

---

# Case study on Symbian OS programming practices in a middleware project

Otso Kassinen, Timo Koskela, Erkki Harjula and Mika Ylianttila

University of Oulu, Department of Electrical and Information Engineering,  
Information Processing Laboratory  
P.O.BOX 4500, FIN-90014 University of Oulu, Finland  
E-mail: firstname.lastname@ee.oulu.fi

**Summary.** In this case study, we provide practical guidelines for Symbian OS software development by analyzing our experiences from a three-year mobile software research project focused on the creation of networking middleware and collaborative applications. The practices that we either conducted in the project or conceived in retrospect are summarized, and re-applicable patterns of activity in development work are identified. Based on the observations, guidelines for essential design and implementation activities are derived and potential pitfalls are addressed through example cases to provide experience-backed information for Symbian OS software developers. Among other things, the guidelines advocate the use of platform-independent solutions, the minimization of project dependencies, and the representation of complex activity sequences in a human-readable form as a routine practice.

**Keywords.** Symbian OS, middleware, mobile software development

## 1 Introduction

Symbian OS is the leading smartphone operating system in the world, with Symbian Ltd.'s recent (January 2008) plans to spread into basic mobile phones as well. This strong position of the operating system emphasizes the importance of high-quality software development processes. Unfortunately, Symbian OS is not the most developer-friendly platform: it is regarded as a programming environment with special challenges and hard-to-learn practices for the developers [1] [2].

On such a challenging platform, wrong decisions in the software design phase may lead to significant difficulties in the implementation phase. Also incorrect implementation practices may complicate software testing and deployment. Indeed, without prior knowledge of the potential pitfalls, unforeseen problems may arise, compromising software quality or increasing development time. Moreover, specifically the creation of mobile middleware systems is a task that poses special challenges to the developers on a wireless platform [6].

In 2004 through 2007, we conducted the three-year All-IP Application Supernetworking research project, during which we designed and developed experimental mobile software for the Series 60 2nd Edition Feature Pack 2 smartphone platform, Symbian OS version 8.0. The project's main deliverables were a mobile middleware platform, called the Plug-and-Play Application Platform (PnPAP) [3], and collaborative applications using the middleware's services. The PnPAP middleware provides several networking-related services: the dynamic switching of different connectivities (wireless network interfaces) and peer-to-peer (P2P) networking protocols, as well as the initiation of application sessions and the distribution of context information.

Four collaborative applications were implemented on top of PnPAP; the word collaborative refers to the applications' ability to perform certain tasks by using the services of each other. The applications are

- FileSharing, a P2P file sharing application;
- NaviP2P, a group-interactive navigation application;
- RealTime, a basic mobile VoIP application; and
- Wellness, a sports diary for monitoring physical exercise and sharing exercise results within user groups.

In this case study, we describe our experiences from the various Symbian OS software components we developed, analyzing the observations in order to identify patterns of activity that can be repeatedly used for beneficial results. We describe what was probably done the right way and what could have been done differently. The aim of this analysis is to provide practical guidelines for developing mobile middleware and applications for Symbian OS, reducing the potential of platform-related risks realizing.

Software project management boils down to the management of complexity; each of our guidelines aims to cut down a specific type of complexity in the development process. While our work was prototype development in a research group and the number of our software-developer personnel varied between four and six, many of the guidelines should also be applicable in industry and in larger projects, where dealing with complexity is even more crucial.

The rest of this chapter is organized as follows. In Sect. 2, the timeline of the observed project is presented, and in Sect. 3, the specific experiences from the project are analyzed and practical guidelines for Symbian OS software development are derived from them. Sect. 4 summarizes the guidelines provided.

## 2 Timeline of the observed project

In this section of the case study, we provide a rough timeline of our project, to explain in what order the key software components were developed. This makes it easier to put into context the specific technical challenges analyzed

later in this case study. The realized timeline of the components' development and their project-internal dependency relationships are illustrated in Fig. 1.

### 2.1 Project year 2004-2005

In mid-2004, after a period of requirements gathering and initial design, we started to develop the first PnPAP version and the FileSharing application, depending on PnPAP, in parallel. The first middleware functionalities were those required by FileSharing: the access to P2P protocols that enable file sharing. At first, FileSharing only used a dummy protocol module that simulated P2P communication, but during spring 2005 a Direct Connect++ (DC++) file sharing protocol module was implemented so that PnPAP contained an actual P2P protocol that the FileSharing application could utilize. As PnPAP was implemented to treat protocols and connectivities as replaceable components with generic programming interfaces, also the GPRS connectivity was implemented as a module. During 2004, design work was done to identify the peer-to-peer (P2P) protocols to be implemented besides DC++. However, DC++ remained the only P2P protocol implemented during the project, as our main focus was on creating novel applications and the mechanism for the dynamic switching of modules so that the applications are always provided the best combination of a connectivity and a P2P protocol.

### 2.2 Project year 2005-2006

In 2005, the first version NaviP2P application was created and was refined during winter and spring 2006; in parallel, support for location data acquisition from an external GPS device was added to PnPAP, along with a Bluetooth connectivity module. Session Initiation Protocol (SIP) was needed for PnPAP nodes' communication; in 2005, a lightweight SIP stack was created, because existing SIP stacks for Symbian did not support multi-connectivity networking. The first versions of Wellness were developed. Towards the end of this project year, there was also progress in creating a State Machine Executor (SME), to be integrated with PnPAP later. The SME's purpose was to run intelligent state machines that control the switching of connectivity and P2P protocol modules. The SME was written in Python, as opposed to all other components written in C++.

### 2.3 Project year 2006-2007

In the summer of 2006, the most part of the RealTime application was created and Wellness was further developed. During the same summer and fall, PnPAP's support for application collaboration in our existing applications was extended so that applications could also be started from within each other in

a context-sensitive way (this was called application supernetworking). Distribution of the users' personal context data within user groups was also implemented so that the context-aware applications Wellness and NaviP2P could make use of it. Wellness and NaviP2P were subjected to user tests in July-October 2006. During the fall and winter, a session management component was added to PnPAP and applied with RealTime; this component enabled the semi-automatic installation of RealTime and launching of a VoIP session in a situation, where the callee does not yet have the same application as the caller has. In the winter 2006-2007, the SME was integrated with PnPAP. PnPAP, in its final form at this project year's end, consisted of about 20.000 lines of Symbian C++ code. In addition, the applications, connectivity and protocol modules together consisted of approximately 50.000 lines of code.

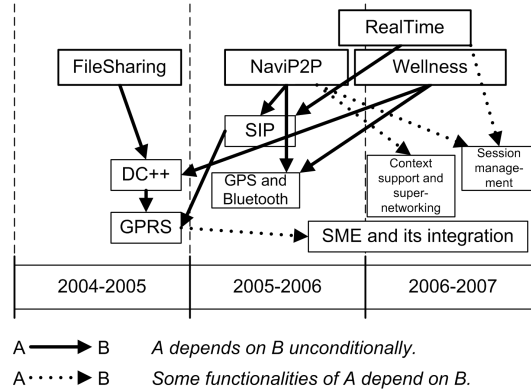


Fig. 1. Project timeline and internal dependencies

### 3 Analysis of the observations from the development work

This section of the case study elaborates the experiences from the project. Based on the experiences, practical guidelines about the operations models and choices of technologies for Symbian OS software development are presented.

#### 3.1 Middleware structure and API

The PnPAP core, the nerve center of the middleware within the mobile device, was designed to be a server process behind the Symbian OS Client-Server interface that is provided by the platform for Inter-Process Communication (IPC). In essence, the applications using PnPAP would issue method calls

to a client-side stub Dynamic Linked Library (DLL), which implements the PnPAP Application Programming Interface (API) and translates the calls to Client-Server requests that are sent to the PnPAP core, to invoke a specific middleware behavior. On the server-side of PnPAP, every application's Client-Server connection was paired with a stub object responsible for handling that particular application's messaging. The architecture is depicted in Fig. 2; dotted lines represent process boundaries.

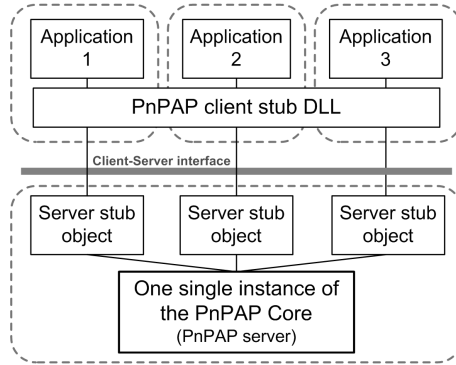


Fig. 2. Project timeline and internal dependencies

Early decisions (in 2004) about the internal structure and external interfaces of the PnPAP middleware affected our development efforts during the rest of the project. In retrospect, the selection of the Client-Server interface for the application-PnPAP connection was not an optimal decision. The Client-Server connection has been designed for the OS internal services with a strict "client calls, server responds" scheme. It seemed to work well for our IPC in the design phase, but when the middleware was implemented, we noticed that the lack of symmetric messaging was an issue, because the applications also had to react to messages that originated from the middleware (i.e. the server calls the client). The reception of messages from the network, as well as the local application collaboration scheme, involved asynchronous messages from PnPAP to the applications.

We had to use the file system as an intermediate storage for the data in server-to-client communication. The data were written to the file by the PnPAP core; when the file was modified, the client application was signaled by a file system callback that was subscribed to with the `RFs::NotifyChange()` method. When receiving the callback event, the client would read the message from the file. Obviously this was a makeshift solution. Local-host sockets with their inherent two-way communication would have made IPC easier and more rapid to implement.

Sockets were later used in 2006-2007 for connecting PnPAP and the SME, and they proved to be more flexible than the Symbian Client-Server connec-

tion. Of course, small inconveniences such as data serialization were unavoidable in both cases, but the Symbian Client-Server interface 1) requires the programmer to learn its quirks while sockets are familiar from other platforms, 2) does not provide symmetric communication while sockets do, and 3) hinders the communication between native code and programs written in other languages. In addition, there is 4) an even more compelling reason to utilize standard solutions for central technical elements (e.g. sockets for IPC): the risk that proprietary solutions may undergo significant changes over time, which is exemplified by the following.

When we tried to port our code-base from Series 60 2nd Edition to the new 3rd Edition in 2006, the Symbian-proprietary Client-Server API, among other things, had been slightly changed in the platform. There were dozens more of similar little differences between the Editions: in the API and also in the details of project definition files, not to forget the new requirement to digitally sign the software in order to enable certain functionalities that were available in the old 2nd Edition without signing. The amalgamation of all these differences rendered the porting of the large code base a frustrating task. After two weeks of trying we gave up and decided to stay with the old 2nd Edition platform. The benefits of porting would not have been worth the costs: for us, the only actual benefit of the 3rd Edition smartphones would have been the availability of a Wireless LAN connectivity for extending PnPAP's connectivity-switching scheme.

In sharp contrast to the Symbian OS APIs, one can safely assume that the details of established technologies such as sockets will remain relatively unchanged. The less changing components, the easier it is to ensure that future software versions will work without big modifications. The key lesson here is that if you can use a tried-and-tested, non-Symbian-specific solution, it may be a better choice than a proprietary API subject to frequent changes. Nevertheless, it should be noted that sometimes the use of platform-specific solutions is justified. For example, dependencies to third-party software may dictate the use of Symbian-specific schemes.

### 3.2 Remote node intercommunication

To enable the PnPAP nodes to communicate with each other over the wireless networks, PnPAP offered the applications the functionality for disseminating context information and other messages embedded in SIP packets. The serialized data structure we designed for the messages was deliberately simple: an array of name-value pairs mapped to each other, values indexable by their names, with no restrictions for the values (i.e. arbitrary byte strings were accepted). Nesting of such lists, by inserting a list as a value in another list, was also supported.

Serialized data were easy to parse at the receiving node, which was a good thing, since parsing is typically more difficult than the creation of the same message, thus improvements in parsing result in more robust code. The data

structure was also reused for the collaboration of local applications, which was a special case of device-internal IPC. A rule can be worded: design the basic infrastructural components to be generic enough, so they do not change often; also their reuse potential increases.

One key advantage in the serialization format was the inclusion of explicit length information about the contained data items, as opposed to, for example, the Extensible Markup Language (XML) in which the parser must traverse each data field until it encounters the field's ending tag. In comparison to XML, the explicit length information used in our solution improves the parsing performance of a single  $N$ -length data field from  $O(N)$  to  $O(1)$ , makes the parser less complex, and eliminates the need for "escape characters" that protect the parser from interpreting data as special characters. As a generalized rule: whenever you can use out-band signaling, do not use in-band signaling. Here the word signaling is used in a broad sense. The rule applies to the design of protocols and file formats. Protocol design is often dependent on the other systems that are communicated with, but if easily parseable messages can be used, it may save computational resources in Symbian software.

### 3.3 Dynamically pluggable modules

Symbian's polymorphic DLLs were used for implementing the dynamically installable components (protocol and connectivity modules that can be downloaded and taken into use on-the-fly) for PnPAP. Polymorphic DLLs, by definition, implement a specific interface, and a polymorphic DLL with the same interface but different behavior can be loaded at any time, without terminating the running process. Polymorphic DLLs were a good choice for dynamically pluggable components. Besides their good on-the-fly replacement ability, polymorphic DLLs have the benefit of being able to contain arbitrary native binary code, which might seem self-evident now, but in early 2004 we had plans to implement the very P2P protocols as state machines within the SME. This would have resulted in overly complex state machines with probably quite limited ways of interacting with the operating system's services. Of course, polymorphic DLLs were far more flexible.

The state machine based solution was, in essence, an example of a task-specific language that needs to be interpreted by the actual Symbian code. The lesson learned is that the option to implement a feature by using a task-specific language should be exercised with care. Of course, there are also situations where the parsing of a control script or other task-specific language in Symbian software is completely justified.

### 3.4 Validation and debugging with observation of time-dependent sequences

During the project, there was one specific aspect of software work that popped up again and again: the human inability to follow complex action sequences

in code, either within one program or within a system of intercommunicating programs. This property of humans has been explicated in [5], where it is suggested that "even the simplest procedural logic is hard for humans to verify, but quite complex data structures are fairly easy to model and reason about. (...) Data is more tractable than program logic." We agree on this, and state that a mapping from temporal sequences in code execution to an easy-to-follow data structure makes software validation and debugging simpler in many cases; this is indicated by our specific experiences on the matter, described below.

In our project, it was not uncommon to be in a situation where a program error had to be searched in a sequence of 10 to 20 nested function calls, many of which were located in different source files. Additional complexity was introduced when there were multiple communicating processes or any asynchronous actions were involved. Needless to say, the complex action sequences were difficult to visualize based on the source codes and to comprehend by the developers, when trying to find out why an application crashed. This is highlighted in Symbian OS, because the ability to observe the course of program execution is very limited in a restricted device, and, moreover, the prevalent coding style of Symbian software advocates the use of complex class hierarchies that may result in long, hard-to-follow chains of method calls through several object instances, even for relatively simple actions.

Of course the debug mode in an Integrated Development Environment (IDE) allows a developer to step through running code, diminishing the need to visualize complex action sequences in one's own head when locating a misbehaving line of code. However, with our IDE and deployment platform, we could not use on-target debugging, and in practice all of our testing had to be done on target hardware (phones), which was time-consuming.

For capturing information about the temporally related actions during the execution of our Symbian software, we made heavy use of sequence logging with plain-text log files accessible from all processes. This simple technique proved to be a great help for resolving of both, network message flows and device-internal program sequences, as it was common to have sequences that involved call sequences through multiple processes.

One key point here is the importance of a stateless logging facility. Before understanding this, we used a logging code that opened a log file at the beginning of the execution and closed it only when the program exited; clearly this was stateful action. Since multiple different processes had to write to the same file, it was not smart to keep the file-handles open. The situation was fixed with stateless logging functions: when a string was to be logged, the logger function would open a file, write into it, and close it immediately. This repeated opening and closing of files caused no discernible loss of performance although Symbian software is often performance-critical. Speaking of text files, the virtues of storing information in plain-text (instead of binary structures that require dedicated access tools) is well explained in [4].

The power of visualizing complex, time-dependent chains of actions in a human-friendly way is also apparent in the sequence diagrams of the Unified

Modeling Language (UML) that often facilitate the important design phase of software.

### 3.5 Developers' support functions

By developers' support functions, we mean the activities that are not strictly part of programming, but do contribute to the success or failure of a Symbian project. Symbian programming is hard enough in itself; the development team should not neglect the appropriate support functions that make their programming efforts at least manageable.

A rule that helps in daily software development work is the assertive indication of details for the developer. Developers should use techniques that, without much additional human effort, indicate assertively to the developer that details in the running code are as they should be and also make it easy to notice the situation when they are not.

An example of assertive indication is the use of proactively displayed build IDs or a similar solution that makes sure that the correct binary version has been updated on the handset after source-code modification. This is emphasized in Symbian OS, where the building and installation of new software versions is a multi-phase task involving multiple devices (the developer's PC and the target hardware). Our solution was to show a small indication of the binaries' successful update in a log or in a pop-up note. For example, when a particular function was altered in the sources, a once-per-build modified tag in the function's log output confirmed that the most recent version of the corresponding source file had been compiled and run. Outputting an automatically modified build ID at the beginning of the program's execution would also have been possible, but its power to confirm the source file modifications would have been less localized.

Some programmers in our team maintained a human-readable log of their activities for their own reference, to be prepared for cases where one suddenly needs to know, for example, how a given software module was fixed 20 days ago. Humans simply forget most of the details of solving a particular problem, but a simple text file remembers those details, for instance, the steps of advancement with the code, all in a highly readable form for a human to glance through. We recommend this kind of logging especially for programmers in Symbian OS, where hard-to-understand roundabout solutions are sometimes used and it is important to trace back how the programmer has ended up with a particular solution. Again, personal logs are an example of mapping a complex action sequence to a tractable representation.

Sometimes the Symbian OS environment seems to be restricted in ways that are not understandable. Developers should take into account the possibly indirect consequences of these shortcomings. For example, we have not seen Series 60 2nd or 3rd Edition phones that would display the detailed panic code on the screen (as a default action) when an application crashes. It is possible for developers to make the panic codes visible. However, if the codes

were displayed as a default action, it would be beneficial for the end-user, as she could inform the software vendor about the details of a program crash. However, for some reason, the system just does not display the code; programs fail silently and often inexplicably.

This shortcoming has two corollaries: firstly, the developers must explicitly enable the indication of panics in the implementation phase; secondly, the panic codes are practically never displayed to the end-user, which is the more serious corollary of the two. This flaw in the Series 60 design causes frustration when searching the error in a program, and indirectly compromises Symbian OS software quality.

In addition to the creation of the `ErrRd` flag file that turns on the displaying of panic codes in Series 60 devices, we used the `D_EXC` software [7] for capturing panic codes and stack traces of crashing programs; another utility for this purpose has been introduced recently in [13], and also the `Y-Tasks` utility [12] works as a crash monitor, resource usage monitor and process manager.

Besides panic indication, the Series 60 2nd Edition platform lacked decent user interfaces to the running processes and to the file system. It is unclear why a smartphone does not provide a computer's basic resource-controlling tools to the user, if the phone is meant to be a handheld computer. For these functionalities, we used certain third-party utilities: `TaskSpy` [11] for monitoring and killing processes (`Y-Tasks` was not available at that time) and `FExplorer` [8] for accessing the file system.

### 3.6 Avoiding dependencies

Dependencies to the activities of outside parties were a major cause for unexpected delays for us. In a Symbian project, it is crucial to minimize external and internal dependencies. If the only provider of a component fails, equivalent components may be impossible to find for the relatively exotic environment.

The most typical external dependencies in our project were the ties to parties who, for some reason, had delays in providing required software components. An example of this was our need for a software library from an outside vendor in 2004-2005. Due to the vendor's unexpected delay, the component came some 12 months late for integration with `PnPAP`.

Another dependency-related challenge in Symbian is the lack of standard components that are available on other platforms, decreasing the choices from which solutions can be selected. In our project, we lacked the familiar `C++ Standard Template Library (STL)` for Symbian OS and had to use Symbian's data-container classes instead. `Penrillian` [10] has recently released a `STL` library for Symbian OS, supporting the `Leave` mechanism and compatibility between descriptors and `C++ strings`; this is a welcome novelty. Another new tool is Nokia's `Open C` [9] for porting standard `C` libraries to Symbian OS, reducing dependencies to Symbian-specific code.

## 4 Conclusions

Symbian OS software developers who are facing similar challenges as those analyzed in this case study may apply the provided guidelines to keep their projects' complexity on a tolerable level. In Table 1, all the guidelines are summarized and their Symbian-specific benefits are emphasized.

Table 1. Summary of the guidelines

Guideline	Benefit for Symbian development
Using a non-platform-specific solution may often be a better choice than relying on the platform's proprietary API.	The project's dependency on the future changes of the Symbian OS API will decrease. Studying earlier Symbian OS versions reveals that changes have often been substantial.
Design the basic infrastructural components to be generic enough.	Reuse of existing code is beneficial: on Symbian OS, programming is relatively demanding and limiting the size of the codebase is favorable.
In protocols and file formats, generally use out-band signaling rather than in-band signaling.	On a mobile platform with scarce computational resources, out-band signaling makes data parsing less complex and more efficient.
Exercise the option to implement a feature by using a task-specific language only with care.	Creating an interpretation environment for a complex task-specific language may be challenging on Symbian OS.
Map temporal sequences in code execution to an easy-to-follow structure.	Visualized sequence logs are especially useful on mobile devices with limited means of indicating the progress of a program's execution.
Use a logging facility that enables logging the actions of interrelated processes as a single sequence; access to shared logs must not involve concurrency-related issues.	See above.
Keep a human-readable log of your software development activities for your own reference.	Sometimes problems with Symbian OS force the developers to use complex solutions in the code. Personal logs may later provide information that is not available anywhere else.
Use assertive indication of details for the developer.	The developer is more confident about the details of the running code; this reduces the uncertainty and errors that may result from the complexity of Symbian software development.
Take into account the possibly indirect consequences of the shortcomings of the platform.	The behavior of Symbian OS may sometimes lead to situations that are difficult to foresee. If the problems are identified, they can often be reacted to.
Minimize your project's external and internal dependencies.	If a component provider fails, a replacement can be especially hard to find on Symbian.

## References

1. B. Forstner, L. Lengyel, I. Kelenyi, T. Levendovszky, and H. Charaf. Supporting rapid application development on Symbian platform. In *Proceedings of the International Conference on Computer as a Tool*. IEEE Press, 2005.
2. B. Forstner, L. Lengyel, T. Levendovszky, G. Mezei, I. Kelenyi, and H. Charaf. Model-based system development for embedded mobile platforms. In *Proceedings of the Fourth and Third International Workshop on Model-Based Development of Computer-Based Systems and Model-Based Methodologies for Pervasive and Embedded Software*. IEEE Computer Society, 2006.
3. E. Harjula, M. Ylianttila, J. Ala-Kurikka, J. Riekki, and J. Sauvola. Plug-and-play application platform: Towards mobile peer-to-peer. In *Proceedings of the Third International Conference on Mobile and Ubiquitous Multimedia*. ACM Press, 2004.
4. A. Hunt and D. Thomas. *The pragmatic programmer: From journeyman to master*. Addison-Wesley, 2000.
5. E.S. Raymond. *The art of Unix programming*. Addison-Wesley, 2003.
6. S. Rothkugel. *Towards middleware support for mobile and cellular networks: Core problems and illustrated approaches*. Ph.D. thesis. University of Trier, 2001.
7. Symbian D-EXC, panic code logger. Available: <http://developer.symbian.com/>. Cited 13 Sept 2007.
8. Symbian FExplorer, file explorer. Available: <http://www.gosymbian.com/>. Cited 13 Sept 2007.
9. Symbian Open C. Available: <http://www.forum.nokia.com/>. Cited 27 June 2007.
10. Symbian Standard Template Library. Available: <http://www.penrillian.com/>. Cited 29 Jan 2008.
11. Symbian TaskSpy, task manager. Available: <http://www.pushl.com/taskspy/>. Cited 13 Sept 2007.
12. Symbian Y-Tasks, system monitor. Available: <http://www.drjukka.com/>. Cited 30 Jan 2008.
13. K. Wang. *Post-mortem debug and software failure analysis on Symbian OS*. M.Sc. thesis. University of Tampere, 2007.