which allowed us to automatically port the whole Pulsar stack to C++.

To validate the approach on a real device, we implemented a feature-complete radio streaming application, optimized for low resource usage, for the Android platform. This application is used as a benchmark for our Java to C++ converter, and to verify the usability of the adapted Pulsar framework on mobile devices.

Acknowledgments

Throughout this master thesis, a lot of people provided advice and mentoring. In particular, i would like to thank:

First and foremost my girlfriend, family and friends for forgiving my constant lack of time during the last six months. Professor Roger Wattenhofer, who gave me the opportunity to work on such an interesting topic. Mark Nevill, who had to endure endless questions about C++. The people at the Distributed Computing Group at ETH, for coffee and table soccer breaks. Remo Gisi, Christian Helbling, Simon Gerber, Christina Bricalli, André Bruggmann, and Eleanor Nevill for providing feedback about this paper. Pascal von Rickenbach, who always had the right words of encouragement. Benjamin Sigg, who implemented the garbage collection support. Most importantly, I would like to thank my supervisor Remo Meier for his impressive knowledge of Pulsar and everything related to Java, the implementation of various components needed for the thesis, countless hours of collaborative development, and being the guy to go to if something did not work out as planned.

Zürich, October 18, 2011

Marc Bruggmann

Contents

1 Introduction **1**
1.1 Multimedia Streaming Services 1
1.2 Contribution 2
1.3 Outline 2

2 Background **5**
2.1 Live Streaming 5
2.1.1 Broadcasting and Transcoding 5
2.1.2 Content Distribution Networks 6
2.1.3 Streaming Protocols 7
2.2 Mobile Devices 8
2.2.1 Platforms 8
2.2.2 Radio Streaming Applications 8
2.3 Automatic Language Conversion 10

3 Content Distribution for Mobile Devices **11**
3.1 Requirements 11
3.2 Content Distribution Network 12
3.3 Target Platform 12
3.4 Application Prototype 12

4 Java to C++ Converter **15**
4.1 Requirements and Scope 15
4.2 Architecture 16
4.2.1 Analyzer 18
4.2.2 Parser 18
4.2.3 Rewriters 19
4.2.4 Optimizer 20
4.2.5 Flattener 20
4.2.6 Backend 21
4.2.7 Runtime Implementation 21
4.3 Data Types 22
4.3.1 Basic Types 22
4.3.2 Object Types 22
4.3.3 Arrays 23
4.3.4 Enumerations 24
4.3.5 Strings 24

4.4 Language Features 26
4.4.1 Classes and Interfaces 26
4.4.2 Inheritance 26
4.4.3 Polymorphism 28
4.4.4 Packages and Imports 28
4.4.5 Loops 28
4.4.6 Exceptions 29
4.4.7 Initializers 31
4.4.8 Generics 32
4.4.9 Threads and Synchronization 32
4.4.10 Network Stack 34
4.4.11 Garbage Collection 34

5 Adapting Pulsar for Mobile Devices **37**
5.1 Player Abstraction 37
5.2 Performance 38
5.3 Codec 38
5.4 Multiple Bitrates 39
5.5 Security 40

6 Radio Streaming Application **41**
6.1 Architecture 41
6.2 Pulsar Streams 42
6.3 SHOUTcast Streams 43
6.4 User Interface 43

7 Evaluation **45**
7.1 Generic Testcases 46
7.2 Radio Streaming Application 49

8 Conclusion **53**
8.1 Achievements 53
8.2 Future Work 54

List of Figures

2.1 Content Distribution Networks 7
2.2 Mobile Radio Streaming Applications 9

3.1 User Interface of the Application Prototype 13

4.1 Java to C++ Converter Architecture 17
4.2 Abstract Syntax Tree 19
4.3 Arrays with Length Information 23

6.1 Radio Streaming Application Architecture 42
6.2 User Interface of the Radio Streaming Application 43

7.1 Smartphones 46
7.2 Generic Testcases: Desktop 48
7.3 Generic Testcases: Nexus One 48
7.4 Generic Testcases: Motorola Defy 49
7.5 Radio Streaming Application: CPU Usage 50
7.6 Radio Streaming Application: CPU Usage by Component 51

List of Tables

2.1 Comparison of Audio Codecs 6
2.2 Worldwide Smartphone Sales 9
2.3 Automatic Programming Language Converters 10

4.1 Optimizer Step 20
4.2 Contents of the Runtime Implementation 21
4.3 Basic Types in Java and C++. 23

7.1 Evaluation Devices 45
7.2 Evaluation Results: Generic Testcases 47
7.3 Evaluation Results: Radio Streaming Application 50

List of Listings

4.1 Enumerations 25
4.2 Classes 27
4.3 Foreach Loops 29
4.4 Finally 30
4.5 Initializers 31

1 Introduction

Mobile devices like smartphones, netbooks, and tablets are becoming part of our daily life. They are selling better year after year, and continuously replace – not only for the young generation – our radios, televisions and desktop computers at home. These new appliances use the ubiquitously available internet connections, be it through wireless hotspots or mobile data connections, to deliver content to the users wherever they are.

One of the most prominent uses for mobile devices is entertainment. Users enjoy having their collection of music, movies or TV shows on the devices they carry with them. This basic need led to the development of the walkman, one of the first devices that reached cult status with a global customer base. Nowadays, Apple's iPod is widely used as a synonym for portable music players, mostly because it is easy to buy, manage and listen to your favorite music from your computer or your various mobile devices.

1.1 Multimedia Streaming Services

An increasingly visible trend in entertainment solutions is the use of cloud services. Several large companies such as Amazon, Google or Apple launched cloud-based music services this year. The advantages of these are that the user's music collection is stored online and thus accessible from different devices any time there is an internet connection. In the cases of Amazon's cloud player and Apple's iCloud, the music storage is coupled with an online store to buy new songs. Purchasing tracks makes them automatically available through the cloud storage.

A similar trend can be seen with radio stations. Traditionally, radio signals are transmitted over the air using the FM and AM frequency bands. A single antenna can serve

an unlimited number of users within its transmission range, and thus makes it a particularly good broadcast method for this purpose. Nevertheless, more and more users listen to radio through the internet. In Germany, for example, more than 16 million people are regularly listening to radio stations through the internet, a number that is projected to grow to 21 million by 2013[1]. More than 40 percent of the internet users are listening to a music service while surfing.

In contrast to the traditional over-the-air radio broadcasting, the increasing number of users listening to the radio over the internet leads to additional costs for the radio stations. Currently available streaming solutions require a certain amount of server and bandwidth resources for every single listener. This means that radio stations need to invest a lot of money into additional server infrastructure to cope with the projected growth of listeners of their streaming services.

1.2 Contribution

This thesis explores a new approach to stream multimedia content, in particular music, to mobile devices. Our solution is based on Pulsar, a peer-assisted content distribution platform. We believe that using a peer-assisted streaming technology leads to various technical and economical advantages in this field. In particular, using Pulsar as a distribution platform allows for a growing audience that consumes rich media content online, without the associated added cost that traditional solutions involve on the server side.

The goal of this thesis is to integrate mobile devices into the Pulsar content distribution platform. This includes in particular:

- Creating a reusable module to access Pulsar streams from mobile devices.
- Optimizing the Pulsar protocol for additional requirements of mobile devices.
- Implementing a fully functional radio streaming application for a smartphone platform.

In order to achieve acceptable performance on mobile devices, an additional tool was developed to convert the Pulsar codebase from Java to C++ automatically.

1.3 Outline

The rest of the thesis is structured as follows: Chapter 2 gives some background for this thesis, including related scientific work and existing technologies and applications in the field of media streaming. Chapter 3 defines the requirements of our content distribution network for mobile devices and describes the experience and conclusions gathered with developing a prototype radio streaming application. The automatic program-

[1] http://corporate.radio.de/werbung.html

ming language converter used to translate the Pulsar codebase to C++ is presented in Chapter 4.

Chapter 5 presents the adapted version of Pulsar. Following this, Chapter 6 describes the smartphone application that we implemented. It goes into more detail about the architecture, the underlying technology as well as the user interface. At the end, Chapters 7 and 8 summarize the results achieved, and give a conclusion and outlook into future developments.

2 Background

Realizing a mobile peer-to-peer system that streams live content to mobile devices requires knowledge about various technologies and research fields. As we deal with a very broad topic, this chapter can only provide a general overview of the involved concepts, relevant publications, and applications.

2.1 Live Streaming

Presenting a continuous video or audio track to the user if the content is not available on the device beforehand is called streaming. Streaming involves a source signal, a delivery channel and a client application. The *source signal* can be a file stored on a server, or an input device such as a video camera or a microphone. This source signal is usually transmitted from a broadcasting application to a media publisher that provides it to potential end users by means of a *streaming protocol*. The internal delivery of the content from the media publisher to end users is controlled by a *content distribution network*.

2.1.1 Broadcasting and Transcoding

Several applications can be used as the source of a live stream. For audio streaming, the most commonly used applications are SAM Broadcaster[1], SimpleCast[2] and Winamp[3].

[1] SAM Broadcaster: http://www.spacialaudio.com/?page=sam-broadcaster
[2] SimpleCast: http://www.spacialaudio.com/?page=simplecast
[3] Winamp: http://www.winamp.com

The broadcasting application is responsible for creating a connection to the media publisher and consistently delivering the source signal of the stream.

It is often desirable to have a stream available in different formats. For example, having a stream available in multiple bitrates and multiple encoding formats allows a listener to choose the one that matches best the available bandwidth and the capabilities of his device. There are two possibilities to achieve this:

- Let the broadcasting software encode the signal in different formats, and send all of them to the media publisher. This requires additional processing power and bandwidth.
- Let the media publisher encode a single source signal into all the required formats (transcoding). This is easier for the broadcaster, but requires additional resources on the media publisher side.[4]

Table 2.1 lists some of the common audio codecs used for streaming [24]. MP3 is the oldest of them, but still the most widely used codec. It is supported on a lot of consumer devices, and thus easy to integrate into a streaming application. AAC+ was developed to provide higher perceived quality at lower bitrates than MP3. [6] gives a nice overview of the algorithms behind the MP3 and AAC+ codecs. The only non-patented codec that is widely used is Ogg Vorbis [15]. Its main advantage is that it can be used without licensing costs; therefore it is often used in open-source projects as well as commercial applications such as Spotify[5]. WMA was implemented by Microsoft in 1999 as a competitor to the MP3 format which was widely used at that time, because it could be integrated in their Windows operating system without paying licensing fees to the Frauenhofer Society.

Codec	Creator	First Release	Patented	Typical Bitrates
MP3	Fraunhofer Society	1993	Yes, Non-Free	128 to 320 kbit/s
AAC+	MPEG Audio Committee	1997	Yes, Non-Free	16 to 320 kbit/s
Ogg Vorbis	Xiph.Org Foundation	2000	No, Free	32 to 320 kbit/s
WMA	Microsoft	1999	Yes, Non-Free	32 to 320 kbit/s

Table 2.1: Comparison of audio codecs.

2.1.2 Content Distribution Networks

An interesting aspect of live streaming is how the content is distributed from the media publisher over the network to the listeners. The internet is designed according to the end-to-end principle [18]. Most of the functionality is implemented at the endpoints, the servers and the clients, while keeping the network infrastructure as simple as possible. This allows building extremely efficient infrastructure, specialized in forwarding data packets.

[4] This is why transcoding on the media publisher leads to significant additional costs.
[5] Spotify: http://www.spotify.com/

While the end-to-end principle is in general a good fit for the internet, it poses some problems for the use case of live streaming. The media publisher has to provide hardware resources and bandwidth for every incoming connection, and thus for every single listener of the stream.

To improve this situation, one can use a content distribution network (CDN). A traditional CDN consists of a collection of edge servers that attempt to offload work from the origin servers by delivering content on their behalf [12]. The edge servers cache content of the origin servers at different locations around the network (see Figure 2.1). Requests can then be routed to an "optimal" edge server, where the notion of optimal can include geographical, topological or latency considerations. [10] describes the techniques used in a CDN: web caching, server load-balancing, request routing, and content services.

Although not originally developed for this purpose, peer-to-peer (P2P) systems can also be used as a CDN. One of the central characteristics of P2P systems – which reflects upon issues of performance, availability and scalability – is their ability to function, scale, and self-organize in the presence of a highly transient population of users, without the need for a central server administration [2]. This makes them particularly suitable for content distribution in streaming systems, because listeners often join and leave streams. Another advantage of P2P content distribution networks is that they do not require lots of edge servers, but can exploit connections between nearby users directly.

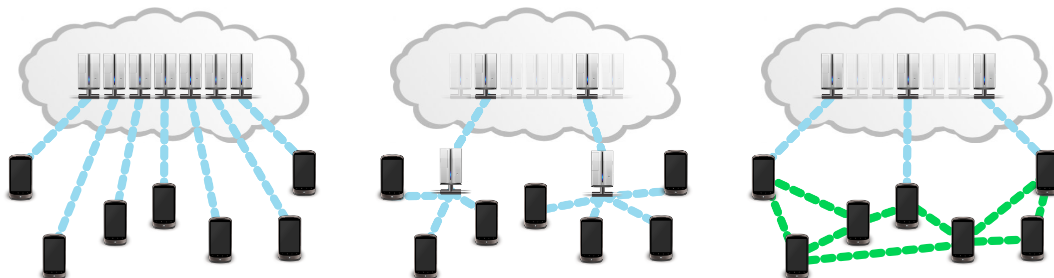

Figure 2.1: Content distribution networks. Left: Traditional streaming without a content distribution network. Center: Content distribution network with edge servers. Right: Peer-to-Peer content distribution network.

2.1.3 Streaming Protocols

The communication between a client application and the media publisher works according to a streaming protocol. Several streaming protocols exist for different purposes.

The de-facto standard for live audio streaming is *SHOUTcast* [16]. It is based on the client-server model and TCP as a transport protocol. The server broadcasts a stream of bytes with raw audio data (usually MP3), and metadata that is interleaved therein periodically. Its simplicity makes SHOUTcast easy to implement for client applications.

Listening to SHOUTcast streams is supported in Winamp[6], iTunes[7], VLC[8], Amarok[9], and various other music players.

An alternative to SHOUTcast, with a similar purpose, is *HTTP Live Streaming* [3]. It works by dividing the stream into small HTTP-based file downloads that each contain a chunk of the audio data. The same chunks are available in different bitrates, allowing a client to continuously choose an appropriate format according to the available bandwidth. HTTP Live Streaming is the preferred format for Apple mobile devices[10] and is also available on Android since version 3.0.

The *Real Time Messaging Protocol* (RTMP) is a streaming protocol developed by Adobe [1]. RTMP is supported by Adobe Flash Media Server and Flash Player. The transport protocol is TCP-based and supports low-latency audio and video distribution as well as a remote procedure call interface.

A streaming protocol extensively used in internet telephony and video teleconference applications is the *Real-time Transport Protocol* (RTP) [11]. RTP is designed to support the real-time transmission of audio and video signals over IP networks. It is used together with a control protocol, e.g. the *RTP Control Protocol* (RTCP), that monitors transmission statistics, synchronizes multiple streams and controls quality of service.

2.2 Mobile Devices

2.2.1 Platforms

Table 2.2 gives some insight into the global smartphone sales in the second quarter of 2011 [9]. One can clearly see that Android is the predominant smartphone operating system as of 2011, with the biggest growth compared to 2010. The second most important platform is iOS, which grew from 14 to 18 percent market share. Symbian, on the other hand, lost its position as market leader. Therefore, Android and iOS with a combined market share of more than 60 percent are currently the primary targets for mobile application development.

2.2.2 Radio Streaming Applications

Even though smartphone producers often bundle web radio compatible media players with their devices, radio streaming applications are still popular in the Android Market as well as the Apple App Store.

[6] Winamp: http://www.winamp.com
[7] iTunes: http://www.apple.com/itunes/
[8] VLC: http://www.videolan.org
[9] Amarok: http://amarok.kde.org/
[10] For example iPod, iPhone, and iPad.

	Q2 2010		Q2 2011	
Platform	Units (Thousands)	Market Share (%)	Units (Thousands)	Market Share (%)
Android	10'652.7	17.2	46'775.9	43.4
Symbian	25'386.8	40.9	23'853.2	22.1
iOS	8'743.0	14.1	19'628.8	18.2
RIM	11'628.8	18.7	12'652.3	11.7
Bada	577.0	0.9	2'055.8	1.9
Windows	3'058.8	4.9	1'723.8	1.6
Others	2'010.9	3.2	1'050.6	1.0
Total	62'058.1	100	107'740.4	100

Table 2.2: Global smartphone sales by platform, second quarter 2010 vs. second quarter 2011.

On Android, TuneIn Radio is by far the most used radio streaming application with more than 10 million installations[11]. XiiaLive appeals with a custom interface[12]. Both applications work with the SHOUTcast online radio station directory and, therefore, are able to browse, search and play more than 50'000 radio stations [16].

Pandora is the most popular radio streaming application for the iOS platform[13]. Pandora generates a personalized stream for every listener, and is in this sense not comparable to the other radio streaming applications, even though most of the streaming considerations still apply. iheartradio focuses on the US market with a custom listing of 750 radio stations of all genres[14].

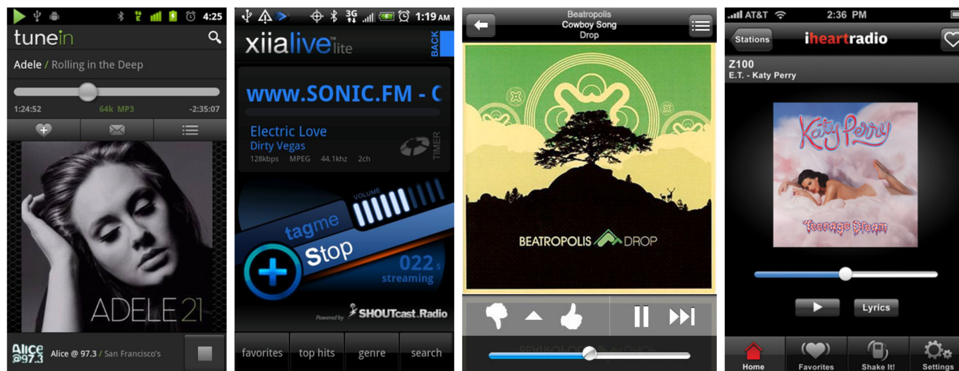

Figure 2.2: Mobile radio streaming applications. From left to right: TuneIn Radio (Android), XiiaLive (Android), Pandora (iPhone), iheartradio (iPhone).

One can see several trends with popular radio streaming applications. They either provide a personalized stream like Pandora, or a directory of radio stations that allows the user to browse and find new stations. Also to note is that the user interface seems to be crucial for achieving widespread adoption, resulting in beautifully designed applications (see Figure 2.2).

[11] TuneIn Radio: https://market.android.com/details?id=tunein.player
[12] XiiaLive: https://market.android.com/details?id=com.android.DroidLivePlayer
[13] Pandora: http://itunes.apple.com/us/app/pandora-radio/id284035177
[14] iheartradio: http://itunes.apple.com/us/app/iheartradio/id290638154

2.3 Automatic Language Conversion

During the implementation phase of this thesis, the need arose to convert a large codebase from Java to C++ due to performance reasons. Doing this work by hand would be an enormous task, especially maintaining a port alongside the upstream development version. Therefore, we investigated frameworks that can translate Java code to native code automatically. Table 2.3 lists some projects in this area.

Converter	Publisher	License	Shortcomings
Toba	University of Arizona	Free	Unmaintained, only JDK 1.1
Java2C	Vishia	LGPL	Alpha state
GCJ	Free Software Foundation	GPL	Difficult to integrate custom C++
JFE	Edison Design Group	Commercial	Expensive, unmaintainable C code

Table 2.3: Automatic programming language converters.

One of the first projects in this domain was Toba[15]. It starts from Java bytecode and translates it to C header and source files. The generated code is not meant for human readability, but rather for optimizing compilers. Furthermore, the project is unmaintained and only supports Java 1.1.

The Java2C project is a recent effort to create a similar solution, released by the author as an open source project [20]. In contrast to the Toba project, Java2C starts from the Java source code and thus generates a human readable output. However, Java2C is currently only in alpha state and not yet tested with large codebases.

There is also a Java frontend for the GCC compiler called GCJ[16]. It is available under an open source license. The main problem with GCJ is that it produces binaries directly, meaning that integrating custom C++ components is difficult.

The most promising solution for our application is JFE from the Edison Design Group[17]. It builds an intermediate representation from Java code, allows performing transformations, and has C and C++ generation backends. It is also widely used in the industry, for example by ARM Ltd., Hewlett-Packard or Texas Instruments. Unfortunately, the JFE is not intended to produce maintainable C or C++ code, and the licensing costs for the technology are between 40'000 and 250'000 dollars.

[15] Toba: http://www.cs.arizona.edu/projects/sumatra/toba/
[16] GCJ: http://gcc.gnu.org/java/
[17] Edison Design Group: http://www.edg.com

3
Content Distribution for Mobile Devices

There are many possible ways of implementing a content distribution system for mobile devices. This chapter describes the requirements we identified for our streaming solution, and the concepts and technologies used to implement them. It also presents a prototype application that was developed to gain insights into the main problems that have to be solved, especially when dealing with mobile devices.

3.1 Requirements

We start by defining the essential requirements for our streaming solution.

Scalability In order to satisfy the growing demand for streaming content on the internet, our solution should scale well with the number of listeners. This means, in particular, that the server infrastructure must support thousands of listeners with modest hardware requirements.

Reliability Like every infrastructure technology, a content distribution system must operate with high availability. In an environment with lots of mobile users, the system should be resilient to listeners joining and leaving in short intervals, different bandwidth limitations due to wireless connections, high mobility of clients, and heterogeneous client devices.

Resource Usage Conserving device resources is particularly important on mobile devices. CPU and RAM usage significantly impacts the battery life and should therefore be reduced wherever possible.

3.2 Content Distribution Network

In order to achieve the necessary scalability and reliability, we chose to use a peer-to-peer content distribution network, namely *Pulsar*. Pulsar is an efficient and robust peer-to-peer streaming system built by the Distributed Computing Group at ETH Zürich [13]. It allows the delivery of audio and video streams to a large number of clients, while keeping the load on the servers low by letting the clients retransmit data to other peers in an elaborate way. It thus scales better than conventional systems for a large number of users while still offering high-quality media streams.

Pulsar was developed for desktop systems. Of course, mobile devices have quite different requirements for a streaming solution. Therefore, it was necessary to adapt and optimize parts of the Pulsar framework to reduce resource consumption and make it usable on resource-constrained devices.

3.3 Target Platform

Android was chosen as the target platform for the development. With a market share of more than 40 percent of smartphone sales in 2011, Android is distributed on a lot of new mobile devices and the user base is growing rapidly [9]. Besides that, the Android platform is an appealing target for the following reasons:

- Android is an open-source platform[1]. There are free development tools and a lot of resources available.
- Android enables the implementation of applications in the Java programming language. This is an advantage because the Pulsar framework is also written in Java, simplifying the initial porting effort.
- There are many different devices available to test against. This is not only limited to smartphones, but the same applications can also run on tablet devices and netbooks.

3.4 Application Prototype

As a first step, we implemented a prototype application for the Android platform which is based on a straightforward port of the desktop version of the Pulsar framework. A background service connects to Pulsar streams, and a simple user interface displays metadata and some controls (Figure 3.1). The notable conclusions were:

- The application successfully received a Pulsar stream and played it back on an Android smartphone.

[1] Android Open Source Project: http://source.android.com/

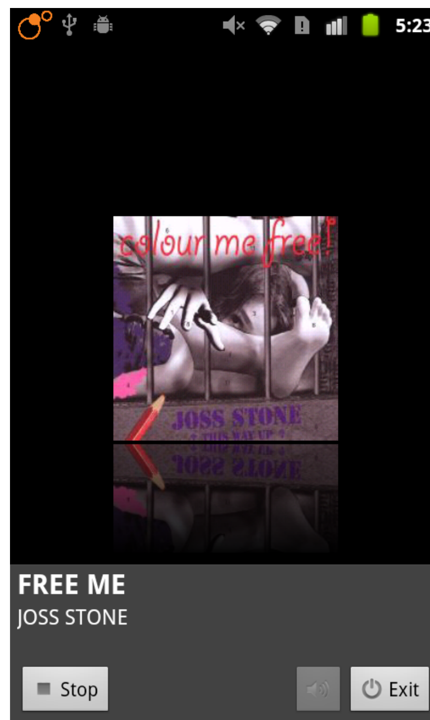

Figure 3.1: The user interface of the application prototype.

- It is currently not possible to integrate a custom streaming protocol into the default media player application.
- The high-level media player component that is available for application developers is not capable of using a custom streaming protocol. Unfortunately, this means that we can not utilize the integrated audio codecs.
- The available sound APIs are limited. They make it easy to play back local files or a stream in a format supported by the platform. Otherwise, audio data has to be decoded by the application and delivered to the system in raw PCM frames.

The major problem with the application prototype was performance. Even with all security mechanisms disabled, no audio decoding and no audio output, the application still used more than 30 percent of the CPU on a Nexus One smartphone. On a Motorola Defy, which runs the older version 2.1 of the Android platform, the application used 100 percent of the CPU. Of course, that is unacceptable for a production application.

The performance problems led us to the decision of implementing the Pulsar framework in native code. Compiling a C++ codebase with an optimizing compiler beforehand should avoid some performance problems that turned up with the Dalvik virtual machine (Dalvik VM)[2] used on the Android platform. This also makes it possible to run the same code on devices that do not support Java in the first place.

[2] Dalvik VM: http://code.google.com/p/dalvik/

Manually implementing Pulsar in C++ is a lot of work and has the disadvantage that a port is difficult to maintain because upstream changes have to be adapted in the port as well. It was therefore decided to create a tool that can convert the Pulsar codebase from Java to C++ automatically (see Chapter 4).

4

Java to C++ Converter

Converting between programming languages is an inherently difficult problem. Tedious to do by hand, it is even harder to get right for a computer program. This chapter presents our new converter capable of translating Java source code to C++. It is built upon well-known, freely available components where it makes sense, but provides superior functionality compared to similar products in areas such as readability of the generated code, performance, or the broad set of supported Java features like threading and synchronization, exception handling, UDP networking, and generics.

4.1 Requirements and Scope

We first define some requirements for our converter.

- The converter shall produce semantically correct C++ source and header files. The generated codebase can be compiled with the GCC compiler [7].
- The generated code shall be human-readable and semantically as close as possible to the original Java code.
- The converter shall support all common Java language features. If something can not be converted due to technical reasons, the converter shall display an appropriate error or warning message.

Furthermore, there are some additional guidelines that we followed during development. For obvious reasons, the generated code should be easy to debug. This is helpful to find problems in the generated code as well as in the converter itself. Class, field and method names are preserved, and the filenames clearly indicate the classes they contain. The converter also keeps Javadoc comments intact.

Another area of concern is portability. High-level libraries and platform-specific APIs such as the Standard Template Library (STL)[1], Boost[2] or the Winsock API[3] are avoided due to our use of the generated code on mobile device platforms. Because these mobile device platforms often use older compilers and libraries, the generated source code also can not take advantage of the new C++11 features[4].

Translating the entire feature set of the Java language and all its libraries is out of scope for this thesis. The following areas were intentionally not investigated:

- There is no support for `NullPointerExceptions`. The translated code will generate a segmentation fault in this case.
- Class nesting is only allowed for a single level. This means that nested classes are allowed, but the use of an inner class or an anonymous class within a nested class is not.
- There are some limitations to the use of generics. See Section 4.4.8 for more details.
- Only parts of the Java standard libraries are included. Notably missing are the filesystem API, regular expressions, the TCP network stack, serialization, and object streams.
- We did not attempt to integrate or translate a user interface framework such as SWT[5] or Swing[6].

4.2 Architecture

Figure 4.1 shows the overall architecture of the converter. It consists of three main parts: the converter program itself, a backend and the runtime implementation.

The converter program generates the C++ source files from the original Java files in five steps. First, the *analyzer* checks the Java code for compatibility and potential translation problems. The *parser* then creates an abstract syntax tree (AST) intermediate representation of the code. The AST is a tree datastructure of all constructs in the original Java code. Multiple *rewriters* transform the AST into a suitable form for C++. The *optimizer* step can do some additional performance optimizations on the intermediate representation. In the end, the *flattener* traverses the AST and produces C++ header and source files accordingly.

There are currently two *backends* that process the generated C++ files. The GCC compiler backend generates shared libraries and binaries. The Android export backend creates suitable build files that can then be compiled with the Android toolchain.

[1] STL: http://www.sgi.com/tech/stl/
[2] Boost: http://www.boost.org/
[3] Winsock: http://technet.microsoft.com/en-us/library/cc958787.aspx
[4] ISO/IEC 14882:2011 on http://www.iso.org/
[5] SWT: http://www.eclipse.org/swt/
[6] Swing: http://download.oracle.com/javase/tutorial/uiswing/

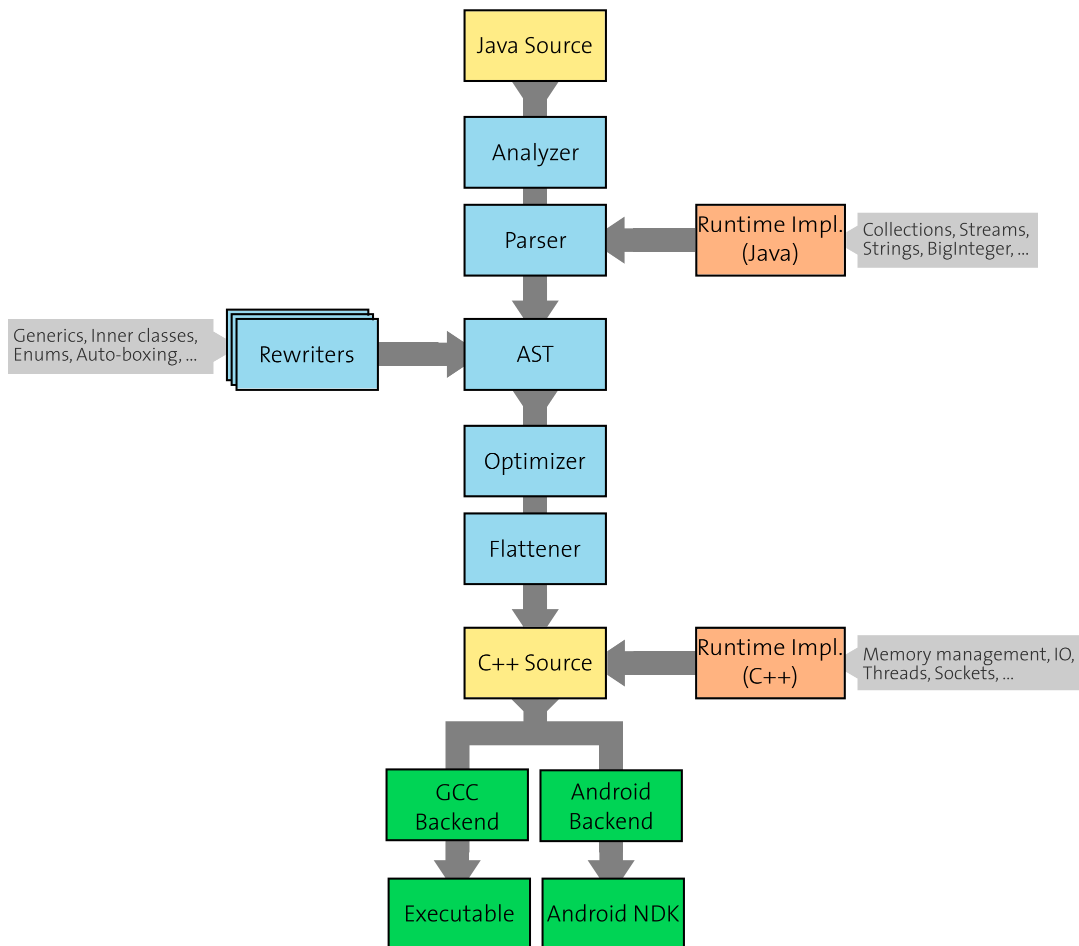

Figure 4.1: Java to C++ converter architecture.

The *runtime implementation* contains the Java libraries and all the code that is required to support the Java language features. The biggest part is contained in the Java runtime and gets converted to C++ together with the original Java source code. Some advanced features such as memory management, threads, and network sockets are implemented in C++ directly and added to the C++ source code before the backend step.

4.2.1 Analyzer

The analyzer is the first step in the translation process. It makes sure that the Java code can be compiled. Additionally, it detects code that can not be converted or that could lead to runtime problems in the generated code:

Missing Symbols If a required Java class can not be accessed by the converter, there are missing symbol errors. This is usually a problem in the input configuration.

Nested Classes A nested class must not contain another inner class or an anonymous class, because this structure can not properly be converted to C++.

Static Initialization Unlike in Java, the order in which static fields are initialized is undefined for C++. Therefore, static fields referencing members or functions of other classes lead to warnings.

Regular Expressions Methods that take string patterns as arguments[7] only work for constant strings and not for regular expressions. If the analyzer can not resolve such arguments to a constant string, or the argument is a regular expression, a warning indicates possible runtime problems.

4.2.2 Parser

A central part of the converter are the Java parser and the AST. Instead of implementing these components ourselves, we used the ones provided by the Java Development Tools (JDT) from Eclipse [23].

The purpose of the parser is to process a Java input file and create a corresponding AST. Figure 4.2 demonstrates the structure of an AST created for a simple program. In the rewriter and the flattener steps, the AST is traversed by classes implementing the visitor pattern [8].

[7] For example `String#split()`.

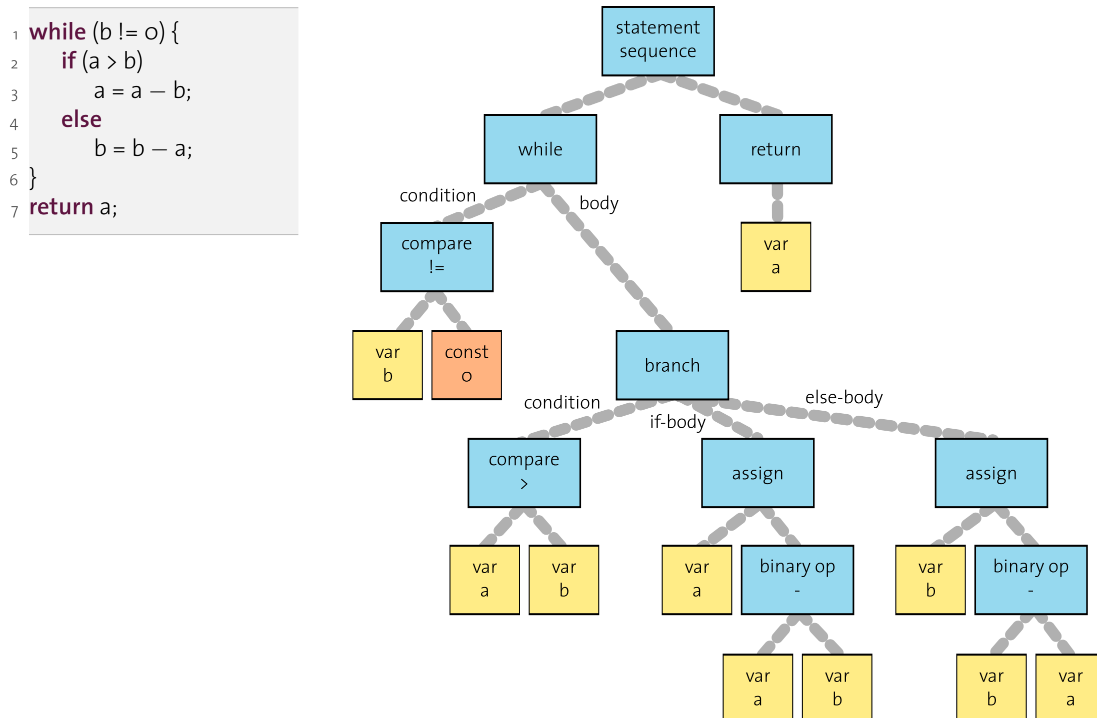

Figure 4.2: Abstract syntax tree (AST). Left: Java source code. Right: Visualization of the corresponding AST datastructure.

4.2.3 Rewriters

To suitably transform the Java code for C++ code generation, several rewriters selectively replace or alter nodes in the Java AST. They fulfill the following purposes:

- Replace expressions not supported in C++ with equivalent statements. For example, a `foreach` loop is transformed to a loop over an iterator, or a string concatenation expression is replaced by a helper method that serves the same purpose (see Sections 4.3.5 and 4.4.5).
- Add language features provided by the Java virtual machine (JVM) explicitly to the code. This includes things like enumeration methods, autoboxing of basic types or accessing the enclosing object from an inner class (see Sections 4.3.4 and 4.4.1).
- Accord for subtle differences in syntax. Besides others, C++ has different suffixes for number literals and does not allow naming methods with a language keyword.

4.2.4 Optimizer

The exact workings of the optimizer step are omitted here because they are not relevant for the topic of the thesis, but Table 4.1 lists some of the operations we do to further optimize the produced code. These operations prepare the code for optimizations by the C++ compiler, for example by simplifying the inheritance hierarchy, making methods non-virtual, or reducing visibilities.

Optimizer	Description
Dead code	Remove unreachable members, methods and classes.
Final Methods	Make methods that are not overridden by any subclass final.
Final Classes	Make classes that have no subclasses final.
Non-Object	Remove the `Object` supertype if it is not needed.
Interface	Remove interfaces that are implemented only by a single class.
Visibility	Reduce the visibility of members and methods.

Table 4.1: Automatic performance optimizations done in the optimizer step.

4.2.5 Flattener

The flattener is responsible for producing C++ source code from the Java AST. It is based on the Java flattener from JDT [23]. In general, each block in the AST corresponds to a block in C++, and each statement in the AST corresponds to a C++ statement. Statements that are not supported in C++ are removed or replaced by the rewriters in a previous step (see Section 4.2.3).

The flattener generates C++ header and source files for every top-level class and enumeration in the AST. The basic syntax of Java and C++ is fortunately quite similar. Thus, most of the simple statements like `if/else`, `for`, `switch/case`, `try/catch`, `new`, and many more can be kept with minimal modifications to the original Java code.

Most of the complexity in the flattener originates from the fact that we use a Java AST as the intermediate representation. This choice allows us to reuse the JDT components from Eclipse, but has the disadvantage that there is no representation for C++-specific language features. As such, the flattener is responsible for adding these on the fly.[8]

Another difficulty is determining which parts of the code have to go into the header or the source files. By default, all declarations are in the header file, and all definitions are in the source file. However, there are cases were the definitions also need to move to the header file, notably inner classes, generic classes and generic methods (see Section 4.4.8).

[8] For example, member initialization lists for constructors are generated in the flattener (see Section 4.4.7).

4.2.6 Backend

The *GCC backend* uses the tools of the GNU Compiler Collection (GCC) to create static libraries and executables on desktop platforms. In a first step, every source file is compiled on its own by invoking GCC. After that, the backend either creates a static library by adding all object files, or a distributable binary by additionally invoking the linker.

The *Android backend* exports the source files into a directory tree that can be built with the Android Native Development Kit (NDK)[9]. It copies the source and header files into appropriate directories and creates buildfiles that are understood by the NDK build system. To make use of the resulting code within an Android application, one still has to include this directory tree in the application folder and invoke the `ndk-build` tool included with the NDK.

4.2.7 Runtime Implementation

The runtime implementation contains all code that is required to support the Java language features as well as implementations for Java libraries. Table 4.2 gives an overview of the contents in the runtime implementation.

Component	Contents	Package	Language	Dependencies
Basic Types	Integer, Long, Double, Short, …	java.lang	Java	
Exceptions	Exception, RuntimeException	java.lang	Java, C++	
Strings	String, Character, StringBuilder	java.lang	Java, C++	
Object	wait(), notify(), synchronized()	java.lang	Java, C++	pthreads
Threads	Thread, Runnable	java.lang	Java, C++	pthreads
Math	round(), log(), sqrt()	java.lang	Java, C++	libm
System	System.out, arraycopy()	java.lang	Java, C++	stdio
IO Streams	InputStream, OutputStream, …	java.io	Java	
Collections	ArrayList, HashMap, HashSet, …	java.util	Java	
Network	InetAddress, DatagramSocket	java.net	Java	
BigInteger	BigInteger	java.math	Java	
NIO	ByteBuffer	java.nio	Java	
Sync	synchronized		C++	pthreads
TCF	try-catch-finally		C++	
GC	Garbage Collection		C++	Boehm GC
UDP	DNS resolving, UDP sockets		C++	BSD sockets

Table 4.2: Contents of the runtime implementation.

There are two possibilities to implement the Java libraries for the runtime implementation. One could wrap C++ libraries, such as the STL, to provide the same interface as Java. This approach promises the best performance but has several drawbacks. The wrapper classes would need to be implemented by hand, which is a lot of work for all the interfaces required by the Java libraries. It would also yield some dependencies on

[9] Android NDK: http://developer.android.com/sdk/ndk/index.html

third-party libraries that may not be available or not properly supported on all target platforms.

The approach we chose is thus to translate as much as possible from a Java implementation. Fortunately, open source implementations of the Java libraries exist. One of them is Apache Harmony[10], a well-established Java runtime that is also used in the Dalvik VM on Android. It is therefore a good choice as the foundation for our runtime implementation.

However, not all parts can be implemented in Java. Components like memory management, threads, and network sockets have to be implemented in C++ directly. The question arises how to interface the code translated from Java with the C++ parts. The Java Native Interface (JNI)[11] is a standard to interface Java with native code. JNI allows Java methods to be declared as `native`, meaning they are implemented in C or C++. Together with a C library that provides access to fields and method calls on objects in the JVM, this allows including native code in a Java program or library. Our converter supports the JNI syntax by acting as a bridge between the generated C++ codebase and the manually implemented C++ components. This permits us to convert Java code that uses JNI, and thus to develop parts of the runtime implementation in C++.

4.3 Data Types

4.3.1 Basic Types

Java has both primitive types and wrapper objects for basic types. This means an integer can be represented as an `int`, carrying just the value, or as a full-fledged `Integer` object that provides additional methods such as conversion from or to strings, and conversion between different number types[12].

Table 4.3 shows the data types that are used for the Java primitives in C++. They are chosen so that the number of bits are the same. As such, operations on primitive types can be copied from Java directly, including bit-operators like shift and xor. We adapted the wrapper objects from Apache Harmony with a custom implementation of the native code accessed through JNI[13].

4.3.2 Object Types

There are two possibilities how to store objects during the runtime of a program: on the heap or the stack. Allocating objects on the heap usually has some overhead because used and unused memory regions have to be managed by the operating system.

[10] Apache Harmony: http://harmony.apache.org/
[11] Java Native Interface: http://download.oracle.com/javase/6/docs/technotes/guides/jni/index.html
[12] For example converting an integer to a double.
[13] We do the conversion of numbers from and to strings through standard C++ methods, whereas Apache Harmony uses an additional floating point and math library.

Java Type	C++ Type	C++ Type Alias	Bits	Range
boolean	bool	jboolean	Java 1, C++ 8	true, false
byte	int8_t	jbyte	8	-128...127
short	int16_t	jshort	16	-32 768 ...32 767
int	int32_t	jint	32	$-2^{31} \ldots 2^{31}-1$
long	int64_t	jlong	64	$-2^{63} \ldots 2^{63}-1$
float	float	jfloat	32	-
double	double	jdouble	64	-
char	uint16_t	jchar	16	\u0000...\uffff

Table 4.3: Basic types in Java and C++. In the generated C++ code, a type alias is used to further unify the look of the Java and C++ code.

In Java, objects are always stored on the heap and accessed through references. C++ allows storing objects on the heap or on the stack, and they can be accessed through either pointers or references. The converter keeps all objects on the heap like it is done in Java. This has several advantages, the main one being the simplicity of the generated code as all objects are stored and accessed the same way. Java addresses objects by reference. Despite the name, Java references are more like pointers in C++[14]. Therefore, Java references are translated to pointers.

4.3.3 Arrays

Arrays can be allocated for all primitive and object types in Java. They are instantiated with a certain size and can not be resized later. This can nicely be mapped to C++ arrays, which behave the same way.

The JVM allows getting the length of an array during runtime. Unfortunately, that is not possible with C++ arrays, so we had to implement it ourselves. Our solution allocates an additional integer to store the length in 4 bytes before every array (see Figure 4.3). On 64-bit systems, four bytes of additional padding have to be inserted before the actual content of the array to fulfill pointer alignment rules[15].

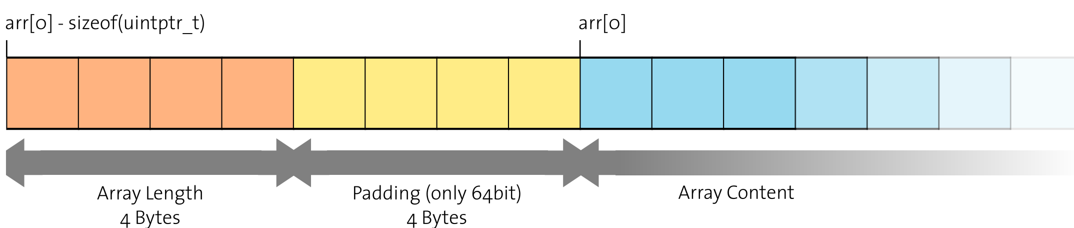

Figure 4.3: How arrays are stored in memory. On a 64bit system, 8 bytes are allocated in front of each array to hold the length (4 bytes) and some padding.

[14] A C++ reference can only point to a single object for its lifetime, whereas pointers can point to an arbitrary object like a Java reference.

[15] Pointers have to be aligned to the size of a pointer. On a 64bit-system, the size of a pointer is 8 bytes and the size of an integer is only 4 bytes, so the padding is required to align the array content.

4.3.4 Enumerations

Enumerations are considered to be objects in Java. They implicitly inherit methods from the `Enumeration` class. In contrast, C++ enumerations are primitives. To support Java-style enumerations, there are two possibilities. Either keep using C++ enumerations and provide the additional methods in a helper object, or skip them altogether and build our own enumeration solution. In this case, we took the second approach because it is closer to the Java model and leads to more readable code. Listing 4.1 demonstrates how an enumeration is converted to a class with the enumeration methods.

4.3.5 Strings

Lots of methods exist to represent strings in C++. We need the following functionality to translate strings from Java:

- Constructing a string from a literal in the source code.
- Constructing a string from an array of bytes as it is produced by serializing a string in the JVM[16].
- String manipulations, for example substrings or converting to lower case.
- String search, for example checking if a string contains another string.
- String concatenation.
- Writing a string to the standard output.

One possibility is to wrap an existing third-party string library to provide the same API as in Java. STL or Boost strings would both work for this approach, but they would require manual wrapper classes for `String` and `Character`, as well as Unicode support. Additionally, it is cumbersome to achieve compatibility with strings serialized by the JVM.

Therefore, our solution uses a C++ part that represents characters as UTF8-encoded `shorts`, and can convert from and to C-style strings[17]. Support for Unicode, string manipulations, search, and concatenation use the translated Java implementation from Apache Harmony. Writing to the standard output and handling string literals in the source code is done through C-style strings.

[16] Compatibility with Java clients, for example when communicating over the network.

[17] 0-terminated character arrays, stored as `char*`.

```java
// provided by the JVM
public abstract class Enum {
    private final String name;
    private final int ordinal;

    protected Enum(String name, int ordinal) {
        this.name = name;
        this.ordinal = ordinal;
    }

    public final String name() {
        return name;
    }

    public final int ordinal() {
        return ordinal;
    }
}

// enum keyword implicitly extends Enum class
public enum TestEnum {
    T1,
    T2;

    public int val() {
        if (this == T1)
            return 1;
        return 2;
    }
}
```

```java
// provided by the runtime implementation
public abstract class CppEnum {
    private final String name;
    private final int ordinal;

    protected CppEnum(
        String name, int ordinal
    ){
        this.name = name;
        this.ordinal = ordinal;
    }

    public String name() {
        return name;
    }

    public final int ordinal() {
        return ordinal;
    }
}

public final class TestEnum extends CppEnum {

    public final static TestEnum T1 =
        new TestEnum("T1", 0);
    public final static TestEnum T2 =
        new TestEnum("T2", 1);

    private final static TestEnum[] VALUES =
        new TestEnum[] { T1, T2 };

    public static TestEnum valueOf(String name) {
        // omitted for brevity: iterate over
        // VALUES, return matching one
    }

    public final static TestEnum[] values() {
        return VALUES;
    }

    TestEnum(String name, int ordinal) {
        super(name, ordinal);
    }

    public int val() {
        if (this == TestEnum.T1)
            return 1;
        return 2;
    }
}
```

Listing 4.1: Converting enumerations. The Java enumeration (left) is converted to a class, and the enumeration methods supported by the JVM are added explicitly (right).

4.4 Language Features

4.4.1 Classes and Interfaces

A Java class contains both declaration and definition, whereas in C++ classes are typically declared in a header file and defined in a source file. We use two separate AST visitors in the flattener to create header and source files. The header visitor inspects the Java AST and creates appropriate declarations, and the source visitor creates the definitions in the source file.

Listing 4.2 shows how a class is converted to C++. In general, the following rules apply to translate classes:

- The class is declared in a header file.
- Class members are declared in the header file. If they are initialized with an expression, this is moved to the member initialization list of the constructor in the source file.
- Static members are declared in the header file. If they are initialized with an expression, the initialization statement is added to the source file.
- Methods, including static ones, are declared in the header file and defined in the source file.
- Abstract methods are declared as pure virtual methods[18] in the header file.
- Native methods are declared in the header file. A wrapper around a call to an external implementation through the JNI naming convention is added as the definition to the source file.

Note that there is no possibility to declare a class as `abstract` in C++. Instead, classes with pure virtual methods or subclasses thereof can not be instantiated unless they implement all pure virtual methods. Furthermore, there is no interface type in C++, so Java interfaces are converted to classes that only contain pure virtual methods.

Nested classes also exist in C++. They are fully defined in the header file and must not contain nested classes themselves. Anonymous classes are assigned a unique name and then added as a separate class to the source file. Generic classes and generic methods are an exception to these rules (see Section 4.4.8).

4.4.2 Inheritance

In Java, a class can only inherit from a single superclass, but can implement multiple interfaces. Since interfaces are represented as regular classes with pure virtual methods in C++, multiple inheritance is required, which is fortunately supported by standard

[18] A pure virtual method has “=0” added to its declaration and does not have a definition in the source file.

```java
public abstract class TestClass {

    private int i1 = 0;
    private int i2;

    private static int si = 1;

    public TestClass() {
        i2 = 2;
    }

    public static int getStatic() {
        return 42;
    }

    public int get() {
        return getNative();
    }

    public abstract int getAbstract();

    public native int getNative();
}
```

```cpp
// testclass.h
class TestClass : virtual public CppObject {
    public:
        TestClass();

        static jint getStatic();
        virtual jint get();
        virtual jint getAbstract() = 0;
        virtual jint getNative();
    private:
        jint i1;
        jint i2;
        static jint si;
};

// testclass.cpp
jint TestClass::si = 1;

TestClass::TestClass() : i1(0) {
    this->i2=2;
}

jint TestClass::getStatic(){
    return 42;
}

jint TestClass::get(){
    return this->getNative();
}

jint TestClass::getNative(){
    return Java_package_TestClass_getNative
            (getCNIEnv(), this);
}
```

Listing 4.2: Converting classes from Java (left) to C++ (right).

C++. A class first inherits its superclass and then all its interfaces. Only public inheritance is used because this is semantically equivalent to Java.

One of the difficulties that arises due to multiple inheritance is the diamond problem [14]. It occurs if a class inherits from the same superclass through more than one inheritance paths[19]. To avoid ambiguities in this case, C++ allows using `virtual` inheritance, which makes sure only a single object of the superclass is allocated and provides facilities for proper casts (see [22] for more details). For simplicity reasons, the converter initially uses virtual inheritance everywhere. To prevent the performance penalty that this imposes, the optimizer step detects and removes most of the unnecessary virtual inheritance relationships (see Section 4.2.4).

[19] For example an interface that is implemented directly and is also implemented by a superclass.

4.4.3 Polymorphism

Polymorphism is a fundamental concept in modern object-oriented programming languages. In Java, polymorphism is used together with inheritance. A variable of a certain type can also hold an object of a subclass of the declared type. Calling a method on this variable will not necessarily call the one from the declared type, but the one of the subclass if it overrides this method.

All method calls in Java use polymorphism. This is not true for C++, where methods have to be explicitly declared as virtual due to the additional overhead at runtime. To preserve semantic compatibility with Java, all non-private methods are initially declared virtual. The optimizer step might identify methods that do not need to be virtual for better performance (see Section 4.2.4).

4.4.4 Packages and Imports

All Java classes declare a `package` statement. Classes can then be referred to by their fully qualified name, which is the package name followed by a period (".") and the class name. An `import` statement takes a fully qualified class name and makes the class available without explicitly specifying the package in the current file. The directory structure of the source files reflects the package naming by creating a directory for every group delimited by a period in the package name[20]. Although this suggests that there is a hierarchy of packages, this is not the case in Java. Semantically, a package can not be embedded inside another one.

C++ namespaces provide a similar functionality as Java packages. They group classes, objects and functions under a common name, essentially dividing the global scope into parts which have to be addressed explicitly with the name of the namespace. Because there is no package hierarchy in Java, it is sufficient to transform every package into its own namespace, replacing periods in the package name with underscores[21]. Like in Java, classes can be referred to by a fully qualified name, which is the namespace followed by the scope resolution operator ("::") and the class name. Similar to the import statement, it is possible to omit the namespace by importing a type with the `using` declaration, or by importing an entire namespace with the `using namespace` directive.

4.4.5 Loops

Loops are semantically very similar in Java and C++. Therefore, `for` and `while` loops can be used the same way they are defined in Java. A special case is the `foreach` loop. It allows iteration over the elements of an array or any iterable collection using a special syntax of the `for` loop[22].

[20] The class org.mycompany.myproject.MyClass resides in src/org/mycompoany/myproject/MyClass.java.
[21] Other naming strategies would be possible as well, although periods are not allowed in a namespace name.
[22] Example: `for (int i: arr) {...}` where arr is an array of integers.

```java
int sum = 0;

// foreach loop for an array
int[] arr = new int[]{1, 2, 3};
for (int i: arr) {
    sum += i;
}

// foreach loop for an iterable collection
List<Integer> list = new ArrayList<Integer>();
list.add(4);
list.add(5);
for (Integer i: list) {
    sum += i;
}
```

```cpp
int sum = 0;

int[] arr = new int[] { 1, 2, 3 };
for (int _i0 = 0; _i0 < arr.length; ++_i0) {
    int i = arr[_i0];
    sum += i;
}

List<Integer> list = new ArrayList<Integer>();
list.add(4);
list.add(5);
Iterator<Integer> _it1 = list.iterator();
while (_it1.hasNext()) {
    Integer i = _it1.next();
    sum += i;
}
```

Listing 4.3: Converting `foreach` loops. In the rewriter step, `foreach` loops (left) are replaced by traditional `for` loops (right).

Because there is no equivalent to the `foreach` loop in C++, all occurrences thereof are replaced by conventional `for` loops in the rewriter step. Iterating over an array uses an additional int variable to store the current position, and iterating over an iterable collection uses an additional iterator variable (see Listing 4.3).

4.4.6 Exceptions

In C++, any object or primitive type may be thrown as an exception. Nevertheless, having a common superclass for all throwable objects, like the `Exception` class in Java, has several advantages. We use the classes from Apache Harmony to express exceptions in the runtime implementation, and subclass them for user-defined exceptions in the original Java code.

Exception handling with the `try-catch` pattern can be done the same way as in Java. However, supporting the `finally` keyword is a difficult problem. C++ does not provide a `finally` construct because it is deemed unnecessary as there is the *resource acquisition is initialization* (RAII) technique to achieve the same effect [21]. Although RAII can be used when developing C++ code from scratch, it is of no help for us when translating from Java. Therefore, we implemented a number of macros that can be used to achieve Java-style finally blocks (see Listing 4.4).

Another non-trivial problem that occurs with exceptions are stack traces. They are extremely useful for debugging purposes, although not supported by standard C++. The GNU libc library[23], commonly used on modern linux desktop distributions, can generate stack traces. Therefore, an optional module that adds stack traces to exceptions was implemented. It is automatically added to the build if the build system detects the

[23] GNU libc library: http://www.gnu.org/s/libc/

```java
boolean b1 = false;
boolean b2 = false;

int i = 0;
try {
    if (b1)
        throw new
                Exception();
    if (b2)
        return i;
} catch (Exception e) {
    i++;
} finally {
    i += 2;
}

return i;
```

```cpp
jboolean b1=false;
jboolean b2=false;

jint i=0;

// defines what happens after the finally
enum TFR_FUN{NOTHING, RETURN,
      BREAK, CONTINUE};

// thrown to reach the finally block
// without returning afterwards
struct TFR_NR {
    TFR_FUN fun;
    TFR_NR(TFR_FUN f): fun(f) {}
};

// thrown to reach the finally block
// and return the value of v afterwards
struct TFR {
    int value;
    TFR(int v): value(v) {}
};

try {
    try {
        if (b1)
            throw new CppException();
        if (b2)
            throw TFR(i);
        throw TFR_NR(NOTHING);
    } catch (CppException* e) {
        i++;
        throw TFR_NR(NOTHING);
    }
} catch (...) {
    i += 2;
    try {
        throw;
    } catch (TFR const& aResult) {
        return aResult.value;
    } catch (TFR_NR const& aNR) {
    }
}
return i;
```

```cpp
jboolean b1=false;
jboolean b2=false;

jint i=0;
tcf_finally_start
tcf_try {
    if (b1)
        throw new
                CppException();
    if (b2)
        tcf_return(i)
    tcf_try_end
} catch (CppException* e) {
    i++;
    tcf_catch_end
} tcf_finally {
    i += 2;
    tcf_finally_end
}
return i;
```

Listing 4.4: Converting `finally` from Java (left) to C++ (center). On the right the code generated by the converter with some macros to increase readability.

GNU libc library. As such, stack traces are currently only supported on Linux systems, but not on the Microsoft Windows and Android platforms.

4.4.7 Initializers

There are various initializers that deal with setting up objects when they are instantiated. Java allows setting the initial value of a field in the declaration, which is called field initialization. The converter transforms all field initializations into a member initialization list that is added to every constructor of the corresponding class in C++.

Another kind of initializers is used for arrays. Initializing arrays with some elements between curly brackets is supported in both Java and C++. However, due to our custom implementation of arrays (see Section 4.3.3), we need to store the length of the array as well. The approach taken by our converter is to count the number of elements in the flattener, and then invoke the `newArrayFromValues` helper method with both the number of elements and the array elements themselves to create the array with the length information.

To do non-trivial object initialization, Java also allows to define so-called object initializers. An object initializer is a block within curly braces inside a Java class that gets executed by the JVM whenever a new object of that class is created. A static initializer can be defined by adding the `static` keyword before an initializer block. Static initializers are executed when a class is loaded by the JVM, and can be used to initialize static members. To reproduce this behavior in C++, the initializers are converted to methods that are called as field initializers (see Listing 4.5).

```
1  public class Initializers {
2      private int i;
3      private static int si;
4
5      // object initializer
6      // executed when a new object is created.
7      {
8          i = 1;
9      }
10
11     // static intializer.
12     // executed when the class is loaded.
13     static {
14         si = 2;
15     }
16
17 }
```

```
1  public class Initializers {
2      private int i;
3      private static int si;
4
5      // object initializer
6      private boolean __oI = __objectInit();
7      private final boolean __objectInit() {
8          this.i = 1;
9          return true;
10     }
11
12     // static initializer
13     private static boolean __sI = __staticInit();
14     private static final boolean __staticInit() {
15         si = 2;
16         return true;
17     }
18
19 }
```

Listing 4.5: Converting object and static initializers. In the rewriter step, object and static initializers (left) are converted to methods that are called through field initializers (right).

4.4.8 Generics

Java introduced generics as a mean to abstract over types in the JDK 1.5. The most common examples are container types, such as Lists or Maps. The Java compiler implements generics as a front-end conversion called *erasure*. The full details of erasure are beyond the scope of this thesis, but one can think of it as a source-to-source translation that transforms the code to a non-generic version. [5]

Because in Java generics are implemented using erasure, their scope is quite limited. Therefore, it is possible to express most of their functionality with C++ templates. C++ templates differ from generics in that the compiler produces different versions of a class or a function for every type argument used in the program. This requires special attention when translating generics and causes some limitations described below.

Generic classes are converted to template classes in C++. From a compiler's point of view, referring to a template class might require adding a new version of this class if the type argument was not seen before. Therefore, it is necessary to have the full definition of a template class available in the header file. The converter thus converts generic classes to header-only template classes.

Special attention has to be given to static methods. Calling a static method on a generic class is possible in Java without specifying the type parameter, whereas in C++ it is never possible to access a template class without a type parameter. The converter uses a rewriter that creates a non-generic superclass for every generic class and moves all static methods to this new class.

Similar to generic classes, generic methods are converted to template methods in C++. The same argument for the compiler applies – template methods have to be fully defined in the header file. The Java compiler also allows to omit the type arguments for generic methods if they can be deduced from the actual arguments or the required return type. Therefore, all type arguments of generic methods are added explicitly to the source code in the rewriter step.

Nevertheless, there are some limitations to what can be successfully translated to C++. Generic type bounds[24] can not be expressed in C++. The analyzer step issues a warning if it encounters type bounds, and then the converter replaces them by the exact type bound. There is also no equivalent for the wildcard generic type.

4.4.9 Threads and Synchronization

The Java thread and synchronization features need low-level threading primitives such as mutexes and conditions, as well as support for creating, joining and stopping user-level threads. Therefore, it makes sense to incorporate a threading library into the generated codebase. Our solution is based on the *POSIX threads* (pthread) API. It has the

[24] Generic type bounds permit to restrict the allowed types by specifying a supertype or a subtype. For example, an interface `MyInterface<T extends String>` would only accept subtypes of `String` as type arguments.

advantage that it is available on all leading desktop platforms including Linux, Mac OS X, Microsoft Windows[25], as well as on the iOS and Android smartphone platforms.

The implementation of threads is based on the `Thread` and `Runnable` classes from Apache Harmony, and a native C++ component based on the pthread API. The native component sets up new threads, runs them and exits them properly. Additionally, the current thread can be identified, and there is preliminary support to interrupt threads on platforms that support it[26]. The start procedure of a user-created thread is as follows:

1. The `start` method is invoked on a `Thread` object.
2. Control is transferred to the native thread component.
3. A new pthread handle is allocated for the thread.
4. A temporary struct with references to the `Thread` and the pthread handle is created.
5. A new thread is created through the `pthread_create` API call. An internal thread setup function is set as the entry point. The temporary struct with the required references is passed to the thread setup function.

At this point, the thread that called the `start` method returns, and the rest of the setup procedure is run in the new thread. The internal thread setup function in the new thread does the following:

1. Stores a reference to the `Thread` object in thread-local storage, so we can later on find out which `Thread` belongs to which pthread.
2. Sets the pthread handle into a field of the `Thread` object, so we can later on find out which pthread belongs to which `Thread`.
3. Adds itself to the list of active threads.
4. Sets the active flag of the `Thread` object.
5. Calls the `run` method of the `Thread` object.

To achieve synchronization between multiple threads, Java provides the `synchronized` keyword. It is based on a so-called *intrinsic lock* associated with every object. A thread is said to own an intrinsic lock in the time between acquiring and releasing the intrinsic lock of an object. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock. When a thread executes a `synchronized` method it acquires the intrinsic lock for that method's object and releases it again when the method returns or throws an exception. It is also possible to explicitly specify the intrinsic lock to acquire in a synchronized block. [17]

The first step in translating the `synchronized` keyword to C++ is converting all synchronized methods to explicit synchronized blocks. We then need to find an equivalent

[25] Through the Pthreads-w32 open source implementation that is included with the converter.

[26] Interrupting a thread using the Java semantics requires the `pthread_cancel` function, which is not available in the pthread implementation that comes with Android.

primitive to the intrinsic lock. We chose to use a `pthread_mutex`, because it can be configured to behave like the `synchronized` keyword, namely providing mutual exclusion, being reentrant, and blocking on acquire. Therefore, a `pthread_mutex` is added to every object.

Just acquiring the mutex at the beginning and releasing it at the end of a synchronized block, however, is not enough. Statements like `return`, `break`, `continue`, or `throw` can transfer the control flow without releasing the mutex at the end of the synchronized block, preventing other threads from accessing the lock in the future. The solution is to use the Scoped Locking pattern [19]. A helper object is allocated on the stack that acquires the mutex in the constructor and releases it in the destructor. Because the C++ language guarantees that the destructors of all objects on the stack are invoked when they go out of scope, even in the case of exceptions, the mutex is always released exactly as required.

Other fundamental building blocks of concurrent Java applications are the `wait()`, `notfiy()` and `notifyAll()` methods provided by the `Object` class, and thus being available for all objects. They require another pthread primitive for every object: a `pthread_condition`. Threads can wait on this condition, and the condition can signal one or all threads waiting on it. All these operations, exactly like in Java, require the calling thread to own an accompanying mutex, which is already present in our case in the form of the intrinsic lock.

4.4.10 Network Stack

Similar to threading, implementing the network stack requires access to a low-level network communication API. The Berkley sockets API, also known as the BSD sockets API, provides a socket-based interface for inter-process communication that is primarily used for communications over the network. The main advantage of using BSD sockets is portability: They are available on almost all Unix-based operating systems (Mac OS X, Linux, iOS, Android) as well as on Microsoft Windows.

There are two main native components that are needed to support the Apache Harmony networking classes. First, the `InetAddress` family of classes needs DNS resolving for hostnames. This can be achieved with the `getaddrinfo` call of the BSD sockets API. Second, the `DatagramSocket` class needs the ability to send and receive UDP packets through local sockets. The `socket` and `bind` calls are used to set up UDP sockets, and the `sendto` and `recvfrom` calls are used to send and receive packets.

All TCP-related functionality is currently not included, although it could be added without major changes to the converter or the runtime implementation.

4.4.11 Garbage Collection

There is no explicit memory management needed in Java. Objects that are not referenced anymore are automatically deleted by the garbage collector, and associated

memory is freed for later use.

In C++, memory management has to be done by the programmer, meaning that all objects that are created have to be deleted explicitly. Although there is no standard garbage collector, various third-party solutions exist.

We chose to integrate the Boehm garbage collector [4]. It is widely used, notably in other virtual machines such as Mono[27], and available on many platforms. The sources of the Boehm garbage collector are added to the generated source code before the build, and all allocations and frees are redirected to call the respective methods in the garbage collector[28].

[27] The Mono Project: http://www.mono-project.com/
[28] All allocations and explicit deletes have to go through the garbage collector.

5

Adapting Pulsar for Mobile Devices

This chapter presents an extended version of the Pulsar streaming technology that was adapted specifically for mobile devices. Various tweaks were done to the Pulsar framework according to the *scalability*, *reliability*, and *resource usage* requirements (see Chapter 3). They mostly evolve around performance and resource usage constraints that are more important on mobile devices than on desktop systems.

5.1 Player Abstraction

As a first step, the client side player infrastructure was decoupled from the rest of the Pulsar framework. It can be accessed through a new player interface (Player API) that contains only the minimum features needed to interact with the Pulsar framework:

- Join or leave streams.
- Play, pause or stop the playback.
- Register listener objects that are informed when the player state, metadata or the error state changes.

This decoupling allows different implementations of the API. Currently, there is a Java implementation, a C++ implementation based on code generated by our Java to C++ converter, and a SHOUTcast implementation. The player abstraction also facilitates to create different user interfaces on top of it, for example the HTML interface that can be accessed from browsers on desktop systems or the native Android interface.

5.2 Performance

To improve the performance of Pulsar streams, various refinements were done on the Pulsar framework itself. The main goal here was to reduce resource consumption. Starting from measurements done with the YourKit Java Profiler[1] on a Linux desktop, the following areas were investigated and improved where possible:

Reduce memory allocations Because the garbage collector has a significant impact on the resource usage of a mobile device, memory allocations were intentionally reduced. For example, frequently used objects like media packets are now pooled and reused instead of created as throw-away objects.

Reduce wakeups To conserve battery life on mobile devices, it is important to let the CPU sleep as often and as long as possible. Therefore, it is desirable to reduce the number of tasks that are executed in regular intervals, as well as batch executing several ones instead of waking up for every one of them. Achieving this was easier than expected because Pulsar already uses an internal "scheduler" to execute tasks, which was optimized for the use-case on mobile devices.

Performance optimizations Some tweaks were done on hot paths, for example doing expensive method calls outside the audio processing thread[2]. Other performance optimizations include caches for often-accessed objects and avoiding debugging constructs in the release configuration.

5.3 Codec

The desktop version of Pulsar uses the MP3 audio codec. Because the input signal for a stream is often already in the MP3 format, this choice avoids the additional overhead involved with transcoding the signal on the server side (see Section 2.1.1). However, using MP3 on mobile devices has the disadvantage that it does not produce very good results at low bitrates.

Therefore, we decided to add support for the Ogg Vorbis codec to Pulsar. Ogg Vorbis is available as open source software and is patent-free. The `libvorbis` library is used to implement the transcoding component on the server side, which automatically encodes the stream signal into the Ogg Vorbis format.

To decode the Ogg Vorbis format on the client side, there are two different implementations. For desktop systems, the same `libvorbis` library is used for decoding. It uses a lot of floating point operations, which are very optimized on desktop processors with dedicated Floating Point Units (FPU). In contrast, most mobile processors as used in smartphones or tablets do not have dedicated FPUs. On these devices, another decoding component based on the `Tremor` decoder is used[3]. Tremor is an integer-only Ogg

[1] YourKit Java Profiler: http://www.yourkit.com/

[2] The audio output thread should not invoke long-running, blocking operations because it leads to audible stuttering.

[3] Tremor Ogg Vorbis decoder: http://xiph.org/vorbis/

Vorbis decoder that is optimized for ARM devices without FPUs. Because Tremor is also available as open source, it could easily be integrated with the rest of the codebase.

5.4 Multiple Bitrates

In order to reduce the overall amount of data that has to be transmitted over mobile data connections[4], it makes sense to use a lower bitrate for audio content. On desktop systems, where usually more bandwidth and processing resources are available, the audio quality should not be reduced by such a procedure. Therefore, we need a solution that can incorporate high-bitrate desktop clients and low-bitrate mobile clients.

To understand what was finally implemented, we first look at how stream overlays and streams work in Pulsar. A *stream overlay* is used to establish connections between different listeners of the same content. A *stream*, on the other hand, is part of a single stream overlay and transmits data of a certain well-defined type and format. To integrate listeners with different bitrates, we have three possibilities:

- Use a different stream overlay for every available bandwidth. Establishing connections between peers is easy, because all peers need to receive the same audio stream in the end.
- Use a single stream overlay for an input source and create different streams for every available bandwidth. Let low-bandwidth listeners only download the low-bandwidth stream, while high-bandwidth listeners also download and distribute parts of the low-bandwidth stream.
- Use a scalable audio codec. In this scenario, the audio data is split into a base-quality layer and several enhancement layers. Compared to the solution above, the advantage is that high-bandwidth listeners do not have to re-download the low-bandwidth stream, because they already have the base-quality layer necessary to seed the low-bandwidth listeners.

Using different stream overlays for every bandwidth has a serious disadvantage: Low-bandwidth listeners usually also have limited upload capacity, resulting in poor peer-to-peer behaviour and higher load on the servers. Currently, we have no working implementation of a scalable audio codec. Consequently, we implemented the second solution. In every stream overlay, there are two streams for 32kbit and 128kbit Ogg Vorbis audio content[5]. Listeners on desktop computers get the whole 128kbit signal for playback, and the additionally download and distribute part of the 32kbit stream for mobile listeners. Mobile listeners only download and optionally distribute the 32kbit signal.[6]

---

[4] For example Edge, 3G, HSDPA, or 4G.

[5] The bitrates and the audio encodings are configurable. For example, it would be possible to add a 320kbit signal as well.

[6] For mobile listeners, re-uploading into the P2P system can be disabled conditionally, for example by a settings option or depending on the connection type.

5.5 Security

In Pulsar streams, packets are authenticated against the public key of the media publisher. The exact authentication mechanisms are out of scope for this thesis, but they are based on DSA signatures that are verified on the client. Verifying signatures is a performance-intensive operation, especially on mobile devices. For these reasons, we changed the signature algorithm to RSA. RSA signatures are about twice as expensive to compute as DSA, but at least five times faster to verify [25].

6 Radio Streaming Application

On top of the adapted version of Pulsar, we built a feature-complete radio streaming application for the Android platform. It plays both Pulsar and traditional SHOUTcast streams and provides the user with a nice user interface to manage and listen to his favorite radio stations. Although implementing an Android application requires quite a lot of in-depth knowledge about the platform, Android-specific implementation details are omitted here because they are not particularly relevant for the topic of the thesis.

6.1 Architecture

The radio streaming application consists of three main parts: the service, the user interface, and protocol implementations for different streaming protocols (Figure 6.1).

The *protocol implementations* provide uniform access to streaming protocols. Currently, there is a protocol implementation for Pulsar streams and one for SHOUTcast streams (see Sections 6.2 and 6.3).

The *service* encapsulates everything related to radio streaming besides the user interface, such that it can be run in the background even when the application itself is not visible to the user. It contains a central stream player component that uses the individual protocol implementations based on the urls that are passed to it. The user interface can access the player component through the Player API (see Section 5.1). Additionally, the service controls the audio backend and the platform integration.

The *audio backend* is responsible to transfer the decoded audio samples of the stream player to the sound card of the device. Specifically, it uses the `AudioTrack` API of the Android platform, which works across all devices that run at least version 1.5 of Android.

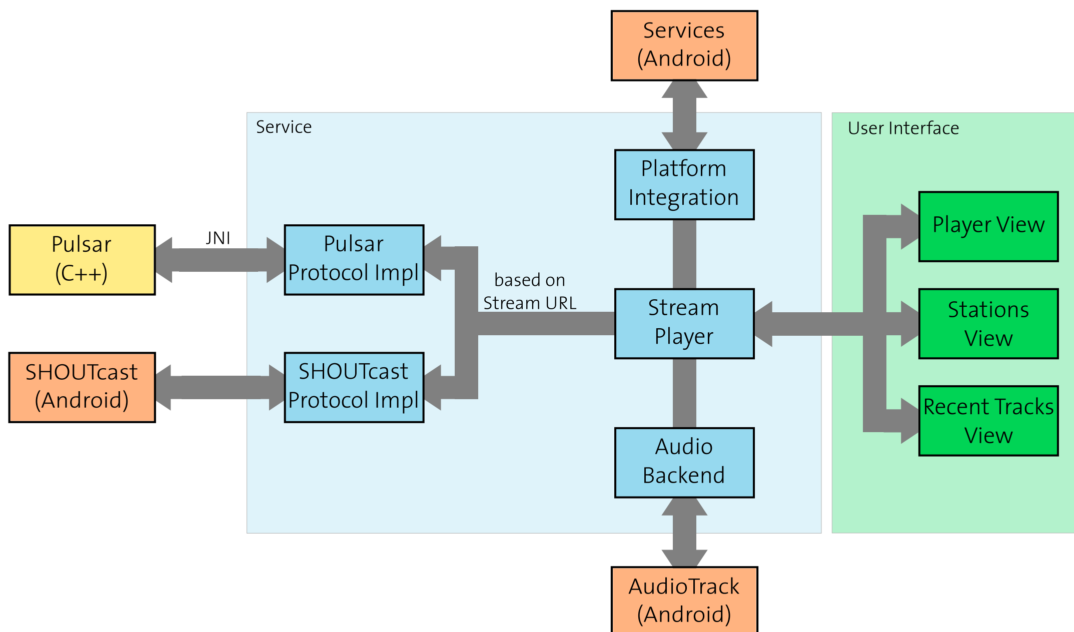

Figure 6.1: Architecture of the radio streaming application.

The *platform integration* component integrates the application with different features provided by the Android platform. It controls the lifecycle of the service[1], makes the application aware of various system properties such as connection or battery state, and advertises its availability to interested applications.

The *user interface* is built on top of the Android GUI components. User-initiated actions are invoked on the underlying service, and different callbacks are registered to update the screen content.[2]

6.2 Pulsar Streams

The support for Pulsar streams is mostly implemented in C++ and added to the application by means of the Android Native Development Kit (NDK)[3]. We use the adapted Pulsar codebase for mobile devices as it is available in Java, and translate all lower level components up to the Player API to C++ using our Java to C++ converter. The generated C++ codebase is then wrapped with a hand-crafted C++ interface.

To make the Pulsar streams accessible from the application, which runs in the Dalvik VM, the protocol implementation for Pulsar streams interacts with the native codebase using JNI. All method calls are forwarded to the native code, and the callbacks invoke the respective methods in the objects living in the Dalvik VM.

[1] Starting up when a user interface connects; making sure the service is not suspended or killed while it is playing music.
[2] For example meta data of the currently playing track.
[3] Android NDK: http://developer.android.com/sdk/ndk/index.html

6.3 SHOUTcast Streams

Providing a protocol implementation for SHOUTcast streams is comparably easy because Android provides a media player component to receive SHOUTcast streams since version 2.2. On earlier versions of the platform, we use a local proxy to convert the SHOUTcast stream to a standard HTTP stream[4]. These shoutcast receivers are then wrapped into a protocol implementation that provides the same API as the Pulsar protocol implementation, so they can be swapped transparently.

6.4 User Interface

The user interface of the application is divided into three tabs, namely the stations view, the player view and the tracks view (see Figure 6.2). These are easily accessible from a unified tab strip along the top, or by swiping horizontally.

The *stations view* shows all the radio streams associated with the application. The *player view* displays the artist and title information of the currently playing track, and fetches associated cover art. Recently played tracks are listed in the *tracks view*.

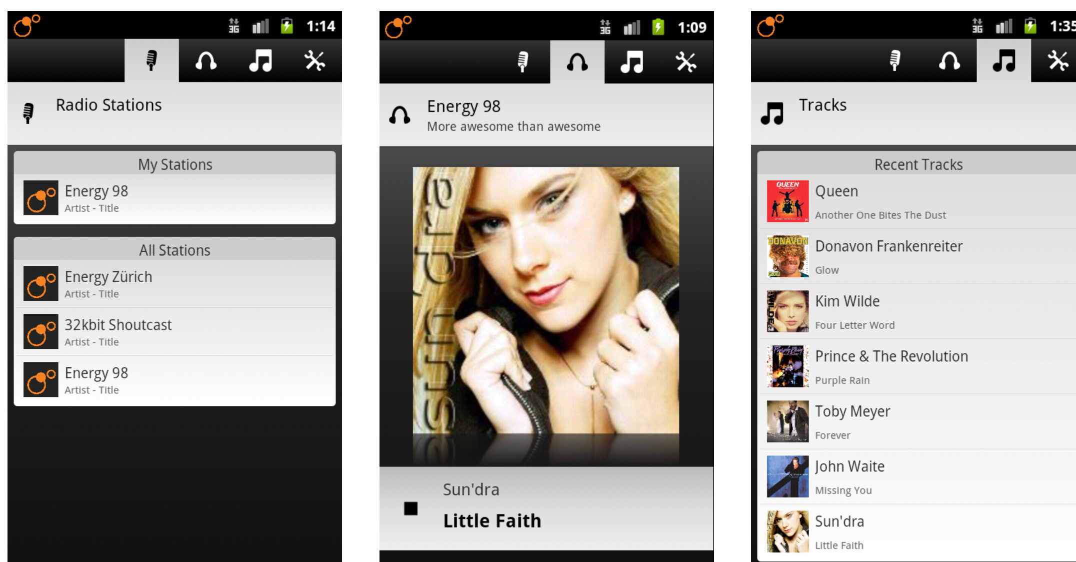

Figure 6.2: User interface of the radio streaming application. Left: The stations view displays the list of radio stations. Center: The player view visualizes the currently playing track. Right: The track view displays a list of recent tracks.

[4] The SHOUTcast to HTTP stream proxy is part of the open-source NPR news application released by Google under the Apache License: http://code.google.com/p/npr-android-app/

7 Evaluation

Evaluating mobile applications is not an easy task – the available tools are often limited. In particular, there is no profiler available that can generate traces for multithreaded native applications on the Android platform. Nevertheless, this chapter provides some insights into the performance of the code generated with our Java to C++ converter as well as the radio streaming application developed for the Android platform.

Device	OS	Kernel	CPU	RAM	Java
Desktop	Kubuntu 11.04	2.6.38	Intel Core i5, 2.67GHz	4Gb	Sun 1.6.0_26
Nexus One	Android 2.3.6	2.6.35	Snapdragon, 1GHz	512Mb	Dalvik 2.3.6
Motorola Defy	Android 2.1	2.6.29	Cortex A8, 800MHz	512Mb	Dalvik 2.1

Table 7.1: The devices used for the evaluation.

Table 7.1 lists the devices used for the evaluation. The desktop system should give an impression of performance on a modern PC. Two smartphones were used to evaluate the system on mobile devices, the Nexus One and the Motorola Defy (see Figure 7.1). The Nexus One smartphone runs the newest Android version available and therefore uses an up-to-date Dalvik VM with a just-in-time (JIT) compiler. On the other hand, the Motorola Defy runs Android version 2.1 which does not yet include the JIT compiler for Dalvik. For this reason, we expect substantial performance differences for Java code running on these devices.

Figure 7.1: The smartphones used for the evaluation: Nexus One (left), Motorola Defy (right).

7.1 Generic Testcases

To get an impression how the code generated by the Java to C++ converter performs on real devices, we developed four generic testcases:

Prime Sieve A simple test that calculates all prime numbers smaller than 10'000'000 using the Sieve of Eratosthenes. It mostly does integer arithmetic and lots of loop iterations.

Array Sort This test sorts an array of 100'000 random integers using the `sort` method of `java.util.Arrays`. The implementation is based on a tuned version of the Quicksort algorithm that iterates over arrays and uses recursion.

ByteBuffer This test allocates a `java.nio.ByteBuffer` for 10'000'000 bytes on the heap. Random bytes are put into the buffer one after the other. It then retrieves all the bytes again and checks that they match the original random source.

BigInteger Verifies the statement of Freeman Dyson[1] up to 2^{50}: the reverse of a power of two is never a power of five[2]. This test uses different arithmetic operations on `java.math.BigInteger` objects.

We implemented the testcases in Java, and then converted them to C++ with our Java to C++ converter. The Java version was run in the Sun JVM on the desktop system and the respective Dalvik VMs on the Android phones. The C++ code was compiled using the GCC compiler with optimizations enabled[3]. Table 7.2 shows the results of running these testcases on the desktop and smartphone devices. The values for a single test-case are measured as follows:

- In the warmup phase, the test is run ten times to warm up the caches and let the JIT of the virtual machines (where available) precompile the code.

[1] Statement of Freeman Dyson: http://www.edge.org/q2005/q05_9.html
[2] For example, 131072 is a power of two, the reverse of it is 270131, which must not be a power of five.
[3] Using the -O3 compiler flag.

- The test is run 100 times. Only the time taken to execute the `run` method is measured, while expensive setup and cleanup operations are done in separate methods for every run.
- The average and standard deviation are calculated afterwards.

		Java		C++	
Device	Testcase	Time (ms)	Std. Dev. (ms)	Time (ms)	Std. Dev. (ms)
Desktop	Prime Sieve	34.86	0.38	39.04	0.54
	Array Sort	9.19	0.77	8.10	0.30
	ByteBuffer	27.43	0.50	128.24	1.34
	BigInteger	2.25	1.13	2.45	0.50
Nexus One	Prime Sieve	955.63	5.36	281.54	8.75
	Array Sort	71.59	2.36	31.45	2.24
	ByteBuffer	3298.90	105.59	918.15	20.34
	BigInteger	128.00	41.27	53.27	2.09
Motorola Defy	Prime Sieve	4538.80	47.36	181.80	10.94
	Array Sort	1401.50	169.20	31.10	0.60
	ByteBuffer	8362.50	279.41	1059.60	49.75
	BigInteger	370.20	123.23	50.30	0.44

Table 7.2: Evaluation results for the generic testcases.

The results of the generic testcases on the desktop system are visualized in Figure 7.2. Except for the ByteBuffer test, the average times per test are about the same. After the warmup period of ten runs, the JVM can execute the tests at about the same speed as the native code generated by the Java to C++ converter. The ByteBuffer test highlights a particular disadvantage of the converter: The `put` and `get` methods of the `ByteBuffer` class are declared `abstract` in Java and implemented by the `HeapByteBuffer` subclass. The converter generates virtual functions in this case (see Section 4.4.3), which are a lot slower to call than statically bound methods.

On the Nexus One smartphone running Android 2.3, the generated C++ code is already a lot faster than the Java code running in the Dalvik VM. Figure 7.3 visualizes the results on this device. All testcases are executed at least two times faster on average. The difference is especially noticeable with the ByteBuffer testcase that does a lot of function calls, where the C++ code performs three times better than the Dalvik VM.

Running the same tests on the Motorola Defy smartphone yields an even bigger difference between the Java and C++ versions. Without the JIT compiler, the Dalvik VM performs much worse than the C++ code in all testcases (see Figure 7.4). The Prime Sieve test, for example, takes 25 times longer to complete in the Dalvik VM than in native code.

To summarize, one can clearly see how the converted codebase yields a better performance for the generic testcases on both smartphones used for the evaluation. On desktop systems the difference is small, with a slight advantage of the JVM.

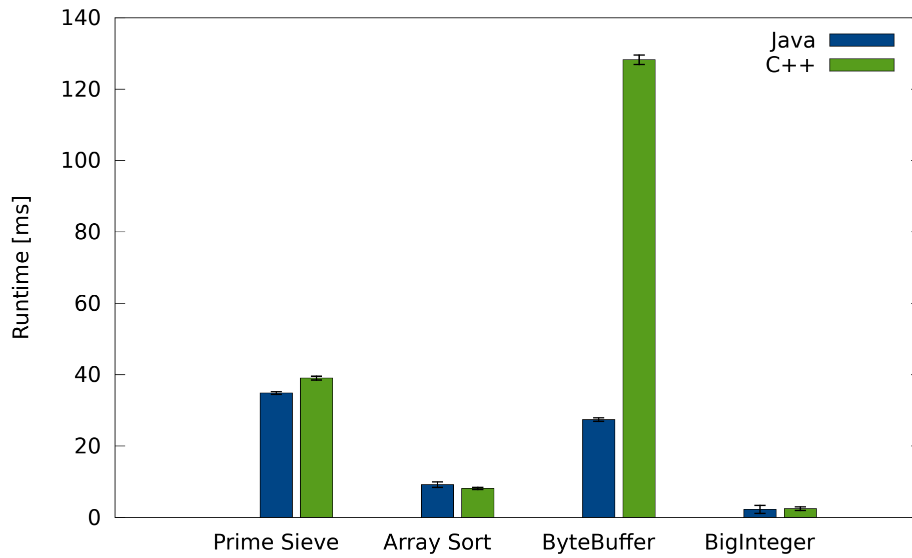

Figure 7.2: Generic testcases on the desktop system.

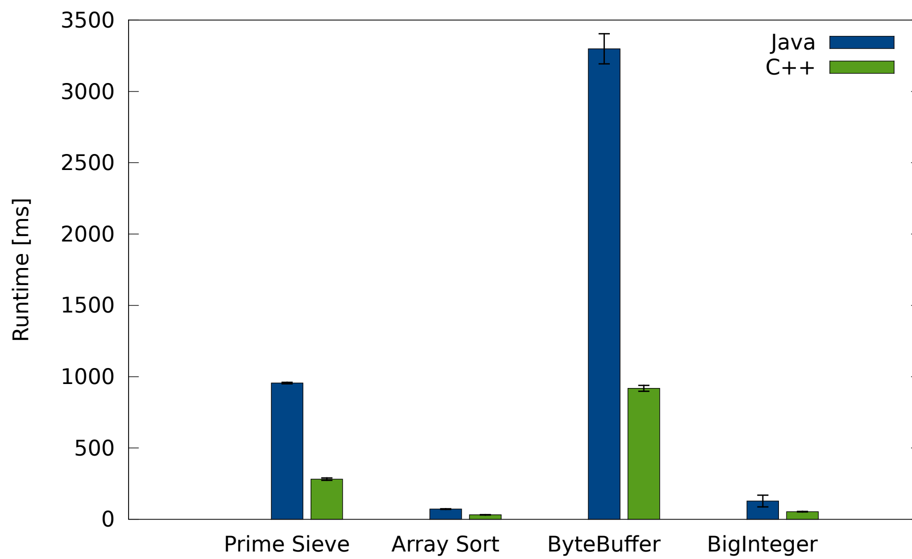

Figure 7.3: Generic testcases on the Nexus One smartphone.

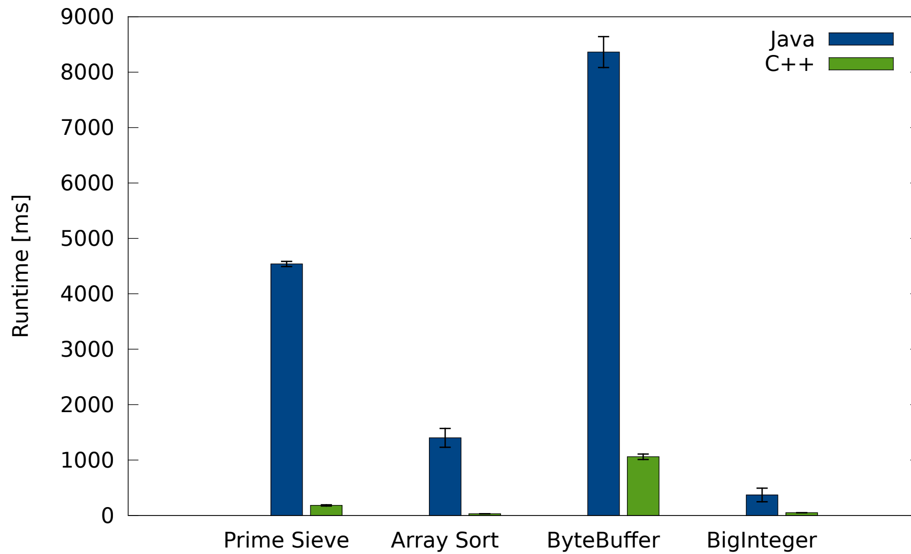

Figure 7.4: Generic testcases on the Motorola Defy smartphone.

7.2 Radio Streaming Application

As seen in the previous section, using the Java to C++ converter on the Android smartphones is appealing due to high potential performance gains. To assess the impact of the conversion to C++ we implemented two versions of the radio streaming application: one based on the original Pulsar framework in Java, and the other one based on the automatically converted codebase. Excluded therefrom is the audio codec, because we use the same Tremor Ogg Vorbis decoder in both applications.

Figure 7.5 compares the CPU usage of these two applications on the Motorola Defy and the Nexus One smartphones. One can clearly see the performance gain achieved by converting the Pulsar framework to C++, although the effect is not as pronounced as with the generic testcases. On the Motorola Defy, the C++ based version brings the CPU usage down from 35.2% to 25.3%. The Nexus One, where the Java code runs in the newer Dalvik VM with the JIT compiler, shows a smaller difference of about 4 percent, from 19.8% down to 15.9%.

In a second step, it makes sense to investigate which components of the system have the biggest impact on the overall performance of the application. Because no suitable profiler is available for native code on the Android platform, the different layers of the Pulsar architecture and the application itself were activated one after another, measuring overall performance with each step. This allows calculating the effect of each component on its own. Table 7.3 shows the results for this evaluation.

Figure 7.6 summarizes the impact of each component on the overall CPU usage. With 6.7%, the Ogg Vorbis codec consumes by far the the most resources, even though we already use a version optimized for mobile devices. The sound output also has a notable impact. This is because we have to transfer the sound samples from the Ogg Vorbis

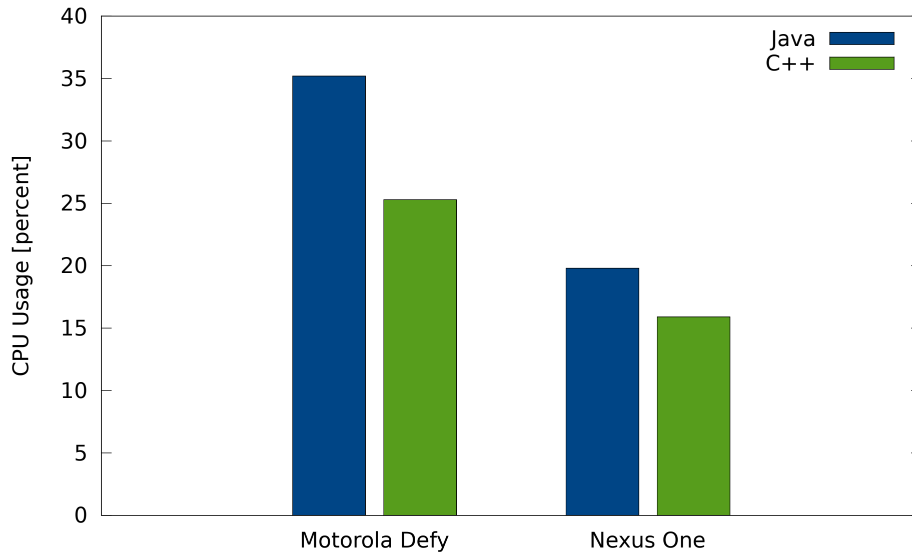

Figure 7.5: CPU usage of the radio streaming application, Java vs. C++.

	CPU (Inclusive)			CPU (Exclusive)	
Component	Total (%)	User (%)	System (%)	Total (%)	User (%)
Protocol	7.1	3.7	3.4	7.1	3.7
Security	7.8	4.3	3.5	0.7	0.6
Playback	8.4	4.6	3.8	0.6	0.3
Codec	14.6	11.3	3.3	6.2	6.7
Sound Output	15.9	13.8	2.1	1.3	2.5
Total	15.9%	13.8%	2.1%	15.9%	13.8%

Table 7.3: CPU usage evaluation results for the radio streaming application on the Nexus One smartphone. The CPU usage is measured every 10 seconds as it is reported by `/proc/cpuinfo`, and averaged over runs of five minutes.

decoder, which runs in native code, back into the Dalvik VM to pass it to the Android system for playback.

Based on the data in Table 7.3, it is possible to further analyze the impact of the Java to C++ converter. Recall that on a Nexus One, the Java version of the application uses 19.8% of the CPU, versus 15.9% with the C++ version. Subtracting the 7.5% caused by the codec and the sound output, which are equal in both scenarios, the C++ version is 32% faster.

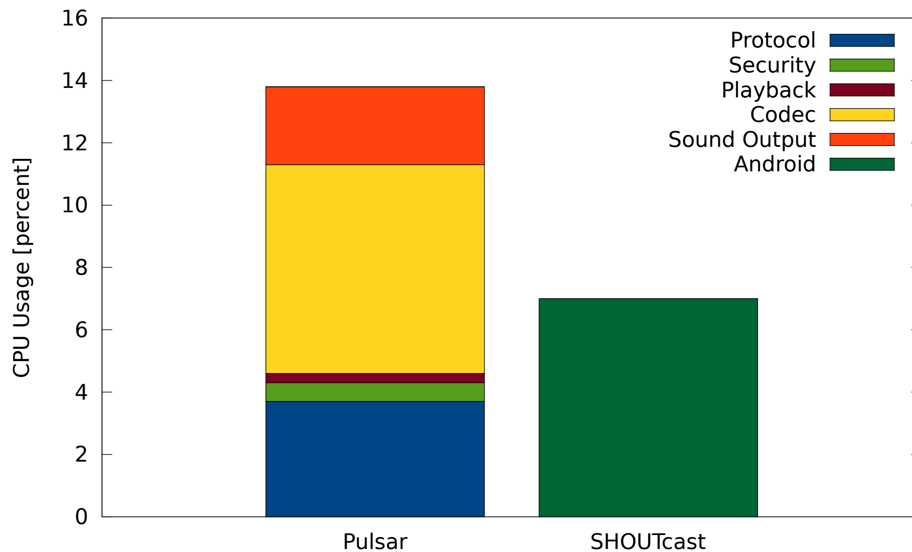

Figure 7.6: CPU usage of the radio streaming application on the Nexus One smartphone, by component. Comparison of Pulsar and SHOUTcast based streaming.

Furthermore, it is interesting to compare the Pulsar-based streaming application to traditional SHOUTcast streaming. Running a SHOUTcast stream in the media player component provided by the Android platform uses around half of the CPU resources of the Pulsar based streaming. We suspect that this speedup is possible because the platform media player uses proprietary, device-specific audio decoders that run in prioritized operating system threads.

8 Conclusion

Within the scope of this thesis, we implemented a feature-complete radio streaming application for the Android platform. Starting with an overview of current media streaming solutions, we defined the basic requirements for a content distribution system for mobile devices. The Pulsar peer-to-peer streaming system was chosen as the foundation for our work, and further adapted for use on mobile devices. To achieve acceptable performance, the Pulsar codebase was translated from Java to C++ automatically with a converter program implemented specifically for this purpose.

8.1 Achievements

In the beginning, we defined three main requirements for a streaming solution tailored to mobile devices: scalability, reliability and low resource usage. The Pulsar peer-to-peer content distribution system provides the necessary scalability and reliability.

Various tweaks and adaptations were implemented on top of Pulsar to make it usable on mobile devices. This includes Ogg Vorbis as the new audio codec, adding a low-bitrate stream, as well as improving the performance of the security mechanism and other parts of the framework.

We developed a tool to convert Java source code to C++ automatically. Compared to similar products, our solution works on the source code level, allowing us to preserve the overall structure and comments, leading to a human-readable port of the original codebase. This is beneficial for various reasons, including easier integration of components written in C++, debugging, and the ability to compile the generated code on various platforms using different compilers and target libraries. Most language features of

Java are supported, including exceptions, generics, threads and synchronization primitives, garbage collection, and UDP networking.

First evaluation results show a lot of potential performance gains when using the converted C++ code on mobile devices. On an Android smartphone, generic computation-intensive testcases run up to 40 times faster in native code than in the Dalvik VM. For the Pulsar codebase, the difference is smaller but still noticeable. Subtracting the CPU usage of the audio codec, which is the same in both the Java and the C++ version, the Pulsar protocol runs about one third faster in native code than in the Dalvik VM.

8.2 Future Work

The presented radio streaming application still has a lot of room for future improvements. One area that we only covered briefly in this thesis is the user interface. For example, it would make sense to include a directory of radio stations, and facilities to browse and search them. One could integrate an Ogg Vorbis decoder optimized for ARM-based devices to further improve the performance. An example for such a decoder is Tremolo[1].

There is a lot of room for more optimizations in the Java to C++ converter. Besides smaller optimizations in the optimizer and flattener steps, the most could probably be gained by using a C++ AST. This would allow us to use C++-specific constructs in the intermediate representation, which would simplify further optimizations across the rewriter, optimizer and flattener steps. Furthermore, the converter could be made more general such that it can be used for other projects. The main point here would be to extend the runtime implementation, for example by including a TCP network stack or filesystem access.

Because the converter already produces portable C++ code, ports of the Pulsar framework to other smartphone platforms should be possible with minimal effort. Preliminary tests showed that the same codebase already compiles and works on Mac OS X, different Linux distributions as well as on Microsoft Windows using the MinGW compiler. In essence, simply adding a user interface and the appropriate packaging should be enough to get a similar streaming application on iOS, Blackberry or Symbian platforms.

Other areas of interest are on-demand streaming, which is already supported by Pulsar on the protocol layer, video streaming, or the integration of a scalable audio codec.

[1] Tremolo Ogg Vorbis Decoder: http://wss.co.uk/pinknoise/tremolo/

Bibliography

[1] Adobe. Real-Time Messaging Protocol (RTMP) specification. http://www.adobe.com/devnet/rtmp.html, 2011.

[2] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys (CSUR)*, 36(4):335–371, 2004.

[3] Apple. HTTP Live Streaming Draft. http://tools.ietf.org/html/draft-pantos-http-live-streaming-06, 2011.

[4] H. Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 197–206. ACM, 1993.

[5] G. Bracha. Generics in the Java programming language. *Sun Microsystems, java.sun.com*, 2004.

[6] K. Brandenburg. MP3 and AAC explained. In *AES 17th International Conference on High-Quality Audio Coding*. Citeseer, 1999.

[7] Free Software Foundation. GCC, the GNU Compiler Collection. http://gcc.gnu.org/, 2011.

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, et al. *Design patterns*, volume 1. Addison-Wesley Reading, MA, 2002.

[9] Gartner. Worldwide smartphone sales (August 2011). http://www.gartner.com/it/page.jsp?id=1764714, 2011.

[10] M. Hofmann and L. R. Beaumont. *Content networking: Achitecture, protocols, and practice*. Morgan Kaufmann Pub, 2005.

[11] IETF Audio-Video Transport Working Group. RTP: A Transport Protocol for Real-Time Applications. http://tools.ietf.org/html/rfc3550, 2011.

[12] B. Krishnamurthy, C. Wills, and Y. Zhang. On the use and performance of content distribution networks. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 169–182. ACM, 2001.

[13] T. Locher, R. Meier, S. Schmid, and R. Wattenhofer. Push-to-Pull Peer-to-Peer Live Streaming. In *Proceedings of 21st International Symposium on Distributed Computing (DISC)*, volume 4731. Springer-Verlag, 2007.

[14] A. Milea. Solving the Diamond Problem with Virtual Inheritance. http://www.cprogramming.com/tutorial/virtual_inheritance.html, 2011.

[15] J. Moffitt. Ogg Vorbis - Open, Free Audio. *Linux Journal*, 2001, January 2001.

[16] Nullsoft. SHOUTcast. http://www.shoutcast.org, 2011.

[17] Oracle: The Java Tutorials. Intrinsic Locks and Synchronization. http://download.oracle.com/javase/tutorial/essential/concurrency/locksync.html, 2011.

[18] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *Technology*, 100, 1984.

[19] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley, 2000.

[20] H. Schorrig and T. Henties. Java2c - developing in java, deployment in c: short paper. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 73–75, New York, NY, USA, 2010. ACM.

[21] B. Stroustrup. Bjarne Stroustrup's C++ Style and Technique FAQ. http://www2.research.att.com/~bs/bs_faq2.html, 2011.

[22] B. Stroustrup and S. T. B. Online. *The C++ programming language*, volume 3. Addison-Wesley Reading, Massachusetts, 1997.

[23] The Eclipse Foundation. Eclipse Java Development Tools (JDT). http://www.eclipse.org/jdt/, 2011.

[24] Wikipedia. Comparison of Audio Codecs. http://en.wikipedia.org/wiki/Comparison_of_audio_codecs, 2011.

[25] C. Wong and S. Lam. Digital signatures for flows and multicasts. *Networking, IEEE/ACM Transactions on*, 7(4):502–513, 1999.