Hedgehog - a high level functional language for rapid development of embedded, soft real time telematic machine-to-machine communication software

Table of Contents

1 Disclaimer 1

2 Introduction 1

2.1 Embedded applications (2p) 1

2.2 Telematic applications and machine-to-machine communication (2p) . . . 4

2.3 Wireless communication (3p) 6

2.4 A brief history and background of Oliotalo (2p) 8

3 Embedded software development 11

3.1 A typical Oliotalo system (2p) 11

3.2 Beaver: the Oliotalo server platform (1p) 13

3.3 Sparrow: the Oliotalo communications protocol (2p) 13

3.4 Target platform: Aplicom (2p) 15

3.5 Target platform: BlueGiga (1p) 16

3.6 Problems in the development process for embedded applications (1p) . . . 17

3.7 A solution to the problems (1p) 19

4 Hedgehog: the original system 21

4.1 Initial prototyping in Python (1p) 21

4.2 The first C versions (1p) 22

4.3 Real use: the garbage compressor (1p) 23

4.4 Garbage collection improvements (1p) 24

4.5 Real use: the liquid container watcher (0.5p) 25

4.6 Simulating the box on a workstation (0.5p) 25

4.7 Mature, but with persistent problems (1p) 26

5 The next generation: the need for speed 27

5.1 A compiler and byte code interpreter (1p) 27

5.2 Achieving speed and small size (0.5p) 28

5.3 Language improvements (1p) 29

5.4 Profiling support (1p) 31

5.5 Library developments (2p) 32

5.6 Application structure: state machines, not threads (2p) 33

6 Application distribution 35

6.1 The logistical problem (1p) 35

6.2 Version management (2p) 35

6.3 Appdist server: centralized application distribution (3p) 37

7 Language design philosophy 40

8 Lessons learned (3p) 41

9 References (1p) 42

1 Disclaimer

I originally wrote this as an excercise to see how much (crappy) text I can produce per hour. I then cleaned it up some, but it is still **highly** crappy. It is mostly available publically to show some of the motivations I had to design and implement Hedgehog Lisp. Don't rely on anything said in this document.

2 Introduction

This chapter gives an introduction to embedded software, telematic applications, and wireless communications. It describes the history and background of Oliotalo, and its employees, because this has an impact on which problems we look at, and how we like to solve them.

2.1 Embedded applications (2p)

The first microprocessor, the Intel 4004, was developed in 1971. It was an extremely simple 4-bit CPU, but was quickly followed more powerful models, especially the 8-bit Intel 8080 in 1974. These processors were still quite feeble compared to modern ones, but they were good enough for large numbers of simple embedded applications. Microprocessors quickly found themselves as controllers for microwave ovens, digital watches, washing machines, etc.

Many embedded applications are still so simple that processors developed in the late 1970's or early 1980's are more than powerful enough. Microwave ovens are a good example: the task of controlling a microwave oven has not changed significantly since 1980. Development has not stood still, however, and much more powerful processors are now available for embedded developers. They are used for tasks where the slow processors are inadequate, or where the greater power allows more implementation options.

There are roughly two classes of embedded applications: mass market and custom. Mass market implies huge numbers of units produced; custom applications tend to produce a relatively small number of units. This means that mass market applications can spend more on research and development to get unit price down, since even a small difference per unit results in large overall savings.

Mass market embedded applications typically strive to use the cheapest processor available. Cheapest usually means slowest. Slow processors require more careful programming, using lower more level tools. Typically they are programmed in assembly language, or the C language. This causes development costs be high.

It is also typically difficult or expensive to service mass market products. Even a minor flaw causes many units to be replaced or fixed, which is quite expensive. To prevent this, more effort needs to be spent on testing. Exhaustive testing is expensive, but still less expensive than replacing a million microwave ovens.

Custom embedded applications may often use somewhat more expensive hardware than mass market applications, if this brings down development costs. For custom applications, development and testing costs are often a significant part of the total system cost, unlike for mass market applications.

It is sometimes economically possible to service or upgrade custom applications. For example, there might be only a few dozen installations, all within easy reach of service technicians. This reduces to stigma of a bug, and makes it possible to save on testing costs.

Custom applications also tend to suffer more from requirement changes during deployment. They have a tendency to require not just bug fixes, but feature updates. Existing, installed units need to have their software upgraded. For a mass market application, changed requirements would mean developing a whole device. If your microwave oven needs to start supporting defrosting in addition to heating, it means you need to buy a new microwave oven,

rather than upgrading the old one's software.

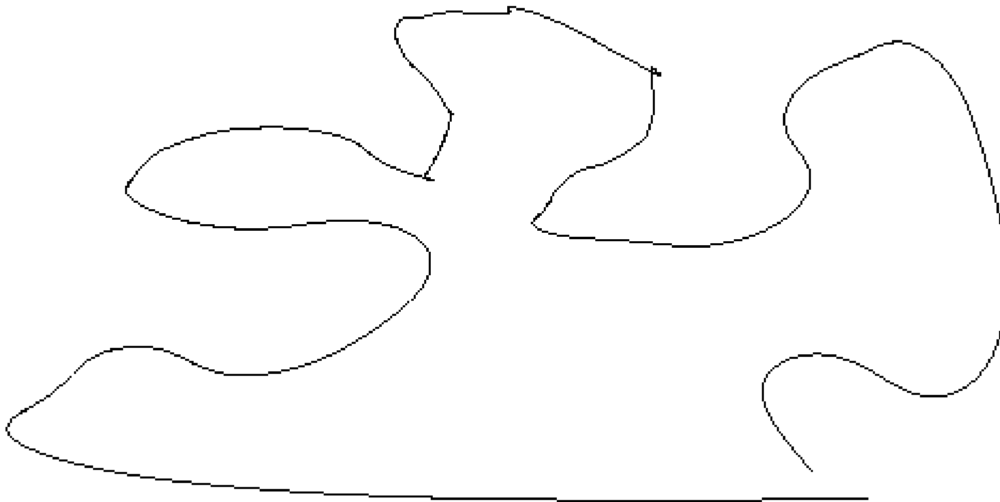

Figure: Memory sizes and CPU speeds in embedded applications as a function of time. E.g.: 4004, 8008, 8080, 8086, 80186, 8051, 6802, 68000, PPC, ARM, VIA/x86

Table: Different kinds of embedded applications.

	Mass market	**Custom telematic**
Installed units	millions	thousands
Development budget	large	small
Development time	sufficient	too short
Testing budget	large	small
Required reliability	severe	medium
Service contract	no	yes
External communication	maybe	yes
Complexity	small to large	medium
Hardware speed	kHz to MHz	tens of MHz
Memory space	1-100 kB	hundreds of kB
Estimated age	years to decades	years to decades
Installed system changes	extremely rare	routine

2.2 Telematic applications and machine-to-machine communication (2p)

For the purposes of this document, a telematic application is one where devices are monitored remotely. The term is often used about cars and other vehicles only, but our use is broader.

Even a simple telematic system can be worthwhile to its users. The primary reason for this is that they enable doing things that are otherwise impossible, often due to economic concerns. Often the telematic system allows operational savings by allowing more efficient logistics.

We will now discuss two example applications, which are typical real business cases for Oliotalo: garbage compressors and Lamborghini cars. The purpose of presenting these examples is to give light on the requirements Oliotalo's systems typically have, which in turn has some influence on the design decisions made while developing Hedgehog.

The garbage compressor system remotely measures how full garbage compressors are. A garbage compressor, sometimes several, can be found in the yard of many shops and factories. The users put garbage in them, and the machine compresses the garbage into a smaller space. Eventually the compressor needs to be emptied. This is done by a waste disposal company using a large truck.

Adding a telematic system to garbage compressors helps improve logistics for emptying the compressors and for maintenance when the compressors break. By monitoring remotely how full the compressors are, the waste disposal company is able to avoid driving a truck to a compressor that is not full. Driving large trucks around is expensive, so this easily results in direct savings. If the telematic system can also inform the service department when the compressor seems to be out of order, customer satisfaction is upwardly adjusted. More importantly, when the customer calls in to report a malfunctioning garbage compressor, the service technician can check via the telematic system whether, say, the customer has

inadvertently pressed the emergency stop button. This reduces maintenance costs by reducing the number of unnecessary service visits.

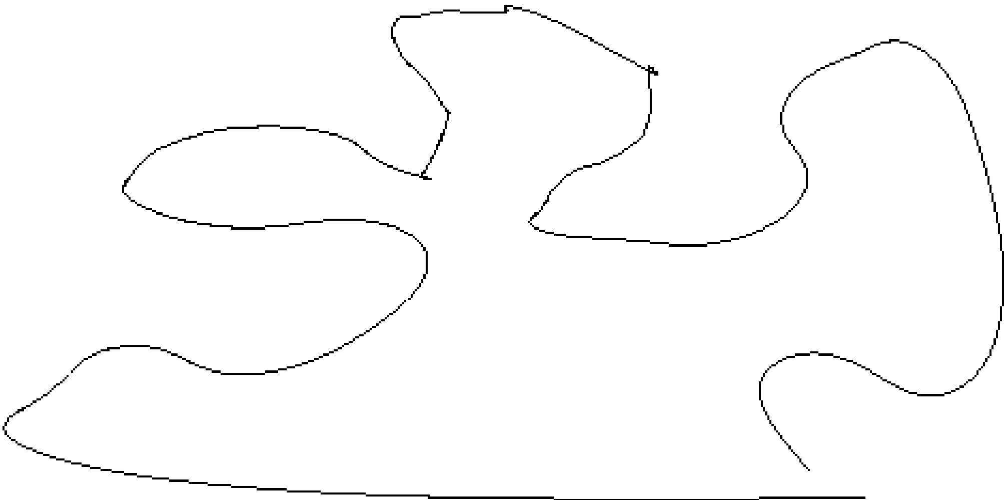

Figure: Garbage compressor telematic system.

Lamborghini is an Italian manufacturer of expensive fast cars. They provide world-wide service for their cars, which is a somewhat costly operation. By equipping their cars with a telematic system, the service technician can make a preliminary diagnosis remotely, and can bring the necessary parts with him. Without a remote diagnosis, he either needs to bring with him all possible parts, which is costly, or order the parts after making a diagnosis on-site, which is slow.

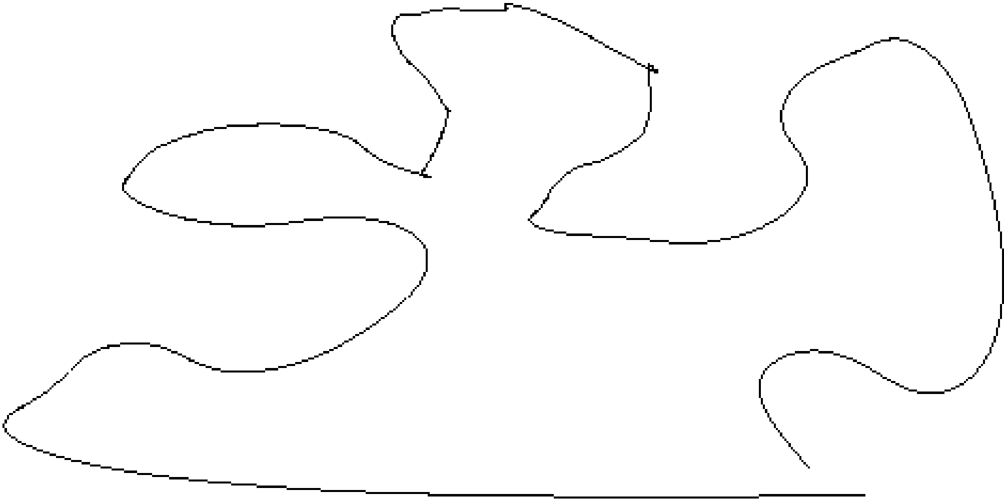

Figure: Lamborghini telematic system.

2.3 Wireless communication (3p)

The systems built by Oliotalo typically require wireless communication. In some cases it might be possible to use wired communication, such as a modem over the plain old telephone system, but often this is not practical even for immobile applications. Setup costs for such connections tend to be high enough that a wireless solution is not significantly costlier, but is much easier.

There are many wireless communication technologies available on the market today, as off-the-shelf products. See table XXX for a summary. They provide different features, cost structures, and availability in different locations. For each project, it is necessary to choose the appropriate communication technology. The tools used for embedded application development must not unduly restrict the choices.

Table: Wireless communication technologies.

	CSD	**SMS**		**GPRS**	**WLAN**	**Bluetooth**
Communication costs	?	?	?	?	?	
Hardware costs	?	?	?	?	?	
Theoretical speed	?	?	?	?	?	
Practical speed	?	?	?	?	?	
Reliability	?	?	?	?	?	
Availability	?	?	?	?	?	
Datagrams? (UDP)	?	?	?	?	?	
Data streams? (TCP)	?	?	?	?	?	

In Europe, the most commonly available wireless network is the GSM mobile phone network. It provides three important ways for data communication: circuit switched data, SMS messages, and GPRS. All three work in most of Europe, and many other countries. Coverage

is usually pretty good, in that all or almost all parts of each country have coverage at least in theory, but the quality of the coverage varies a lot. For example, the ground floor might have good coverage, but the cellar not at all.

GSM circuit switched data, or data calls, is the equivalent of using a modem in the plain old telephone system. Speed is fairly slow, though usually quite fast enough for the systems Oliotalo builds. The connection is billed by time, since it reserves a channel in the GSM network cell the GSM modem is located in. Thus, the connection cannot be kept open constantly, and can be opened only during brief moments when there is a known need to communicate.

GSM SMS messages, or text messages, are small data packets that can be sent through the control channel in the GSM network. They can carry up to 140 bytes of 8-bit data. SMS messages are carried by special SMS centers, using a store-and-forward messaging model. Transmission of a message can be arbitrarily delayed, though it is guaranteed that a message is killed if it can't be delivered in time. Different billing schemes exist for SMS messages, ranging from a fixed monthly fee without a limit on the number of messages to each message costing a significant amount. Some operators have special billing schemes for telematic applications.

GPRS, short for General Packet Radio System, is an extension of the base GSM network to provide a packet network similar to the Internet. In fact, using GPRS an embedded device can be considered to be connected directly to the Internet, since PPP and TCP/IP are run on top of the GPRS network layer. GPRS speed can be fairly high, but it also varies much with the quality of the radio networking.

Wireless LAN, or Wi-Fi, is a small area wireless network that essentially replaces Ethernet cables. TCP/IP is run on top of it. Speed is tens of megabits per second, i.e., very fast for our telematic applications. Range is usually up to a few tens of meters. Wi-Fi networks require no licensing to be used, so they are good for building ad hoc networks. There is also no operator

billing for Wi-Fi. Wi-Fi equipment is somewhat cheap, but requires fairly much power.

Bluetooth is another technology for ad hoc wireless local area networks. It has a range of up to a hundred meters, reasonable speed, and it is designed to use little power and be cheap to manufacture.

There are also other wireless communication technologies that could be used, such as UMTS and Tetra, and non-GSM mobile phone networks in, say, the US or Japan. These are uninteresting to Oliotalo, since we operate in Europe for the time being, and the networks described above are sufficient for us. Our solutions are not, however, tied to the ones we have chosen to use now. On the contrary, we have tried to be as network neutral as possible.

As we have seen above, communication networks usually provide a datagram service, a data stream service, or both. Our application level protocols must adapt to any of these. We have chosen to use application level protocols based on datagrams in a store-and-forward network, i.e., the lowest common denominator. This has worked well.

It is sometimes necessary to use several wireless technologies at once for the same system. For example, to cut costs, there may be a need to have only one unit with a GPRS modem, and use Bluetooth between that and other units nearby, relaying messages between the two networks as necessary. In other projects it may be required to use GPRS when it is available, but fall back on SMS when necessary.

2.4 A brief history and background of Oliotalo (2p)

The Finnish company Akumiitti was founded in 1993. Originally, it worked with Internet technologies, but in 1997 it moved into mobile communications. Akumiitti did both entertainment (for example, ring tones for Nokia phones via SMS messages), and industrial applications in the form of telematics or remote monitoring of vehicles. During the IT boom of the late 1990s, Akumiitti grew fast.

In early 2001 Akumiitti split off its telematics department into a subsidiary called Akumiitti Telematics. The subsidiary grew somewhat, though not very fast since the IT boom was subsiding. Before the split, they had produced a system for Daimler-Chrysler in Germany for managing a fleet of trucks. With the split, there was a desire to move from doing customer projects to making products.

Akumiitti Telematics had a competent server platform, written in Java, but not a good embedded platform. There was a prototype one, written in C, but it was deemed not good enough. Wirzenius was hired in July, 2001, to develop it into a new one. It was developed to be a function library, plus some data types, that was linked into the actual application code. The library abstracted away some of the complications of the hardware platform and simplified application writing in other ways.

In early 2002 Akumiitti Telematics changed its name to Keko Technologies and participated with the new name in CeBIT. The two products, KEKO Server and KEKO Terminal, were exhibited and there was plenty of interest, though little concrete action. The business of selling products, especially expensive middleware, is characterized by long sales processes.

Later in the spring, there were attempts at porting the Terminal to a GSM modem with extra serial ports and digital inputs. This would have been an easier and cheaper alternative than running the Terminal and its application code on a separate embedded device, connected to a GSM modem over a serial port. Since the Terminal and the code that implemented the GSM protocol stack were running on the same CPU, whenever the Terminal was changed, the whole thing would have to be certified by the relevant GSM authorities. This is a slow and expensive process.

By making the Terminal configurable, so that the actual application would be implemented by loading some configuration values, the Terminal code itself would not change, even though the application was. The way to express the configuration values was not finalized, but an interpreted C-like language was discussed. This would have had obvious benefits for avoiding

continuous re-certification.

In June, 2002, Keko Technologies went bankrupt. Immediately afterwards, Keko’s CTO Kaius HÃ¤ggblom, started a new company, Oliotalo, and recruited five people from Keko’s R&D department. In the spring of 2003, one of these five would leave the company. As of this writing, the remaining four form Oliotalo’s development team.

Oliotalo’s first project was a measuring engine and chassis data for remote diagnosing of Lamborghini cars. This was a projects that originally started at Keko. It uses Aplicom’s C-series devices to read the Controller Area Network (CAN) bus in the car, measure various data, and report a compressed summary of these via SMS to the server. The user interface is via a web browser to the server.

The Lamborghini project had to be implemented in quite a hurry. Work started immediately in early July, and continued early September. During this time, we wrote a server and terminal software somewhat similar to those developed at Keko, but written from scratch. Again, the implementation languages were Java for the server and C for the terminal. Development went quite well. Partly this was thanks to having an engine simulator for the Lamborghini at our office; this made it simple to verify that our code was actually working.

After the Lamborghini project, Oliotalo has done several other projects. Due to the general economic downturn and other reasons, things have been hard economically. We have received funding from TEKES, a Finnish government agency for advancing technological progress, for developing our platform, and especially the embedded side has been in need of radical changes.

3 Embedded software development

This chapter describes the embedded software development process, from the Oliotalo point of view.

3.1 A typical Oliotalo system (2p)

A simple system built by Oliotalo has one or more embedded devices that communicate wirelessly directly to the server. This is a very simple architecture, and when it is sufficient, it is also a good one. In some cases, however, the cost of communication to the server may be an issue. For example, in a warehouse, there can be hundreds of fork lifts and it is fairly expensive to have each one communicate directly to the server over GPRS. GPRS modems cost money, and monthly GPRS account costs are also significant.

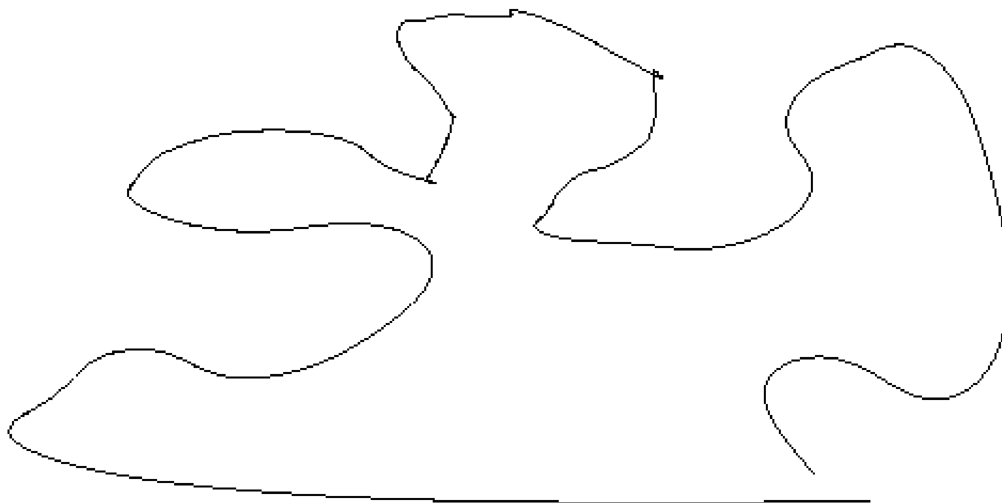

Figure: Simple application architecture.

When it is necessary, a more complicated system is built, where the embedded devices talk to a relay station, which then talks to the server. In the warehouse example, it can be possible to use Bluetooth between the fork lifts and the relay station, and GPRS between the relay station and the server. Bluetooth devices are cheaper than GPRS modems, and have no running costs.

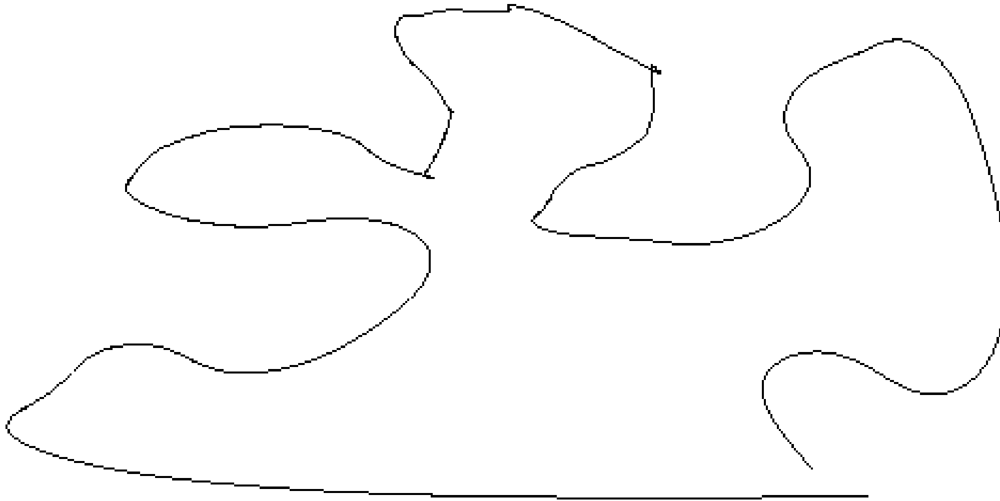

Figure: Architecture with relay station.

In either case, the same application level protocol is used. The devices in the fork lifts don’t know they’re talking to a relay station, instead of directly to the server. Similarly, the server doesn’t know about the relay station. Each device (terminal, relay station, server) queues messages appropriately and functions as a store-and-forward service for messages that aren’t meant for itself.

All connections are initiated by the end further away from the server. The fork lift connects to the relay station, and the relay station connects to the server. This is required, because the address of the client end may be variable: GPRS operators often use Network Address Translation for their GPRS networks so that the GPRS has one IP address within the GPRS network, and they all share the same address (but with mangled port numbers) towards the Internet at large.

The various parts of the system work together as follows. First, the bootstrap code in the device fetches the byte code for the application from a special byte code distribution server. The application starts running, and makes any measurements it is programmed to do. It then connects to the actual application server, and sends the measurement data. The server sends an acknowledgement. The embedded device resends as necessary, if it does not receive the acknowledgement. The server stores the data it receives in a database and discards any

duplicates it may receive. The user interacts with the system via a normal web browser, which connects to the server's user front-end.

The system operates almost identically if there is a relay station in use. The embedded device connects to the relay server, instead of the byte code distribution or application server. The relay server then connects to the relevant server and passes messages between the server and the embedded device. If necessary, the relay station queues messages, if the recipient is not on-line.

The server may have a need to send data to the embedded devices. For this, it keeps its own queue. When the embedded device connects (or a relay station, which proxies for the embedded device in question), the server sends the messages in its queue. When it receives an acknowledgement, it removes the message from its queue.

3.2 Beaver: the Oliotalo server platform (1p)

The Oliotalo server platform is called Beaver. J2EE. EJB. Persistent message queues. Databases for persistent storage. Web UI.

The server platform is not available as an independent product. Internal development tool. Hedgehog, too. Oliotalo does projects, not products.

From the Hedgehog point of view, the server is just a communication partner. Sparrow is the interface. Hedgehog doesn't care what Beaver does, just exchanges messages.

3.3 Sparrow: the Oliotalo communications protocol (2p)

The protocol Oliotalo uses over the air is called Sparrow. It works on top of various carrier wired and wireless protocols, such as SMS, GPRS, or Bluetooth, or over higher level protocols over these, such as TCP. Sparrow is by nature a store-and-forward datagram protocol, because this is what seems to work best with the various conditions in which

embedded applications have to work.

Sparrow is a generic protocol for encapsulating the data in the kinds of projects Oliotalo expects to do. The genericity allows code reuse at both ends when the protocol does not have to be implemented from scratch for each application. Less to keep track of for developers. The size penalty of the genericity is fairly small, so there is little need to make application specific protocols.

Genericity is achieved by dividing a Sparrow message into a header and a payload. Both of those are further divided into components, using the same encoding. Each component has an identifiers. Header identifiers have the same meanings for all applications, whereas payload identifiers are reused for each application. This gives us two 8-bit name spaces for components, which is enough. The payload can actually be of in formats as well, since its format is given in the header, in case some future application requires this.

The encoding of components is designed to be generic enough to suit all foreseeable situations, and compact enough to fit into our bandwidth limitations. An SMS message can contain up to 140 octets, and typically each message costs money, so avoiding bandwidth waste has a direct financial impact. The encoding gives each component a length, a type, and contents. Zero length indicates end of header; any further components will be the payload. Length is encoded using a variable width technique, to avoid wasting an extra byte or two when the typical case fits the length into seven bits.

Bandwidth is further conserved by using compact identifiers for embedded devices. Each message must identify the embedded device that sent it, since the server or relay station cannot deduce it from the address. Instead of using, say, the embedded device serial number (up to tens of characters), each device is assigned a 32-bit identifier by the server. The serial number is used by the embedded device for querying its short identifier. Further messages will use only the short identifier.

When an embedded device is built at the factory, Hedgehog is loaded into its flash memory. The device is generic, it contains no application. It does contain bootstrap code that will connect to the application distribution server, which returns the right application. This simplifies the logistics for application distribution greatly, since the embedded device manufacturer does not have to worry about which version of which application to load for each of its customers. The customer does not have to manually load software into the devices, which simplifies their manufacturing process also, and reduces the number of things that can go wrong.

When the application has been loaded into the device, it then connects to the application server, and asks for a configuration. The configuration includes, among other things, the short identifier it should use in further communication. With the configuration, the application is ready to do actual work.

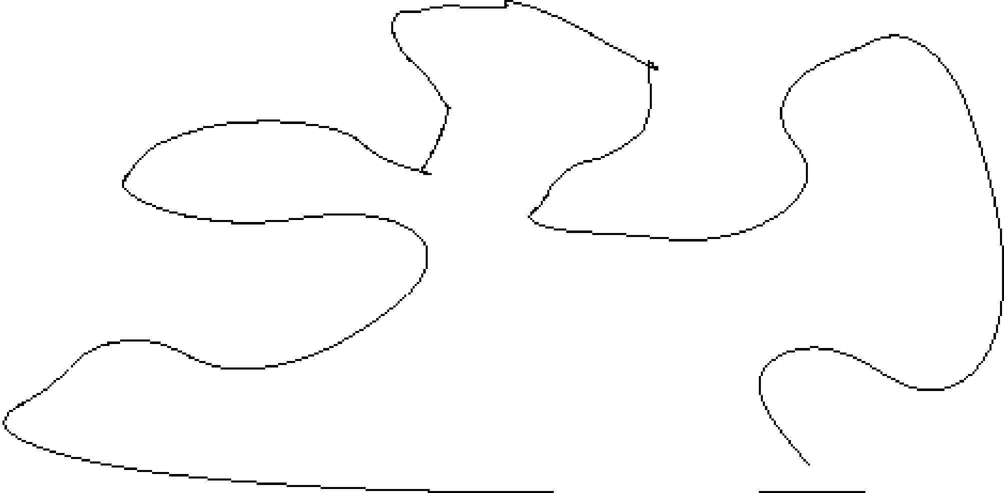

Figure: Timeline of protocol transactions.

3.4 Target platform: Aplicom (2p)

The first target platform for Hedgehog is the Aplicom C-series device. Aplicom is a Finnish manufacturer of embedded computers for installation in vehicles of various sorts. The device has plenty of interfaces, which makes it flexible enough to be suitable for many applications.

Its power requirements are fit for vehicles. It is designed for cars and trucks, but is not normally fit for more demanding environments, for example where there is a lot of water.

Table: Aplicom C series features.

CPU	ARM, 24 MHz
RAM	512 kilobytes
Flash	2 * 512 kilobytes
CAN bus	Yes
I2C bus	Yes
RS-232C	Three, but with all pins
Bluetooth	No
Ethernet	No

Aplicom provides an operating system and libraries for application development. The operating system supports multithreading and is soft real time. The libraries provide access to all hardware features, including all interfaces to other systems. The operating system and libraries are linked to application code. The application does not run as a separate process, but rather as one of many threads, some which are started by the libraries.

Applications are written in the C language. An expensive commercial compiler, running on Windows, must be used: the operating system is not ported to any other compiler. This makes development outside the office (at home or at a customer site) somewhat awkward, unless it is possible to buy several licenses or dedicate a laptop for running the compiler.

3.5 Target platform: BlueGiga (1p)

The secondary target platform for Hedgehog is a BlueGiga Access Server. BlueGiga is another Finnish company, specializing in Bluetooth devices and applications. The Access Server is a small Linux system. It has fewer hardware interfaces than the Aplicom one, but does have Bluetooth, which Aplicom does not have internally.

The BlueGiga device is used to relay messages between Aplicom devices and Beaver, when the Aplicom devices have Bluetooth instead of GPRS. This gives a solution that is cheaper overall than having each Aplicom device contain a GPRS modem. Instead, the Aplicom devices are equipped with an external "Bluetooth antenna", develped together by BlueGiga and Aplicom.

Since BlueGiga uses Linux, the applications are stored as files inside the device, and it is thus possible to easily update them. Also, the Linux system makes for a nicer target to develop for. However, since the BlueGiga device lacks some vital hardware interface for Oliotalo's applications, Aplicom is the primary target platform for Hedgehog.

Table: BlueGiga Access Server features.

CPU	ARM, 24 MHz
RAM	3 megabytes
Flash	1 megabyte
CAN bus	No
I2C bus	No
RS-232C	One, uses as console
Bluetooth	Yes
Ethernet	Yes

3.6 Problems in the development process for embedded applications (1p)

Developing embedded applications is in theory not different from other kinds of software engineering. The operating system tends to be a thinner layer on top of the hardware, so the applications are written closer to the hardware. Due to inefficient hardware, the programming language tends to be lower level than typical for modern applications. For example, assembly

language and C are the major languages. However, modern embedded devices are often somewhat faster than those of yesteryear, and there is some surplus power.

The low level of abstraction makes embedded development tedious. C much better in this regard than assembly language, but it is still very good. Both languages require the application programmer to care for many details, which higher level languages do automatically. The tedious nature of coding slows down software development and is a source of bugs. It amount of detail that has to be handled manually also makes it more difficult to find and fix bugs.

Debugging is additionally made difficult since the application code needs to run on the embedded device. Some, though not all, devices have corresponding emulators that run on development workstations, and these can help a lot. Often there is a need for a CAN bus, or other connections, which cannot be fully emulated on the workstation. Thus, ther eis a need to reduce the amount of time spent on debugging by making it easier, as well as reducing the number of bugs created.

It is generally accepted that a short iteration cycle between editing and executing code speeds up development. A typical cycle on a workstation, with an interpreted language, is very short: edit, run. For a compiled language, it is still very short: edit, compile, run. Developing an application in C for the Aplicom box is longer: edit, compile, load to flash, run. Not only is there an extra step, but it takes time, up to about twenty seconds. Worse, the flash loading sometimes failes, which means it has to be re-done.

Portability of embedded code is often bad. This makes it hard to switch hardware platforms. Different embedded devices often have very different operating systems, and application programming interfaces. Portability is often not a high priority, but it would sometimes be useful. For example, it might be good to be able to switch from an expensive device to a cheaper one, when the extra power or many I/O connections of the expensive one are not needed. Having to rewrite the whole application for this is unpleasant.

An additional problem comes when there is a need to fix a bug, or the customer changes the requirement specification. In this situation, there is a need to update the software in the embedded device. For typical devices, this means upload the compiled to the device over a serial line. It is quite expensive to visit each device. (Just catching a Lamborghini is difficult enough.)

3.7 A solution to the problems (1p)

The obvious solution to many of the problems described in the previous section is to use a higher level language. If it is also an interpreted language, it becomes possible to update the application over the air. During the autumn of 2002, Oliotalo decided to implement its own interpreted high level language.

Although we did not make an explicit requirement specification for the language, some things were discussed informally. Our requirements weren't strict, even when fully formulated.

- Expressive. The language should not limit the range of applications we wanted to make. It should also be easy to write them, without writing a lot of code.
- Functional. The major reason for for this was Wirzenius's desire to play with functional programming. However, we also thought it might reduce some classes of bugs related to updates to data structures.
- Garbage collected. Manual memory management is tedious and risky, and memory leaks are especially traumatic in embedded applications.
- Simple to implement. We were going to implement it ourselves, and did not want to spend a lot of effort on it.
- Already familiar or at least easy to learn to all of us. Otherwise we wouldn't even use it ourselves.

Table XXX lists some languages and how they filled our fuzzy requirements.

Table: Language possibilities for Hedgehog.

Common Lisp	A fairly large, but powerful language. ANSI standard exists.
Scheme	A smallish, but powerful language. Still, larger than what we needed.
FORTH	Typically very small and simple. Unfamiliar to all but one at Oliotalo.
BASIC	Can be small, but usually not very expressive. Has a stigma of being regarded as a toy language.
C, interpreted	Somewhat small, somewhat simple. Quite tedious. No garbage collection ineherent in the language. This is what we wanted to get out of.
C++, interpreted	Very large and quite complicated. Tedious. No garbage collection inherent in the language.
Java	A fairly large language with complicated semantics. Variants exist for embedded devices, but higher-end than ours.

In the end, we chose Lisp as the model for our language. Lisp was at least somewhat familiar to most of Oliotalo's staff. We wanted to make our own implementation. We need to fully understand the implementation ourselves, to be able to deal with problem situations, and to be sure we can adapt the implementation to any embedded system we may have to support it on, and the best way to do this is to write it ourselves.

We also decided to create our own dialect of Lisp. We assumed, possibly wrongly, that a full implementation of, say, Common Lisp or Scheme, would be too big for our target platforms, and would also contain large parts we would not be needing. If we didn't implement those languages fully, we would have to document carefully what was missing, and be constantly aware of this when writing code. Worse, we might want to have to invent extensions to deal with our special need. Admittedly, we also simply wanted to play language designers.

4 Hedgehog: the original system

This section describes the first attempt at implementing the Hedgehog Lisp system.

4.1 Initial prototyping in Python (1p)

Having chosen Lisp as the model for our programming language, we then wanted to experiment what it would feel like to write programs in it. We wanted a prototype implementation to be able to play with programs, and Wirzenius quickly wrote one in Python. The first version was very small, only about 200 lines of code.

The first prototype implemented a very simple language, which nonetheless had most essential features. For example, it used had garbage collection, by expressing Lisp objects as Python objects, and relying on Python's garbage collection. Python proved to be a good language for rapid prototyping. The Lisp community has a tradition of using Lisp itself, and especially its macro system, for prototyping with new language features, but Wirzenius, who was doing the work, did not know Lisp at the time.

Given the ease of prototyping, we went through several quick iterations for various language features. Most of this work was done by Wirzenius, by writing toy programs to experiment various language features, and changing the prototype implementation as needs arose. Even a very small language proved to be quite expressive.

An early decision was to make the language functional, or at least to disallow changing existing data elements, what language theory calls mutation. Common Lisp and Scheme, and most other Lisp dialects, are not strict about this. A strict ban on mutation makes some things harder to efficiently, but seems to also prevent many difficult bugs.

We did some minor experimentation with adding object oriented features. However, no working, nice, simple model for this was found by ourselves, and there was no time to

research the literature. Object orientation did not seem to be all that important, either, so the idea was quickly dropped.

4.2 The first C versions (1p)

After some prototyping, the language definition seemed to stabilize and it was time to start making an implementation in C so that we could test the language on an embedded device. We needed to verify that the speed and memory consumption of the language would be reasonable before we invested too much in the project.

The C implementation was done in two stages. First, the Python implementation was rewritten in Python, but using constructs that would easily map into C. For example, simple garbage collection was implemented, and data structures were simple arrays, rather than Python's lists or dictionaries. Using Python for the first stage of the rewrite made it easy to experiment with various approaches and to find abstractions that fit C and worked well. The rewritten Python code was then translated by hand to C.

The C code was written in three stages. Memory management was implemented first, so that all the rest of the implementation would be able to use that. Implementation of parsing and finally of execution quickly followed. The C code was quite easy to write after the prototyping in Python. The goal at this stage was correctness, not speed.

The first target for the C code was, of course, Linux, not the embedded device. It is much easier to debug things on a workstation. As soon as the Linux version worked, it was ported to the Aplicom device. The port proved to be easy, with the exception that we did not have an easy way to load Lisp code into the device. Initially, we compiled the Lisp code into the interpreter, which works, but is quite tedious. Ultimately, we changed the interpreter to load the Lisp code from a serial port. We used no transfer protocol, just receiving raw ASCII text. Luckily, we had few transmission errors.

The Lisp interpreter proved to be fast enough and small enough to run well enough in our test on the embedded device. Our tests weren't very thorough, since we did not have a real application yet to write for or port to the Lisp system, so it was hard to specify speed requirements. What we did test, however, showed that our chosen approach was feasible.

This was the situation in late 2002.

4.3 Real use: the garbage compressor (1p)

Eventually, the first real project appeared for which we could use our nascent Lisp implementation. The project was to measure how full a garbage compressor was and report this to the server. The system allows the company that empties the garbage compressor and moves the waste to a suitable place to optimize its logistics better, and be notified quickly in case of malfunctions.

A Lisp application was written to measure the hydraulic pressure of the garbage compressor during the compression. The fill ratio of the garbage compressor can be computed from the hydraulic pressure, though not trivially for all models. There was a need to support many models, with different compression cycles and correspondingly different functions for computing the fill ratio.

The application was also required to be loaded over the air, not a serial port. We used no persistence in the hardware: when the embedded device lost power, the Lisp application and all its data vanished from memory. We wrote a small bootstrap to load the Lisp application code over GPRS. The bootstrap was also written in Lisp, and compiled into the interpreter.

Unfortunately, Hedgehog proved to be rather too slow. The measurement of hydraulic pressure was just fast enough, but computation of the fill ratio was much too slow. A significant problem was that the garbage collection of the Lisp interpreter was pretty slow. We worked around this by simplifying the problem: we only measured on the embedded

device, and sent all data to the server to be analyzed and computed.

Other problems also occurred. For example, since the Aplicom device has no display, people monitoring the device could not know whether it was just slow, or whether it was actually doing something. The device does have three three-colored LEDs, which we began to use to report device status.

The biggest problem was the need to debug remotely. Someone, usually the customer, was standing near the box, explaining over the phone to the programmer sitting at the office, who was looking at server logs. This was very hard, and extremely irritating. At the time, we did not have a laptop to bring on-site and see logs from both the embedded device and the server simultaneously. This situation would be repeated in several later projects, and be arguably our biggest obstacle in the development on the embedded side.

4.4 Garbage collection improvements (1p)

The biggest acute problem with garbage collection was that during the garbage compression phase, the measurement could stop for several seconds. Sometimes even longer, and in fact so long that the hardware watch dog in the Aplicom device would reboot the device before the garbage collection would be finished. This was, of course, not acceptable.

The Jones and Lins book *Garbage collection: algorithms for automatic dynamic memory management* was useful for suggesting improved methods for garbage collection. The old method was a very simple stop-and-copy. A simple generational approach was selected for the new method, and some additional care was taken in its implementation. Where the old method's implementation had stressed correctness, not speed, the new one's code was also designed to be fast.

The new garbage collector proved to be fast enough for the garbage compressor application. The longest garbage collection pauses were a fraction of a second, and it was thus possible to

collect garbage several times a second without it being readily noticeable.

4.5 Real use: the liquid container watcher (0.5p)

After Hedgehog had been more or less proven in a real application, it could be used for new ones. The garbage compression application had been written by Wirzenius together with a co-worker, the next application was to be written by the company CEO, Kaius HÃ¤ggblom. The application would be monitoring how full a container for liquid (typically oil or gasoline) is, by measuring the output of a pressure sensor at the bottom of the container.

HÃ¤ggblom managed to write the Lisp application quite quickly, with only minimal instruction and hand-holding. This verified that our language was simple enough to be learned quickly, which had been one of the goals.

At this stage, it proved lucky that we had decided to generate reference documentation automatically. As part of the compiler build process, documentation for built-in functions is extracted and inserted into an HTML template. HÃ¤ggblom could look things up himself. This opens up the possiblity that people outside Oliotalo might be able to use Hedgehog, which had not been a goal, but would certainly be pleasant.

4.6 Simulating the box on a workstation (0.5p)

Writing embedded applications in Hedgehog Lisp was immediately quicker than writing them in C. The multi-step iteration loop (edit, compile, upload, test) become shorter (edit, upload, test). It was still slow, however, since the data transmission over a serial port was done at the slow speed 9600 bit/s to avoid transmission errors and the need for hardware handshaking. To work around this, we implemented some features to the Linux version of the Hedgehog interpreter to simulate the Aplicom device. This allowed the bulk of the application development to be implemented on the workstation, and markedly sped up development.

Of necessity, the level of simulation was very low. A full simulation or emulation would have been much too much work, so compromises were made. The simulation was just good enough, however, that it helped enough, and allowed better simulation to be implemented in the future.

4.7 Mature, but with persistent problems (1p)

Towards the spring of 2003, the first generation of Hedgehog Lisp had shown that the decision to implement a language interpreter was the right one. It was good enough for our projects, and stable enough and with few enough bugs that they did not disturb us.

Hedgehog was not, however, without problems. The biggest problem was execution speed. For some things we wanted to do, a drastically faster implementation was required.

Another problem was the size of the Lisp code that had to be transferred. Since the devices would be downloading it over GPRS, the smaller the code is the faster it happens and the less it costs. We had started a library, to be included into every application, and had to manually remove such parts of it that weren't used by a particular application. This was error prone and time-consuming.

It was clearly time to re-think the whole implementation. A byte code approach was natural, being the de facto standard way to implement interpreted languages. Hedgehog was Wirzenius's first real language implementation. It was time to hire an expert.

This was the situation in April, 2003.

5 The next generation: the need for speed

This chapter explains how the second generation of Hedgehog came to be.

5.1 A compiler and byte code interpreter (1p)

In April, 2003, Oliotalo contracted Kenneth Oksanen to re-implement Hedgehog. During the specification phase, it was decided that more emphasis would be put on getting the interpreter mature than the compiler. The compiler, after all, would only be at Oliotalo, and would be easy to fix later. The interpreter would be installed in many devices, and would be expensive to change. Also, it was deemed that a simple compiler would suffice, whereas a limited interpreter would hurt badly, as it would prevent some applications from working at all. The compiler would not need to have much optimization.

Apart from the compiler and the byte code interpreter, the third major component of Hedgehog would be the standard library. The library would also reside outside the interpreter, and would be included by the compiler with the application code. This would make byte code files somewhat larger, but we decided this was less important that the flexibility we gained from being able to change and fix the library over the air. Once the library would stabilize, we could embed it later, if it were to be deemed beneficial.

Oksanen quickly came up with a design and the first versions of the compiler and byte code interpreter. In the first benchmarks, they were 100 to 600 times faster, which was nice. Oksanen wanted to make some small changes to the language. They were, however, very small, and improved the language, and thus were acceptable.

The responsibilities in Hedgehog development were divided early on between Oksanen and Wirzenius. Oksanen took care of the compiler and interpreter core. Wirzenius took care of language design, library development, and adaption of the interpreter to the embedded environments.

5.2 Achieving speed and small size (0.5p)

In addition to a compiler with only the most basic of optimizations, we wished for other tools to help us write good code quickly. Ultimately, the speed and size of a program depend on the programmer, not the language implementation. This makes a profiler important.

A few optimizations were, however, deemed vital. The most important one was that the compiler should not include in the byte code file such functions that were never called. This would make it possible to significantly enlarge the library, without it causing byte code files become huge. Even the most simplistic implementation of this optimization was enough: if the name of the function, in any meaning and scope, was used anywhere, it was included in the byte code file. Function names are chosen naturally so that they are rarely used as names for local variables.

Another important optimization was support for being able to include or not include unit tests for library functions in the compiled code. Our library is in a single file (it is large only in an embedded context), and it seems most natural to keep unit test functions in the same file. This was the original reason to implement macros in Hedgehog.

Once Hedgehog matures and there is less need to work on the interpreter, any number of optimizations can be added to the compiler.

- Automatic inlining, where the compiler replaces calls to short functions with the body of the called function. This can improve performance, and may open the door for further optimizations, such as for constant folding function arguments are constant.

- Automatic un-inlining. Sometimes code size can be made smaller by recognizing that the same code segment is produced at several locations in the source program. The compiler can then generate a new function and replace the code segments with calls to that function.
- Common subexpression elimination, where the compiler recognizes that the same thing is computed several times, and re-uses the result. This optimization tends to be easiest to implement in functional languages, since the analysis simpler than in languages where data can be mutated. Potential benefits from it are therefore bigger than in, say, C or Java.
- Constant folding, where the compiler computes the results of functions at compile time. When the function is pure (i.e., has no side effects, such as I/O), and its arguments are constant, its value can be computed at compile time. We are unsure how often this actually happens, since most values in the program should be results from I/O operations.

Perhaps most interestingly, since the Hedgehog compiler sees the whole program, including all the libraries, at once, it could do global optimization of the whole program. Traditionally, compilers have had to support separate compilation, but this is not an issue for Hedgehog. The Hedgehog compiler sees not only the whole application program, but also the whole library at once.

5.3 Language improvements (1p)

Once the basic compiler and interpreter were working, we could start adding new features to the language: macros, conditional compilation, tuples, and AVL trees.

The macro system is based on simple pattern matching. There can be several macro definitions, which differ in number of arguments, or which arguments are literals. Variable argument lists for macros are supported. We use macros mostly to avoid function call overhead, for example with the `(nil? p)` function, which translates to `(eq? p nil)`. Another use is to define new control structures without having to change the compiler. Two that have proven useful are `cond` and `let`.

```
(def-syntax (cond ?c ?t ... ?r)
  (if ?c ?t (cond ... ?r)))

(def-syntax (cond ?c ?t)
  (if ?c ?t))

(def-syntax (cond ?e)
  ?e)
```

Example: Macros to implement `cond` on top of `if`. The `cond` syntax is easier than a nested sequence of `if`s.

Conditional compilation is currently implemented using a syntax borrowed from the C preprocessor, see example XXX. The syntax is considered ugly and may change to something more Lispish later. At the moment, conditional compilation is only used to enable or disable unit tests from the compiler command line. Further use does not seem very likely at the moment.

```
#ifdef DEBUG

(def-syntax (fail-unless ?expr)
    (if (not ?expr)
        (panic "\nERROR: Condition "
               (quote ?expr)
               " failed.\n")))

#else

(def-syntax (fail-unless ?expr) nil)

#endif
```

Example: Conditional compilation is used to determine whether unit tests should be included or not.

Tuples and AVL trees are attempts at making more efficient data structures than traditional Lisp cons lists. A tuple is essentially a cond cell, but with more slots, indexed with integers. Because the whole tuple cell has to be re-created whenever an element is changed, they're mostly useful when their contents change rarely. Otherwise they will generate much garbage in the heap. In practice, tuples are used mostly for returning more than two values from a function.

AVL trees are included in the language to provide for quick lookups. We considered using tries, but AVL trees were thought to be more general. Our implementation allows any type of key, via a user defined key comparision function. We have an optimized version for the typical key types (symbol, integer, string).

Hedgehog Lisp continues to be a very small language, however, and this is good. The embedded devices we use are small, and a big language would inevitably inflate the interpreter size. A small language is also easy to learn, and has fewer pitfalls to the application programmer. More importantly, slow change is important to avoid conflicts within Oliotalo: we need to concentrate on making applications, and this is not helped if the language changes all the time.

In the future, it may be necessary to add a module system to Hedgehog. Although the language proper is now fairly stable, the library continues to grow, and eventually it may become large enough to warrant a module system.

5.4 Profiling support (1p)

At the moment, profiling support is still very simple. The byte code interpreter counts the number of times each position in the byte code file is executed and dumps this information when it exits. Tool translates to source code lines. No call traces, so no gprof.

```
outputs from profiling tools here
outputs from profiling tools here
outputs from profiling tools here
outputs from profiling tools here
outputs from profiling tools here
outputs from profiling tools here
outputs from profiling tools here
outputs from profiling tools here
outputs from profiling tools here
outputs from profiling tools here
outputs from profiling tools here
outputs from profiling tools here
outputs from profiling tools here
outputs from profiling tools here
outputs from profiling tools here
```

Figure: Profiler output.

The simplicity of the profiling feature, combined with some difficulty in using it, has resulted in it being used only little. Hence, performance problems. However, --speed.

Eventually, the profiling support will hopefully be improved to support coverage analysis as well. Good for testing. Testing important, since debugging embedded apps in the field is hard.

5.5 Library developments (2p)

The current Hedgehog compiler optimizes away functions that are not called. We have used this to grow the Hedgehog standard library significantly from what it was during the first C implementation. Also, the much higher execution speed allows for more abstractions in the library, when function call is no longer usually significant.

- Finite state machine framework to model threads.
- Pre-made state machines for various communication tasks, such as bringing up GPRS connections or exchanging Sparrow messages with Beaver, with automatic

acknowledgement handling.

- Basic list management, such as `map` and `filter`.
- Simple unit testing.
- Data structures, such as dictionaries ("hash maps") and functional queues.

The library uses macros to speed up some common operations, by inlining some functions. Automatic compiler optimizations would be nicer, but they are too much work to implement, for now.

We intend to continue to grow the library as we recognize similar needs in several of our applications. We prefer an approach where we first recognize a need and respond by creating a library function, rather than one where we put random functions into the library and hope they'll eventually be useful.

5.6 Application structure: state machines, not threads (2p)

All of our applications are written as a set of parallel finite state machines. The state machine is an abstraction of a thread. We have decided not to add real threads to Hedgehog, since they are a common source of hard bugs. Thread programming requires careful attention to locking and we prefer to keep things simple, especially in embedded devices, where debugging is much harder than in a server. The state machines are executed one by one, in a round-robin fashion, which gives the benefits of parallel execution and abstraction which threads would give, but requires no locking.

The functional nature of Hedgehog Lisp creates restrictions on how the different state machines can communicate. They can't, for example, change global variables. We have solved the problem via a shared data structure. Each state machine function is given the shared data structure as an argument, and returns it with any modifications it wants to make.

The new value is then given to the next state machine function, and so on. Garbage collection handily takes care of freeing the old values.

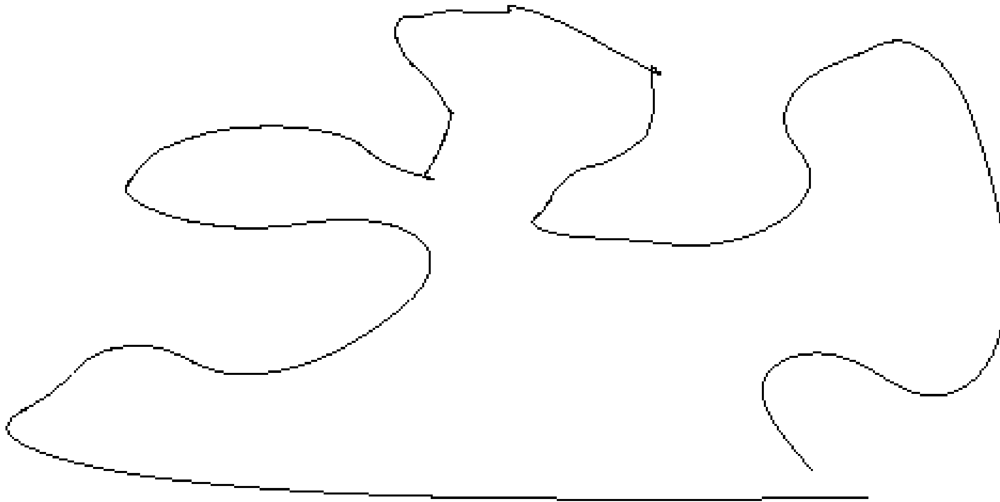

Figure: State machines in the garbage compressor application.

The shared data structure is a dictionary (a map from symbols to arbitrary values). Each state machine uses some unique keys to index the dictionary and to store its private values in it. Shared keys are used to communicate between state machines. Uniqueness of keys is established via name prefixes or by requiring that the application programmer provide the keys.

This approach proven to work quite well, even if it is somewhat repetitive from one application to another.

The data structure by the state machines is used all the time, so its speed is crucial. Originally, the dictionary was implemented as a simple list of key-value pairs. This proved to be a bottleneck, which was fixed by introducing AVL trees. The dictionary interface towards the application programmer did not change, so there were no code other code changes needed. Abstraction paid of here.

6 Application distribution

This chapter explains how Oliotalo deals with the logistics issues of making sure every box has a good version of the interpreter, and gets the right version of the right application compiled with the right version and right configuration of the compiler.

6.1 The logistical problem (1p)

At this time, October 2003, there are a few tens of units with Hedgehog in production use. It is expected that the number of installed units will grow by at least hundreds, possibly thousands within a year. The expected life of a unit is at least several years, possibly tens of years. Oliotalo will have to support all of them, and to keep maintenance costs down, the Hedgehog in each unit may only be upgraded in an emergency, or when there is another reason for the embedded device to be switched to another one.

The large number of units, and their longevity, creates several problems. When a new version of an application is produces, it has to be tested against, and possibly ported to, every version in active use in the field. This requires bookkeeping to track the versions. There is a need to provide each device with a byte code file produced by the correct version of the compiler. In addition, not only the compiler version matters, in some cases the compiler can be configured in several incompatible ways for the same target device.

6.2 Version management (2p)

Central to managing the large number of expected versions is a conservative approach to making new versions of the interpreter. The compiler can change, as long as it remains compatible with the corresponding version of the interpreter. Likewise the library. It does not matter, for example, if a compiler gets a whole new optimization pass, or the library gets a large pile of new functions, as long as the byte code interface stays compatible.

Eventually it will become necessary to update the interpreter. To manage this change, we have devised a version number scheme for the byte code binary interface. It consists of a three level version number and a configuration checksum. The version number consists of a major number, a minor number, and a patch level. The major number gives the generation of the implementation: Wirzenius's implementation is version 0, Oksanen's is version 1. The minor number and patch level give the version of the byte code interface within a generation. Whenever a new byte code instruction is added, the patch level is incremented. Whenever a byte code instruction changes meaning in an incompatible way, or is removed completely, the minor number is incremented, and the patch level is reset to zero.

As an example, suppose we are at version 1.0.0. If we add a new byte code instruction, for example to add support for AVL trees, all the existing byte code instructions retain their meaning. We therefore increment the version number to 1.0.1.

Suppose, on the other hand, that we notice the byte code instruction corresponding to the built-in primitive Lisp function `log2` is rarely used. We can make the interpreter a bit small by removing it from the interpreter and implementing it in the library instead. We then increment the version number to 1.1.0.

The byte code file uses a checksum approach to ensure correctness during transfer and compatibility between compiler and interpreter. When the compiler produces a byte code file, it embeds a checksum of everything else in the file. When the interpreter receives the file, it checks the checksum and refuses to execute the file if it is corrupt. This verifies correctness.

The checksum algorithm uses a seed number that is computed from the byte code instruction names and numbers for the configuration in use. This way, even if the version numbers match, but the configurations are different, the checksums will be different. This might happen, for example, if the compiler is configured for a BlueGiga device, but the application is configured for an Aplicom device.

The three level version number is good for the interpreter, but the compiler and library need an additional level. If the byte code interface, and the interpreter in general, is OK, but there is a bug in the compiler or the library, it needs to be possible to fix the bug without incrementing the version number of the interpreter. This reduces the need to update the interpreters. We solve this by adding a letter to the three level version numbers: 1.0.1a, 1.0.1b, etc. The letter is incremented when only the compiler or library changes, but in a way that retains compatibility with the interpreter. In effect, the interpreter always has the same version number, without letters. If the interpreter is changed, at least one of the three numbers in the version numbers must change. A compiler is only compatible with an interpreter with the same version number, except for the letter.

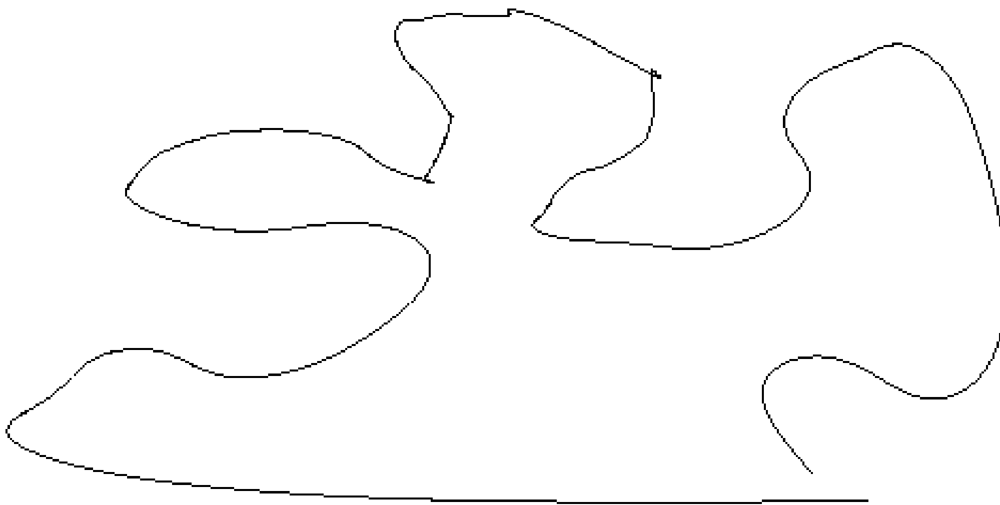

Figure: Version tree.

6.3 Appdist server: centralized application distribution (3p)

Hedgehog has a classic bootstrap problem: when it is installed in an embedded device, it does not contain any application code and needs to load it somehow. Actually, the interpreter executable can contain an byte code file. This should not, however, be the actual application, since that would require building an interpreter executable for each application, which makes

logistics harder. Instead, a special bootstrap application is included in the executable and has the responsibility for loading the application into the embedded device, if there is none already. The download can be over a serial port, GPRS, or Bluetooth, i.e., any communication method that the hardware supports.

The bootstrap code uses our Sparrow protocol to request the application code from a server. We have a dedicated server for distributing applications to the embedded devices, and the Internet location of these is hard coded into the bootstrap application. This allows the application servers to be moved more easily.

The byte code distribution server needs to know which version of which application to send, and also needs to know with which compiler the application should have been compiled. This information is kept in a database, which is accessed using as keys the serial number of the embedded device, and the version number and checksum seed of the interpreter in the device. This scheme allows the embedded devices to be generic, not application specific, and also provides a centralized place where to make a change when the application version to be run in a particular unit needs to be changed.

In most applications, the code has some parameters that need to be tweaked for each installation. For example, the type of a garbage compressor is needed so that the application can use the correct function for computing the fill ratio. All embedded units need to know at least their short identifier used instead of the serial number to conserve bandwidth. This information could be embedded in the byte code file sent to the unit by the byte code distribution server, but then there would be a separate byte code file for each unit, a massive number of files.

The solution is to have a two stage bootstrap. The bootstrap application fetches the byte code file for the application, and the application then fetches the unit specific configuration information from the application server. Only location of the application server needs to be configured into the byte code file, and is shared for all units for a given application.

There still needs to be a different byte code file of the same application for each interpreter version. This cannot be avoided. Building the application for each version can, however, be automated, All the versions should, ideally, be tested against all the versions, but this may prove to be impractical.

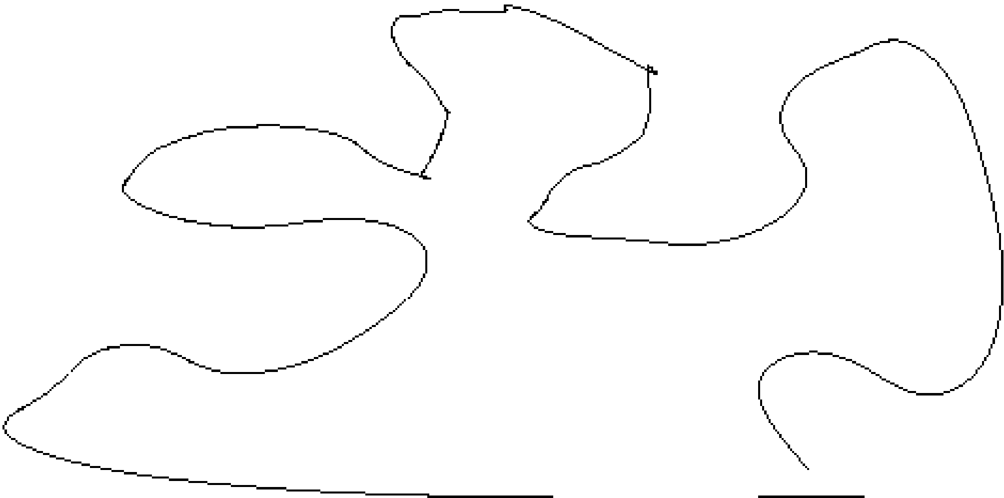

Figure: Application distribution.

7 Language design philosophy

This chapter explains the philosophy in the design of the Hedgehog Lisp dialect.

- The influence of Paul Graham and Arc (0.5p) This section explains how liw did not have any experience in Lisp or language design, but how he was inspired by Paul Graham's writings on the subject.
- Fewer parentheses (0.25p) This section explains how traditional Lisp dialects have had many parentheses that liw though unnecessary. cond and let are discussed in particular.
- Common operations should have short names (0.25p) This section explains how liw agrees with Graham that pr and fn are better names than princ and lambda.
- Simple, fast, easy to implement
- Common things easy and short, complicated stuff not too tedious either

8 Lessons learned (3p)

This chapter discusses the lessons learned from implementing and using Hedgehog.

- Wrap operating system facilities thinly (1p)
- Get the byte code interpreter right early: customers get nervous when it changes (0.5p)
- Static typing would be helpful (0.5p)
- Performance is always a problem: if nothing else, the customer wants more features (0.5p)
- The customer does not know what he wants, so be flexible (0.5p)
- Communications costs always surprise (0.5p)
- Functional programming: is it worth it? (1p)
- Syntactic problems with Lisp (1p): returning multiple values

9 References (1p)

This chapter contains the list of sources referred to.

McCarthy also dislikes cond syntax: "The unexpected appearance of an interpreter tended to freeze the form of the language, and some of the decisions made rather lightheartedly for the ‘‘Recursive functions ...’’ paper later proved unfortunate. These included the COND notation for conditional expressions which leads to an unnecessary depth of parentheses, and the use of the number zero to denote the empty list NIL and the truth value false. Besides encouraging pornographic programming, giving a special interpretation to the address 0 has caused difficulties in all subsequent implementations."

Richard Jones, Rafael Lins, *Garbage collection: algorithms for automatic dynamic memory management*, John Wiley and Sons Ltd, 1996, ISBN: 0471941484.