Guidelines for the Implementation of Cross-platform Mobile Middleware

Otso Kassinen, Erkki Harjula, Timo Koskela, Mika Ylianttila

University of Oulu, Oulu, Finland
firstname.lastname@ee.oulu.fi

Abstract

We describe and analyze the experiences from a cross-platform mobile middleware project. Our aim is to identify best practices and provide guidelines for solving similar problems in the context of cross-platform-targeted programming in mobile as well as other environments. The main deliverable of the observed project was a cross-platform implementation of a protocol for maintaining DHT-based peer-to-peer (P2P) networks, with an emphasis on mobile operation. The target platforms of the software were Symbian OS, mobile Maemo Linux, and desktop/server Linux distributions. Considering the large number of incompatible mobile platforms in the market, the fluency of cross-platform mobile software development is of particular importance. The observations in our work include the importance of maintaining the relevant conventions of the various platforms in the code and understanding the asymmetric difficulty in porting code between different platforms. We also determine the amount of cross-platform support related code in an example software module.

Keywords: Cross-platform programming, Linux programming, middleware implementation, mobile networking, Symbian OS programming.

1. Introduction

There are various situations where the source code of a specific software project must be written in such a way that the sources can, with little effort, be compiled to different kinds of executables, each of which will work on a different platform. The aim is to implement identical functionality for a number of systems that are based on different hardware, different operating systems, different system libraries, and so on, without the need to maintain multiple – platform-specific but functionally identical and thus largely redundant – versions of the same codebase in parallel. Source code that meets the demands of being directly compilable for all desired target platforms is called cross-platform code.

It should be noted that cross-platform software could also mean programs whose binary form itself is cross-platform-runnable or programs that require parallel platform-specific versions of their entire source code tree. However, in this article, cross-platform software specifically means programs, which 1) are cross-platform-compatible at the source code level but result in different kinds of binaries for different platforms (as opposed to code that is acceptable on several platforms as a single unmodified binary, e.g. Java bytecode, or as an interpreted program), and 2) are based on a single codebase, in other words, the source files do not exist as parallel versions for each platform.

Cross-platform source code is particularly important in the case of mobile software due to the large variety of different mobile platforms in existence, since no clear winners of the platform war have emerged yet, and even those mobile platforms that have achieved some popularity do have several partially incompatible versions in the wild; thus, a developer who wants to provide applications for as many mobile platforms as possible will probably benefit...
from the use of cross-platform code. It is not uncommon that several source-incompatible
versions of the same mobile platform exist, and essentially those versions must be treated as
different platforms in the development process.

The implementation of software systems where the same codebase must be run on several
heterogeneous platforms often requires special techniques and precautions. The reasons for
this include the large number of different APIs and the continuous change in operating
systems and hardware [1]. Techniques proposed for facilitating cross-platform software
development include, for instance, reflection and platform-independent component
descriptions [1], the use of virtualization for effortless build and testing activities [2], and the
systematic abstraction of the work phases in the build tool-chain [3]. The special requirements
of mobile middleware are also discussed in the literature. In the mobile environment, the
construction of middleware is complicated by limited computational capabilities and
unreliable radio network connections, among other things [4]. Portability and
interoperability across heterogeneous platforms are some of the key requirements in mobile
middleware systems [5].

A great number of cross-platform utility libraries are available for mobile and/or fixed-line
systems – random examples include GTK+, OpenGL, Qt, and Simple DirectMedia Layer
(SDL) – although in the case of many of these pieces of software, the meaning of the word
“cross-platform” might be slightly different from the definition we gave at the beginning of
this article. Tools aiming to be a complete cross-platform development environment include,
among others, Lazarus, REAL Studio, and ZooLib.

Many programming languages provide good cross-platform support in the sense that they
can be compiled for, or interpreted on, several platforms. However, this is not a sufficient
prerequisite for creating cross-platform software in the real world. Platform-specific
properties, such as different application programming interfaces (API) and the differing
restrictions of the environments where the program must operate, must be taken into account.
Key questions include which platform-specific technique to use in a given situation, and how
to make sure that the platform-specific characteristics of the code will not cause problems on
the other platforms and that the code in general will be easy to write and understand.

In this article, we provide guidelines for cross-platform programming by describing the
techniques we used and the observations we made during a mobile peer-to-peer (P2P)
networking middleware project. The middleware is an implementation of Peer-to-Peer
Protocol (P2PP), draft -01. The name of the middleware is Mobile Peer-to-Peer Protocol
(MP2PP) and its functionality is described in [6]. It should be noted that even though P2PP
originates from the P2P SIP working group in the IETF, our MP2PP did not aim only at
providing SIP-like functionality. The main focus was to enable the peers to act as the nodes of
a distributed database, suitable for more generic usage, although we also have implemented a
SIP system on top of this decentralized service-provision layer. The hosts that use MP2PP are
able to participate in a distributed hash table (DHT) based P2P network, which acts as a
database where digital resources and services can be published, searched, and consumed.

The target platforms of the middleware are 1) Maemo Linux, 2) Symbian OS, and 3)
desktop or server Linux distributions, as illustrated in Figure 1. However, because the
codebase in question contains nothing strictly Maemo-specific, in practice our code has two
target platforms: 1) Linux and 2) Symbian OS. Creating middleware for mobile devices was
the main objective in the project; Maemo and Symbian are mobile operating systems. Also
support for the non-mobile platforms was included and tested, which leads to wider
applicability for the software – at a marginal additional cost in the development phase – in
situations such as easy simulations of large-scale DHT overlays on servers or introducing resource-rich special nodes for enhanced service provisioning in an overlay network.

The middleware was implemented in the C++ language. Thus, whenever we refer to “source code” in this article, it should be assumed to be code written in C or C++, unless stated otherwise. Other design choices, such as the libraries used, are described later.

Before moving on, we want to define the target audience of this article, i.e. what is the level of knowledge about the platforms that is required in order to be able to benefit from the guidelines proposed in this article. The guidelines are applicable by any developer who is familiar with the basic concepts of software development. Nevertheless, some of the provided guidelines and examples are most readily understood by developers who have experience in our selected target platforms or have other kinds of experiences from similar situations. Moreover, because no specifically cross-platform oriented toolkits or utilities were used, our guidelines are applicable in a situation where the developer cannot or does not want to use a third-party-provided cross-platform development environment.

The rest of this article is organized as follows: in section 2, we analyze programming practices from the viewpoints of the code’s structure and readability. In section 3, we discuss techniques for accessing the system resources on diverse platforms. In section 4, we analyze the implications on software project management and in section 5, we estimate numerically the effort overhead caused by cross-platform development in our project. In section 6, we conclude the article.

2. Language-internal constructs

Some aspects of the cross-platform compatibility of code are related to the pre-processing and compilation processes and stylistic matters, not directly to the available libraries and other external resources of a program. These properties, which in the absence of a more established term can be called language-internal constructs, often have a significant effect on the functional correctness and readability of the code.

In cross-platform programming, it is not uncommon to be in a situation where the developers cannot absolutely stick to the requirements of a given platform, because the requirements contradict those of another target platform. The developers need to use different adaptation mechanisms to force code that follows the conventions of one platform to be acceptable also under the other platforms.

Often, it is advisable to drop some of the ordinary requirements of some target platforms and adopt only the selected conventions that are suitable for the cross-platform project; examples are provided later in this article. We call this design principle considered omission. It should be noted that here we refer to aspects that are visible to the developer, but not to the end user, because these aspects are not about selecting or omitting different user-level functionalities of the software on different platforms.
It is also possible that a given target platform does not have strong conventions for a specific aspect of the code, but another target platform (or any larger subset of the set of all target platforms) does have such conventions; then, it is in some cases good to let the strong convention prevail in the code, resulting in code that in some ways bears the distinctive characteristics of one target platform – and retains properties that might be essential for the correctness of the code when compiled for that platform – but is acceptable also for the other target platforms. We call this design principle *considered bias*.

2.1. Appearance and readability of the source code

There are often recommendations for the stylistic aspects of source code on different platforms. For example, the rules may suggest certain styles for using braces ({}), indentation, variable names, and whitespace between the specific parts of a statement. If the rules of different target platforms contradict each other, the developers must pick the rules that are most suitable for the project. Although stylistic matters may seem to be relatively unimportant or nearly arbitrary decisions, they can have an effect on the fluency of writing and reading source code.

An example of the adoption of the conventions of one target platform in the cross-platform code in the MP2PP project was the usage of Symbian OS originated syntax for function and variable names. In Symbian OS, the developer must pay a lot of attention to the handling of dynamically allocated memory areas. Unlike other mobile programming environments of the present day, in Symbian OS the code must always be prepared for the so-called *leave* event, which is essentially an exception that is generated by the operating system in an out-of-resources situation, most notably when memory has run out and the code attempts to reserve bytes from the heap. The developer must prepare for the potential leave events by placing the pointers to dynamically allocated memory in the “cleanup stack”.

This had two important corollaries on the cross-platform code of our project. Firstly, we had to follow the practice of postfixing the names of functions, which might leave, with “L”. This is required by the Symbian OS coding conventions, and when we created any new functions in our cross-platform code, we had to consider their behavior from the viewpoint of Symbian OS. Thus, the “L” marks are part of the function names although the postfix is meaningful only for the Symbian OS target platform. Secondly, since the pushing of pointers in the cleanup stack must be explicitly invoked in the code by the developer, we had to select a technique for making the PushL() method calls happen when compiling for Symbian OS while excluding them on the other target platforms. Instead of creating for every platform a set of dummy functions that look like the Symbian OS cleanup stack access functions but do nothing, we decided to define pre-processor macros that expand to cleanup stack push or pull functions when compiling for Symbian OS. This also makes the platform-specific nature of the operations explicit since the macros with uppercase letters are easily distinguished from the rest the code.

We decided to adopt the most useful variable-naming conventions of the Symbian OS coding style: we prefixed function arguments with “a” and objects’ data members with “i”. In addition, we made the role of stack variables explicit by prefixing their names with “t”, which also increased the general consistency of variable naming. These conventions made it easier to spot potential mistakes in the use of variables. The variable-naming conventions, although originating from Symbian OS, were useful for the cross-platform code as they deal with concepts that are equally important on all target platforms: the scope of definition of data items, and so on.
Still another convention that we had to adopt from Symbian OS was the distinction of classes to those that are intended to be heap-instantiated (class names prefixed with “C”) or stack-instantiated (class names prefixed with “T”), as the memory management model in Symbian OS requires this differentiation of classes. Whereas the above-mentioned naming system of variables provided added value also from the viewpoint of the other target platforms, the class-naming scheme was not beneficial for the other platforms but in any case had to be followed.

To improve the consistency of the source code, the developer should always use a fixed order for platform selections when using e.g. conditional pre-processor directives for separating the platform-specific code sections. For example, in our MP2PP code, whenever we needed to select between two target platforms, these pre-processor selection constructs started with #ifdef SYMBIAN, and the Linux-specific section always came after (never before) the Symbian OS specific section.

Another similar habit that makes it easy to glance through the code is to include platform-specific headers in logical groups; this particular technique is of course only a minor improvement, but nevertheless, applying this kind of simple rules on the various elements of source code may enhance the readability of code as a whole and eliminate bugs, because errors are usually easier to notice in a codebase that follows consistent conventions.

Consistency is beneficial also in the comments of the source code. Whenever a comment contains information that is specific to a particular target platform, it is good to tag the comments with the platform’s name (or its abbreviation or any similar marking) in a consistent way; thus, the developers can instantly see that the comment is relevant only when examining the code from the viewpoint of compiling for a particular platform.

The mentioned consistency-rules make it easier to glance over the code in order to obtain a picture about the sequence of execution on a specific platform. Naturally, the required level of consistency and formalism varies according to the size and purpose of the project. For example, in the MP2PP project, the rules about the formal tagging of comments or the grouping of header file names were not enforced, but some developers independently followed similar practices in the code modules they were responsible for.

### 2.2. Special rules in basic utility code or in the compilation process

Another consideration in the MP2PP coding project was the requirement to derive all “C”-prefixed classes – whose naming scheme we mentioned in section 2.1 – from the common base class CBase that is defined in Symbian OS for the purposes of handling heap-instantiated objects in a uniform manner. No similar requirement existed on the other target platforms. The CBase class also has the property that the contents of a CBase object are automatically initialized to binary zero, when such an object is created.

There were at least two alternatives how to implement CBase derivation in the cross-platform code. We could define a dummy CBase class for all the other platforms (compare this solution to the dummy cleanup stack functions mentioned in section 2.1), or we could use pre-processor directives to force CBase derivation to happen only when compiling for Symbian OS. We decided to implement the latter alternative, because it makes the code less bound to Symbian OS conventions as a whole, and the developers do not need to waste their time and effort memorizing the extra rule about zero-initialization that affects all data in CBase-derived classes in the code.

Consequently, we saw that it is a good practice to make all required zero assignments explicit. This enables the use of the same code on any platform – possibly in an entirely different software project – without knowledge about the fact that some special mechanism...
behind the scenes should be magically zero-initializing the contents of certain types of objects. It is much better to see explicitly that a particular variable is intended to be zero-initialized (iVar = 0) than to rely on a special “automatic” technique that might not even be familiar to all developers in the project. Developers should always make explicit those properties in their code that are otherwise prone to introduce logical errors silently, especially in cases where the normal, expectable behavior of code may be altered by obscure platform-specific rules.

It is difficult to keep in mind the rules whose existence is not immediately visible when examining the source code: the CBase-class zero-initialization rule is a clear example of this, as the rule would practically never be known to a cross-platform developer who has not heard about it somewhere – at least until an inexplicable bug demands the developer’s attention and finally the zero-initialization rule is learned the hard way, frustrating the developer who simply did not know about such an arbitrary rule beforehand.

Abstaining from CBase-derivation on all platforms except Symbian OS was an example about avoiding bad conventions of one target platform from spreading to the cross-platform code. The following example is about avoiding a different class of problems: the platform-specific, non-standard behavior of programming language constructs that the developers assume to work in a uniform manner on all target platforms. If the developer does not have intimate knowledge about the internal workings of all target platforms, surprising bugs may be generated.

For example, a developer, who is mostly accustomed to programming in standard-obedient C/C++ language on platforms such as Linux, may run into problems when relying on the proper working of a copy constructor in certain situations on Symbian OS. It is perfectly legal to define a copy constructor in Symbian OS code. However, it will not get called in all situations where a Linux developer would expect it. If there are – as there were in the MP2PP project – objects or structs whose construction requires the implicit calling of the copy constructor in an assignment statement, the call will not happen, because the compiler does not generate the required code in the resulting binary file. Instead, the runtime operation will be such that the data members of the object in question are copied byte-by-byte to the object that is being assigned to. Needless to say, pointer members inside objects are usually not safe to be copied this way. The “wrong” copying operation is invoked silently; the unsuspecting developer will notice the error only when runtime problems occur, probably related to the dereferencing of a bad pointer.

The above-mentioned compiler behavior was observed at least with Symbian OS 9.2, Series 60 3rd Edition Feature Pack 1, used in the MP2PP project. After encountering unexpected behavior in code where there seemed to be nothing unusual, we noticed that the code that pushed a struct containing protocol-message-related data into a Standard Template Library (STL) container (vector) relied on the implicit calling of the copy constructor: the copied object contained pointer members and also internal STL-based linked-list-objects that were not copied properly. The problem was of course fixed by creating a new object by explicitly calling the constructor, forcing correct values to all data members, and then pushing the resulting object into the STL container. This is an example of a situation where one line of code on one target platform can require tens of lines on another platform.

Instead of pre-processor directives or in addition to them, also other tools for inserting platform-specific code during a platform-specific build process are available. The developers can choose to use another intermediary language that enables specifying platform-specific sections and will be automatically parsed resulting in compilable code. Visual programming language related systems with code auto-generation from the representation, configured to
produce code that follows the practices of each target platform, are another tool that can be
used for creating cross-platform code, possibly with illustrative conventions to distinguish
platform-specific sections from other code. The careful decoupling of application-logic
related code and graphical user interface (GUI) related code is probably a self-evident choice
for any developer who starts writing cross-platform code; as noted in [1], the developers can
take one step further by using platform-agnostic techniques for describing GUI elements.

Some properties of the underlying hardware platform emanate to the source code level,
even if no special APIs or libraries are involved. An example of this is the endianness of
multi-byte values: are the values stored with their least significant or most significant byte in
the lowest memory address? Of course, in any project it is good practice to write code that is
not dependent on whether the machine has a little-endian or big-endian architecture, but in a
cross-platform project the importance of such programming habits is more emphasized.
Another, nearly self-evident practice related to basic data types is the provision of helpful
typedef-settings for data types of specific bit widths, such as int32.

Furthermore, in cross-platform development, compiler warnings should be treated with
severity. In some cases, a code feature that produces only a compiler warning on one platform
may lead to run-time errors on another platform.

3. Accessing system resources

Practically no programs operate in a vacuum where all actions are only about manipulating
data structures within strictly one program. Typically, the creation of cross-platform code
involves access to system resources such as libraries, the file system, networks, timer and
concurrency facilities, and inter-process communication (IPC) mechanisms. All the
mentioned types of resources were also used in the MP2PP project.

Special libraries not available on all target platforms can complicate the development
process. More generic resources, such as the file system, are usually available, but are not
completely uniform across the different platforms. While the interfaces to file systems are
fairly similar with directory trees and other shared concepts on many platforms, there are
differences that are definitely visible to the developers. For example, some systems use drive
letters while others do not, and certain characters may be prohibited in file names on a given
platform.

Concurrency support is often vital for networking-related and other software. Different
platforms provide different facilities for execution concurrency, including varying kinds of
threading, timer event subscription, and mutual exclusion (mutex) constructs. Concurrency-
related bugs are often hard to find; thus, it is important to create concurrency-support
mechanisms that are suitable for cross-platform usage. For our P2PP implementation, in the
Linux-based environments we used the Posix threads API to implement threading. In
Symbian OS, we used the active objects API for achieving the required level of concurrency.

TCP and UDP sockets are a widely supported technology on different platforms. Network
access related code in the MP2PP middleware, however, was realized as two separate
platform-specific sets of functionally overlapping files, because the socket APIs on Linux and
Symbian OS were dissimilar enough to justify the separate source code files. We made a
similar split of code in platform-specific files when implementing the logging system.

The character sets in use on different platforms have an effect on cross-platform
compatibility. Although dealing with character sets may not be a key problem in middleware
development that we are especially handling in this article, it is true that for example in
Symbian OS the default binary representation of characters is a 16-bit encoding. That might
be different from the encoding used internally by the middleware or the related
communication protocols and thus must be taken into account in the API of the networking middleware when dealing with strings that originate from an application but need to be stored or manipulated in the middleware. In the MP2PP project, we chose to make it the caller’s responsibility to provide strings in a uniform encoding to the middleware API; thus, the middleware itself did not need to apply platform-specific conversions to the input strings, which arguably would have been more error-prone.

Especially in a project whose set of target platforms contains platforms with greatly differing capabilities, it is good to pay attention to the factors in the software that may cause excessive load on the less powerful platforms. We are not talking about optimization here; the optimization of code should, of course, be the compiler’s responsibility, since an automaton will nearly always optimize code better than a human being could. Instead, we are talking about situations where some functionality that generates load is not necessary and can be eliminated. For instance, the extensive logging of the details of program execution in a file may skew the results of performance measurements or waste resources such as disk space, which may be scarce in a mobile device. Of course, if the developer observes that most of the logging can be eliminated on a specific target platform but some of the loggable items – for example, warning messages or indications of major actions – are important enough to be logged regardless of the platform in use, the developer can take into use the well-known system of “log levels”. This means that each log message has a priority setting, and only the messages of a specified or higher priority level will get logged in a file, and naturally the setting can vary by platform; thus, on a server the code might be run at a maximum log level, while in a mobile phone it might only log the error situations. Even when operating on a single platform, the usage of log levels makes it possible to run the software with varying levels of information resolution.

Sometimes a given platform has “resources” or properties that do not have any corresponding property on the other platforms. As an example we mention the “capabilities” system of Symbian OS, which is designed to restrict an application’s access to potentially dangerous API calls and thus prevent the execution of malicious program code. The feature caused problems, when we tried to implement a P2P usage plugin for direct MP2PP middleware access from within the default Web browser, in order to enable Web-based P2P applications on the Symbian OS platform in the N95 devices. The plugin had to be implemented as a DLL, and DLLs in Symbian OS need to have at least the same capabilities as the application that they are used by. Thus, we were not able to create the plugin without the permission to use very sensitive capabilities that the Web browser did have and that are difficult to obtain for typical developers. This kind of surprising limitations can paralyze cross-platform development work on some target platforms, while the work on other platforms is completely unaffected.

Moreover, the developers cannot rely on the existence of certain resources that they may take for granted on their favorite platforms; for example, developers of Linux software cannot always assume standard output, input, and error streams, shell scripting, named pipes, or similar facilities to exist on the other platforms. The differences of the platforms’ software distribution package formats and available installer tools can affect the uniformity of the software installation procedure. Still another point to consider is the license agreements of different libraries and other tools on different platforms that affect the license status of the software being developed, which may require systematic control in order to avoid situations where licensing causes problems on some of the several platforms.

The binary file that constitutes our MP2PP middleware is an ordinary shared library (in Linux) or DLL (in Symbian OS), which is linked against when compiling the applications
that depend on MP2PP. This is different from the implementation of our previous middleware, PnPAP [7], which was run as a separate process, and the applications using the middleware sent requests to it over an IPC channel.

Speaking of IPC operations, we suggest that it is good to keep IPC simple and based on serialized strings that can be passed over various IPC channels. This is an example of the rule that designing for portability is good for potential cross-platform compilability as well.

4. Considerations for project management

4.1. Code-base management

In every non-trivial software project, the usage of the source code files – and often also different kinds of data files and utility scripts – during the compilation process is managed with a suitable facility, for example, the widespread makefile system. The purpose of such facilities is to define, in a reusable and comprehensible way, the relationship between the source files and the files (typically binaries) that are created as a result of the build process.

In cross-platform programming, these compilation facilities may be different for different target platforms. In the MP2PP project, for Symbian OS targeted builds we used the Symbian OS specific system of MMP files, while for Linux-targeted builds we used the makefile system. As a result, for example the addition of a new source code file involved adding the filename to both systems. In large projects, the redundant work in various code-management activities can be minimized in some cases by automating the operations, thus, the developer does not need to manually repeat them for each target platform. Moreover, on different platforms one must sometimes adapt to differing recommendations for matters related to code management, such as the directory structures for organizing the code files.

Even revision control systems and other project support systems can offer tools for cross-platform development. For example, it is possible as a routine practice to mention the target platforms affected by a bug when reporting it in the bug database. Or, to indicate the platforms specifically affected by a code modification in the comments of a source-code commit transaction in revision control. These kinds of practices can be adopted if the formal tracking of platform-specific changes is desired in the project.

4.2. Managing the complexity of cross-platform adaptation

In a perfect world, the developers would always know before the beginning of a project all the target platforms on which the created software must run. In reality, it may occur that new required target platforms are introduced, when some code has already been written. Even when only one target platform is expected to be enough for a software project, it is usually a good practice to design the code keeping in mind the possibility for its future cross-platform usage.

Suitable project-related software tools such as the selected integrated development environment (IDE) can also facilitate certain aspects of cross-platform programming. For example, in some cases a developer could utilize the visual formatting and highlighting properties of the editor for showing platform-specific code sections in a particular color. This would involve writing pattern matching rules for the color-coding system, and would probably be practical only in cases where it is crucial to quickly and repeatedly recognize which parts of the code being edited are specific to a given target platform.

There are certain factors that affect the complexity of modifying code, which is originally intended for a single target platform, to be cross-platform compatible instead. Let us consider
a situation where there are two final target platforms, A and B. Often asymmetric difficulty is evident when one compares the relative difficulty of porting from A to {A, B} versus porting from B to {A, B}. The asymmetry arises from the fact that the platform-specific conventions of A are more easily accommodated on B, while those of B require more complex operations in order to be accommodated on A.

To ground this statement into reality with an example, we mention our related observations – one of the most important findings in this article – from the MP2PP project. It became evident that when a particular class, library, or similar entity had been originally written for Linux, it was relatively effortless to modify it into a cross-platform version that works with both Linux and Symbian OS. In the other direction, however, the process was not as straightforward: code that was originally developed with only Symbian OS usage in mind was more challenging, or at least more time-consuming, to modify into cross-platform code. The reason for this was that the facilities upon which Linux-based programs heavily rely, such as the C standard library, are available on Symbian OS, but the Symbian OS specific headers and libraries – not to forget some exotic services of the operating system – are not available on Linux systems. Thus, creating a system that mimics the peculiarities of Symbian OS can be a frustrating and inefficient attempt, while cross-porting in the other direction (from Linux to Symbian OS) is often facilitated by the relatively good support of standards-based code on Symbian OS; as even more libraries from the other platforms are being ported to Symbian OS – and the APIs of Symbian OS, for a good reason, are not being made available on other platforms – this property is only getting stronger.

It is justified to ask, whether the notion of asymmetric difficulty is relevant in practice. If there is no possibility to choose whether A or B is the target platform of the initial single-platform implementation, the knowledge about the asymmetric difficulty cannot influence the platform decision; it may even seem that the rule can only be used in retrospect. However, this is not necessarily the case. When developing for a single platform whose code is known to be relatively hard to port to other platforms, the developers can proactively take into account the possibility of future cross-platform support and utilize programming practices that make it easier to convert the codebase into a cross-platform implementation.

In cross-platform development, the importance of communication between developers is even greater than in single-platform development. It is possible that not everyone in the developer team knows all the target platforms equally well. For example, a developer might be an expert on one platform and only know the basics of the other platforms. Thus, having experts of each target platform in the developer team is not enough, if they cannot dynamically share their knowledge during the development process. Because all developers typically write code that must run on all platforms, they must be able to communicate, for example, in order to ask a platform expert whether a given code section is correct for a specific target platform. In the MP2PP project, communication between developers was ensured by keeping the developers physically close to each other: all programmers of the core components of the middleware were located in one single room, and moreover, distractions were minimized because no other projects’ staff was located in the same space.

4.3. Testing

In addition to the kind of testing where the implementation is run on only one target platform at a time, cross-platform software projects may also involve testing where multiple heterogeneous target platforms are involved in a single test scenario. This may explicitly be interoperability testing of the software builds on different platforms – with the software
instances on different platforms communicating, for example, over a network – or the testing can have entirely different aims.

In the MP2PP project, we ran test scenarios where both fixed-line computers and mobile devices acted as peers in the same DHT-based P2P overlay. The successful interoperation of the builds on different target platforms increased confidence in the correctness of the code.

Many bugs in code are platform-independent. Finding and eliminating such a bug on one platform results automatically in eliminating the bug on the other platforms as well. Thus, it is beneficial that in the set of the target platforms there is at least one platform, where debugging is relatively effortless or can be done using tools that the developers know well. In the MP2PP project, the developers were more comfortable with run-time debugging on Linux-based platforms than on Symbian OS. Thus, it was good that we were able to run the code in Linux and do the run-time debugging in that environment. However, even if the nature of a particular bug is platform-independent, it might occur that the effects of the bug are detectable by testing only in a subset of the target platforms.

5. Development overhead analysis

It is a complicated task to assess numerically the amount of work used for creating a particular software component and to determine which roles in the software production (e.g. lead programmer, regular programmer, or tester) or which individual had the most influence on the outcome of a project. It should also be noted that the related metrics are prone to be misleading or even abused by management: for example, the canonical mistake would be to think that the developer who writes the most lines of code is surely the most productive. Despite these limitations, we attempt to analyze numerically how much overhead in the workflow of software developers is caused by the cross-platform compatibility related tasks during a project. We perform a case study on our MP2PP project.

We distinguish two lines-of-code (LOC) based metrics: added LOC and affected LOC. Added LOC refers to the number of lines that can be seen as added to the corresponding single-platform implementation in order to achieve cross-platform support. Affected LOC is the number of lines that are modified in order to achieve cross-platform support, i.e. lines that are different when compared to a single-platform implementation, but would exist also in the single-platform implementation. If a specific line is counted as part of the affected LOC, it is never counted as part of the added LOC, and vice versa; thus, the two metrics do not overlap.

LOC-based metrics are arguably a weak means for assessing the actual amount of work invested in the code and moreover our selected metrics are not entirely unambiguous: it may not always be easy to determine, for example, whether a given line should be included in the affected LOC. A metric better than LOC could be the time overhead used for implementing cross-platform related properties of the code; this is, however, obviously much harder to measure than LOC. Moreover, the greatest overhead in the development time of cross-platform code might not be incurred by the writing of cross-platform code; instead, the majority of overhead may originate from the testing, debugging, and other similar activities on the several platforms. Those aspects of the effort probably have a rough positive correlation with LOC, but the observation of this correlation is excluded from our study.

We scrutinize the source code files of a selected software module from the MP2PP project and calculate the added LOC and affected LOC. We selected the P2PP module, which contains the P2PP implementation, and excluded other parts of the middleware such as the implementation of the DHT algorithm, the network access mechanisms, and the SIP stack that utilizes the P2PP overlay as a distributed storage system for user information. The size of the P2PP module, comprising seven .cpp files and seven .h files, was about 8,400 LOC. For
facilitating the LOC calculations, we used the open-source utility program “CLOC”, which is able to distinguish (among other things) whitespace and commented lines in the code files.

We define the following symbols for the different measured LOC quantities: the total LOC number of the actual cross-platform software module is $N_{\text{cross}}$; this includes all lines, even those that contain only whitespace or comments. We also separately measured $n_w$, the number of lines that contain only whitespace or are completely blank, and $n_c$, the number of lines that contain only comments.

We also measured the affected LOC number of the entire module, $n_{\text{aff}}$, and the added LOC number of the entire module, $n_{\text{add}}$. It should be noted that the set of lines counted in $n_{\text{aff}}$ does not overlap with the set of lines that were counted in $n_w$ or $n_c$. The lines counted in $n_{\text{add}}$ contain also some whitespace-only (or blank) lines, because some of the Symbian-specific code sections between pre-processor directives contain also blank lines. The number of these added blank lines is $n_{\text{addw}}$.

The number of lines in a hypothetical single-platform (Linux) version of the module can be calculated: $N_{\text{Linux}} = N_{\text{cross}} - n_{\text{add}}$. The single-platform version does not exist, but we simulate a situation where we have a single-platform version, since it is interesting to estimate how much effort would be needed for changing it into a cross-platform version.

Then, we calculate the number $N_{\text{cross_funct}} = N_{\text{cross}} - n_w - n_c$ of lines that are “functional” (i.e. not comments or whitespace-only) and the number $N_{\text{Linux_funct}} = N_{\text{Linux}} - n_w - n_c$ of lines that are functional and would be part of the single-platform code module.

Finally, we can calculate $n_{\text{gen}} = N_{\text{cross}} - n_{\text{aff}} - n_{\text{add}}$, which is the number of generic lines. The generic lines are those lines that do not have any platform-specific properties. If only functional generic lines are counted, their number is $n_{\text{gen_funct}} = N_{\text{cross_funct}} - n_{\text{aff}} - n_{\text{add_funct}}$, where of course $n_{\text{add_funct}} = n_{\text{add}} - n_{\text{addw}}$.

Based on the above-mentioned LOC quantities, we report the percentages of LOC belonging to different categories. In Figure 2, we show the percentage of certain observed LOC quantities in relation to $N_{\text{cross}}$ and to $N_{\text{cross_funct}}$. The values are calculated as follows:

- $p_{\text{cross_gen}} = \frac{n_{\text{gen}}}{N_{\text{cross}}} \cdot 100\%$
- $p_{\text{cross_gen_funct}} = \frac{n_{\text{gen_funct}}}{N_{\text{cross_funct}}} \cdot 100\%$
- $p_{\text{cross_aff}} = \frac{n_{\text{aff}}}{N_{\text{cross}}} \cdot 100\%$
- $p_{\text{cross_aff_funct}} = \frac{n_{\text{aff_funct}}}{N_{\text{cross_funct}}} \cdot 100\%$
- $p_{\text{cross_add}} = \frac{n_{\text{add}}}{N_{\text{cross}}} \cdot 100\%$
- $p_{\text{cross_add_funct}} = \frac{n_{\text{add_funct}}}{N_{\text{cross_funct}}} \cdot 100\%$.

Figure 2. LOC quantities in relation to $N_{\text{cross}}$ and $N_{\text{cross_funct}}$. 
In Figure 3, we examine the situation from the viewpoint of the hypothetical single-platform implementation; in other words, we show the percentage of the LOC quantities in relation to $N_{\text{Linux}}$ and to $N_{\text{Linux\_func}}$. The aim is to assess the amount of work needed for modifying the single-platform (Linux) version of the code into a cross-platform (Linux and Symbian OS) version. The values are calculated as follows:

- $p_{\text{Linux\_gen}} = (n_{\text{gen}} : N_{\text{Linux}}) \cdot 100\%$
- $p_{\text{Linux\_gen\_func}} = (n_{\text{gen\_func}} : N_{\text{Linux\_func}}) \cdot 100\%$
- $p_{\text{Linux\_aff}} = (n_{\text{aff}} : N_{\text{Linux}}) \cdot 100\%$
- $p_{\text{Linux\_aff\_func}} = (n_{\text{aff\_func}} : N_{\text{Linux\_func}}) \cdot 100\%$
- $p_{\text{Linux\_add}} = (n_{\text{add}} : N_{\text{Linux}}) \cdot 100\%$
- $p_{\text{Linux\_add\_func}} = (n_{\text{add\_func}} : N_{\text{Linux\_func}}) \cdot 100\%$.

Among other things, we see the following results from the figures. According to Figure 2, 18.5% of the cross-platform code is somehow Symbian-adjusted (affected or added), or 28.0% if only the functional lines’ proportions are taken into account. According to Figure 3, changing the single-platform implementation into a cross-platform implementation would involve modifying 12.5% of the lines of the single-platform implementation and adding 7.3% more lines to it, or modifying 21.4% and adding 9.2% if only the functional lines’ proportions are taken into account.

6. Conclusion

We discussed techniques related to the development of cross-platform mobile middleware. As for language-internal constructs, adaptation mechanisms can be used for making the cross-platform code follow the conventions of the multiple target platforms to an adequate degree. This may involve dropping some conventions of certain target platforms and making sure that the applied conventions keep the relevant platform-specific details visible in the code but still remain understandable also for developers who have knowledge only on a subset of the target platforms. The objectives include promoting consistency and preventing any bad conventions.
of a given platform from dominating the cross-platform code. As for accessing the system resources or utility code on heterogeneous platforms, it was pointed out that the calling code should not make too many assumptions about the facilities provided by the underlying platform. Guidelines for support activities such as source-code management were provided.

As a case study, we analyzed how the quantities of LOC in a cross-platform (Linux and Symbian OS) P2P protocol middleware implementation were distributed across one platform-independent category and two kinds of platform-adjusted categories. The analysis provided an approximation of the work overhead caused by cross-platform adaptation.

7. Acknowledgments

The research work was done in the DECICOM project at the University of Oulu. We acknowledge the valuable contribution of Janne Julkunen in the implementation of MP2PP. The authors would also like to thank TEKES (the Finnish Funding Agency for Technology and Innovation), Ericsson, Nokia, and Nethawk for supporting the project financially.

8. References


Authors

Otso Kassinen is working as a research scientist in the MediaTeam Oulu research group at the University of Oulu, Finland. He received his M.Sc. degree in Information Engineering from Oulu University, Finland, in 2007. He is currently working towards a Ph.D. degree at the University of Oulu. In his research work, he is focusing on mobile middleware systems, protocol software development, and application-level P2P networks.

Erkki Harjula is working as a research scientist and the project manager of the DECICOM project in the MediaTeam Oulu research group at the University of Oulu, Finland. He received his M.Sc. degree in Electrical and Information Engineering from the University of Oulu, Finland, in 2007. He is now studying on a Ph.D. degree at the University of Oulu. The research interests include P2P protocol optimizations for mobile and other constrained-capacity networks.
Timo Koskela received his M.Sc. degree in Information Engineering in 2005, and M.Sc. degree in Management and Organization in 2009, from the University of Oulu, Finland. He is now working as a researcher in the MediaTeam Oulu research group at the University of Oulu, and is focusing on his Ph.D. degree.

Dr. Mika Ylianttila is an Adjunct Professor at the Computer Science and Engineering laboratory and Associate Director in the MediaTeam Oulu research group, at the University of Oulu, Finland. His research is focused on multimedia communication, including topics on overlay networking, community and content management and user experience.