# Kannel Architecture and Design

## $Revision: 1.20 $

**Lars Wirzenius**
**Gateway architect**
**Wapit Ltd**

**liw@wapit.com**

**Kannel Architecture and Design: $Revision: 1.20 $**

by Lars Wirzenius

This document examines the purpose of a gateway in the Wireless Application Protocol (WAP) architecture. WAP is a technology for implementing services on moible phones using hypertext similar to the World WideWeb (WWW). We examine a particular WAP gateway, called Kannel and will discuss its design and look how well it works.

# Table of Contents

# List of Figures

# Chapter 1. Introduction

It has been a long-time dream of science fiction authors and fans to have a computation device with you at all times. This dream device would allow you to do many of the same things a full-size device would, whenever and whereever you would need it. In addition, just by always being with you, it allows you to do things a static device won't. For example, in Arthur C. Clarke's novel *Imperial Earth* everyone has a small device that works as a notebook, dictionary, encyclopedia, and recording device. This itself is nothing special: real-life computers have been doing such things for decades by now. However, by virtue of being small enough for its user to always carry it, it suddenly becomes much more powerful. You have instant access to all the information you need, when you need it. You always have all your notes with you, allowing you to get things done wherever you are, and not just in the office.

A lone computer is nice, but quite boring compared to a networked one. Instant communication with anyone, anywhere, at anytime is another science fiction dream. The communication device worn by characters in the *Star Trek* television series is an example: just by tapping the device once, you can talk to anyone.

The real world has been catching up with science fiction. Mobile phones have allowed near-instant communication with anyone else for several years. They aren't as small as the Star Trek devices, nor as fast, but they are small enough to be carried at all times, and they're extremely popular.

Mobile computers are also reality. The first portable computer, the Osborne-1, was produced in 1981. It was the size of a suitcase and weighed more than ten kilograms, so it wasn't particularly easy to carry, but its descendants have evolved into smaller and smaller versions each year. Current technology allows laptops that are small enough to be carried in a bag to most places. Even smaller devices, palmtops, really are small enough to be taken everywhere, in a pocket, just like modern mobile phones are. Palmtops aren't as powerful as laptops, or desk computers, but powerful enough to do many useful things.

We now have mobile phones, providing connectivity, and laptops and palmtops, providing processing power. Combining them is a natural next step, and actually one that has, to some extent, already been taken. Mobile phones can function as wireless modems to mobile computers, providing network access anywhere. This has some technical problems, however, due to the limited bandwidth and high error rates in mobile networks. It also requires carrying two devices, and connecting them in some way. Having a single device capable of both communication and data processing is likely irresistible to the mass market. Some such devices already exist, such as the Nokia Communicator and the Ericsson R380, but they have so far been priced out of reach for most consumers.

With hundreds of millions of mobile phones in use all over the world, the market for services targeted at mobile users is already immense. Even simple services find plenty of users, as long as they're useful or fun. Being able to get news, send e-mail or just be entertained wherever you are is extremely attractive to a large number of people. More sophisticated services make things even more attractive to even more people.

One technology for implementing mobile services is WAP, short for Wireless Application Protocol. It lets the phone act as a simple hypertext browser, but optimizes the markup language, scripting language, and the transmission protocols for wireless use. The optimized protocols are translated to normal Internet protocols by a *WAP gateway*.

Designing and implementing a WAP gateway is a straightforward excercise in well-established software engineering practices. However, when the system has to work efficiently and reliably for a huge number of concurrent users, some special problems arise.

This Master's thesis discusses the problems and the design decisions of a particular WAP gateway, called Kannel. We also provide benchmark results to justify the design. We do not only discuss technical issues, but touch some project management issues as well, since the Open Source nature of the gateway affects the way development has been and is being done.

Kannel is an Open Source project by the Wapit Ltd company launched in June, 1999. I was employed by Wapit to head the project. It is not the first Open Source project with which I am involved, but it is the first one where I am getting paid for the work. Kannel is an important and necessary part of Wapit's approach to implementing mobile services, but not one from which the company expects to make money directly. The reasons for making Kannel Open Source are discussed.

This document was originally written to be a Master's thesis for Lars Wirzenius. The thesis has not (as of this writing) been finished, but a work-in-progress version has been converted to be the Kannel architecture document, since the old architecture document had not been updated for a year. The thesis version and the architecture document version of the text will evolve in parallel.

# Chapter 2. Problems in implementing services for mobile phones

This chapter explains the problems in designing and implementing services for mobile phones. There are both technical problems (how to do it at all) and business problems (how to make money doing it, in the short and the long term). There are various approaches to solving these problems; we will briefly summarize the important ones.

## Technical problems

In order to be usable to their users, mobile phones have to be small in size and light in weight. This puts rather severe limits on their design, which results on several problems:

- The battery has a fairly low capacity, resulting in more limitations due to having to keep power consumption down for every part.
- Small screen and keyboard, resulting in very limited input and output possibilities and making user interfaces awkward.
- Slow processor and little memory, resulting in little computation being possible on the phone itself.

Some of these limitations apply only to phones and other mobile devices do better. For example, the screen size of a *Palm* or *Psion* device is large enough that simple text processing is doable. For every device meant to be mobile, however, the limitations will apply to some extent. It is not really possible to comfortably carry around a full size keyboard, mouse, and screen.

The wireless mobile network also has severe limitations, compared to a wired local area network. The total amount of bandwidth that all mobile users in a geographical area can share is limited. With cables it is always possible to expand the bandhwidth by installing more cables, but the total spectrum of radio waves available for mobile networking is limited, both by physics and by the way it has been allocated to various purposes by governments.

Radio waves are also inherently error prone, since they are affected by many sources of disturbances: other devices and the Sun cause intereferences by sending their own signals, and buildings, mountains and other parts of the landscape distort and in some cases prevent the radio signals from reaching their destination. Even if nothing else is a problem, the distance to the nearest base station for the mobile network may be too large.

This results in a network with limited bandwidth and a high error rate. Normal networking protocols, such as TCP/IP, have been designed for an environment with low error rates, which makes them partly unsuitable for a mobile network. Additionally, the various protocols used in the Internet (and that's about the only interesting global network for mobile users as well) on top of TCP/IP are textual, meaning that the messages they send are plain text. This makes them easier to specify and understand, and much easier to implement them and debug the implementations, but when bandwidth is very limited, they do waste it.

# Business problems

The technical problems outlined in the previous section are fairly straightforward to solve in isolation. However, in order to be viable from a business point of view, the choice of technical solutions needs to be guided by business needs. The basic business requirement for a mobile network operator is that building a system for mobile services needs to be profitable: the cost of building the system must be less, in the long run, than the income generated from its users.

Mobile phone networks already support data connections, so the basic problem of getting data from and to the mobile devices is already solved. The current solutions are not optimal, but they are good enough for now. The shortcomings of data connections in current networks will be solved by next generation mobile network technology, such as GPRS and UMTS. This, however, results in a new problem: the work done for implementing services in today's networks must not be wasted when newer networks are employed. Thus, today's solutions must be designed so that they will work tomorrow as well.

Telecommunications equipment needs to be interoperable: devices from different manufacturers must be able to talk to each other without problems. The interoperability requirement results in the need for a standard for the way mobile services are implemented. This standard may be a formal international standard, or an industry de facto standard, as long as it is open and adhered.

The standard should also be as broadly applicable worldwide as possible. This makes it easier to reach a critical mass of users: when there are enough users, it becomes easier to mass produce devices and that keeps the cost of the devices low, thus allowing even more users to buy one. Also, a mobile device is more attractive, if it allows interaction with many other users.

Reaching critical mass is impossible, if the devices or services are hard to use. Thus, whatever solution is chosen, it must allow user interfaces that are easy to learn and use.

Business requires that service users can be billed. For real mass market use, billing needs to be efficient and simple, both for the service provider and the user. Billing can be arranged in various ways: pre-paid, paid with phone bill, or via credit card, for example.

In summary, the way mobile services are implemented must fill the following requirements:

- Leverages on existing mobile phone networks.
- Must be adaptable to future mobile networks.
- Standardized, for interoperability and mass market.
- Allows easy user interfaces.
- Allows billing.

We will next look at how current solutions work.

# Pre-WAP solutions

Current (GSM) networks have two ways of implementing services for mobile phones: normal voice calls and short textual messages (SMS messages).

A mobile phone can be used to make a normal voice call to a service number, which is typically answered by a computer. The computer plays recorded voice messages, and the user can interact by pressing the number buttons on his phone. This is awkward and slow, and if there is any information that

needs to be saved, the user needs to make notes with a pen and paper. This is uncomfortable enough that it is not a viable solution except in very rare cases.

SMS messages are short textual messages (up to 160 characters) that are sent from or to GSM phones. Similar functionality exists in alphanumeric pagers in the US, and elsewhere. The messages are sent to a particular phone number, and can be processed by a computer. The computer can then send a reply message. This allows many simple services. For example, one could implement a service where by sending the words `WEATHER HELSINKI` to a given number, one could get the current weather forecast for Helsinki in reply.

SMS based services fill most business requirements given in the previous section. They work in current networks, and a billing infrastructure already exists. They are not, however, very user friendly, since the messages are short, and it is difficult to memorize long lists of service keywords and arguments.

An additional current option is to run a dial-up Internet connection over a GSM data call. The phone would then use normal HTTP to fetch normal HTML pages. This is doable immediately, and requires modification to the phone only. The mobile network and the services can exist as they already do. The main problem is that existing HTML pages are written in such a way as to require fast connections, fast processors, large memories, big screens, audio output, and may require fairly efficient input mechanisms. That means they can be made attractive for users of traditional computers and networks. However, portable phones have very slow processors, very little memory, abysmal and intermittent bandwidth, and extremely awkward input mechanisms. Most existing HTML pages simply will not work on them, nor will they ever do that. Even simple HTML pages with minimal markup will be hard to use on a display with only a few lines of text with a few words each. Content needs to be specially tailored for such small screens.

Even if content were tailored, normal HTTP is fairly verbose for use in a mobile network with slow bandwidth. An HTTP request constists of up to dozens of lines of text, or up to a couple of hundred of bytes. This can be made much more efficient by using a special protocol.

# Chapter 3. The Wireless Application Protocol

This chapter explains the goals of WAP, and summarizes its history so far. It introduces WML and WMLScript, and explains why they are used instead of the normal Internet languages (HTML, Java, JavaScript, and other formats). It explains shortly the on-the-air protocols of WAP, and why the normal TCP/IP or other existing protocols weren't chosen, and the role of the gateway in translating between air and Internet protocols. It covers features such as sessions.

This section also covers the current status of WAP (what is implemented and in use, and market penetration), and the expected future, with GPRS (and wouldn't it be sensible to use HTTP with GPRS?), UMTS, and 3G coming (and I need to look up what those acronyms mean). XXX does it?

## Goals and history of WAP and the WAP Forum

The WAP Forum, which creates and maintains the WAP specifications, was founded in June, 1997 by Ericsson, Nokia, Motorola, and Unwired Planet (later Phone.com, later Openwave Systems). The 1.0 specification was published in April, 1998, but wasn't implemented in phones. The 1.1 specification came out in July, 1999, and was the first one implemented in publically available phones. 1.2.1 came out in June 2000, and phones implementing it (and specially push) are now available, too. Work on WAP 2.0 is closing. It will converge WAP with Internet protocols and W3C markup languages.

The goal of the WAP Forum is to develop an open, freely licensed specification that is not tied to any network technology, nor to any specific device. They want to do this in a way that is as compatible as possible with existing Internet technologies, to allow existing content providers to use existing content when creating mobile services.

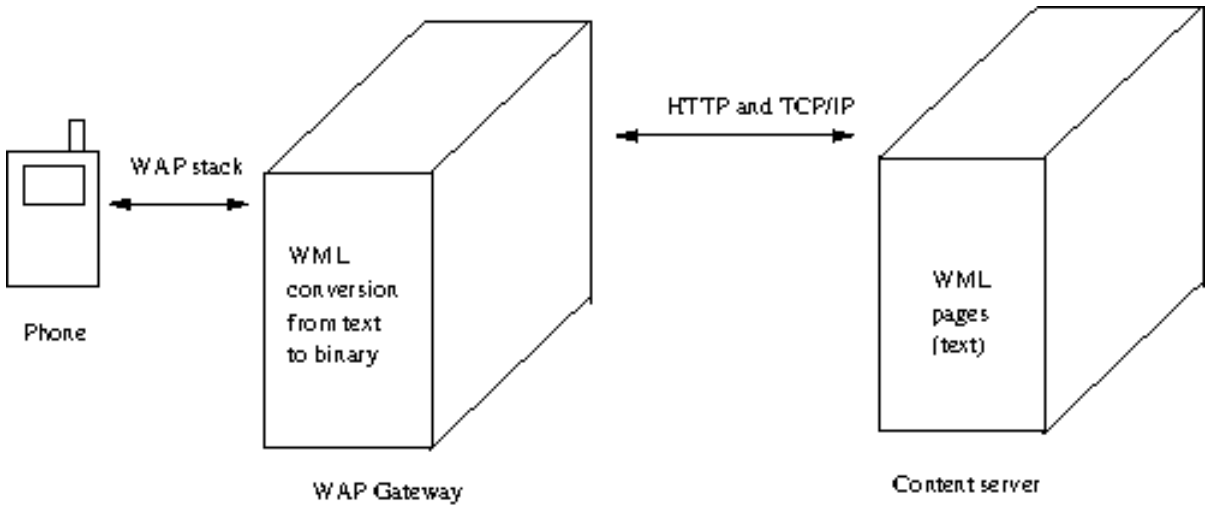
## The WAP architecture

WAP is a collection of languages and tools and an infrastructure for implementing services for mobile phones. WAP makes it possible to implement services using hyper-text, similar to the World Wide Web.

Here WAP Push and Pull are handled separately, because they are, indeed, very different services. Meaning of these terms will, hopefully, come clear later.

WAP pull does not bring the existing content of the Internet directly to the phone. As discussed in *Pre-WAP solutions*, existing content is unlikely to display properly on a phone anyway. Instead, WAP defines a completely new markup language, the Wireless Markup Language (WML), which is simpler and much more strictly defined than HTML, making it easier for the hypertext browser in the phone to interpret and display it. WAP also defines a scripting language, WMLScript, which all browsers are required to support. To make things even simpler for the phones, WAP even defines its own bitmap format (Wireless Bitmap, or WBMP).

**Figure 3-1. WAP architecture**




WAP defines a protocol semantically equivalent to HTTP, but being in a binary and compressed format it reduces the protocol overhead to a few bytes per request, instead of up to hundreds of bytes. However, to make things simpler also for the people actually implementing the services, WAP introduces a gateway between the phones and the servers providing content to the phones.

The WAP gateway talks to the phone using the WAP protocol stack, and translates the requests it receives to normal HTTP. Thus, the content providers can use any HTTP servers, and can utilize existing know-how about HTTP service implementation and administration.

In addition to protocol translations, the gateway also compresses the WML pages into a more compact form, to save bandwidth on the air and to further reduce the phone's processing requirements. It also compiles WMLScript programs into a bytecode format.

# WML and WMLScript

Content and services in WAP are presented to the phone using the Wireless Markup Language (WML) and the WMLScript programming language. WML is a simple markup language defined with XML and is used to mark the contents of the file as actual text, title, hyperlinks, etc. [1]

A WML page is a *deck* of *cards*. One card at a time is displayed by the phone. It is possible to switch between cards on the same deck quickly, since the whole deck is downloaded at once. A WAP application might fit onto one card, or be divided into several, depending on its size and how big decks the phone accepts.

WMLScript is a simple programming language based on ECMAScript and JavaScript, which are usually but not always implemented in WWW browsers. A WAP browser is required to implement WMLScript. WMLScript is used to make WAP pages more dynamic. It is not always enough to provide only a static page. The application might, for example, use WMLScript to let the user only fill in valid values into a

form. The validation can be done on the content server as well, but then it will need to be sent there and the result needs to be fetched back. This is both slow and potentially expensive, if a new connection to the gateway needs to be established.

WMLScript also defines a number of libraries for controlling phone functionality. This could, for example, be used to implement better phone book browsers than what is implemented in the phone itself.

A WML page is typically provided by the content server in its textual form, and the WMLScript code as source code. The gateway will then translate WML into a binary format, which is more compact, and WMLScript into bytecode, which is simple for the phone to execute and requires no CPU intensive parsing on the phone. The phones typically can't handle textual WML or WMLScript source code, but rely on the gateway to do the translations.

# The WAP protocol stack

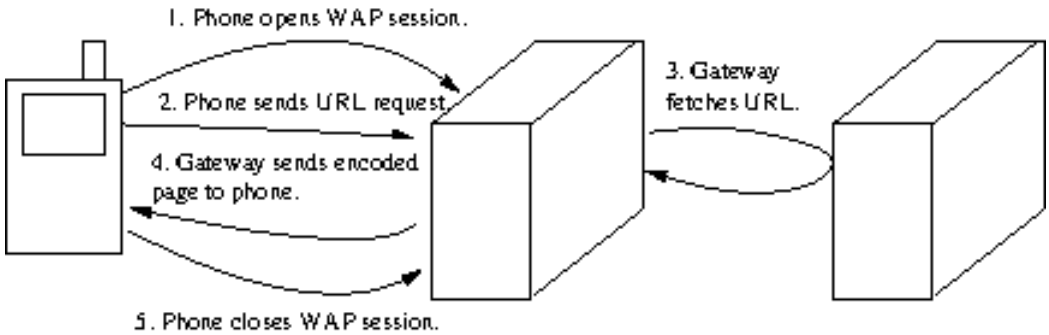
The WAP protocol stack takes care of transporting requests for pages from the phone to the gateway, and transporting the pages (possibly converted to a binary form) back to the phone. The protocol stack consists of three core layers:

- Wireless Datagram Protocol (WDP): Moves single packets to and from the phone. This is the lowest level layer defined by WAP. It is implemented on whatever suitable mechanism is available on the underlying network. For TCP/IP networks, it maps directly to UDP packets.
- Wireless Transaction Protocol (WTP): Implements a single request-response pair between phone and gateway. The request may be for a new page, or it may be something related to the higher level protocols.
- Wireless Session Protocool (WSP): Takes care of handling actual requests for pages. Sessions are used to optimize bandwidth usage.

A representative use case for the protocol is shown in Figure 3-2.

1. Phone opens session, using WSP. Phone and gateway negotiate protocol features and HTTP headers to be used in requests gateway makes on behalf of the phone.
2. Phone sends URL for the page the user has configured as his home page.
3. Gateway makes HTTP request, with negotiated headers.
4. Gateway encodes page in a binary form and sends it to the phone.
5. User shuts down the browser and the phone terminates the session.

**Figure 3-2. A representative WAP session**




The behavior of each end of the WAP connection is defined by a set of layer-specific state machines, which define the handling of timeouts and other errors. We shall not discuss these at any length, since the details are not relevant this document.

The WAP protocol stack contains several additional optional layers and optional features of the layers we have mentioned. These are not interesting for the discussion of the design of Kannel.

# The duties of a WAP gateway

This section summarizes the duties of the WAP gateway. The basic duty is to implement the WAP protocol stack, as outlined in the previous section, so that the user can actually use WAP services. Additional duties may include, depending on how the gateway is used, user authentication and billing.

The gateway can often identify the actual user. For example, if the bearer used is GSM SMS messages, the user's phone number is known to the gateway. The gateway can then pass on this information to the content services. This is useful when the service can use this information to provide personalized service: it might remember the user's preferred settings, for example, or let him access e-mail without filling in a separate login form.

An identified user can also be billed. WAP content can be priced in various ways, and if it is to be billed to the user, the user needs to be identified. In addition, something in the WAP system needs to keep track of what billable items the user actually uses, and the gateway is one good place to do this. The gateway doesn't actually include a billing system itself, but it provides user and usage data to the billing system of the operator.

From the user's point of view, the gateway is also responsible for optimizing WAP usage as far as possible: the gateway should keep the number of packets small, to keep costs down and make best use of available bandwidth.

# WAP Push Architecture

Previous chapters defined pull mode of operation: a phone starts everything and a content server acts

passively. It is, however, sometimes useful that the server can start a transaction. Email notifications, news services and stock quotes are some examples of this mode of service.

However, simple pushing a content to the phone causes problems. User experience is badly detoriated, if incoming push messages can interrupt ongoing tasks. So instead an actual content, one pushes Service Indication, which tells that a service has become available and contains an URL telling where the user can found it. Then the user can accept or reject the service. The phone may confirm that is has received the push content.

In addition of this basic service, *PAP document* defines way to control the push operation. A push initiator can select network and bearer used, set delivery time restrictions and define quality of service.

SI document contains, too, some additional attributes defining more complex services than one defined before. For instance, a push initiator can force a SI content to appear on the screen of a phone immediately the phone receives it, or instruct the phone to put it into box, so that the user can read it later. It is possible, too, to define a expiring time for a content.

WAP Push uses established document defining standards, like XML and MIME and its protocols operates over well estabilised ones like HTTP and WSP (well, you can argue with that). This shortens the development cycle and reduces errors.

WAP Push is an application layer protocol, and so in principle totally independent the transport layer used. So WAP 1 uses WSP/WTP and WAP 2 HTTP/TCP.

# WAP Push Protocols

WAP Push suite defines protocols for sending content from server to a push capable phone. Confirmed push includes sending a confirmation from the phone to the gateway, and, if this is asked for, from the gateway to the initiator. There are two protocols: Push Over the Air Protocol and Push Access Protocol.

- OTA (Over The Air) protocol. Lightweight protocol used for sending content from gateway to the phone. It maps quite directly to WSP, but it is possible to use it with other protocols.
- PAP (Push Access Protocol) used for communications between a push initiator and a gateway. Protocol headers and protocol data are packed into XML documents, and, if necessary, MIME multipart messages and these documents or messages are exchanged over HTTP.

Push can be unconfirmed or confirmed. Unconfirmed one goes following way (fetching the user may ask for is omitted):

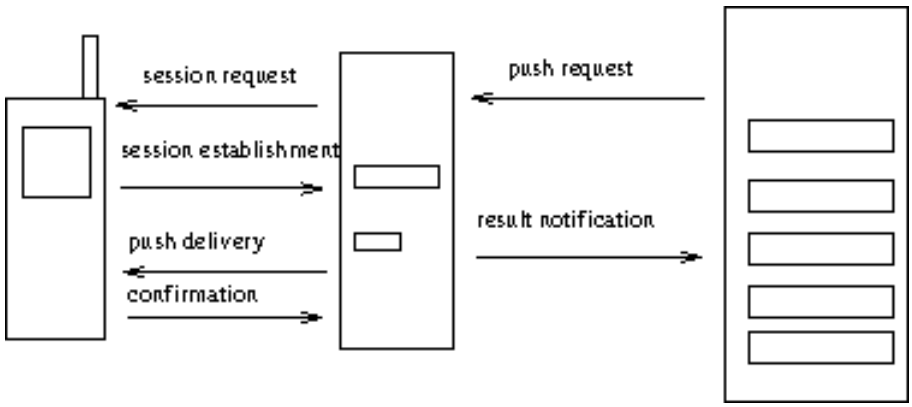
1. A push initiator send PAP message to a gateway, requesting a unconfirmed push of a content to the phone.
2. The gateway makes an OTA request for pushing the data to the phone. It is delivered using WSP sessionless services.

and confirmed (and session-oriented) one, see Figure 3-3:

1. A push initiator send PAP message to a gateway, requesting a confirmed push of a content to a phone.

2. The gateway sends an OTA request to the phone, asking it to establish a session with it (a gateway cannot do this by itself), if there is no session already open.
3. The phone establish a session using WAP protocol stack mechanisms.
4. The gateway sends the content to the phone.
5. The phone sends a confirmation to the gateway.
6. The gateway sends a result notification, meaning a confirmation to the push initiator.

**Figure 3-3. A confirmed WAP push**




# WAP Push XML languages

PAP protocol data is packed into a XML document. Language used for this is called PAP. User data can be of any MIME type, however, there are specific contents expressed with specific languages: Service Indication and Service Loading.

- Service Indication (SI). A gateway sends this type of a document to a phone for a push initiator. It tells to the user that a service has come available. He can reject, accept or postpone pulling of it. The indication may not interrupt the currently running task of the phone.
- Service Loading (SL). This is for a quite similar service, except now pulling in principle starts without an user intervention. However, the phone may implement a security mechanism suggested by SL specification and ask the user does he want to start pulling.

# Duties of Push Proxy Gateway

These are similar to WAP Gateway, because when pushing, the push application forms the external interface of a gateway. However, there is a possibility that PPG works very closely with PI, which does storing, billing and other similar services for it. This is a quite realistic alternative, because PI has the content and so it, not a "simple" gateway, provides visible services to the user.

Of course, PPG must spare bandwith, too. Even though SI and SL documents are small (url being their main content) they must be delivered to the phone overs SMS. Difference between one and two SMS messages can be truly great indeed.

# Notes

1. Using XML gives some syntactical benefits, such as fairly simple rules for parsing, which means that the parser in the WAP browser in the phone can be kept simple. It is also likely that the WAP gateways and the browsers in the phones will not tolerate syntactic and semantic errors in WML pages very well. Much of the complexity in WWW browsers is a result of having to cope with numerous versions of HTML, some of which conflict with each other, and accomodating for bad HTML in a heroic attempt to present the user with at least something interesting from the web page. Since WAP browsers are embedded into phones, and phones won’t be upgraded as easily or as often as WWW browsers, WML needs to evolve much more carefully than HTML has done and WML pages need to follow the specifications.

# Chapter 4. The Kannel Open Source WAP Gateway

This chapter is the most important part of the thesis. It covers the design and implementation of the Kannel Open Source WAP Gateway.

## Introduction to and status of the project (1 p)

This section describes the Kannel project at Wapit: why it was started, and when, and what its goals are. It gives general, very high level requirements for the gateway, from Wapit's point of view, and explains why the project is open source and not a proprietary thing. It gives the current status of the project and its product. It probably even covers why we use C and not Java.

Wapit Ltd was founded in the fall of 1998 to develop services for mobile phone users, originally based on SMS. During the spring of 1999, when the company started growing and became more amibitious, it decided to start developing services and authoring tools for the upcoming WAP platform as well. As part of its strategy, it decided that it made sense to develop its own WAP gateway, and to make it as open source. There were few existing commercial gateways on the market, and all of them were quite expensive. Since Wapit intended to deliver its service platform to many customers all around the world, it would have been quite expensive to buy a new gateway for each customer. So expensive, in fact, that it was cheaper to develop a new one. On the other hand, since Wapit had no interest in making money directly from the gateway, it made sense to create an open source project to develop the gateway. This way, it would be possible to get help from other companies, and to some extent individuals, in developing the gateway, and specially in testing the gateway in various environment.

The gateway project was launched in July, 1999, at the WAP Forum meeting in San Fransisco. The goal was to produce a gateway that was technically good enough at least for small operators and corporate level service providers. The author was hired at the end of June, 1999, to lead the project. By that time, there existed a very primitive proof of concept level prototype for an SMS gateway, which did not yet do anything for WAP. This was demonstrated in San Fransisco and it seemed that there was a huge interest in an open source WAP gateway.

Wapit decided that it made sense to make a gateway that was both a WAP gateway and an SMS gateway, because there was already a huge existing user base of SMS capable users, and few or no WAP users. Also, WAP itself can benefit from SMS, for example via over-the-air configuration messages (OTA) for phones, to make it easier to configure the WAP phone for a particular operator or service.

Initially, there was no formal requirements specification for the gateway - indeed, it did not even have a name. The gateway was just supposed to be 'fast enough', but a more strict formulation was not even possible, since it was unclear what kinds of usage levels the intended customers would have. During the fall of 1999, the following formulation emerged:

> The gateway needs to be able to serve thousands of concurrent users on reasonably priced hardware: less than ten PCs with high-end Intel CPUs, 128 MB of memory, fast network interfaces. It needs to be scalable to even higher levels of perfomance by adding more hardware (meaning that there can't be a single bottleneck that prevents more users from being served).
>
> —Kannel Architecture and Design document

For reliability, the requirement was that when the gateway was being run on several hosts (the architecture allowed this), crash or failure of one node should not affect the others.

The gateway was finally named Kannel in January, 2000. A kannel is a traditional Finnish musical instrument, but the name has no meaning for the gateway; it is just a nice name.

In the summer and fall of 2000, when this is being written, the gateway has been in light production use for several months, both as an SMS gateway and a WAP gateway.

# Requirements

This section lists the requirements of the gateway and its design.

The gateway must be able to share load between several hosts. This is also necessary for fault tolerance.

The gateway needs to be able to serve hundreds of concurrent users on a PC with a 400 MHz Pentium CPU, 128 MB of memory, 10 Mbit/s network interface.

The gateway needs to be able to serve thousands of concurrent users on reasonably priced hardware: less than ten PCs with high-end Intel CPUs, 128 MB of memory, fast network interfaces. It needs to be scalable to even higher levels of performance by adding more hardware (meaning that there can't be a single bottleneck that prevents more users from being served).

If one of the hosts running the gateway crashes (or the gateway component running on that host crashes), the rest of the gateway must continue to work. If the crashed component recovers, the rest of the gateway needs to start using it again. Likewise, for load balancing, new components must be able to connect to the rest of the gateway while running. Sessions and transactions that were running on the crashed component may be terminated, i.e., it is not necessary to migrate them to another component.

The architecture design needs to be simple. We have limited resources, and it is simply not realistic to assume that a complicated design is implementable quickly. We can assume that performance is adequate as long as no single bottleneck exists in the design.

The gateway needs to support both WAP and SMS services. New bearers and transport protocols must be simple to add.

# Gateway architecture

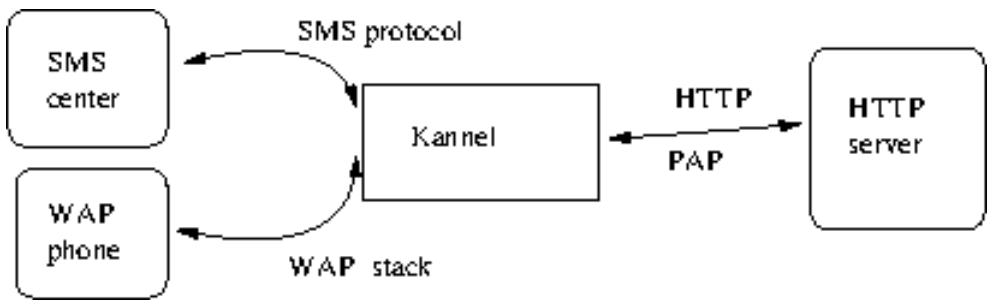
This section explains the gateway architecture. It gives requirements in more detail for perfomance, scalability, reliability, and simplicity of implementation and adding new boxes at run time. It covers the division of gateway duties to processes (bearer, wap, and sms boxes), and explains the duties of each process. It covers interprocess communication between the different boxes, and covers gateway-level communication issues such as heartbeats. It should probably cover the different approaches to the design that were considered, and give the reasons for major (and some minor) design decisions.

## External interfaces of the gateway

The gateway has interfaces to three external agents:

- SMS centers, using various protocols.
- HTTP servers, to fetch WAP and SMS content and to push WAP Content. The pull protocol is HTTP, push PAP.
- WAP phones, implementing the WAP protocol stack and (for push) WAP Push client.

**Figure 4-1. External interfaces of Kannel**




There are several vendors of SMS centers and most of them use a vendor specific protocol. The protocols are fairly similar in spirit, but the details of course differ. The differences are not relevant to this document, though. The protocols essentially follow the following pattern:

- Client logs into the SMS center.
- When an SMS message from a phone arrives, the SMS center sends it. The client is expected to acknowledge it.
- When the client wants to send an SMS message, it sends a request, and the SMS center acknowledges it.
- When the client is done, it logs out.

The various SMS center protocols are implemented using somewhat different approaches within Kannel, but each implementation uses the same interface towards the rest of Kannel. This simplifies the rest of Kannel.

Each SMS center account is bought from a mobile operator. Each account is assigned a number to which SMS messages are sent, and which typically also appears as the sender number for messages sent via that account. (Some connections will allow the account user to set the sender number, though.) There can usually be only one connection to an account at a time. This restricts Kannel's design somewhat. Multiple concurrent connections would allow for higher performance and reliability.

The HTTP protocol is a fairly pure request-response protocol. The client connects to the server, sends a request, and then the server responds and this completes the transaction. Multiple requests may be done over the same connection, for performance, but the basic flavor of the protocol is still the same.

The WAP protocol stack and WAP Push have already been briefly described above.

In order to achieve maximum throughput, Kannel needs to be able to communicate with each external agent independently from each other, i.e, by multitasking internally. For example, it is not good enough to read one request via SMS or WAP, fetch the requested content via HTTP, send the content to whoever requested it, and only then read the next request. This might be fast enough, if the HTTP servers were extremely fast, but they are not. Each HTTP transaction can potentially take an indeterminate time, without failing, and Kannel must not let one slow request prevent every other client from getting service.

What Kannel needs to do, then, is read requests from each external interface as fast as possible, and keep them in an internal queue. Then it needs to make the HTTP requests for the contents as fast as possible, and then send the responses back to the requesters. Depending on how fast the HTTP requests take, the responses will be sent to the requesters in different orders. Things needs to be designed so that this works.

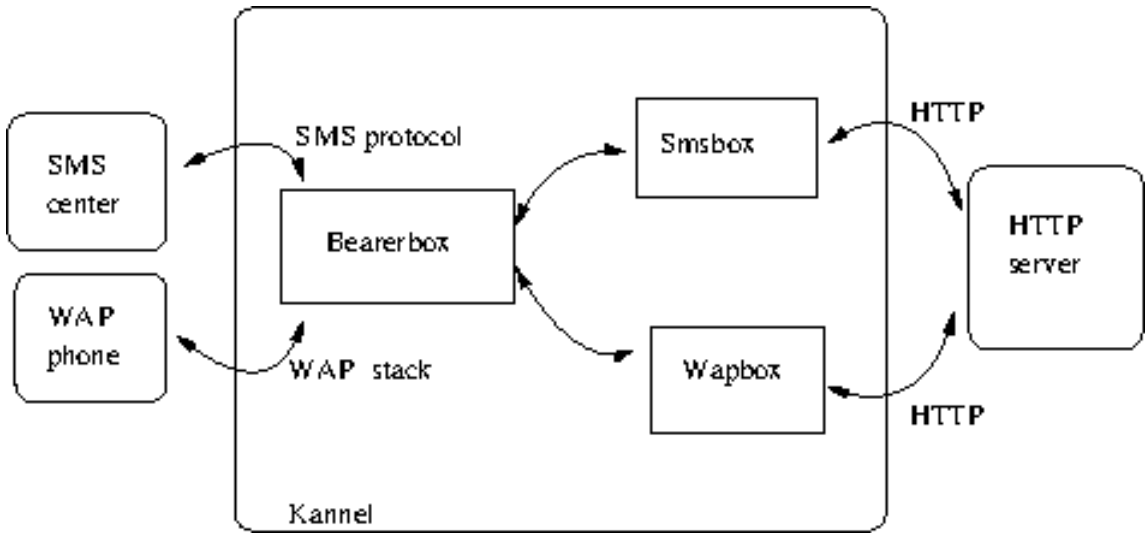
There is a potential reliability problem in this kind of design: if Kannel reads many requests into an internal queue, and crashes, the requests will be lost. This can be expensive to the clients, if they use SMS (whether for SMS based services or for WAP), because each SMS message costs money when it is received by Kannel, not when Kannel sends the response. Ideally, Kannel should keep the list in persistent memory (on disk), but it does not do so at the moment, because of implementation complexities.

## Division of duties to processes: the boxes

Kannel divides its various duties into three different kinds of processes, called boxes, [1] mostly based on what kinds of external agents it needs to interact with:

- The *bearerbox* implements the bearer level of WAP (the WDP layer). As part of this, it connects to the SMS centers. Kannel currently implements SMS only as a WAP push bearer. When it does this fully, its SMS gateway functionality will have to interact with the WDP layer: it needs to be possible to use a single SMS center connection for both textual SMS based services and as a WAP bearer.
- The *smsbox* implements the rest of the SMS gateway functionality. It receives textual SMS messages from the bearerbox, and interprets them as service requests, and responds to them in the appropriate way.
- The *wapbox* implements WAP protocol stack and WAP Push (an application level protocol). If wapbox is used for pushing, it is called *Push Proxy Gateway* or *PPG* Another term for fetching data is *pulling* For Kannel box structure for pulling, see Figure 4-2, and for pushing, see Figure 4-3.

**Figure 4-2. Boxes of pull Kannel**

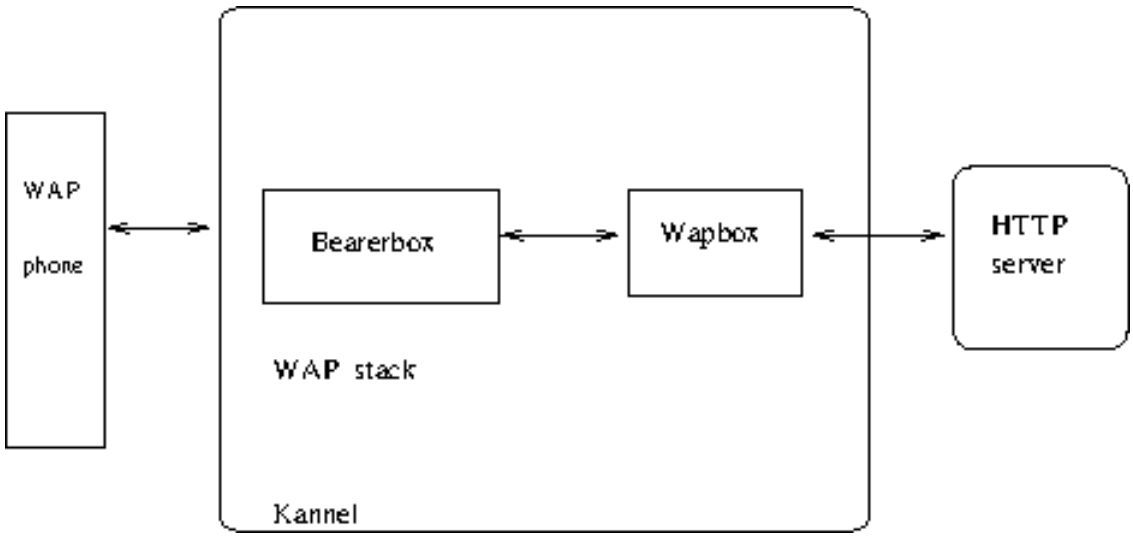



There can be only one bearerbox, but any number of smsboxes and wapboxes. Duplicating the bearerbox is troublesome. If there were multiple bearerboxes, they would still have to be known by the same IP number for WAP phones, which needs help from network routers. Also, each SMS center can only be connected to by one client. While it would be possible to have each SMS center served by a different process, this has been deemed not to give enough extra reliability or scalability to warrant the complexity.

Having multiple smsboxes or wapboxes can be beneficial when the load is very high. Although the processing requirements as such are fairly low per request, network bandwidth from a single machine, or at least operating system limits regarding the number of concurrent network connections are easier to work around with multiple processes, which can, if necessary, be spread over several hosts.

Each box is internally multithreaded. For example, in the bearerbox, each SMS center connection is handled by a separate thread. The thread structures in each box are fairly static: the threads are mostly spawned at startup, instead of spawning a new one for each message.

**Figure 4-3. Boxes of push Kannel**




## Making sure things are working: heartbeats

In addition to shuffling packets between the phones (directly or via SMS centers) and the other boxes, also keeps track of the *heartbeat* of each box. Each box sends a heartbeat message, essentially saying 'I am still alive', and the bearerbox will keep track of when each box has sent it last. If a box stops sending heartbeat messages for too long a time, the bearerbox will close the connection to it.

A parameter to the hearbeat message is the *load factor* of the box. The bearerbox uses this to decide which box it should send each package to. If all boxes were alike, a simple round-robin system would usually work, but this is not something the bearerbox can assume.

# The Bearer Box

This section explain how the bearer box works: its internal architecture with messages, message queues, thread structure, heartbeats inside the box, and how communication between internal and external modules happens.

At the moment, the bearerbox implements only UDP as a bearer for the WAP protocol stack. In the future, it will support SMS messages as well. The bearerbox already implements the necessary SMS center connections, but they are used for SMS gateway functionality only, and are thus ignored for this

thesis. (This is true in spite of an implemented WAP Push. Using SMS as a pull bearer requires reassembly of SMS messages and routing them to one of wapboxes.) [2]

The bearerbox receives UDP messages from the phones, sends them to wapboxes, receives reply messages from the wapboxes, and sends the corresponding UDP messages to the phones. Since there can be several wapboxes connected to a single bearerbox, the bearerbox needs to route the UDP packets so that all packets from the same phone are sent to the same wapbox. In practice, the routing problem is simplified so that all packets from the same IP number go to the same wapbox, even though it is not guaranteed that the IP number hasn't been assigned to a new phone. Phones are allocated IP numbers dynamically - when they make the data call, they get an IP number, and when they terminate the call, the IP number is released and can be allocated to the next caller. Thus, it might make sense for the bearerbox to know when a WAP session or transaction is finished so that when the new phone uses the same IP number, it can be assigned to a different wapbox with a lower load. In practice, however, this is unlikely to have much benefit, so the added complexity is useless. It will be added, of course, if benchmarks later show it to be useful.

The bearerbox uses several internal threads and message queues. Figure 4-4 shows their structure. There can be multiple UDP sockets open, since the WAP protocol uses different UDP port numbers to identify the type of message: whether the message is for a session mode transaction or a sessionless transaction, whether it uses the security layer, etc. The bearerbox has a separate thread of type `udp_receiver` for each such UDP socket. That thread reads the UDP packets, converts them to message objects used internally within Kannel, and puts those into the `incoming_wdp` queue. All `udp_receiver` threads use the same queue.

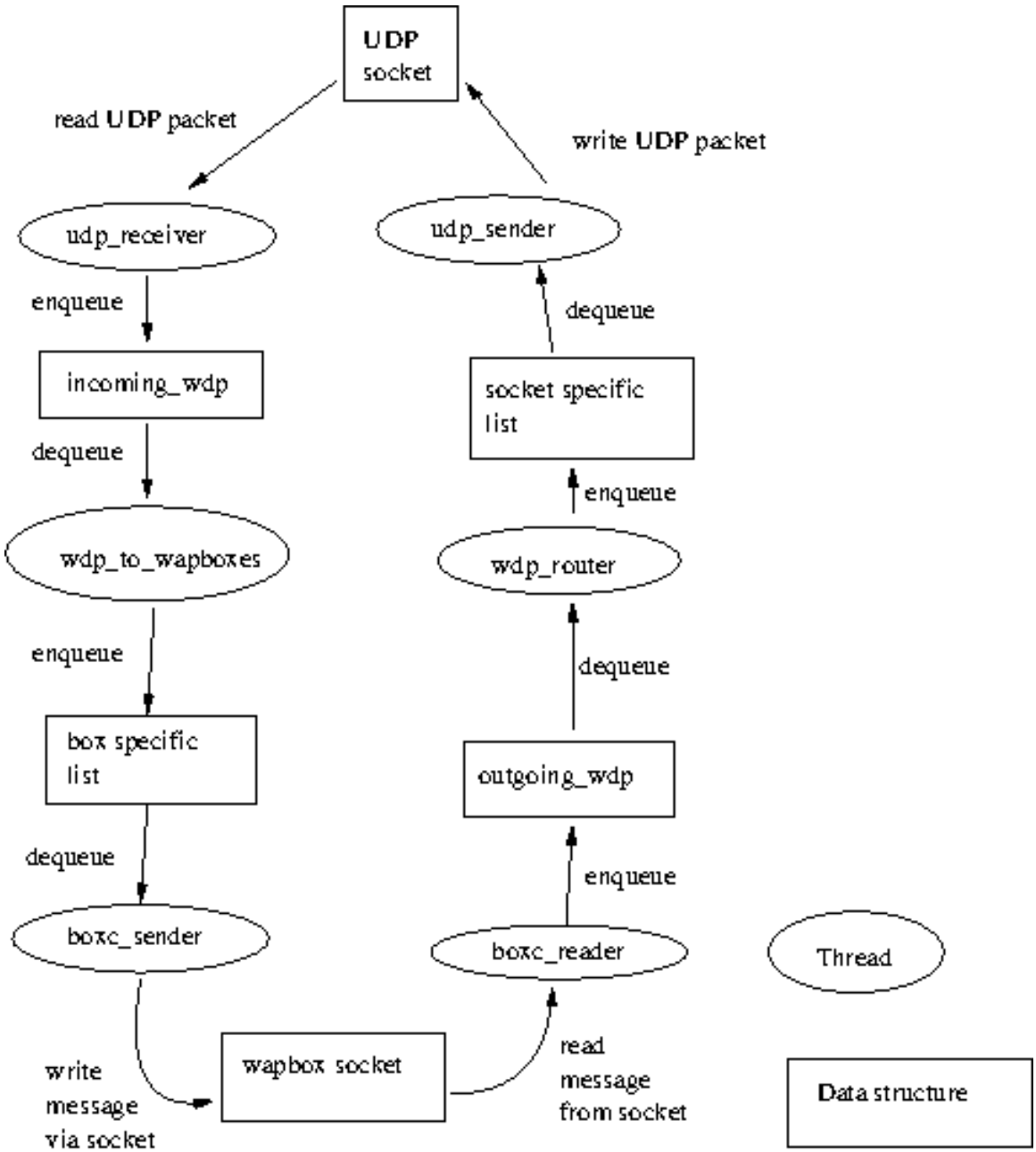
The `wdp_to_wapboxes` thread removes the messages from the `incoming_wdp` queue and routes them to the correct wapbox. Each wapbox is represented inside the bearerbox by a separate message queue and two threads: `boxc_sender` and `boxc_reader`. The `wdp_to_wapboxes` thread thus moves messages from the `incoming_wdp` queue to the wapbox specific queues. The `boxc_sender` thread removes the messages from the wapbox specific queue and sends them to the wapbox proper via a network connection.

The message traffic in the other direction is a mirror image. The `boxc_reader` thread reads messages from the wapbox, via the network connection, and puts them in the `outgoing_wdp` queue. The `wdp_router` thread removes the messages from that queue and puts them in the queue specific for the socket, so that they will be sent from the correct port. The `udp_sender` thread then removes messages from the socket specific queue, converts them into UDP packets, and sends those to the phones.

Multiple queues within the bearerbox were used because routing can change at any time. If a wapbox disappears, all messages waiting to be sent to it should be sent to other wapboxes instead, so that they can be dealt with in a useful manner.

The bearerbox tries to balance the load between different wapboxes. This is implemented using heartbeat messages sent by the wapboxes to the bearerbox. Each wapbox computes its own load factor. In the current implementation, this is the number of open and pending HTTP transactions it has. Periodically, the wapboxes send a heartbeat message containing their load factor to the bearerbox. The bearerbox remembers the latest load factor it got from each wapbox and assigns each new phone to the wapbox with the lowest load factor. This keeps the wapboxes balanced approximately evenly in the long run. To make sure a sudden surge of new phones won't all be assigned to the same wapbox, the bearerbox increments the load factor it stores for each wapbox when it assigns a new phone to it.

**Figure 4-4. Bearerbox architecture**



# The WAP Box

This section explains how the WAP box works. It lists the features of WTP, WSP and WAP Push that are implemented. It covers internal architecture: data structures (state machines, events), thread structure, code modules, inter-module and inter-state machine communication, and probably explains the preprocessor trick used to ease the implementation immensely.

When pulling, wapbox reads messages from the bearerbox, maintains internal state for each client and makes HTTP requests on behalf of clients. It responds to messages according to WAP specs. The basic duties are pretty simple, but things become somewhat more complex when need to deal with large loads.

Only WTP and WSP are implemented. WTLS exists as a patch, but is not covered by this thesis. Only the UDP bearer for WDP is supported at the moment, which means that the Wireless Control Message Protocol (WCMP) is not implemented.

Push can be confirmed or unconfirmed. Unconfirmed is simple: PPG sends the push content to the bearerbox (essentially asks it to do a sms push). If confirmed push is session-oriented, the gateway first asks the phone to establish a session with it. PPG maintains session and push data for initiators and send confirmations (result notifications) to them, if they have asked it. After both kind of pushes, Kannel *can* work as a pull gateway.

Both unconfirmed and confirmed push are implemented, but only unconfirmed one is tested with a real phone. Wapbox does push over SMS, but resulting fetch uses an IP bearer.

## Thread structure

Each WAP protocol stack layer has its own thread. Push OTA protocol implementation is divided between OTA layer (requests) and application (indications and confirmations) layer. There are two threads corresponding PAP protocol, one communicating with OTA layer and other accepting push requests from a push initiator (Session- oriented WSP and confirmed push, these are treated concurrently, because, as mentioned before, WAP Push is followed by a pull.):

- Bearerbox communication layer. This layer corresponds to the WDP layer inside the wapbox; network related parts of WDP are implemented in the bearerbox. Wapbox segmentates the push content before it sends it to the bearerbox, and selects the bearer. This layer is used for both pushing and pulling.
- WTP responder layer. This layer exchanges messages with the bearerbox communication thread and the session mode WSP thread, and manages WTP responder state machines. Messages coming from the phone are routed to this layer. This layer is used for pulling.
- WTP initiator layer. This is similar to WTP responder, but now the gateway starts the transaction. This layer is used for pushing.
- Session mode WSP layer. This layer exchanges messages with the WTP responder, WTP initiator, the application layer and OTA layer, and manages the WSP session, method and push state machines. This is used for both push and pull.

- Application layer. This layer exchanges messages with two WSP layers and makes HTTP requests on their behalf. In addition, it implements indications and confirmations of push OTA protocol for confirmed push, sending them to OTA layer. So it is used btoh for pushing and pulling.
- HTTP client layer. This layer implements the actual HTTP requests. It is independent of WAP protocol stack, and only implements the HTTP protocol. This layer is used for pulling, and it belongs to the application layer.
- OTA layer. This layer receives messages from PAP layer and makes OTA requests to sessionless and session mode WSP layers. It is used for pushing.
- PAP layer. This layer accepts HTTP or HTTPS requests from PI and it is used for pushing. If PPG is not configured for HTTPS, there are three threads: `ota_read_thread`, `http_read_thread` and `pap_request_thread`.If it is, there are three or four threads: `https_read_thread` is now mandatory, and `http_read_thread` is optional.

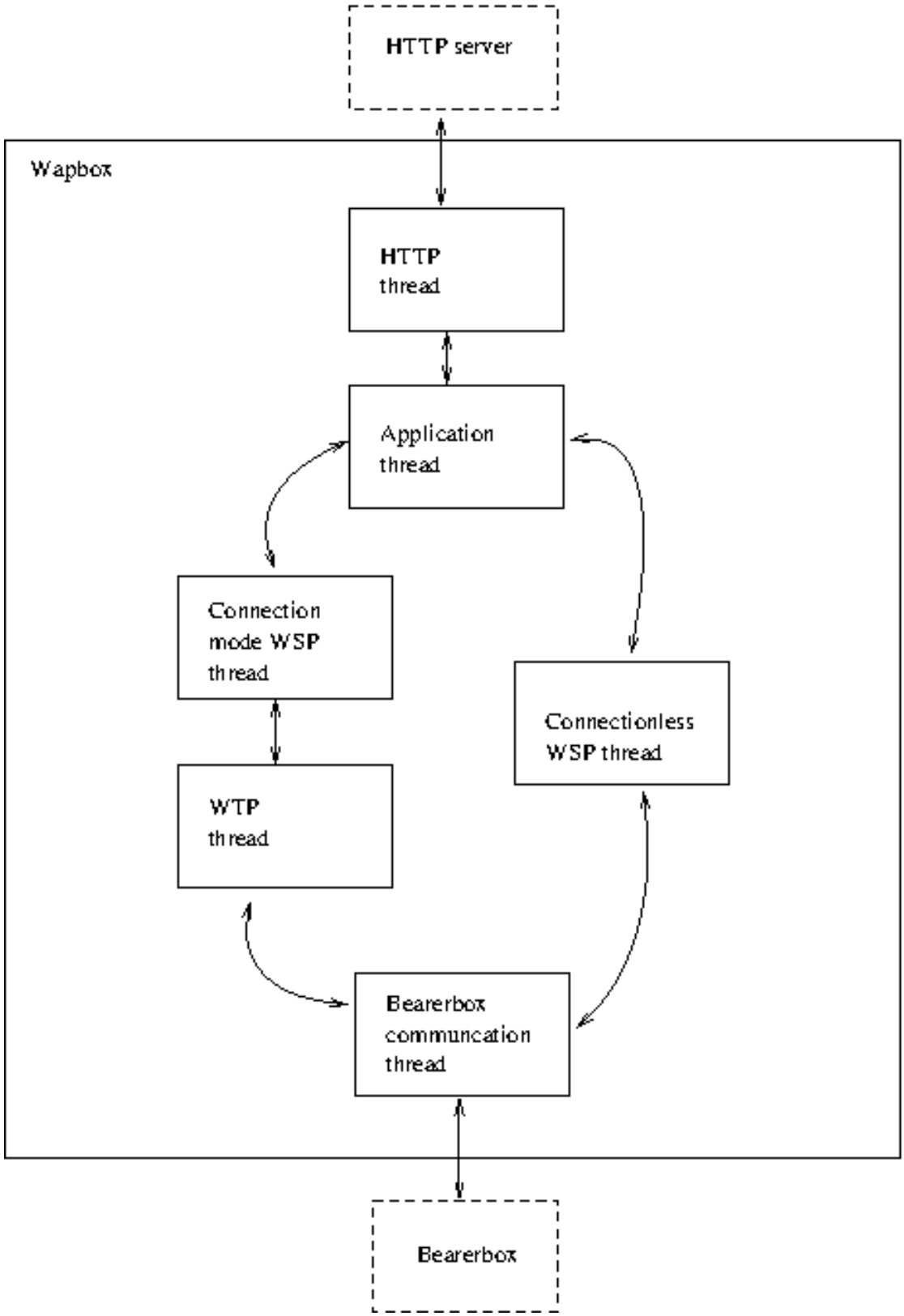
  `ota_read_thread` communicates with application layer. It sends push confirmations (called result notifications) to the push initiator.

  `http_read_thread` and `https_read_thread`accept pushes from the push initiator.

  `pap_request_thread`parses MIME content sent, compiles pap control document, implements required PPG services and converts push content to the tokenized form. It informs push iniatiator, by sending relevant PAP messages to it, errors happened during these processess and communicates with OTA layer.

Push application (PAP and OTA layers) layer are in principle independent of WAP protocol stack, but they do use WTP initiator and WSP pushing facitilies. This means that only WSP session facilities are common with push and pull. For usage of threads when pulling, see Figure 4-5 and when pushing, see Figure 4-6. Here arrows marked with ’1’ refer session establishment, ones marked with ’2’ content pushing and ones marked with ’3’ push confirmation.

**Figure 4-5. Wapbox thread structure for pulling**



Sessionless WSP and unconfirmed push services use only a part of whole stack:

- Bearerbox communication layer. This layer is always the same.
- Sessionless WSP layer. This layer exchanges messages with the bearerbox communication layer, the application layer and OTA layer. It has no state machines as such, so it is very simple. It is completely independent of the WTP and session mode WSP layers, despite the name similarity. It can do both push and pull.
- Application layer. In this case, this layer is used only for pull: the phone sends no confirmation messages to the server.
- OTA layer. This is similar for confirmed and unconfirmed pushes.
- PAP layer. Now `ota_read_thread` is missing; it is used for receiving phone confirmations.

All communication between internal (ones communicating with other Kannel layers) threads is via message queues. These layers (and therefore the threads) send events to each other, no other form of communication is used. Only the event data structures are exposed by each layer: all other data is internal to the layer, and is only manipulated by the single thread that implements that layer. This reduces the need for locking datastructures, which both simplifies the program code and makes execution faster.

One layer of WAP Push (PAP) forms the external interface to the push initiator. Therefore it includes a HTTP server thread, which has an external originator for its events. In addition, Push implementation has an internal layer (OTA).

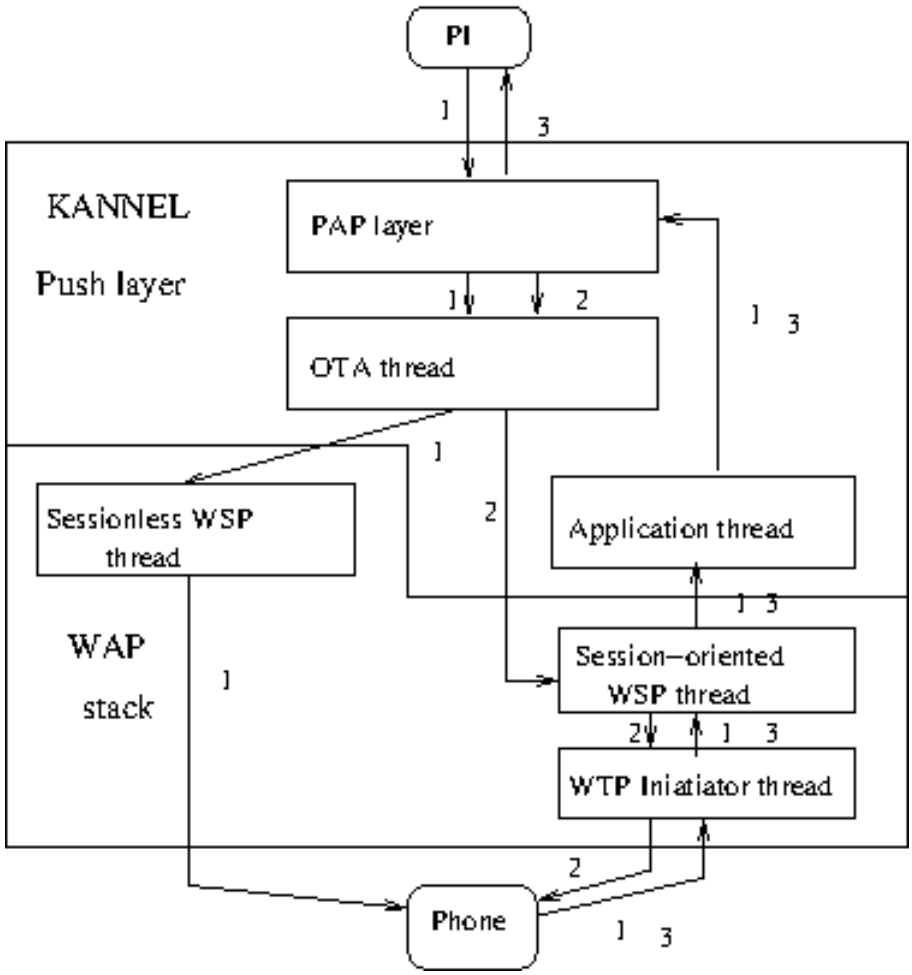
The events used correspond to the ones used in the WAP protocol specification. Events related PAP protocol correspond attributes of XML document.There are minor differences due to implementation details, but it has been a goal to make the implementation follow the specification as closely as possible to reduce errors. However, making push over SMS requires adding quite many additional fields, but these are well separated from ones corresponding protocol primitives.

An earlier design for the gateway spawned a new thread for each incoming WDP packet, had all data structures available to all threads, and had elaborate locking to get things to work. This proved too hard to get working in the everyone-hacks-everything style of program development adopted by the project because of its open source nature. Experiments also showed that it would have been too expensive, computation wise, at high levels of usage.

The current design has a static thread structure. All threads are started when at program startup, and remain running until the program stops. There is a potential scalability problem on machines with a larger number of processors than the wapbox uses: many of the processors will be completely idle. It is, however, possible to run multiple wapboxes, so it should be easy to utilize them anyway.

The implementation of each layer is basically the same: each has a queue of incoming events, to which other layers append events. The layer extracts an event from the queue, handles it, and possibly sends other events to other layers. The only locking needed is when accessing the event queue for each layer. The queue operations are very fast (constant time), so the time spent waiting for a lock is short.

**Figure 4-6. Wapbox thread structure for pushing.**




## Implementation of protocol state machines

The WTP and connection mode WSP layers are implemented in terms of *state machines*, as described in the WAP protocol specification. At the source code level, there are a number of various options for implementing state machines. To ensure the correctness, however, it is necessary to keep the source code easy to compare to the actual specification, and most options would obscure the actual protocol related parts of the implementation with unnecessary detail of state machine implementation. We have therefore taken an approach where we describe the state machine in the source code using a macro language (the C preprocessor) and use that to transform the description into compilable and efficient C code.

See *C Preprocessor Trick For Implementing Similar Data Types* for more information.

## Implementation of push sessions

WAP Push architecture makes possible for a push initiator to establish a connected session with a phone, so that it is unnecessary to reconnect every time. This session data is stored in a data structure similar to protocol state machines. Currently connectionless pushes are, too, implemented using a push machine. This will, however, change.

## Efficient implementation of HTTP requests

It is necessary for a WAP gateway to implement HTTP requests at high usage levels efficiently. This is the only useful thing the WAP pull gateway does on behalf of the client, and there are potentially a very large number of clients, so the quality of the HTTP implementation affects gateway performance a lot.

The straightforward solution would be to implement each HTTP request in a separate thread. This keeps the implementation simple, but is somewhat expensive in computation resources. For example, if there are ten thousand concurrent users, each making a new request every fifteen seconds, on average, and each request taking one second, on average, there are about 670 concurrent requests at any one time. On Linux, each thread uses 8 kilobytes of kernel memory, minimum, so 670 threads would use over 5 meagbytes of extra memory, in addition to userspace memory requirements such as stack. In addition, starting and stopping threads has a cost, and having lots of threads will cause more context switches, an additional CPU cost.

Thus, while it works well at low usage levels, the straightforward solution will cause too much overhead at high usage levels. It is thus necessary to implement HTTP as a static number of threads, for performance reasons, even if it somewhat complicates the implementation.

This is similar to the ‘C10K’ problem (*The C10K problem*) known to implementors of HTTP servers. Even though the hardware is more than fast enough to handle ten thousand simultaneous clients (ten thousand concurrent HTTP requests), the operating system and HTTP server software have designs and implementations that won’t scale up to that high loads. A WAP gateway is very much similar to HTTP servers in this regard.

The basic steps in making an HTTP request are, expressed in network operations, are:

1. Look up the IP number for the HTTP server. This involves making a Domain Name Service (DNS) query, which can take up to several minutes, if the name server is slow or does not respond at all. Due to stupidities in C library interfaces, only one thread in a process can do a name lookup at a time. This will be discussed in more detail in the Section called *Making concurrent domain name lookups*.
2. Open a TCP connection to the HTTP server. This can take some time, if the server is slow or the network is congested.
3. Write the request. If the request is very large, this can again take quite a while, but for typical requests, it will be a single TCP packet. For the writer, the operation finishes when the data has been copied from the writer’s buffers to the operating system network buffers, so typically this is very fast.
4. Read the reply. Even if the reply is fairly short (and for WAP pages it almost always is only a few kilobytes), it can take several seconds for the reply to arrive from the HTTP server to the client.
5. Close the connection. This should be very fast, since the client can continue processing immediatly and leave it to the operating system to negotiate closing with the HTTP server.

Between these network operations, the HTTP client only waits. It has no other processing to do (the gateway as a whole might have other processing to do, but we're talking about the HTTP client thread). Thus, it won't even help to add more processors to the machine: even a single processor can execute quickly any number of idle threads.

What does cost, however, is that every time one thread wakes up, it needs to be scheduled, and all its registers and possibly other data needs to be loaded to the processor. This takes processing time, and the more threads we have, the more often this will happen. If we only have one thread, it will be running all the time, since at any one time presumably at least some of the HTTP connections will be active, so it does have something to do. This will mostly eliminate context switch overhead on a multi-processor machine, which can allocate a processor for HTTP only.

A single thread might, however, be too little at very large loads. If there are very many HTTP responses coming, more than a single processor can deal with, then it would make sense to have more threads, one per processor. This may happen in the future.

To implement the single thread model, we need to wait for input from several sources at the same time. This is accomplished with one of the Unix system calls `select` and `poll`. These do essentially the same thing, but have different interfaces. They both get a list of network connections and wait until at least one of those has data waiting to be read in the operating system's buffers (meaning that the read operation can be done without the thread blocking). Thus, the thread essentially does this:

1. Wait until there is something to read.
2. Read it into a connection specific buffer.
3. If the buffer has a complete HTTP reply, return it to whoever made the request.
4. Repeat forever.

Further details can be found in the source code.

XXX explains how threads are really used in http

## Making concurrent domain name lookups

The Domain Name Service (DNS) is an integral part of the Internet and implements a directory service that maps textual domain names into IP numbers, which are used in actually routing packages between hosts. It is implemented as a custom distributed database system, and has caching and other features to make it quite efficient.

Unfortunately, the portable C library interface to it, the `gethostbyname` function, does not support concurrency. It is not thread safe, and even its thread-safe and somewhat portable version, `gethostbyname_r`, only protects against multiple concurrent calls, but often does not implement queries concurrently. This is a large efficiency problem, because it can take several minutes for a single request. It is also a security problem, because a malicious user could have the gateway to fetch large numbers of pages with domain names that take the maximum time each to look up. If the gateway can only do one request at a time, this will efficiently prevent any real users from using the gateway.

There are several possible solutions to this. One would be to implement the DNS protocol within the gateway. This is problematic, because `gethostbyname` does more than DNS lookups: depending on how the host the program runs on has been configured, it can also look names up in files in various formats, and it can be quite important for a system administrator that the gateway follows this logic.

It is therefore simpler to run sub-processes to do `gethostbyname` calls. Each sub-process does a single call at a time, but since there can be multiple sub-processes, concurrency is achieved.

This does not, however, completely eliminate the security risk. An attacker could use the feature to cause the gateway to start huge numbers of sub-processes, each taking several minutes to do the host name lookups, and thus making the gateway host run slower because of the increased load. The sub-process implementation needs to deal with this in some way.

# Converting XML languages and WMLScript to binary

The conversion of WML and WMLScript to their binary forms is done by two fairly independent conversion modules. They are fairly simple applications of compiler theory.

The WML compiler gets as its input the WML source code, plus whatever character set information that may have been present in the HTTP headers. It returns the binary form of the WML deck, or an error indication, if the page was faulty. The character set information is needed because the services and phones may generate and accept different character sets. XML, which is used to define WML, specifies Unicode, encoded with UTF-8, as the default, but this is not always practical to use in real life.

In principle, the WML compilation process is very simple. The WML language is very easy to parse and the binary format follows closely the textual one, merely encoding start and end tags, attributes, and other parts of the language in a more compact form. There is, however, a feature that can be used to minimize the size of the output: string tables. When the textual content of the WML page is converted to binary form, it can contain references to a table of strings. As a simplification, think of this as each binary byte either being a character or an index to the string table. The WML compiler decides what goes into the string table. This allows the output text to be compressed: the WML compiler can fill the string table so that the total output of the binary form is minimized. The decision as to how the string table is filled needs to string compression techniques, which complicate the WML compiler somewhat.

Push XML languages (SI, SL, CO) are similar to WML, but their compilation is even simpler (no variables here, for instance). They do not use a string table either, because push documents, which are sent over SMS, *must* be small.

The WMLScript compiler is a more traditional programming language compiler. The binary output format is a bytecode, for which it is easy to write a small interpreter, suitable for a phone. Thus, the WMLScript compiler needs to parse the input file, form a parse tree of it, optimize the tree, then generate bytecode, and further optimize the bytecode. The output is a binary string, which can be sent to the phone as is.

The WML compiler does some simple basic optimizations, such as constant expression elimination, but does not otherwise work very hard to keep the size of the output small. This could be improved, but since WMLScript programs have so far rarely been large, it has not been sensible to put the effort into this.

# Notes

1. *Box* seemed like a nice, non-technical term that should be understandable for marketing people, specially if each box is run on its own host. In this case, each Kannel box corresponds to a physical box, which should be clear enough.

2. Once the thesis is submitted to the university, descriptions of the SMS gateway functionality will be added.

# Chapter 5. Experiences From Implementing and Using the Gateway

This chapter discusses the experiences of the project, in a wholly subjective manner. It is divided into four parts.

Part 1: Discussion of what was done right and what was done wrong in the project, with regard to architecture, implementation, and project management.

Part 2: Discussion of how the open source development model has affected the project.

Part 3: Benchmarks on how the gateway performas at various load levels, and how it recovers from crashing wap and bearer boxes. Experiment designs are explained and results presented and discussed.

Part 4: Discussion of feedback from people using the gateway.

## Subjective evaluation

In this section I try to evaluate the Kannel project and describe things we've done right and things we've done badly.

Most of the problems have been in general project management issues, such as making up and keeping time schedules, and building up the development team at Wapit and the open source development community. During the first year of the project, approximately from June 1999 to August 2000, there was intensive pressure, from Wapit, as far as the development speed was concerned. At first, there was pressure to get at least something to work, to get something to sell. When the SMS gateway was ready for production use, in September 1999, the pressure switched to getting WAP working. When this happened, in January 2000, the pressure switched to implementing the whole specification and ensuring compatibility. These needs were mostly filled by the end of August 2000, or at least the features were there, even if the software still had some bugs.

As soon as the first production installation was made, in September 1999, it was also necessary to fix bugs and implement missing things that the production installations needed. This slowed actual development down somewhat, but in general the quality of the software is higher because of early production use. We have been able to concentrate on things that are actually needed, instead of following a feature checklist made up by marketing or by users who hadn't ever used a gateway before.

It has not, however, made it easy to predict development speed, since at any time it may become necessary to throw aside the current development task and fix a customer problem, and the Kannel team has often failed to accurately predict when a version with a desired feature set would be available.

In hindsight, the intense pressure for development speed was probably too intense, and resulted in slower development speed. Although some amount of pressure is good for getting people to work faster, the Kannel team had too much stress, and this resulted in overly optimistic time schedules and when they slipped, in further stress. From a software engineering and management point of view, the only redeeming feature of Kannel's project management is that the software made it to the market sufficiently early and with sufficient quality in order to succeed.

Building the Kannel development team at Wapit has been fairly straightforward. In June 1999 there were three people, including myself, and every couple months until March 2000 the team acquired a new member, and in December 2000 we were six people. Two people have left the team, one to another department inside Wapit, the other to another company. This slow growth of the team has worked out well, even though the almost constant need to train new people has cost somewhat in development speed.

Building an open source development community around Kannel has proved to be a much harder task. Partly this is because Kannel is of interest to a fairly small group of people, but mostly it is because specially at the beginning the development discussions were not open, in practice, to people outside Wapit. Many of the discussions were held face to face between Wapit developers, or on internal Wapit mailing lists, and this effectively shut out outside developers. The situation has since changed, and the development community is now slowly growing.

Our choice of open source license, which essentially allows anyone to do anything, as long as they credit Wapit in their documentation, may have had a negative impact on the growth on the development community, but this has not been investigated. Our license does not require other developers to publish their changes, and there are several other companies working on their own versions of Kannel, without submitting back any changes to the Kannel project. This is acceptable, according to the license, but may have cause the development community to grow slower. On the other hand, if a more forceful license, such as the *GNU GENERAL PUBLIC LICENSE* (GPL), had been chosen, the other developers might not have wanted to use Kannel at all, since it is often perceived that the GPL makes it impossible to build a profitable business. It would be interesting to investigate this in the Kannel context, but we have not had sufficient time for this yet.

The software development process itself has been loosely based on the spiral model although adapted to an open source development model, with rather fuzzy goals for each iteration. In short, the philosophy has been to get at least something working, so that people can try it out and even use it in production, and then improve and possibly rewrite it to make it better. Unlike many projects with this approach, the Kannel project has actually spent much time on the rewriting: code gets rewritten once it gets too buggy or it fits too badly with the parts around it that have changed. We have tried to keep the general architecture and internal interfaces clean, and thus rewrites have mostly been local. An excellent example of this is our HTTP implementation: the first one was made quickly, and served well for almost a year, and once its bugs and limitations in speed and features became problematic, it was rewritten completely from scratch without affecting the code calling more than by trivial calling convention changes.

A small, but very important things we did correctly was to set up a ‘nag’ script: a simple script to compile the current version, directly from the version control system, and mail the developers any error and warning messages. In principle, this script does what every developer should do, but it is hard to force developers to use a particular set of compilation options, and even if they are willing, it is easy to forget one. The script helps by doing it automatically for all developers, and by doing it systematically every night. Additionally, when the script was run on multiple platforms, every night, it helped find several portability problems.

Later, we added some automatic test cases, which can be run by each developer. Even though the tests are simple, they do check for all the basic features of Kannel, and make sure that a change won’t break those. As time goes by, we add more tests, making it easier to catch more and more mistakes.

We recommend the nightly automatic compilation test and the automatic testing for all projects.

Like most open source projects, we have been using a bug tracking system that anyone can browse. Our use of it has been unsystematic, though, with most bugs reported via email on the development mailing list. This has, at times, caused bugs to be ignored or forgotten. As Kannel gains users, bug tracking will

have to become more systematic.

Quality control in general has received rather little attenation from Kannel developers. Except for the simple automatic test suite described above, the general approach of the developers has been to make their code or changes available, via the version control system, as soon as possible, so that others can participate in the testing. This is partly good, because the developers do not even have access to all mobile devices to do a complete test, and partly bad, because those areas of Kannel that are hard to test or require specialized hardware, such as the SMS center protocol implementations, have been tested fairly lightly. On the whole, things have worked out, though.

# Effects of choosing to be open source

An open source project, with an ever changing group of developers, each with their own goals for the software, is by necessity different from a traditional software project. The major effects of being open source for the Kannel project have been more varied testing, more people doing debugging, and a need to keep the source code simple.

WAP is being used all around the world, and implemented on many phones that only work in certain parts of the world. Thus, for Kannel to be compatible with all phones, it needs to be tested by people around the world, and in a traditional software project this would be quite hard to do. As an open source project, Kannel has users from around the world, and they have helped in testing Kannel against almost all WAP capable phones in the world. Even though the testing is informal, i.e., there is no specific set of tests run by the users for each new Kannel version, it is quite effective: as soon as a new and incompatible phone becomes available, or if the developers break Kannel for some phone, the development mailing list gets bug reports.

This informal and distributed approach to testing has been applied to most parts of Kannel development. The assumption is that if we have enough users, with different usage patterns, all or most code paths are exercised and if there are problems, we will hear about it. This, of course, flies in the face of conventional software engineering, but seems to work for us as it does for many open source projects.

The distributed approach also applies to debugging. One of the popular slogans for open source development is “when you have enough eyes, all bugs are shallow” (see *The Cathedral and the Bazaar*). This does not mean that all bugs are easy to solve, but if there is an urgent problem with Kannel, there will usually be many people working on finding it. They all work independently, but communicate about theirs findings and share theories. The end result is that the process of finding a particular bug is sped up significantly compared to having only one or two people working on it.

With many people working on same parts of the code together, communicating only over e-mail, it is important for the source code and program structure to be simple, so that everyone can understand it and so that fewer mistakes are made because of, for example, complicated interfaces or arcane programming tricks. Those parts that are complicated or tricky also tend cause more questions and more bugs.

The major impact of being open source, however, is more time spent communicating over e-mail. In a traditional project, much information is shared only orally, but since e-mail is the only common communication medium for the Kannel project, more time is spent reading and writing e-mail. On the other hand, much less time is spent sitting in meetings, and on the average the communication cost is probably about the same for Kannel as it would be if the project wasn’t open source.

# Benchmarks

XXX This section will be written after I have some time to do some benchmarking. It will probably be the last one I write.

# User feedback and experiences

XXX I will have to send out some questions to users@kannel.org, devel@kannel.org and talk to our sales and marketing people at Wapit. Time enough to do that later.

# Chapter 6. Plans for the Future

This chapter discusses the plans for the future of the gateway: implementation of missing features (specially WTLS and WCMP), more bearers, updates to new versions of WAP. It also outlines needs and plans for higher reliability (keeping transactions and sessions alive over box crashes) and load balancing (migrating jobs between boxes according to load).

## New features

The Kannel gateway has been ready for production use since September, 1999 as an SMS gateway, and June, 2000 as a WAP gateway. This does not mean that it is finished. On the contrary, it is only the beginning, and many important features are missing. Being merely important, they are not necessary for all production use, and thus they can be implemented even after the software is being employed. Among the missing features are the WAP security layer (WTLS), features from new versions of the WAP specification (WAP 1.2 and the June 2000 versions), and using other WAP bearers than UDP, specially SMS messages.

Some of the work for these is already going on. There is an implementation of WTLS for Kannel, provided by 3UI.COM, but this has not been integrated into Kannel source code, because of legal and technical issues. The legal ones, involving patents and usage and export restrictions on cryptographic software in USA and other countries, are solved, or can easily be solved, and WTLS is now being added to the Kannel source repository.

Using SMS messages as a bearer for WAP, i.e., implementing WDP over SMS, will require implementing the Wireless Control Message Protocol (WCMP) as well. Since SMS messages are so small compared to UDP messages, it is then also necessary to implement segmentation and reassembly of WDP packets into multiple SMS messages, and this will result in a substantial WDP layer. Wapbox does segmentation for WAP Push, but reassembly of received SMS remains to be implemented.

## Better quality

In addition to new features, Kannel can be improved as regards to security, reliability and speed.

At the moment, Kannel is open to several forms of denial-of-service attacks. This is a security problem. The most common form of these potential attacks is feeding it too much data, causing it to run out of memory, and crash. Kannel design needs to be adapted so that it won't try to buffer too much data read from the network. This will require changes in several parts of Kannel, but they will be small and mostly local.

Many Internet server programs written in C suffer from a 'buffer overrun' type of security problem: the program has a bug that allows an attacker to feed it enough data so that it overruns the bounds for an array. The C language design is such that this kind of programming mistake is easy to make. Kannel uses an internal abstraction, called the octet string, which reduces the risk for this problem to the level of any programming language with array bounds checking.

The main reliability problem Kannel has is that it will lose SMS messages it has received from an SMS center if it crashes. Kannel should store the messages in persistent memory so that it can recover from a crash without losing messages. This is important for SMS use, because each message potentially costs money to the end user. WAP session data will also be lost, but this is unimportant, and will easily be re-established by the phone, in some cases invisibly to the user. Of course, when WAP uses SMS as a bearer, it should not lose the SMS messages related to WAP, either, even if the session data itself is lost. There are several possibilities for a persistent storage for SMS messages, but this issue has not yet been studied by the Kannel developers.

To improve application reliability, Kannel should allow users using its SMS push feature ("sendsms") to track which messages have been delivered, and which have not. This will require bookkeeping by both the bearerbox and the smsbox, and the design will have to be done carefully to avoid bottlenecks and keep performance up. It is not a good idea, for example, to respond to the sendsms HTTP request only after the message has actually been sent to the phone, since this will require keeping so many more HTTP connections open that socket port numbers will run out at high load.

Kannel is about as fast as the speed requirements made for it in the fall of 1999 dictate: at least a hundred SMS messages per second and a hundred WAP requests per second. In some installations, this is not quite enough at peak traffic levels, and thus Kannel performance will require some attention in the future. It may be necessary to change the architecture somewhat, to require less message passing and less locking, but no benchmarking or profiling has been done yet to warrant changes. It should be possible, however, to speed Kannel up by an order of magnitude, if it becomes necessary. The current design and implementation stress simplicity, not performance.

Migrating jobs between smsboxes or wapbox might improve performance as well. Then if one box becomes loaded much more than the other ones, it could move some of its load to the other boxes, and thus even the load. However, since the transactions tend to be very small, spreading the load only at the start of the transactions is probably enough. This needs benchmarking and profiling.

# Bibliography

*Imperial Earth*, Arthur C. Clarke, 1975.

Science fiction novel featuring mobile computing devices.

*Star Trek*, Gene Roddenberry.

Popular science fiction TV series shown since 1966. Has mobile communication devices.

*The Cathedral and the Bazaar*, Eric S. Raymond.

Essay on how open source development paradigms work. See http://www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/index.html (http://www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/index.html)

*GNU GENERAL PUBLIC LICENSE*, Free Software Foundation.

License used by the Free Software Foundation for the GNU Project. See http://www.fsf.org/copyleft/gpl.html (http://www.fsf.org/copyleft/gpl.html).

*Wireless Application Protocol White Paper*, WAP Forum.

Overview document about WAP. See http://www.wapforum.org/what/WAP_white_pages.pdf (http://www.wapforum.org/what/WAP_white_pages.pdf).

*Wireless Application Protocol Architecture Specification*, WAP Forum.

Describes the general architecture of WAP, the place of the gateway in it, and the various layers of the WAP protocol stack. See http://www.wapforum.org (http://www.wapforum.org).

*C Preprocessor Trick For Implementing Similar Data Types*, Lars Wirzenius.

A C preprocessor trick for handling many similar structured data types, such as packets in a communication protocol, is described and justified. See http://www.iki.fi/liw/texts/index.html#cpp-trick (http://www.iki.fi/liw/texts/index.html#cpp-trick).

*The C10K problem*, Dan Kegel.

A few notes on how to configure operating systems and write code to support thousands of clients. See http://www.kegel.com/c10k.html (http://www.kegel.com/c10k.html).