An Embedded Ubiquitous Control Architecture for Low Power Systems

James Neil Weatherall

Queens' College

A dissertation submitted for the degree of

Doctor of Philosophy

July 11, 2002

Abstract

Improvements in microprocessor technology over the past decade have led to a proliferation of hand-held microprocessor-controlled consumer electronics. Commodity electronic devices which have traditionally used custom embedded chips to function can now be built using low-end general purpose processors. Application areas where micro-chip technology would once have been too expensive or power-hungry are now possible, leading to an ever-expanding range of consumer electronics. As the complexity and number of devices a user carries increases, it becomes increasingly important that those devices cooperate in some way, to avoid the task of maintaining them outweighing their benefits.

Wireless networking technologies such as infra-red and short-range radio have also benefited from improvements in chip technologies, making it possible to embed rudimentary communications capabilities into even the tiniest of devices. Such technologies go some way toward reducing the burden of using several mobile devices, but fall short of providing a complete solution, due to issues of compatibility, efficiency and ease of use.

This dissertation presents a novel architecture named Eucalyptus, designed to allow low power, wireless devices to interoperate on a peer-to-peer basis. The architecture is based around an object-oriented middleware platform tailored to low power environments and to context-aware applications.

The argument of this dissertation is that currently available wireless communications systems are designed for specific target applications and environments and that no one system can be used to allow arbitrary devices to communicate. By contrast, Eucalyptus may be implemented for even extremely trivial devices, allowing them to communicate as peers and become part of any desired application.

This dissertation describes both the Eucalyptus architecture and its component technologies, as well as infrastructural services that extend the useful range of devices. Many of the techniques discussed could equally well be applied to other adverse communication environments.

Preface

Except where otherwise stated in the text, this dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.

No part of my dissertation has already been, or is being currently submitted for any such degree or other qualification.

This dissertation does not exceed sixty five thousand words, including tables, footnotes and bibliography, and does not contain more than one hundred and fifty figures.

Publications

Aspects of the work presented in this dissertation have been published in the following:

1. J. Weatherall and A.Hopper
   "Predator: A Distributed Location Service and Example Applications"
   Proceedings of Cooperative Buildings 1999, Springer-Verlag Lecture Notes in Computer Science. Springer-Verlag Technical Report, October 1999

2. J.N.Weatherall
   "The R2 Low-Power Messaging and Rendezvous Layer
   Power efficient communications for Tetherless Computing"
   Proceedings of SIGOPS European Workshop, September 2000

3. James "Wez" Weatherall and David Scott
   "Mobile Computing with Python"
   Proceedings of the 9th International Python Conference, March 2001

4. James Weatherall
   "Middleware for Low Power Mobiles"
   Poster presented at the Mobile Middleware Workshop, Middleware 2001, November 2001

5. James Weatherall and Alan Jones
   "Ubiquitous Networks and their Applications"
   IEEE Wireless Communications Magazine, February 2002

6. James "Wez" Weatherall
   "A Ubiquitous Control Architecture for Low Power Systems"
   Proceedings of the International Conference on Architecture of Computing Systems, Springer-Verlag Lecture Notes in Computer Science. April 2002

Acknowledgements

I would like to thank my supervisor Andy Hopper, for his guidance in this work. I would also like to thank the staff of AT&T Labs Cambridge, in particular Sai-Lai Lo and Duncan Grisby and the members of the PEN project, for their invaluable advice and support, as well as Alan Jones, Andy Harter, Quentin Stafford-Fraser and Gray Girling for taking time to proof-read and comment on this dissertation. Anthony Rowstron and John Bates deserve thanks for their help in the early stages of this work.

Thanks are also due to the members of the Laboratory for Communications Engineering, to my friends at Queens' College, and to my assorted housemates over the years, for putting up with me all this time!

Finally, thanks to my family for their support, and, last but not least, to Jen Yee.

Contents

Chapter 1

Introduction

Improvements in microprocessor technology over the past decade have led to a proliferation of handheld microprocessor-controlled consumer electronics. Relatively powerful computing capabilities can now be cheaply embedded in commodity devices. Consumer electronics which have traditionally used custom embedded chips to function can now be built using low-end general purpose processors and Read-Only or FLASH memory chips. Application areas where micro-chip technology would once have been too expensive or power-hungry are now possible, leading to an ever-expanding range of consumer electronics. As the complexity and number of devices a user carries increases, it becomes increasingly important that those devices cooperate in some way, to avoid the task of maintaining them outweighing their benefits.

Wireless networking technologies such as infra-red and short-range radio have also benefited from improvements in chip technologies, making it possible to embed rudimentary communications capabilities into even the tiniest of devices. Such technologies go some way toward reducing the burden of using several mobile devices, but fall short of providing a complete solution, due to issues of compatibility, efficiency and ease of use.

This dissertation presents a novel architecture designed to allow low power, wireless devices to interoperate on a peer-to-peer basis with conventional computer systems, and with each other. A thesis of this dissertation is that systems of this kind must use protocols which are both power-efficient and standard, and that custom protocols optimised for particular tasks are unnecessary for a broad class of target applications. A general purpose object model and APIs based on the OMG's CORBA specification forms the core of the system. Remote Procedure Call and Event-oriented APIs are provided and mapped to appropriate low-level wireless communication constructs by the implementation.

In order to be of practical use, both mobile and static wireless devices must be capable of operating for prolonged periods using a single battery charge[1]. This typically requires that devices spend most of their time powered down, in an "idle" or "deep sleep" mode, during which no processing or communication may take place. A suitable low-power messaging protocol is described, in which devices periodically "wake up" and advertise their services, spending the rest of their time in deep sleep. A side effect of this solution is that messages sent to low-power devices may be delayed significantly at the sender, while the messaging layer waits for the recipient to awaken. It is argued that the resulting latency is an intrinsic feature of systems using "deep sleep" modes, and that therefore conventional Local-Area Network protocols are unsuitable for low-power networks. The use of a "middleware" layer supporting multiple programming abstractions and implementing these natively using appropriate low-level mechanisms is advocated to provide both low-power and low-latency operation without the need for application-specific extensions.

An architecture is also proposed, based upon this middleware layer, to address the problems of discovery, naming and reservation of devices in an ad hoc wireless environment. Ad hoc discovery of devices presents security problems in establishing secure communications, authenticating devices and authorising operations upon them. Issues of authentication and encryption are dealt with in other related work [67] and so are not explored in detail here. Authorisation of operations is discussed, in particular where it affects the resolution of conflicting requests.

Some simple automation models are proposed through which novice users may apply context-specific rules to their environment. Previous work in this area has tended to concentrate on technical issues such as specification and evaluation of composite event algebra and efficient filtering and distribution of events. The

[1] It may also be possible to operate from environmental power sources such as sunlight or movement.

work presented here draws on existing event-based techniques for context-aware computing and attempts to provide a more intuitive environment.

The work presented in this dissertation has been both enabled and motivated by developments in the areas of embedded systems and wireless networking made over the past decade. Some of the effects of these advances are discussed in the following sections.

1.1 Embedded Systems

The “microprocessor revolution” which reduced previously building-sized machines down to the size of a typewriter and thus made home-computing viable has more recently spawned a proliferation of powerful consumer electronics. Improvements in processing and storage technology, following the predictions of “Moore’s Law”, have led to home computers with vastly improved capacity and computing power. At the same time, they have led to similar albeit more subtle changes in the area of embedded systems.

Handheld consumer electronics such as programmable calculators, PDAs and mobile phones have been the most obvious examples of products which have benefited from improvements in technology. Devices which would ten years ago have had a few hundred Kilobytes of memory now have several tens of Megabytes. Microprocessors such as the Motorola 68000 series and the Intel 80486, which were originally designed as high-end processors for desktop systems, are now sufficiently cheap to embed in other micro-chips and sufficiently low-power to operate that they have found their way into traditionally special-purpose, embedded application areas. As improvements in fabrication technology have made previously expensive general-purpose CPUs cheap enough to embed, they have similarly made custom chips cheap enough to make and low-power enough to operate that previously “dumb” devices can now be controlled by microprocessor.

As more features become feasible to embed in commodity microchip technology, devices become ever more complex. Unless new features significantly improve the usefulness of the device, their benefits may well be outweighed by this complexity. At the same time, the ease with which particular functions may be provided increases the likelihood that they will be supported by more than one of a user's devices. By allowing devices to communicate with one another, their collective usefulness to the user can be improved, while also allowing replicated functions to cooperate to avoid confusion.

1.2 Wireless Networking

In much the same way that the "microprocessor" revolution made home computing possible, the more recent "wireless revolution" has led to the popularity of mobile communications devices such as pagers and mobile phones.

Where once the only wireless technology in the home may well have been the infra-red remote controller used to change television channel, home users now take a much wider range of wireless technologies entirely for granted. From radio-based key fobs capable of locking or unlocking a car from several metres away, through pager systems and wireless telephone handsets to computer networks such as WaveLAN and Bluetooth, wireless technologies are becoming increasingly cheap to produce and consequently increasingly popular.

PDA manufacturers have quickly taken advantage of the availability of cheap, low-power wireless systems to enable users to swap data with one another and to backup or synchronise their files as desired, without having to physically connect the devices involved. In the early days of consumer wireless however, the reality was often that each different device would at best use a different, incompatible set of protocols. At worst, the protocols used by different systems would interfere in a destructive manner, causing one or other or both to fail.

In order to create a successful market for handheld devices, manufacturers have been forced to specify and adopt standards which describe in detail the hardware, software and protocols comprising the various wireless technologies. Despite the efforts of these groups to create suitable standard protocols, there has been little tangible progress made to the consumer. One exception is the IrDA standard, which specifies characteristics for devices communicating using infra-red. This standard has benefited from the relative simplicity and cost-effectiveness of infra-red hardware and the minimal requirements of target applications and has become widely adopted. The ability of devices from different manufacturers to interact has helped wireless systems move from being a niche specialist technology to mainstream acceptance. By contrast with IrDA, Bluetooth, a similar standard based on short-range radio, has failed as yet to achieve widespread acceptance. Initial implementations of the Bluetooth standard have mostly either failed to comply to the specification adequately, or have exposed ambiguities in it. It is likely that refinements to the specification and improved compliance of devices will provide the promised levels of interoperability within a few years.

Consumer wireless systems now exist in a number of forms. In applications where the physical context of an interaction between devices is important, infra-red is often preferred to radio. Infra-red systems benefit from the similar line-of-sight properties of infra-red beams to those of visible light, which limit the effective range of an infra-red remote control to those devices the user can actually see. This intuitive approximation of the locality of devices is used extensively in household remote control technology, but is also a popular method of exchanging data between co-located portable computers. The line-of-sight property enables users to unambiguously identify which devices are to interact without requiring them to be physically connected.

Radio technologies, by contrast, are typically adopted in areas where continuous access to a service is required regardless of the user's location. Products such as pagers, mobile phones and wireless LANs rely on powerful base-station servers to allow mobile client devices to operate at lower power levels. Radio is also used to some extent for context-aware communications between mobile devices. The Bluetooth standard specifies protocols and interoperability requirements for supporting local-area streaming and data transfer between devices. Several similar projects, such as PEN, also use short-range, low-power radio to provide peer-to-peer communications between devices.

1.3 Summary

Because the improvements in wireless communication technology have been motivated mostly by the needs of commercial applications, communications protocols between handheld and other devices are usually proprietary and application-specific. Although standards do exist for particular application areas, there exists no real candidate for the role of lowest-common-denominator network protocol.

The following chapter discusses the limitations of existing technologies and the consequent motivation for the development of a general-purpose architecture for distributed systems that is both standard and efficient.

1.4 Outline

The remaining chapters of this dissertation have the following layout:

- Chapter 2 discusses the motivation behind the development of the Eucalyptus architecture.

- Chapter 3 gives an overview of the Eucalyptus architecture, proposed to address the issues raised in Chapter 2.

- Chapter 4 describes a number of related technologies in the areas of ubiquitous, wireless and home computing.

- Four main component technologies have been developed as part of the Eucalyptus project. Chapter 5 describes the first of these, a messaging library providing low power communications primitives. Support services for the messaging system are also discussed, in particular two low power discovery protocols.

- Chapter 6 describes a wide-area location service for mobile objects. The location service forms the basis for a wide-area routing system, designed to provide equivalent functionality to the messaging library described in Chapter 5. Details of how local-area primitives are implemented for a wide-area network are used to illustrate the system's design.

- The features of an object request and event broker, built upon the work described in Chapters 5 and 6, are discussed in Chapter 7. The broker aims to provide both efficiency and interoperability, by providing several interactions methods for remote objects and using the most appropriate underlying network primitives to implement each.

- Chapter 8 describes the architectural components of the Eucalyptus system, which build upon the work described in Chapters 5 to 7 to complete the Eucalyptus architecture.
- Chapter 9 illustrates use of the Eucalyptus architecture with a number of example applications which have been implemented for it.
- Finally, Chapter 10 summarises the contribution of this dissertation, and proposes possible directions for further work in the area.

Chapter 2

Motivation

This dissertation proposes a distributed programming model, dubbed "Eucalyptus", designed to be capable of operating over extremely low power, high latency embedded wireless interconnects. The protocols used are intended to be simple enough to be mimicked by even very tiny devices with no processing capability.

The motivation for developing such an architecture stems from the technological and commercial trends described in the previous chapter. Advances in semiconductor and communications technologies provide scope for new application areas. At the same time, however, the proliferation of complex commodity electronics poses problems for users which existing technologies do not solve.

2.1 Networking Simple Devices

The technology trends described in the introduction lead to a gradual increase in the number and complexity of computer systems with which humans interact on a daily basis. As the number of independent devices operated by a user increases, the degree of replication of functionality and data increases correspondingly. The management overheads involved in keeping replicated data synchronised between disparate devices can have a severe negative impact on their combined utility value to the user.

Synchronisation tasks are typically *passive*. No direct user intervention is usually required to trigger synchronisation tasks to take place. Indeed, in many situations, a requirement for direct action from the user to initiate synchronisation would not only be unnecessary but tiresome. Passive tasks have distinct requirements, particularly as regards timeliness of completion, to their *interactive* counterparts. A command from a remote controller to start playing a video tape should be processed within a second of receipt, or less, while backup of data from one device to another can often be performed over a period of several minutes without affecting the user.

The following sections summarise some of the existing techniques for synchronising data between networked devices. These range from distribution of data by central authorities to clients, to ad-hoc exchange of information by co-located peers.

2.1.1 Time

Digital time-keeping technology provides an example of duplicated functionality commonly found in home and office environments. A typical user will own alarm clocks, watches, computers, palm-top devices, hi-fi equipment, electric cooking equipment, video recorders and television systems, all of which have time-keeping facilities. Many countries maintain "summer time", "daylight savings time" or similar notions, causing all clocks to require adjusting twice per year. In the absence of seasonal changes to clocks a degree of maintenance is nonetheless required, to cope with skew arising from small amounts of clock "drift" which can accumulate to become significant over prolonged periods.

Manual synchronisation suffers both from inaccuracy and from ambiguity as to which device provides the "authoritative" time. One solution is provided by the use of broadcast clock signals keyed to the atomic clocks responsible for keeping UTC. Problems arise, however, when devices which depend upon broadcast UTC are moved to regions where the signal is unavailable or is broadcast on a different frequency. Ideally a user's devices should cooperate to remain synchronised with each other, and with an authoritative time-keeper where available, via a shared network. To do this requires both the capability to exchange data between devices and suitable logic to decide when synchronisation should occur.

2.1.2 Personal Information Management

Similar problems arise with personal electronic devices supporting "PDA"-like facilities. Inexpensive products are now available which will manage telephone numbers, addresses, email addresses and other personal information. Combinations of different sets of database features will be provided by devices, often with some degree of overlap. This overlap presents replication and synchronisation issues to the user. Telephone contact information may exist on a user's PC, on a handheld PDA, in a more compact dedicated device and in their mobile phone, for example.

While early devices providing personal information management features did not address the issue of synchronisation, subsequent products have supported increasingly intelligent synchronisation mechanisms. The approach most commonly adopted for synchronisation between mobile devices is to synchronise each device separately with the equivalent application running on a host PC. This can be achieved while the device is "docked" to recharge, with data transfer via serial or USB, or via an infrared connection, for example. Because the PC remains a central "hub" for all synchronisation operations, problems of inconsistency can often be avoided.

Several problems exist with this approach. Firstly, the mobile devices involved cannot be synchronised in the absence of a host PC. Secondly, each device will usually have its own distinct transfer protocol and custom data formats tailored to the applications it supports. This requires that synchronisation software be installed on the PC for every device it may be required to synchronise with.

In an effort to simplify synchronisation of personal information between these types of device, the SyncML [69] consortium has been formed. This group has proposed standard protocols and data representation schemes designed for synchronising personal information databases. Although SyncML supports a flexible set of synchronisation methods, the system's reliance on a centralised client-server model limits its usefulness in mobile environments.

2.1.3 General Synchronisation

The existence of the SyncML protocol and similar efforts in other application domains, such as personal messaging services, illustrate the importance of automated management of replicated state. The common requirement to perform all synchronisation tasks via a central authority simplifies issues of consistency and concurrency but also makes it impossible for two mobile devices to exchange information automatically in an ad hoc manner.

By supporting some form of peer-to-peer networking, these issues may be tackled more effectively. The need for manual intervention when transferring information between embedded devices can become sufficiently onerous to significantly reduce the usefulness of collections of devices. With suitable cooperative facilities, going beyond simple transfer of files and database entries, devices can impose less of a cognitive burden on users and automatically perform many of the menial data-juggling tasks required.

The lack of generalised support for management of distributed state between embedded devices stems from two primary obstacles and their interaction:

- Firstly, the underlying technology changes too quickly to allow protocols to mature. Mobile consumer electronics is a relatively new area of computing and is the focus of a great deal of research and industrial development. Development of manufacturing and design processes intended to reduce costs and increase yield can lead to sudden and radical shifts between the use of different technologies. In order to keep the time-to-market of a product as short as possible, it is often easier to put together a proprietary protocol which can support all the features and limitations of the device properly, than to take an existing standard and rework the product until it can be fully supported.

- Secondly, the adoption of a standard relies on the agreement of a significant proportion of the relevant manufacturers to support it. Supporting diverse devices in a single standard is particularly troublesome, since it is likely that those designing mobile phones, PDAs, PCs and Hi-Fi equipment will all have different priorities and interests in the new standard. Ironically, this is the situation where standardisation is most required. Achieving a consensus on a standard can be a slow process and the resulting specification may be too complex or even impossible to implement.

2.2 Wireless Technologies

As has been remarked in the previous section, a well-accepted approach to replicated data between mobile devices is to have them synchronise with a central, controlling entity such as a PC. This is often performed via a one-to-one serial connection.

Using a serial connection, each port on the PC can be connected to a single handheld device. The number of potential device connections which can be made is thus limited by the number of available ports on the PC. The recent development of USB, Firewire and other one-to-many, "plug and play" connection technologies allows this limit to be extended through the provision of "hubs" which allow a single port on the PC to be multiplexed between several devices. Although this makes it possible to add more devices by cascading hubs, the solution quickly becomes clumsy to operate.

The IrDA specification is used in many palm- and lap-top computers to allow communication and file transfer between devices using infra-red. Version 1.0 of the protocol supported one-to-one serial connection between two devices, behaving effectively as a wireless serial port. Version 1.1 provides for communication with multiple devices in the access control layer (IrLAP), by nominating one device as the Primary device in interactions and all other devices as Secondary. All data must be sent either to or from the Primary device; no direct communication between Secondary devices is supported.

Although using wireless technology of this kind removes the physical limit on the number of simultaneous connections that can be made, it is often the case that the limit remains, imposed by the software and protocols used. IrDA is most commonly used to provide a serial port interface within the operating system. This has the advantage that legacy applications which operate only via a serial interface can be made to use IrDA but makes it impossible for applications to use the more advanced features of IrDA, including support for multiple connections.

Because the IrDA standard is designed for peer-to-peer communication between co-located devices, the presence of support infrastructure is no longer required in order to synchronise information between machines. Removing this requirement allows the user greater freedom in how they use devices and helps match more closely the communications properties one might intuitively expect of nearby devices.

The decreasing cost and increasing effectiveness of radio-based wireless systems leads to interest in their use in place of systems such as IrDA. The use of infra-red light as a transmission medium limits communication to occurring only between devices within line-of-sight of each other. While this is a desirable feature for user-initiated actions such as file transfer, in which line-of-sight provides a useful proximity heuristic, it severely limits the capabilities of more passive applications.

Using radio communications, wireless devices can perform tasks without the need for human intervention. Since radio signals can penetrate clothing, furniture and even buildings, they are an ideal medium over which to communicate to perform passive tasks. A PDA can backup sensitive data and download the latest emails for a user while tucked away in a pocket or briefcase, for example.

2.3 Efficiency

Embedded systems are traditionally regarded as having distinct efficiency requirements to conventional "big-box" desktop or server computers. This distinction arises from the limited processing, storage and communications capabilities of embedded devices, which make many programming practices used on larger systems impractical.

Mobile and wireless embedded systems impose further limitations on software developers because of the need to operate without a mains power supply. Laptop PCs and rechargeable PDAs work around the problem to some extent by relying on the availability of mains power in most locations, allowing such devices to be recharged daily if required. Mobile phones benefit similarly from the convenience with which they may be recharged on a daily basis. This reduces how long they must be able to operate for on a single charge to only a few days. The ease with which many "wearable" computers may be recharged leads to claims that low power technologies are unnecessary and that software may safely assume power to be in plentiful supply. As will be discussed in the following section, this assumption fails to hold if the user operates more than a handful of rechargeable devices.

2.3.1 Composite Devices

For a collection of only a few devices, daily recharging is an adequate solution. For even as few as four or five devices, however, the requirement to provide sufficient power outlets and to connect each device with the correct power supply can quickly become tiresome.

The simplest solution is to reduce the number of devices the user carries. This can be achieved by providing multiple functions in each device. The user's mobile phone, for example, might also function as an MP3 player or radio or both. Although this reduces the number of devices that must be recharged, it prevents the user from upgrading or replacing individual functions as required. As increasing numbers of functions are combined into a single unit, the likelihood of the user finding a device with all the features they desire from each of its functions decreases significantly.

Composite devices also suffer from the need to support multiple, possibly very different sets of functions into a single coherent user interface. The user interfaces for each function of the device must be merged into a single unit. The resulting interface is often biased toward one of the supported functions and is less intuitive to use for the others. For one or more of the device's functions, the interface may simply not match the user's preferred interface style. Composite devices will often therefore be of less use to their owner than individual devices providing equivalent features.

2.3.2 Power Management

A more satisfactory approach to the problem of frequent "feeding" of multiple electronic devices is more efficient power management. The processor, memory and peripheral components which a typical device will contain often support some level of power management. This support ranges from control over whether or not the unit is powered at all, through to fine-grained control over the level of power consumption (e.g. lowering a processor's power requirements by reducing its clock frequency). For the vast majority of their time, handheld devices are doing little more than storing data and all components other than memory can be completely powered down. During periods of prolonged inactivity the contents of volatile memory can be written to a disk or non-volatile memory, allowing the device to switch off completely. These power saving techniques are often applied at a macroscopic level in devices such as laptops and PDAs, which power down disks after periods of inactivity and provide low-power "standby" modes.

Devices performing tasks passively and without user intervention can benefit more from power saving than handheld devices, if the task has no intrinsic time-critical elements. Tasks such as backing up data, and enabling a burglar alarm system at the end of the day have relatively relaxed timing requirements and the devices performing them can therefore be made to power down for longer periods. This is particularly useful for wireless sensors and other small devices which might be located in hard-to-reach places and would therefore be hard to recharge.

These problems motivate consideration of power issues at all levels of the system, in particular in the messaging protocols used between devices. Using suitable protocols, devices may remain asleep most of the time to save power, while still offering full functionality.

2.4 Interoperability

Many users maintain address book information on their PDA and their PC. Subsets of their unified address book may exist on each of their mobile phone and email systems. Often such information may be replicated in different formats between applications running on the same system. When each application uses a different proprietary format in which to store the same information, a separate tool is required to perform synchronisation between applications. SyncML and other similar standardisation efforts in this application area show the need for well-defined interfaces through which otherwise proprietary systems can share information.

2.4.1 Custom Protocols

The same compatibility issues arise with use of more general distributed systems. Established principles of component re-use cannot be practically applied to systems employing closed or application-specific protocols. Even in systems where standard network protocols have been employed, higher-level parts of the communications stack such as application protocols and data formats are often incompatible despite sharing similar semantics.

The use of custom protocols is motivated by a number of influences. Custom protocols can be developed quickly and tailored to be as efficient as possible. Although the initial costs of developing custom protocols can usually be justified, a great deal of development effort can result from later attempts to add functionality or to interface between “legacy” systems.

Further motivation for the use of proprietary protocols arises from the need to support custom features in otherwise standard devices. Most consumer entertainment products provide one or more of a small set of basic functions, plus a number of supplementary features specific to the manufacturer or to the model of device. As a result, remote controllers for different devices are often incompatible. Incompatibility between different manufacturers’ devices is desirable for manufacturers because it locks users in to a particular range of products. Conversely, incompatible protocols between different products from the same manufacturer can be used to force consumers to upgrade not just one device but several, by preventing different versions of device from interacting.

2.4.2 Standard Protocols

Standards such as IrDA and the IEEE 802.11 family (often referred to as “WaveLAN”) provide support for conventional wired-network communications primitives over wireless media. The IrDA standard is most commonly used to provide a wireless replacement for serial cables, while the WaveLAN system supports standard Ethernet packets being transmitted over a radio channel. Both systems benefit from their compatibility with the wide ranges of applications and protocols designed to use their wired counterparts.

The Eucalyptus architecture, proposed in this dissertation, takes the view that operating standard wired-network protocols over wireless networks is undesirable because of the differences in characteristics of the underlying media. Wired network protocols can assume relatively high bandwidths and low latencies. By contrast, current wireless technologies are often either low-bandwidth or high-latency or both. Wireless networks such as PEN, which power devices down for prolonged periods and thus can only accept incoming data at pre-defined intervals, can increase the effective latency by several orders of magnitude.

More fundamentally there is little requirement of wired protocols to cope gracefully with sudden disconnection. By contrast disconnected operation is an intrinsic feature of wireless communications. TCP connections, for example, cope extremely poorly with sudden disconnection of the underlying media.

Although it is desirable to tailor application protocols to the underlying network, the vast majority of application logic need not be concerned with such detail. Instead, applications should use a standard, cross-platform abstraction wherever possible, which will map high-level operations to suitable network primitives. Characteristics common to classes of network, such as high latency in the case of low power networks, should be abstracted wherever possible to allow applications to remain platform-neutral. Specific devices may still be "tuned" by low level configuration of network parameters, provided that such tuning does not affect the application's semantics.

The CORBA specification provides an object model and standard protocol and API elements to support it. Standard APIs allow implementations to vary widely internally, for example by using special-purpose protocols between compatible endpoints. Standard protocols allow objects providing a specific interface to be accessed seamlessly regardless of the CORBA implementation or programming language they use. This enables elements of distributed systems to be more easily extended, replaced or re-used.

The architecture proposed in this dissertation follows the view that a standard distributed object model as used by CORBA provides a natural mapping between physical resources and software components and that applications can benefit from the features the model provides.

Although capable of operating over a variety of underlying networks, Eucalyptus has been used primarily with the PEN network, because PEN has characteristics representative of low power wireless networks in general. The PEN low power messaging protocol has been extended to support multiple communications primitives in a unified fashion. A custom ORB modelled on the CORBA standard has been built which provides both object-oriented RPC and event notification and implements these interfaces using the most suitable PEN primitives. As will be shown later, although the CORBA object model provides a number of features desirable in supporting Eucalyptus, CORBA itself is unsuitable for use with PEN and similar networks.

2.5 Summary

This chapter has discussed the effects of proliferation of microprocessor controlled consumer electronics caused by recent advances in that technology. The extra complexity in managing such devices and in maintaining consistency amongst the data they contain motivates the development of a ubiquitous communications medium to allow them to perform these tasks themselves. Similar developments in wireless communications to those found in microprocessor technology make this a feasible goal to achieve.

The following chapter provides an overview of the Eucalyptus system, proposed to address some of the issues of ubiquitous communication between devices ranging from embedded sensors to full computer systems.

Chapter 3

Overview

This chapter gives an overview of the Eucalyptus architecture, proposed to address the issues raised in Chapter 2.

3.1 Physical Model

The Eucalyptus architecture is intended to allow home, office and personal appliances to interoperate seamlessly over a variety of network transports. Such appliances range from high-end computer systems, through desktop PCs, hi-fi equipment and household electronics, to simple embedded sensors. A thesis of this dissertation is that in order to support devices with such a wide spectrum of capabilities the architecture must be simple and compact and make as few assumptions about the available resources of an appliance as possible.

A Eucalyptus system consists at the computational level of a set of objects known as *components*, distributed over an ad hoc network of *appliances*. A Eucalyptus appliance is any piece of hardware capable of hosting, or of *appearing* to host Eucalyptus components. Appliances may be stand-alone, hosting an entire enclosed Eucalyptus system within a single unit, with no networking to other appliances, although this will not usually be the case. Typical Eucalyptus appliances support at least one network interconnect through which components residing on other appliances may be contacted.

Typical examples of simple appliances are sensors for environmental conditions such as light level or temperature. Such sensors can be placed in convenient locations around a building in order to monitor the environmental conditions in an office. Each sensor will operate for several years on a tiny cell of the type used in many digital watches, or using ambient power sources such as light, heat or movement. The requirement to be able to place sensors precisely where readings are desired motivates the use of a wireless system in this example, while the need to operate for several years without human intervention, or to operate indefinitely using an ambient power source, emphasises the low power requirement. Figure 3.1 shows a system consisting of several sensors and a handheld monitor device.

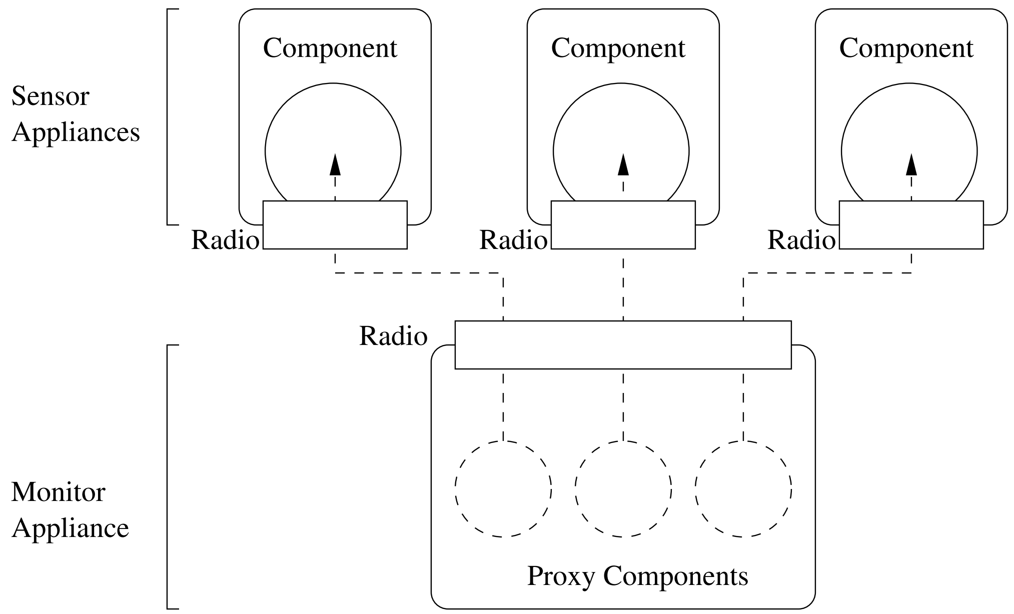


Figure 3.1: A simple sensor system using Eucalyptus

Sensors and other minimal Eucalyptus appliances will typically comprise a power source, a wireless network interface and a PIC processor. Such devices will transmit broadcast datagrams containing their current sensor reading at regular intervals, embedded in a hard-coded datagram mimicking the Eucalyptus protocols.

In general Eucalyptus systems will involve more complex appliances, supporting multiple software components and controlling each other to achieve tasks on behalf of users, rather than simply relaying sensor data. Such appliances will typically employ a comparatively powerful microprocessor and will use more complete implementations of the Eucalyptus protocols.

The next sections describe in more detail the object and computational models adopted by the system to support general interactions.

3.2 Object Model

Eucalyptus follows an object-oriented computational model, sharing many properties with that defined by the OMG's CORBA [47] specification. The object model is implemented by the Koala Object Request Broker (ORB) [77]. Each object implements a single *interface*, comprising a number of method and attribute definitions, in contrast to systems such as HAVi [24] in which objects are little more than communication endpoints. Only methods and attributes defined in an object's interface are accessible to clients. New interfaces can be *derived* from one or more existing interfaces and extended as required. Objects implementing derived interfaces can be used in place of objects implementing an interface from which they are derived.

Objects are split into a *servant* which implements the specified methods and attributes, and a *stub* through which they are accessed by clients. Based upon a supplied interface definition, an IDL compiler will generate a suitable stub for use by clients, and a *skeleton* upon which servant implementations may be based. While the servant may contain extra methods and attributes used internally by the object's implementation, the stub provides only those items defined by the object's interface. When the servant resides on a different appliance to the client, the stub will perform the required marshalling of parameters and cause a Remote Procedure Call (RPC) [66] to be dispatched over the network. Even when servant and client are co-located, all access to the object is performed via the stub, which will simply call the relevant method directly. Figure 3.2 shows two objects being accessed, one of which resides on the same appliance as the client, the other of which is implemented remotely.

In addition to supporting RPC interactions in a similar fashion to CORBA, the Koala ORB also allows objects to be *watched* by clients. Using the watch mechanism, clients may be notified of changes in the state of particular objects' attributes via a distributed event transport. The ORB implements event notifica-

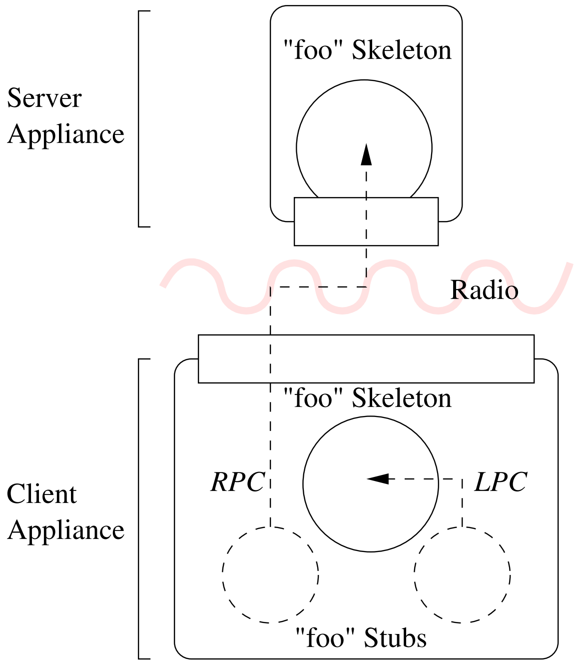


Figure 3.2: Accessing servants locally (LPC) and remotely (RPC)

tions using the most appropriate messaging mechanisms made available by the underlying network. Not only does native support for monitoring of Koala objects allow the ORB to optimise the implementation, but it also reduces the complexity of many applications, particularly context-aware ones.

The use of abstract interfaces also makes it possible to *proxy* between the Koala protocol and other semantically similar protocols such as CORBA [47] or RPC, as illustrated in Figure 3.3, in which a simple wireless sensor is queried by a client residing in a wired network via a Koala-to-CORBA proxy. Abstract interfaces also provide a contract between client and servant. Instead of coding applications to be compatible with each other, they may be coded for compatibility with the specified interface, improving the chances of compatibility with future implementations.

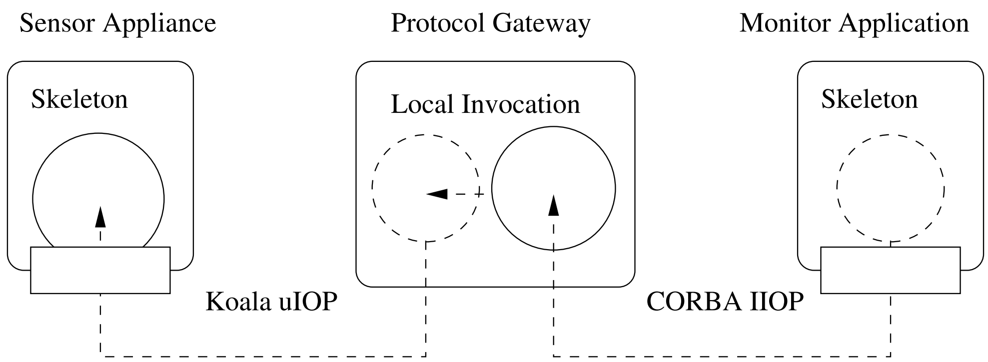


Figure 3.3: Accessing wireless sensors from the wired network

3.3 Computational Model

The Koala ORB follows an object-oriented abstraction of computational resources distributed amongst host appliances. By providing diverse communications primitives within this abstraction, the ORB can free applications from the need to deal directly with network-specific interfaces, without reducing their efficiency. The ORB does not prescribe particular interfaces or impose structure on the components of a distributed system, nor does it make assumptions about the platforms upon which it will run, or about the correspondence between computational objects and physical components. Such issues are instead dealt with by the Eucalyptus architecture, of which the Koala ORB is a core part.

Although it abstracts away from the details of underlying networks, the Koala ORB does not attempt to conceal aspects of networked operation from clients completely. In particular, the ORB does not attempt to deal with communications failure modes internally, instead providing high-level notification of such problems to clients so that they can take appropriate measures. This is achieved by raising language exceptions from failed method invocations, or by distributing failure events to interested clients, for example.

Each Eucalyptus appliance hosts one or more objects corresponding to their functional components. The term “host” is used even for appliances which lack sufficient resources to run a full Eucalyptus system. The tiniest sensor is considered to be a host, despite the fact that in practice it simply fills in blank slots in pre-formed datagrams and transmits the completed packets.

Because the object model employed by Eucalyptus makes no distinction between objects implemented locally and those implemented on other appliances, the two cases appear identical to higher layers of the system. This simplifies the system from the programmer's point of view, since code need not be rewritten for multiple special cases. At the computational level, Eucalyptus systems can consist entirely of collections of inter-related objects with no correlation between objects and their hosts.

In practice, it is useful to be able to refer to the physical layout of the system when performing certain types of task. Information on the physical layout of the system is often required when a user wishes to select a suitable object to interact with. For example, a mobile user attempting to listen to an audio stream probably wants to listen through a speaker on the appliance they are carrying and not through the speakers of a passing vehicle.

The problem is that by abstracting away from the physical layer of the system, we have created a flat, uniform environment with little or no context available upon which to base selection decisions. Relationships between distributed objects are often expressed through the use of naming services, traders, and other lookup techniques. The layout of lookup services and the types of query they support allow much of the lost context information to be recovered. At the same time, the abstraction of objects' location allows them to be used, once selected, regardless of their location.

3.3.1 Components

To support structuring of Eucalyptus systems, a number of standard object interfaces are defined. The most basic of these is the *component* interface, which is used to identify functional components of a Eucalyptus appliance. The component interface is primarily used as an identifying feature of objects which can be usefully used in their own right, distinguishing them from objects generated as part of some particular task.

Each speaker connected to an amplifier, for example, might host its own component object. Internally, the amplifier would then be manipulated via a set of object interfaces to produce the effects required by the speaker components. Unlike the high-level interfaces used to control the volume, bass and treble of the speakers, the user should probably not be made aware of these low-level interfaces.

3.3.2 Devices

One particular class of components dealt with specially by Eucalyptus are those requiring arbitration between conflicting operations from different clients. These components, termed *devices*, will usually be configurable in some way and must therefore be capable of dealing with conflicting configuration requests.

Eucalyptus aims to be application-neutral wherever possible. As such, it does not provide support for the arbitration of access to devices. Instead, it is assumed that devices themselves, or third-parties aware of their properties, have a better idea of how to deal with conflicts than the general-purpose architecture. Eucalyptus therefore defines an basic reserve/release interface for devices. It is assumed that the vast majority of devices will implement mutual exclusion between clients, and that more complex arbitrations schemes, such as mixing of audio channels, for example, will be performed by third-party components. Eucalyptus' access control mechanisms are described in greater detail in Chapter 8.

3.3.3 Traders

The Koala ORB does not specify how objects hosted by remote appliances should be discovered prior to use. Eucalyptus provides discovery via a simple *trader* mechanism. Traders are containers for other Eucalyptus objects, including other traders. They may be queried for lists of objects with particular characteristics, and may additionally be subdivided into a hierarchical tree of sub-traders. Authorised clients may add and remove objects from a trader.

This functionality is distinct from that provided by the Predator location service, described in Chapter 6. While the trader mechanism allows clients to discover nearby Eucalyptus objects in an ad hoc manner, the Predator location service is instead used to track the location of appliances of which clients are already aware.

The monitoring appliance back in Figure 3.1 must be able to keep track of which sensors are locally available, for example. The use of a short range, low power network allows an appliance's contactability to be used as a locality metric. If the appliance cannot be contacted then it is deemed non-local. This in turn can be used as a convenient context heuristic.

This locality information is made available to the sensor monitor application via a *local import trader*. The import trader is a normal Eucalyptus trader populated by a network discovery process. The process links itself to the low level radio interface and monitors the table of local appliances. When a new appliance arrives, the components it hosts are added to the import trader. The monitor application is notified every time the contents of the import trader change, allowing it to discover any suitable sensors which have arrived, as illustrated in Figure 3.4.

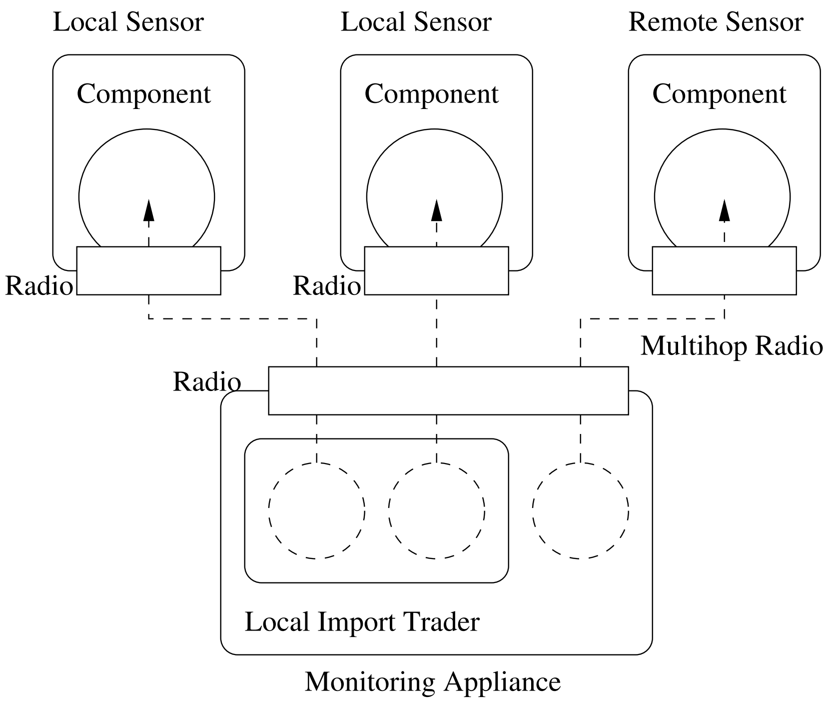


Figure 3.4: Monitor appliance performing discovery using an import trader

3.4 Programming Model

As was previously described, the Eucalyptus architecture uses an object model supporting both RPC and event mechanisms. Although these are sufficient for conventional programming by trained users, they are unsuitable for the majority of non-technical users to learn.

In order to be used by non-technical users, Eucalyptus must provide high-level abstractions through which the system's behaviour may be expressed and reconfigured. These abstractions must be reasonably intuitive to use but at the same time capable of expressing common interaction types. The problem then is that of defining the term "intuitive to use".

As an approximation, it is assumed that a useful subset of intuitive abstractions are those which can be related to physical interactions. The most obvious example of this is the provision of standard *connection* interfaces, allowing uni-directional data flows to be expressed, between components of a Eucalyptus system. The connection abstraction is deliberately designed to have semantics that closely match those of physical connections such as those made between hi-fi and speakers or switch and lamp. These types of user programming model are desirable for two main reasons. Firstly, they are typically familiar to users and the learning curve is therefore reduced. Secondly, users will feel more comfortable "playing" with them.

We can argue in favour of such an assumption based on the target application domains of the Eucalyptus architecture. The architecture aims both to make tetherless manual control of appliances easier, and to allow automated control of appliances. The former application domain in particular involves replacing physical control interactions with wireless ones. Using Eucalyptus a user will tell an audio speaker to connect itself to a particular audio source, rather than physically plugging the two devices together, for example. Similarly, wireless sensors can be monitored using a connection-oriented abstraction, since wired sensors would, by definition, need to be connected to the monitoring device.

3.5 Summary

This chapter has described the main features of the Eucalyptus architecture, in particular the *component*, *device* and *trader* mechanisms. The assumptions made about the layout of the appliances upon which Eucalyptus implementations will reside have also been discussed. The following chapters describe technologies developed to support the Eucalyptus architecture, including a low power messaging library and wide-area routing algorithms. Chapters 7 and 8 describe the Koala ORB and Eucalyptus architecture in more detail.

Chapter 4

Background

The Eucalyptus project spans a number of areas of academic and commercial research and aims to provide a lowest-common-denominator communications infrastructure for diverse appliances. As a result, numerous bodies of research and commercial development exist which are relevant to particular parts of the project. Few projects exist which overlap extensively with parts of Eucalyptus. This chapter therefore concentrates on related technologies with unusual features or target application domains in common with Eucalyptus.

4.1 Ubiquitous Computing

The PARCTAB experiment [72] is widely cited as the founding work in the field of Ubiquitous Computing. The project aimed to provide users with the ability to access computing resources in an untethered, mobile fashion. By using infrared as the physical communications channel, the PARCTAB system also provided location information to applications, allowing them to be tailored to the user's environment.

PARCTABs were hand-held devices with three buttons, a touch-sensitive LCD display and an infra-red transceiver. A Tab will beacon at a fixed frequency when active in order that its owner can be located. These beacons and other infrared traffic are monitored by base-station transceivers in each room of a building and relayed to Agents for each Tab. Agents run on a backbone network of PCs, and provide an interface between the Tab hardware and services and applications running on the network.

The project explored a variety of aspects of mobile, ubiquitous computing, among which were design of user interfaces for small devices, design of location-aware applications and uses of this client systems to handle mobile users. Eucalyptus differs significantly from the PARCTAB project, in that Eucalyptus devices are capable of interacting without supporting infrastructure, in a low power manner. By contrast, PARCTABs were unable to operate outside their home environments, and had extremely limited battery lives.

4.2 Sentient Computing

Hopper [28] introduces the notion of *sentient computing*. Sentient computing systems monitor real world environments using sensors, and take actions accordingly. The primary focus of work by AT&T Laboratories Cambridge on sentient computing has been on location information, for which a variety of active sensor systems have been developed, including Active Badges [71] and Active Bats [73].

The SPIRIT [22] system has been used to support context-aware applications such as visualisation, routing of phone calls to users, and "teleporting" of roaming computer sessions [64].

López de Ipiña [36] describes a vision based passive tag location system, TRIP, and a sentient computing architecture with similar applications to SPIRIT. Spreitzer et al [65] describe an infra-red sensor system in which tags are placed in the environment and are monitored by mobile agents themselves.

The Eucalyptus architecture can provide similar location-like context metrics to the tag systems described above. Whereas the SPIRIT system relies inherently on computing infrastructure to process incoming data from wired sensors, Eucalyptus appliances can obtain sensor and context information in an ad hoc fashion from any suitable devices nearby. By bridging between such systems, SPIRIT can benefit from wireless sensor data, and Eucalyptus clients can benefit from the availability of pre-processed context information.

4.3 Home Area Networks

The Home Area Networks [21] project at Cambridge University Computer Laboratory considers issues ranging from low level interaction between devices to Human Computer Interaction (HCI) in the home. Much of the project assumes the presence of an ATM [2] network in the home, through which all electronic appliances are connected. The Warren Control Protocol [12] allows even extremely primitive home appliances to participate in such a network.

The AutoHAN project [18] attempts to provide automation support for HAN appliances by merging events received from sensors and input devices around the home using a composite event language. Both basic and composite events may then be used to trigger actions.

A significant goal of the HAN project is to make intelligent and distributed operation simple enough for mass-produced, cheap commodity electronics to support. This is similar to the aims of the Eucalyptus project. By limiting the kinds of events used by a HAN system, Eucalyptus and HAN could feasibly be bridged. Both systems are desirable because HAN is ideally suited to supporting high-bandwidth multimedia applications, while Eucalyptus provides ad-hoc, tetherless control of appliances. Combining the two would allow the same PDA, mobile phone or other user interface device to be used seamlessly to control appliances both inside and outside the home.

4.4 HAVi

HAVi (Home Audio and Video) [24] defines interfaces and semantics for audio, video and control devices attached to IEEE1394 (Firewire) [29] networks. The specification assumes a single shared Firewire network to which all devices in a home environment will be connected. Devices may then communicate as required in a peer-to-peer fashion, without the need for any central authority to perform access control or manage their resources. Conversely, a network master device may be used if required, to apply structured rules to the configurations of connected devices.

The architecture allows devices with varying degrees of awareness of HAVi protocols to interoperate, and supports dynamic uploading of code into controllers to allow them to manipulate newly installed equipment.

4.5 X10

X10 [19] is a standard for programmable control of mains-powered home appliances through existing mains wiring. Control signals are transmitted between devices over the electrical wiring of the user's home, using a modulated carrier signal overlaid on the normal AC power supply. This makes X10 particularly suitable for a home environment.

The control commands available are limited and there is no intrinsic verification of compatibility between a device and the control commands it is being sent. The address space for devices is limited to 256 entries which must be manually administered, which in some cases can cause clashes between devices in homes sharing common electrical wiring, which have been allocated the same address.

4.6 Philips RF-Lite

Philips RF-Lite [61], recently adopted as the basis for the IEEE 802.15.4 standard, is a home-area network with similar target applications to the X10 system described in Section 4.5. RF-Lite provides low bit-rate bi-directional communications between up to 255 devices, of which one device must be the network master, and another the user's remote controller. RF-Lite provides basic network connectivity, without specifying particular architectures or application-level protocols to use, a characteristic shared with the earlier HomeRF [27] system.

RF-Lite differs from the PEN network, upon which the Eucalyptus project has been based, by using a master/slave network topology similar to that found in systems such as Bluetooth [6] and IrDA [31].

4.7 Stick-e Notes and Contextual Information Services

The Stick-e Note project [58] presents a simple model for the association of information with particular contexts. The system is modelled to match the characteristics of small physical notes which may be placed around a user's environment. Stick-e Notes mimic this behaviour to allow electronic documents to be presented automatically to users, according to their present location. More advanced notes can perform actions for the user based upon their current location, such as downloading their latest email to their PDA whenever they enter their office. Because the project defines a technology-neutral model for context-aware computing, Stick-e Notes may be supported by location information from short range radio systems, Active Badges or network connectivity. Equally, notes might be stored by devices near where they are intended to be discovered, or may instead be carried by mobile users and indexed by location.

Pascoe [59] proposes the Contextual Information Service (CIS), an extension of the Stick-e Note model to support general context information. The CIS consists of a number of distinct layers. The raw sensor layer consists of sensors providing real-world context information in sensor-specific forms. A sensor abstraction layer then allows physical sensors to be translated or combined to produce context information in standard forms. Abstracted sensor information is plugged into *templates* describing the attributes of different kinds of real-world objects. A template representing a human might contain name, age, location, height and weight attributes. The location, height and weight would be obtained automatically from attached sensors for each subject, while their name and age would be entered manually, to produce human *entities*.

The lcron [25] system provides similar functionality to Stick-e Notes by extending the standard Unix "cron" service to cope with location information in addition to timed events.

The Eucalyptus system could be used to support Stick-e Notes or the Contextual Information Service by providing context information based on radio proximity, or by accessing nearby sensors to supplement information from in-built ones.

4.8 CoolTown

Hewlett Packard's CoolTown project [26] provides interoperability between a wide variety of devices, using exchange of Universal Resource Identifiers (URIs) as the primary interaction method. The system is designed around the notion of *web presence*. Each device, component or document has its own URI, through which it is manipulated. Like the Stick-e Notes system in Section 4.7, CoolTown is almost entirely technology neutral. URIs may be exchanged between devices using any compatible network technology, whether infra-red, RF or wired. Of particular relevance to Eucalyptus is the notion of CoolTown *beacons*. These are tiny embedded tags which transmit a pre-configured URI to nearby clients. Because beacons have a short range, URIs may be distributed in a context-aware fashion.

Web-oriented interoperability between devices works well for manual interactions because it allows devices to remain trivial, capable only of sending and receiving URIs, while most of the system's intelligence comes from the user. Specific applications may, however, attempt to parse documents retrieved from URIs in order to perform tasks. An art gallery may tag each display with a CoolTown beacon, which users' PDAs would collect as they moved around. Once at home, the collected URIs would be sent to an Internet TV which would download digital images of each piece and display them. Alternatively, the same URIs could be uploaded to printers, or exchanged with other PDAs.

Eucalyptus aims to support many of the simple control and configuration interactions supported by CoolTown. References to Eucalyptus objects may be exchanged in much the same way as CoolTown's URIs. Because Eucalyptus objects are statically typed and provide well-defined interfaces to clients, recipients of references must be aware of the corresponding interface details in order to use them. Although this is an issue for user interface and bridging systems, it does not limit the automated interactions available, since CoolTown devices must understand the content obtained from a URI in order to process it.

4.9 Summary

The projects presented here comprise a core set of ubiquitous computing technologies representing the state of the art in both the academic and commercial sectors. Many related projects exist with more esoteric or domain-specific features, some as concepts, others as prototypes. In particular, a plethora of proof-of-concept implementations of various ubiquitous computing applications exist, that explore specific aspects of the research space.

Chapter 5

Low Power Messaging

This chapter describes the R2-Layer [78], a library of low power protocol primitives designed to support a wide variety of application domains. The R2-Layer builds on earlier work on the PEN Rendezvous Layer (R-Layer). R2-Layer primitives are used by the Koala ORB to implement object-oriented communications in a power-efficient manner. While parts of the ORB's implementation must be tailored to make best use of the R2-Layer primitives, the majority of its functions are platform-independent, motivating a clear separation of functions. The R2-Layer's primitives may be used via the Koala ORB by portable applications, or directly by platform-specific code.

5.1 The Prototype Embedded Network

The Prototype Embedded Network (PEN) [5] is an embedded low power, radio frequency (RF) network. It is intended to provide the lowest level of discovery, rendezvous and communications for all electronic devices. Nodes can in principle be miniaturised enough to be embedded as tags on books, people, tools, key-rings, etc, as well as sensors and more conventional electronic equipment.

The more recent PicoRadio [63] project describes an RF network with similar properties and target applications to PEN. The SmartDust [33] project explores some similar wireless sensor applications, using free space optical communications, and with a view to miniaturising nodes down to the size of motes of dust.

Feature	Prototype	Production Device	Production Sensor
RAM	128K	32K	<1K
FLASH/ROM	256K	32K	<4K
Size	10x5x2cm	2x2x0.4cm	1x1x0.2cm
Power (peak/avg.)	300mW/1mW	4mW/6μW	1mW/4μW

Table 5.1: Characteristics of PEN prototypes and estimates for production units

The prototype units used as the basis for this work comprise cheap off-the-shelf processor, radio and memory components, an RS232 [14] interface for development, and other peripheral components. They operate in the 418MHz ISM band and provide 24Kbps peak bit-rate. Applications for PEN are supported by the EEK runtime, a modular system based around the μCOS [34] micro-kernel. Table 5.1 summarises the characteristics of the prototype units [32] and those that could be expected from production equivalents.

5.1.1 Media Access Control Layer

PEN uses a variant of the Singly-persistent Data Sensing for Multiple Access (1-DSMA) scheme to avoid simultaneous transmission of datagrams by multiple senders. This protocol not only further reduces the bandwidth available to PEN applications, but results in random stochastic transmission timings and success rates. The results presented in this chapter therefore assume systems of PEN nodes operating in lightly loaded radio conditions, in which the MAC layer's behaviour is most stable. By definition, all guarantees of discovery, rendezvous and datagram delivery described are in practice statistical guarantees, subject to the level of service provided by the MAC layer.

The MAC layer provides basic datagram transmission and reception facilities between nodes, using 32-bit source and destination addresses. A datagram's destination may either specify a unique device address, or a group address. Only unicast datagrams are acknowledged by recipients. Support for direct peer-to-peer communication distinguishes PEN from other short range wireless technologies such as Bluetooth [6] and IrDA [31], which employ *star* network technologies in which all communication between *slaves* occurs via a single *master*. Peer-to-peer communication removes the need for one device in a group to bear an unfair power or processing burden. It also allows PEN to act as an introductory network for more specialised, high power wireless systems such as Bluetooth or WaveLAN [74, 30].

5.1.2 The Rendezvous Layer

The primary focus of the PEN project has been low power data transfer protocols. Devices using these protocols spend most of their lifetimes in dormant "sleep" states, unable to communicate, and wake only infrequently to rendezvous with others. Similar schemes are used to great effect by pager systems such as POCSAG [9] and FLEX [15]. These systems achieve battery lifetimes of up to a year by synchronising low power clients with high-power transmitters at regular intervals.

PEN's low power protocols are designed for decentralised communication, allowing co-located peers to communicate without supporting infrastructure. Devices rendezvous in sender/receiver pairs and agree suitable transmission patterns according to their own particular requirements and observations of local radio traffic. Nodes which are far apart must use low power multi-hop routing schemes such as those described in [16]. In the presence of backbone infrastructure, wide-area routing schemes such as those described in Chapter 6 may be used.

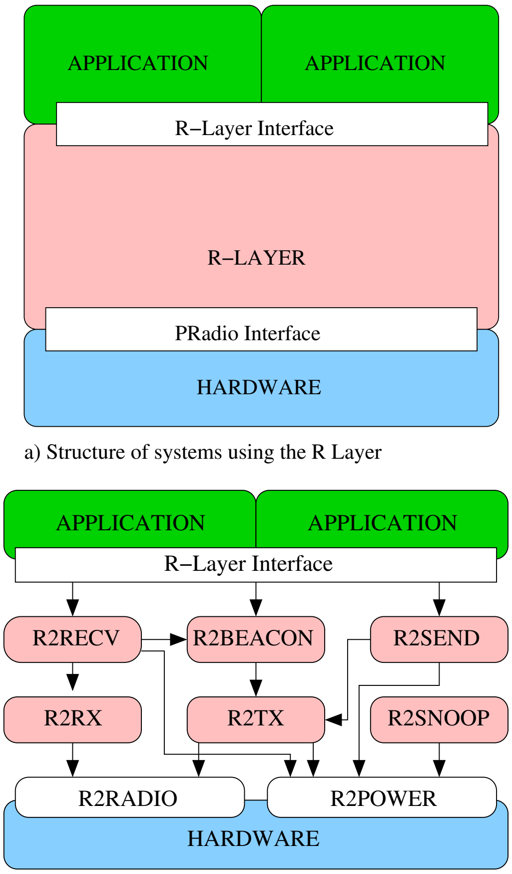
The Rendezvous Layer was implemented as a single component, in a similar manner to many TCP/IP stacks. Applications access the R-Layer via a single well-defined interface, used both to send and receive messages and to configure the network stack.

5.2 The R2 Layer

Three design principles, established through initial experimentation with the standard PEN Rendezvous Layer, affect the makeup of the R2-Layer;

- *Multiple messaging modes.* The R2-Layer aims to provide protocol primitives to support multiple messaging schemes, motivated by the variation in requirements of different applications.

- *Modular structure.* A system of modules implementing protocol primitives is used, allowing devices to contain only those modules they require.
- *Separation of control and data flow.* It is often the case that modules wishing to send or receive data are best implemented with power management performed by a separate, third-party module.



a) Structure of systems using the R Layer

b) R2 Layer implementation of the R Layer interface

Figure 5.1: Structure of the Rendezvous and R2 Layer libraries

Figure 5.1 shows the difference in structure of the standard R-Layer from an equivalent R2-Layer implementation of the R-Layer's interface.

5.2.1 Protocols

The standard PEN Rendezvous Layer implements one particular low power communications protocol, providing an equivalent level of service to PEN's raw radio interface. The R-Layer itself does not deal with issues of disconnected operation. This both simplifies the R-Layer's design and allows disconnection to be dealt with on a per-application basis.

The R-Layer protocol places the responsibility for arranging for rendezvous between two devices with the transmitting device. This uneven division of workload works particularly well for ad hoc delivery of unsolicited datagrams, since it requires no prior negotiation with the receiving device before transmission can occur. Many applications of short range wireless networks fit an unsolicited transmission model. Remote user interfaces controlling household or office appliances, for example, must transmit control datagrams whenever the user operates them but may remain in a deep sleep state at all other times.

By contrast, there are other applications of such networks which lend themselves more naturally to a solicited scheme, in which the sender transmits data in a well-defined fashion and requires recipients to perform the rendezvous steps to receive it. Appliances which monitor environmental conditions, such as simple light sensors for example, will be shown to benefit from receiver-managed rendezvous schemes.

The principal motivation behind the development of the R2-Layer was to provide multiple standard low power protocol primitives in a single library. The R2-Layer implements two rendezvous schemes, one transmitter-managed scheme based on that employed by the R-Layer, and a new receiver-managed scheme targeted at extremely simple sensor devices. The differences in power saving properties and in quality of service when interacting with the local environment and users serve to illustrate why support for many schemes in a single protocol stack is beneficial. At the same time, the transmitter-managed protocol optimisations developed for the R2-Layer could in principle be implemented for the R-Layer.

5.2.2 Transmitter-Managed Operation

The R2-Layer implements a transmitter-managed messaging protocol, allowing datagrams to be transmitted to suitable recipients without prior solicitation. Devices wishing to receive data must transmit broadcast datagrams at regular *beacon intervals*, in order to advertise their availability. After transmitting a beacon, the receiver listens awake for a short period, termed the beacon's *listen window*, before returning to sleep. The R2-Layer allows multiple listen windows to be advertised by each beacon, evenly spaced through the subsequent beacon interval.

Devices wishing to transmit datagrams must power their radio and listen for at least one beacon interval, for a beacon from the target node. If the recipient is active and within range then the sender will hear its beacon and may transmit data in the subsequent listen window. Figure 5.2 illustrates this transmitter-managed protocol.

Although the beacon mechanism works well for low power devices, it introduces unnecessary latency when communicating with powered devices. In this case, the sender must still wait for a beacon from the target node, even though the powered node is continuously listening and can therefore accept data at any time. Like the R-Layer, the R2-Layer employs a *plea/offer* mechanism to optimise this special case. Whenever a datagram is to be sent, the sender first broadcasts a plea for the target node. If the target hears the plea then it responds with an offer to indicate that it will remain powered for a short while to receive data.

5.2.3 Receiver-Managed Operation

The transmitter-managed rendezvous and messaging scheme is ideal for ad-hoc systems, since it avoids the need for a negotiation phase prior to communication. Because contact is initiated by the sender on a per-datagram basis, the protocol works well for RPC and similar protocols in which datagrams are typically delivered between nodes in request-response pairs. A node servicing requests need only have sufficient resources to process a single request and to transmit the result.

Transmitter-managed rendezvous poses efficiency issues for simple nodes providing a service to many clients. This is typically the case for embedded sensor nodes. The node must monitor the attached sensor and transmit updates to inter-

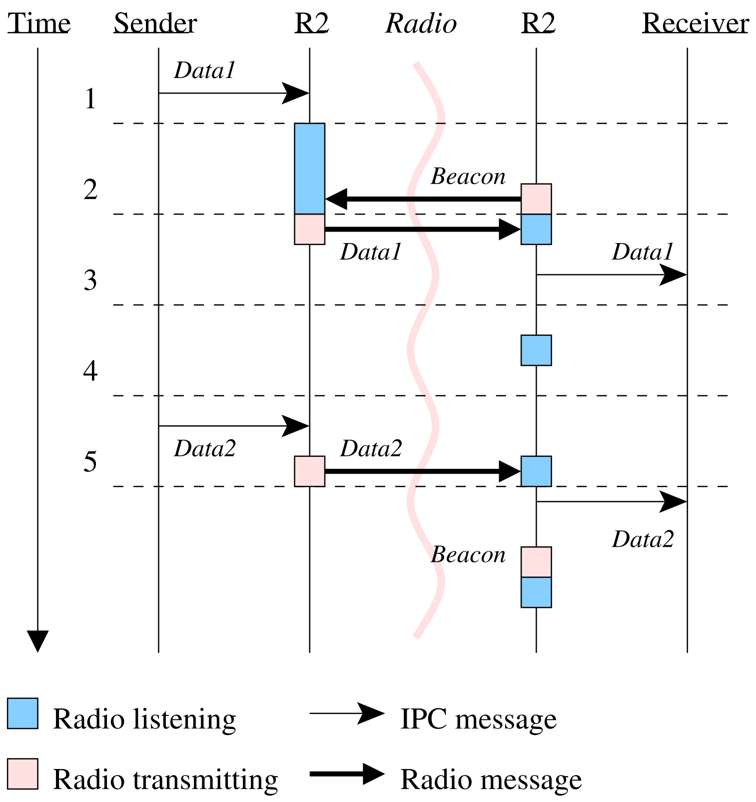


Figure 5.2: Transmitter-Managed rendezvous

1. Application A queues a message with the R2-Layer, to be transmitted to application B.
2. The sending device remains powered until a beacon is received from the target device.
3. The sending device transmits the data to coincide with the receiver's listen window.
4. No further data is queued so the transmitter remains inactive.
5. A queues more data to be sent to B. Because B has advertised multiple receive slots per beacon, the R2-Layer can transmit the datagram in the next available window without waiting for a beacon.

ested clients. Each client will have registered with the sensor, requiring the sensor to both accept incoming registration requests and to maintain a table of interested parties.

Both the complexity and the power consumption of simple sensor devices can be greatly reduced by including the most current sensor state in each beacon that the sensor node transmits. Clients are then responsible for synchronising themselves with sensors in order to receive data, removing the need for registration of interest, or for multiple re-transmission of the same sensor data to multiple clients.

Receiver-managed communication places the emphasis upon the device wishing to receive data to perform the necessary steps to rendezvous with the transmitter. In its simplest incarnation, transmitters simply embed small amounts of data in regular beacons. Receivers then synchronise to these beacons to retrieve the data. To support sensor applications in which state is constant for prolonged periods but may change at a moment's notice, optional transmission slots provide a way for sensors to both conserve power and deliver fast response times. Receivers can opt to wake only for the guaranteed beacon datagrams, or to wake for each optional transmission slot as well.

Figure 5.3 illustrates the operation of a receiver-managed scheme.

If the data fed from a sensor is sufficiently compact then the beacon datagram may contain both sensor information and sufficient supplemental information that the type of sensor can be established without requesting such information explicitly. Multiple sets of sensor information can be cycled in consecutive beacon payloads in order to increase capacity at the cost of timeliness, with static information such as the sensor type being transmitted less frequently than more dynamic information.

Sensors using a receiver-managed scheme in this way can be made significantly simpler than with a transmitter-managed implementation. The sensor does not need to maintain a list of interested parties, nor must it re-transmit the same information to multiple clients. The reception part of a standard PEN node can be omitted if there is no other reason for clients to send information to the device.

Conversely, receiver-managed and transmitter-managed protocols can co-exist in a single device in order to exploit the best of both worlds. A transmitter-managed RPC protocol can be used to obtain static information about a sensor,

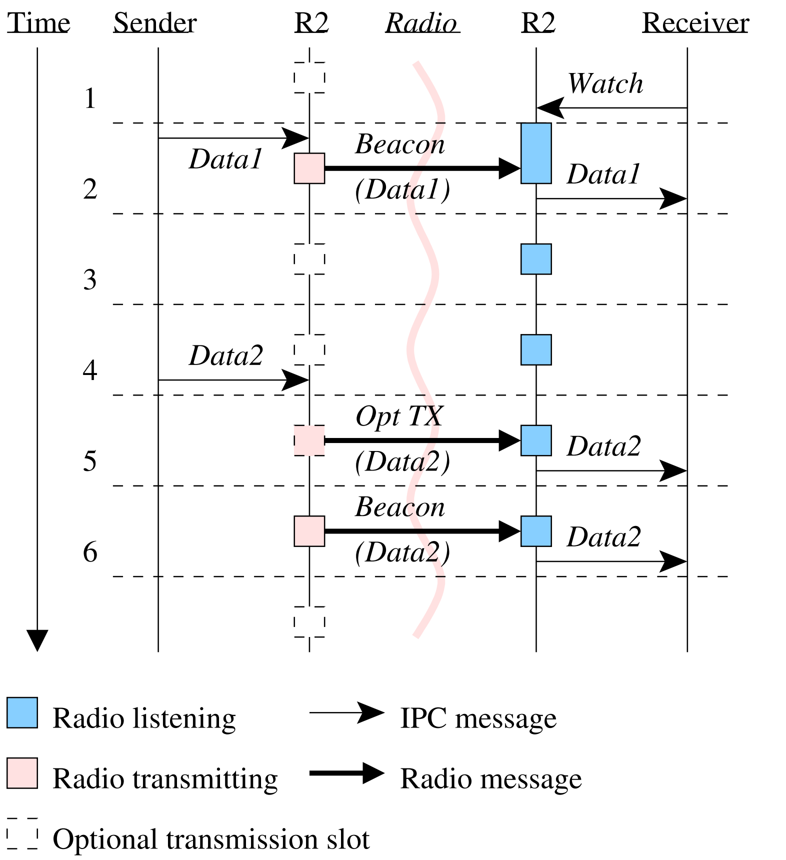


Figure 5.3: Receiver-Managed rendezvous

1. Application B informs the R2-Layer that it wishes to receive beacons from application A.
2. A's R2-Layer beacons on A's behalf. B's R2-Layer catches the beacon and passes it to B.
3. B listens for A's optional transmission slot but A has no new data to transmit.
4. A passes a new beacon payload to the R2-Layer.
5. The payload is delivered in an optional transmission slot.
6. The R2-Layer transmits the normal beacon packet.

such as its type and the units in which it will supply readings. Readings may then be supplied in beacon datagrams, reducing the burden on the sensor.

5.2.4 Discovery

To convey data between two low power PEN nodes requires a rendezvous stage followed by actual transfer of data. Before this can take place, however, the involved parties must be made aware of each other. This is achieved during a *discovery* phase, prior to rendezvous.

When operating from a relatively limitless[1] power source, such as mains power, a PEN node may remain awake and able to receive data at all times, making discovery of newly arrived devices a trivial task.

Discovery is more troublesome for low power devices, which must spend most of their time asleep and unable to hear newly arrived nodes' beacons. The existing R-Layer implementation leaves the issue of device discovery to be implemented by each application according to its needs. The R2-Layer allows applications similar freedom in their choice of discovery algorithm and provides a set of power-aware discovery primitives to simplify their implementation.

5.2.5 Triggered Reception Burst

Perhaps the simplest scheme for discovering nearby nodes is to wake and listen for a pre-defined *listen window (LW)* whenever a particular event occurs, such as the user pressing a button. The window to listen for should be at least as long as the maximum *beacon interval (BI)* of nodes which the listener may be interested in, so as to ensure that they will be discovered if they are present.

Triggered reception bursts benefit from the fact that they are manually triggered by the user — unless the user requests that the node attempt to discover nearby devices, no power is used. For semi- or fully-autonomous devices, however, manual intervention by the user is unacceptable, motivating the need for more passive discovery schemes.

[1]A power source is deemed "Relatively limitless" if the maximum power which a PEN node is capable of draining is significantly less than the power source is designed to support.

5.2.6 Regular Burst

The most obvious way to adapt the triggered burst scheme to perform passive discovery is to use a timer internal to the node as the trigger. The timer will typically be set to trigger a burst once every few minutes.

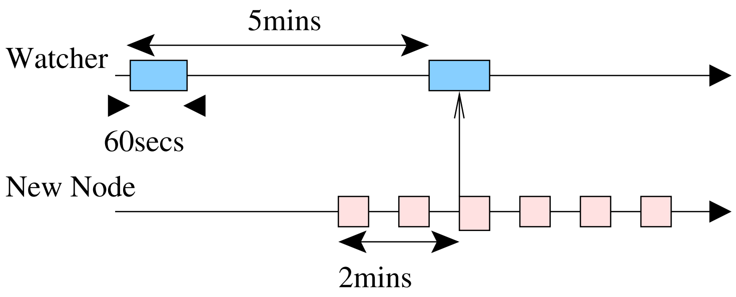


Figure 5.4: Discovery using regular bursts

Figure 5.4 shows a PEN node watching for devices with beacon intervals of up to sixty seconds. The watcher uses a *listen interval(LI)* of five minutes, and a listen window of sixty seconds. In the example, an arriving device remains undiscovered for two minutes because it arrives during the sleep part of the watcher's listen interval. The listen window is chosen to ensure that, if present, the device will be discovered, and must therefore be at least as long as the device's beacon interval. This assumes that the communication medium is ideal and that all beacons are therefore guaranteed to be received.

The maximum delay between a device's arrival and its discovery using regular bursts is termed the *Maximum Discovery Latency (MDL)*. Assuming a perfect communication medium, and that the target's beacon interval does not exceed the listen window, the MDL is given by Equation 5.1. Not all beacons will be correctly received in practice, meaning that the discovery guarantees we consider here are inherently soft — adverse network conditions may invalidate them. Only in lightly loaded radio environments can we meaningfully specify an upper limit on discovery times.

$$MDL = LI \tag{5.1}$$

Thus, in our example,

$$MDL = 5min \tag{5.2}$$

5.2.7 Co-Prime Regular Burst

When a PEN node wishes to discover other devices with a particular beacon interval, the regular burst scheme provides optimal results. If the node instead wishes to discover any devices which use one of several different beacon intervals, regular bursts are no longer optimal.

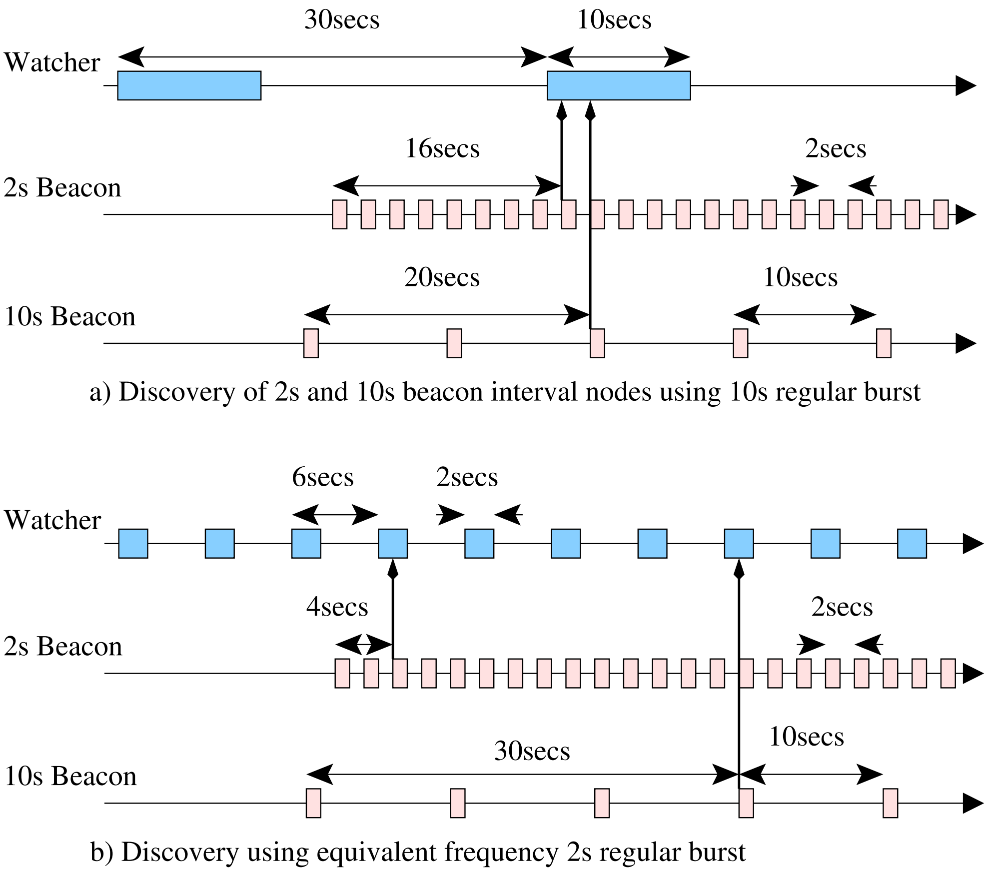


Figure 5.5: Discovering nodes with different beacon intervals using regular bursts

Figure 5.5 illustrates why regular bursts are sub-optimal when listening for nodes with different beacon intervals. The example shows two devices, with beacon intervals of two and ten seconds respectively, being discovered.

Figure 5.5a shows a node using ten second listen windows and a listen interval of thirty seconds to discover the two devices. Using equation (5.1) we find:

$$MDL(2s) = MDL(10s) = 30sec \tag{5.3}$$

Figure 5.5b shows the watcher using two second listen windows and a listen interval of six seconds, providing the same power saving as in 5.5a. Using (5.1) we find:

$$MDL(2s) = 6sec \tag{5.4}$$

Assuming that we are guaranteed also to discover 10s, it is the case that only one in every five of the two second windows will "hear" the beacon, since each window covers only one fifth of the ten second beacon interval. Only after five complete listen intervals are we guaranteed to have discovered the 10s node. We must re-write (5.1) as:

$$MDL = (LI * (BI/LW))) \tag{5.5}$$

from which we obtain:

$$\begin{aligned} MDL(10s) &= (6sec * (10sec/2sec)) \\ &= (6sec * 5) \\ &= 30sec \end{aligned} \tag{5.6}$$

It follows from (5.5) that reducing the listen window to be shorter than the minimum desired beacon interval provides no benefit, since we may then take several listen intervals to guarantee discovery.

It is important, however, to consider whether or not we are guaranteed to discover node 10s in the latter case. We are clearly guaranteed to discover node 2s, since each listen window is long enough to discover the node. We are guaranteed to discover nodes with beacon intervals greater than the length of the listen window if and only if the beacon and listen intervals are co-prime. Co-prime intervals share no common factors, so that beacons will explore all potential listen windows as time progresses, thus guaranteeing that an actual listen window will eventually be encountered. To verify that the two intervals are co-prime, we first normalise them into multiples of listen windows,

$$Interval_{norm} = Interval/LW \tag{5.7}$$

before testing for co-primality:

$$guaranteed = coprime(BI_{norm}, LI_{norm}) \tag{5.8}$$

It is important to note that we cannot proceed unless all normalised intervals calculated using (5.7) are integer. To ensure that this is the case, we alter our choice of listen window to be the Greatest Common Divisor(GCD) of the supplied beacon intervals, rather than the minimum.

$$LW = GCD(BI_1, BI_2, ...) \tag{5.9}$$

Ideally, the minimum beacon interval should also be the greatest common divisor and will therefore be the chosen length of listen window, since this provides the greatest efficiency. For the example given in Figure 5.5b, using 5.8 we find:

$$\begin{aligned} LW &= GCD(10sec, 2sec) = 2sec && (5.10)\\ guaranteed &= coprime((10sec/2sec), (6sec/2sec)) && (5.11)\\ &= coprime(5, 3)\\ &= True \end{aligned}$$

In this case we have, by sheer good fortune, picked a suitable listen interval to guarantee discovery of both 2s and 10s.

Implementation

A simple implementation of this scheme uses the following algorithm:

1. Determine the listen window using equation (5.9). e.g. from Fig.5.5b,

$$LW = GCD(10sec, 2sec) = 2sec \tag{5.12}$$

2. Normalise the supplied intervals into "slots" (numbers of listen windows) using (5.7). e.g. from Fig.5.5b,

$$BI(10s)_{norm} = 10sec/2sec = 5slot \tag{5.13}$$

and

$$BI(2s)_{norm} = 2sec/2sec = 1slot \tag{5.14}$$

3. Select a suitable normalised listen interval using the formula:

$$LI_{norm} = (MDL_{norm} * min\_BI_{norm})/max\_BI_{norm} \tag{5.15}$$

and round down to the nearest integer. e.g. from Figure 5.5b,

$$LI_{norm} = (15slot * 1slot)/5slot = 3slot \tag{5.16}$$

4. While the normalised listen interval is non-coprime with one or more of the desired beacon intervals, increment the selected value.

5. Having potentially increased the listen interval, we must verify that the Maximum Discovery Latency is still valid. We calculate the new MDL using (5.5) and compare the result to the desired MDL.

6. If the new MDL exceeds the desired MDL then we wrap the sequence of listen windows within the desired MDL, so as to provide sufficient listen windows to guarantee discovery while at the same time providing the required maximum discovery latency.

Given a set of possible beacon intervals of nodes we wish to be able to discover, we can ensure discovery while at the same time minimising the latency in discovering nodes with shorter beacon intervals by use of the Co-Prime Regular Burst algorithm.

5.2.8 Statistical Burst

An alternative to co-prime discovery is to use randomly distributed listen windows. Random listen windows cannot be used to guarantee discovery within a specified time limit but can be parameterised so as to provide statistical guarantees as to the likelihood of discovery. As will be shown, the motivation for using a Statistical Burst discovery algorithm rather than Co-Prime Burst is that a typical implementation can be significantly simpler.

Theory

We perform statistical burst discovery using fixed-length Listen Intervals, each containing a single Listen Window. Assuming devices operate the discovery algorithm for an indefinite period, we can calculate the probability of discovering a nearby node in one of an effectively infinite number of listen intervals in order to determine the algorithm's characteristics.

Firstly, we define $P_{HIT}(BI, LW)$ to be the probability of discovering a device with beacon interval BI, using a single listen window LW. The window will be placed randomly within a particular listen interval LI. We also define $P_{MISS}(BI, LW)$ such that

$$P_{HIT}(BI, LW) = 1 - P_{MISS}(BI, LW) \tag{5.17}$$

for convenience.

Note that $P_{HIT}(BI, LW)$ is independent of the listen interval used. To understand why this is the case, it is helpful to regard the listen window as occurring at a fixed point in time, with beacons occurring periodically but with a random skew from the listen window. Regarding the system in this way, it is clear that the probability of hearing at least one beacon in a given listen window is dependent only upon the frequency of beacons and the size of the window. The only effect of changes to the interval between listen windows can be to reduce or extend the delay in detecting a beacon, rather than the probability of doing so.

We therefore find that:

$$P_{HIT}(BI, LW) = \frac{LW}{BI} \tag{5.18}$$

It is worth commenting that equation (5.18) is valid only when the beacon interval is greater than or equal to the listen window. In the case that the beacon interval is less than or equal to the listen window, discovery is guaranteed, since it is impossible that the entire listen window be contained between two consecutive beacons. Equation (5.18) is safe to use for our purposes because we choose the listen window to be the minimum of the beacon intervals that we are interested in discovering, as described in Section 5.2.8.

The independence of P_{HIT} from the listen interval is an important result since it greatly simplifies the task of choosing a suitable interval. The listen interval must be chosen such that the probability of discovering nodes with the required beacon intervals within a certain time exceeds a minimum limit. To do this, we first establish the number of listen intervals required to achieve the target probability of discovery. We define $P^N_{HIT}(BI, LW)$ to be the probability of discovery after N intervals (and similarly $P^N_{MISS}(BI, LW)$, such that:

$$P^N_{HIT}(BI, LW) = 1 - P^N_{MISS}(BI, LW) \tag{5.19}$$

Since the listen window is placed randomly within each listen interval, it is the case that the probabilities of discovery in those intervals are independent of each other, so that:

$$P^N_{MISS}(BI, LW) = P_{MISS}(BI, LW)^N \tag{5.20}$$

and thus:

$$P^N_{HIT}(BI, LW) = 1 - P_{MISS}(BI, LW)^N \tag{5.21}$$

This simple result allows us to find the required number of listen windows which must occur within a given period to ensure the desired probability of discovery. Given a required probability of discovery of P^{min}_{HIT} (and hence a corresponding required upper limit upon the probability of failure P^{max}_{MISS}), we select the lowest value of N such that $P^N_{HIT}(BI, LW) >= P^{min}_{HIT}$. Once N is established, it can be combined with the specified time limit to calculate the listen interval.

Implementation

We implement statistical burst discovery using a similar set of steps to those of the coprime burst algorithm.

1. Determine the listen window. This is a simpler task for statistical burst than for coprime burst because we no longer suffer the constraint that all beacon intervals be multiples of listen windows. Instead, we can simply select the smallest beacon interval and use that as the listen window:

   $$LW = BI_{min}$$

   e.g. from Fig.5.5b we find

   $$LW = 2sec$$

2. Calculate $P_{HIT}(BI_{max}, LW)$ using (5.18). We need only consider the probability of discovery per interval of the longest beacon interval in which we are interested, because it is the least likely to be discovered.

   e.g. from Fig.5.5b,

   $$\begin{aligned} P_{HIT}(BI_{max}, LW) &= P_{HIT}(10sec, 2sec) \\ &= \frac{2sec}{10sec} \\ &= \frac{1}{5} \end{aligned} \tag{5.22}$$

3. Determine how many listen intervals we must fit within the Maximum Discovery Latency to achieve the desired probability of discovery, P_{HIT}^{min}. We must ensure:

   $$P_{HIT}^{N}(BI, LW) >= P_{HIT}^{min} \tag{5.23}$$

   or, equivalently:

   $$\begin{aligned} P_{MISS}^{N}(BI, LW) &<= P_{MISS}^{max} \\ P_{MISS}(BI, LW)^{N} &<= P_{MISS}^{max} \end{aligned} \tag{5.24}$$

Using this latter form, we can select a suitable value of N. We first find a value of x such that

$$P_{MISS}(BI, LW)^x = P_{MISS}^{max} \tag{5.25}$$

using the equation

$$x = \frac{\log P_{MISS}^{max}}{\log P_{MISS}(BI, LW)} \tag{5.26}$$

We may then round x up to the next integer to obtain a suitable value for N.

e.g. from Fig.5.5b, with $P_{MISS}^{max} = \frac{1}{4}$,

$$\begin{aligned} P_{MISS}^N(10sec, 2sec) &<= \frac{1}{4} \quad (5.27) \\ P_{MISS}(BI, LW)^N &<= \frac{1}{4} \\ \tfrac{4}{5}^N &<= \frac{1}{4} \end{aligned} \tag{5.28}$$

which, using (5.26) gives

$$\begin{aligned} x &= \frac{\log \frac{1}{4}}{\log \frac{4}{5}} \\ &= \frac{-1.386}{-0.223} \\ &= 6.212 \end{aligned} \tag{5.29}$$

Given $x = 6.212$, we round up to obtain $N = 7$. i.e. To ensure a 75% chance of discovery of node 10s using 2sec listen windows, we must use at least 7 listen intervals.

4. Calculate the listen interval from the Maximum Discovery Latency and required number of listen intervals, N:

$$LI = \frac{MDL}{N} \tag{5.30}$$

e.g. from Figure 5.5b,

$$LI = \frac{30sec}{7} \approx 4.29sec \tag{5.31}$$

Therefore we must listen for 2 seconds out of every 4.29 seconds in our example.

5.2.9 Implementation Tradeoff

Qualitatively, statistical burst discovery has little to offer over the coprime burst scheme. While coprime burst guarantees discovery of devices, if present, with strictly constrained latency, statistical burst offers only that it will "probably" discover devices. Because of the stochastic properties of the PEN MAC Layer, the guarantees provided by coprime burst schemes are also probabilistic, albeit with greater likelihood of success.

To achieve a reasonable probability of discovery, statistical burst will generally require more bursts than coprime schemes, reducing the scope for power saving. For applications involving very simple appliances, particularly networks of wireless sensors, this may be outweighed by the simplification in the calculation of a suitable listen window and interval. In relatively static environments, the maximum discovery latency may be increased significantly to improve power saving with little impact on the application's performance.

Perhaps the most significant advantage of statistical burst is that it can be used to discover appliances with unknown beacon intervals. If a node has a beacon interval which is a multiple of the listen interval used by coprime burst, then it is likely that it will never be discovered. If the beacon sequence is phase-shifted from the listen window sequence then it will theoretically remain so, making it impossible that the node be discovered. Since statistical burst uses randomly placed windows, all nodes will eventually be discovered, regardless of their beacon interval.

A combination of the two schemes could be used to discover devices with unexpected beacon intervals with greater efficiency, by calculating a co-prime interval and applying a random jitter to it. Such a scheme would be as complex to operate as co-prime burst, and it is not clear that significant benefit would be gained for day-to-day operation.

5.3 R2-Layer Implementation

The R2-Layer is implemented as a library of partially interdependent modules, each implementing a particular low power protocol primitive.

5.3.1 Control vs Data Flow

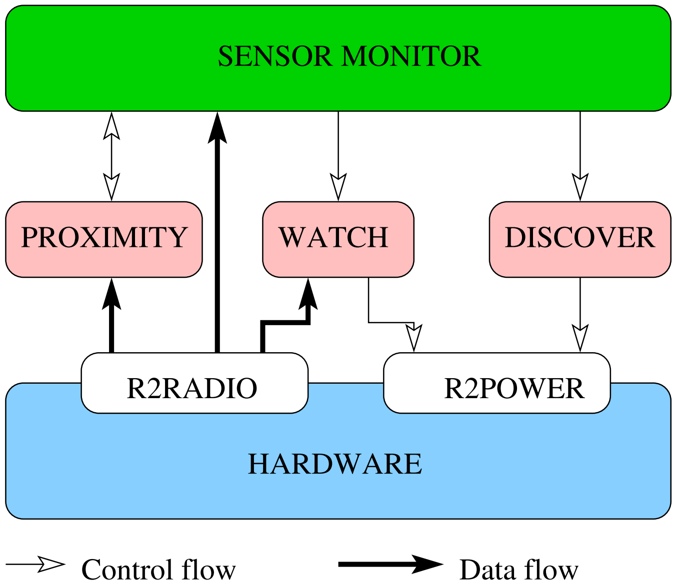


Figure 5.6: Control and data paths in a sensor monitoring appliance

The R2-Layer avoids the merging of data and control paths between modules wherever possible. Figure 5.6 illustrates the usefulness of this separation by considering a device that monitors any local sensors it encounters on its travels. This task requires the device to operate a discovery algorithm to detect new sensors arriving, and a synchronisation algorithm to "watch" discovered sensors until they depart.

The sensor monitor uses a standard R2-Layer module, labelled "DISCOVER" in our example, to handle discovery on its behalf. It supplies information on the likely beacon intervals of sensors it wishes to discover and on the maximum discovery latency it requires. The discovery module takes responsibility for powering the radio at appropriate intervals to ensure discovery of other nodes but takes no other part in the system's operation.

The monitor application also uses a standard proximity-store module to maintain a list of nearby sensors. The proximity store registers itself with the radio in order to receive incoming datagrams but performs no power management, instead relying on that performed by the discovery module to suffice. Whenever a new sensor arrives, the proximity store informs the monitor application, that may then take steps to monitor the sensor. For other applications, the same proximity-store implementation could be used in the absence of the discovery module, relying on power management performed by other components to opportunistically discover nearby appliances without using extra power.

To monitor the sensor, a "watch" module is used, which controls the power interface in such a way as to ensure that the monitor node is awake whenever a sensor it is monitoring transmits a beacon. To obtain the information conveyed in sensors' beacons, the application registers itself with the radio module to receive any incoming datagrams which contain payloads.

5.3.2 The Core Modules

Each R2 Layer module is implemented as a *class*, of which new *instances* may be created and existing instances destroyed at any time. While the majority of R2 Layer module instances will be created at boot time and never destroyed, some modules are re-used extensively to perform transient, short-lived operations.

Dependencies between modules are enforced by requiring instances of modules upon which a particular primitive depends to be passed to the constructor function. If a module supports optional functionality that requires the presence of extra modules in addition to those it normally depends upon, then the optional functionality will require instances of the extra modules to be supplied. In this way, dependencies may be checked at compile time, greatly simplifying development.

Despite its modular design, the R2 Layer has a core set of modules which will be implemented in most, if not all, devices. The core radio and power management modules, described in Section 5.3.3, are required for an R2-Layer device to operate in any useful way.

5.3.3 The Radio and Power Interfaces

The data transfer and power control functions of the standard PEN radio interface are split into two distinct interfaces, described below.

R2RADIO

The R2RADIO interface provides functions to transmit datagrams and to specify where incoming datagrams should be delivered. The interface is not concerned with power management issues or multiplexing between multiple clients, greatly simplifying its implementation. If a datagram is passed to the radio module for transmission while the hardware is not powered then it will be queued until the radio is next activated. Similarly, if the radio hardware is not powered then no datagrams can be received.

R2POWER

All power management of the radio interface is achieved via the R2POWER interface. Each R2POWER object provides a reference-counted power management interface. If multiple instances exist then their reference counts are treated as "votes" for the state of the radio hardware. Only when all reference counts are zero will the radio hardware be powered down. The destructor for an R2POWER instance may be registered with the PEN runtime to be called when a specified thread quits or crashes, allowing the impact of failed applications upon power consumption to be minimised.

The PEN platform's radio interface accounts for approximately one third of a node's power consumption when fully active, excluding peripheral devices such as sensors or actuators. The CPU and memory modules account for a further third each, approximately, as described in Section 5.4. Power management of the radio module is especially important in PEN systems because the CPU must remain active when listening for incoming data. This is required in order to service incoming radio data quickly enough, since there is a significant delay in activating the CPU while waiting for the crystal oscillator to settle before useful work can be done.

5.3.4 Packet Framing and Multiplexing

The R2-Layer provides a standard and extensible datagram format, shown in Figure 5.7, providing a number of optional fields in addition to the mandatory PEN packet headers. Each field is annotated with its length. Fields with a length of 0-N are variable-length fields, and include encoded integer fields such as TX-slots and RX-slots, as well as the optional payload.

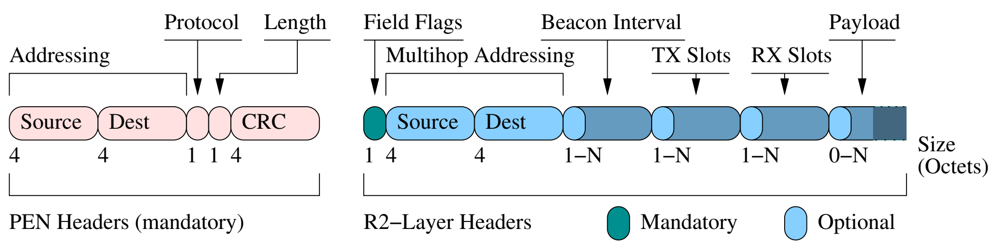


Figure 5.7: Structure of R2 Layer datagrams

R2-Layer datagrams contain a mandatory “flags” field, some of whose bits are used to indicate the presence of the other, optional, fields. Optional fields include “beacon interval”, “TX slots” and “RX slots”, which are used by higher-level modules to convey beacon and rendezvous information to other devices. Supplemental source and destination MAC address fields are provided, in order to support routing schemes — the PEN equivalents provide addressing on a hop-by-hop basis.

R2TX

Packet framing of outgoing datagrams is performed by the R2TX module. Clients supply a set of flags to indicate which of the optional R2-Layer datagram fields they wish to pass in the transmitted packet and a set of values to place in those fields. A module designed to perform periodic beacons, for example, would pass only a beacon interval value to the R2TX module. If the device in question were a simple sensor, the payload field might also be specified, in which the sensor's state would be stored.

Although the R2 Layer follows a principle of separation between the data flow and power control components of the system, the R2TX module is a notable exception. Whenever a datagram is passed to the R2TX module, it will by default ensure that the radio hardware is powered until the datagram has been transmitted by the lower level R2RADIO module. In doing so, the R2TX module is implicitly assuming that datagrams should be delivered as soon as possible after they are passed to it.

If the module did not perform power management to ensure immediate transmission, then datagrams might instead be queued and only sent when some other action caused the radio to become powered. Because most low power devices will spend significant amounts of their time in sleep states, unable to receive data, this kind of "lazy" transmission is not usually of great use. Instead, datagrams should be sent as soon as the higher level protocol modules queue them, since higher level modules will know the required transmission timing for the low power protocol they implement.

Implicit power management during transmission can greatly simplify clients of the R2TX module but violates the principal of separation between data flow and power control, so R2TX also allows datagrams to be specially queued without implicit power management.

R2RX

The R2RX module performs the complementary unmarshalling operation to the R2TX module, but does not perform any power management. Implicit power management works for datagram transmission because it can be assumed that when a datagram is queued, it should be sent as soon as possible. To receive data, clients instead register themselves as recipients for incoming datagrams and must arrange appropriate power management separately. The R2RX module allows multiple clients to be registered simultaneously, and can perform filtering based upon the presence or absence of specific optional R2-Layer fields in a datagram, to avoid passing irrelevant datagrams to clients where possible.

If implicit power management were performed by the R2RX layer then either the module or its clients would be required to implement power management and rendezvous algorithms. Either approach would lead to extra complexity in either the R2RX module or its clients and would tie the system to one particular power management scheme.

Instead, the R2RX layer assumes that power management will be performed by a third party. R2RX clients may opt to perform their own power management, or can instead defer the task to a standard module dedicated to the task. One side-effect is that a module may receive datagrams from R2RX even when it has not powered the radio up. This can occur when another module in the system has powered the radio — datagrams will be passed to all registered endpoints regardless of which endpoint requested the radio to remain powered.

It is often the case that protocol modules can optimise their behaviour by taking advantage of information contained in or inferred from incoming datagrams that they are not expecting. A proximity store module, discussed later, which is responsible for keeping a table of local nodes, can take advantage of this behaviour to detect newly arrived nodes by registering itself with the R2RX module and handling any datagrams it is passed. The store module will not perform power management itself, but will instead rely on other system modules to perform that task.

5.3.5 Protocol Modules

Messaging Modules

The R2-Layer provides modules through which clients may access transmitter-managed and receiver-managed protocols. For each type of protocol, transmitter and receiver modules are provided, so that devices need include only the functionality they require. Further modules are also provided to allow multiple applications to co-exist on a single device, by assigned each its own APplication IDentifier (APID). Some well-known APIDs are defined for particular services, and new APIDs may be assigned on the fly for transient operations.

Although the R2-Layer defines one transmitter- and one receiver-managed protocol, new implementations may be used interchangeably without requiring changes to other levels of the system, at the cost of compatibility between devices using the new and old R2-Layer implementations. New power management strategies capable of re-using the existing R2-Layer datagram format may maintain some level of compatibility with older implementations.

Discovery

Discovery is divided between two modules in the R2-Layer. The R2PROXIMITY module registers itself to receive all incoming datagrams and uses these to maintain a table of nodes which have been heard from recently.

In keeping with the principle of separation of data and control flow within the R2-Layer, power management is not a function of the proximity store. Instead, power management for the discovery function is performed by the R2SNOOP module. Like the messaging modules, the snoop module has a well-defined interface allowing different discovery schemes to be used interchangeably. This allows a device to be switched from Regular Burst to a Co-Prime Burst scheme without altering surrounding code in the device.

Low Battery Notification

The PEN prototype hardware makes it possible to monitor the battery status in order to provide a low battery indicator. User interface devices can use this to display an appropriate icon or to power down sections of the unit. Small devices such as sensors may seldom be seen by the user, rendering such notifications of little use.

The R2-Layer provides a battery monitor module which checks the battery level at hourly intervals. If the battery level drops below a predetermined threshold then the module will add a short battery-level message, addressed to the well-known battery notification service APID, to the payload of subsequent beacon datagrams at regular intervals. The payload consists of a single byte containing the battery level. The practical size of the payload includes an extra byte to encode the length, plus four bytes of source and destination APID information. At 24Kbps, the extra six bytes adds 2ms transmission time to each beacon, therefore it is added to beacons only infrequently, so as to minimise the extra power drain.

Alternatively, a battery monitor module may be installed which uses the `R2TX` module to transmit a notification datagram directly to a local battery monitor service. This suffers from the need for the transmitting node to first discover whether or not a battery monitor is available. If no monitor is available then the node will wake for a full beacon interval without having performed any useful function.

A second battery monitor module may be installed which increases the minimum beacon interval allowed as the battery level drops, to reduce power consumption and prolong the node's life. This may prevent the beacon module from satisfying all requested beacon intervals, in which case clients whose requests cannot be satisfied are notified of the failure.

5.4 Performance

To illustrate the advantages of receiver-managed rendezvous for monitoring applications, a set of wireless lamps and light sensors have been constructed using the R2-Layer. The lamps each synchronise with a sensor to obtain light level readings. When a sensor reports light levels below a particular threshold, any lamps watching it will switch on. This simple application was constructed with no power management, with transmitter-managed rendezvous and with receiver-managed rendezvous.

5.4.1 Reference Implementation

The reference implementation uses PEN's raw radio interface to communicate between lamp and sensor. No power management is performed. The lamp simply listens for broadcast beacon datagrams from nearby sensors. Sensors transmit broadcast beacon datagrams at regular intervals, which contain their current sensor state.

This implementation requires that both sensor and lamp remain powered at all times. Keeping the radio hardware powered accounts for around 30mA. To handle incoming datagrams in a timely fashion requires the CPU to remain powered, consuming a further 30mA. RAM access by the CPU drains another 30mA at peak. A node therefore drains 90mA to operate all three components. A more efficient idle routine can reduce the CPU contribution and almost eliminate the drain due to RAM, leading to a total drain of 55mA. In tests, PEN nodes continuously listening for incoming datagrams can operate for around eighteen hours from a single UltraLife 9V battery [70]. Because no power management is performed, response times are in the order of a few hundred milliseconds.

5.4.2 Transmitter-Managed Implementation

The transmitter-managed implementation is based upon the primitives provided by the `R2TX` module. Lamps are configured with 2s beacon intervals, to provide favourable response times. At start of day, lamps enter a discovery phase, listening for nearby sensors. When a suitable sensor is discovered, they transmit a request to it to register an interest in notifications of changes in its state. Sensors will notify each registered lamp with a transmitter-managed, unicast notification message every time their state changes.

Because they need only accept requests during the initial registration phase, sensors beacon at 10s intervals. Each beacon takes 20ms to transmit and is followed by a 30ms listen window, during which incoming datagrams may be accepted. The rest of the beacon interval is spent with the radio, CPU and memory powered down, resulting in power consumption of around 100μW.

Waking for only 50ms every 2s, a PEN node attached to a bike lamp can operate on an UltraLife battery for around one month, with a separate battery pack powering the lamp itself. Sensors waking for 50ms every 10s can notify a single lamp for a projected three months. Registration of multiple lamps reduces this lifetime, since the sensor must notify each lamp individually of any change in state. With no lamps registered, the sensor can operate for five months. The system's response varies between twenty milliseconds and two seconds, since sensor data can only be transmitted to a lamp immediately after the lamp has beaconed.

An R2-Layer implementation could improve the lamp's performance through the use of optional reception slots between beacons. These slots would allow the lamp to wake for 50ms every 10s, to transmit a beacon, with four further 30ms listen windows between each beacon, increasing the lamp's lifetime by almost 50%.

5.4.3 Receiver-Managed Implementation

The final implementation uses the R2-Layer's receiver-managed rendezvous protocol, simplifying the sensor device compared with the transmitter-managed implementation and further reducing the system's overall power consumption. Lamps discover sensors by using the R2-Layer discovery primitives to power the radio until a beacon is received from a nearby sensor. Once a beacon is received, the lamp uses the R2-Layer to synchronise with subsequent beacons.

Sensors are configured with 10s beacon intervals as before. Each beacon takes 20ms to transmit and need not be followed by a listen window, since registrations of interest are no longer required from lamps. To provide shorter response times, sensors also advertise optional transmission slots every 2s. These slots are only used if the light level changes between beacons. If the light level does not change then only standard beacon packets are used. The behaviour of lamps depends upon the required response time. Using the sensors described above, lamps can operate with response times of either 10s or 2s, depending upon whether or not they choose to listen for optional transmission slots. Reception of sensor data requires lamps to wake for only 30ms, since no beacons need be transmitted.

Sensor lifetimes range from one year to two months, depending upon how variable the light level is. Lamps can operate for eight months or fifty days, depending upon the required response time. These lifetimes are unaffected by the number of lamps watching a sensor, because each sensor data payload is broadcast once to all lamps. In busy radio environments, some sensor beacons or optional transmissions may be lost, resulting in longer response times.

Property	PEN (raw)	TX (N clients)	RX
Duty Cycle (ms/s)	1000	5+2N	2-10
Response time (ms)	50	20-2000	20-2000
Est. Lifetime (days)	0.75	75	75-375

Table 5.2: Properties of Sensor implementations

Property	PEN (raw)	TX	TX+opt slots	RX (2s)	RX (10s)
Duty Cycle (ms/s)	1000	25	17	15	3
Est. Lifetime(days)	0.75	30	44	50	250

Table 5.3: Properties of Lamp implementations

5.4.4 Results

Tables 5.3 and 5.2 summarise the performance of the raw, `R2TX` and `R2RX` implementations of the lamp and sensor application. These results are based on relatively short-term observations of node's behaviour using 9V UltraLife batteries and assume ideal battery behaviour. Martin et al [40] discuss the shortfalls of duty-cycle based estimates for lifetimes of low power appliances. Such estimates often fit observed behaviour very poorly, owing to non-ideal battery characteristics at high loads. The PEN prototype hardware drains sufficiently little power even when fully active that the battery will be operate more or less ideally, especially since peak power is only consumed in extremely short bursts. Future PEN hardware is envisaged to have at least an order of magnitude lower peak power consumption, further reducing non-ideal battery effects in theory. Provided that battery technologies are chosen carefully, the results presented here will remain valid.

5.5 Summary

The main goal in designing the R2-Layer was to provide a single low power protocol stack capable of supporting a variety of applications and host appliances. By dividing the protocol stack into a number of loosely interdependent modules, appliances may be constructed which contain a minimal body of resident code. The power consumption and responsiveness of appliances can be tailored to their applications by using appropriate primitives. Most primitives can support dynamic reconfiguration, allowing beacon intervals to be varied according to the requirements of clients, for example.

The R2-Layer services presented here support only single-hop communication between co-located devices, although the system is designed to be extensible to support multi-hop or tunnel routing, as described in Chapter 6, if required. While the R2-Layer interfaces may be used directly to build simple applications, as we have illustrated in Section 5.4, its interfaces are too low level to build more complex distributed applications upon directly. Chapter 7 describes an object-request broker architecture designed to make full use of the R2-Layer's features while at the same time presenting a more palatable interface to applications.

Chapter 6

Mobility

6.1 Mobile Devices

In the previous chapter we discussed techniques for supporting ad-hoc low-power communication between arbitrary devices. Although short range communication is sufficient, and in many cases necessary, for a significant number of the applications we wish to consider, there nonetheless exists a valuable body of applications requiring support for wide-area communication.

Mobility is the characteristic of embedded wireless devices most directly motivating support for wide area communication. If wireless devices are highly mobile then they will move rapidly in and out of range of each other, rendering most useful interactions impossible. The area over which a node can communicate is often termed a radio "cell". We describe a wireless device as "highly mobile" when the time taken for the device to traverse a distance equal to its radio cell size drops below some threshold, i.e. when the device travels through at least N cells in a given time period.

Highly mobile devices using the standard single-hop R2-Layer protocols are effectively incapable of communication, since they will usually have moved out of communications range of any device they have sent a request to before it can transmit a reply. Some form of routing is required to extend the communications range of even very simple devices beyond that provided by the PEN hardware.

6.1.1 Multi-hop Routing

If devices are well powered and are therefore always able to communicate, then multi-hop routing schemes such as those defined by MANET [39] and CarNet [43] for IP may be used. Hyphos [62] describes a non-IP routing scheme for small devices. WINS [80] describes application level multi-hop routing protocols for networks of sensors.

Since low power devices spend most of their time unable to communicate, these protocols are inappropriate. Girling et al [17] describe some multi-hop routing schemes suitable for low power networks. Continuously mobile nodes will fare poorly using multi-hop routing over low power PEN nodes, since the high latencies in communication between nodes along the route will cause routes to be established too slowly.

The speed at which routes are established may be improved if all the nodes along a multi-hop route are powered, since this allows them accept incoming data at any time, not just after their beacons. High power multi-hop nodes must be attached to a power source such as the mains, but may otherwise remain wireless and require minimal maintenance.

6.1.2 Wired Backbone Routing

As the distance between two PEN devices increases, the feasibility of multi-hop routing decreases. The probability of finding a suitable route drops, while the likelihood of low power and therefore slow-response nodes participating in the route increases.

For long-distance communication, it is preferable to offload datagrams to a wired backbone network. Such networks are typically higher bandwidth and lower latency, and can use more efficient static-routing protocols.

Wired backbone routing requires that gateway PEN nodes be placed at the periphery of the wired network. These gateways accept data from nearby PEN nodes, targeted at remote mobile devices, and route them across the wired backbone to the gateway closest to the target.

6.2 Location

The most significant obstacle to supporting wide-area messaging is the tracking of mobile devices. Tracking highly mobile devices over a wide area becomes increasingly difficult as more devices are added. Lookup and routing services for wired networks are, unsurprisingly, best suited to nominally static environments. Such environments can be highly optimised through the use of local caching of location information. Naive caching schemes perform poorly in highly mobile environments because caches must be frequently flushed in order to avoid returning stale location information.

One example of large-scale wide-area location services are those used to support mobile telephone networks. Mobile phone services must track individual handsets' locations whenever they are active, and must be able to make this information available efficiently to callers. Mobile phone services are divided into cells, such that locating a particular handset involves that the nearest cell to it be known. Because cells are fairly large, a mobile phone user moving within a building, or around a village or small town, will probably remain within a single cell and therefore appear static to the rest of the system. PEN cells are typically two orders of magnitude smaller than those used by mobile telephony systems, greatly increasing the apparent mobility. Eucalyptus' design also assumes that typical users will carry more than one appliance, while it is normally reasonable to assume that users will carry only a single mobile phone.

6.2.1 World Model

In designing the routing services for PEN, we have assumed an environment of roaming mobile devices, communicating intermittently via *contact-points* with a backbone of static wired network infrastructure. Wireless-to-wired *gateways* are scattered over this backbone as required to provide the desired coverage.

The "location" of an entity in such an environment is described purely in terms of the contact points through which it may be accessed. When locating mobile wireless devices, those contact points are the gateways between the wireless and wired networks. When locating mobile CORBA objects, for example, the contact points would simply be computers capable of hosting such objects.

The main technology supporting the routing services described in this chapter is Predator [75], a distributed, scalable and robust location service capable of managing large numbers of highly mobile objects.

6.2.2 Naming and Addressing

For the purposes of this discussion, a distinction is drawn between the notion of *addressing* and that of *naming*. Whereas names refer to an entity and therefore have some logical significance, addresses simply refer to particular network locations at which named entities may reside. Addresses will often be structured so as to simplify the task of routing datagrams. Clients referring to objects by their address rely implicitly on those objects remaining static. Whenever a resource is relocated on the network the path required to access it may change, in which case it will usually receive a new address by our definition.

Mobile objects must therefore be referred to by their name. Objects are assigned names which stay with them regardless of their location on the network. Clients contacting such objects must resolve their name to find a current address for them. This might be performed transparently by the underlying network, or, more commonly, by a separate third-party residing at a well-known address in the network.

Needham [45] defines two kinds of names:

1. *Pure names* are bit patterns used as identifiers. They contain no internal structure and so may only be used to compare identities with each other. This comparison capability may be used to build tables in which to look up other information related to a pure name.

   Such names may be used to uniquely identify an individual within a population. The MAC addresses assigned to Ethernet devices may be considered pure names, for example.

2. *Impure names* are those which contain additional information. In addition to being comparable, this information may be used to guide searches for the named entity. Impure names effectively consist of a pure name, plus

further information sufficient to identify the naming authority responsible for it. DNS [42] names, for example, may be considered impure names by this rule, since they consist of a name valid only within a particular DNS domain, followed by the impure name of that domain.

Mobile objects should ideally be assigned *pure* names, by which they may be identified and looked up in location tables, but which convey no location-dependent component. Clients wishing to contact an object pass its name to a location service which finds the object's current address and returns it.

Pure Naming

This approach is used by the Globe [23] Object Model to allow objects to be migrated and replicated transparently. Globe uses a distributed search tree to locate objects. The search tree's layout should ideally match the topology of the underlying network. Because the names of Globe objects are pure, they do not contain information regarding the path to the object within the tree. Instead, when an object is registered, a record is created for it at its local Globe node, and at all ancestor nodes. Each ancestor keeps a record for the object indicating to which of its children queries for the object should be forwarded.

Although a robust and reasonably efficient lookup scheme, the Globe location service suffers from the need for increasing storage space in nodes nearer the root of the tree. The root node, for instance, must store a forwarding pointer for every object in existence. While the service's tree structure ensures that this is not normally a performance bottleneck, it reduces the practical scalability of the system.

Mobile Addressing

The Internet Protocol Version 4 locates devices using a hierarchically assigned address, describing the physical location of the device, relative to the underlying network topology. Most devices sharing a common address prefix must be located within a particular *subnet* of the network if routing is to be performed efficiently. The IPv4 Routing specification [4] describes exactly how addresses are used to determine the next hop at each stage of a packet's journey. IPv4 devices can typically move within their own subnet without changing their address.

The Mobile-IP [60] extension to the IPv4 specification allows standard IPv4 devices to move between unrelated subnets without changing address, thus providing full mobility. Each mobile device has a *home location* and a *transient location*. The home location is a static point in the network, to which all packets destined for the mobile device must be sent. The transient location is the current location of the device, to which packets will be forwarded by its home location. This arrangement requires that each device remain in contact with its home at all times, and is hence more susceptible to network outages and latency problems. It is particularly inefficient when data is to be exchanged between two nearby devices via a home location that is a great distance away, which leads to the pathological "triangle routing" case.

The Internet Protocol Version 6, at the time of writing, uses network addresses with two distinct parts. The first part, like IPv4, is a hierarchical address specifying the physical location of the device, while the second is a unique value assigned to the device. It is intended that the final incarnation of the protocol should include a more scalable and robust mobile routing specification than that available for IPv4. The Predator location service, described in this chapter, could provide an alternative implementation for mobile IPv6 routing, by using the hierarchical portion of a device's name as a hierarchical Predator name.

Impure Naming

The Domain Name System (DNS) [42] specification allows hierarchical but logically assigned names to be resolved to addresses. The DNS hierarchy need not reflect the underlying network topology in any way, although in practice there is often some correlation. DNS names are impure in that each part of the name is used by clients to locate the DNS server capable of resolving the next part. Resolving a DNS name therefore involves a chain of lookups, following a path through the network.

In spite of this separation, DNS is not suitable for tracking mobile devices because it uses caching schemes optimised for static networks, and because it suffers from similar performance issues to Mobile-IP when a target device is far away from its name server.

Hybrid Addressing/Naming

The ALICE [10] project takes a similar view to Mobile-IP, assigning a home location to CORBA objects as part of their name. The home location, however, is itself mobile. Whenever an object's name is returned to a client, it is transparently "swizzled" to reflect the object's present location. If a client attempts to refer to the object by an older, no longer valid name then the old home location forwards the client to the new home.

In practice this system is extremely efficient even for highly mobile objects. Provided that clients keep themselves reasonably up to date, the system performs better and is more robust to certain kinds of network failure than statically-homed schemes. With static schemes, an object which has migrated close to one of its clients may not be locatable, if the network connection to its home location is lost. Because ALICE objects' homes follow them, this is less of an issue. Problems arise only when extremely out of date clients attempt to access ALICE objects, since it is not clear for how long forwarding locations should be retained. Equally, if an object migrates through several homes, one of which fails, then the chain is broken and clients retaining names referring to older homes will fail to locate the object. The more quickly an ALICE node moves, the longer the lookup chain will become[1], and the greater the likelihood of such a failure occurring.

6.3 The Predator Location Service

The Predator Location Service, developed as part of the Eucalyptus project, uses a similar distributed structure to the Globe Location Service, but uses hierarchically structured impure names to aid scalability. Predator aims to be both scalable and robust in the presence of network failures.

While the Eucalyptus architecture defines a trader mechanism through which computational objects hosted by other appliances are discovered, the Predator ser-

[1]This assumes that nodes do not double back on themselves and revisit old home locations.

vice is instead used to locate entities of which the client is already aware as they move around. In the Eucalyptus system, Predator is used to track appliances rather than to discover the objects they host.

The service runs on all devices wishing to act as contact-points for mobile objects, or hosting clients of such objects. Predator nodes are arranged in a hierarchical tree structure roughly matching the layout of the underlying network where possible. This tree represents the *root domain*, within which similar, hierarchically structured trees representing *sub-domains* reside. The layered arrangement and use of hierarchical names orthogonal to the layout of the underlying network layout is similar to the approach taken by the ODP Trader specification. The Predator service could feasibly be used to support an alternative implementation of the ODP Trader interface.

Unlike the root domain, sub-domains are themselves mobile. Sub-domains are effectively *distributed objects* since all nodes provide the same view of them. Each node is therefore an *implementation object*. As nodes are added and removed from a sub-domain, the domain is said to *expand* or *contract*. Note that the term "distributed object" in this context refers to an object whose implementation is distributed amongst several hosts, in contrast to CORBA's "distributed objects", which are objects implemented in one place and made available to distributed clients.

Unless otherwise specified in the following discussion, the term *object* will be used to refer both to the *mobile objects* we wish to locate and to the *nodes* comprising sub-domains.

6.3.1 Search Tree Structure

Each node in a Predator domain tree maintains a list of ancestor nodes, ordered to range from its most immediate parent to the domain's root node. Queries for objects which a node has no record of are passed to the node's most immediate ancestor.

Nodes in the base tree are ideally grouped according to network locality, but may be further partitioned to match physical locality where appropriate, to aid performance. For example, all the Location service nodes running on the PCs in a room may be grouped under a single parent node which represents that room. The parent nodes for each room will then in turn be grouped under a node responsible for the building containing those rooms and so on.

Figure 6.1 shows a simple search tree structure that might be used to handle mobile devices in a small building.

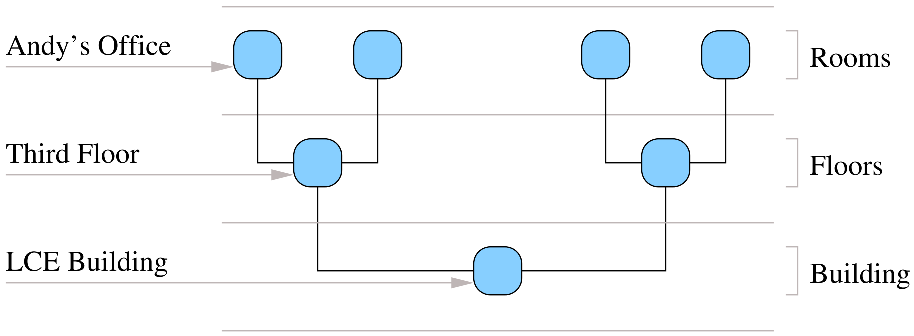


Figure 6.1: Hierarchical tree structure for a small building

6.3.2 Names

Mobile objects are assigned hierarchical, DNS-style names through which they may be located by Predator. While existing DNS names are partially location-based and partially organisational, Predator names should normally be organisationally and semantically-assigned. Devices carried by a user belonging to a particular company might, for instance, have names composed of the company's name, the user's name, and the device's function.

For example, there may be sub-domains of the root domain named "Paging" and "Email". If an object belonging to one of these sub-domains is registered at a particular location, then a tree node implementing the domain must also be registered there. Sub-domain nodes may also exist at points in the network where there are no corresponding objects, in order to improve the performance of location queries.

Figure 6.2 shows example physical and logical layouts of part of a backbone network, hosting several mobile objects. Figure 6.3 shows the structure of the

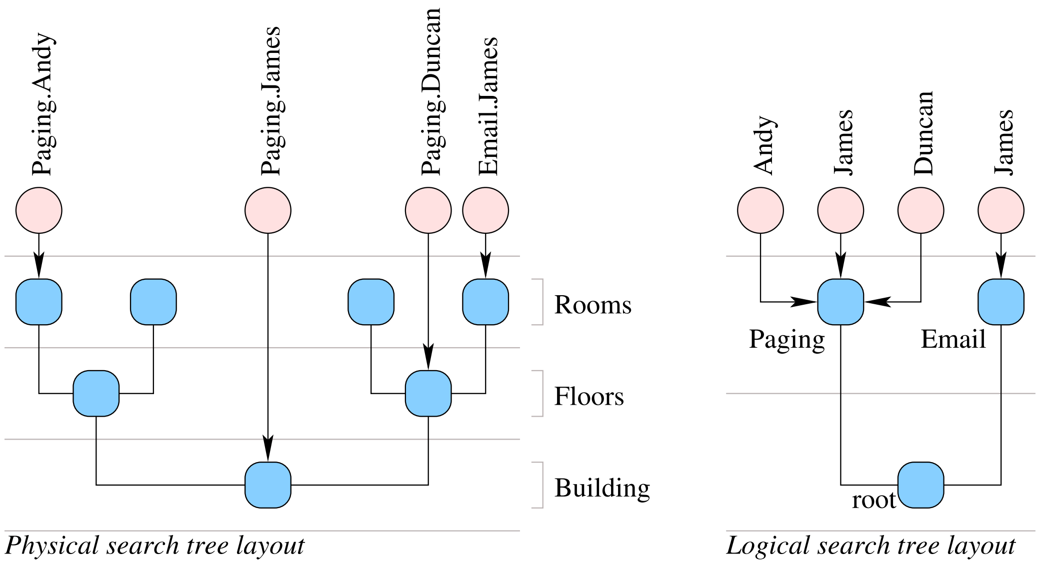


Figure 6.2: Physical and Logical layouts of an example network

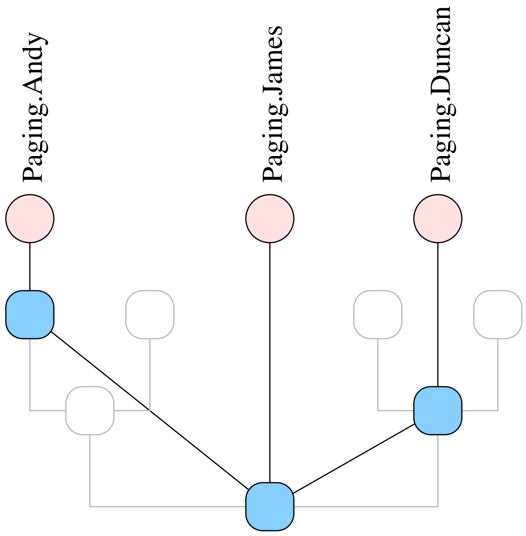


Figure 6.3: Layout of the Paging domain search tree

search tree corresponding to the "Paging" domain, overlaid on the physical network layout. The "Paging" and "Email" domains could span multiple buildings and organisations, in addition to those shown. Further subdomains would then be used to improve scalability. The full name of a pager appliance at AT&T Labs Cambridge might be "Paging.AT&T.Research.UK.DuncanGrisby", for example.

The "Paging" domain can be considered as a distributed object, since it is made up of a number of scattered nodes working to provide a single, unified function—that of locating objects used for the task of "Paging". The "Email" domain can be considered, similarly, as a distributed object, even though it has only one implementation object. The root domain is special in that it exists everywhere and its structure reflects the layout of the underlying wired network.

6.3.3 Failure Recovery

The Predator service is designed to provide a reasonable level of service even in the presence of network failures. By reasonable, we mean that clients querying a particular node should be able to locate any object registered with another contactable node. Equally, we cannot expect to be able to locate objects residing in parts of a network which have become inaccessible. Standard DNS implementations, for instance, would not qualify as providing a reasonably robust service, since the failure of a DNS server can prevent clients from resolving each other's names, even though they are still contactable.

The following sections describe briefly the mechanisms used to recover from failures of Predator service nodes managing the root domain. Nodes belonging to subdomains handle failures by searching their parent domain for other nodes of the same subdomain, in order to restore the subdomain's search tree.

Ancestry

Each node in the root domain tree maintains a list of its ancestors within the physical search tree. In the case of failure of the parent node, but continued network connectivity, this allows the child to temporarily re-parent itself to the next accessible ancestor. The child will retain sufficient information to return to the missing parent if and when it recovers. Recovery by reversion to ancestry is used only as a last resort, since it causes a flattening of the search tree.

Sibling Promotion

In addition to its ancestry, a node will additionally maintain a list of its siblings. Because of Predator's tree structure, this list will usually contain no more than a few hundred entries. The list is updated by the parent as new siblings are added and old ones removed, and includes supplemental information such as the amount of memory available to each Predator node, the speed of its network connection, etc. When a parent node fails, its children apply heuristic tree rewrite rules to the available sibling information and nominate a surrogate parent from among their ranks, to replace the failed parent.

6.3.4 Locating a Mobile Object

When a client wishes to locate an object using Predator, it passes the object's hierarchical name to the root domain node on the local machine. Devices incapable of supporting Predator nodes may instead contact another device to locate a root domain node. The result of a query is a reference to the object at its current location.

The root domain node will attempt to resolve the first part of the name (if the desired name was "Paging.Andy", for example, the first part would be "Paging") locally. If the name does not exist locally then the node's parent is queried, possibly causing the request to propagate to more distant ancestors. If the query propagates to the root of the tree without being satisfied then the client will receive an exception indicating that no object exists with the desired name.

A successful query on the first part of the name will provide a reference to a node of the names domain's search tree. The node may be anywhere in the world but, as a consequence of Predator's tree structure, will be the nearest available part of the specified domain, in terms of network structure. The root domain node satisfying this portion of the lookup will then contact the located sub-domain node, and request that it resolve the remainder of the name.

The request will propagate through sub-domain until the entire name has been resolved, at which point the location of the target object will be returned to the caller.

Figure 6.4 shows an example search in which client object (1) attempts to

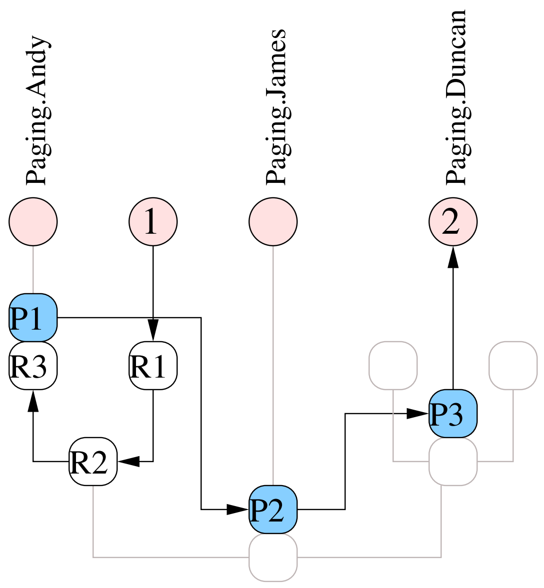


Figure 6.4: An example search of a layered search tree

locate mobile object "Paging.Duncan". The "Paging" domain is implemented as a tree of co-operating nodes—those labelled *P1*, *P2* and *P3* in the diagram. Once a client can see *any* node which is part of the "Paging" domain, it can then locate *any* object that belongs to that domain, simply by searching the domain's tree in the same way as it would search the global tree.

1. The client (1) asks the Location service at node *R1* to find the object "Paging.Duncan" (2).

2. Node *R1* first tries to locate the nearest node in the "Paging" domain.

   - *R1* cannot see any nodes in the "Paging" domain locally, so it falls back to its parent, *R2*.
   - *R2* has been told by *R3* that it knows of a node in the "Paging" domain, so *R2* forwards the request to *R3*.
   - *R3* can now return a reference to *P1*, direct to node *R1*. Note that *P1* is not the base of the "Paging" domain's tree but the node in that tree that is closest to *R1*. This helps us avoid long-distance searches throughout a tree when the desired mobile object is actually local to us.

3. Node *R1* now asks *P1* to find the object called “Duncan” in the “Paging” domain. A similar search is then performed of the “Paging” domain to that performed on the global domain, returning *P3* as the current location of the object called “Duncan”.

4. The resulting object reference is returned to the client and may be used until it fails for some reason, at which point the client can fall back to another search for the name “Paging.Duncan”. Caching schemes can help reduce the search path on subsequent attempts to relocate an object that has moved, by storing the address of the Predator node that satisfied the location query, which is more likely to be close to the object’s new location.

Registering a Mobile Object

Registering an object with Predator under a particular name follows a similar pattern to locating an object. The main difference is that instances of the required sub-domains will be created at the Predator service host local to the client, if they are not already present.

For example, consider the case in which a new object is to be registered under the name “Email.Andy”, as in Figure 6.5a;

1. The new object (1) calls down to its local Location service node and asks the Location service to register it as “Email.Andy”.

2. The Location service performs a normal search, to locate the nearest node in the “Email” domain and finds node *E1*, as shown in Figure 6.5b.

3. If the nearest node is not located at the client’s local Predator service instance then a new “Email” domain node is created locally and configured to use the existing node as its parent.

4. If there are no existing “Email” domain nodes then one is created locally.

5. The object is now registered with the local “Email” domain node, *E2*, under the name “Andy”, as shown in Figure 6.5c.

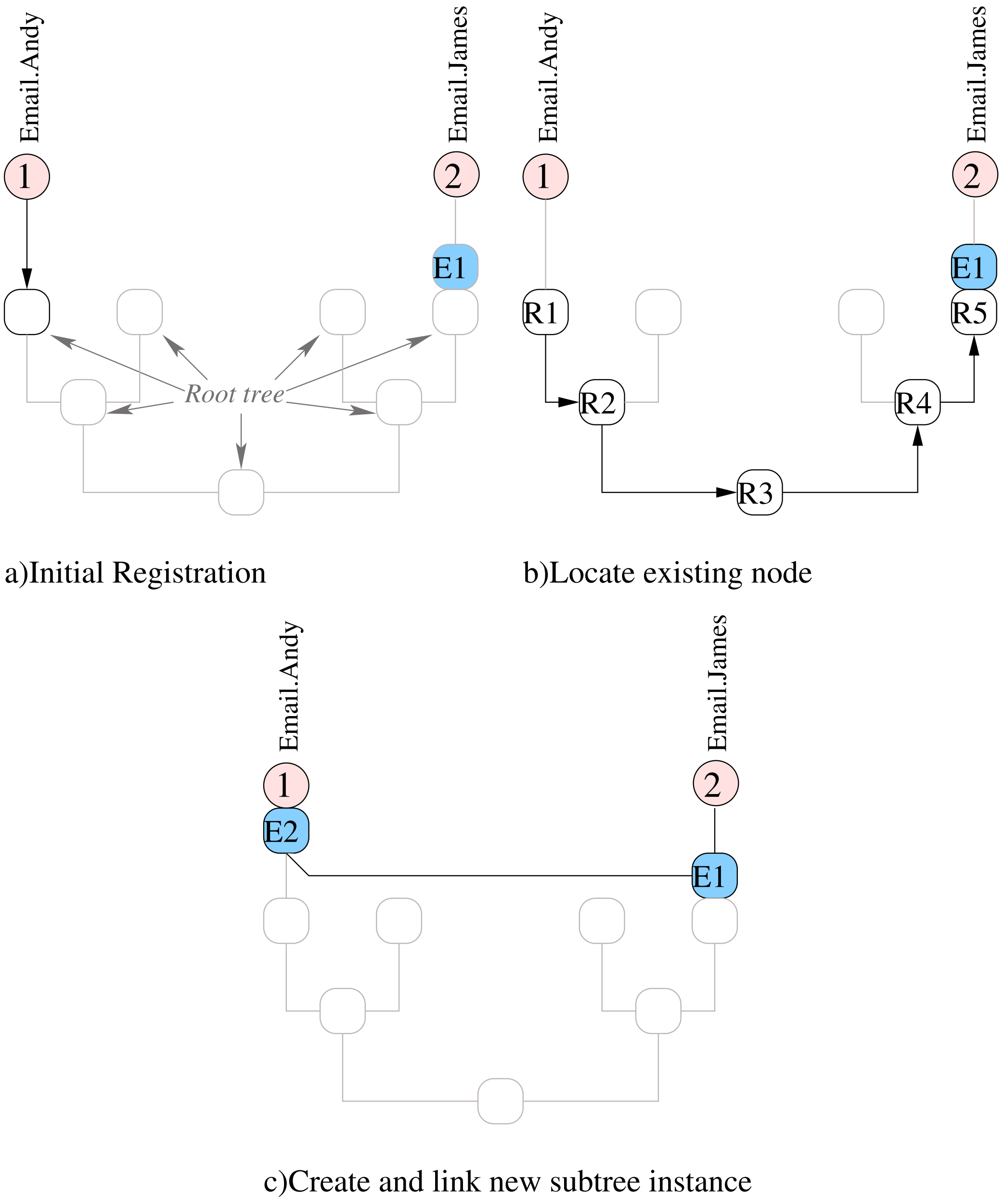


Figure 6.5: An example registration of a mobile object

6.3.5 Discussion

The Predator Location Service has been implemented and deployed over a small local-area network. A wide-area deployment would be required to verify the performance and reliability properties of the system in practice. To motivate widespread use of Predator for testing services, an HTTP proxy service has been designed based upon it. This could be made publicly available in the future to obtain practical results. Predator's design attempts to distribute location information in such a way that search operations for nearby objects can be satisfied locally. Although this results in a performance degradation for individual queries on average, when compared to centralised systems, it allows Predator to be more scalable and robust to network or host failures.

Wide-area routing and service provision mechanisms have been designed, based upon Predator. Predator is used to track mobile objects representing individual PEN nodes, through which the nodes themselves may be contacted. This allows a PEN node to be contacted from any location in the wired network, provided that it is near to a gateway between the PEN and wired networks.

6.4 Routing Through Wired Networks

This section describes the R-Link service [76], a mobile wide-area messaging service for PEN, based upon the Predator location service. The communications primitives provided by the R2-Layer are designed to be used by applications residing on low power devices to cooperate with software hosted by a small number of nearby devices, via a shared-medium network. The characteristics of this environment differ considerably from those of switched media networks such as the Internet, making it infeasible to provide semantically identical primitives over such networks.

The R-Link service aims to provide equivalent primitives to the R2-Layer. Rather than attempt to mimic their local-area behaviour exactly, the R-Link service makes assumptions about their logical behaviour and intended use. In addition to supporting both transmitter- and receiver- managed messaging, the R2-Layer provides several distinct addressing modes, suitable for different tasks. Each addressing mode is dealt with individually by the R-Link service, so as to provide the most suitable mapping between local- and wide- area behaviours.

The following section describes the addressing modes supported by the R2-Layer, and is followed by discussion of the wide-area equivalents provided by the

R-Link service.

6.4.1 R2-Layer Addressing Modes

The R2-Layer allows co-located appliances to communicate using transmitter- and receiver-managed primitives. Low level modules implementing these primitives allow data to be exchanged between nodes identified by their MAC address. Higher level equivalents of these primitives are also provided which allow source and destination application identifiers (APID) to be specified in addition to MAC addresses. This allows multiple applications hosted by a single appliance to be contacted independently. A set of well-known APIDs are also used to refer to common PEN network services.

The provision of APIDs allows the addressing modes available to applications to be extended to support different modes of communication. While datagrams may be delivered directly to a particular node and APID, for example, they may also be delivered to any nearby node advertising a particular APID. The transmitter- and receiver-managed addressing modes are described in the following sections.

- `TX_SPECIFIC(address, apid)`

  Transmit data to the node with the specified MAC address, if it is advertising the specified APID.

  Destination-specific addressing is typically used to access previously discovered services hosted by particular devices. Despite being *unicast*, such messages may in practice be received by other local devices. This allows the unintended recipients to optimise the transmission of their own messages, and to infer details about the communicating parties. Conversely, plaintext unicast datagrams may be intercepted and read by third parties, motivating the use of suitable encryption schemes.

- `TX_ANY(apid)`

  Transmit data to *any* node advertising the specified APID.

This form of addressing is typically used to discover and solicit services from nearby devices. The transmitting device broadcasts a plea for the required APID and listens for a short period for corresponding offers of service or broadcasts advertising the APID. The datagram is then delivered to the sender of the first offer or beacon which matches the requested APID. The client device may power down as soon as the message is delivered, since only the first server to respond should receive the datagram.

- `TX_ALL(apid)`

  Transmit data to *all* nodes advertising the specified APID.

  Like the `TX_ANY` primitive, this mode causes data to be transmitted to devices which provide a given APID. Unlike `TX_ANY`, the sender remains powered after transmitting the datagram to the first suitable recipient, and continues to deliver it to all recipients which respond within a particular time interval. This is typically used to maintain consistency between local instances of a particular service, by providing a crude high-level broadcast facility.

- `RX_SPECIFIC(address, apid)`

  Synchronise with the *target* device's beacons and receive all data they contain that is associated with the specified APID.

  Targeted reception of data embedded in beacon datagrams is generally used to monitor specific resources such as individual sensors or switches. When the device is no longer contactable, the R2-Layer will inform the calling application. Some applications will use this notification to obtain context information and will cancel the monitoring operation once the target becomes unavailable. If the target becomes contactable again, then the R2-Layer will inform the application and continue monitoring.

- `RX_ANY(apid)`

  Synchronise with any device providing the specified APID and fetch the data conveyed in its beacons.

RX_ANY provides PEN devices with a means of monitoring data which should be consistent amongst all instances of a service, such as GMT or the regional time. Additionally, it may be used to fetch local information, such as the name of the user's current location or its GPS coordinates, from nearby nodes. The R2-Layer will automatically locate and monitor a new service instance, if one is available locally, when the old one is no longer contactable.

- `RX_ALL(apid)`

  Synchronise with all devices providing the specified APID and fetch the data contained in their beacons.

  As for the `TX_ALL` primitive, the R2-Layer attempts to reduce the impact of this primitive upon power saving through the use of timing information stored in a local proximity-store.

The R-Link service provides equivalent primitives to those of the R2-Layer itself, adapted to better suit wide-area networks. Implementation details of the R-Link primitives are described in the following sections, to illustrate that while their semantics may differ from those of the R2-Layer's primitives, their behaviour will more closely match that intended, as a result.

6.4.2 TX_SPECIFIC

The R2-Layer `TX_SPECIFIC` primitive delivers datagrams to a specified host and APID. We assume that all such datagrams should be delivered to their destination if possible, since directed transmissions are typically used to communicate with previously-discovered services.

Wide-area `TX_SPECIFIC` requires that both the sender and the receiver use replacements for the standard R2-Layer `R2SEND` and `R2RECV` modules, which are responsible for sending and receiving `TX_SPECIFIC` datagrams, respectively. R-Link provides supplementary implementations of these modules, which build upon them and provide the same interfaces to applications. Programs which use `TX_SPECIFIC` messaging need not therefore be altered in order to use the R-Link service.

Sender Operation

The sending application passes outgoing datagrams to the R-Link Sender module, named `R2SEND-R`, where they are processed to establish whether or not the target device is available locally. In order to achieve this, the device must host a *proximity store*, responsible for monitoring the local environment and maintaining a table of other local nodes.

If the target device has an entry in the proximity store to indicate that it is local to the sender then the datagram is passed to the standard `R2SEND` module for delivery. If local delivery fails then the proximity store is updated and the transmission request is re-processed by the `R2SEND-R` module.

If the destination device is not marked as local by the proximity store, the `R2SEND-R` module instead attempts to locate an instance of the well-known R-Link Gateway APID. This is performed using a `TX_ANY` request via the `R2SEND` module. Extra fields are used in the request message to provide the required addressing information, as shown in Figure 6.6.

If an R-Link Gateway is available locally then it will acknowledge delivery of the datagram. The R-Link Sender module keeps a record of the datagram transmission and returns a result to the caller only when delivery confirmation is received from an R-Link Gateway, or when a timeout specified by the gateway upon receipt of the datagram has expired.

Figures 6.7 and 6.8 illustrate the operation of the `R2SEND-R` module for local and remote transmission.

Receiver Operation

Datagrams routed through a wired network are passed to the R-Link Receiver module (`R2RECV-R`) instance residing on the target device. Unlike its counterpart `R2SEND-R` module, the `R2RECV-R` module is not used in place of the standard receiver module. Instead, it registers itself with the standard `R2RECV` module as the owner of a well-known R-Link Receiver APID, in order to receive incoming R-Link datagrams.

Datagrams received via wired networks are tagged with a bit in their flags field to indicate to clients that they originate from a remote host. Wide-area routed

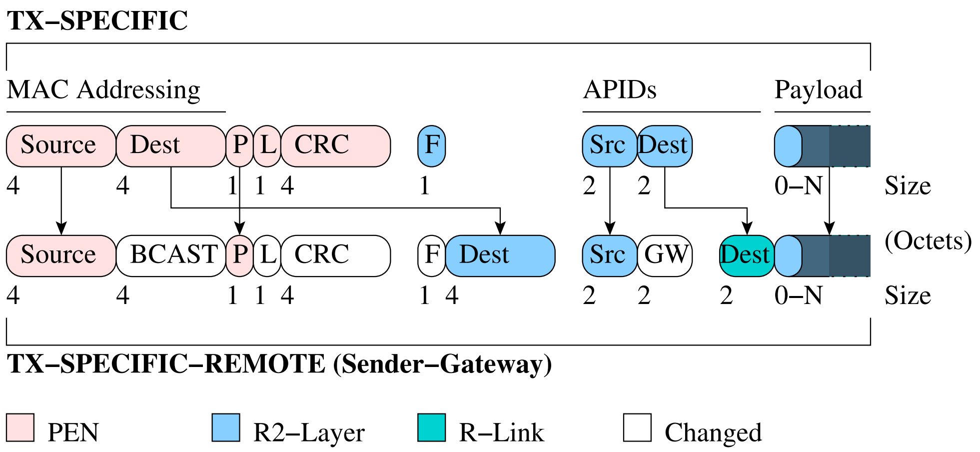


Figure 6.6: TX_SPECIFIC Sender-to-Gateway datagram structure

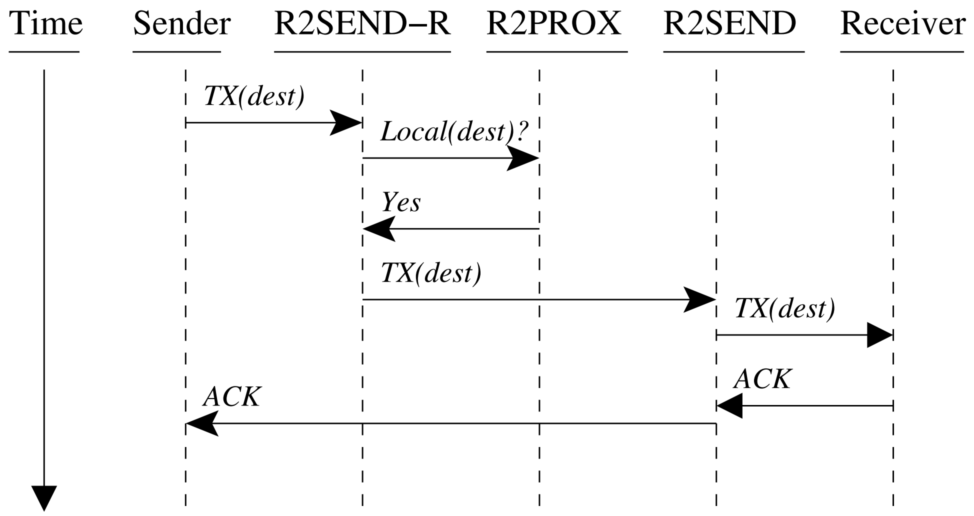


Figure 6.7: Local transmission using the R-Link Sender (R2SEND-R) module

datagrams include the R2-Layer multi-hop source address field, to indicate the original sender. The presence of this field could be used as an implicit indicator that the datagram was not delivered directly by the sender. In some situations, datagrams are routed via powered gateways even when the destination device is

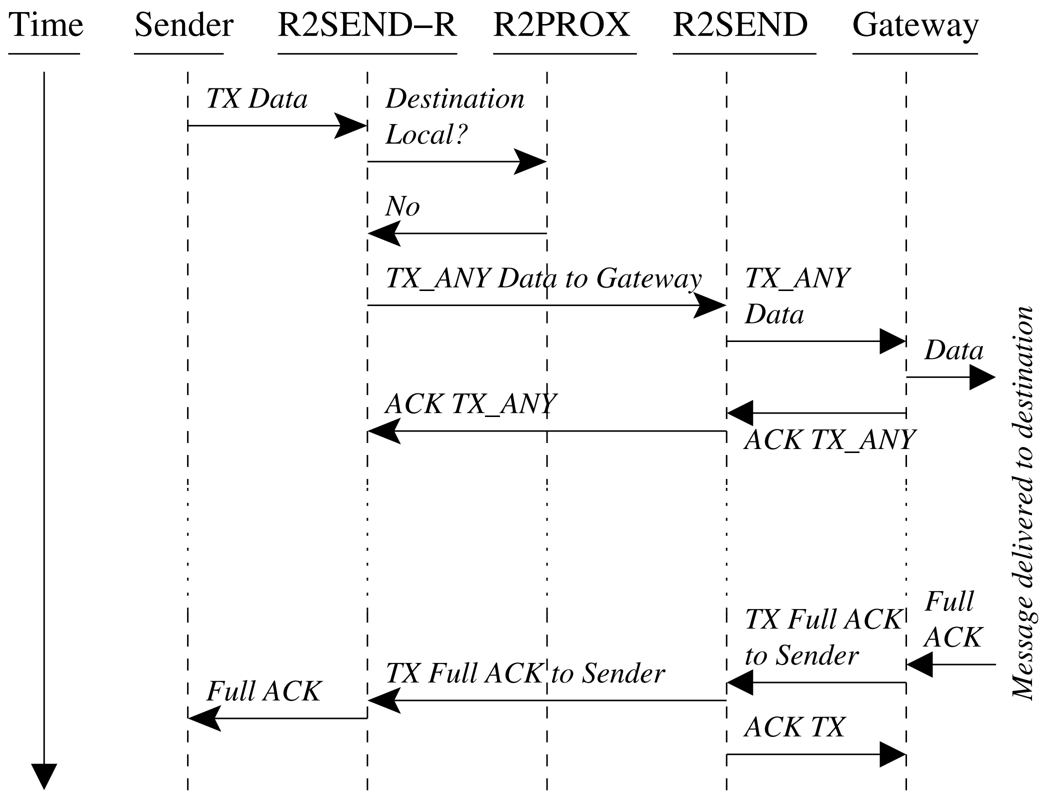


Figure 6.8: Remote transmission using the R-Link Sender (R2SEND-R) module

contactable directly by the sender, to allow the sender to save power. In this case, the extra routing fields would be present, in spite of sender and receiver being co-located. Figure 6.9 shows the datagram structure used by gateways to deliver routed data to recipients.

Gateway Operation

R-Link Gateways advertise the well-known R-Link Gateway APID to nearby clients using the `R2RECV` module. R-Link Clients will transmit datagrams to this APID when they wish them to be delivered via the wired network. Such datagrams must include the optional *destination* MAC address field supported by the R2-Layer's basic packet format, since the destination address provided in the lower-level PEN headers will contain the gateway's own address. This requires that the datagram's payload does not exceed the Maximum Transmission Unit

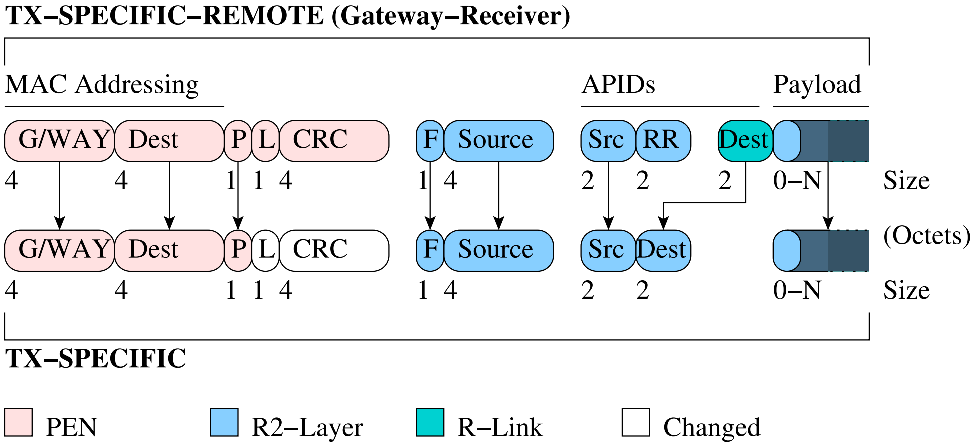


Figure 6.9: TX_SPECIFIC Gateway-to-Receiver datagram structure

(MTU) after the supplemental addressing fields have been added.

The gateway service uses the supplied destination address to query the Predator location service for a gateway with access to the target device. The datagram is then passed to this gateway to be delivered to the device. If the device has moved beyond range by the time the datagram arrives at the destination gateway then the process is repeated, using its last known location as the starting point. A second attempt will typically succeed, since it is unlikely that a PEN device will have moved a significant distance from the original gateway.

To maintain up-to-date location information, R-Link Gateways listen for all incoming datagrams from nearby devices, regardless of their intended destination, via the low-level `R2RX` interface. This poses no problem for gateway devices, since they will usually be fully powered and must listen for incoming transmission requests in any case. Figure 6.10 shows the R2-Layer configuration of a gateway device.

In addition to providing basic routing functionality, gateways may also offer proximity-store features to local devices. Using the `R2SNOOP-R` extension module, mobile devices may embed supplemental data in their beacons to request notifications of proximity-store changes from local gateways. The `R2SNOOP-R` module accepts incoming notifications from gateways and uses these to update the

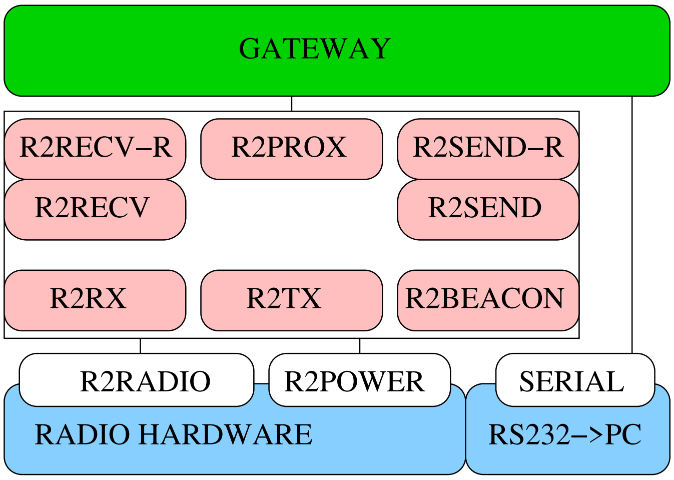


Figure 6.10: Structure of an R-Link Gateway

node's local proximity store. The standard `R2SNOOP` module can be temporarily disabled whenever a gateway is providing proximity information, allowing low power devices to further reduce their power consumption.

Nodes running a full proximity store implementation may reduce the time spent active in rendezvous with target nodes using timing information stored by the proximity store. R-Link aware nodes with only rudimentary proximity service support may obtain similar power saving benefits by routing data via nearby gateways even when co-located with the target device. Although this incurs some latency as a result of the extra hop and doubles the required bandwidth, it allows such devices to offload the task of rendezvous to a better powered node.

6.4.3 RX_SPECIFIC

The R2-Layer implementation of this primitive causes the recipient to synchronise with the transmitter's beacon datagrams to fetch their contents. This arrangement fails when used with a switched network. It is, however, reasonable to assume that whenever a recipient requests that another node be monitored, that this request should be satisfied regardless of the location of the target device. The R-Link implementation of the `RX_SPECIFIC` primitive therefore explicitly requests that the backbone network listen for beacon datagrams from the sender, via a local R-Link gateway.

Receiver Operation

`RX_SPECIFIC` receivers must include a replacement reception module. The `R2WATCH-R` module wraps the standard R2-Layer `R2WATCH` module, providing the same interface to clients. A proximity store implementation must also be run on receiving devices, to maintain locality information.

If a request is received to watch a device contactable locally, then it is passed to the standard `R2WATCH` module to be satisfied. If the target device is non-local then the `R2WATCH-R` module instead sends a `TX_ANY` request to the well-known R-Link Gateway APID for the wired network to monitor the target device on its behalf. Figure 6.11 shows the mapping of `RX_SPECIFIC` requests to R-Link requests.

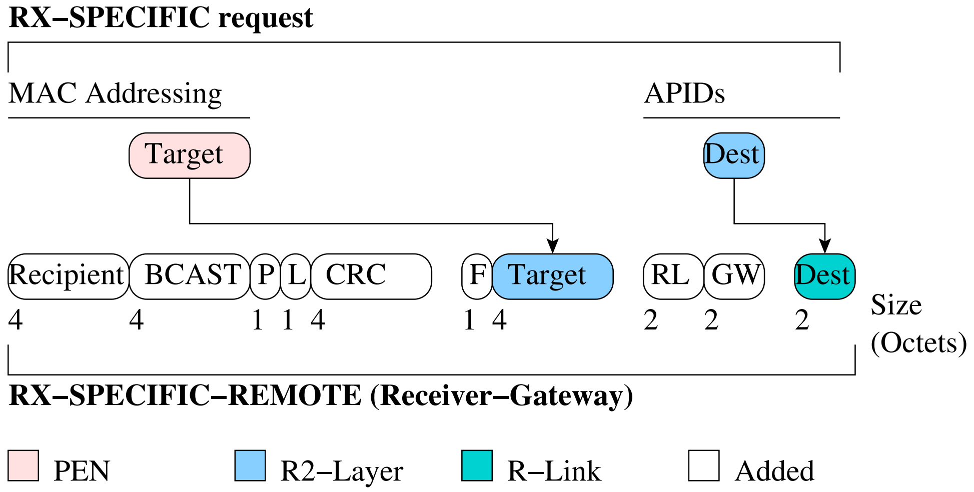


Figure 6.11: RX_SPECIFIC Receiver-to-Gateway datagram structure

Since monitoring operations are long-lived, it is highly likely that the target device will move into or out of direct contact with the recipient. The `R2WATCH-R` module deals with this transparently, dispatching new requests to nearby gateways if the target moves away, or cancelling existing requests if it becomes co-located.

Gateways will cancel requests if their originators become unavailable or stop accepting updates.

Sender Operation

Unlike their `TX_SPECIFIC` counterparts, devices providing data in their beacon datagrams may be monitored by remote recipients without modification. This is ideal for the tiny sensor devices at which receiver-managed communication is mainly targeted.

Gateway Operation

When an `RX_SPECIFIC` request is received by a gateway, it first checks to discover whether the device to be watched is available locally. If the target is local then the address of the request's originator is added to a table of interested parties. The gateway stores one such table for each nearby mobile node. If the target is non-local then the request is forwarded to a more appropriate gateway using Predator.

Whenever a beacon datagram containing a payload is received by a gateway, the originator's table of interested parties is checked. The payload is then delivered using the `TX_SPECIFIC` mechanism to each registered recipient. Because the wide area `TX_SPECIFIC` primitive is used to deliver beacon payloads, recipients may roam between different gateways without needing to re-register their interest in remote nodes.

Watch registrations always migrate towards the gateway nearest the target node. This is implemented by having the previous gateway forward registration information to the new gateway when it discovers the node is no longer local. It is possible using this technique that some beacon payload data may be missed, since the registration information will lag behind the node. An alternative approach is for the new gateway to query the old one whenever a new node arrives, to obtain the registration information. Unfortunately, this imposes an overhead even when the arriving node is not being watched. Although the latter technique may be optimised, for example by requesting registration information only if a new device's beacon contains a payload, the former scheme has been implemented for

R-Link. It is assumed that higher level protocols will deal with the occasional loss of beacon payloads.

Figure 6.12 shows the structure of a target's beacon datagram, and the resulting `TX_SPECIFIC` datagram delivered to the recipient.

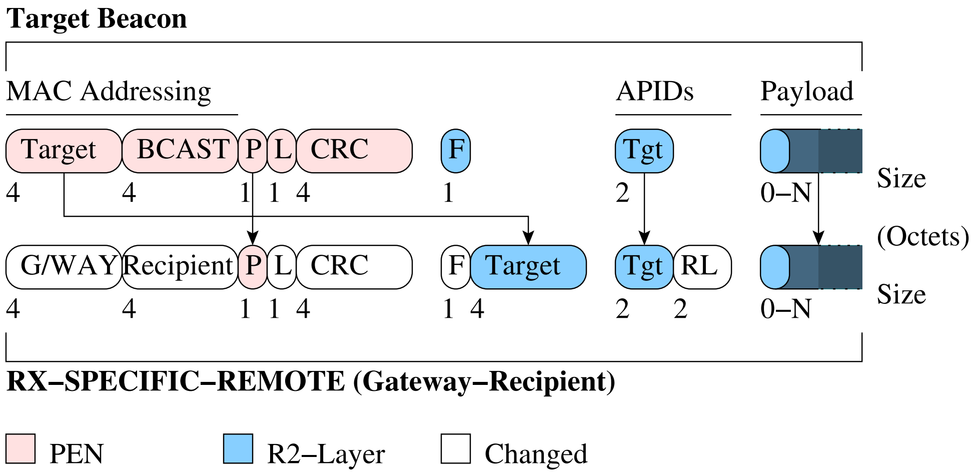


Figure 6.12: RX_SPECIFIC Gateway-to-Recipient datagram structure

6.4.4 TX_ANY and TX_ALL

The R2-Layer `TX_ANY` primitive allows devices to locate instances of required services. This functionality could be supported by wide-area routing services in much the same way as `TX_SPECIFIC`. An assumption implicit in the use of `TX_ANY` by many applications is that only local devices that provide the required service will be contacted. This forms the basis for the design of context-aware PEN applications, which use `TX_ANY` to discover nearby service instances, then use `TX_SPECIFIC` to exchange data with them. It is therefore undesirable that such datagrams be delivered to remote devices.

Better constrained and more generally useful semantics are obtained by assuming that `TX_ANY` datagrams should be delivered only to services hosted in the wired network, rather than to other mobile PEN devices which support them. Services accessed in this way can be pre-configured to ensure that they are used only by client nodes within a certain context. Clients can then safely assume that the service instance which replies is always a suitable one, avoiding the need to check the originator of the reply or to reissue the request.

For example, UTC (Universal Coordinated Time) services normally provided by local low power UTC receivers may be replaced by centralised NTP (Network Time Protocol) based services hosted in an organisation's wired network. This replacement would be transparent to clients.

Client Operation

In contrast to datagrams delivered using `TX_SPECIFIC` semantics, `TX_ANY` datagrams may be delivered to services hosted in wired networks transparently by gateways, without the need for client-side support. This is possible because `TX_ANY` datagrams may be delivered by the R2-Layer to any device providing the required service. The extra routing fields provided by the R-Link datagram headers are therefore unnecessary, since it is perfectly acceptable for clients to believe that a service is provided by the gateway device itself.

This is ideal, since it allows arbitrary services to be hosted by wired networks. A wireless Internet gateway service might be provided, for example, by a mobile phone. The same service may also be provided by an organisation's wired network. Since all instances of the service are effectively interchangeable, the distinction is irrelevant to clients.

Server Operation

Services reside in a wired network and are accessed via R-Link Gateway service nodes. Services implement a simple message delivery interface, through which requests are passed to them. Each service registers at least one object supporting this interface with the Predator location service.

In addition to receiving requests, services must be able to dispatch replies and other datagrams to clients. This is achieved in one of two ways, described below.

- *Globally-addressed*, using a dedicated service MAC address.

  This technique allows the service instance to reply to the client using its own unique MAC address. The service locates the gateway nearest the client device in the same way as if routing a `TX_SPECIFIC` datagram. It then passes a datagram with its own MAC address as the source, to be delivered in the normal way.

Subsequent transmissions by the client may be addressed directly to the same service instance, which must register its own `TX_SPECIFIC` gateway object with the location service. Long-lived transactions are well served by this approach, since per-client state maintained by the service instance need not be replicated to service instances nearer the client each time it moves.

- *Locally-addressed*, using the MAC address of the gateway nearest the client.

  Again, the service must first locate the gateway nearest the client, before passing a datagram to be delivered. Unlike `TX_SPECIFIC` datagram delivery, the resulting transmission will be tagged with the MAC address of the gateway.

  The service itself need not have a MAC address of its own. Datagrams may also be delivered to nodes which do not operate R-Link service components, since no extra addressing information is required, and so the datagram need not be encapsulated by R-Link. This allows the service to be used even by devices which are not explicitly aware of the wide-area routing services.

  If a client moves between gateways then datagrams delivered in this way will appear to originate from the currently local gateway, rather than from the one to which the a request was originally sent. This makes locally-addressed transmission unsuitable for long-term transaction communications. If the client ignores the MAC address of the reply's originator then it will not be aware of the distinction between the two methods.

Gateway Operation

Low power devices capable of providing services to nearby clients advertise this by embedding the APIDs of the supported services into their regular beacon datagrams. This approach is unsuitable for use by R-Link Gateway nodes, since it does not scale well.

Firstly, beacon datagrams may only carry relatively small payloads, limiting the number of APIDs which may be advertised in a single beacon. If services are being provided by an extremely wide-area network such as the Internet or even a large corporate intranet then the number of available services may quickly exceed the number that may be advertised in a single beacon.

Secondly, advertisement of services in beacons requires that each gateway is explicitly aware of every service hosted by the wired network. Again, this does not scale well when applied to extremely large networks, with many gateways and hosted services.

Instead, gateways take advantage of the R-Layer and R2-Layer optimisations for well-powered devices. Transmitter-managed devices wishing to send data to a particular node or service first transmit a *plea* datagram. If the required node, or a node providing the required service, is nearby then it will respond with a corresponding *offer* of service. Upon receipt of an offer packet, the sender is able to transmit the datagram. If no offer is received then the sender will continue to listen for beacons from the target device, or which advertise the target service.

R-Link Gateway nodes may therefore capture incoming plea datagrams, via the low-level `R2RX` module, in order to discover when local devices are searching for services. Whenever a plea is received, the gateway first checks the list of APIDs it currently advertises.

If the requested APID is currently advertised then it is marked as having been used recently. If the requested APID is not currently advertised then the gateway attempts to locate an instance of the service hosted in the wired network. If an instance is located then the APID is added to those advertised by the gateway. Addition of a new APID to beacon advertisements causes an offer packet to be broadcast by the gateway device. Since the search process takes at most a few seconds, and since the requesting device will typically need to listen for several seconds for a beacon advertising the service, it will receive the offer packet and deliver the datagram.

One useful side-effect of the transparent capture and delivery of `TX_ANY` requests is that the R-Link `TX_SPECIFIC` protocol may be implemented in the same way as any other service, in the wired network.

TX_ALL

The TX_ALL primitive is, for scalability purposes, treated in the same way as TX_ANY when propagated through a wired network. It is assumed that service instances implemented in the wired network will use more appropriate consistency protocols internally if required.

A user's PDA, for example, might attempt to distribute their business card to all local users using the TX_ALL primitive. A business-card service residing in the wired network of the building would receive their card and could then commit it to the host companies' contact database. At the same time, it could contact employee's own PDAs and update the sender's details with them, if already present. This arrangement allows TX_ALL to be used to distribute a business card to all local nodes, and to update stale information held by non-local nodes. A less intelligent wide-area implementation of TX_ALL would deliver the business card information to all PEN nodes within the office building, regardless of relevance.

6.4.5 RX_ANY and RX_ALL

The RX_ANY primitive allows applications to receive data from any local instance of a service, regardless of which device hosts it. As with the TX_ANY primitive, many applications will rely on the operation using only local instances of the requested service. Accordingly, we assume that such requests should not generally be dispatched to devices via the wired network. Instead, RX_ANY requests are applied only to services hosted in the wired network.

Receiver Operation

In order to use the RX_ANY primitive in the wide area, the recipient must operate a wrapper for the standard R2-Layer R2WATCH module. This wrapper presents the same interface to clients. When a request is made to watch a particular service, the R2WATCH-R module first attempts to locate a local instance and watch it using the R2WATCH module. If this fails then the request is dispatched to any nearby R-Link gateway using the TX_ANY mechanism.

If the service subsequently becomes available locally then any prior registrations of interest made via R-Link are cancelled. Similarly, if all local service instances become unavailable then the R2WATCH-R module will fall back to using the R-Link mechanism. As a recipient roams between gateways, it must re-register interest in services with each, to ensure that the service instance used remains suitable.

Sender Operation

RX_ANY services are hosted by backbone networks and must register themselves with Predator in order to be discovered by gateways. Services are passed registration requests by gateways, containing the recipient's addressing information and the address of the gateway which handled the request. Payloads are delivered to clients via the specified gateway. If the recipient node has roamed to a new gateway then the notification will fail and the node's registration will be cancelled.

RX_ALL

The RX_ALL primitive is treated in the same way as RX_ANY, to aid scalability. To support applications such as massively distributed sensor monitoring, proxies for the sensor service would be hosted by the wired network. These would monitor all sensors in a particular environment such as a building or room and would pass collated readings to interested parties.

6.5 R-Link Applications

Use of the R-Link service is illustrated in the following sections, with two example applications built using the service, to support office environments.

6.5.1 A Battery Monitor Service

Wired network services may be provided through R-Link to clients that operate only the standard R2-Layer protocols. To illustrate this, a battery-monitor service was constructed, hosted by a PC connected to a lab-wide Ethernet network. The

R2-Layer provided battery monitor modules which transmitted battery status notifications when the level dropped below a preset threshold. Each battery monitor registered itself with Predator, under the well-known battery notification service APID.

When R-Link gateways placed around the laboratory received beacons containing payloads for particular APIDs, they located the corresponding service and delivered the appropriate content to it. Non-beacon packets received by gateways were delivered to wired network services in the same way. This allowed the battery monitor service to accept battery status notifications delivered using transmitter and receiver managed schemes in the same way. When notifications were received, the service could look up the node's ID to find the email address or other contact information for its owner, and inform them of the problem.

6.5.2 Fat-Client Door-signs

Door-sign devices, consisting of a bi-stable cholesteric display, three buttons and a PEN node, were constructed by AT&T Laboratories Cambridge. As part of an independent project, the devices were installed with a number of in-built applications, including a personal appointment manager and Active Badge query services. The latter function was provided using a simple RPC-like protocol, communicating with a central node responsible for providing up to date Active badge information.

Using R-Link, the central node was replaced with a software service hosted within the wired network and accessed via gateways. This allowed door-signs to be deployed throughout the building, by attaching gateways to existing PCs attached to the wired network.

6.6 Summary

This chapter has discussed techniques for routing data between wireless devices through a wired backbone network, and for provision of services hosted in wired networks to roaming wireless devices. Routing through backbone networks is preferable for extremely wide-area interactions because it avoids the reliability and latency issues inherent in low power multi-hop routing. The R-Link service operates over the R2-Layer protocols described in Chapter 5 and provides equivalent primitives, removing the need for existing clients of the R2-Layer to be rewritten to use it. The Koala object request and event broker, described next, in Chapter 7, can therefore make use of R-Link routing services without modification.

The Predator location service and R-Link messaging service have both been implemented and deployed over a small local area network. R-Link service code for mobile PEN nodes has been produced, and used to support some simple applications.

Chapter 7

The Koala Object Request Broker

7.1 Introduction

Koala [77] is an Object Request Broker (ORB) inspired by the OMG CORBA [47] standard and tailored for use with low power ubiquitous computing systems. The use of an ORB as the basis for application-level communication was primarily motivated by the desire to aid interoperability between devices. Appliances carried by users will typically come from different manufacturers, making standardisation of protocols of paramount importance.

Conversely, to work well with embedded system with stringent power, storage and processing limitations, networking protocols must make efficient use of devices' capabilities, and must be easy for applications to use efficiently.

Finally, the availability of a standard programming interface avoids the need for programmers to adapt to a new, proprietary network interface every time they wish to port code to a new device. Common functionality can then be replicated more easily between devices by re-using existing code for one platform on another.

Although the CORBA specification satisfies the interoperability and ease of programming requirements, its design is not well suited to extremely high latency networks such as PEN, nor to embedded systems. The decision was therefore taken to design a new ORB, similar in concept to CORBA, but designed with embedded and low power systems in mind.

7.2 Object Model

Koala uses a similar object model to that defined by the OMG CORBA specification. Like the MICO [41] ORB, Koala may be implemented in a modular fashion, allowing both minimal code bases for simple applications and further extensions for more complex ones.

Koala objects are opaque objects accessible only via a well-defined *interface* consisting of strongly typed *methods* and *attributes*. Only attributes and methods specified in the interfaces an object supports are accessible remotely. The contact details of objects are similarly concealed, so that local and remote objects are handled identically by clients.

Methods of Koala objects are passed zero or more parameters and will either return a value or raise an exception. Attributes values are read and written, as in CORBA, by invoking `set` and `get` methods on the object. The definitions of these methods are generated automatically for each attribute by the IDL compiler. Attributes of Koala objects, however, may also be monitored by remote clients using the event interface described in section 7.8.1.

Although each object is only permitted to implement a single interface, interfaces support *multiple-inheritance* and may therefore be derived from zero or more other interfaces. Interfaces are defined using a platform-independent Interface Definition Language (IDL) compatible with that of CORBA [56]. They describe the methods and attributes that objects implementing them must provide, and that their clients may therefore assume to be available.

7.2.1 Stubs and Servants

Like CORBA objects, Koala objects are implemented by a *servant* object and are accessed transparently, whether locally or remotely, via *stub* objects. While servant objects will often contain data fields and methods used internally by their implementation, stub objects provide only those methods and attributes specified by the interface the corresponding servant implements.

In addition to any *user* defined exceptions they may raise, a Koala invocation may raise one of a set of *system* exceptions, indicating failures in the underlying transport mechanisms. Although these exceptions will only normally be raised by invocations to remote objects, clients must assume that any object may be implemented remotely and could therefore raise system exceptions.

7.3 The micro Inter ORB Protocol μIOP

The Koala ORB uses a proprietary protocol derived from CORBA's General Inter-ORB Protocol (GIOP) [54]. μIOP attempts to reduce the amortised request length as compared to that required by GIOP, as well as to provide ways to reduce the impact of high latency.

A similar effort to adapt GIOP for use with low-bandwidth environments is the LightWeight Inter-ORB Protocol (LW-IOP) [13] proposal. This proposal suggests data compression and caching techniques aimed at transparently reducing the bandwidth required by standard CORBA applications. The protocol's target environment is dial-up networking, an environment in which the primary limitation is bandwidth.

μIOP is instead tailored to the PEN environment, in which both bandwidth and latency limit performance. Many techniques aimed at reducing bandwidth, particularly those involving caching of data elements, lead to increased numbers of round-trips in maintaining shared state. Tradeoffs must therefore be made between lowering the required bandwidth and reducing the effects of high latency.

7.3.1 Transport Requirements

To provide a compromise between simplicity and performance, Koala places the following requirements upon the underlying transport layer:

- *Connection-oriented.* Capable of notifying the ORB when a connection is broken by a peer.
- *Semi-reliable.* Capable of performing packet retransmission and back-off if necessary.

μIOP does not, in particular, require that ordering of packets be preserved. This allows retransmission to be implemented more efficiently, since servants need not stall the dispatch of later requests pending the retransmission of earlier ones.

These requirements are a subset of those of the GIOP protocol. The requirements of the GIOP protocol are highly tailored to the TCP/IP-based IIOP [55] specialisation of it, which assumes a connection-oriented, reliable byte-stream transport. This is unsurprising, since these are almost exactly the properties of the TCP/IP protocol itself. The Open Network Computing (ONC) RPC [66] protocol provides a TCP/IP mapping making similar assumptions about the underlying transport, but may also operate over the unreliable UDP/IP protocol. Since ONC RPC takes no measures to provide reliability over unreliable transports, applications must take their own measures to cope with failed transmissions. Koala's transport requirements are a compromise between the requirements of performance and reliability for small devices.

7.3.2 Protocol Enhancements

Different versions of the μIOP protocol may support different message formats not supported by earlier versions. Some implementations may support both the standard core protocol and further proprietary protocols. Proprietary protocols are used to optimise performance only if both participants agree on their use. Some ORB implementations may support schemes to implement *inout* parameters more efficiently, for example, as described in Section 7.3.8, while others will not.

Extremely simple Koala devices will usually support only the core protocol, consisting simply of request, reply and event delivery message types. More advanced devices indicate support for additional functionality by sending protocol extension messages. Upon receiving a protocol extension message, the recipient may enable the specified feature if available, or ignore the message entirely.

7.3.3 Alignment

The CORBA Common Data Representation (CDR) [52] is significantly optimised towards high-speed implementations. Multibyte values such as integers and floating point numbers must be aligned within each request, to allow them to be read directly from incoming data buffers in an efficient manner. Similarly, values are never compressed in any way, since CPU overheads already dominate performance.

The μIOP protocol requires only that values be byte-aligned, and allows certain values to be compressed to further reduce message sizes. Although this increases the processing overhead in both transmission and receipt of messages, it can result in bandwidth reductions of between ten and eighty percent, depending upon messages' contents. Over high latency networks, the extra processing overhead is insignificant compared to a message's time in transit.

7.3.4 Data Representation

Bit-wise Packing

Many integer values passed to remote objects will have a limited range of valid values, usually considerably smaller than that provided by the standard integer types. If integer range annotations of the type supported by Modula-3 [46] are supplied in an interface's definition then the IDL compiler can pack values with smaller ranges more tightly within messages. `boolean` values, for example, are equivalent to `unsigned long[0..1]` values, and can be packed into a single bit each. Eight `boolean` values could therefore be marshalled into a single octet.

Bit-packed values must be shifted and masked before use and may span consecutive bytes, imposing significant processing overheads. More importantly, byte-level compression schemes fare poorly on data packed in this way. This is because characteristics of one value marshalled into a byte will be masked by those of other values in the same byte. If each byte contains, or is part of, only a single data value then there is a semantic correlation which compression algorithms may take advantage of to pick an appropriate dictionary. μIOP therefore supports only byte-aligned marshalling of data values.

Size Values

The CORBA CDR prefixes variable-length values with their length, encoded as a 32-bit quantity. This allows sequence values, for instance, to contain up to four billion elements. In practice, it is extremely uncommon for sequence or string values to exceed a few dozen elements. This is especially true when operating over low bit-rate networks. Values of enumerated data types are similarly marshalled as 32-bit quantities, in spite of typically having only a few dozen valid values.

Koala's Size values address this by providing support for integers up to the maximum size supported by the host system, which are marshalled as lists of 7-bit sections, using the high-order bit of each byte as a termination bit. Integers between 0 and 127 may thus be encoded in a single byte. The Size type is used to encode integer values which tend to be biased towards smaller values. Both sequences and enumerated types are therefore marshalled using the termination bit encoding.

7.3.5 Compression

In principle, compression could be used to further reduce bandwidth requirements. Each datagram would be block-compressed before being transmitted to its recipient. Most compression schemes cannot cope well with small datagrams because of the overhead in encoding a symbol dictionary [79]. Many compression schemes have sufficiently large computational overheads that they are unsuitable for use with embedded systems. Koala has been designed taking the view that compression is best handled at the application level, allowing it to be tailored to particular interfaces or omitted entirely. Application-specific compression schemes may be plugged into the ORB using the protocol extension mechanism and enabled on a per-object basis, allowing application-level compression to be supported transparently if required.

7.3.6 Caching

The LW-IOP specification proposes the use of client and server caches for large data values such as strings. Frequently-used strings of text or binary data are sent once and referred to by a unique identifier in subsequent requests. Caching schemes of this kind assume that several small messages may be transferred in less time than one large message, allowing cache coherency protocols to operate efficiently. This assumption is only valid in networks with both low bandwidth and relatively low latency and fails when applied to networks with extremely high latency such as PEN.

Koala does not perform implicit caching of parameter values on behalf of clients, instead assuming that individual applications should choose caching techniques appropriate for their specific needs. The ORB may be extended to support application-specific caching protocols on a per-object basis, or the application may instead use interfaces that provide caching explicitly at the application level. Devices supporting bitmap graphics should allow clients to cache font data, for instance. The application itself will have a better idea of when caching is appropriate and when it is not, given the tradeoffs, than the ORB.

In the absence of general-purpose caching of string values, Koala attempts to reduce bandwidth in other areas, described in the following sections.

7.3.7 Method Ordinals

General-purpose string caching will cause the names of repeatedly accessed methods to become cached, reducing bandwidth.

A similar reduction could be achieved by hashing all method names to 32-bit values. Reducing strings to hash values may result in multiple names with the same hash, causing ambiguity when the hash is used. In a static system, a client might revert to a second hash function to avoid clashes. In a distributed environment, it may be the case that the server is aware of a clash between two supported methods in different interfaces, while a client interested in only one of the interfaces will not be aware of the clash. Even so-called “perfect” hashing schemes, which assign a unique hash to each possible value, cannot address this issue, since assignment of unique hashes requires complete knowledge of all possible data values.

A simpler and more robust scheme providing a reasonable reduction in bandwidth is to specify methods by *ordinal*. The first invocation of a method on a particular object will be made using the method’s full name. The reply to such an invocation will include an extra field, marshalled as a size value as described previously, indicating the ordinal of the method within the server. Since interfaces typically have fewer than a hundred methods, most methods’ ordinals will reduce to a single-byte value when marshalled. Subsequent invocations to the object will specify a zero length method name, followed by the method's ordinal value.

The ordinal encoding results in a saving of 5-15 bytes per invocation when used with the Eucalyptus architecture's sample applications. For requests with few parameters, this can amount to a reduction of around 30%. Since ordinals are assigned statically by the servant, method invocation by ordinal does not suffer the same coherency problems as caching schemes. For extremely resource-limited clients, there exists a tradeoff between bandwidth savings and storage requirements, since a table of ordinal values must be stored by each stub. Because this table contains a single 32-bit value for each method a stub supports, the overhead is not excessive for even moderately well equipped appliances. Optimisations may also be applied to take advantage of the fact that similar objects hosted by the same appliance will usually use the same set of method ordinals.

7.3.8 An Example Protocol Extension: InOut Parameters

This section considers an optimisation of the `inout` parameter-passing scheme, illustrating a possible extension to the core Koala protocol. The default implementation of the `inout` parameter passing scheme transfers such parameters from the client to the server initially, and then returns them in addition to the invocation's normal return value.

If the parameters passed through the `inout` mechanism are not modified by the servant then the bandwidth required to return them to the client is wasted. The behaviour of methods using `inout` parameters can be optimised by tracking changes made to such parameters during invocations. Simple parameters such as integers and strings need only be returned if their values have been changed. The servant passes an extra bitmap field to the client when returning `inout` parameters to indicate which parameters are present.

Complex data types such as structures, sequences and arrays need special handling, since they will usually only change in part. As for simple values, changed parameters are indicated in the returned bitmap field. Instead of sending the new value, only changes to the parameter are returned, in a type-dependent manner, examples of which are given below.

- `struct` values are expressed as a second bitmap field indicating which fields of the structure have changed and which have not. The changed parameters are then encoded in order, following the bitmap.

- `sequence` values are expressed as a sequence of changed regions. Each changed region consists of start and end indices, followed by the changed values.

Efficient change-detection algorithms may not be possible on some platforms. Simple comparison of values is reasonable for many simple types but can become prohibitively costly for complex types, especially those that are nested within one another. This optimisation is therefore an optional addition to the standard Koala protocol. If client and server both support the differential encoding then the servant may return `inout` parameters using the scheme. Even if both parties support the encoding, the servant is still permitted to choose between it and the normal encoding on a per-invocation basis.

In practice `inout` values are not commonly used in many existing CORBA applications, and their use is arguably a point of bad coding style. The ease with which the Koala ORB may be extended to optimise their behaviour means that in spite of their rarity, applications which use them need not necessarily suffer a performance penalty.

7.4 Latency

Latency poses a significant problem for appliances using low power networking protocols such as those provided by PEN. Three main sources of latency affect the transmission of datagrams over low power protocols. These are described below.

7.4.1 Fundamental Latency

When operating on full power, PEN is capable of delivering messages between devices with around 2-4ms latency[1]. This is termed *fundamental latency*, since

[1] Note that this value is valid only for the generic prototype units used and would likely be an overly conservative estimate for production systems.

it is a consequence of the way in which PEN handles messaging at the hardware level and so cannot be reduced by improvements in protocol design. Fundamental latency sets a lower limit on the latency of the PEN system.

7.4.2 Hard & Soft Latency

When fully powered, a pair of PEN devices will suffer a total *hard latency* of around 10-15ms per datagram, due to the plea/offer and acknowledgement mechanisms of the R2-Layer protocol. When operating using low power protocols the mean latency rises dramatically to around 15,000ms for a typical sensor device. This is termed *soft latency* since its effect can be reduced through careful use of R2-Layer primitives.

The greatest contributor to messaging latency in PEN devices is soft latency, caused by the need to wait for devices to wake before datagrams can be delivered to them. Soft latency can be reduced by considering patterns of usage and optimising the R2-Layer's behaviour accordingly. Although the R2-Layer itself can take measures to reduce soft latency this is not generally desirable. More intelligent power saving decisions may be made by higher level interfaces such as those provided by Koala than by the low level R2-Layer messaging interfaces themselves.

Figure 7.1 shows the steps involved in an R2-Layer datagram transmission, and the associated latencies. This chapter primarily considers the soft latencies of interactions, since this is the greatest contributor to overall latency.

7.4.3 Latency-Aware Primitives

Koala performs some implicit optimisations of the R2-Layer based on programs' use of high-level ORB interfaces, which are described later in this chapter. Although these optimisations greatly improve the performance of applications that were not designed with low bandwidth, high latency networks in mind, more considerable gains are to be made through the use of specialised high-latency primitives. The following sections describes the high-latency primitives provided by Koala.

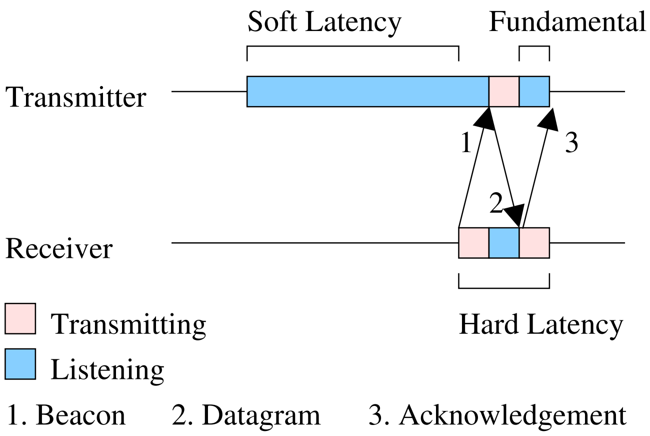


Figure 7.1: Datagrams transmission latencies using the R2-Layer

7.5 Remote Procedure Calls

Remote Procedure Call systems provide a convenient mapping between simple request-response interactions via a network and a more familiar procedure-call semantics. RPC implementations "capture" local invocations of methods and transparently send them over an underlying network, where the appropriate method of the remote servant object is executed. Fundamental to the operation of conventional RPC systems is the assumption that a remote invocation will only "return" when execution of the call on the remote host has completed.

When dealing with objects accessed over a high-latency network, the requirement to block the calling thread until the invocation has been processed incurs a significant latency penalty. While some method invocations must necessarily be invoked sequentially in order to preserve application semantics, many sequentially specified invocations could instead be performed in parallel. Using sequential RPC systems, parallel invocations may only be performed by spawning extra threads and invoking the required methods separately from each thread. This avoids the latency problems associated with sequential invocation but results in considerable overhead in terms of system resources. Threads are non-trivial entities to produce on demand and in addition require the allocation of supplementary resources. Asynchronous invocations to individual remote objects would ideally

```
# - Iterate through the sensors, querying them
for sensor in sensors:
    print ``Result:'', sensor._g_Reading()
```

Figure 7.2: Code to query multiple sensors sequentially

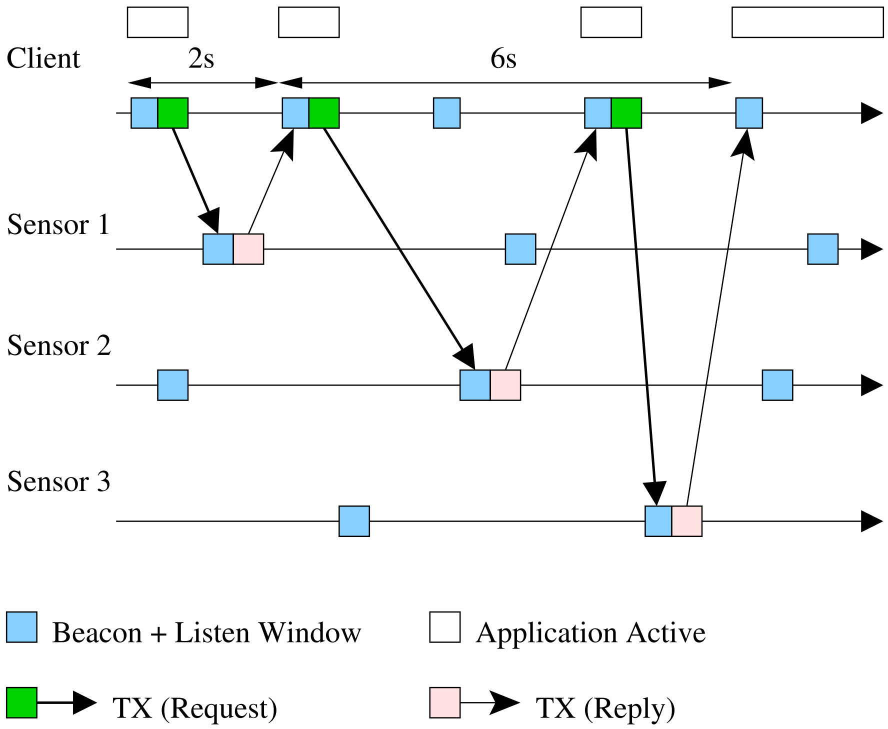


Figure 7.3: Synchronous querying of three sensor devices

be dispatched and their results collected by a single thread, using a single communications channel. Although CORBA provides several asynchronous interfaces, including the Asynchronous Messaging Interface, these are not well suited to supporting general object-oriented interactions over high latency networks, as will be shown.

Figure 7.2 illustrates the use of synchronous RPC to query three separate sensor appliances. The example code snippets in this section are in Python, unless otherwise specified. The asynchronous version of a stub method returns an object which must be *dereferenced* to obtain the actual result. In the examples, asynchronous result objects are Python callable objects, such that calling them with no parameters dereferences them. Figure 7.3 shows how the application might behave using synchronous RPC. The figure illustrates the delays between applications passing data to the radio layer for transmission, and that data being received by the intended recipient. Details of the transmission process are omitted for clarity. Note that the latency of the query operations varies considerably, since it is dependent upon the relative phase-shifts of the communicating appliances' beacons.

7.5.1 Asynchronous Invocation

The Koala ORB provides native support for asynchronous method invocations, rather than having clients achieve the same effect using threads. This reduces the resource requirements of such programs, and can greatly simplify client-side programming. It is a responsibility of the ORB to provide a convenient asynchronous invocation interface and to map this to the most suitable implementation for the target network.

Koala is capable of invoking each method an interface defines either synchronously or asynchronously. Both forms of invocation accept the same parameter list. Synchronous stubs return their results directly, usually by value, when the remote method has completed. Asynchronous stubs instead return a *lazy result* object as soon as they have dispatched the request to the servant, without waiting for it to be executed.

To the application programmer, lazy results bear similarities to the reference data-types supported by many modern programming languages. They must be *dereferenced* in order to obtain the actual value they contain. Unlike language-level reference types, lazy results do not necessarily refer to a value, but instead may contain fields identifying a pending remote procedure call. When a lazy result is dereferenced, it will return the value it contains, if any. If no value is available then the calling thread will be blocked until a reply is received for the pending RPC, in the same way as for synchronous invocations. Lazy result objects may be discarded by applications if they are no longer required, in which case the application will never be blocked awaiting their results. Depending upon the

```
# - Iterate through the sensors, querying them
results = []
for sensor in sensors:
    results.append(sensor._g_Reading())
# - Iterate through the results, printing them
for result in results:
    print ``Result:'', result()
```

Figure 7.4: Code to query multiple sensors in parallel, using lazy results

protocol extensions it supports, the client ORB may inform the servant ORB that the result is no longer required, to avoid wasting resources delivering it.

Figure 7.4 illustrates the use of lazy results to query three of sensor devices in parallel. Figure 7.5 shows how the application might behave. Note that the performance benefit is relatively small, as a consequence of the highly variable network latency.

This implementation will not process the result from Sensor 3 when it is first received, because it is instead blocked when it attempts to dereference the lazy result from Sensor 2. As with synchronous invocation, the calling thread is unable to perform any other processing while blocked awaiting a result. One approach is to maintain a separate background thread to continue processing while the main thread is blocked. While resource-limited appliances may not be capable of dispatching individual RPCs in separate threads, they may be able to support statically-allocated threads.

The CORBA Dynamic Invocation Interface (DII) [53] provides asynchronous polling and callback invocation facilities, as does the CORBA Asynchronous Messaging Interface (AMI) [51]. These facilities allow applications to continue processing if necessary while waiting for the results of asynchronous invocations. Whether polling or callbacks are more suitable is heavily dependent upon the application and host architecture, although frequent polling is seldom desirable, especially for low power appliances.

The Koala ORB provides all three models of asynchronous invocation, allowing the most appropriate to be chosen for each invocation. Lazy results provide a `_ready()` method through which they may be polled and a `_callback()` method through which a callback may be registered. Figures 7.6 and 7.7 illustrate the use of the polling and callback mechanisms to implement our sensor querying

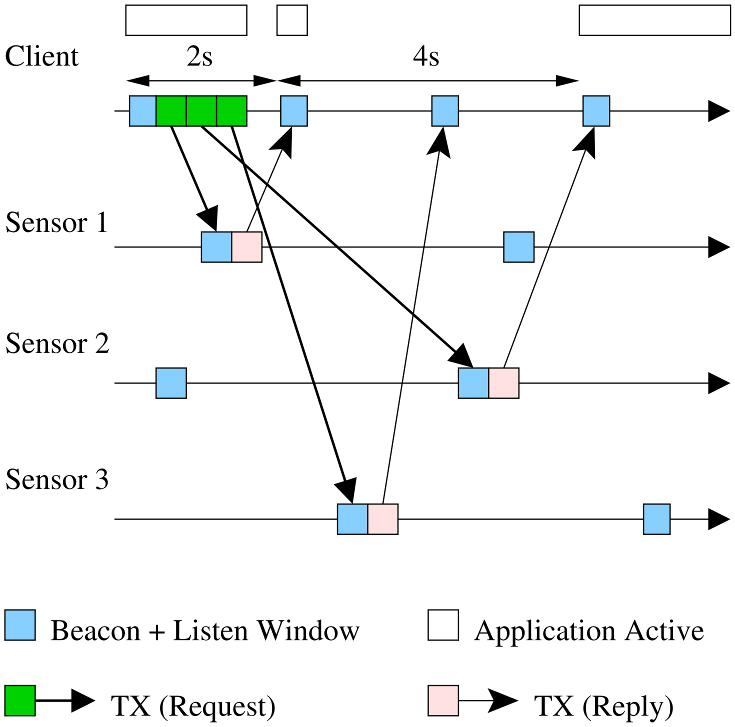


Figure 7.5: Asynchronous invocation by blocking on lazy results

```
# - Iterate through the sensors, querying them
results = []
for sensor in sensors:
    results.append(sensor._g_Reading())
# - Poll for results arriving
while results:
    for result in results:
        if result._ready():
            results.remove(result)
            print ``Result:'', result()
    # - Do some other processing
    ...
```

Figure 7.6: Asynchronous invocation using polling

```
def MyCallback(object, result, data):
    print ``Result:'', result()
# - Iterate through the sensors, querying them
for sensor in sensors:
    result=sensor._g_Reading()
    result._callback(MyCallback, None)
# - Do some other processing
...
```

Figure 7.7: Asynchronous invocation using callbacks

```
result=sensor._g_Reading()
try:
    print ``Result:'', result()
except:
    print ``Unable to query sensor!''
```

Figure 7.8: Catching exceptions when using lazy results

application.

7.5.2 Using Lazy Results

The use of lazily-evaluated results may alter the semantics of remote invocations. Many methods will return data values. When a client attempts to use these values it will block if the relevant request has not yet completed. In using a value, a client implicitly ensures that the associated request has completed. Exceptions raised by invocations are propagated to the clients only when they attempt to use the lazy result value, as shown in Figure 7.8. If a callback has been registered with the result object then the exception will be raised only when the callback attempts to use the result, as shown in Figure 7.9.

Asynchronous invocations, like parallel invocations dispatched by multiple threads, provide no guarantees on the order in which they will be dispatched by

```
def MyCallback(object, result, data):
    try:
        print ``Result:'', result()
    except:
        print ``Unable to query sensor!''
result=sensor._g_Reading()
result._callback(MyCallback, None)
...
```

Figure 7.9: Catching exceptions when using callbacks

servants. If methods have no side-effects then the potential for re-ordering of asynchronous requests does not pose a problem. If a method does have side-effects that must be visible before further invocations can safely be dispatched then the client must explicitly synchronise with the request by dereferencing the lazy result object.

Methods with no return value will almost always have side-effects. These methods must therefore return a lazy result object in the same way as those with return values, to allow clients to synchronise with them. Lazy results from methods with no return value may be dereferenced to ensure synchronisation but will return a null value. Explicit synchronisation is often unnecessary if methods do return values, since implicit synchronisation caused by normal use of lazy results will provide the required guarantees on ordering of requests.

Koala supports asynchronous invocations using separate stubs to those used for synchronous ones. These stubs are returned by a special method of synchronous stubs, thus indicating that calling code is designed to cope with asynchronous invocations.

7.5.3 The "Const" Pragma

Rather than having applications choose between synchronous and asynchronous stubs explicitly, the IDL compiler could be made to build mixed stubs, with some methods synchronous and others asynchronous. In order to do this, the compiler requires the programmer to tag interfaces to indicate which methods affect the target object and which do not.

The IDL compiler can allow individual methods to be tagged with a `const` pragma, to indicate that the method does not change any internal state in the object, as shown in Figure 7.10. Such methods can in principle be dispatched asynchronously and in any order, since they cannot affect the results of subsequent invocations. Non-const methods are dispatched synchronously, ensuring that they have taken effect by the time execution returns to the caller. All asynchronous requests dispatched by a thread must have completed before any synchronous requests are dispatched, to avoid inconsistencies if requests are reordered when received by the servant.

```
interface AttributeStore {
    Any GetAttr(in string name) const;
    void SetAttr(in string name, in Any value);
    readonly attribute unsigned long AttrCount;
};
```

Figure 7.10: IDL definition of a simple attribute store

7.6 Distributed Algorithms

The Eucalyptus architecture defines a distributed trader mechanism through which references to objects matching certain characteristics may be obtained. Traders support a `find()` operation, which accepts the name of a particular well-known interface, and returns a list of all objects supporting that interface.

Trader-local queries against pre-cached object references will complete effectively instantaneously. The trader will collate the matching objects into a single *sequence* and return this to the client. The entire query process will ideally involve only a single round-trip between client and server. In practice, trader-local queries will be delayed further as the trader contacts objects it has not previously indexed to discover whether they satisfy the current query.

Queries may also propagate into other traders to which the originating trader contains references. The time taken to execute a query may therefore depend upon how quickly it can be processed by these “child” traders. Synchronous trader implementations must query each child sequentially, incurring further round trip delays. Asynchronous implementations using lazy results can dispatch all child queries simultaneously, then collate the results and return them to the client. Although this reduces the number of round-trips significantly, the originating trader must still wait for all of its children to respond before returning a result. The time taken to resolve the query therefore depends on the slowest of the children. Using PEN’s low power primitives, as the number of children increases, the round-trip delay will tend towards the worst-case upper limit of two beacon intervals. If the query is propagated from a child to further grand-children then the query will take still longer to complete.

Results should ideally be returned to the client as and when they become available. If a client is interested in searching for objects which match a given characteristic then the order in which it receives them is in most cases irrelevant. By returning results as they become available, the trader allows the client to contact objects to perform further verification while subsequent results are arriving. In many cases, one of the first objects returned will prove suitable, in which case the client may discard the rest.

7.7 The Lazy List Abstraction

This functionality is provided by the lazy list primitive, a high level abstraction inspired by the functional programming technique of the same name. Lazy lists are composed of nodes, each containing a *value* and lazily-evaluated *successor*. The successor returns the next node of the list, or raises an exception if the list contains no more data.

Programs may construct finite lazy lists using a concatenation primitive. Lazy lists with arbitrary contents are constructed by supplying a replacement successor value, allowing lists of natural numbers, primes, etc to be supported.

Since they are a generalised primitive, lazy lists may be used not only to support trader queries, as in the Eucalyptus architecture, but also to improve the performance of distributed sensor fusion systems, for example. Readings would be returned lazily to the first level of processors, which might filter out erroneous or duplicate entries and return "sensible" readings lazily, for a further stage of processing.

7.7.1 Receiving Lazy Lists

Operations such as trader queries, which return multiple unordered results, potentially collated from separate sources, are specified in the remote object's interface as returning a lazy list.

If the client invokes the method via a synchronous stub then the query method will return the head of the list of results, consisting of the first available result value and a successor value. The client will therefore be blocked until the first result is available.

Subsequent results will be received by the ORB in the background, as the client iterates through them. If the client processes data faster than it is received then the dereference operation will block until more is available, in the same way as when dereferencing lazy results. If the client does not wish to process the remaining entries in a lazy list then it simply deletes it. The ORB will notice the deletion and inform the remote servant that no further data is required.

7.7.2 Creating Lazy Lists

Lazy lists may be populated with a mixture of immediately available data values, lazy result objects, and other lazy lists. The list abstraction merges data from the three different sources so as to make it available to recipients as quickly as possible.

Simple Lists

The simplest lazy lists contain only immediate data values, and behave similarly to standard sequence data-types. Immediate values are returned in the order in which they are appended to the lazy list. If a lazy list contains both immediate values and data from other sources then the immediate data will always be supplied to clients first.

Merging Lists

In our trader example, a query may propagate to multiple child traders, collating the results and returning them to the client. This involves merging all results into a single list, without performing further processing.

Koala's lazy list primitive provides both `append()` and `merge()` methods. The former method operates in the normal way, attaching one lazy list to the end of another. This implicitly requires that the first list will terminate at some point. The latter operation does not require that the first list terminate. Whenever the successor value is dereferenced, it returns a value from any one of the merged lists. If several lists contain accessible data then items are returned using a round-robin scheme. If one or more lists were returned from remote method invocations then an item will be chosen from a list that has at least one value ready to be accessed. The merged lists are polled for available data via their successor values, and if none have available data then callbacks are registered to inform the lazy list implementation when new values have arrived.

A trader wishing to propagate a query and return all results it receives to the client simply merges the returned lazy lists into one and returns it.

Lazily Processed Lists

Specialist traders capable of supporting more advanced domain-specific queries than the general-purpose traders previously described may be used by particular applications. Such traders query standard traders to locate objects of the required types. Each new object should be processed as it is received, and passed on to the client application only if it satisfies their requirements.

To use the lazy list abstraction, however, the specialist trader's query methods must populate a list with potential results and return it to the caller, via the ORB. Ideally, the list should be returned from the query almost immediately, and should then continue to be dynamically populated by the query function. This requires that lazy lists of potentially suitable objects obtained from standard traders be *filtered* before being returned to the client.

Koala provides filtered lazy lists to support this common mode of operation. The specialist trader first queries one or more standard traders and merges the returned lazy lists into one. It then creates a lazy filter, passing the original list and an appropriate filter function. The filter list's successor function behaves in the same way as that of the original list, with the exception that it tests each item against the supplied filter. Items are returned to the caller only if they pass the filter.

7.7.3 Transferring Lazy Lists

Lazy lists require special handling when they are to be passed between servant and client across the network. Because pending lazy results from asynchronous invocations may be contained within lazy lists, the servant ORB may need to perform further processing on the list even after it has been returned by the servant. Further complication arises when processing distributed operations such as nested trader queries. A specialist trader, for example, may query several standard traders remotely, and merge and filter the resulting lazy lists. When a query is made to the specialist trader, it will build the filtered list and immediately return it to the client. At this point, the specialist trader's ORB must continue processing the lazy list's contents, and the pending queries made to remote traders, even though the method that created them is no longer executing.

Requiring the ORB to continue processing lazy lists after the method that created it has finished executing can pose performance problems, especially for resource-starved appliances. Lazy lists may contain large numbers of items, or may even be capable of producing arbitrary sequences of items on demand. An object could, for, example, return a lazy list of prime numbers, with each new value being dynamically calculated by the successor function. Clearly, it is not desirable for the servant's ORB to continue processing the contents of such a list once the client has no further use for it. Ideally, lazy list items should be delivered on-demand to clients. This is infeasible in high latency environments, however, since fetching each new item then requires a full round-trip.

The Koala ORB tackles this problem by returning items in batches. Any immediate data contained in a lazy list is returned first, in a single batch. Whenever a lazy item becomes available, the ORB attempts to read as many items as possible from the lazy list, up to a pre-configured limit. Ideally, all available items would be read and delivered in a single batch to the client. This is not possible in the case where the list is calculated on-demand, since an infinite number of items will appear to be available.

The servant ORB returns lazy list items in batches. Each batch is marshalled as a sequence of the relevant type and delivered to the client ORB. The client ORB receives the sequences of batched results and stores them. While the caller reads data from one batch, a second is requested and stored by the ORB. Once the first batch is exhausted, the ORB starts returning items from the second to the caller, and requests that the servant ORB return the third batch, and so on.

If the end of the lazy list is reached then the servant ORB informs the client ORB, allowing it to terminate the client-side copy of the list appropriately. Similarly, the client ORB will inform the servant if the caller destroys the list without processing all the items it contains. The servant may then destroy the original lazy list, which may in turn result in other servants being notified to allow them to free any associated resources. The same process is executed if the connection between client and servant ORBs is broken while there is still lazy list data pending.

7.8 Monitoring Objects

Koala objects' attributes are normally accessed using `set()` and `get()` methods generated by the stub compiler. Reading or writing an attribute therefore involves dispatching a remote invocation to the servant.

It is often necessary in sentient computing systems to monitor the state of an object's attributes. A user may wish an alarm to sound when a sensor reading exceeds a certain value, or to a light to activate when a switch is set to the appropriate state. Applications could monitor objects' attributes by polling them, continually re-requesting their new state and checking it against some criterion.

Continuous polling of remote objects results in excessive use of bandwidth if used over standard networks. To reduce the required bandwidth, the frequency with which an object is polled must be limited, which in turn limits the responsiveness of the system. If a large number of objects are being monitored then the polling rate is further limited by bandwidth constraints. Over PEN's low power protocols, polling via RPC performs particularly badly, consuming scarce bandwidth while delivering extremely poor response times. Muller et al [44] argue against the use of polling as a means of monitoring the state of sensors within low power devices because of the resulting power drain. Similar arguments apply to polling of remote components by PEN devices.

Ideally, clients should be notified of changes in the state of objects they are monitoring without the need to repeatedly poll them. This can be achieved at the application level by having clients register callback objects with servants they are interested in. Whenever the servant's state changes, it will call a particular method of the callback object. Although there is a certain elegance in the reducing state change notifications to RPC operations, the scope for efficient implementation by the ORB is limited. Appliances capable of being monitored must host code to maintain a table of interested parties and to call those parties back. Care must be taken in the design of the code, because of the disparity between the blocking behaviour of RPC and the asynchronous behaviour we would desire from notification operations.

The notification schemes described above can be implemented at the application level, using an ORB supporting standard RPC. Because the ORB will be optimised for standard RPC operations, notifications layered over RPC may behave poorly. Some desirable notification protocols, such as broadcast notification, will be impossible to implement. The ORB can instead provide a high level interface to support monitoring of remote objects' attributes. The ORB can then implement notification in the most suitable manner for the underlying network. To further simplify applications which do not require guarantees as to the coherency of attribute values, the ORB can use the event mechanism internally to support incoherent attribute caching, as in [35, 57].

7.8.1 Events

The Koala ORB attaches extra semantic significance to attributes defined in interfaces. In addition to allowing access to attributes via `set()` and `get()` methods, the ORB provides a `watch()` method, through which clients may obtain notifications of changes to an object's state. The `watch()` method returns an *event notifier* object, providing similar functionality to asynchronous lazy result objects. This allows clients to poll, block, or receive callbacks for each event. Unlike lazy results, event notifiers will continue to return available event data to the caller until they are destroyed.

Koala events are directly associated with changes to the state of objects, in contrast to support for arbitrary events in systems such as SIENA [8] and the Cambridge Event Architecture [3]. The semantics of Koala events are therefore well-defined, making their behaviour more intuitive to users when compared with general composite event systems. Koala events may nonetheless be exchanged with context-aware architectures such as ECA [37] and SPIRIT [22] via bridges, in spite of the difference between their event models. This might achieved by preprocessing the Koala IDL definitions to CORBA IDL definitions with the addition of appropriate CORBA Event service interfaces.

Clients may specify which of an object's attributes they wish to watch to the granularity of particular interfaces. Depending upon the target language, clients may pass a list of interfaces to the object's `watch()` function, or may invoke separate `watch`*<interface>()* functions for each required interface. If an interface is watched then changes to any of its attributes will cause an event to be dispatched to the client, containing the values of all the interface's attributes. If attributes from several of an object's watched interfaces change simultaneously then a single composite event containing attribute values from all the changed interfaces will be dispatched. Internally, interface-level event filtering may be performed either by the client or servant ORB. Servants residing on appliances with little or no processing capability may ignore filter specifications and instead dispatch notifications for all state changes. Notifications which are surplus to requirements will normally be filtered out by the client ORB, except on extremely limited appliances. Applications must therefore cope with receipt of unsolicited event data on such platforms.

Each Koala servant maintains a *change counter*, incremented each time the servant notifies the ORB of a change in one of its attributes. The counter is used with some event transports to suppress duplicate events. With slow event transports, it is used to allow multiple similar events to be compared and only the most recent from each interfaces to be dispatched. Clients will therefore be notified at most once of each event.

Native event support allows the ORB to choose the most efficient transport for events. Three event transport mechanisms are outlined below.

Broadcast Events

Simple devices exporting a single object with a small number of attributes can avoid the overheads of reliable event transports by using broadcast events. PEN's R2-Layer allows a small amount of supplemental data to be appended to beacon packets. A sensor might embed a standard Koala event packet in the payload area of its beacon, for example. ORBs running on nearby devices which hear the beacon will unmarshal the event and pass it to any interested clients.

Event delivery using beacons is unreliable, since the loss of a broadcast packet cannot be detected by the sender. Broadcast events are supported for simple sensor devices in non-critical applications such as home control. Sensors embed sensor events in each beacon regardless of whether their state has changed, so that clients missing one beacon will be notified by the next. If multiple events occur between beacons then only the most recent will be dispatched.

An extension to the broadcast event mechanism is for the receiving Koala ORB to cache the most recent values of an object's attributes for later use by clients. Attribute caching of this kind is more straightforward than would be the case if using other transport methods, since events may only be received at discrete intervals. If a sensor beacons every ten seconds and delivers events in this way then attribute data is guaranteed to remain "valid" for ten seconds after receipt.

Events-over-RPC

An approach favoured by the OMG CORBA Event [49] and Notification [50] services is to pass events in method invocations. Internally, Koala supports an RPC event delivery mechanism based on *counting events*. The ORB watches remote objects by invoking a private `await()` method, passing the event counter value of the last event received. The invocation blocks in the servant until the event count exceeds that specified, at which point the most current attribute and counter values will be returned.

If events occur more quickly than they can be delivered then successive events received by clients will not have consecutive event counts. In a general event system, this would mean that some events had been discarded. Because Koala events indicate changes in the state of objects' attributes, the missing events will simply describe intermediate attribute states. For most applications, intermediate attribute values will not be required.

Because counting events are implemented by having the `await()` invocation block until an event occurs, events are delivered as RPC replies. This means that the event delivery is a much more lightweight operation than would be the case with RPC-callback notifications. The ORB's use of the R2-Layer is optimised for the common case that a client ORB dispatches a new call to `await()` immediately after receiving the results of the last.

Streamed Events

Many event services deliver events to applications via RPC-style callbacks, but transfer those events through the network using a separate *event stream* mechanism, tailored to the target environment. State-change events from Koala objects can be delivered in much the same way. The client and servant ORBs negotiate using protocol extension messages to determine whether they share a common streamed event protocol. If the two ORBs are compatible then the client uses the event stream protocol to obtain events from any objects hosted by the servant ORB.

Again, it is necessary to ensure that servants which change their state excessively quickly do not flood the network. One way to achieve this when using the R2-Layer is to only dispatch an event to a stream when confirmation of delivery of the previous event has been received. The ORB may be optimised to handle streamed events in a similar way as for events over RPC.

7.8.2 Degrees of Accessibility

In addition to monitoring the state of an object, it is often useful to be able to monitor whether or not it exists, and whether it is contactable. Wireless environments introduce the possibility of disconnected operation. Applications cannot assume the same constant connectivity afforded by wired equivalents.

To provide connectivity and existence information to clients, the Koala ORB defines a number of standard exceptions. Whenever a client invokes a method on, or attempts to retrieve an event from, a Koala object, one of these system exceptions may be raised in addition to any exceptions defined by the servant's interface. Koala recognises three main *degrees of accessibility*:

- `HOST_INACCESSIBLE` - the object's host device cannot be contacted.
- `NO_SUCH_PORT` - the object's host can be contacted but does not recognise the relevant network port.
- `NO_SUCH_OBJECT` - the object's host is contactable, and is aware of the specified network port, but does not recognise the object key.

The `NO_SUCH_PORT` and `NO_SUCH_OBJECT` exceptions will usually be treated similarly by applications, because they both indicate a permanent failure of the target object. It is assumed that if the object or its host application are unavailable then this is an unrecoverable error. Systems in which such failures are recoverable must instead return `HOST_INACCESSIBLE` exceptions when objects become unavailable.

The `HOST_INACCESSIBLE` exception is used by the ORB to indicate that an object's host is not *currently* contactable. In wireless environments, it will often be the case that remote hosts become temporarily inaccessible. The exception should not necessarily be taken, therefore, to indicate that an object is permanently unavailable. Clients monitoring an object's state will receive an `HOST_INACCESSIBLE` exception if the object cannot be contacted, followed by a normal event when it becomes contactable again.

To applications residing on PEN appliances supporting only local-area messaging, the `HOST_INACCESSIBLE` exception provides proximity information. In order to be contactable, an object must reside on an appliance in direct contact with the client. If the host appliance supports wide-area or multi-hop routing, however, then an object may be contactable even though its host appliance is far away from the client. Koala therefore provides a fourth degree of accessibility, using the `is_local` object attribute. The `is_local` attribute is a boolean, used to indicate whether or not the object's host is local. The attributes value may be watched in the same way as other attributes, so that context-aware applications can therefore use it in place of `HOST_INACCESSIBLE`, to provide a reliable proximity heuristic. The details of `is_local`'s implementation may vary from platform to platform without requiring applications to be modified.

7.9 Low Power Optimisations

In addition to providing latency-aware primitives and multiple high-level communications mechanisms to applications, the Koala ORB can perform a number of automatic optimisations. Each optimisation can be enabled or disabled on a per-client or per-servant basis, if required.

7.9.1 Event Notification Chaining

In certain applications, clients dispatch new requests to servants immediately in response to receipt of events from them. This is most commonly the case when the event's originator is some form of user interface. In response to input from a user, indicated by receipt of an event, the remote device may wish to update the user interface to display new information.

If the event source wakes to transmit events and immediately returns to sleep then subsequent requests from a client must wait until the source wakes again and transmits a beacon. This is a consequence of the way in which PEN's low power messaging primitives operate. Koala servants likely to receive requests in response to event notifications can maintain a listen window immediately following event delivery, to listen for new requests.

7.9.2 Sequential RPC Chaining

Soft latency affects the round-trip time of individual requests, as well as the round-trip time between delivery of a reply and receipt of the subsequent request. When a request is dispatched, it will usually be dealt with immediately by the servant and a reply delivered. After dispatching a request, however, the client device will usually enter a sleep state. The servant must therefore wait for the client to beacon before it can deliver the reply. The Koala ORB can be configured to keep open an extra listen window immediately following transmission of a request, so that methods with little processing overhead can return their results without waiting for the next beacon. Similarly, the ORB can open an extra listen window following transmission of a reply to a client, to allow the client to immediately dispatch the next request.

By introducing extra listen windows following dispatch of requests and replies, remote procedure calls may be "chained". The communicating nodes enter a brief period of high-power activity, keeping their network interfaces powered until the interaction is complete. RPC chaining is unsuitable for applications in which significant processing must be performed before a result is returned, in which case the client simply wastes power. A more conservative optimisation is to have the client provide an extra listen window only when querying the state of an attribute and not to do so when invoking other methods.

Secure Chaining

Sequential RPC chaining poses a potential security threat, particularly to battery powered devices. By repeatedly dispatching requests to a host device, a malicious attacker can perform a denial-of-service attack. The receiving host will incur power, processing and bandwidth costs in replying to the incoming requests, even if they are essentially meaningless, if only to deliver appropriate exception messages. In the absence of RPC chaining such attacks usually have significantly less impact, since the attacker must wait a full beacon interval between requests. With a naive RPC chaining implementation, requests can be dispatched as quickly as the target can respond, preventing it from entering the sleep state and thus flattening its battery until it "dies".

Because chaining is implemented by the ORB rather than by the R2-Layer, more intelligence can be exercised. The Koala ORB will not use RPC chaining when an invocation raises an exception, for example. In itself, this cannot protect the device from repeated attempts to read an attribute, in order to perform the attack. The ORB therefore limits the number of chained invocations to the host based upon a pre-defined invocation rate. The maximum rate at which invocations may be received can be reconfigured by applications if required. The ORB's rate limiting features can be applied not only to PEN but to other networks such as TCP/IP, to provide protection against brute-force denial-of-service attacks.

7.10 Evaluation

The Koala ORB attempts to support distributed object-oriented systems efficiently by implementing a variety of high level primitives using the most appropriate underlying transport mechanisms. Of these, the lazy list primitive is of particular importance in simplifying the management of invocations performed simultaneously on many objects.

Consider the case of a user wishing to collect readings from all nearby sensors of a particular type. The user may wish, for example, to verify that the temperatures in different parts of an office are within specified bounds. The user carries a PDA hosting the monitoring application. The application discovers nearby sensors using a *trader* hosted by the PDA. The trader may be queried for all local components of a particular type. To perform a query, all local appliances must be contacted to discover whether components they host are of the desired type.

7.10.1 Synchronous RPC

A small office may contain twenty appliances, of which four might be sensors, for example. For simplicity, it is assumed that each appliance hosts a single component and that all appliances transmit beacons at ten second intervals. The user's PDA is assumed to use a two second beacon interval for the duration of the operation. Using sequential invocation, it will take five seconds on average to transmit a request for the type of a component, and a further second for the reply to be delivered. Since only one request can be in progress at any time, the whole process will take 120 seconds on average. Once all the appliances have been contacted, the sensors must be queried, taking a further 24 seconds.

7.10.2 Asynchronous RPC

If the trader implementation instead uses asynchronous RPC then the initial discovery phase may be drastically shortened. Requests for components' types will be dispatched in parallel, to a close approximation. This means that all local components can be queried within six seconds on average, and twelve seconds at most[2]. Because the trader must wait until all components have been contacted before returning the list of sensors to the application, the slowest component to respond determines the response time of the whole operation. As more components enter the environment, the average time required to execute a query will tend towards the upper limit of twelve seconds. Only when all components have responded will the application receive the list of sensors, whereupon it may query them for readings in parallel in at most ten seconds.

[2]This assumes that no request or reply packets require retransmission.

7.10.3 Lazy Lists

The trader implementation can be further improved if the query operation is made to return a lazy list. The trader will dispatch queries in parallel using asynchronous RPC to all local components, and return a lazy list to the application. Components are appended to the list with a filter function which verifies that they are of the desired type.

Upon querying the trader, the application will receive the resulting lazy list as soon as the first result becomes available. The application will iterate through the list by dereferencing each successor value in turn, and will be blocked whenever no new sensors are currently available. Only when all the components have been contacted will the successor raise an exception, to indicate that all results have been returned.

Assuming that sensor's beacons are evenly distributed through a beacon interval, it will take one and a quarter seconds on average to deliver the query request to the first sensor node. There will be a further one second delay on average before the reply is received by the PDA. It will therefore take just over two seconds before the first available sensor is obtained. The application will then dispatch an asynchronous request to retrieve the sensor's reading, an operation taking a further ten seconds on average, owing to beacon timings. The final sensor will be discovered by the appliance after just under ten seconds, so that the total time to discover and query all sensors is at most twenty seconds.

7.10.4 Lazy Lists and RPC Chaining

Much of the remaining latency when using the lazy list implementation to support the sensor monitoring application arises from the soft latency in contacting discovered sensors. This latency can be almost completely eliminated if sensors use RPC chaining. RPC chaining will cause sensors to remain active, listening for additional requests, immediately after replying to the initial trader query. Using lazy lists, the sensor reference will be returned to the application as soon as the sensors responds to the trader. The application can therefore dispatch its query while the sensor is still listening for a chained request.

In ideal radio conditions, RPC chaining will successfully reduce the total time to discover and query all sensors to under ten seconds. In busy radio conditions, RPC chaining will fail, resulting in a total time of just under twenty seconds. Without the use of lazy lists, RPC chaining will also fail, because the application will only dispatch queries to sensors once all of them have been discovered.

7.10.5 Beacon Advertisements and Events

Many sensor implementations may be so trivial that they are incapable of accepting method invocations. Such sensors will advertise the interface types of their hosted component in their beacons, and will accompany this with an event containing their sensor reading. The PDA can cache these readings for a short period, and return them in response to queries from the application. Again, this eliminates any delay in querying sensors' readings. The time required to query a sensor component's type is similarly reduced, since the information becomes available to the trader as soon as a sensor's beacon is received. This assumes that the sensor is of exactly the type required by the application, however.

7.10.6 Results

Table 7.1 gives estimated performance results for the sensor monitoring example, using synchronous RPC, asynchronous RPC, and asynchronous RPC in conjunction with the lazy list primitive. Results are also given to illustrate the effects of request chaining and beacon advertisements.

Implementation	Discovery Time (s) (min/avg/max)	Query Time (s)	Total Time (s)
Sequential	120	24	144
Asynchronous	12	12	24
Lazy List	2.25/6/9.75	10	12.25/16/19.75
Lazy List + Chaining	2.25/6/9.75	< 0.05	2.25/6/9.75
Lazy List + Beacons	1.25/5/8.75	0	1.25/5/8.75

Table 7.1: Estimated performance of a sensor monitoring appliance

7.11 Summary

This chapter has described the mechanisms provided by the Koala ORB to reduce the impact of high latency and low bandwidth networks upon the design of distributed applications. The ORB provides several high-level communications mechanisms and implements these using the most appropriate features of the underlying network. A sufficiently complete ORB has been implemented natively for the PEN platform to validate the reasoning behind the results given. A more complete implementation has also been produced using the Python programming language, to support Koala applications on desktop PCs and PDAs, and to illustrate interoperability across diverse platforms.

In the following chapter, the Eucalyptus architecture is described. The architecture builds upon the Koala ORB to provide discovery, access control and configuration facilities to appliances.

Chapter 8

The Eucalyptus Architecture

8.1 Introduction

Earlier chapters have discussed the design of a low power protocol suite for embedded devices and the issues arising from its use. In particular, these issues have motivated the design of the Koala object request and event broker, which provides asynchronous invocation and attribute state-change notification primitives in addition to more conventional object-oriented RPC. Applications built using these primitives can significantly reduce the impact of high network latency without resorting to direct programming of the underlying network.

The Koala ORB does not impose any constraints upon the structure of distributed systems that use it. Indeed, the ORB itself does not address a number of basic requirements of distributed systems, such as discovery or access control. Distributed architectures built to use Koala must provide their own structures to support such operations. This chapter describes the interfaces defined by the Eucalyptus architecture to provide these functions, using the Koala ORB's extended feature-set wherever possible to improve performance.

8.2 Discovery

The Koala ORB allows servants hosted by one appliance to be accessed transparently via the network from another. In order to access an object, the client must hold a valid reference to it. References to Koala objects are normally obtained either directly from the servant itself if it is co-located with the client, or as the result of a request to another remote servant. This poses a bootstrapping problem for clients, since in order to obtain an object reference they must already hold one to another object. The Eucalyptus architecture must provide a *discovery* mechanism, through which initial object references may be obtained.

8.2.1 Centralised Discovery Services

Distributed systems for wired networks can typically rely upon the presence of a central repository of resources through which suitable references are discovered. Clients must be pre-configured with the address of this central resource and may then use it to discover services attached to the network, and to advertise services they provide. This technique is used in most IP-based wired networks, by providing central DNS services to perform name resolution and configuring each client with their address. Further addresses are then obtained by passing names to the DNS service to be resolved.

In IP networks, the client configuration phase can be eliminated by using the Dynamic Host Configuration Protocol (DHCP). DHCP uses broadcast datagrams to discover nearby DHCP servers, which then assign the client a unique IP address and provide it with the addresses of local services, such as DNS and routing gateways.

IP-based wireless networks behave almost identically to the wired networks they replace, using the same DNS servers and routing gateways. Although some degree of ad hoc configuration can be afforded through the use of DHCP, there is an unavoidable reliance on the presence of supporting infrastructure. In aiming to provide a base level of ubiquitous interoperability between appliances, the Eucalyptus architecture has an intrinsic motivation for supporting fully ad hoc operation. Pairs of Eucalyptus devices must be able to discover each other even when effectively isolated from the rest of the world. Previous chapters have discussed algorithms for low power rendezvous and discovery of nearby nodes implemented by the PEN R2-Layer. If the goals of interoperability and scalability are to be met then an abstract discovery interface must be provided by Eucalyptus, rather than having applications deal directly with network-specific system interfaces.

8.2.2 Traders

Eucalyptus uses a lightweight *trader* mechanism to allow applications to import and export object references in a network-independent manner. Traders are containers for collections of references to related objects, and may be queried for lists of objects matching certain criteria. Eucalyptus traders are more primitive than those described by the ANSA [1] and CORBA [48] specifications, which are capable of indexing objects based on attributes associated with them when they are first registered with the trader. The Bluetooth trader [7] and the Jini Lookup Service [68] both provide attribute-based query facilities, to reduce the amount of network bandwidth and post-processing of results required when performing non-trivial queries.

The Eucalyptus trader specification allows objects to be indexed based upon the interfaces they implement and their positions within a hierarchical name-space. More complex queries must be implemented by specialist traders, or by post-processing results obtained by querying standard traders. Eucalyptus trader implementations can be a great deal simpler than more advanced traders, and are therefore capable of being hosted by even the most resource-starved appliances.

Eucalyptus traders maintain a hierarchical name-space internally, into which object references are linked, as shown in Figure 8.1. Each path in the name-space can have one object associated with it. The same object may be linked at several different paths within the same trader. When a query is made to the trader, only those objects whose interfaces match those required and whose path contains the specified prefix will be returned. Traders provide methods to unambiguously obtain the object linked at a specified path, and to find all paths at which a particular object is linked. The trader mechanism is used by Eucalyptus as the primary discovery service available to applications.

8.2.3 Nested Traders

Since traders are implemented as standard Koala objects, they may themselves be registered with other traders. If one trader is *nested* within another in this way,

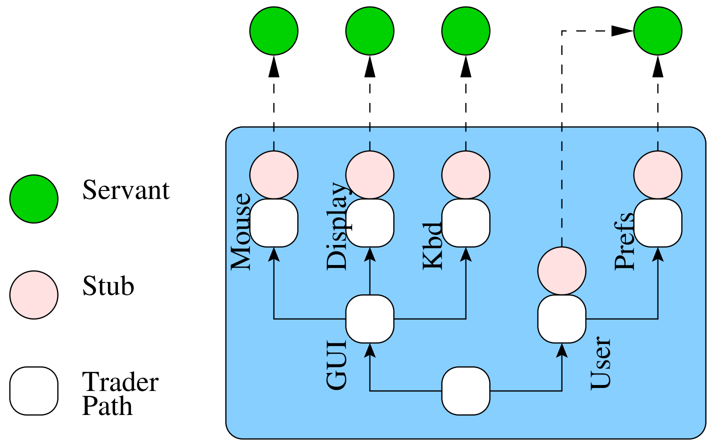


Figure 8.1: Layout of a Eucalyptus trader for a simple PDA

then all queries which reach the path at which it is registered will be propagated to it.

Nested traders may be hosted upon different appliances, potentially resulting in significant latency when propagating queries and obtaining results. Trader implementations may use asynchronous invocation and lazy results to propagate queries to multiple child traders in parallel. Although this reduces the overall latency, the minimum delay in satisfying the query is nonetheless limited by the time taken to query the slowest child trader.

More significant latency savings may be obtained if we relax the ordering constraints on the results obtained from nested trader queries. Each trader guarantees that the first results from any query will be those objects registered with it locally. These objects will be ordered by the lengths of their paths. Results obtained from nested traders will then be appended to this list of objects. If multiple nested traders are to be queried then their results are merged into a single lazy list, which is then appended to the list of local results.

Given this relaxed ordering constraint, trader implementations may make use of the Koala ORB's lazy list mechanism. Upon receiving a query, the trader first appends locally available matches to a lazy list in order. It then propagates asynchronous queries to any nested traders located within the query's path, and merges the lazy list of results returned by each to form a single new lazy list. Finally, the merged list is appended to the list of local results and the composite list is returned to the caller. Figure 8.2 shows the stages of a nested trader query.

The first results received by the caller will be the ordered local results, since these were placed at the head of the results list by the trader. These results will be available as soon as the trader has finished querying its locally-registered objects, with no extra delay incurred due to remote queries.

Subsequent results will be passed transparently via the original trader to the client as soon as they are returned by nested traders. By comparison, the simpler technique of collating all results into a single list within the original trader before returning them to the client would return no results at all until the slowest nested trader had replied.

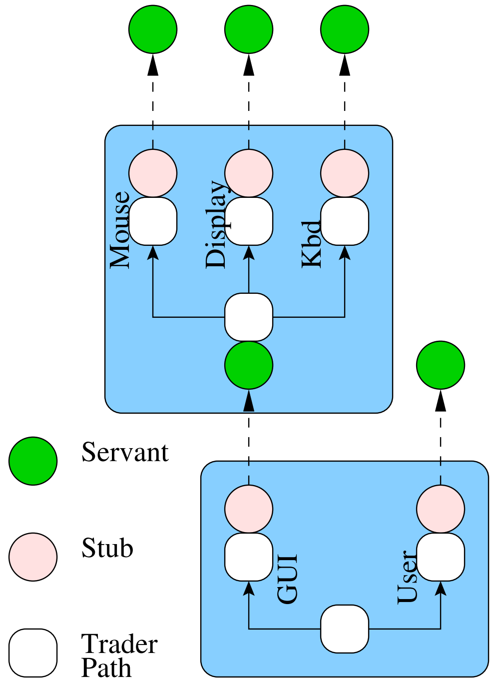


Figure 8.2: GUI appliance trader nested within a user's PDA trader

8.2.4 Import Traders

For each network transport supported by the Koala ORB, Eucalyptus should provide an *import trader* implementation. Import traders provide the standard Eucalyptus trader interface and respond to requests from clients in the same way as other traders, with the exception that attempts to add or remove objects from the trader are disallowed. Instead, import traders populate themselves with objects hosted by nearby appliances, using network-specific discovery protocols. PEN import traders may use the R2-Layer to perform low power discovery of nearby PEN nodes, while Ethernet traders may simply monitor network traffic on their local subnet. Objects hosted by nearby appliances are linked into import traders using the host's unique address as the path, allowing clients to limit the scope of queries to individual appliances easily.

Because traders are normal Koala objects, clients may watch them in order to be notified whenever an object is added or removed. This provides locality-based context information sufficient to support a wide variety of context-aware applications. How quickly the trader responds to an appliance arriving or leaving is heavily dependent upon the underlying network. An import trader for the IEEE 1394 network, for example, can respond extremely quickly, by watching the network for reset messages, which are broadcast to all connected appliances whenever an appliance is added or removed. Wireless import traders will typically respond more slowly, since it is usually harder to distinguish disconnection from excessive network congestion. Import traders therefore make no guarantees as to their response times.

8.2.5 Export Traders

In addition to an import trader implementation, a corresponding *export trader* implementation should be provided for each network transport supported by the ORB. Like the import trader, the export trader implements the standard Eucalyptus trader interface, with the difference that it supports both queries of the stored objects, and manipulation of them. The export trader attempts to *advertise* all the objects registered within it to nearby appliances, using appropriate primitives of the underlying network. Advertised objects will thus be discovered by the import traders of nearby appliances and made available to applications.

Precisely how many objects may be advertised by an export trader depends upon the capabilities of the underlying network. The only practical guarantee is that at least one object may be advertised, since the existence of the host appliance is sufficient to *implicitly advertise* a single object without requiring further information to be conveyed to clients. Further objects must be *explicitly advertised* by supplying extra information to clients using the network-specific advertisement and discovery protocol.

Import and export traders have been implemented for PEN and TCP/IP networks, to illustrate the use of network-specific advertisement protocols to support the standard trader interface. Some of the advertisement protocols implemented for these networks are described below.

8.2.6 Advertisement Protocols

Import trader implementations for PEN use the low power discovery primitives provided by the R2-Layer to receive beacons from nearby appliances. Each beacon will contain a list of APIDs that the appliance is willing to receive data for. If the well-known Eucalyptus APID is one of those listed in a beacon then the import trader assumes the beacon's originator to be a Eucalyptus appliance. The import trader will populate itself with references to objects advertised by discovered Eucalyptus appliances in one of two ways, depending upon the export trader implementation.

Implicit Advertisement

The simplest export trader implementations support implicit advertisement of a single object. The host appliance may register one object with the export trader, causing the object's key to be re-set to a well-known value (usually the null key). The export trader does not allow the registered object to be removed, nor may it be registered at any path other than the root of the export trader. This makes implicit advertisement suitable only for appliances with static advertisement requirements. Import traders discovering nearby Eucalyptus appliances will always assume that the remote appliance hosts an object with the null object key and will therefore add a reference to it. Figure 8.3 shows this process in operation.

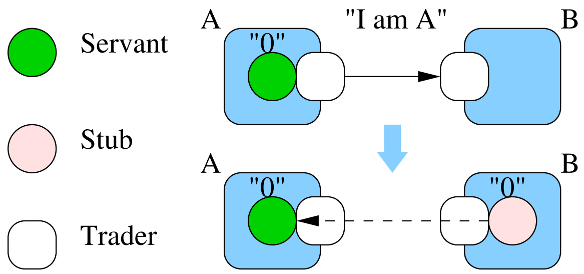


Figure 8.3: A single object being advertised and imported

More advanced export traders can support advertisement of several objects. This is achieved on the PEN platform by adding a small payload to each beacon datagram the appliance transmits, containing a count of how many objects the appliance wishes to advertise. As new objects are advertised, the trader assigns them the next object keys in a well-known sequence and increases the advertisement count. As with single-object advertisement, the export trader will not allow objects to be removed, meaning that this form of advertisement is suitable only for use by appliances with static advertisement requirements. Figure 8.4 shows multiple objects being advertised and imported.

Because the sequence of object key assignments is well-known, import traders can populate themselves with appropriate object references based on the advertised object count. Each object is linked into import traders at a path composed from the host appliance's network address and a component derived from the object's key. This is required in order to satisfy the requirement that only a single object is linked into a Eucalyptus trader at any given path.

Publishing Objects

The implicit advertisement protocols used by Eucalyptus are limited in their support for dynamic changes to the table of advertised objects, and in their use of pre-defined trader paths. These limitations arise because implicit advertisement

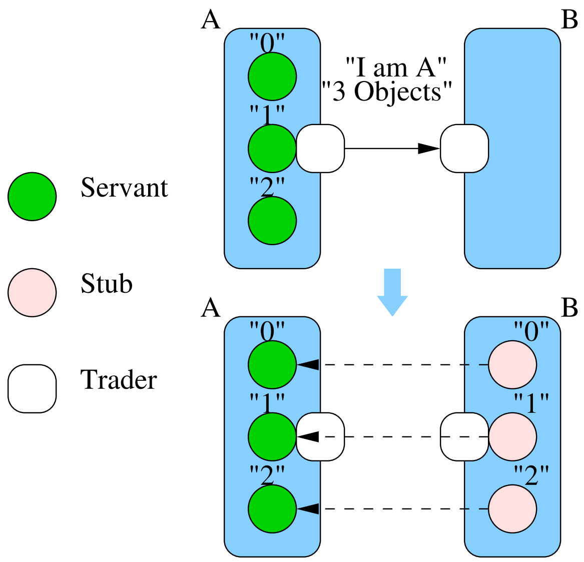


Figure 8.4: Several objects being advertised and imported

protocols must re-set objects' keys from a pool of well-known keys, and because the trader paths used for such objects are based upon these keys. Support for removal of advertised objects implies re-use of these well-known keys, which poses problems since keys are defined by the Koala ORB to be unique, allowing clients to cache information about them indefinitely.

Although static advertisement is sufficient for most simple appliances in practice, some appliances, most notably personal computers, require dynamic advertisement to operate effectively. Dynamic advertisement is implemented by taking advantage of the query propagation characteristics of the Eucalyptus trader specification. The export trader in this case is a standard Eucalyptus trader. Clients may therefore add and remove objects as required, at arbitrary paths within the trader. The only distinguishing feature of the export trader from any other is that it is the only object advertised by its host appliance. Queries made to the import traders of nearby appliances will propagate trivially into export traders implemented in this way, allowing clients to discover the objects they contain. Objects discovered in this way are said to have been *published*, to distinguish them from those discovered by advertisement. Figure 8.5 shows an export trader publishing objects to a client. Queries made by the client will pass to the remote trader, allowing remote objects to be discovered.

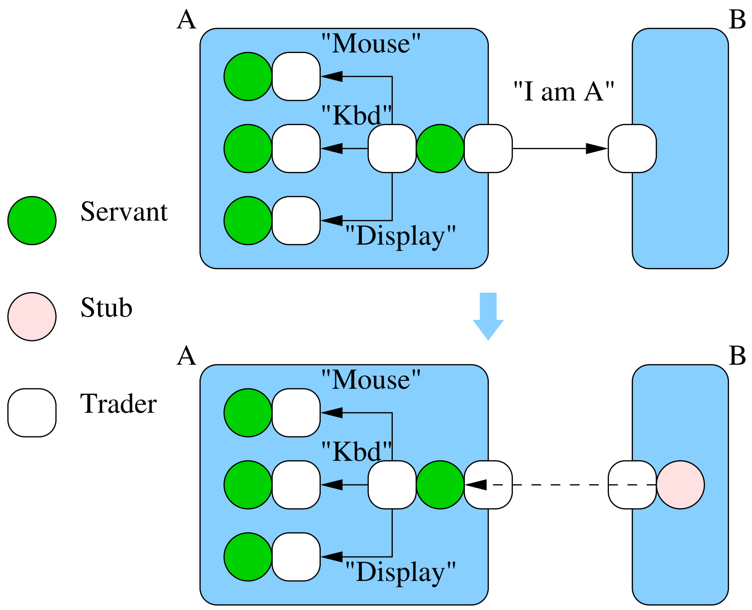


Figure 8.5: Objects being published by a dynamic export trader

Because published objects are returned to clients only in response to queries based upon the interfaces they support, recipients need not contact the objects directly to verify that they support those interfaces. This makes publishing of objects more efficient in many situations than implicit advertisement, since implicit advertisement protocols convey no interface information to clients. Advertisement can be improved in this respect, for small numbers of advertised objects, through the use of *explicit advertisement*.

Explicit Advertisement

Implicit advertisement protocols convey sufficient information to clients to allow them to imply the existence of objects with particular keys. For many simple appliances, the appliance's existence is sufficient information in itself. Because objects are not explicitly described in the advertisement protocol, however, no interface information or other per-object details can be provided. Clients must therefore contact implicitly advertised objects before manipulating them, to verify that they support the required interfaces.

Explicit advertisement allows interface information to be provided within the advertisement protocol, eliminating the need to contact objects directly. When operating over PEN, explicit advertisement packets are added to beacon payloads. Each packet contains an object key, one or more interface names and a sequence of hash values. The client ORB caches the supplied interface names and hash values. Figure 8.6 shows this process.

If a client subsequently attempts to verify that an object provides one of the named interfaces then the query will trivially succeed, without requiring the object to be contacted. If the required interface is not one of those named then the list of interfaces hashes is considered. The hash sequence contains an entry for each interface that is supported by the object but not named directly in the advertisement. If the hash of the required interface name is one of those supplied in the advertisement, then it is probable that the object supports the interface. The ORB will contact the object to verify this. If the required hash is not present then the object definitely does not support the interface, and so the operation fails without waiting to contact the object.

Ideally the full list of supported interface names would be supplied to clients in advertisements. Because advertisements are contained within broadcast beacon datagrams, however, their size is extremely limited, since the larger the datagram, the lower its likelihood of successful transmission. The Maximum Transmission Unit (MTU) for normal PEN datagrams is around 250 bytes. Broadcast datagrams should ideally be much smaller than this, to increase their chances of being successfully received by other appliances.

Explicit advertisement is used to augment implicit advertisement protocols with interface information. Appliances which publish objects using a dynamic export trader, for example, implicitly advertise the trader to clients. They will usually use explicit advertisement in addition, to inform clients that the advertised object is indeed a trader. Explicit advertisements may also be included for objects published by the export trader, to similarly optimise access to them.

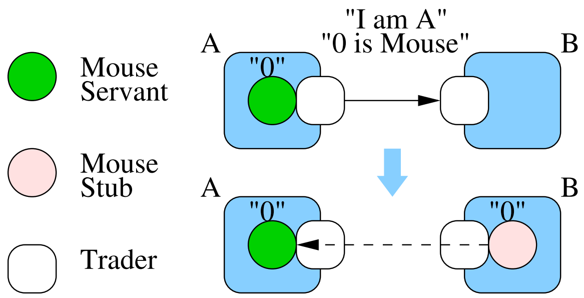


Figure 8.6: A wireless mouse advertising the Mouse object

8.3 Context

The trader mechanism allows applications to discover locally available objects providing particular interfaces. When combined with availability information provided by the Koala ORB, this is sufficient to support context-aware applications in ad hoc environments. Appliances are limited to context-aware behaviour based on their own surrounding environment, however, and cannot easily access the context of remote objects with which they are interacting. The characteristics of RF communications pose further problems for proximity-based heuristics, since an appliance that is contactable via radio need not necessarily be visible to the user.

Eucalyptus addresses these issues through the `Component` and `Domain` interfaces, which provide further structural information to distributed Eucalyptus systems in a standard way.

8.3.1 Components

As has been commented upon, Eucalyptus can make proximity based context information available to applications via the import trader mechanism, with additional support from the underlying Koala ORB. Although applications can use traders to discover objects based upon their local context, they cannot easily discover objects relevant to remote parts of the system. A CD player, for example, can discover speakers near to it but cannot necessarily discover speakers near to the user's remote control handset.

Eucalyptus addresses this issue by defining a class of object, termed *components*, which provide access to their home import trader. This will usually be the import trader of the component's host appliance. Components typically correspond to functional parts of an appliance, and constitute the majority of objects an appliance will publish.

In addition to providing access to their context, components support a pair of human-readable *annotations*. These take the form of one read-only and one read-write string attribute. The former annotation provides information regarding the kind of component, such as "Hi-fi" or "Speaker", whilst the latter allows the user to add an extra description to a component, such as "Living Room TV Tuner" or "Bathroom Lights". By combining the two annotations, components may be described to users and similar components may be distinguished by their user-specific annotations.

8.3.2 Domains

Although Eucalyptus is designed to support ad hoc interaction between distributed components, reasonably static environments such as offices or homes can still benefit from being pre-configured by an administrator. Pre-configuration can be used to avoid many of the problems of ad hoc operation, in particular those of the proximity context heuristic provided by RF systems.

Infra-red wireless systems, for example, can use a similar contactability metric to obtain a list of nearby devices from which to determine their context. Because infra-red signals behave similarly to visible light and will not penetrate walls, doors or other solid objects, context obtained in this way will relate directly to what is visible to the user. While this makes it difficult for infra-red systems to provide continuous connectivity while concealed, or while roaming through cluttered environments, it means their notion of context is more intuitive when applied indoors. RF systems fare better when providing connectivity to backbone services, but can provide erroneous results if used to deliver context information, because they can communicate through solid objects.

The context-aware behaviour we would like Eucalyptus to exhibit is that only appliances within the *user's* context should be discovered. In practice, this is most closely matched by the line-of-sight scope of infra-red, combined with per-user preferences regarding remote services such as email or pager services. Although Eucalyptus itself could easily be made to operate over infra-red networks, it is desirable that the system operate as expected when used with RF networks such as PEN, too. Additionally, both the infra-red and RF networks for which Eucalyptus is designed are short-range. This allows them to provide context information but also limits the area over which they can discover relevant components. Repeater nodes could be used to tackle this, by watching for nearby appliances and advertising them on their behalf to others using multihop techniques. Although repeaters would solve the range issue, they would reduce the granularity of the system's context awareness, since discovery operations would be more likely to retrieve components hosted in other rooms.

The *domain* abstraction allows mobile clients to perform discovery operations based on logical context information, as opposed to direct observation of their local environment. At the same time, such devices may opt to disregard domain services and instead perform their own discovery by some other means.

Pre-configuration

Domains are pre-configured groups of associated devices, usually arranged into some consistent logical hierarchy, arranged to mirror the layout of a building for example. Each domain provides its full hierarchical name, and access to a trader containing all objects registered with it. An administrative user, perhaps the occupant of an office, can therefore register the primary component of each appliance within the office with the office's domain.

Access control restrictions may be applied to the trader's contents, to prevent unauthorised clients from altering the registered contents, or from using the domain to discover objects without permission. As will be discussed later, domains may also be made to assume ownership of components registered with them, allowing access control to particular appliances to be centrally administered.

Domains may equally be applied to groups of mobile devices, perhaps those associated with a particular user. This allows the user to "discover" components relevant to them, as well as those relevant to their current context. A user could request a nearby terminal and use it to display their email, for example, by selecting the required components from the local domain and their personal domain respectively.

Using domains, context-aware discovery operations can be constrained to exclude false results, or widened to include appropriate results that would not be available through more ad hoc means. Because devices can always revert to ad hoc operation, the domain mechanism does not reduce the flexibility of Eucalyptus — it instead optimises its behaviour in a particular common situation.

Context-awareness

In addition to supporting discovery of components that are part of a logical grouping, the domain mechanism also provides for more general context-aware operation. Many applications will wish to alter their behaviour according to their user's context, for example, but do not require access to locally available components. Such applications can obtain context information by monitoring the locally available components, without actually using any of them.

Applications which must adapt to the user's context can benefit from the presence of pre-configured domains, since they provide a better-defined notion of context than proximity-based methods. Applications can monitor the currently selected domain and activate context-specific actions accordingly. A PDA, for example, may be configured to synchronise with desktop email clients when at home or in the user's office, and to work offline at all other times. The same device may also log the user in automatically whenever they enter their office.

Clients roaming through pre-configured environments will often be aware of multiple locally available domains, in addition to any domain of which they are a member themselves. Eucalyptus must determine a suitable current domain for the purposes of context-aware operation, even if this is the case. The following heuristics are applied, depending upon the number of domain names discovered:

- *No Domains*. If no domains are available locally then the appliance will fall back to ad hoc operation, with no current domain set.
- *One Domain*. If all local appliances are part of the same domain then the appliance will automatically change the current domain to match.

- *Multiple Domains, one dominant*. If multiple domains are available locally, but one is particularly common then the device will assume it to be suitable. The threshold used by the heuristic to determine "dominance" in this way is configurable by the user.

- *Multiple Domains, common prefix*. If many local appliances belong to distinct domains, none of which are dominant, but all share a common prefix then the common prefix is assumed in place of their actual domain. The heuristics are then re-applied to the list of available domains, often resulting in the common prefix becoming the dominant domain. A mobile device which discovers domains for several rooms in an office building, for example, can safely assume that it is indeed within the office building, even though it cannot determine precisely which room it is in.

- *Multiple Domains, no common prefix*. If many domains are available locally and neither of the other multi-domain heuristics apply then the decision is instead left to the user, and the currently active domain is retained.

At all times, the user may opt to override the automatic domain-tracking behaviour and select from the available domains manually. If appropriate thresholds are set then the heuristics outlined above tend to provide good results in office and home environments. More complex heuristics may be applicable in other environments. In particular, it is possible in the presence of Active Badge or similar services to obtain the current domain from infrastructural services, and to then use it to discover local components.

Domain Membership

Appliances capable of participating in domains in their own right publish an object implementing the `Domain` interface, which provides the domain's name and trader. Ideally, each appliance should maintain a cached directory of all other appliances within their domain. Poorly equipped devices incapable of caching domain information may instead return a reference to a domain trader hosted by a better equipped member of the domain. Appliances which are part of a domain with no well-equipped devices simply make the name of the domain available to clients but do not provide a trader implementation — clients may therefore benefit from well-defined context information, although the scope of discovery operations will still be limited by the communications range.

Because domains will usually be replicated across multiple appliances, such as PCs, TVs and hi-fis, clients may dispatch queries to them in parallel, relying on the fact that they will have the same contents.

Like many other Eucalyptus components, domains must be protected from malicious use, or from being misconfigured as a result of simultaneous conflicting requests. The following section discusses access control both in the context of domain configuration, but also for more general components.

8.4 Access Control

Many Eucalyptus components may be accessed simultaneously by more than one client without risk of conflict. Components such as sensors, whose outputs are dependent upon environment conditions, will not be affected by client queries and may therefore be re-used without difficulty.

Components whose internal state may be modified by clients require special consideration in environments with multiple potential clients. A lamp providing an on/off interface to clients, for example, must cope with conflicting requests for state changes from clients. If the lamp is intended to be controlled by a particular switch then it should ideally accept requests from that switch and not from other clients. At the same time, it should be possible for a user with a remote control to temporarily override that relationship and control the lamp itself.

8.4.1 Devices

Eucalyptus provides a standard reservation interface to support conflict resolution between clients. In their simplest form, implementations of the `Device` interface enforce a mutual-exclusion policy upon clients, allowing only one client to reserve and manipulate the device at any time. More complex implementations can provide the same interface but allow multiple clients to reserve the device simultaneously, using appropriate conflict resolution.

Configurable components are typically reserved for use by clients through the generic `Device` interface. Devices may be reserved for exclusive access and later released. Reserved devices may also be loaned to other clients. Mechanisms are also provided to ensure that devices are released when they are no longer required, whether this is because the client is no longer active, or because it has moved out of communications range. Tracking of *ownership* to support automatic release of devices is discussed in Section 8.4.6.

Devices are reserved through the `Reserve` method, which first verifies that the device is free for use, before returning a new object reference for it. The client may then access the device via the returned object, with the guarantee that its actions will not clash with those of any other client.

To allow clients to filter out devices which are unavailable due to having been reserved previously, each device provides the `Reserved` attribute. Since devices may be watched for changes to their attributes, clients may watch reserved devices in order to discover when they are released. This can be used to maintain traders containing only unreserved, available devices, for example.

Precisely how the device manages multiple clients internally is unimportant to clients. Most physical devices will simply store a unique identifier referring to the client currently authorised to access them, and return the original object reference from the `Reserve` method. When the object's `Release` method is invoked, the stored identifier is erased and the object is once more available. Only one client may therefore use the device at any time. Unambiguous and secure identification of clients depends upon authentication support from the Koala ORB, which is described in more detail in Section 8.4.5.

An alternative approach would be for the device itself to resolve conflicting requests from clients internally, allowing several clients to access it at the same time. Using an exclusive access mechanism, an intermediary must instead be used, between clients and servant, to perform this operation. The differences between the two methods are described in the following subsections.

8.4.2 Implicit Arbitration

Implicit arbitration requires each device to contain logic to resolve conflicts between requests from clients. An actuator with a simple on/off interface, such as a lamp, could receive a request from one client to turn on, while another requested it to remain off. An in-built arbitration function would consider the states requested by each client, and calculate the actual state to change to.

A suitable arbitration function for a household lamp might be a simple boolean *OR* of the requested states. The lamp would therefore stay active while one or more of its clients requested that it do so. If one of the lamp's clients was a light sensor, however, then it may be reasonable for the sensor to override other clients when it is sunny and the extra light is unnecessary.

Because of its inherent lack of flexibility, implicit arbitration would be unsuitable in many other environments. It is not clear, for example, what the most natural scheme to use when connecting multiple audio sources to a single output is. Should the streams be mixed, or switched between? While implicit arbitration seemingly simplifies systems by removing the need for clients to enter into negotiation with a component prior to using it, it works well only in environments for which there is only one sensible behaviour. Sufficient ambiguity exists in the definition of what is "sensible" that it is generally bad practice to hard-code arbitration functions based on a component's assumed target environment.

8.4.3 Explicit Arbitration

By convention, Eucalyptus expects that most devices will provide only the basic device implementation, enforcing mutual-exclusion between clients. This requires that *explicit arbitration* be used if multiple clients must be allowed access to a device simultaneously. Explicit arbitration places a proxy component between the device and its clients. The proxy provides an identical interface to the original device, but does not enforce mutual exclusion via the device interface. The proxy accepts requests from multiple clients simultaneously, and resolves conflicts in some way, before manipulating the device accordingly.

Explicit arbitration allows particular systems of components to be tailored to their environment without requiring devices to be reprogrammed or replaced. Libraries of suitable arbitration functions can be provided by manufacturers, or obtained from third parties. Proxies may be cascaded if required, to compose existing behaviours to better suit particular applications.

Arbitration functions can be hosted by an appliance other than the clients or servant, at the cost of increased latency. Ideally, arbitration should be performed on the appliance that hosts the device itself. Devices will often provide a suite of in-built arbitration functions specifically for this purpose. Alternatively, the host appliance could support the dynamic addition of new code to the device, allowing new arbitration functions to be downloaded from a manufacturer or third-party. Regardless of whether or not arbitration is performed on the appliance that hosts the device, it should remain a logically separate part of the system's operation. Clients may then use externally hosted arbitrators if those supported locally are not suitable.

8.4.4 Configurable Local Arbitration

Byte-code based language runtimes have matured to a stage where even relatively power-starved and tiny electronic devices can support them [11]. Byte-code interpreters may be embedded alongside the core Eucalyptus functionality in configurable devices to allow arbitration functions to be loaded dynamically as required, independent of the microprocessor used by the appliance.

Use of dynamically loaded code is appropriate only where the time spent delivering the new code is likely to be compensated for by the performance improvement. In the case of arbitrator functions, which will usually remain in place for prolonged periods, this tradeoff will usually be a reasonable one. Many arbitrator functions, such as the Boolean logic functions described later in this chapter, may be encoded in a sufficiently compact manner to fit into a single datagram. Even very short interactions may therefore benefit from code migration.

8.4.5 Authenticating Requests

Both devices and the proxies used to arbitrate access to them must be able to reliably and securely authenticate the clients which access them. This may be achieved trivially by returning a separate object reference to each client, with a unique identifier embedded in the object's key. The object reference then behaves effectively as a capability allowing the client access to the device. Care must be taken to avoid attacks from eavesdroppers, and to ensure that the identifier space is sufficiently sparse to avoid brute force attacks.

A simpler and more robust technique is to use unique identifiers assigned by the ORB. While processing a request, the servant may query the identity of the current client and act accordingly. Identifiers are pure names, used solely to distinguish clients. It is the responsibility of the ORB to ensure that clients are properly authenticated, to avoid malicious clients invoking methods using other clients' identities. The ORB must therefore authenticate clients in some way. Authentication may then be handled at a more appropriate level of the system, perhaps using techniques similar to those proposed by Stajano [67].

8.4.6 Ownership

The previous sections have discussed authentication of remote clients and resolution of the potentially conflicting requests that they might make upon devices. While reservation of devices for exclusive use can be used to avoid the issue of conflicting requests, it also enables denial-of-service attacks that would not otherwise be possible. Without suitable preventative measures, malicious roaming clients can reserve devices at will, rendering them inaccessible to other users. Similar attacks may be made against proxy devices provided by arbitrators, consuming system resources and leaving them unavailable for use by other clients.

Ideally, clients should always release reserved devices once they no longer require access to them. Release might be triggered by the user closing a graphical user interface they have been using to access a device, or by an automated control action being removed from the system.

Since we wish to support ad hoc interactions between appliances which are potentially highly mobile, we must cater for the case in which the client is moved out of communications range of devices it controls. When reserving devices, clients must therefore supply a reference to an *owner* object. Devices monitor this object using the ORB's *watch* mechanism and may be configured to automatically release themselves when it is no longer considered *accessible* or *local*, or when it ceases to exist. Other Eucalyptus components may discover the owner of a device by querying the object's `ReservedBy` attribute.

8.4.7 Delegated Ownership

The device mechanism makes it possible to express relatively short-lived ownership of components, usually for the duration of a transaction or user interaction. In its basic form, the device mechanism cannot conveniently support more long-term configuration of devices. In particular, it is not possible for a malicious user to have their ownership of a device revoked by the device's real owner.

To support administrative configuration of devices and revocation of access if required, the ownership of a device may be temporarily delegated to third parties by their owner. The device's real administrator first assumes ownership of the device, then requests that it accept further reservation requests. The delegated device resets its `Reserved` to False to indicate to clients that it is available, but leaves its `ReservedBy` attribute unchanged. Attempts to reserve the device will succeed, usually allowing the client full access to the device.

While a device is loaned to a child client, the parent may not access the methods or attributes of the device, with the exception of those provided by the `Device` interface. When a parent owner chooses to revoke a child's permissions to access a device, the device must either erase its internal state, or revert to the state it had before being reserved by the client, in order to avoid misuse of the revocation facility by a malicious parent.

Delegated ownership can be used to help avoid trivial denial-of-service attacks by allowing the administrative owner of a device to specify a minimum criterion for the automatic release of the device. The administrator may specify, for example, that the device cannot remain reserved if a client moves out of direct communications range. Equally, the original owner could be allowed to specify more general access control settings for the device to observe. Subsequent clients to which ownership is delegated would be permitted to further reduce access to the object for the duration of their lease.

8.5 Automation

This chapter has discussed interfaces which make the underlying technologies described in previous chapters easier for applications programmers to exploit. The mechanisms described up to this point have not addressed the issue of stand-alone operation of Eucalyptus systems without input from their users, however. Automation of simple tasks is one of the more controversial goals of ubiquitous computing projects. While researchers tend to agree that contemporary systems require too much manual intervention, few can agree on exactly where the human should be able to sit back and let the computer take over.

At one extreme, we might imagine systems in which computer systems infer their users' needs and wants from their behaviour, and from the state of the world around them. The user would then have to take matters into their own hands very rarely, when the computer system had no way to help them.

The problem with this kind of system is that the computer must possess an equivalent degree of understanding of the user's context to the user themselves. This requires nothing short of full artificial intelligence — the computer would effectively operate as an electronic replacement for a human servant. Assuming such technology were to become available, it could still not reliably provide full automation for even a fraction of the user's daily tasks, since to do so would not only require human-like intelligence, but awareness of the user's feelings and desires.

In the absence of an artificial intelligence identical in all respects to the user's own mind, we must cope with the fact that attempts at intelligent automation will often make the wrong choices. This means that systems which attempt to infer the user's requirements and automatically configure themselves will quickly become annoying. A more practical approach is to infer the user's requirements and to use them to suggest suitable actions to the user in a non-intrusive way. If the user does wish actions to be performed without their intervention then the reasoning behind such actions should always be made available.

The Eucalyptus system provides interfaces to support automation but does not stipulate the method used to configure it. The interfaces defined are suitable for direct use by users via a suitable interface, but may equally be manipulated programmatically. The relationships between components and those they control or are controlled by are made explicit by Eucalyptus' interfaces, making it easier for users to understand the causal relationships between inputs to the system and its resulting actions. The following sections describe the core models for configuration and automation support provided by Eucalyptus.

8.5.1 Connections

One of the simplest yet most powerful automation models is that of connecting components together. Component connections are instantly familiar to users, because they closely mirror the behaviour of physical connections between appliances. When installing new appliances in the home, the first step involved is invariably to connect the appropriate wiring between them. Speakers must be physically connected to an amplifier and the amplifier to one or more audio sources, such as CD players, in order to be useful.

Once appliances have been physically connected to each other, the composite system can be used without further configuration. When the user requests that a hi-fi play a cassette, the sound will be produced by the speakers attached to it. Similarly, in a virtually connected system, once connections have been made, the system can operate without requiring further input from the user. Just as physical connections may be broken and re-established with different appliances, virtual connections may be created and destroyed as required. Virtual connections, however, may be manipulated more easily than physical ones, and may be implemented over either wired or wireless networks while providing the same semantics.

In their simplest form, Eucalyptus connections are made between pairs of *source* and *sink* components, such that each sink is connected to only one source component. Connections may be uni- or bi-directional, as required. Simple sensors, for example, will usually support only uni-directional connections to read data from them, while connections between remote controllers and appliances will often be bi-directional, to provide feedback to the user.

Eucalyptus imposes the restriction that each sink may be connected to at most one source, while a single source may supply data to multiple sinks. This allows several clients to monitor an individual sensor using the connection mechanism, and makes it easy for users to determine where a particular sink is obtaining data from. Because bi-directional connections are, in effect, pairs of uni-directional connections, they must connect exactly two endpoints.

Shared Sources

Ideally, any source should be capable of being *shared* by any number of other components. This is a reasonable expectation for most sources because the semantics of multiple connections to a single source can be tightly specified. In general, we simply require that all clients receive the same data from the source at any particular time. The source controls the rate at which data is supplied, avoiding the need for synchronisation algorithms.

Multiple speakers could be connected to a shared CD player, for example. The resulting effect will be that all of the speakers play the same audio stream. Note that this does not imply that the speakers are well-synchronised, simply that they are supplied with audio data simultaneously. Further processing of audio streams will usually be required to maintain true synchronisation, especially if the speakers are distributed over a large network, or accessed via multihop wireless, for example.

If each set of speakers were required to play different tracks from a CD, however, then the player would need to provide a distinct audio source component for each set of speakers to connect to[1]. In effect, the single physical CD player appliances would appear to the system to be a set of distinct logical CD player devices.

Connection Compatibility

The connection mechanism may be used to support both data streams, such as audio or video, or control requirements such as a switch or sensor manipulating a lamp, for example. Because the connection interface has well-defined high-level semantics but does not stipulate precisely how connections are handled by lower levels of the system, it may be used for both kinds of connection-oriented interaction, plus many more.

While the potential for diverse source and sink component types through a common interface makes it possible to build general-purpose user interfaces to manage connections, it also introduces much scope for confusion of non-technical

[1]This assumes the existence of CD player hardware capable of playing multiple tracks simultaneously, of course.

users. A connection-management interface, for example, should not allow the user to attempt to connect an audio sink to a switch, for example.

The `Source` and `Sink` interfaces are derived from a common `Connection` interface, which provides the `ConnectionType` attribute. Source and sink can use the attribute to verify each other's types when they are connected. More importantly, clients may check the connection type of components in order to avoid attempting to connect incompatible components in the first place.

Although this solves the connection typing problem, it is not ideal, since the client must contact source and sink to verify their types before continuing. Contacting the two components is a redundant operation, in practice, since their types will never change at run-time. Ideally, a component's connection type should be conveyed to clients when they first receive a reference to the component.

Constant Attributes

One approach to avoiding the need to explicitly contact `Connection` components to determine their type is to have them dispatch unsolicited attribute-state events to the client the first time it contacts them. The notification would contain the value of the connection type attribute which could then be cached by the client for later use.

Unfortunately, because the client is not aware that the connection type is an immutable attribute, it must watch both source and sink in order to be sure that attribute values it caches are kept up to date. This seems an unreasonable burden when dealing with values which cannot change.

More satisfactory results can be obtained if by permitting interface definitions to be annotated with extra optional tags in addition to those specified by OMG IDL. If a `const` pragma is provided to indicate that an attribute is immutable then a suitably equipped ORB can optimise its behaviour to take advantage of the fact. Implementations which do not recognise the pragma will simply ignore it and use the standard behaviour.

ORB implementations supporting the `const` pragma will embed the values of any immutable attributes in object references they produce. Implementations which do not support immutable attributes simply omit this extra data segment from their object references. Similarly, clients incapable of using attribute data segments can safely ignore them, with the caveat that they should ideally be retained so that they can be included in object references passed on to third parties.

The connection abstraction is commonly used to support the configurable arbitration of access to shared resources by third-party components. A simple subset of such arbitration components are those implementing Boolean logic, which are described in the following section.

8.5.2 Boolean Connections

Many control tasks can be expressed using boolean logic. A switch controlling a light, for instance, provides an output value which is either True or False. A light capable of being either on or off can be controlled by Boolean True or False values. Using the `Source` and `Sink` interfaces the two can be connected.

Components which do not deal directly with Boolean data can be adapted to do so in many cases. Light sensors, for example, will usually provide the light level as a calibrated value within a well defined range. By setting a threshold on the light sensor's value, it can be adapted it to control a Boolean lamp.

Boolean Components

Automated systems using Boolean logic are constructed from a number of different types of components.

- *Stateless* components accept one or more input values and provide one or more outputs calculated from them. For a given set of input values, a stateless component will always return a well-defined output value. Stateless components may also be termed *functional* components. Stateless components include the standard Boolean `and`, `or` and `not` operators.

- *Stateful* components accept one or more input values and provide one or more outputs. In addition to the supplied input values, some internal state is used in determining the resulting output. This state will usually be modified by changes to the input values. As a result, there is no linear correlation between sets of input and output values. Stateful components include flip-flops, counters and latches behaving in the same way as their digital electronics counterparts.

- *Sensor* components provide one or more outputs and accept one or more non-Boolean inputs and zero or more Boolean inputs. Sensor components provide Boolean inputs to the system based on external state. A Sensor component could be constructed, for example, to monitor a CD player and to toggle its output state every time the current track is changed. The CD player itself might provide a Sensor component directly, to indicate whether or not it is playing a track. Other examples of Sensor components include clocks, timers and thresholds for physical sensors.

- *Actuator* components accept one or more Boolean inputs and provide one or more non-Boolean outputs and zero or more Boolean outputs. Actuator components provide a way for collections of Boolean components to manipulate their environment. A lamp, for example, might present an Actuator component which controls its state. A lamp attached to a dimmer might require a percentile input, in which case it could be controlled by Boolean logic only via a suitable Actuator component. The Actuator component would translate the Boolean True and False values to preset dimmer states. These are examples of stateless Actuator components. An Actuator component which caused a CD player to skip to the next track whenever the Boolean input to it became True would be considered stateful.

One caveat in the implementation of Boolean components for Eucalyptus is that their present state is delivered to clients not as a native `boolean` value but instead using an `unsigned long` counter. Whenever the state of a Boolean component is to be changed, this counter is simply incremented. The Boolean value represented by the counter is obtained by considering only the lowest order bit of its value.

The Koala event distribution model assumes that clients are interested only in an object's current state. There are situations in which this assumption can significantly alter a system's behaviour from what the user might expect. If a click button presents a simple Boolean value as its output, for example, then several clicks may be merged to produce a single event. Clients will therefore miss clicks in a non-intuitive way, confusing the user. By conveying Boolean state information using counters, recipients can take into account how many state changes have occurred between events, or may simply use the lowest order bit directly if only the current state is important. Boolean actuators will usually consider only the current value of a Boolean, while functional components such as `and`, `or` and `not` will propagate the number of state changes to their clients where appropriate. A slide-show being displayed by a PC and controlled by a click button, for example, will also consider the number of state changes between each received event, so as to skip the correct number of slides.

To illustrate the use of counters to convey Boolean state, consider the case of a switch and light sensor, connected via the `or` operator, to a lamp. With both sensor and switch set to `false`, the user flicks the switch, but immediately flicks it back. The switch's output value will therefore increase by two, and the effective Boolean value will remain `false`. The `or` operator will receive an event indicating the switch's change in state, and will calculate how many times its own output would have change as a result. It will therefore increase its own output value by two as well. Had the sensor been set to `true`, indicating that it was dark and that the light should be on anyway, then the switch's state would have no effect on the `or` operator's output, and so the `or` gate's output counter would not change.

Boolean Scripting

Components in a Eucalyptus system may be connected via Boolean components in order to perform required tasks. These connections may be constructed directly by the user, via a GUI for example, or introduced programmatically using a simple scripting language. Both approaches rely on the availability of factories for the standard Boolean components.

Highly mobile devices should support a basic set of Boolean components locally in order to allow them to operate independently of infrastructure. Devices residing in pre-configured, relatively static environments such as homes or offices can instead benefit from the availability of central component factories.

Because the semantics of the connection interface are well defined, in particular in the case in which multiple sinks are connected to a single source, it is possible for Boolean scripts to re-use components created as part of other scripts. This reduces the network bandwidth required to support the execution of the script. It also allows pre-configured Domains to make a set of standard Boolean values available to local clients. A Domain managing a meeting room might provide a “Meeting in progress” Sensor component, which would use Active Badge information, audio sensors or thermal sensors to determine when the room was in use.

8.5.3 Fuzzy Logic

Boolean logic provides a well-defined and intuitive way to configure collections of simple components. It is limited, however, in the flexibility with which systems may be configured. While a light may be switched on or off by a switch, a configuration expressible using the Boolean model, it may instead have its light level controlled by an analog dial.

Using fuzzy logic, configuration of analog components can be supported, such as light or volume levels. Instead of dealing with Boolean True or False values, fuzzy logic manipulates percentile values. Similar primitives may be used to connect sensors and actuators to those used with Boolean systems. Further primitives are also available which make more explicit use of the features of fuzzy logic. An example of such a primitive is the `average` operation, which accepts one or more inputs and provides their average as its result.

Boolean `and`, `or` and `not` operations, for example, are implemented as minimum, maximum and complement, respectively. A hi-fi system might be configured to play audio if a user has requested it *and* they are in the room. The request to play audio will be a percentile measure of how loud a sound should be produced. The second parameter will indicate the certainty of the system that the user is in the room. Using this rule, the volume of the audio actually played will vary as the user moves in and out of the room, or as they change the requested level.

8.6 Summary

In this chapter we have described the standard interfaces and structures defined by the Eucalyptus architecture to support discovery and configuration of components hosted amongst groups of appliances. These interfaces are intended to provide a minimal common functionality sufficient to support most monitoring, control and automation applications. By basing such interfaces on the Koala ORB, Eucalyptus applications need not be concerned with low-level communication details. This provides similar portability and interoperability benefits to the use of CORBA with big-box computer systems but also allows the use of underlying network primitives to be tailored to resource limited environments without affecting application code.

Of particular importance are the `Trader` and `Device` mechanisms. The `Trader` mechanism provides a flexible way for appliances to publish the components they host, which is at the same time simple enough for even extremely trivial appliances to implement. The `Device` interface provides a simple resource reservation mechanism sufficient to support human interactions with appliances.

Chapter 9 describes a number of Eucalyptus appliances constructed to demonstrate the architecture's operation. Applications of these appliances are also described, which exercise different features of the architecture, and which show the ease with which appliances designed for one purpose may be re-used for another.

Chapter 9

Example Applications

Several appliances and applications have been developed using the Eucalyptus system, each illustrating different aspects of the architecture's design. Of particular importance in motivating the use of a common architecture across many applications is the ability to re-use appliances for different purposes.

Table 9.1 lists the appliances constructed and summarises their characteristics. All of the constructed appliances transmit regular beacon datagrams, as indicated by the *Beacon Interval* column. Some appliances may broadcast further data in optional transmission slots, indicated by the *Opt TX* column. The *RPC* and *Events* columns indicate whether the appliance's hosted components accept RPCs and how they deliver event notifications, respectively. Finally, the *Device* column indicates whether the hosted component may be reserved for exclusive user by clients.

9.1 Environmental Monitoring

Sensor appliances have previously been described which may be used both to control suitable actuators automatically and to provide environmental sensing for other applications. Temperature and light level sensors have been developed. These sensors embed their readings in the beacon payloads, for delivery to clients.

Because they are wireless and battery-operated, Eucalyptus sensors can be deployed precisely where readings are required. Sensor data is collected by software

Appliance	Beacon Interval	Opt TX	RPC	Events	Device
LDR	10s	2s	No	B'cast	No
Thermistor	10s	2s	No	B'cast	No
Lamp	10s	2s	Yes	B'cast	Yes
Fan	10s	2s	Yes	B'cast	Yes
Clicker	10s	N/A	Yes	RPC	Yes
Button	10s	N/A	Yes	RPC	Yes
Tab	4s	N/A	Yes	RPC	Yes
Store	10s	2s	Yes	B'cast	Yes
Serial	10s	N/A	Yes	RPC	Yes
APIDgate	10s	N/A	Yes	RPC	Yes

Table 9.1: Capabilities of prototyped Eucalyptus appliances

running on PCs attached to the office's wired network, via R-Link gateways. Environmental information for any part of a building accessible via R-Link can be collected and displayed. The system's range could be further extended through the use of multi-hop routing.

Ad hoc monitoring of the local environment can also be performed by low power wireless devices roaming around the office environment. Sensor nodes already in use by an office-wide environmental monitoring system can be accessed simultaneously by mobile clients without impacting power consumption or increasing radio traffic, owing the beacon event delivery method.

9.2 Environmental Control

Lamp and cooling fan appliances have been constructed which provide Eucalyptus interfaces using PEN. Both types of appliances may be reserved by ad hoc clients and manipulated directly. Clients may also monitor the current state of such appliances. Similarly, office-wide environmental management software can manipulate lamps and fans explicitly using the R-Link system.

In addition to direct manipulation, both types of actuator may be slaved to follow the state of another component. A threshold may be applied to a light sensor's reading, for example, and the resulting Boolean used to determine whether a lamp should be on or off. Similarly, one lamp may be slaved to another, allowing both to be controlled by a single switch or sensor without the need for multi-hop routing.

Slaving of environmental actuators allows them to continue to operate while the component to which they are slaved is available, regardless of the state of other parts of the system. Slaved appliances may be reconfigured by services hosted in an office's wired network, or in an ad hoc fashion by PDAs or other personal user interfaces.

9.3 Switch

Buttons and toggle switches have been built using Eucalyptus. Buttons present a simple counting interface to clients, with a single attribute that is incremented whenever the button is pressed or released. Switches perform a similar function with the difference that they toggle between their pressed and released states when manipulated by the user. Because this difference is purely mechanical, both kinds of button present the same Eucalyptus interface and they can be used interchangeably.

Because the Eucalyptus system's Boolean logic model is implemented using counters rather than native Booleans, buttons may be used as Boolean "sensors" in a Eucalyptus system. They may be used to control any Boolean actuators, including the lamps and fans previously described. They may also be combined via Boolean functional components with suitable sensors to provide more intelligent control over such appliances.

9.4 Personal Information Stores

The prototype PEN nodes upon which the Eucalyptus implementation is based provide enough RAM to store a user's personal information, such as their address book, to-do lists, diary and so on. Nodes have been constructed for this purpose, providing an *attribute store* interface similar to a simple file-system. Attributes are specified using (key, value) pairs. Values are transferred and stored using the

`Any` type, allowing the node to store data in arbitrary formats, while allowing clients to perform run-time type-checking to ensure that values they retrieve are appropriate.

The store provides a simple transactional model to clients, via the Eucalyptus `Device` interface. Clients must reserve the store before adding data to it. Once they have added or modified all the required values, they invoke a `Commit()` method which makes the changes visible to other clients atomically. If they release the store without invoking `Commit` then all changes are discarded. Several clients may reserve the store simultaneously to make changes. Only if contradictory changes are made to the same attribute will the commit attempt fail, raising an exception to indicate that the action must be retried.

Personal Information Store nodes publish a single object, providing the attribute store interface, as their only hosted component. Stores could equally well be hosted by better equipped appliances, in particular by PDAs, in which case the attribute store might be one of several hosted components. Users carry their Personal Store with them wherever they go. Other appliances such as office or home PCs, PDAs or mobile phones will discover the store and synchronise any appropriate information with it. The user's address book will therefore remain synchronised between their office and home PCs. If the user buys a new mobile phone then their phone-book will be uploaded to it automatically by their personal store. Because the store can communicate with nearby appliances via RF, the user need never directly manipulate the device — it can perform its function entirely passively if required.

In addition to allowing personal information to be carried with a user, a personal store provides similar benefits to synchronisation of appliances using a PC as a "hub". If all appliances with which the user interacts synchronise with their personal store then support for general peer-to-peer synchronisation and conflict resolution is not required. At the same time, any appliance the user carries which hosts an attribute store may be used as a backup hub, providing resilience to failure of the main store.

The implemented attribute stores use IDL structures derived from the VCard and VCal standards to store address-book and schedule information. The store could be extended to provide specialised read/write methods in addition to the generic `Any`-based versions, allowing clients to use static typing of such data and removing the need for typing information to be used at runtime. Data stored via statically typed interfaces remains accessible via the generic interface to allow content-neutral stores to be built, to which arbitrary data may be backed up, for example.

The operations described above pose important security issues, since personal information stores may be accessed via RF by any nearby appliances, without line-of-sight constraints present in free-space optical networks such as IrDA. It is therefore important that secure authentication and encryption be established between attribute stores and the appliances with which they will exchange data. Initial authentication can be achieved by having the user register a pass-phrase with the store which must then be supplied to other appliances to allow them to authenticate with the store. Alternatively, physical contact could be used to exchange key information between appliances, via metal contacts on their casings, for example. The Dallas Semiconductor "1-Wire" technology, used by their i-Button [11] products, could be used for this task, for example.

9.5 Wireless Console

Server rooms in large organisations often contain large numbers of rack-mounted PCs, routers and patch panels, and the required cooling apparatus. Although rack-mounted servers can usually be removed and a monitor and keyboard attached in order to access the console, this can be difficult and risks data loss. Because console access is only normally required when network access to a server is unavailable, it is highly likely that such a machine will already be in an unstable state. Console access is often required in situations where a machine has completely crashed, to view any error messages displayed during its final death throes which may provide clues as to the nature of the fault.

Eucalyptus appliances have been constructed which can be attached to any 9600 baud serial port on a standard PC. The serial appliance will read data from the port and cache up to eight Kilobytes internally, for later retrieval by clients via the PEN radio interface. Data may be provided via the radio interface to be delivered to the attached device via the serial interface. Both functions are performed via a single component published by the serial interface node. The `Serial` component supports both direct manipulation via RPC and a connection-oriented interface, allowing `Serial` components to be connected to exchange data directly between devices.

Servers with serial interface nodes attached can be configured to provide their console output and a login shell via their serial port, making both available via the radio interface. Client software running on other nearby PCs can be used to retrieve the most recent output from the server and to send commands to it. The same software can instead be run on a PDA, providing mobile wireless access to the same interface, simplifying simple maintenance tasks. The serial-to-serial connection feature allows a standard VT100 display to be placed nearby, and a wireless switch used to display the output from different servers as required.

9.6 Thin-client Tabs

Amongst the PEN devices developed by AT&T Labs are low power displays, based on the standard PEN prototype units. PEN displays include a bistable cholesteric display and three buttons. Monochrome data may be rendered and will be retained by the display even while it is not powered, allowing considerable power savings over conventional LCD displays in spite of the extra power cost while rendering.

PEN displays can be used as thin-client “Tabs”, in much the same way as PARCTABs, and slaved to a Tab manager hosted by a separate appliance. Unlike PARCTABs, PEN Tabs may be managed by any nearby appliance, rather than requiring the presence of Tab management infrastructure. A hi-fi, for example, could manage nearby Tabs and provide track information and selection functions to users. A television could use a Tab as a remote controller and could display information on the current programme on it.

9.6.1 Jukebox Controller

An audio jukebox for MP3 data has been constructed in software and hosted by a standard PC. The jukebox communicates with nearby Eucalyptus appliances via an R-Link gateway node attached to one of the PC's serial ports. Using the gateway node, the software mimics the behaviour of a "native" Eucalyptus appliance and publishes an audio source component. Similar software is hosted by other PCs within the lab, which publish audio sink components providing access to speakers attached to each PC. The resulting system is equivalent to those made possible by the Warren and HAVi architectures, with the addition of a wireless control and configuration layer.

Both hi-fi and speakers can be assumed to be mains-powered, reducing the need for power-aware protocols. The same embedded interfaces could also be used with portable audio equipment, however, in which case low power operation would be desirable. In an energy efficient home or office, mains powered audio equipment can disconnect itself from the mains power supply when not in use. Only the embedded PEN interface would remain active, using an internal rechargeable battery.

When the user wishes to access local audio equipment, they press a button on their Tab, causing it to enter a low power discovery phase. Because the discovery task is user-initiated, this will usually take the form of a single prolonged listen window, in order to discover all local appliances with minimal latency. The Tab will display a list of local Tab-manager components, adding items as they are discovered. The user can navigate the list and select a component. The speaker and jukebox appliances described above can host their own Tab-managers or the function may be performed on their behalf by a separate appliance. Well-powered appliances hosting Tab-managers may actively advertise themselves to nearby Tabs, removing the need for Tabs to perform a discovery phase.

The user interface presented by a speaker will allow its volume and current audio source to be viewed and modified. If the speaker is connected to a source then a shortcut is available through which the user interface for that source may be accessed. The jukebox user interface allows the current track to be viewed and a new track selected using similar methods to conventional hi-fi equipment. Figure 9.1 shows a Tab presenting an audio control interface.

Because the appliances involved are well-powered, they can disable PEN's power saving features in order to reduce the latency of the user interface. Reducing latency is especially important when using Tabs, because the responsiveness

Figure 9.1: Wireless audio remote controller

of the user interface is directly dependent upon the network delay. Tabs provide acceptable performance even when managed by low power appliances by using RPC- and event-chaining techniques. Tabs listen for a short period after notifying managers of user input events, allowing the user interface to be updated with minimal latency. Similarly, Tabs provide listen windows immediately following receipt of any request to render to the display, allowing several rendering operations to be received and performed in a single batch.

9.6.2 Centrally-Managed Tabs

Like PARCTABs, PEN Tabs may also be managed by services hosted within an organisation's wired infrastructure. Services run on standard PCs and are accessed by Tabs using the R-Link service. Wired network resources such as Active Badges, email and web servers can be provided to Tabs via such services with little effort.

A door-sign application built using this architecture allows each sign to display information on the usual occupants of a room and on their present whereabouts, for example. Signs can also display away messages, or may indicate that a meeting is in progress and that the participants do not wish to be disturbed. Users can navigate between the different information displays as required. Door-signs can be used to display information relevant to locations other than their own, but will revert to their default if left idle for a predefined period. The default information displayed by each sign can be reconfigured, to cope with relocation of signs or replacement of faulty hardware.

More advanced door-sign applications use the Eucalyptus architecture to provide context-aware features to Tab services. In addition to the GUI components it hosts, each Tab also operates its own import trader. This imposes no extra burden on the appliance, since an export trader must be hosted anyway to support discovery of Tab managers. To reduce the impact on power consumption of hosting the trader, it uses the radio interface's discovery features only when being watched for changes. By providing the import trader as a standard Koala object, accessible to remote clients, the Tab can support applications that use radio proximity as a context heuristic. If clients do not watch the trader, however, then its presence has no performance impact on the Tab.

9.6.3 Roaming

One criticism of the PARCTAB system was the inability of Tabs to operate outside the PARCTAB-aware office environment. PEN Tabs improve the situation by allowing any nearby appliance hosting a Tab manager to control them. The usefulness of Tabs may be further extended by allowing them to roam away from an office or home environment while continuing to access its resources.

Using R-Link, Tabs can access particular Tab managers they have previously encountered, to request their services, via any wired network. Given a ubiquitous backbone network such as the Internet, a Tab can be managed regardless of its location, provided it is within range of an R-Link gateway. Office services may therefore be used from home, via the Internet or a phone line. They may similarly be accessed by a mobile user via a PEN-equipped mobile phone.

Latency issues pose a problem for wide-area interactions with fully thin-client devices, motivating the use of higher level user interface systems such as the Generic Control Language [38]. An adaptive component-based GUI would ideally be used, allowing Tabs with differing capabilities to host GUI facilities ranging from basic bitmap rendering to full widget sets [20].

9.7 Summary

This chapter has described a number of simple systems constructed using the Eucalyptus architecture. Individually, few applications gain significant benefits from the use of Eucalyptus over proprietary protocols. More significant benefits become apparent when appliances and components are re-used by diverse applications. The simplest sensors, switches and actuators may be used freely by applications, in many cases simultaneously, as a consequence of the use of well-defined and standard protocols and interfaces. Eucalyptus applications can not only share components and co-exist, but may also be aware of each other when appropriate.

With suitable pre-configuration, for example, an alarm system could arrange to disconnect audio sinks from sources whenever an alarm occurred, to reduce the likelihood of the alarm being overlooked. When the alarm was over, the original audio configuration could be restored. Similarly, an automatic lighting system in a meeting room could be disabled whenever the room's projector was in use. In both cases, the two interacting systems are part of separate applications but, because they can co-exist and interoperate, they can be made to cooperate with minimal effort.

Chapter 10

Conclusions

This dissertation has presented an object-oriented architecture for control of low power embedded devices using wireless networks. This chapter presents a summary of principal contributions of this research and suggests areas for future work.

10.1 Summary

The Eucalyptus architecture addresses the basic issues of access control and naming in a lightweight but flexible manner. The architecture allows even very simple embedded sensors to interoperate seamlessly with full computer systems. At the same time, resource limited devices can cooperate as peers without support from surrounding infrastructure. This allows individual appliances to be re-used more easily by different applications, an aim shared with component-based software systems. Re-use of physical devices has the effect of amortising their cost over many applications, thus reducing it, and increasing the chances of commercial acceptance.

Eucalyptus demonstrates that embedded and low power systems can support high-level abstractions for distributed programming efficiently if those abstractions provide suitably versatile primitives. Eucalyptus uses the Koala object request and event broker, which provides a number of object-oriented communication primitives that are less susceptible to high network latencies than conventional RPC.

Clients may watch Koala objects for changes in their visible state using a native event mechanism. Separating event notification from the semantically distinct function of method invocation allows it to be implemented more effectively than would be possible with more common events-over-RPC systems. Events are linked intrinsically to changes in the state of objects' attributes, providing more tightly defined semantics than more general event systems, and allowing more effective optimisation of their implementation.

One of the most effective primitives provided by Koala is the lazy list. Lazy lists allow local data and pending results of asynchronous method invocations to be merged into a single list. Values are returned from lazy lists in the order in which they become available, greatly reducing the effects of high network latency while requiring only minimal, often cosmetic, changes to many applications.

Wide-area routing and service provision mechanisms have been described, based around a scalable distributed location service. These services extend the useful range of wireless devices beyond that possible with multi-hop technologies. They make it possible for the same wired network through which wireless data is routed to host services and to make those services available over a wide area. Central to the design of the R-Link system is the assumption that only explicitly routed requests should be passed to remote devices through the wired network, and that all other requests should instead be passed only to services hosted by the wired network.

A low power rendezvous and messaging system providing multiple primitives has been discussed. The R2-Layer illustrates the importance of tailoring the behaviour of protocols to the target application but also shows that this can be achieved without building new protocols for each application. Using a modular design and providing many primitives allows applications to make more appropriate tradeoffs between performance and power saving. Control and data flow between modules are separated, reducing the scope for redundant code and allowing many modules to be used without modification with different power management strategies.

10.2 Future Work

Although several applications and their supporting infrastructure have been constructed, the Eucalyptus system remains incomplete in several areas. Further work would be required if the architecture were to be deployed as a commercial product. Similarly, further technologies based upon Eucalyptus need to be developed if specialised application areas such as sensor fusion or intelligent automation are to be well supported.

- The Eucalyptus system has been implemented natively for the PEN platform in C, and for PC, Psion 5 and Compaq iPAQ platforms in Python. Although the Python implementation could in principle be adapted to support code migration, the language is unsuitable to be embedded in extremely tiny devices. Support for dynamic uploading of portable code to PEN devices is desirable in order to fully implement generic user interface components. This requires that a bytecode or script interpreter that is capable of operating in an extremely small memory footprint and requiring minimal processing resources be developed.

- If code can be transferred between devices in an ad hoc manner then further reduction in the effects of latency may be possible. Code to adapt the output of analogue sensors to a Boolean value based upon a threshold could be uploaded to a light, for example, removing the need for adaptation by a third party. The costs of code migration may outweigh the benefits, however. The feasibility of migration may be heavily dependent upon the application and the characteristics of participating appliances, requiring the development of decision-making algorithms to control it.

- Eucalyptus assumes that the Koala ORB can authenticate clients, but provides little support for security beyond this. It is not clear whether ad hoc and consumer wireless systems can be integrated with more complete security schemes without rendering them effectively useless.

- Koala provides applications with sufficient information on the contactability of objects to allow them to deal with disconnected operation. Most of the examples applications considered either depend intrinsically upon the contactability of remote resources, or use disconnection as a key feature of their function. A substantial body of applications rely on the availability of a network but should be able to fall back gracefully to other means if necessary. Techniques for disconnected operation of embedded devices may merit future attention.

- No caching of invocation parameters is performed by Koala, since caching schemes may adversely affect performance and have limited applicability to simple control tasks. If Koala is to be used for more general applications then a suite of caching primitives tailored to different application domains may be beneficial.

Bibliography

[1] Architecture Project Management Ltd. The ANSA Model for Trading and Federation. AR.005.00, February 1993.

[2] The ATM Forum. At `http://www.atmforum.com/`, 2001.

[3] J.M. Bacon, J. Bates, R.J. Hayton, and K. Moody. Using Events to build Distributed Applications. In *IEEE SDNE Services in Distributed and Networked Environments*, pages 148–155. Whistler, British Columbia, June 1995.

[4] Requirements for IP Version 4 Routers. RFC 1812, June 1995. F.Baker (editor).

[5] Frazer Bennett, David Clarke, Joseph B. Evans, Andy Hopper, Alan Jones, and David Leask. PicoNet - Embedded Mobile Networking. *IEEE Personal Communications, Vol. 4, No. 5, pp 8-15*, October 1997.

[6] Specification of the Bluetooth System Version 1.0A. From `http://www.bluetooth.com/`, July 1999.

[7] Service Discovery Protocol. In *Bluetooth Core Specification, Version 1.0A*, chapter 3. Bluetooth Consortium, July 1999.

[8] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design of a Scalable Event Notification Service: Interface and Architecture. Technical report, Department of Computer Science, University of Colorado, August 1998.

[9] CCIR / Post Office Code Standardisation Advisor. *CCIR Recommendation 584, Radiopaging Code 1*, February 1981.

[10] Raymond Cunningham. Architecture for Location Independent CORBA Environments. Dissertation for MSc in Computer Science, Trinity College, Dublin, September 1998.

[11] DALLAS Semiconductor. iButton homepage. At `http://www.ibutton.com/`, 2001.

[12] D.Greaves and R.Bradbury. Warren: a Low-Cost ATM Home Area Network. *IEEE Network Magazine*, 1998.

[13] DOLMEN Project. *Response to OMG Telecom RFI on Wireless Access and Mobility*, August 98.

[14] Electronic Industries Association (`http://www.eia.org/`). *EIA232E - Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange*, D edition, January 1987.

[15] FLEX Technologies. At `http://www.motorola.com/MIMS/MSPG/FLEX/`.

[16] Gray Girling, Jennifer Li Kam Wa, Paul Osborn, and Radina Stefanova. The Design and Implementation of a Low Power Ad Hoc Protocol Stack. In *IEEE Wireless Communications and Networking*, September 2000.

[17] Gray Girling, Jennifer Li Kam Wa, Paul Osborn, and Radina Stefanova. The PEN Low Power Protocol Stack. In *IEEE International Conference on Communications and Networks*, October 2000.

[18] D.J. Greaves, J. Bates, and E.J. Briscoe. AutoHAN Project Proposal. `http://www.cl.cam.ac.uk/Research/SRG/netos/han/AutoHAN/docs/proposal.html`, October 1997.

[19] Timothy C. Green. FAQ:comp.home.automation (Version1.09). `http://www.cs.ualberta.ca/~wade/HyperHome/Faq/Faq_outline.html`, February 1995. Also other informal sources.

[20] Duncan P. Grisby. *A Distributed Adaptive Window System*. PhD thesis, Computer Laboratory, University of Cambridge, 1999.

[21] Home Area Networks. At `http://www.cl.cam.ac.uk/Research/SRG/netos/han/`, June 2001.

[22] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The Anatomy of a Context-Aware Application. In *5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, August 1999.

[23] Franz J. Hauck, Maarten van Steen, and Andrew S. Tanenbaum. A Location Service for Worldwide Distributed Objects. Technical report, Dept. of Math. and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, 1997.

[24] HAVi, Inc. *The HAVi Specification*, 1.1 edition, May 2001.

[25] John Heidemann and Dhaval Shah. Experiences with User-configurable, Location-aware Scheduling. Technical Report 98-675, University of Southern California, April 1998.

[26] Hewlett Packard. CoolTown Homepage. At `http://cooltown.hp.com/`, 2002.

[27] HomeRF website. `http://www.homerf.org`.

[28] Andy Hopper. The Clifford Paterson Lecture, 1999 Sentient Computing. In *Philosophical Transactions*, number 358, pages 2349–2358. The Royal Society of London, 2000.

[29] IEEE, `http://www.ieee.org/`. *IEEE Std 1394-1995 IEEE Standard for a High Performance Serial Bus*, 1995.

[30] IEEE, `http://www.ieee.org/`. *Supplement To IEEE Standard For Information Technology- Telecommunications And Information Exchange Between Systems- Local And Metropolitan Area Networks- Specific*

Requirements- Part 11: Wireless LAN Medium Access Control (MAC) And Physical Layer (PHY) Specifications: Higher-speed Physical Layer Extension In The 2.4 GHz Band, January 1999.

[31] IrDA SIR Data Specification. At `http://www.irda.org/standards/pubs/IrData.zip`, February 1999.

[32] Alan Jones and Andy Hopper. The prototype embedded network (PEN). *Computer Networks*, 35:377–390, 2001.

[33] J.M. Kahn, R.H. Katz, and K.S.J. Pister. Next Century Challenges: Mobile Networking for "Smart Dust". *ACM/IEEE Intl. Conf. on Mobile Computing and Networking (MobiCom 99)*, August 1999.

[34] Jean J. Labrosse. *MicroC/OS, The Real-Time Kernel.* R & D Publications, Inc., 1992.

[35] Sai-Lai Lo. The Omni Object Services - the use of automatic code generation to tackle common distributed programming problems in CORBA. In *2nd MICO/CORBA Workshop*, March 2001.

[36] Diego López de Ipiña. *Visual Sensing and Middleware Support for Sentient Computing*. PhD thesis, Cambridge University Engineering Department, January 2002.

[37] Diego López de Ipiña and Eleftheria Katsiri. An ECA Rule-Matching Service for Simpler Development of Reactive Applications. *IEEE Distributed Systems Online*, 2001.

[38] Katriona Elizabeth Macphee. A Generic Control Language. Master's thesis, Computer Laboratory, University of Cambridge, 1998.

[39] GO project MANET. At `http://go.cs.hut.fi/technology/manet.html`.

[40] Thomas L. Martin and Daniel P. Siewioriek. Non-ideal Battery Properties and Low Power Operation in Wearable Computing. In *International Symposium on Wearable Computers*, pages 1001–106, October 1999.

[41] MICO - Mico Is COrba. At `http://www.mico.org/`, November 2000.

[42] P.V. Mockapetris. Domain Names - concepts and facilities. RFC 1034, RFC 1035, November 1987.

[43] Robert Morris, John Jannotti, Frans Kaashoek, Jinyang Li, and Douglas Decouto. CarNet: A Scalable Ad Hoc Wireless Network System. In Paulo Guedes, editor, *Proceedings of the Ninth ACM SIGOPS European Workshop*, pages 61–65. ACM, September 2000.

[44] Henk Muller and Cliff Randell. An event-driven sensor architecture for low power wearables. In *ICSE 2000, Workshop on Software Engineering for Wearable and Pervasive Computing*, pages 39–41. ACM/IEEE, June 2000.

[45] Roger M. Needham. Names. In Sape J. Mullender, editor, *Distributed Systems*, chapter 12. Addison-Wesley Publishing Co., 1993.

[46] Greg Nelson, editor. *Systems Programming with Modula-3*. Innovative Technology. Prentice Hall, 1991.

[47] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.4 edition, 2000.

[48] Object Management Group. *Trading Object Service Specification 1.0*, May 2000.

[49] Object Management Group. *Event Service Specification, Version 1.1*, February 2001.

[50] Object Management Group. *Notification Service Specification, Version 1.0*, February 2001.

[51] CORBA Messaging, Section 2: Message Programming Model. In *The Common Object Request Broker : Architecture and Specification, Revision 2.4.2*, chapter 22.4. Object Management Group, February 2001.

[52] CDR Transfer Syntax. In *The Common Object Request Broker : Architecture and Specification, Revision 2.4.2*, chapter 15.3. Object Management Group, February 2001.

[53] Dynamic Invocation Interface. In *The Common Object Request Broker : Architecture and Specification, Revision 2.4.2*, chapter 7. Object Management Group, February 2001.

[54] General Inter-ORB Protocol. In *The Common Object Request Broker : Architecture and Specification, Revision 2.4.2*, chapter 15. Object Management Group, February 2001.

[55] Internet Inter-ORB Protocol. In *The Common Object Request Broker : Architecture and Specification, Revision 2.4.2*, chapter 15.7. Object Management Group, February 2001.

[56] OMG IDL Syntax and Semantics. In *The Common Object Request Broker : Architecture and Specification, Revision 2.4.2*, chapter 3. Object Management Group, February 2001.

[57] Distributed Systems : Omni Object Services. At `http://www.uk.research.att.com/corba.html`, 2002.

[58] Jason Pascoe. The Stick-e Note Architecture: Extending the Interface Beyond the User. In Johanna Moore, Ernest Edmonds, and Angel Puerta, editors, *1997 International Conference on Intelligent User Interfaces*, pages 261–264. ACM, ACM, January 1997.

[59] Jason Pascoe. Adding Generic Contextual Capabilities to Wearable Computers. In *The Second International Symposium on Wearable Computers*, pages 92–99. IEEE Computer Society, October 1998.

[60] IP mobility support. RFC 2002, October 1996. C. Perkins (editor).

[61] Philips Semiconductors - Wireless Connectivity - RF-Lite. At `http://www.semiconductors.philips.com/wicon/rflite/`, 2001.

[62] Robert Dunbar Poor. Hyphos : A Self-Organizing Wireless Network. Master's thesis, Massachusetts Institute of Technology, June 1997.

[63] Jan M. Rabaey, M. Josie Ammer, Julio L. da Silva Jr., Danny Patel, and Shad Roundy. PicoRadio Supports Ad Hoc Ultra-Low Power Wireless Networking. *Wireless Computing - p42-48*, July 2000.

[64] Tristan Richardson, Frazer Bennett, Glenford Mapp, and Andy Hopper. Teleporting in an X Window System Environment. IEEE Personal Communications Magazine, Vol. 1, No. 3, pp 6-12, Fourth Quarter, 1994.

[65] Mike Spreitzer and Marvin Theimer. Providing Location Information in a Ubiquitous Computing Environment. In *SIGOPS*, pages 270–283. ACM, 1993.

[66] R. Srinivasan. *RPC: Remote Procedure Call Protocol Specification Version 2*. Sun Microsystems, August 1995. RFC 1831.

[67] Frank Stajano and Ross Anderson. The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks. In B. Christianson, B.Crispo, and M.Roe, editors, *Security Protocols. 7th International Workshop Proceedings*, Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 1999.

[68] Service Architecture. In *Jini Architecture and Specification, Version 1.0.1*, chapter 2.3. Sun Microsystems, November 1999.

[69] SyncML Initiative, `http://www.syncml.org`. *SyncML Specification Version 1.0.1*, 2001.

[70] Ultralife Battery Specifications. At `http://www.advanced-battery.com/Ultralife.htm`.

[71] Roy Want, Andy Hopper, Veronica Falcao, and Jonathan Gibbons. The Active Badge Location System. ACM Transactions on Information Systems, Vol. 10, No.1, pp 91-102, January 1992.

[72] Roy Want, Bill N. Schilit, Norman I. Adams, Rich Gold, Karin Petersen, David Goldberg, John R. Ellis, and Mark Weiser. The ParcTab Ubiquitous Computing Experiment. *Mobile Computing*, pages 45–102, 1995.

[73] Andy Ward. *Sensor-driven Computing*. PhD thesis, Cambridge University Computer Laboratory, 1998.

[74] WaveLAN. At `http://www.wavelan.com/`.

[75] James Weatherall and Andy Hopper. Predator: A Distributed Location Service and Example Applications. In Norbert A. Streitz, Jane Siegel, Volker Hartkopf, and Shin'chi Konomi, editors, *Cooperative Buildings*, number 1670 in Lecture Notes in Computer Science, pages 127–139. Springer, October 1999.

[76] James Weatherall and Alan Jones. Ubiquitous Networks and their Applications. *IEEE Wireless Communications Magazine*, 9(1):18–29, February 2002.

[77] James "Wez" Weatherall and David Scott. Mobile Computing with Python. In *The Ninth International Python Conference*, pages 87–96, mar 2001.

[78] J.N. Weatherall. The R2 Low-Power Messaging and Rendezvous Layer. In *Proceedings of the Ninth ACM SIGOPS European Workshop - Beyond the PC: New Challenges for the Operating System*, pages 43–48. The ACM, September 2000.

[79] David Wheeler. Personal correspondence.

[80] Wireless Integrated Network Sensors. At `http://www.janet.ucla.edu/WINS/`, February 1999.