ContextLogger2

- A Tool for Smartphone Data Gathering

Tero Hasu
HIIT Technical Reports 2010-1

Tero Hasu
Helsinki Institute for Information Technology HIIT
Aalto University
PO Box 19800, 00076 AALTO, Finland
Tero.Hasu@hiit.fi

Version 1.0, finalized 26th August 2010.

Helsinki Institute for Information Technology HIIT
HIIT Technical Reports 2010-1
ISBN 978-952-60-3323-5 (electronic)
ISSN 1458-9478 (electronic)

ContextLogger2
A Tool for Smartphone Data Gathering

Tero Hasu

Helsinki Institute for Information Technology HIIT
Aalto University
PO Box 19800, 00076 AALTO, Finland
tero.hasu@hiit.fi

Abstract

Researchers often want to gather activity-related data available on mobile phones. Based on our experience with data collection in numerous field studies, we identified a number of requirements for a smartphone data logger, and set out to create logging software fulfilling those requirements. Our software—named ContextLogger2—now comes close to meeting the goals set for it. Furthermore, it has portable technologies in its core, potentially serving as a basis for logger implementations for a variety of platforms.

KEYWORDS: context, data gathering, smartphone

1 Introduction

A mobile phone tends to be intimately associated with its owner, hence providing a useful window into the owner's behavior. This has motivated a number of tools capable of capturing and making available information regarding user activity on the phone and the context of the phone. A *context logger*, as we define it, is a software tool that captures such information automatically and unintrusively on the background.

To avoid getting on the way of normal operation of the phone, typical requirements for a context logger include autonomy, robustness, and low resource consumption. These requirements alone are a challenge to logger developers, and the situation is further aggravated by the fact that gaining access to detailed information about a system tends to require deep system integration, exposing the developer to unfriendly low-level systems programming languages (usually either C, C++, or Objective-C) and APIs, as well as variability between systems.

We have implemented *ContextLogger2* (CL2), a context logger for Symbian OS. The Symbian platform is implemented mostly in C++, and its native API overall is of a relatively low level of abstraction. Symbian based smartphones have been on the market for many years, and we attempt to support a number of different generations of the OS, spanning dozens of different devices, many with model-specific firmware customizations and a number of firmware releases.

In implementing a context logger we faced the challenges imposed by the platform and the stringent autonomy, robustness, and unintrusiveness requirements. In this report we discuss the technologies chosen and technical decisions made in order to address those challenges. We also cover the overall design of ContextLogger2, the functionality it provides, and issues relating to its deployment in real-world user trials.

2 ContextLogger2

ContextLogger2[1] (CL2) is a software tool for smartphone data collection, created primarily for research purposes. The software is available as open source, and runs on off-the-shelf mobile phones based on Symbian S60 3rd or 5th Edition.

Symbian OS is natively written in a language commonly referred to as "Symbian C++", which is valid C++, but with non-standard APIs and idioms. Symbian has been designed from the ground up for resource-constrained, long-running systems, and these design goals are a good fit with those of a context logger. Symbian coding standards [14] advice developers to optimize code prioritizing for reliability, size, and speed, in that order, and these priorities also align with ours in implementing CL2.

CL2 consists of a logger "daemon" and a set of supporting programs and libraries. These components together form a system capable of automatic and unintrusive recording of information regarding the use and context of its host device.

The set of data sources (that we informally refer to as "sensors") is configurable, and can include GPS and GSM cell ID readings, keypress times, application focus changes, etc. The daemon that observes the sensors and logs sensor events is automatically started at boot, and runs in a background process without direct control by the user. It supports uploading of collected data to a server at configured intervals.

CL2 is in some sense a successor to ContextPhone [12], which runs on Symbian S60 1st and 2nd Edition, and also includes logging functionality. However, CL2—unlike ContextPhone—is not a *framework* for context-aware applications. While CL2 does reuse some ContextPhone code, its sole focus is on logging rather than supporting other applications. This specialization has helped in keeping down the size of the codebase, albeit at the cost of general utility.

One way in which CL2 *does* achieve wider applicability is in that its codebase is only partially tied to the Symbian platform. While the sensor code is Symbian C++, the core of the daemon is portable C with few dependencies on non-standard libraries. C was chosen as the implementation language as this avoids difficulties in bridging with the native (C++) APIs, allows for a high degree of control over resource consumption, and avoids a dependency on laborious-to-port runtimes such as Java ME.

The core of CL2 only has a dependency on a small subset of the widely available GLib library. We might have chosen to use C++ and its standard library, but there are platforms for which the standard library is not available (Symbian was one of them for a

[1] www.contextlogger.org

long time), and with C++ there are also questions as to which subset of the language is actually supported by the compiler that is available for a given embedded platform.

Granted, attempting to make context logger type software portable is not particularly fruitful, but even partial portability is helpful to developers, allowing some of the system to be tested natively on the development machine. This flexibility does come at the cost of increased complexity in configuration management, however, but we deemed it worth it in at least in the Symbian case, where the test/debug cycle on an actual device can be quite time consuming due to platform security restrictions in software installation.

3 Requirements

Based on previous experience in using context loggers (such as the one in the ContextPhone application suite), the following requirements were specified for ContextLogger2: [8]

1. Capability for non-intrusive, persistent, and robust data logging with updates to a server with configurable intervals.
2. Data compression and perhaps encryption.
3. Access to necessary services and resources in the phone as enabled by existing software such as ContextPhone.
4. An XML-based logging format (or some other easily deployable format).
5. Researcher terminal. A terminal for remotely: (1) monitoring logging of data; (2) triggering recording and data transfer events; and (3) configuring data logging parameters on an individual user's phone.
6. A server for all above that is able to handle a user trial with at least 200 users.
7. Optionally, a simple interface for a researcher to examine logged events on a timeline and perhaps "playback" user's actions on the display and events in the context.
8. "User experience", i.e., easy to install and configure on a user's phone and from server-side.

The present CL2 implementation meets most of the above requirements, with some exceptions: there is data encryption (in transit), but no compression; there is remote monitoring of logging, but the data cannot be directly queried remotely (an upload request can be initiated remotely); the optional requirement for a timeline interface for examining logged events has not been implemented; and the "user experience" has been found adequete given sufficient training, but could be further streamlined. The CL2 server setup has not actually been tested with 200 simultaneous users, but neither does CL2 rely on any custom server application that might cause scalability problems.

4 Features

The core features of ContextLogger2 are:

- Non-intrusive, persistent, and robust data logging.
- SQLite3[2] based logging format.
- Uploads to a server at configurable intervals as an HTTP POST (SSL optional).
- Local and remote monitoring and control facility.
- Scalable if the servers are (we are using Apache[3] and ejabberd[4]).
- Easy to install and configure (except when things go wrong).
- A number of "sensors" available for Symbian, including:
  - Focused application change
  - Keypress times
  - Profile change
  - GSM cell ID change
  - GPS sampling
  - Call status
  - SMS receival/sending
  - Bluetooth proximity scan
  - Inactivity
  - Message from another application
  - etc.

4.1 Features for Developers

CL2 features of interest to developers include:

- Builds for Linux are supported, albeit with lots of functionality missing. This is to allow for more convenient testing.
- Named build configurations are supported by the build system. This is to help developers achieve repeatable builds.

[2] www.sqlite.org
[3] http://httpd.apache.org/
[4] http://www.process-one.net/en/ejabberd/

- CL2 includes an integrated Lua[5] programming language runtime. This allows for use of dynamic and mobile code, and gives developers the option of implementing some functionality in Lua.

5 Log File Format

The logged data is internally stored and also uploaded in SQLite3 format. The use of a relational database engine and its associated file format (instead of "flat" text files, as typically used for logging in the Unix environment, for instance) is due to our desire to support random access on-device queries to the data.

At the same time, our choice of data format comes with some degree of support for introspection and backward compatibility. Using SQLite3 tools it is possible to query for the database schema of a given database file, and it is possible to ask for specific fields by (table column) name; new fields appearing later do not break "parsing" of data files, as no particular order is assumed when querying for fields by name.

Support for on-device data queries is only partially implemented at present. The primary motivation for such a feature is to offer peace of mind to researchers by allowing them to remotely check (through the remote control facility) that data is being collected as expected. If this feature was unnecessary, we would likely indeed have opted to just append logged data to a text file, probably achieving better performance. As we now are using SQLite3 format for logging, we also use it as our transport format, to avoid burdening the mobile device with unnecessary work on data conversion.

As to the data contained in the database, we generally prefer to record data in its original, raw form, and to look for suitable representations and interpretations later, during analysis. Adhering to this philosophy we are less likely to perform potentially lossy conversions that we might later regret.

Naturally raw data tends to be platform-dependent, or rather API dependent. If a given API is available on multiple platforms, and we source our data through that API, then likely the acquired data will also have the same semantics across the supported platforms.

5.1 Sample Data: Status Screen Indicators

While the contents of SQLite3 data files are not directly printable, their content can be dumped using the `sqlite3` command-line tool. For instance, the command

```
sqlite3 log.db \
  "select datetime(unixtime, 'unixepoch'), value, caps \
   from indicator_scan;"
```

might output something like

```
2010-01-02 13:17:46|3|7
```

[5] www.lua.org

```
2010-01-02 14:40:10|2|7
2010-01-02 16:15:37|0|7
2010-01-02 16:17:32|2|7
2010-01-02 16:35:55|0|7
2010-01-02 16:36:19|2|7
2010-01-02 16:42:09|0|7
2010-01-02 16:42:24|2|7
2010-01-02 16:43:48|0|7
2010-01-02 16:43:52|2|7
2010-01-02 16:49:36|0|7
2010-01-02 16:49:37|2|7
2010-01-02 16:52:40|0|7
2010-01-02 16:52:42|2|7
2010-01-02 17:04:23|0|7
2010-01-02 17:04:42|2|7
2010-01-02 17:12:51|0|7
2010-01-02 17:13:20|2|7
2010-01-02 17:17:41|0|7
2010-01-02 17:17:57|2|7
2010-01-02 17:18:27|0|7
2010-01-02 18:10:38|2|7
```

This is an extract of actual CL2-collected data, where the second value encodes the phone "Home" screen indicators as a bit field. The data was collected during a train ride, explaining the frequent toggling of the network availability indicator (i.e., the indicator value changing between 2 and 0). The fairly raw data at least in this case (where each entry is essentially a set of three integer values) results in the log entries being quite compact, although we do no actual compression.

5.2 Sample Data: Call Status

```
2010-01-05 10:10:58|3|+35850301****|Antti Lindqvist|||
2010-01-05 10:11:16|4|||||
2010-01-05 10:11:16|6|||||
2010-01-05 10:13:46|8|||2010-01-05 10:11:16|0|0
2010-01-05 10:13:46|1|||||
2010-01-06 13:53:56|3|(suppressed)||||
2010-01-06 13:54:08|8||||-2|-8090
2010-01-06 13:54:08|1|||||
```

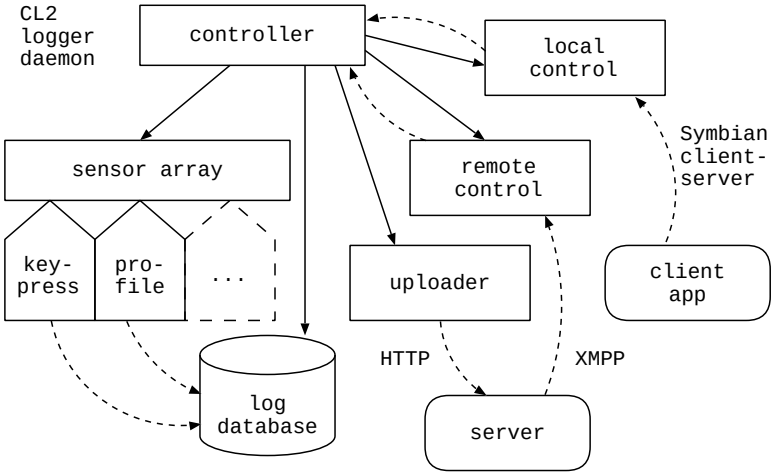
Above is an extract of "call status" log data, showing information regarding line status changes. A single phone call typically has multiple status changes, and this is the kind of detailed information about the progress of each call that is not available from the "Log"

application of the phone. Particularly notable are the call termination reason codes, in this case `0|0` and `-2|-8090`, which provide the phone's and the network operator's idea of the reason for call termination.

When analyzing call status logs, something to keep in mind is that CL2 occasionally misses some status events due to the real-time requirements of handling phone calls taking a priority over logging. Indeed, CL2 prioritizes unintrusiveness over completeness of the data.

6 System Architecture

The CL2 logger daemon architecture is pictured in the diagram below.



- The *controller* manages the components that make up the logger daemon. It also handles some overarching concerns such as failures in critical components or lack of resources (e.g., disk space on the logging medium), typically by simply shutting down the daemon.

- *Sensor array* manages the individual sensors that have been compiled in to the logger. Sensors may be individually started and stopped. Some sensors also have dynamic configuration options; the satellite-based positioning update interval can be adjusted at runtime, for example. A current limitation of the sensor array design is that there is no "tuple space" type intermediary data storage and propagation system that would facilitate the implementation of "logical" sensors.

- *Log database* manages the database file that contains the logged data. The functionality is based on the SQLite3 database engine, which we ported for Symbian as it was unavailable.[6] In-memory databases worked more or less as is, but filesystem support required some porting effort.

[6]It seems a shame that while current Symbian OS based devices include Symbian SQL, an SQLite3-based database engine, the native SQLite3 C API is not exposed in the OS. Recent Symbian documentation suggests that the Symbianˆ3 platform release will expose the SQLite3 API.

- *Uploader* is responsible for uploading log database files at configured times. To receive uploads, we are using a short PHP script. (A sample PHP program showing how to receive log files can be found from the CL2 website.[7]) Our script is hosted by Apache, which should be able to handle a large number of users, especially if uploads are infrequent. Should there be a very large number of users, one might consider configuring slightly different upload times for each.

- *Local control* is a component presently implemented only for Symbian OS. It provides access to the logger control facility, which is based on the Lua runtime and a set of Lua bindings to some of the "internal" logger APIs. Any communication is initiated by client applications, and the communication is based on the Symbian client-server architecture.

- *Remote control* makes use of the same control facility as the local control component, but instead receives commands via the XMPP protocol. Due to limitations of mobile networks, the logger daemon must be connected to the (configured) Jabber server in order for clients to send commands. We are intending to address this issue by implementing SMS-based triggers for connection initiation. For the time being, until such triggers have been implemented, the remote control feature is not really ready for use in the field. However, it already has its uses in testing as it allows developers to issue commands and queries to the logger programmatically directly off the PC, or at the very least using a proper keyboard.

The diagram omits some details, such as the components used to access and maintain information about the static and persistent configuration of the logger. Also, the diagram does not show the supporting libraries and components, which include:

- A *key event scanning library* named `keyevents`. The library was adopted from JaikuEngine Mobile Client[8], and consists of a Symbian animation DLL and an associated server. The logger includes two variant implementations of the keypress sensor, and this library is strictly required for the more robust implementation.

- *CL2 C++ client library*. Provides a C++ API for controlling the logger via its local control facility.

- *CL2 Python client library*. A Python wrapper for the C++ based logger control API.

- *Configuration wizard*. A Python for S60 script intended to make it easier to set up CL2 on a "fresh" phone. Not intended for the end user, and no icon for launching the program is provided.

[7] http://contextlogger.org/upload.php.html
[8] http://code.google.com/p/jaikuengine-mobile-client/

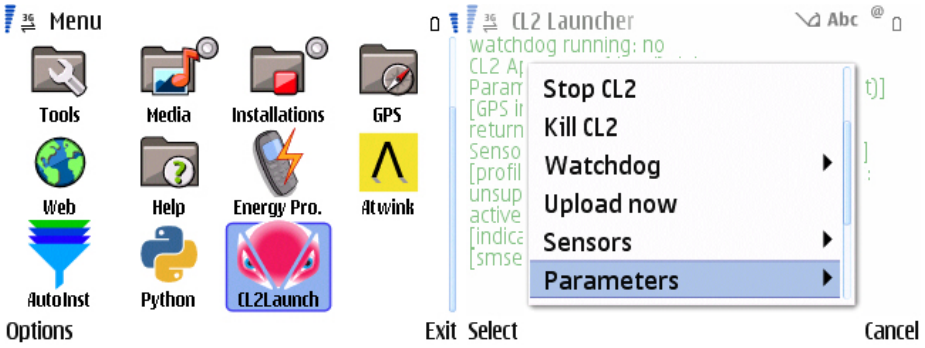


Figure 1: Launcher.

- *Launcher*. A Python for S60 application that serves as a "control panel" for the logger. May be used to launch and stop the daemon, and to set select configuration parameters, for example. We feel that although this application is very useful for developers, it is not well-suited for providing end users with control over logging (for instance to allow them to opt out of being logged in certain situations). We are considering implementing a simpler and flashier application for that purpose.

- *Watchdog*. A small and simple "pure Symbian" application for starting up and restarting the logger daemon. This program is registered with the system for startup at boot, and exists for the sole purpose of observing the logger daemon process and keeping it running. If the logger process dies, the watchdog waits for a little while before attempting to relaunch it.

6.1 Configurability and Logging Control Facilities

Given the goal of unintrusiveness, CL2 was designed to be both configurable and automatically controllable. The configuration and control facilities provide means to avoid having to ask the user anything.

There are a number of configuration mechanisms in the logger, namely: compile-time configuration, static configuration (in a configuration file), and dynamic configuration (maintained in a persistent database). Compile-time configurability exists to make it possible to optimize the software for a specific trial. Static configuration makes it possible to have "unchangeable" configuration parameters, primarily for reasons of security; it is not possible to remotely change the address of the server via which remote control happens, for instance. Dynamic configuration, in turn, is important in making it possible to enable/disable battery-hungry sensors as desired, for instance.

Despite all of these configuration facilities, it has never been our idea that the user should perform complex configuration. Possibly there could be an interface for the user to see whether the logger is running, to stop and restart it as desired, and perhaps even toggle the use of power hungry sensors. Otherwise, our approach to logger control is to have optimization and configuration done pre-trial as much as possible, perhaps performing

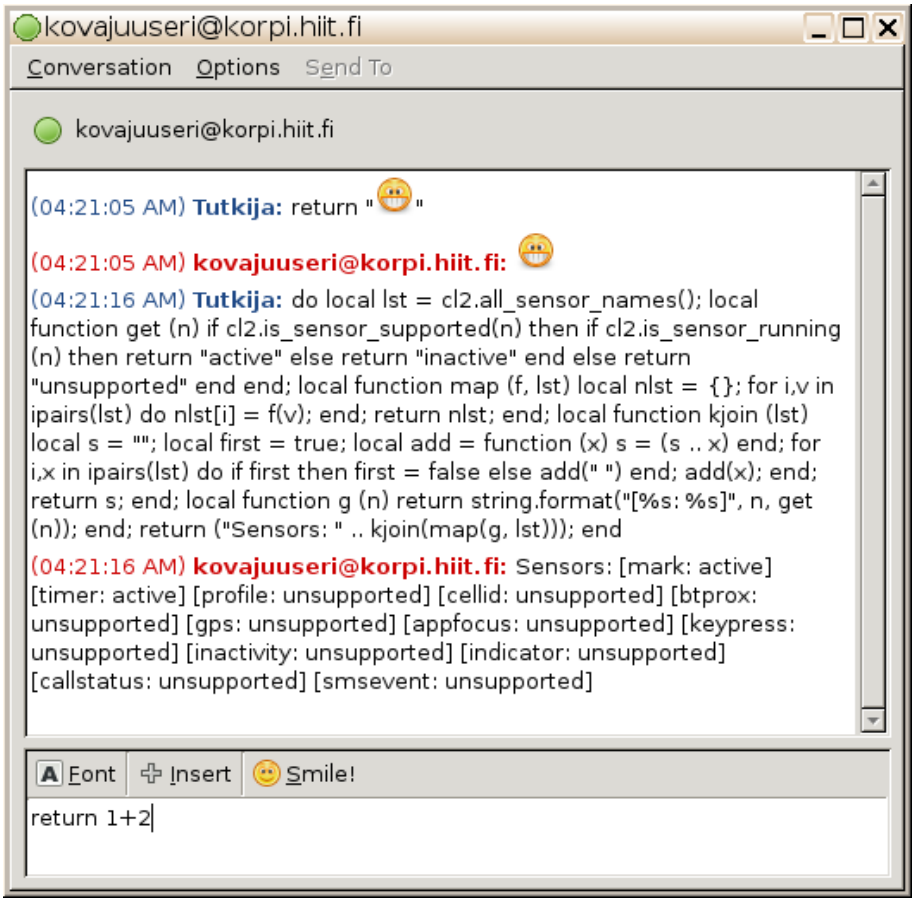


Figure 2: A remote chat session with the logger daemon.

some automated dynamic control from local and/or remote applications, and to have researchers rather than users (remotely) observe and control the logger as required. Just about any XMPP-supporting chat client functions as a “researcher terminal”.

7 Implementation Technologies

In implementing the logger daemon we used (and in some cases created) a number of technologies that warrant a mention.

7.1 Portable Languages and Libraries

In choosing programming languages and libraries for the project, it was clear from the start that we were going to have to write at least some of the code in Symbian C++, targeting Symbian APIs. This is because (despite Symbian having been around for many years) the platform has not enjoyed sufficient popularity among developers to have comprehensive language binding coverage for any other language. We elected to use C++ for

the Symbian-specific code to avoid having to deal with language integration issues, and to let sensor developers have the most direct access to the platform APIs.

For implementing platform agnostic functionality we decided to favor portable technologies for a "write once, target everything" experience. We made this decision to allow for partial testing of the system on development machines, and also to avoid having to recode the entire system if we one day wanted to support another target OS.

We should note that we do not consider large and complex virtual machines as portable technologies. For instance, while Java ME is the only programming option for many "closed" mobile phones, it is rather hard to port to platforms for which it is not available. Rather, we decided to focus on smartphones (which by our definition do allow for the installation of native software), and assume the availability of a C compiler. The availability of an SDK with a vendor-provided C compiler and at least some of the standard C library tends to be the norm also for embedded platforms, even if not much else can be taken for granted when targeting less popular platforms.

Following our decision to favor portability, we proceeded to implement the CL2 core functionality in ANSI C. We are also using a small subset of POSIX and GLib APIs. Both APIs are relatively widely supported, and Nokia's Open C libraries offer good coverage of both of them.

The decision to have the core system be C code does not necessarily preclude the use of higher-level languages. In another internal Symbian project we tried out the Vala[9] programming language without problems. Vala is a language resembling C#, but it compiles down to C and relies on GLib for its runtime library.

With CL2 we instead decided to try the GOB2[10] preprocessor, which essentially just generates the boilerplate code for defining GObject-style APIs, while letting the actual functionality be defined in plain C. A side benefit is that one need not maintain separate interface (`.h`) and implementation (`.c`) files for an API. Again, we did not encounter any problem in using GOB2, but neither did we end up using it extensively as we presently have no particular requirement to define GObject-style APIs in CL2. That said, taking advantage of the introspection facilities of the GObject object system might be one option for making C objects scriptable in Lua. (The information available for introspection *is* useful, but not necessarily sufficient, which is why there is a GObject Introspection[11] effort for maintaining introspection data.)

We are indeed using Lua internally in CL2, but thus far we have created any required bindings manually. A pleasant surprise for us was that Lua was trivial to port to Symbian OS, requiring little else than setting some configuration options in the Lua compile-time configuration file. Lua is written in ANSI C, making use of some of the C standard library. The primary motivation for adopting Lua was that we wanted some way to specify commands to the logger remotely. By choosing a full-blown dynamic language interpreter for the purpose we ensured that the solution would almost certainly be flexible enough for any command that we could conceive. Security was another important con-

[9] http://live.gnome.org/Vala

[10] http://www.jirka.org/gob.html

[11] http://live.gnome.org/GObjectIntrospection

sideration, and Lua makes it easy to instantiate a runtime that has no access to even the standard Lua APIs, making it easy for us to choose what APIs we would allow access to remotely.

SQLite3 was slightly harder to port in that the code in the filesystem layer did require some porting effort, but apart from that we have been satisfied with the library. SQLite3 is coded in ANSI C, generally relying only on memory related functions from the C standard library; more platform specific code is cleanly encapsulated within internal interfaces.

While SQLite3 and Lua are commonly used on embedded platforms, and famed for their high-quality code, a lot of other C libraries have their problems when brought over to a platform such as Symbian. As many libraries are originally developed for environments with plenty of memory and/or *optimistic memory allocation* (a scheme in which allocations succeed without there being a guarantee of the memory actually being available when accessed), there tend to be shortcomings in handling out-of-memory (OOM) situations cleanly. In choosing third-party libraries for use in CL2 implementation we ruled out any libraries that do not provide at least some mechanism for OOM handling.

SQLite3 diligently checks for OOM errors, and in most cases reports them by returning with the value `SQLITE_NOMEM`. Lua uses its own exception mechanism to report OOM errors. In our remote control component we are using the iksemel[12] XMPP library, which by itself does not comprehensively check for OOM errors. It is not difficult to find seemingly unsafe iksemel code such as the snippet shown below. However, the safety of the shown code depends on the semantics of the `iks_malloc` function.

```
ret = iks_malloc (IKS_CDATA_LEN (x));
memcpy (ret, IKS_CDATA_CDATA (x), IKS_CDATA_LEN (x));
```

As iksemel does provide a hook for defining custom behavior for `iks_malloc`, it can perhaps be regarded as an OOM-safe library, even if it is not immediately obvious. To achieve safe behavior, one may have `iks_malloc` cause program termination or a non-local return upon a memory allocation failure. CL2 presently does neither, and as a result, we cannot yet regard the remote control component as robust. We are still debating whether to work on hardening our current remote control implementation or to switch to a different technology. The downside with the current implementation is that it uses a custom fork of iksemel that provides an asynchronous interface (which we deemed a better fit for the logger architecture), but for reduced maintenance effort it would be better to adopt an XMPP library that is designed to be used asynchronously (e.g., QXmpp[13]).

7.2 Build Configuration Management System

For repeatable builds, the CL2 logger build system supports uniquely named build configurations. Each build configuration is specified in its own file in the `variants` directory, and a build "variant" specification essentially consists of a set of (key, value) pairs. Such

[12] http://code.google.com/p/iksemel/

[13] http://code.google.com/p/qxmpp/

fixed, named configurations are in contrast to the widely-used autotools (autoconf[14] et al) system, in which configurations are host-specific by design.

Logger configurations are specified through a Scheme language based, object-oriented API. To avoid duplication, a configuration can inherit from another one and override values. It is also possible for one configuration setting to depend on another one to help avoid ending up with a nonsensical configuration. The predefined base configuration has a few options for enabling/disabling individual features, and disabling a feature generally means that the code implementing it will not even be included in the binary. The idea here is that if one only includes the features required for a given trial, the logger process will be more lightweight and there is less to go wrong.

The configuration system is implemented in PLT Scheme, which has recently been renamed as Racket[15]. We chose to use Racket as: it includes an extensive library featuring flexible APIs for relevant functionality such as command-line parsing and invoking external programs; Racket's module system [9] makes it easier to keep the sometimes surprising complexity of build systems in check; and Racket's powerful macro system [4] and other language extension facilities make it possible to define custom build configuration definition syntax if desired. A downside of Racket is that program startup times are quite long (which is a problem with many VM-hosted languages), but we do not consider this a significant issue as switching between build configurations tends to be infrequent.

When the configuration system is invoked, naming a build variant, it generates include files for a variety of languages. The generated files are imported from C and C++ source files as required, and are likewise used by a variety of build scripts to adapt for the current configuration. CL2 Symbian builds presently support the native Symbian ABLD toolchain, whereas Linux builds are based on GNU Make.

As found by Kantee et al [5], portable software development can involve the use of multiple build systems, leading to maintenance overhead and potential errors due to build settings being out-of-sync. Our configuration system addresses one aspect of this problem in allowing each configuration to be specified only once. It does not, however, directly address the specification of exactly what must be built and how in order to produce an executable corresponding to a given configuration.

7.3 Component Dependency Management System

The CL2 logger is a sizeable application, with a number of configuration options. Building it may result in the compilation of up to 150 source files (approximately, not counting header files), depending on the chosen configuration. For linking, one must also specify all the system and third-party DLLs that the application uses.

To address such complexity, we believe it would be beneficial to have a component dependency management system which allows one to treat a set of source files as a logical whole (a "component"). The system should allow one to specify, for each component, in a modular and reusable way: (1) how to build the component (from which source

[14] http://www.gnu.org/software/autoconf/

[15] www.racket-lang.org

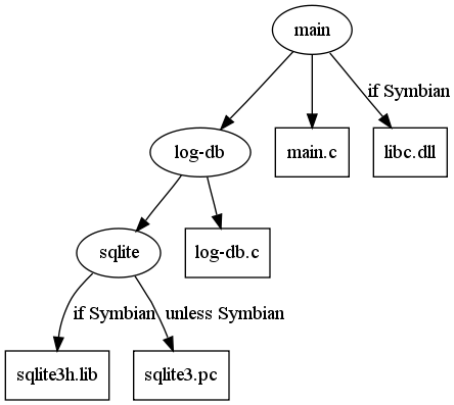


Figure 3: An imaginary build dependency scenario, albeit inspired by CL2. Ellipses depict named components, arcs depict (possibly conditional) dependencies, and boxes depict things that "belong" to a component.

files and with what build options); and (2) what other components it has as dependencies. Figure 3 visualizes a small example scenario in which the build information for the `main` component can be deduced by traversing a tree of dependencies. The dependencies may be conditional on the current build configuration.

We have started sketching out a dependency management system matching the above description, and do have an initial, rather primitive design and implementation. The component specification format is unfriendly, and there is only limited and experimental adoption of said specifications in the logger codebase. Our current implementation is based on Racket, making it possible for the system to directly read Scheme-based build configuration files. It is clear that our current implementation would require more work in terms of features and convenience before we would consider fully adopting it. We are also considering alternative designs that better leverage existing work.

The system we envision could probably also be implemented in terms of an existing build tool. GNU Make, for example, has support for dependency resolution built in, and it also supports conditional expressions. In considering a Make-based implementation there are, however, two caveats to keep in mind: (1) to achieve modularity of specifications, one must split build information to one makefile (or Make include file) per component; and (2) there are a variety of native build systems [5] not driven by Make, and "simulating" these in terms of Make requires additional maintenance effort.

If the native (vendor provided) build systems are reliable and provide the desired functionality, it is perhaps more practical to generate input files for them rather than dictate that the programmer write Make rules for building binaries for all targets, hence requiring considerable knowledge about the native toolchain of each target platform. The Qt cross-platform application framework's `qmake` tool, for instance, takes the approach of generating platform-native makefiles based on Qt project files. This is also the approach taken by our current system.

Implementing a component management system in terms of an existing *makefile com-*

piler such as qmake and CMake[16] would give us cross-platform support "for free", but we have yet to explore the feasibility of such an approach.

A "traditional" way to split systems software into components is as platform-native libraries, whether statically or dynamically linked. A DLL is the more self-contained option, with linking information included. Furthermore, there already is a fairly popular (particularly in the GNOME community) tool, namely `pkg-config`[17], for the purpose of finding out how to build and link against a DLL. One defines the relevant information for a library in a `.pc` file, and `pkg-config` can then be used to deduce relevant build options. Unfortunately, Symbian ABLD does not directly support invoking tools such as `pkg-config` to dynamically collect build information, nor is the output of the tool compatible with ABLD `.mmp` files. Also, `pkg-config` is rather geared towards dynamic linking, and we have found DLLs problematic when targeting Symbian.

Symbian has long had limitations regarding the use of static writable data in libraries and applications, and this has traditionally been one of the biggest issues when porting applications to Symbian. More recent releases of the platform do allow static writable data in applications and optionally in DLLs, but due to a bug in the compiler shipped with S60 SDKs, with a typical S60 SDK setup it is still not possible to have writable data in a DLL. (Applications and static libraries do not have this restriction.) DLLs also have associated runtime overhead. Indeed, we decided that requiring that each CL2 component be a DLL would be impractical. For a flexible solution a "logical" component must be able to take on some other concrete form as well.

There is existing work on component systems such as Knit [13] and Koala [3] that likewise do not require DLL-based components. These systems, however, are focused on component composition, and deal with component information down to the level of individual exported and imported symbols. Although we consider this overkill for CL2 at present, we might consider the adoption of such a system provided it was suitably mature, and offset the cost of component interface specification by automatically generating C or C++ header files from component descriptions.

7.4 Code-Generation for Cross-Cutting Concerns

The CL2 logger implements a number of sensors for Symbian, and each sensor has a number of cross-cutting concerns that must be reflected in a number of places in the CL2 codebase. We address this through code generation of most of the database access layer and sensor management code related to each sensor. We specify the information required by the code generator in a mostly declarative manner, in a custom *domain-specific language* (DSL), a sample of which is shown in Figure 4.

[16] www.cmake.org

[17] http://pkg-config.freedesktop.org/

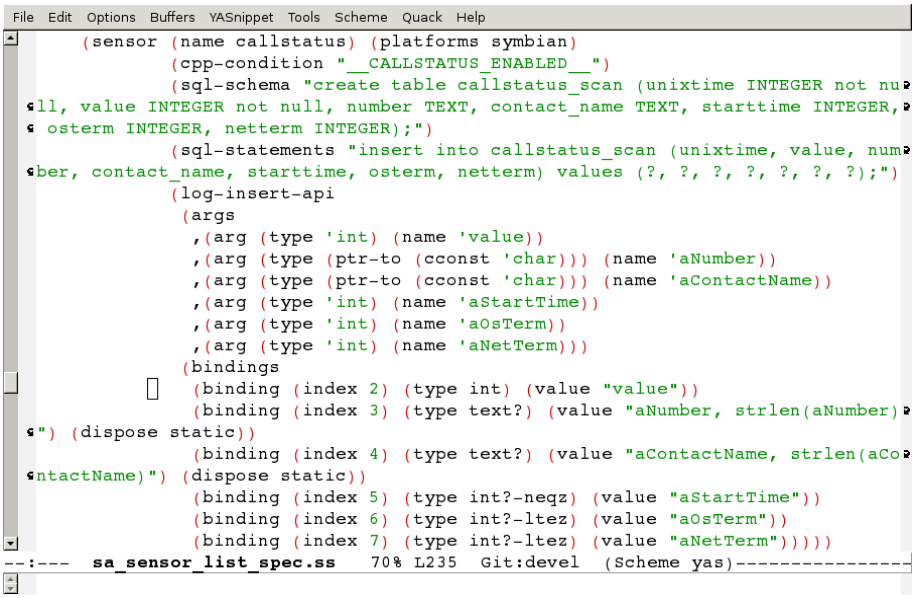


Figure 4: Specifying the "callstatus" sensor in a domain-specific language.

7.5 Ragel State Machines for Asynchronous Event Handling

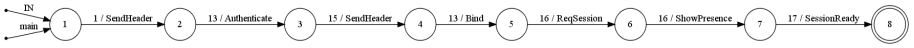
The CL2 logger is implemented as an event driven system, and this approach is well suited for a context logger; to avoid impacting battery life, one wants to just make OS requests and do nothing until the OS responds. Symbian OS is also event driven throughout, and its idiomatic way to represent individual tasks is as *active objects*, i.e. objects that maintain their own state, and are collectively driven by a built-in thread-specific active scheduler. Unlike multitasking solutions based on multiple threads of control (e.g., coroutines or preemptive threads), Symbian-style multitasking can result in complex state machines that obscure the flow of control.

As an experimental way to achieve readable control flow and "free" state tracking, we included an explicit state machine description in the remote control session establishment code of the logger. The actual state tracking and switching code was then generated with the Ragel [16] state machine generator, which can generate (among other languages) ANSI C without dependencies. In this case the end result was a rather trivial, linearly progressing state machine to track protocol state. The machine does not run to completion in one go, but rather its progress is driven by the triggering of asynchronous events corresponding to requests made in the state machine actions.

```
protocol = connected >SendHeader
           features_tag >Authenticate
           success_tag >SendHeader
           features_tag >Bind
           iq_tag >ReqSession
           iq_tag >ShowPresence
```

```
presence_tag >SessionReady ;
```

The control flow of the Ragel-based protocol description (shown above) is clear, and it is also possible to automatically render a graphical representation of same (shown below). There is no need for hand-written `switch` statements for state tracking.



The one issue that our use of Ragel does not resolve, however, is that of *stack ripping* [1], which means that any state on the stack must be short-lived due to the requirement of frequently returning control to the event loop. Moreover, the programmer must manually split (or "rip") each function that might block, as well as its ancestors in the call stack. [6] In our case we retained the required state in the state machine state structure, and encoded each of the "ripped" function pieces as state machine actions, which in Ragel results in somewhat more lightweight syntax than defining a separate C function for each. In Ragel-generated code all the pieces end up in the same function, with their selection for execution governed by a `switch` statement.

Event system driven, formally defined state machines are not new. For instance, the Qt State Machine framework [11] is designed to support state transitions triggered by asynchronous events. The difference between our use of Ragel and the Qt framework is that in the Qt case state machines are defined in C++ and constructed at runtime.

There is also work on addressing the stack ripping issue. Tame [6], for example, is a solution for C++ that frees the programmer of manual stack management, overcoming language limitations by including both a C++ library and a source-to-source translator. There is also work on making it easier to implement Tame-like solutions through the use of extensible compiler technology. An example of such technology is Xoc [2], an "extension-oriented" C compiler for which a Tame-like extension has been written.

7.6 Future Work in Technology Adoption: Tools-Assisted Mocking and Qt Integration

CL2 makes use of libraries (such as Lua, SQLite3, and iksemel) that are—to a large extent—platform agnostic, due to their data processing oriented nature. Some functionality, however, inherently requires a platform-specific implementation (due to platform-specific hardware drivers, service protocols, file formats, etc.), and CL2 mostly accesses such functionality via Symbian-specific libraries. This increases the cost of testing and maintenance. Even where a portable implementation of a library providing a required service is impossible, that does not mean that separate implementations of a library could not expose the same interface on multiple platforms.

In many cases porting an API to another platform may be impossible (due to lack of hardware peripherals, for instance) or at least a lot of work, but one could consider substituting unportable components with *mock objects* [15] during testing. Such objects (components that to some extent simulate the behavior of "real" components) are presently not included in the CL2 codebase, however. There are tools (such as Google

Mock[18]) to assist in their creation, but one might want to first look for existing libraries that come "pre-mocked".

The Symbian platform is presently moving to Qt for its application framework, likely making it necessary to integrate CL2 with Qt at least for Symbian^4 platform release. This—in our opinion—is a welcome development, as there is a selection of Qt cross-platform APIs [10] for accessing phone services (e.g., services providing access to contacts and location information and hardware sensors). There is also a simulation solution for Qt that customizes some Qt libraries to include interactively configurable and scriptable mock objects for select phone APIs. To take advantage of Qt APIs in developing sensors for CL2, one should merely need to modify the build scripts and switch to the Qt event loop instead of the Symbian one. This is because the Qt event loop has been integrated with Symbian so that it can also handle event dispatch for Symbian native events.

8 Deployment

For software that may get used in larger user trials, it is quite important for deployment to be relatively straightforward, lest the installation consume a prohibitively large amount of time. When installing on people's personal, existing phones, there also tends to be emotional pressure to get the installation done quickly and professionally, as each person will typically be observing the installation process and waiting to get his or her phone back.

We have thus far polished the deployment process enough for it to be workable in smaller (20 people or so) trials, even for non-technical deployment personnel with a small amount of training and "technical support" a phone call away. Had we done larger trials, we would have likely ended up streamlining the process more.

Ideally, we would just send the user a link to the software, and that software would come fully configured for the specific user, and automatically install itself and any required libraries and configuration files. We are not close to this ideal; a currently supported deployment procedure is outlined below.

The lack of streamlining in the deployment process is not all due to lack of effort, as Symbian platform security is also to blame. For example, we found out the hard way (during a trial) that due to Symbian restrictions any watchdog cannot be installed as an embedded SIS (Symbian Installation Source) file if it is to get started up at boot. This restriction means that it is difficult[19] or impossible to (1) package everything into a single SIS file and to (2) keep the watchdog separately (un)installable; one must choose one or the other.

[18] http://code.google.com/p/googlemock/

[19] We have not confirmed this, but it might be possible to have a SIS file unpack another SIS file containing the watchdog, and then, during installation, launch a program that would "directly" install the unpacked watchdog SIS file, thus working around the restriction.

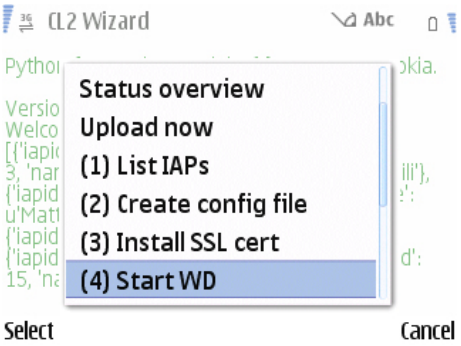


Figure 5: Configuration wizard.

8.1 Suggested Deployment Procedure

Below is one possible CL2 deployment procedure that we have found to work in practice. Note that this procedure does not consider setting up remote control, nor very large trials making it necessary to get the software signed by Symbian to lift IMEI code restrictions.

1. Set up the web server and write some PHP code to receive logs.
2. Collect the IMEI codes of the phones and create a Symbian code-signing developer certificate covering the codes.
3. Do a trial-specific build, and sign it with the newly created certificate.
4. Install the software as SIS package files onto users' devices. This can be done without incurring data transfer costs for instance by sending the files from another phone over Bluetooth.
5. Configure user-specific settings with a wizard. The wizard also supports installing a server-specific SSL certificate as required.
6. Launch the logger and do an upload to verify installation. Here it is useful to have some convenient way to check what the most recent uploads to the server are, perhaps via a web interface from the researcher's phone.

8.2 Things That Can Go Wrong

There are a number of things that can go wrong during deployment, as we have found. We list some of these below, but the problem of course is that there tend to also be problems that one fails to anticipate.

- Incompatible phone. Not everyone knows exactly (and correctly) what phone model they have, let alone whether it runs Symbian or not.

- Platform errors or pre-existing data corruption. At one point, for instance, we encountered a phone on which the App. manager would crash at launch, making it impossible to install any software.
- Lack of resources. For example, not enough storage space, or no memory card.
- Developers' lack of knowledge. An example of this was given above, namely that of embedded SIS executables not getting started up at boot.
- Misconfiguration. E.g., incorrect access point, or wrong boxes ticked for an SSL certificate during interactive installation. Even if one realizes that one has made a mistake, one may not immediately know how to fix that mistake. (For instance, an SSL certificate cannot be reinstalled without first removing it, and the removal function is not that easy to find on S60.)
- Transient network errors. One may think that one has misconfigured the system, when in fact the problem is due to a network malfunction.
- "DLL hell". An incompatible version of a required library having already been installed on the phone. There may also already be applications installed that rely on the incompatible version, and indeed it may be that the incompatible library got installed with such an application.[20]
- Some software must be uninstalled before upgrade. The Symbian installer is such that in some cases only a "clean" install will succeed, and installing on top of an older version will not result in a working installation, even if it does not get immediately noticed.

8.3 Future Work for a More "Product-Like" Experience

There are some aspects of CL2 that still reveal its research prototype like nature. Taking the actions listed below to achieve a more "productized" experience would also ease deployment.

- Remove dependencies to underutilized libraries (EUserHL, stdcpp, etc.). Each dependency makes deployment harder, as there is no dependency manager on Symbian (unlike on Maemo, for instance). Due to the more research-oriented aspects of the project, we quite happily tried quite a few libraries, not all of which ended up being used much in the end.
- Get an SSL certificate signed by a proper CA. Currently we are using self-signed certificates on our servers, meaning that it is necessary to install a custom certificate on each phone to enable SSL encryption.

[20] For example, Nokia Maps 3.04 beta got released with Open C/C++ 1.7, but Open C/C++ 1.7 turned out to break binary compatibility with version 1.6, and was later withdrawn by Nokia. Worse yet are libraries that make no attempt to preserve binary compatibility across versions.

- Create a proper "control panel" application with a decent look-and-feel, with functionality chosen with the end user in mind. Perhaps the only functions required would be: Display status, Start logger, Stop logger, and Disable/enable autostart (at boot).

9 Related Work

There are a number of smartphone context information scanning solutions that typically focus on (1) logging, (2) streaming data to web services, or (3) serving as middleware for local context-aware applications. In Table 1 we list software packages that include some context data collection facility, regardless on whether the collection happens locally (with possible batch uploads) or by streaming to a server. We leave out "pure" middleware that by themselves do not collect data.

As seen from the table, ContextLogger2 is the only free open source software (FOSS) context logger offering for modern Symbian devices that offers tight system integration (in the sense that it: executes natively; starts up and runs on the background automatically; and supports direct access to native APIs without language interfacing issues). ContextPhone, for example, has not been ported to S60 3rd Edition, whereas BeTelGeuse [7] does not get launched automatically at phone boot time.

10 Conclusion

We have created ContextLogger2, a smartphone data collection tool that runs on most Symbian S60 based phones shipped to date. The software is freely available, with most of the code covered by the MIT license. There are few restrictions regarding use and redistribution.

The software has been found to be quite robust, and a degree of effort has been put into facilitating testing. CL2 is partially portable, allowing for instance for interactive testing of the core system and unit testing of individual portable components on the desktop.

CL2 is being actively maintained, with near-term plans for improvement including a remote control SMS triggering facility and additional sensors (e.g., phone call recording).

In this technical report we have outlined the functionality, architecture, and implementation of CL2. We have shared our experiences with trying out technologies for which we identified a need in the creation of software of this kind. We have pointed out that some of these needs arise merely because of limitations of the Symbian platform, and hence do not concern context logger development in general.

Acknowledgements

This work has been supported by the Academy of Finland projects Helsinki Privacy Experiment and Context Cues and the HIIT project ContextLogger 2.0. We thank: Antti

Name	Platforms	Language	Availability	Features
BeTelGeuse	Java 1.3, MIDP 2	Java, Python	FOSS	$\approx$10 phone sensors + support for external sensors over Bluetooth + higher-level context inference.
CAES	PocketPC	C++	OSS (unmaintained)	Experience sampling.
ContextLogger2	S60 v3–up	C, Symbian C++	FOSS	$\approx$10 sensors, autostart, background operation, watchdog, batch uploads, scripting in Lua.
(ContextPhone) ContextLogger	S60 v1–2	Symbian C++	FOSS	$\approx$10 sensors, autostart, background operation, watchdog, batch uploads.
IYOUIT	S60 v2–3	Python	free service	$\approx$10 sensors, background operation, social networking.
(Mobile Context Toolbox) Logger	S60 v3	Python	OSS by request	$\approx$10 sensors, autostart.
MyExperience	WinMo v5–up	C#	FOSS	$>$50 sensors, autostart, batch uploads, context-triggered experience sampling, scripting in XML and Simkin.
Nokia Simple Context	S60	Python	private beta	
Nokia SP360	S60 v2–3	Symbian C++	private	Many sensors, background operation, batch uploads.
RECON	WinMo v5–up	C#	private	Experience sampling.
SocioXensor	WinMo v5–up	(.NET)	private	$\approx$10 sensors, batch uploads, experience sampling.
(Zokem) Mobile Media Tracker	Android, Blackberry, iOS, S60, WinMo	Java, C++	commercial	$>$30 sensors, background operation, batch uploads, context-triggered experience sampling.
(Zokem) Zoki	Android, Blackberry, iOS, Java ME, S60 v3–up, WinMo	Java, C++	free service	Social networking.

Table 1: A summary of various smartphone context data collection software.

Oulasvirta for initiating the project and for helpful suggestions regarding this report; Antti Lindqvist and Antti Salovaara for organizing real-word user testing, and providing requirements and feedback; Mika Raento for a variety of advice; Petteri Nurmi for information regarding BeTelGeuse; Hannu Verkasalo for information regarding Zokem products; and Elina Du for assisting with prototyping.

References

[1] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In Carla Schlatter Ellis, editor, *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, pp. 289–302. USENIX, 2002. ISBN 1-880446-00-6. doi:http://www.usenix.org/publications/library/proceedings/usenix02/adyahowell.html.

[2] Russ Cox, Tom Bergan, Austin Clements, Frans Kaashoek, and Eddie Kohler. Xoc, an extension-oriented compiler for systems programming. In *ASPLOS 2008*. March 2008.

[3] M. de Jonge. Multi-level component composition. In Jan Bosch, editor, *2nd Groningen Workshop on Software Variability Modeling (SVM'04)*, 2004-7-01. Reseach Institute of Computer Science and Mathematics, University of Groningen, December 2004.

[4] Matthew Flatt. Composable and compilable macros: You want it when? In *ICFP*, pp. 72–83. 2002. doi:http://doi.acm.org/10.1145/581478.581486.

[5] Antti Kantee and Heikki Vuolteenaho. Experiences in portable mobile application development. In Sergio F. Ochoa and Gruia-Catalin Roman, editors, *IFIP 19th World Computer Congress, First International Workshop on Advanced Software Engineering, Expanding the Frontiers of Software Technology, August 25, 2006, Santiago, Chile*, volume 219 of *IFIP*, pp. 138–152. Springer, 2006. ISBN 978-0-387-34828-5. doi:http://dx.doi.org/10.1007/978-0-387-34831-5_11.

[6] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *Proceedings of the 2007 USENIX Annual Technical Conference*. June 2007.

[7] Joonas Kukkonen, Eemil Lagerspetz, Petteri Nurmi, and Mikael Andersson. BeTelGeuse: A platform for gathering and processing situational data. *IEEE Pervasive Computing*, 8:49–56, 2009. ISSN 1536-1268. doi:http://doi.ieeecomputersociety.org/10.1109/MPRV.2009.23.

[8] Antti Oulasvirta. ContextLogger 2.0 project description. Unpublished, 2008.

[9] Scott Owens and Matthew Flatt. From structures and functors to modules and units. In John H. Reppy and Julia L. Lawall, editors, *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pp. 87–98. ACM, 2006. ISBN 1-59593-309-3. doi:http://doi.acm.org/10.1145/1159803.1159815.

[10] Qt Mobility Project 1.0: Qt Mobility Project APIs overview, 2010.

[11] Qt 4.6: Qt reference documentation, 2010.

[12] Mika Raento, Antti Oulasvirta, Renaud Petit, and Hannu Toivonen. ContextPhone: A prototyping platform for context-aware mobile applications. *IEEE Pervasive Computing*, 4:51–59, 2005. ISSN 1536-1268. doi:http://doi.ieeecomputersociety.org/10.1109/MPRV.2005.29.

[13] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *OSDI*, pp. 347–360. 2000.

[14] Symbian Developer Network. *Symbian OS C++ coding standards*, January 2003.

[15] Dave Thomas and Andy Hunt. Mock objects. *IEEE Software*, 19(3):22–24, 2002. doi: http://www.computer.org:80/software/so2002/s3022abs.htm.

[16] Adrian D. Thurston. Parsing computer languages with an automaton compiled from a single regular expression. In Oscar H. Ibarra and Hsu-Chun Yen, editors, *Implementation and Application of Automata, 11th International Conference, CIAA 2006, Taipei, Taiwan, August 21-23, 2006, Proceedings*, volume 4094 of *Lecture Notes in Computer Science*, pp. 285–286. Springer, 2006. ISBN 3-540-37213-X. doi:http://dx.doi.org/10.1007/11812128_31.

Researchers often want to gather activity-related data available on mobile phones. Based on our experience with data collection in numerous field studies, we identified a number of requirements for a smartphone data logger, and set out to create logging software fulfilling those requirements. Our software — named ContextLogger2 — now comes close to meeting the goals set for it. Furthermore, it has portable technologies in its core, potentially serving as a basis for logger implementations for a variety of platforms.

Helsinki Institute for Information Technology HIIT
Tietotekniikan tutkimuslaitos HIIT (in Finnish)
Forskningsinstitutet för Informationsteknologi HIIT (in Swedish)

The Helsinki Institute for Information Technology HIIT is a joint research institution of Aalto University and the University of Helsinki for basic and applied research on information technology.

Its research ranges from fundamental methods and technologies to novel applications and their impact on people and society. HIIT's key competences are in Internet architecture and technologies, mobile and human-centric computing, user-created media, analysis of large sets of data and probabilistic modeling of complex phenomena.

HIIT works in a multidisciplinary way, with scientists from computer, natural, behavioural and social sciences, as well as from humanities and design. The projects are conducted in collaboration with universities, companies and research institutions.

http://www.hiit.fi

HIIT Technical Reports 2010-1

ISBN 978-952-60-3323-5 (electronic)
ISSN 1458-9478 (electronic)