Policy-driven Middleware for Personal Area Networks

Individual Project Report

Neil Madhvani
nm300e@doc.ic.ac.uk

MEng in Information Systems Engineering
Imperial College, London

June 2004

Supervisor: Dr Naranker Dulay
Second Marker: Dr Emil Lupu

Abstract

It is quite shocking to note that only ten years ago, a mobile telephone was generally seen as a luxury item and the vast majority were used for business purposes. Handsets were large in size and typically heavy, with battery standby times of less than a single day. Over the last decade we have experienced a revolution in the market for mobile consumer electronics devices, with the proliferation not only of mobile phones, but also products such as MP3 music players, personal digital assistants (PDAs) and handheld entertainment systems. As innovations in the electronics industry have lowered production costs and raised CPU power and battery life, many of us are finding ourselves carrying an increasing number of these devices. Whilst we have generally come to expect an improvement in our standard of living as a result of advances in technology, this experience may unfortunately be short-lived. The problem is that we are being faced with an ever-increased 'management burden' in operating and configuring these devices to work with one another in harmony. Technologies such as Bluetooth and WiFi are increasing in ubiquity and are useful for connecting devices in personal area networks (PANs); however very few attempts have been made to automate the way in which devices behave when in close proximity to one another.

In this report we firstly investigate recent developments in areas of research related to this problem domain. We then propose an architecture for middleware that we have developed, which enables personal devices to communicate together in an effective, efficient and appropriate manner, whilst minimising the amount of input needed from the end user. Our approach is based around the concept of a self-managed cell (SMC), containing a core set of management services. Later in the report, we present a case study showing a sample application of our solution and conclude with a discussion of the contributions and limitations of our work, as well as our thoughts for possible future extensions.

Acknowledgements

I would like to take this opportunity to thank my supervisor, Dr Naranker Dulay, for his invaluable assistance and guidance throughout the course of the project. I am particularly grateful to him for giving up so much of his time (quite often at short notice!) to discuss my ideas at length and providing me with useful feedback.

Thanks also to my second marker, Dr Emil Lupu, for the helpful advice he provided me with at our project review meeting.

Finally, thanks to the numerous colleagues of mine who have offered me helpful ideas and suggestions along the way.

Contents

Appendices

1 Introduction

The objective of this project is to attempt to design a set of middleware that will allow devices within a personal area network (PAN) to communicate together in an effective, efficient and appropriate manner, with a view to minimising the amount of input needed from the end user, thereby enabling more than just the 'geekiest computer nerds' to embrace and make best use of portable technology devices as they becomes available. As well as improving the quality of existing applications, it is envisaged that work in this area will open up a whole new range of exciting paradigms that were previously thought to be impossible or infeasible.

1.1 Motivation

It is somewhat shocking to discover that only sixty years ago, Thomas Watson, the then chairman of IBM, made the comment: "I think there is a world market for maybe five computers." Perhaps even more alarming is that up until the early 1980s, it was generally believed that the widespread deployment of computer technology for consumers was infeasible and unlikely. This is epitomised by the following quote from Ken Olson in 1977, who was president, chairman and founder of Digital Equipment Corp: "There is no reason anyone would want a computer in their home." This theory has of course been proved wrong. The dramatic fall in the price and size of microprocessors, together with the increase in their computational power, has revolutionised our lives. Today, almost all homes are dependent on microprocessor-controlled electronic devices, including personal computers, televisions, video recorders, DVD players, set-top boxes and even many kitchen appliances.

A more recent revolution, and one which is still in its infancy, concerns the proliferation of mobile consumer electronics devices and their increasing ubiquity. This includes mobile phones, personal digital assistants (PDAs), MP3 players, digital watches etc. Consumers nowadays are carrying an increasing number of these devices, resulting in an increasing amount of time spent configuring and operating them. The following is an example of the type of situation that many users face today:

Bob is wearing headphones and listening to his favourite piece of classical music on his portable MP3 player. His mobile phone starts to ring, but it takes him a while to realise that he has an incoming call from Alice, because the music happens to be quite loud. Once he becomes aware of the call, he fumbles to find the pause button on his MP3 player. By the time he is finally in a position to be able to answer the call, Alice has

already been transferred to Bob's voicemail and he ends up having to call her back. He sighs and concedes that these kinds of problems are only going to intensify as he adds to his collection of 'gadgets'.

As the example above illustrates, whilst technology is continually providing us with a range of new opportunities, the benefits of these developments are unlikely to be realised unless personal electronic devices are able to communicate with each other, and reduce the **management burden** on users. To date, very few attempts have been made to automate the management of these devices and this is already causing inconvenience and frustration for many people. Whilst one may argue that users ought to perhaps carry a single universal device that could incorporate a myriad of functions, this is unlikely to be a feasible option. Whilst we can already purchase devices that offer more than one function, such as a combined phone/PDA, these tend to be larger in size and may also lack flexibility. Therefore the issues of **self-management** and **device collaboration** (discussed later) require our attention today, if we are to ensure that new technology will continue to improve our quality of living rather than be viewed as an inconvenience. Going back to our scenario with Bob and Alice, this is perhaps what we would like to see instead:

Bob is wearing headphones and listening to his favourite piece of classical music on his portable MP3 player. His mobile phone starts to ring, and the music automatically pauses without any intervention. He hears a message through his headphones that includes the name of the caller, Alice. Bob has been expecting this important call, and answers it immediately. Whilst talking to Alice, he picks up his PDA, which has already sensed that he is on the phone to Alice and alerts him to the fact that it was her birthday a few days ago. He wishes her a belated happy birthday. At the end of the call, his MP3 player resumes playback, again without Bob having to do anything. He smiles and thinks to himself – isn't technology just wonderful!

Whilst our focus is directed towards the interaction of consumer devices, similar concepts are already being considered for applications such as e-healthcare. In the field of medicine, it is becoming increasingly important for those caring for a patient in a hospital for example to have access to up-to-date information and to be alerted about changes in health conditions. A slightly different application is where teams of people on a rescue mission need to work together in an environment with no inherent networking infrastructure. If the devices of several emergency forces could collaborate with one another, without having to waste valuable time configuring hardware and software on arrival, this could have a significant positive impact on the success of the operation.

A complementary vision is that of **utility computing** where enterprises and users will be able to tap into potentially vast computational resources from 'low-end' devices. BusinessWeek comments that the idea is to "make computing power into another pay-as-you-go service, like water or electricity" [BUSN03]. Whilst there are a number of challenges to be overcome in this field, it is almost certain that the areas of utility computing and self-management will meet somewhere in the middle.

1.2 Key Project Objectives

Before we discuss the goals of the project in any level of detail, it is worth defining the operative words in the choice of title for this piece of work:

- **Policy-driven –** a policy in this context refers to a specification of some rule that governs the behaviour of one or more managed objects, triggered by some event in the system, such as a time event, or the discovery of a service. Policies are used in a range of areas such as in the network security arena, where firewalls that prevent data from passing from one network to another may be configured through the definition and parsing of such policies. In this project we focus on obligation policies, which are modelled around the concept of **event-condition-action (ECA)** rules, i.e. we use the policy to specify what behaviour (action) should occur when the event occurs, if the given condition evaluates to true. Policies provide with a degree of flexibility and dynamic configurability. For example, if devices are grouped in a hierarchical fashion, a single policy, when activated, may affect the behaviour of a range of devices rather than just a single one. We will look at the policy-driven paradigm in further detail in the background study, which follows in Chapter 2.

- **Middleware -** Middleware is used to describe sets of services and abstractions that facilitate the development and deployment of distributed applications in heterogeneous computing environments. Middleware is frequently used in the client/server paradigm, where it is beneficial (e.g. for performance reasons) to introduce an additional middle layer between a server and its users. In the context of this project, we use middleware to refer to a proposed architecture in which devices with different low-level implementations of functionality share a set of common services and abstractions, permitting development, deployment and management of Personal Area Network applications. In the next chapter, we will look at the types of middleware solutions available today and investigate a selection of them.

- **Personal Area Networks** – often abbreviated to PAN, these networks are similar to Local Area Networks (LAN), however the objective of a PAN is to network devices that are close to one person, such as phones and PDAs. The devices may or may not belong to the person in question. The reach of a PAN is typically just a few metres. PAN is quite a generic term, and there are specific types such as a Piconet, which is one that can be formed using devices that communicate over Bluetooth. In addition, there are several esoteric PANs, such as those that exploit the electroconductivity around the body to transmit data, and we will look at an example of this in the following chapter.

The most important areas of interest that we have selected in particular for investigation are:

- **Generic device support** – the proposed solution will not be tied to a specific set of devices. This is of particular importance in the consumer electronics arena, where the range and variety of devices is changing on a daily basis. Instead, the focus will be of defining how device functionality can be grouped into **profiles**, and how additional profiles can be defined and used. In the context of our work, profiles define the events that can be generated by devices, and the actions which can be performed on them. The analogy here is with Bluetooth profiles, which define functionality that is supported on top of the main protocol stack.
- **Flexible and expansible architecture –** a modular, object oriented approach will be advocated and used wherever possible, enabling enhancements to be made by replacing one or more pluggable components within the architecture.
- **High configurability** – one of the key objectives is to develop a solution that is highly configurable. An area to be investigated here is how we can use policies not just for defining device collaboration, but also for the purposes of the configuration of the whole architecture. These may be higher-level management policies.
- **Mobility** – it is clear that mobile devices tend to move in and out of proximity of each other on a regular basis. It is therefore appropriate to consider how these situations should be handled in as seamless a manner as possible.
- **Low management overhead** – whilst portable devices are becoming smaller in size, unfortunately improvements in battery technology are not keeping up at anywhere near the same pace.

Therefore power should be seen as a scarce resource and its consumption should be minimised wherever possible.

1.3 Summary of contributions

The work that we present in this report begins with a background study that we conducted in the early stages of the project, to investigate recent developments in relevant, related areas. In addition to looking at issues such as policy specification and ad hoc networking, we also spent time investigating the work conducted by project such as MIT's Project Oxygen and the AMUSE Project that is a joint effort between Imperial College, London and the University of Glasgow. AMUSE is very closely aligned to our work, and we draw on several of the concepts that this project has proposed, such as the self-managed cell (SMC) architecture. We also looked at the area of **autonomic computing** in some detail, based on work initiated by IBM.

We then build upon our background study and propose a Policy-driven Middleware for Personal Area Networks. We take a two-staged approach to the presentation of our design. Firstly, we cover the entire system in high-level detail, focussing on the key interactions between the management components and PAN devices. We then provide a detailed architectural design, describing how our design concepts can be translated into a software implementation. We subsequently present a case study, showing how our solution may be used to solve a set of problems facing a user who owns a range of consumer devices. We then describe a 'proof of concept' prototype implementation that we carried out using Java and a selection of other supporting tools.

The key contributions that we believe this work has made in this area are as follows:

- **Definition of a SMC structure for PANs** – we propose a set of mandatory management components for our SMC and provide a detailed design for each.
- **Development of a flexible architecture** – wherever possible, we enable behaviour to be defined through configuration data rather than being hard-coded.
- **Explicit support for processed and unprocessed contextual data** – the approach we advocate is that contextual data may be obtained from devices such as sensors that exist outside of a SMC but may also be generally internally, e.g. translation of low-level concepts such as numerical values into high-level ones such as {high, medium, low}.

- **A model for handling correlated events** – we recognise that raw events on their own may not be of use. We therefore propose a scheme by which policies can trigger based on event patterns and ordering. We use a state machine based approach and provide pointers for future development.
- **SMC policies** – in addition to using policy notation to specify device behaviour, we use SMC policies to configure various aspects of the middleware.
- **Pure asynchronous event bus** – we advocate the use of a single asynchronous event bus for all communication, based on the one-shot paradigm rather than request/reply. Our approach is to keep the communication model as simple as possible, in order to minimise overhead.
- **Event quenching** – in our implementation, producers of events do not transmit them on to the bus unless there are consumers who are interested in them. This significantly reduces unnecessary network traffic and improves efficiency.
- **Action encapsulation** – we propose a scheme in which actions to be performed on devices are encapsulated in an object and sent in a serialized form for execution. We suggest that this is a neater solution that carrying out a remote method call on a device.
- **Distributed, multi-threaded architecture** – our design enables management components to be distributed across several different devices if required, though we suggest that a centralised approach may be more efficient in many cases. In addition, our domain and policy management agents exploit concurrency by running as individual threads.

1.4 Report outline

Chapter 1 (Introduction) provides the motivation for our work, presents the key project objectives and summarises our achievements.

Chapter 2 (Background) presents a study into areas of research related to our problem domain, including ad hoc networking, policy specification and recent developments in hardware and software platforms for personal devices. We also look at a selection of ongoing research projects that are working on similar issues.

Chapter 3 (Project Specification) defines the scope of our work, the key requirements for the solution we have developed and the main interactions with the system.

Chapter 4 (Architectural Design Overview) presents a high-level description for our middleware architecture. We start by looking at the components that make up a self-managed cell and then discuss the behaviour of each of these.

Chapter 5 (Detailed Architectural Design) builds upon the work presented in the previous chapter. Whilst we looked at relatively high-level concepts in Chapter 4, here we discuss implementation issues and use UML notation to show the structure of the software that we propose.

Chapter 6 (Case Study) attempts to take a step back to the user and application domains, in order to consider how the solution we have developed could be used to solve a type of problem that we may expect to come across. The intention is that this should also serve as a form of tutorial for developers wishing to use the design to solve similar problems.

Chapter 7 (Building a prototype) discusses the work that we carried out later in the project to construct a working demonstration of a subset of our architecture. The aim of this exercise was to carry out 'proof of concept' tests in order to integrate our new functionality with existing third-party products we are using such as the Elvin event system.

Chapter 8 (Evaluation and future directions) reviews the work we have carried out, highlighting what we consider to be the key achievements and limitations of our approach. We also make suggestions for possible extensions to our work and potential areas of interest for the future.

Appendix A1 (Requirements Capture) provides extensions to the specification material presented in Chapter 3. In this appendix, we make use of UML use case notation which is useful for capturing the key 'actors' in a system and how they require to interact with it.

Appendix A2 (Bibliography) provides references to sources used during this project.

1.5 Glossary of terms

API	Application Programmers Interface
CCE	Context & Collaboration Engine
CF	Compact Framework
CIM	Common Information Model
CORBA	Common Object Request Broker Architecture
CVS	Concurrent Versions System
DEN	Directory Enabled Network
DMA	Domain Management Agent
DMTF	Distributed Management Task Force
DSTC	Distributed Systems Technology Centre

ECA	Event-Condition-Action
GoF	Gang of Four (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides)
GPRS	General Packet Radio Service
GSM	Global Standard for Mobile Communications
HTTP	Hypertext Transfer Protocol
IDL	Interface Definition Language
IEEE	Institute of Electrical & Electronic Engineers
IETF	Internet Engineering Task Force
IrDA	Infrared Data Assocation
J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edition
JVM	Java Virtual Machine
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
MANET	Mobile Ad hoc NETworks
MP3	MPEG I Layer 3 Audio
OMG	Object Management Group
ORB	Object Request Broker
P2P	Peer-to-Peer
PAN	Personal Area Network
PDA	Personal Digital Assistant
PMA	Policy Management Agent
QoS	Quality of Service
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SDK	Software Development Kit
SMC	Self-Managed Cell
SOAP	Simple Object Access Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
UDDI	Universal Description, Discovery and Integration
UDP	User Datagram Protocol
UML	Unified Modelling Language
WAN	Wide Area Network
WAP	Wireless Application Protocol
WiFi	Wireless Fidelity
WSDL	Web Services Description Language
XML	eXtensible Markup Language

2 Background

This chapter provides a summary of the background study carried out within fields related to this project. The aim of this exercise was to investigate and assess the impact of recent developments in the mobile computing, cellular telephony, policy specification and ad hoc networking arenas. The chapter ends with a summary of the key technical challenges that have been identified.

2.1 Overview of areas of interest

The Alice and Bob scenario that we considered in our motivational section in the previous chapter provides us with a useful starting point for determining the areas that we should investigate in our background study. Firstly, we are interested in a method of expressing the behaviour of devices in response to the occurrence of events. For example, we would like to state that when an incoming call from Alice comes in on Bob's phone, he should be alerted to this via his MP3 player. We therefore investigate the area of policy specification including some of the existing approaches.

Our requirement for devices to be able to communicate with each other leads us to consider the types of middleware systems available and to find a suitable approach for the product that we are building. We then consider networking technologies in Local Area Networks and Personal Area Networks and investigate recent developments in Mobile Ad hoc Networking. Since we would like to produce a prototype later in the project, we include as part of our background study, a consideration of what is 'state-of-the-art' in terms of hardware and software platforms.

Finally, we look more specifically at other work that is currently being carried out in the area of PAN self-management.

2.2 Policy specification

In this section, we investigate policy specification, an active research topic and pertinent to this work, since we are proposing the use of policies to specify device behaviour in a personal area network. Policies are rules governing the choices in behaviour of a system [SLOM02]. Two classical types of policies are:

- **Authorisation Policies** – these policies define which resources or services a subject can access. A subject may be a managed object, a management agent or a user.
- **Obligation Policies –** these are event-triggered condition-action rules. They define what a subject must or must not do.

Current policy research includes proposals for trust and privacy specification, digital rights management, context-aware policies and intrusion response policies.

There is an important distinction between policies and one-off commands in that policies are persistent. The authors suggest that the main motivation for the recent interest in policy-based systems is to support dynamic adaptability of behaviour by changing policy without recoding or stopping the system. Such a scheme makes the use of a policy-driven architecture particularly useful in a distributed environment where it is necessary to distribute the policies across many heterogeneous components.

Of particular interest to this project is the ability to specify policies relating to groups of entities rather than just to individual resources. Just as for implementing security policy, it may be desirable to divide up the resources into departments, for management policies we may wish to divide up devices into different types, based on the functionality that they can provide. These groups are often referred to as 'domains' [SLOM94], and a management domain can be defined as a collection of managed objects which have been explicitly grouped together for the purposes of management.

This paper also draws a distinction between direct and indirect members of a domain:

- **Direct member** – if a domain holds a reference to an object then the object is said to be a direct member of that domain and the domain is said to be its parent.
- **Indirect member** – domains may be members of other domains in a hierarchical fashion; this relationship is known as a

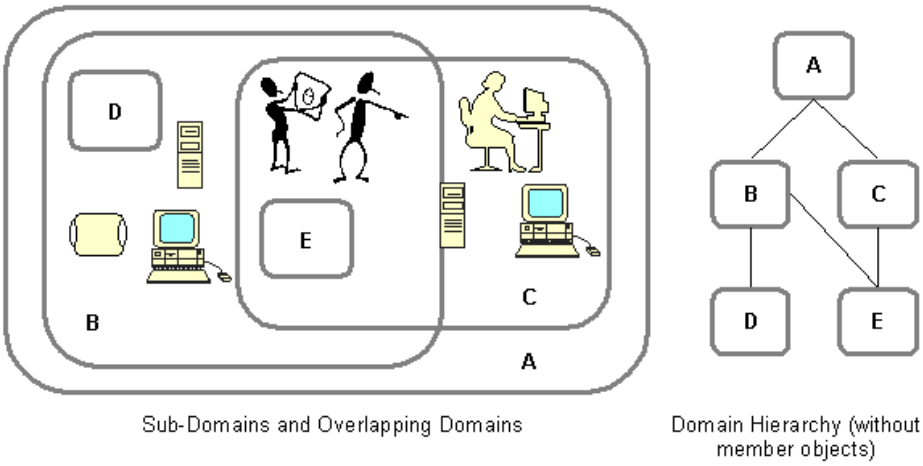

Figure 2.1: Management domain structure [DAMI99].

'subdomain' of a parent. Members of a subdomain are then indirect members of the parent domain.

This model provides flexibility and is similar to hierarchical file systems. Entities can be direct or indirect members of multiple domains. If an entity if a direct member of multiple domains, then the parent domains are said to 'overlap'. This domain structure fits well with the requirement in this project for devices to be able to belong to multiple groups, e.g. a combined phone/PDA device may belong to both the Phone and the PDA group. Figure 2.1 shows an example of a domain hierarchy with sub-domains and overlapping domains.

2.2.1 Ponder (Imperial College, London)

Ponder is a declarative, object-oriented language for specifying security and management policy for distributed object systems [DAMI01]. Both obligation and authorisation policies are supported. Work has also been done to map Ponder policies on to various access control mechanisms such as firewalls, operating systems, databases and Java [SLOM02]. A significant feature of Ponder is its support for domains that enable objects to be grouped in a hierarchical manner. Membership of domains in Ponder is explicit and not defined in terms of a predicate on object attributes. The scheme of direct and indirect domain membership is implemented as we discussed earlier in this chapter, enabling policy propagation.

Although Ponder supports a range of additional types of policies such as delegation and refrain, we only consider the two main ones here, namely authorisation and obligation. In addition we take a brief look at composite policies. Authorisation policies define what activities a member of the subject domain can perform on the set of objects in the target domain and have the following syntax:

```
inst ( auth+ | auth– ) policyName "{"
        subject [<type>]    domain-Scope-Expression ;
        target [<type>]     domain-Scope-Expression ;
        action              action-list ;
        [ when              constraint-Expression ; ]          "}"
```

The type auth+ denotes a positive authorisation policy (actions that subjects are *permitted* to perform on target objects), whereas auth- denotes a negative authorisation policy (actions that subjects are *forbidden* to perform on target objects). An example of a positive authorisation policy is:

```
inst auth+ sendTextMessages {
        subject      /users/children ;
        target       /dev/cellular/phones ;
```

```
        action      send_text_message(msg) ;
        when        time.between("0900","1900") ;
}
```

The above policy specifies that users in the /users/children domain are permitted to send text messages on devices in the /dev/cellular/phones domain between the hours of 09:00 and 19:00. Similarly, we could define an auth- (negative authorisation) policy to explicitly prohibit the activity between a given time period.

In contrast, obligation policies are event-triggered condition-action rules. These policies define the activities that subjects must perform on objects in the target domain. The syntax for obligation policies in Ponder is as follows:

```
inst oblig policyName "{"
        on                  event-specification ;
        subject [<type>]    domain-Scope-Expression ;
        [ target [<type>]   domain-Scope-Expression ; ]
        do                  obligation-action-list ;
        [ catch             exception-specification ; ]
        [ when              constraint-Expression ; ]        "}"
```

An example of an obligation policy we could possibly use for the self-management of consumer devices is:

```
inst oblig incomingCellularCall {
        on          eventIncomingCellularCall(callerID) ;
        subject     s = /dev/cellular/phones/Manager ;
        target      t = /dev/music ;
        do          t.mute() -> t.playCallerID(callerID) ;
        when        s.profile != "do not disturb" ;
}
```

The above policy is triggered when the eventIncomingCellularCall event occurs. At this time (i.e. an incoming call is coming in), all devices that are in the /dev/music namespace will be asked to mute by the Manager object sitting in the /dev/cellular/phones domain, and to then play out the incoming caller's ID to the user. However these actions will not be carried out if the phone's profile is in the "do not disturb" mode, i.e. the user doesn't want the music to stop when a call comes in!

Note that the subject and target can be individual entities or domains. In the example above, the subject was a single entity (a policy management agent). If we had specified s = /dev/cellular/phones, then effectively all devices in that domain would have to perform the action on all of the devices in the target domain.

As specified in the Ponder language, the basic policy constraints can be derived from:

- **Subject / target state** – reflected by attributes at an object's interface.
- **Action / event parameters**
- **Time constraints**, e.g. between 0200 and 0400.

In addition to the primitive policies discussed above, Ponder also supports composite policies in order to be able to group policies and structure them to reflect organisational structure, preserve the natural way system administrators operate, or simply provide reusability of common definitions [SLOM02b]. There are two key types of composite policies:

- **Roles –** provide a semantic grouping of policies with a common subject, generally pertaining to a position within an organisation. For example, we may wish to specify policies in terms of manager positions rather than specific individuals, as if an individual leaves that position it is inconvenient and inefficient to have to re-specify the affected policies. We can also use a role to specify the policies that apply to an automated component acting as a subject in the system.
- **Relationships –** groups the policies defining the rights and duties of roles towards each other. Relationships provide an abstraction for defining policies that are not the roles themselves but are part of the interaction between the roles.

The policies above are specified by enveloping the relevant auth and oblig policies within an outer *type role <role Name> (params)* or *type rel <rel Name> (params)* as appropriate. Further details can be found in the Ponder documentation [SLOM02b].

As well as a language for policy specification, the work on Ponder has also resulted in the creation of a development toolkit including a compiler that maps policies to low-level representations such as Java code and Windows 2000 security templates. These tools are of some interest to this project, since the scope exists to develop extensions that can generate low-level policies suitable for our implementation from higher-level Ponder policy definitions.

2.2.2 CIM Policy Model (IETF & DMTF)

The IETF's Policy working group [IETFWWW] and the Distributed Management Task Force (DMTF) [DMTFWWW] have jointly developed an object-oriented policy model that enables constructing policy rules of the form: *if <condition(s)> then <action(s)>* [DMTF04]. They define a policy as a 'definite goal, course or method of action to guide and determine present and future decisions'. The proposed policy model is an extension

to the Common Information Model (CIM), a hierarchical architecture comprised of a specification and a schema. The CIM Specification defines the details for integration with other management models, while the CIM Schema provides the actual model descriptions.

The CIM Policy Model does not however distinguish between authorisation and obligation policies. In addition, the policy rules do not include an explicit triggering event. It is assumed that the agent interpreting the event will evaluate the policy when an implicit event occurs [SLOM02]. Aggregation is supported through the use of nested policy groups. Conditions and actions can be specific to a particular rule but can also be stored in a separate policy repository and then reused by multiple rules.

A possible advantage of the CIM Policy Model is that whilst CIM is designed to be technology and implementation-neutral, the IETF have defined a mapping from CIM to the Lightweight Directory Access Protocol (LDAP). This work is referred to as the Directory Enabled Network (DEN) initiative [DMTF04b] and is an ongoing project, however the mapping from CIM v2.5 to LDAP is available today.

2.2.3 Potential policy caveats

One of the most interesting areas of active research in the field of policy specification concerns **conflict analysis**. In a system with many policies, it is quite likely that the decision process affecting the behaviour of an object at a particular moment in time may not in a decisive course of action if more than one policy is relevant and there is disagreement between those policies. For example, on an incoming phone call, one policy wishes to mute all music devices, whereas another simply wants to lower the volume. There is a conflict since we cannot determine which of the two actions should be performed. The IETF framework advocates a solution consisting of priorities that are assigned to each policy, however it is suggested [SLOM02] that this is difficult to implement in large systems where many different people are involved in the specification of policies.

The approach taken in Ponder is to detect modality conflicts (those that arise between corresponding auth+ and auth- policies) using syntactic analysis. Other types of conflicts can only be detected by understanding the actions being performed by the policies. It is suggested that constraints should be specified using meta-policies and the policy set should then be analysed against the constraints to see if there are any conflicts.

Whilst the focus on this project is not to investigate or try to resolve these types of policy-related problems, we can still take steps in our design to reduce the chances of these occurring by using tighter policies.

2.3 Middleware systems

The IEEE Distributed Systems group [DSOL03] suggests that the role of middleware is to ease the task of designing, programming and managing distributed applications by providing a simple, consistent and integrated distributed programming environment. Essentially, middleware is a distributed software layer, or 'platform' which abstracts over the complexity and heterogeneity of the underlying distributed environment with its multitude of network technologies, machine architectures, operating systems and programming languages.

In this section we take a look at several off-the-shelf middleware solutions that can be used to build distributed systems. This area is of particular interest to this project since we would like to find an efficient method of enabling heterogeneous portable devices to communicate in a personal area network. There are a number of different types of middleware, the most important of which are:

- **Object-based middleware** – applications are structured into potentially distributed objects that interact via location transparent method invocation. Examples are OMG's CORBA, Java RMI and Microsoft's Distributed COM architecture. Communication between objects is of the request-reply style.
- **Event-based middleware** – employs 'single shot' style communication rather than request-reply. This type of middleware is particularly suited to the construction of non-centralised distributed applications that must monitor and react to changes in their environment. An example is Mantara's Elvin.
- **Message-oriented middleware –** similar to the above but focussed towards applications in which messages need to be persistently stored and queued. A popular example is IBM's MQSeries.

Next, we take a look at a selection of real middleware solutions.

2.3.1 Java RMI

Java's RMI (Remote Method Invocation) allows Java developers to invoke object methods and have them execute on remote Java Virtual Machines (JVMs). RMI is basically an object-oriented RPC (Remote Procedure Call) mechanism. One of the major advantages of RMI is its ability to pass and return entire objects as parameters. Any object can be passed as a parameter, meaning that new code can be sent across a network and dynamically loaded at run-time by foreign virtual machines [REIL00]. RMI makes use of a registry – a remote object that maps names to remote objects. A server registers its remote objects with the registry so

that they can be looked up. When an object wants to invoke a method on a remote object, it must first lookup the remote object using its name. The registry returns to the calling object a reference to the remote object, using which a remote method can be invoked.

RMI is being increasingly used as an object-based middleware in a range of distributed applications, however we have highlighted a few potential drawbacks of using RMI in the context of this project:

- **Java only** – it would be preferable not to tie ourselves to a particular language. Although Java is portable across platforms, many personal consumer devices still do not support it.
- **RMI not supported in J2ME** – many phones and PDAs support a cut-down version of Java, known as Java 2 Micro Edition (J2ME). This does not provide any RMI support.
- **Potentially high overhead** – RMI uses a synchronous request-reply scheme and there is an overhead involved in object serialisation. For collaboration about change of state in a personal area network, an event-driven approach is likely to be preferable, since we only need to broadcast a limited amount of information.

2.3.2 CORBA

CORBA (Common Object Request Broker Architecture) is similar to RMI in many ways but offers greater portability by not being tied to one particular programming language [REIL00]. CORBA can't be used to send executable code across to remote systems however. CORBA services are described by an interface, written in the Interface Definition Language (IDL). There are IDL mappings for many languages and there is future scope for adding CORBA support to other languages. However a CORBA implementation requires the deployment of ORBs (Object Request Brokers). ORBs are like registries; they are used to hold published service interfaces, which clients query and then bind to. Communication proceeds directly once clients have bound to servers. This presents an additional overhead and therefore may not be an appropriate choice of middleware for personal area networks.

2.3.3 Web Services: SOAP, WSDL, UDDI

Web services are a relatively new method of application-to-application interaction and developments in this area have been fuelled by the growth in use of the Internet to conduct business transactions. The intention is to replace ad hoc and proprietary protocols with a systematic and extensible framework that runs on top of existing Web protocols and

uses open XML standards [CURB02]. The framework can be divided into three key areas:

- **communication** – the Simple Object Access Protocol (SOAP) which was initially created by Microsoft is an XML-based protocol for messaging and remote procedure calls. One of the advantages of SOAP is that rather that defining a new transport protocol, it works on existing ones such as HTTP and SMTP. In addition, SOAP implementations exist for a range of programming languages including C, Java and Perl.
- **service description** – the Web Services Description Language (WSDL) is an XML format developed by IBM and Microsoft that describes the interface of a Web service and provides users with a point of contact. WSDL effectively provides a formalised description of client-service interaction including the permitted exchange of messages.
- **service discovery** – the Universal Description, Discovery and Integration (UDDI) specifications offer users a unified and systematic way to find providers of Web services through a centralised registry. It is analogous to an online "phone directory".

Web services are already being used with portable devices such as PDAs – for example, Microsoft's .NET Compact Framework allows developers to create applications for the Pocket PC environment that can 'consume' Web services. It is also possible to deploy such applications on a range of new Smartphone devices which tend to be smaller than PDAs. Web services are however more ideally suited to applications where it is necessary for client devices to retrieve information from a server, rather than for sharing of events between devices.

2.3.4 Elvin

Elvin [DSTCWWW] is an event-based middleware product originally developed by the Distributed Systems Technology Centre (DSTC) at the University of Queensland, Australia. The product has recently been commercialised and is now owned and developed by Mantara Software [MANTWWW]. Elvin uses a client-server architecture for delivering notifications. Clients establish sessions with an Elvin server process and are then able to send notifications for delivery or register to receive notifications sent by others. Clients can act as both producers and consumers of information within the same session.

The task of an Elvin server, or router, is to manage client connections and route notifications from producers to consumers. Consumers express their interest in a notification by registering a subscription with the server. This

subscription expresses selection criteria in terms of the content of each message. When the server receives a notification, it checks the content of the message against the registered subscriptions and forwards the notification to each client with a match. A single notification can match any number of subscriptions and is delivered to all active clients with a match [DSTC03].

More recent versions of Elvin also support the concept of **'quenching'**. A quench is a reverse subscription and provides added efficiency by only distributing notifications when there are subscribers who are interested in them. This feature is of particular interest for this project, since devices may have the ability to send information about a wide range of different types of state change, however it makes sense to only transmit as much information as is required by other devices that comprise the system.

The Elvin protocol is extremely lightweight and simple. A notification is simply an arbitrary length list of name-value elements, similar to a Hashtable in Java. For example, we could use the following notification to notify devices about an incoming mobile phone call:

```
eventName: "onIncomingCellularCall"
callerID:  "+447082919204"
timeOfCall: 20040525125434
```

Subscriptions are then formed using Elvin's own subscription language [MANT04], which is based on logical expressions. An example that would trigger for all notifications of the above type except when the callerID value is null can be given as follows:

```
require(eventName) && eventName == "onIncomingCellularCall"
&& callerID != null
```

To make use of Elvin, it is firstly necessary to install the Router, which is the centralised component that all consumers and producers talk to. Currently, this product can only run on the Microsoft Windows environment and various flavours of UNIX. This effectively means that it is likely to prove difficult to install Elvin on a PDA for example at the current time. However Elvin SDKs are available for Java, C/C++, Microsoft COM and Python and this makes it possible to run Elvin-based applications on pocket devices using the Jeode PersonalJava virtual machine, for example. The advantages of high throughput and low overhead make Elvin a good candidate for a personal area network middleware.

2.3.5 xmlBlaster

xmlBlaster [XMLBWWW] is an example of a message-oriented middleware. The key features of this product are:

- Both publish/subscribe and point-to-point communication types supported.
- 100% Java based – the xmlBlaster server can only run on a Java platform.
- Language neutrality for clients – supports many languages including C/C++, Java, Python, PHP, Javascript, Perl, C#, Visual Basic .NET.
- Multi protocol support – includes CORBA, RMI and xmlRPC. Clients are free to choose their preferred protocol.
- Use of standard XML XPath expressions for subscriptions.

The xmlBlaster product is developed under the open source model, and is therefore available for commercial and non-commercial use at no charge. Messages can contain virtually anything, including GIF images, Java objects, XML data and plain text. xmlBlaster is an extremely flexible architecture with relatively low overhead. The results of tests carried out by xmlBlaster indicate that the best performance can be achieved using CORBA as the protocol, however RMI is only slightly slower.

2.4 Wireless LAN & PAN technologies

Over the last few years, there has been a prolific growth in the use of Bluetooth and WiFi in particular, as the costs of these technologies have fallen. Conceptually, the difference between a PAN and a wireless LAN is that the former tends to be centred around one person while the latter is a local area network (LAN) that is connected without wires and serving multiple users. Cellular radio networks fall into the category of Wireless WANs (Wide Area Networks) since communication between terminals and base stations is at a much greater distance.

2.4.1 Bluetooth (IEEE 802.15)

The Bluetooth standard was developed by a Special Interest Group (SIG) consortium including Ericsson, IBM, Intel, Nokia and Toshiba and was envisaged to be a replacement for cable, infrared and other connection media [SUNM04]. Bluetooth is designed to connect small devices like PDAs and mobile phones. The technology has the following primary advantages:

- **automatic** – there is the potential for devices to find each other and communicate automatically without the need for initiation.
- **inexpensive** – many devices already have Bluetooth transceivers built-in and the cost of incorporating Bluetooth into a device is now less than $20 per unit.
- **unlicensed radio band** – most governments have agreed on a single standard so the same devices can be used in almost all countries.
- **robust** – Bluetooth, unlike infra-red does not require a line-of-sight link. Signals are omni-directional and can pass through walls and briefcases.

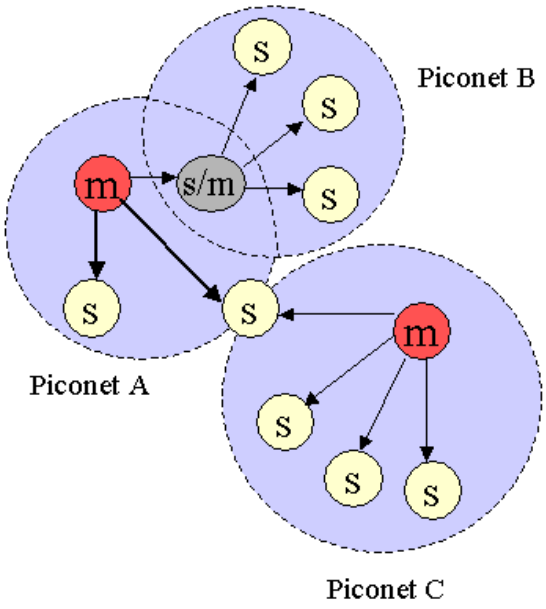

Figure 2.2: Scatternet of 3 Piconets [SUNM04]

Bluetooth is already being used in a range of applications such as wireless headsets that connect to mobile phones, device synchronisation, car kits and to a limited extent for mobile payments, where a mobile phone may be able to communicate with a vending machine in order to conduct a transaction for a can of soft drink where the charge is applied to the customer's bill.

However as Groten and Schmidt [GROT01] comment, until recently, Bluetooth was mainly pictured as a cheap technology enabling peer-to-peer communications between a central terminal such as a mobile phone and peripheral supporting devices such as wireless headsets. The concept of wireless ad hoc networks has changed this however, and Bluetooth is now perceived to be a much more useful technology for use in Personal Area Networks, where multi-hop routing and lack of fixed infrastructure are particularly important characteristics. Bluetooth-enabled devices are organized in groups called *piconets*. A piconet consists of a master and up to seven active slaves. A master and a single slave use point-to-point communication; if there are multiple slaves, point-to-multipoint communication is used. A master unit is the device that initiates the communication. A device in one piconet can communicate to another device in another piconet, forming a *scatternet*, as depicted in Figure 2.2. A master in one piconet may be a slave in another.

The Bluetooth specification is now quite mature and includes a Protocol Stack as well as series of profiles that are intended to ensure interoperability among Bluetooth-enabled devices and applications from different manufacturers and vendors. The most relevant of these to this project is the Bluetooth Personal Area Network (PAN) profile. An ad hoc network in the PAN profile consists of a single Bluetooth piconet. The PAN profile does not cover scatternet networking [AFFI04], however this is not considered to be a problem, since most users are unlikely to have more than 8 devices in their personal area network.

2.4.2 WiFi (IEEE 802.11a/b/g)

WiFi is currently one of the best performing areas of today's communications business. In 2002, annual industry revenues exceeded $1 billion and are expected to exceed $4 billion by 2007 [HENR02]. The term WiFi is generally used as a friendly name to refer to the IEEE 802.11b standard, which is the most popular and widely deployed technology for wireless local area networking in both business and home environments. The technology was originally designed primarily for private applications but is also being used in public places to create hotspots where users with WiFi hardware can obtain wireless broadband Internet access.

A key advantage of WiFi technology is that many devices already support it, either internally or via an optional card. In addition, it is possible to run WiFi in infrastructure or ad-hoc mode, where the latter is of particular interest for peer-to-peer (p2p) networks. The range for WiFi is considerably higher than for Bluetooth – typically several hundred metres as opposed to only a few metres. However WiFi has a much higher power consumption as a result and as a result it could be argued that Bluetooth is a more efficient technology for building personal area networks consisting of small, battery-powered devices.

The 802.11b standard operates at speeds of upto 11Mbps, whereas the newer 802.11a and 802.11g technologies increase this to a maximum of 54Mbps. These speed increases are likely to be useful for wireless local area networking, where the devices being networked are typically desktop and notebook PCs. However for the foreseeable future, we are unlikely to see higher-speed wireless technologies supported in phones, PDAs and other small consumer devices since the benefits are unlikely to outweigh increased power requirements and complexity of hardware.

2.4.3 Proprietary low-range radio networks

Whilst Bluetooth and WiFi have positioned themselves as the most popular wireless communication mechanisms in the last few years, several vendors have developed alternative proprietary low-range

solutions. An example is Cybiko [CYBIWWW], who have designed and produce pocket-sized devices that they refer to as 'wireless inter-tainment' computers. Each Cybiko has a radio range of around 150m indoors and 300m outdoors, but multi-hop routing is a built-in feature of the system, meaning that Cybikos form an ad hoc network to increase the effective communication range. Whilst the lack of compatibility with other devices clearly limits the potential usefulness of Cybiko, it is a useful practical demonstration of the underlying principles and applications of wireless ad hoc networking.

Figure 2.3: A Cybiko unit [STRE04]

2.4.4 Infra-red (IrDA)

IrDA is an infrared wireless communication technology developed by the Infrared Data Association and is a specific use of infrared light as a communications medium [PREN04]. IrDA is not considered to be of particular interest for building personal area networks, but it has been included here for completeness, since it was an extremely popular technology for short-range communications before the emergence of Bluetooth and WiFi. Because infrared uses the nonvisible infrared light spectrum, IrDA communication is blocked by obstacles that block light (such as walls, doors, briefcases, and people). In addition, the effective range of infra-red is only around 1 metre and line-of-sight is generally required. The advantages of IrDA are low power consumption and low cost, however Bluetooth also fits well into these categories and provides enhanced range. In particular, since Bluetooth does not require a line-of-sight link, it is likely to prove much popular as a choice for linking devices in a personal area network.

2.4.5 Emerging PAN technologies

There is a considerable amount of ongoing research in the personal area networking arena. One of the more unusual and exciting areas is Thomas Zimmerman's technology [ZIMM96] that uses the natural electrical conductivity of the human body to transmit electronic data. Using a small prototype transmitter (roughly the size of a deck of cards)

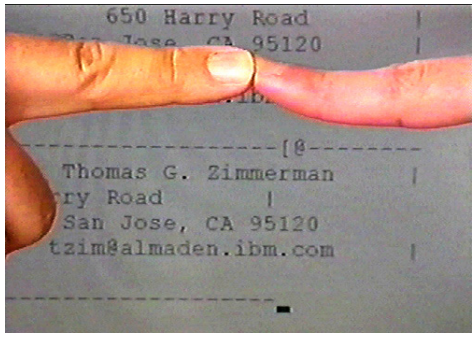

Figure 2.4: Data exchange in a Zimmerman PAN [IBMR96]

embedded with a microchip, and a slightly larger receiving device, the researchers can transmit a pre-programmed electronic business card between two people via a simple handshake [IBMR96]. The motivation behind this work is the theory that the natural salinity of the human body makes it an excellent conductor of electrical current. Zimmerman's proposed PAN solution takes advantage of this property and transports data over an external electric field. It has been shown that data transmission at around 2400 baud is currently possible, however the theoretical maximum is considerably higher than this.

Whilst there is clearly a lot more work needed in this area before such schemes are likely to be used in commercial devices, intra-body PAN networks have a range of potential advantages mainly focussed around the ability to make use of the human body as a transport mechanism and effective hub for surrounding electronic devices.

2.5 Ad hoc networking

Giordano [GIOR00] describes ad hoc networks as being typically composed of equal nodes, which communicate over wireless links without any central control. In an ad hoc network, all hosts are required to support the network and act as routers. This type of behaviour is known as multi-hop routing and enables hosts that are not directly within range of each other to communicate via one or more other intermediary hosts. An example of this is shown in Figure 2.5, where hosts A and B are not directly in range of each other but can communicate via B, which acts as an intermediary router.

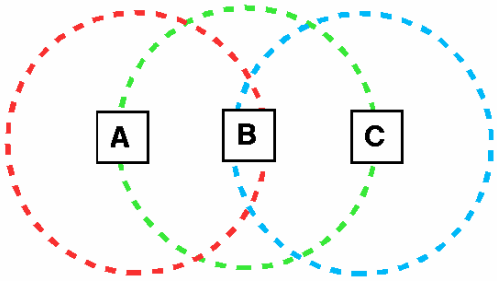

Figure 2.5: Multi-hop routing in ad hoc networks [DORS98]

Ad hoc networking is of particular relevance to the construction of personal area networks, since it is generally desirable for small handheld devices to be able to communicate with other similar devices without the need for the presence of fixed infrastructure such as DHCP addressing servers and dedicated routers. Ad hoc networks can be delivered using technologies such as Bluetooth and WiFi, which were considered earlier in this chapter.

2.5.1 IETF MANET

The IETF's working group on Mobile Ad hoc Networks (MANET) [MANEWWW] is standardizing routing in ad hoc networks. The group studies routing specifications, with the goal of supporting networks

scaling up to hundreds of routers. MANET's work relies on other existing IETF standards such as mobile-IP and IP addressing. Work produced by the group to date includes specifications for the Ad hoc On-Demand Distance Vector (AODV) routing and Dynamic Source Routing (DSR) algorithms. These are optimized routing algorithms designed specifically for use in mobile ad hoc networks.

2.5.2 Current research issues

Current research in this area falls into the following areas:

- **Addressing** – since there is no central server to allocate addresses, the problem of address duplication is harder to solve. Several proposals have been made, including an IP address Autoconfiguration scheme from the IETF in which a host uses a temporary IP address in the 169.254/16 range and broadcasts a Route Request (RREQ) packet. If no response is received within a certain period and several retry attempts have been made then the requesting host assumes that the address can be taken as its own. Otherwise the node randomly picks another address and tries again.

- **Routing –** several groups are looking into innovative routing algorithms [SCHU04] to take issues such as Quality of Service (QoS) and congestion control into consideration. A primary goal is of course to try and minimise the amount of routing overhead in mobile ad hoc networks, to reduce power consumption.

- **Resource Allocation** – this area is concerned with how to price scarce resources and how to allocate them in a way that is considered to be fair.

- **Security** – Hubaux et al [HUBA01] comment that so far research on mobile ad hoc networks has been focussed primarily on routing issues and security has been given a lower priority. We take a more detailed look at this important research area in the following section.

2.5.3 Security in ad hoc networks

Security in a mobile ad hoc network is a major concern due to its characteristics of open medium, dynamic changing topology, cooperative algorithms, lack of centralized monitoring and management point, and often lack of a clear line of defence. As Hubaux et al [HUBA01] point out, the security requirements depend very much on the kind of objective for which the ad hoc network has been conceived. Clearly, a military network will have much more stringent security requirements than an

informal civilian one. They believe that there are effectively two types of threats – attacks on basic mechanisms such as routing, and attacks on security mechanisms such as the key management mechanisms.

Attacks on basic mechanisms are possible because nodes of ad hoc networks cannot be assumed to be secured in locked cabinets; they therefore risk being captured and compromised [HUBA01]. In addition, all communications are performed 'over the air' so eavesdropping and active interference may also be a problem. We also tend to assume that nodes that carry out intermediary routing behave cooperatively, but in reality this may not be the case, leading to an unfair allocation of resources. Another serious issue relates to neighbour discovery – for example, if a Bluetooth device is not properly configured then it may be discovered and exploited by a rogue user. There are several ways that we can protect against these types of attacks:

- **tamper resistance** – this may involve embedding cryptographic data in a smart card, using the same principle as for GSM SIM cards.
- **routing-based mechanisms** – a 'watchdog' node could be put in charge of identifying nodes that misbehave, and a 'pathrater' which defines the best route circumventing these nodes. A misbehaving node would effectively be locked out of the network by its neighbours.
- **neighbourhood** – a solution based on pseudonyms has been proposed by one research group [CAPK04]. If the identity of a device changes for each session, then it becomes harder for an intruder to track its location.
- **service enforcement** – the authors propose two charging models based on a virtual currency known as a "nuglet". Nodes remunerate each other for the service they provide to each other. By doing this, a node can make use of the network only if it also contributes to the benefit of the community. Of course such a scheme requires that the virtual 'purse' be cryptographically protected to prevent cheating.

In terms of the second issue of security mechanisms, a major issue is how two parties can establish a shared secret key. A common approach is to use the Diffie-Hellman public key approach, where two parties exchange random values, from which they both compute locally the same key. The standard Diffie-Hellman proposal is open to active attacks such as a 'man in the middle' however several modifications have been suggested that counter this threat. In their paper, Hubaux et al propose

an alternative and novel self-organised Public Key Infrastructure (PKI) scheme for the safe distribution of public keys.

2.6 Hardware devices & software platforms

Since the latter part of this project encompasses the development of a 'proof of concept' simulation using a selection of devices available today, in this section we take a brief look at the most popular hardware and software platforms available today in order to be able to arrive at appropriate choices. The approach taken is to look at the key software platforms and consider related hardware devices within those sections.

2.6.1 Symbian OS

Symbian was established as an independent company in 1998 by Ericsson, Nokia, Motorola and Psion in order to promote the interoperability of data-enabled mobile phones with mobile networks, content applications and services. Symbian OS phones tend to be more sophisticated than ones that support J2ME and generally have several megabytes of memory available [NOKI03]. The two major hardware vendors of Symbian OS phones are currently Nokia and Sony Ericsson. Nokia's Symbian-based products include the 7650, 3650 and 6600 (see Figure 2.6). These tend to resemble traditional phones and have a standard style keypad but a much larger display. Sony Ericsson manufactures the P800 and P900 that are more like PDAs with phone functionality. Both of these devices use a stylus-based 'touchscreen' input system.

Figure 2.6: Nokia 6600 running the new Symbian OS v7.0 [MOBI03]

One of the biggest advantages of Symbian OS is that applications can access all hardware and software features of a phone, including SMS, WAP, infra-red ports, Bluetooth and voice telephony features. This provides developers with the ability to produce better-integrated and full-featured applications compared to a platform such as J2ME. In addition, Symbian OS applications are compiled to native machine code which means that they typically execute quicker than Java-based platforms.

There are however a few possible drawbacks. Firstly, Symbian OS applications can only be developed in the C++ language. Whilst stable SDKs are available from Symbian, Nokia and Sony Ericsson, these products arguably have quite a steep learning curve. Particular care must be taken to deal with garbage collection and memory leak issues. In

addition, whilst manufacturers that deploy Symbian OS make use of the same underlying operating system, the devices may still be incompatible since there are two different user interface standards in use today – Series 60 that is typically used by Nokia, and UIQ that is typically used by Sony Ericsson. Therefore applications developed for one platform are unlikely to work properly, if at all on the other.

2.6.2 Java 2 Micro Edition (J2ME) and PersonalJava

Many new phones and PDAs have some form of Java support, meaning that applications developed for Java can generally run on a wide range of platforms. The most common Java implementation for mobile handsets is Java 2 Micro Edition (J2ME). This is an open standard, easily learnable by programmers with Java experience and provides many of the benefits of Java technology such as automatic garbage collection. Java doesn't of course run natively however and is an interpreted language, therefore applications are likely to be slower than those written in a language such as C++. Most handsets support over-the-air (OTA) provisioning, meaning that applications can be downloaded and installed to a user's handset within seconds.

The main problem with J2ME is that due to its extremely lightweight nature, the API it provides to developers is somewhat limited. Only a handful of classes are provided and it is either impossible or very difficult to access features of the phone such as the IrDA and Bluetooth ports. This makes J2ME an unpopular choice for applications where communication with other devices is paramount. In addition, J2ME is unsuitable for large applications – the limit tends to be just 64K and even lower on many devices. Devices with J2ME support only tend to have reasonably small displays which may also hinder the usefulness of this technology.

An alternative to J2ME is PersonalJava, which provides almost all of the functionality of standard Java 1.1 plus a few extensions from Java 2. Although it is being phased out and there is a suggested migration path from PersonalJava to J2ME with some new profiles that provide additional Java classes, PersonalJava is still a popular choice for many applications that are to run on devices that typically have more processing power and memory such as PDAs. At the present time, use of the PersonalJava is particularly advantageous since JVMs are available for Symbian OS, Microsoft's Pocket PC and Windows CE, PalmOS, Sharp Zaurus (Linux-based) and several other types of devices [JAVA01]. Therefore PersonalJava arguably provides the best cross-platform ability available today.

An interesting research project (JXME) underway at the moment is developing a JXTA implementation for J2ME [JXMEWWW]. This will promote true peer-to-peer communication between devices running J2ME and increase the range of opportunities that the technology provides. The project is still in its early stages and the current implementation requires the use of a proxy for communication. In addition only the HTTP protocol is supported for communication – it is not possible to use TCP/IP sockets. The group are currently trying to develop an efficient p2p implementation given critical constraints such as small application size, limited memory and CPU power.

2.6.3 Microsoft .NET Compact Framework

The .NET Compact Framework (CF) from Microsoft is part of the company's .NET initiative targeted at 'smart devices' such as mobile phones and PDAs. The framework provides developers with a rich API and makes it relatively easy to develop applications that allow devices to consume XML Web services. A key benefit is that applications developed for the .NET CF can run on all devices that have the framework installed, making them portable [YUAN04]. Whilst it may be necessary to develop different user interfaces for different types of types of devices, the underlying code can remain the same. Also, since the .NET CF is a cut-down version of the .NET Framework, the same development tools can be used. Visual Studio .NET 2003 provides a rich IDE with emulators and debugging tools to speed up the development process. Also, developers can choose to develop in any language that is supported by the Common Language Runtime (CLR), including C# and Visual Basic .NET. It is even possible to have a single system consisting of components that were written in different languages!

Figure 2.7: HP iPAQ h5550

Devices that support the .NET Compact Framework tend to be quite sophisticated. Figure 2.7 shows a recent product from HP, the iPAQ h5550 that includes Bluetooth, WiFi and biometric fingerprint recognition. However, one of the main concerns with the use of .NET is that we are effectively restricting ourselves to a single OS platform – Windows. Whilst Windows CE and Pocket PC run on a multitude of devices from a range of vendors, there are also a range of popular platforms in existence which cannot run .NET code, such as PalmOS, Symbian and Linux-based platforms. It could also be argued that the .NET CF is still in relative

infancy and therefore in many ways Symbian OS and Java-based systems are currently more flexible for certain applications.

2.7 Bringing it all together…

Whereas much of the content in the earlier parts of this chapter have looked at specific areas of interest to this project, such as policy specification and ad hoc networking, this section looks at a range of current research projects that are bringing various aspects of these technologies together and are specifically focussed on the interaction of devices in a personal area network. The purpose of this exercise is to understand what is considered to be 'state of the art' in this field of research with the intention of discovering issues that it would be worthwhile to investigate in this project.

2.7.1 MIT: Project Oxygen

Project Oxygen [OXYGWWW] at MIT, which has an objective of *"bringing abundant computation and communication, as pervasive and free as air, naturally into people's lives*" has already demonstrated the benefits of self-management in a range of applications. They distinguish between *basic physical* and *basic virtual* objects. The former senses or actuates a physical entity, whereas the latter collects, generates and transforms information, e.g. extracting information from an incoming electronic form and sending the results on to a particular device. The project also advocates the use of a scripting language to enable the tasks that need to be automated to be specified easily and rapidly.

It is suggested that in the future computing will be human-centred and freely available everywhere. Users will not need to carry their own devices around with them and will instead make use of configurable generic devices that will adapt to their needs. They categorise these devices as:

- **Enviro21s (E21s)** – these are embedded devices which may be installed in homes, offices and cars. They communicate with each other and with nearby H21s through dynamically configured networks (N21s). The main purpose of an E21 is to sensors and actuators etc to monitor and control the environment. E21s may for example be embedded in walls and may also provide large amounts of computational power that nearby H21s can be use to offload work.
- **Handy21s (H21s)** – these are anonymous handheld devices. Rather than storing large amount of local state information, H21s configure themselves through software to be used in a wide range

of environments. H21s typically have less computational power than E21s since they tend to be smaller and use battery power.

- **Networks (N21s)** – these are built around the ad hoc paradigm. Project Oxygen defines these as 'flexible, decentralised networks that connect dynamically changing configurations of self-identifying mobile and stationary devices'. N21s deal with issues of resource and location discovery and security. Domains are referred to as 'collaborative regions' – computers and devices may belong to several regions at one time. Region membership is dynamic and devices may enter and leave different regions as they move around.

The area of greatest interest to us is that of software policies. There are two key layers in the architecture:

- **abstraction** – these characterise components that carry out computations and objects used in computations. The abstractions provide applications with specialised interfaces that avoids them having to talk directly to the underlying layers.
- **specifications** – these make abstractions explicit and contain information about the modules and capabilities available locally, sources for obtaining code across the network and details about module dependencies.

The strategy used by Project Oxygen is to hold code, data objects and specifications in a common, persistent object-oriented store that supports transactional semantics for concurrent access. This enables users to interact with software and data from any location by bringing applications 'just-in-time' to handheld devices.

2.7.2 AMUSE

The AMUSE (Autonomic Management of Ubiquitous Systems for e-Health) project is investigating self-managing adaptable infrastructures to support e-Health and e-Science applications. In the project proposal [LUPU03], the authors comment that whilst a range of advanced devices for enhancing healthcare are now available, many of which have wireless communication capabilities, there is little or no software infrastructure currently available that allows them to work together in a configurable and adaptable manner. Body sensors are also becoming increasingly smaller in size and this is increasing the opportunities for a ubiquitous computing environment that can significantly improve the quality of healthcare services provided to patients. However in order to achieve this goal, the management functionality needs to be hidden, with autonomous devices managing their own evolution and configuration changes without

the need for user intervention. The paper suggests that the issues surrounding self-management are applicable not only to healthcare applications, to which the AMUSE project is focussed, but also to other areas such as embedded devices in the home.

The project advocates the concept of a **self-managed cell (SMC)** which consists of a set of hardware or software components which function autonomously and are therefore capable of self-management. It is proposed that a cell should contain a certain set of mandatory services such as **service discovery**, **event correlation** and **policy management**. The policies specify actions that should occur in response to changes of state either in a managed object or in the environment. A cell is a 'closed-loop' system where state changes lead to events which can trigger actions that can modify the state of the system, and possibly lead to new events. An example of a self-managed cell is shown in Figure 2.8. We observe that interaction between components in the cell takes place via a common asynchronous event bus. In terms of implementation, this may be delivered using an event-oriented middleware such as Elvin that we considered in some detail earlier in Section 2.3.4.

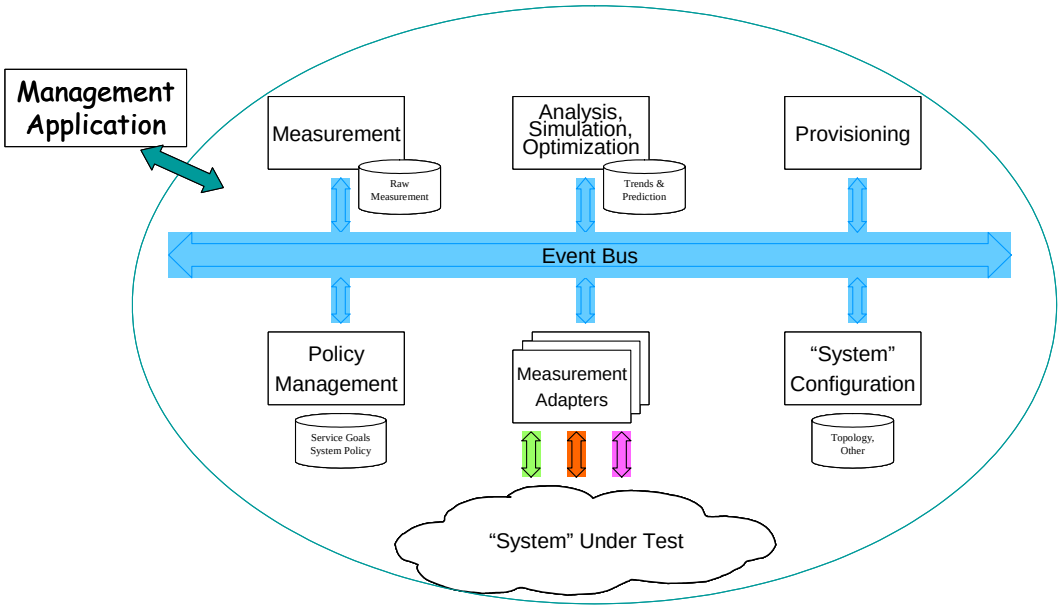

Figure 2.8: An example of a self-managed cell (SMC) [SVEN03]

Several of the issues being researched by AMUSE are of very specific interest to this project, including:

- What management functionality a self-managed cell needs to be provide, and how the management components should behave.

- How to use contextual information, i.e. information from the surroundings.
- Distinguishing between components in a cell that should be mandatory and optional. In addition, making the SMCs extensible, so that they can be specialised to particular environments.
- Defining how cells should interact with each other. AMUSE suggests 3 types of interaction: *composition* (composed SMCs form a single administrative domain, similar to subtyping in an OO environment), *federated* (peer-to-peer interactions between SMCs to provide integrated services), and *layered* (hierarchical layering of services analogous to the OSI model).
- Determining how policies should be expressed, deployed and enforced.
- How a cell should behave and adapt when new resources and services are dynamically added or removed. In addition, we are interested in how these entities can be discovered.
- Instantiation of an SMC and the deployment of its components across many distributed components.

The AMUSE project was only instantiated a few months ago and is still in its early stages, however concepts such as the ones defined above provide us with helpful pointers to the types of current research issues that will require consideration in this project, in order to develop a policy-driven middleware for personal area networks. An area that we will certainly give particular attention to in this project is how we can use policies to not only specify interactions between devices, but also how policies can be used to define the behaviour of management components, making the architecture highly flexible.

2.7.3 Autonomic computing

Autonomic computing is a term coined by IBM to describe a computing system that possesses at least one of the following four attributes [BANT03]:

- **Self-configuring** – automates the installation and setup of its own software.
- **Self-healing** – monitors its own platform, detects errors and automatically takes remedial action as necessary.
- **Self-optimising** – optimises use of its own resources.

- **Self-protecting** – automatically configures and tunes itself to achieve security, privacy, function and data protection goals.

Autonomic functionality can be implemented at different levels, including locally e.g. power management, within a peer group e.g. knowledge-sharing in a "grid" computing environment or network-based e.g. remote backup. Bantz et al propose a general architecture for autonomic systems and Figure 2.9 shows the building block, known as an autonomic element (AE). Each AE consists of an autonomic manager (AM) and a set of managed components. Managed components communicate their events to the local AM. The AM makes decisions based on policies, facts and rules which are held in a database as well as input received from the managed components, and communicates directives and hints to them.

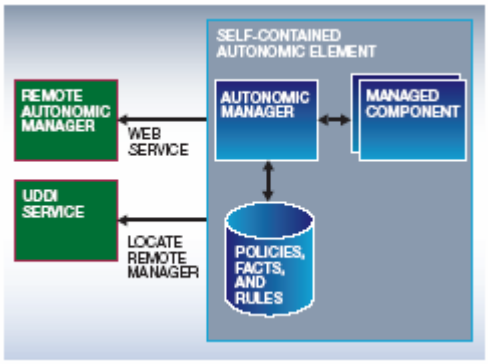

Figure 2.9: An autonomic element – the basic building block of autonomic systems [BANT03]

An interesting approach to the problem of exerting control over objects in a hierarchical system is proposed. Two styles of control are suggested, *delegation* and *guidance*. In the former, a local AM passes control of some of the resources it manages to a superior, whereas in the latter, a local ATM receives information from its superior and implements then with respect to its own resources. Only one AM is ever in direct control of a resource. Control over a selected set of resources can be delegated to another manager through the use of *client virtualisation,* in which we can define virtual AMs that group resources together and link them to a remote AM.

There are a number of other useful sources of information in this arena. Dr Mitchell Waldrop [WALD03] in his recent article on *Autonomic Computing: The Technology of Self-Management*, refers to a "continuous control loop", i.e. each component of the system (hardware and software) should now only know how its assigned tasks but should also have internal mechanisms that constantly monitor its own operation, and make corrections as needed. A key feature of such a scheme is that each device would handle as much as possible locally – and yet still have the means to call on the larger system when it needs help. Dr Waldrop also suggests that the scheme could be recursive, so that when the call for help reaches that system, it may decide to call for help to a still larger system for help.

2.7.4 Summary of findings & directions for focus

In this section we have looked at a selection of related research projects. Whilst a considerable amount of work has already been carried out in the more general areas of device interaction in ad hoc networks and autonomic computing, there has been very little focus to date on developing a common policy-driven middleware for personal area networks. For example, whilst MIT's Oxygen project proposes a ubiquitous **human-centred architecture** in which the management functions are effectively hidden from users, the emphasis is on allowing users to interact with the system using speech and vision techniques, in a similar way to how they might interact with other users. In addition, the project focuses on allowing users to pick up "anonymous" handheld devices which adjust to their needs for a relatively short period of time, rather than personal devices such as mobile phones that tend to be owned by a single user. In this project, we wish to tackle the issues surrounding the self-management of consumer devices that can communicate over a very short distance rather than the transfer of user data between such devices or human-computer interaction.

The AMUSE project is much more closely aligned to this work. Whilst AMUSE is looking at solutions for e-Health, the principles behind the self-managed cell can be equally applied to consumer devices in personal area networks. It therefore seems appropriate to utilise the SMC concept as a basis for this work, however a number of additional issues have been identified which will be given more specific focus:

- **Prerequisite cell components** – a core set of components needs to be defined for our architecture, specific to the needs of consumer personal area networks. The behaviour of these components also needs to be specified and the interactions defined.
- **Device detection / discovery** – whilst work has already been done in this area, the discovery of devices is of particular importance in personal area networks, since devices may be frequently entering and leaving them. How do we detect that devices have left the network and when should we inform other interested parties about this?
- **Context-sensitive behaviour** – external events should trigger state changes as well as internal events generated by other components in the system. We consider variables such as time, temperature and location as external events. In addition, it should be possible to evaluate policy conditions against the current state of external variables, to determine if those policies should run when triggered by another event.

- **Efficient event generation and consumption** – since the devices we are targeting will typically be battery-powered, we will investigate ways of minimising the management traffic. For example, it is inefficient for devices to have to continually transmit details of their state, if that information is not required by other entities.

- **SMC policy** – earlier in this chapter, we considered obligation and authorisation policies, such as those implemented by the Ponder language (see Section 2.2.1). Rather than simply using these policies to define interactions between devices, we will look at how we can specify SMC policies that control the behaviour of the system, e.g. to define the behaviour of the device discovery mechanism. The intention is to make our proposed architecture as flexible and configurable as possible.

- **System policy vs user policy** – we will divide policies into these two distinct categories. System policies tend to be relatively static and embedded at the time of manufacture, or held in flash memory. These policies define standard behaviour towards other devices and in addition specify the capabilities of user policies. For example, system policies may state that when a text message arrived on a mobile phone, it should also be made available on all PDAs within the cell. User policies allow users to customise their devices to an extent, as defined by the corresponding system policies. Users may wish to override specific behaviour to meet their needs. For example, a user policy may be used to define a call diversion from one mobile phone to another when the battery of the first device is about to run out. The user specifies the policy, since they define where calls are diverted to.

3 Project Specification

This chapter specifies the requirements for our policy-driven middleware architecture. We first define the scope of this work to narrow down the problem domain and then consider the types of usage interactions that will be made with the system. This work is continued in Appendix A1, where we provide a list of system requirements.

3.1 Scope

From the background research that has been carried out into related technologies and projects (see Chapter 2), we observe that the field of policy-driven middleware is quite a broad one, and there are many potential issues that we could investigate and attempt to provide solutions for in our architecture. Due to time constraints, this project is unable to address all of these issues, and it is therefore important to formalise the scope of this work as follows:

- **Language-independent architectural design** – the main deliverable of this project is a detailed design for a flexible and extensible middleware that is not tied to a specific choice of implementation language or tools. The second deliverable will consist of an implementation based on this design, as a demonstration of the capabilities of the middleware.
- **Device grouping and profiling** – we will define how consumer devices in a personal area network can be grouped by common functionality, how this functionality can be defined, and how devices can belong to more than one group or domain.
- **Device discovery** – the work of discovering devices that have entered a cell, and those which have disappeared will be the onus of a discovery manager component rather that functionality that each device needs to implement. We will define the behaviour of this component.
- **User customisation of behaviour** – our middleware will support the definition of policies that allow users to customise the behaviour of the system to the required level.
- **Policy syntax** – we will define the notation for policies under our architecture and provide examples of acceptable policies for a selection of scenarios.
- **Efficient event bus** – we will look at the problem of passing events between devices in an efficient manner, with particular attention to reducing the amount of management (overhead) traffic.

- **Event correlation** – we consider how events can be aggregated and the specification of policies that can use these aggregation features, e.g. carry out a particular action if event x has occurred more than 5 times in the last 60 seconds.

The following issues are considered to be out of scope:

- **Policy compilation** – whilst we will specify syntax for policies used in our architecture, and provide examples of valid policies, we will not implement a compiler or modify existing compilers to generate executable code from these.
- **Cell interaction** – we do not consider how cells can be placed together to form larger structures, or how they may be able to work co-operatively in a peer-to-peer fashion. These areas are being investigated as part of the AMUSE project.
- **Mechanisms for transfer of user data** – we are concerned specifically with event-based action-condition-effect interaction between devices in a personal area network. We will not consider methods of sending other types of data between devices, e.g. music files. These are wider mobile ad hoc networking issues.
- **Intelligent human-computer interaction techniques** – we will not implement interaction between users and the system using speech and vision techniques. A key objective of this project is that the system should work in the background, with minimal user intervention.
- **Design of context sensors** – whilst we will consider external context-sensitive events, we will not specifically consider the design of sensors that can provide this functionality. Our flexible and extensible architecture will however allow for new types of context devices to be added.
- **Security & trust issues** – there are many issues in the area of security and trust including building a security model using authorisation policies. However our goal is to develop a simple, working policy-driven middleware architecture that can later be extended to incorporate security features as part of future work.
- **Low-level ad hoc networking** – for example, we will not consider optimal algorithms for multi-hop routing. Much work has already been done in this area. We will utilise an ‘off-the-shelf’ ad hoc networking solution.

3.2 Use cases

The following use cases highlight the desired functionality of our middleware solution. Note that since our architecture is inherently designed to involve little human intervention, many of the use cases refer to actions performed by devices. The concept of use cases comes from the Unified Modelling Language (UML), though we have chosen not to use the full notation; we are more concerned with capturing the types of activities that devices may perform, so that we can be sure that they are included in our design, which is presented in the following two chapters of this report:

- **Cell instantiation** – cell management components must be started in the correct order, before devices can be allowed to connect to the cell.
- **Discovery of a new device** – a discovery server will check for new devices on a regular interval. If new devices are found, it is necessary to inform the domain system about these changes.
- **Communication with a device lost** – the discovery server should check that devices that appear to be connected to a cell are alive and contactable. If communication is lost, the domain system should be informed.
- **A device changes state** – a state change should result in one or more events being placed on the event bus. However if there are no consumers listening, then we would prefer that these events were not put onto the bus for reasons of efficiency.
- **A device subscribes to an event** – the events engine must register the subscription and ensure that devices producing this event are notified that the quench should be lifted.
- **A device unsubscribes from an event** – the events engine must remove the subscription and inform producers of that event that the number of consumers has dropped by one.
- **A user modifies one or more user policies** – user policies allow users to customise aspects of the system. Upon loading of user policies, the system should check that the user is authorised to carry out that action.
- **System policies are modified** – these modifications take place via firmware updates. The system must be shutdown and brought back up for this activity.

- **SMC policies modified** – these are configuration policies. The system should be fully shutdown before making any modifications, and should be brought back up once the changes have been made.
- **Cell standby** – this mode allows the cell to go into a 'sleep' state such that it can be resumed with minimal effort. The data for all the key management components must be written out to disk and held in a place that can be accessed when the system is restored.
- **Cell shutdown** – this is a full shutdown of the system. No data is written out, and a restart after a full shutdown will result in policies having to resubscribe with the events engine.

3.3 Summary

In this section we have defined the scope for our middleware architecture, by distinguishing between the areas that we will focus on, and those that will not be considered as part of this work. In addition, we have looked at how the system should behave in response to actions carried out by the 'users' of the system.

4 Architectural Design Overview

This section is the first of two that presents the design for our proposed middleware. The purpose of this chapter is to provide a high-level design for the whole system, including the key components, their interactions and expected behaviour. The material here forms the foundation for the detailed architectural design, which follows in the next chapter.

4.1 Self-managed cell structure

Figure 4.1 shows a conceptual overview of a **self-managed cell (SMC)** under our proposed architecture.

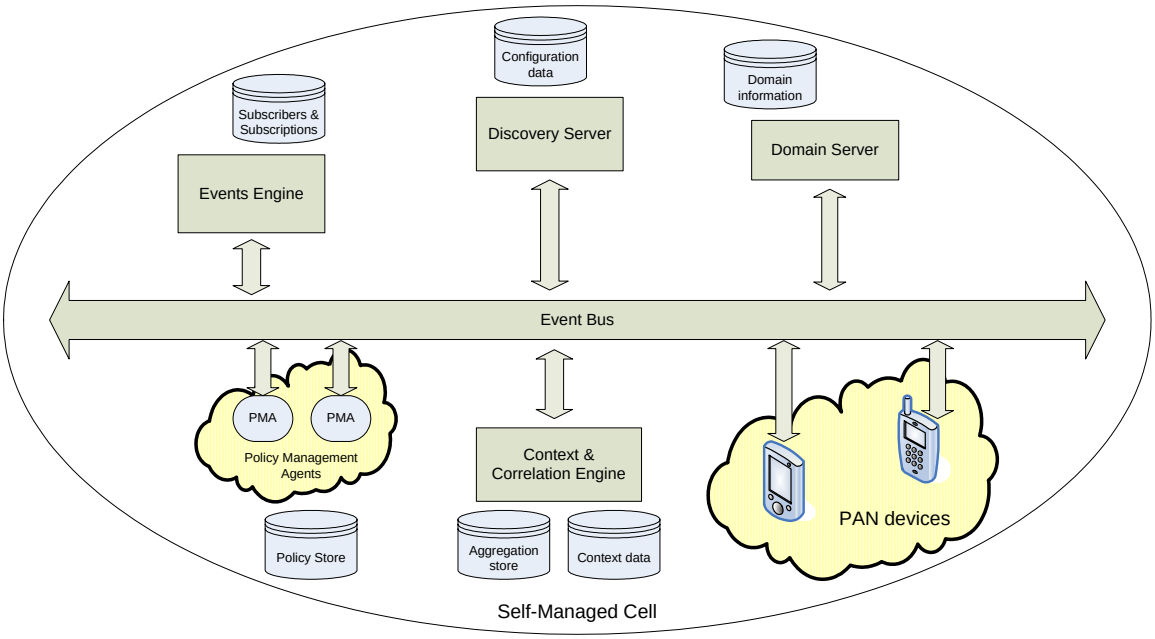

Figure 4.1: Self-managed cell, consisting of management components, devices and an asynchronous communication bus.

Our self-managed cell is based on the concepts that have been proposed by the AMUSE project (see Section 2.7.2). The cell consists of a set of management components along with their respective data stores, policy management agents that are responsible for executing one or more policies and a set of hardware devices that form a Personal Area Network. Note that we have shown the PAN devices as being part of the cell itself, since this is a conceptual model of the system. In an actual implementation, the cell would contain adapters that interact with their respective hardware devices. The key components are as follows:

- **Asynchronous event bus** - At the heart of the self-managed cell is an asynchronous event bus that is used as the means of communication between all entities in the cell. Note that devices

do not communicate with each other directly, and all communication takes place through the generation of events that are placed on the event bus.

- **Policies** - Policies are stored as part of the cell and consist of system policies and user policies. The **system policies** are those that are not user configurable and are typically semi-static, i.e. they tend only to be modified by updates to the firmware. **User policies** enable extra configurability, as permitted by system policies. The cell also contains a series of **policy management agents (PMAs)** that are responsible for executing one or more policies, and it is these entities that effectively manage the policies. A PMA evaluates each policy that it is responsible for at start-up, and translates the policy data into an event subscription. The subscription is registered with the *events engine*, which then notifies the PMA if an event meeting those requirements occurs. This prevents the PMA from having to constantly poll to check the state of devices. In a simple model, there would be a one-to-one relationship between a policy and a PMA.

- **Domain server -** The domain server maintains a domain structure which essentially enables for devices to be grouped based on the functionality that they offer. Policies can refer to a domain rather than a single entity, and in this case the domain server is responsible for ensuring that the relevant actions are carried out on each of the members of that domain, including propagation through to any sub-domains if they exist. As well as devices, the domain server could also be used to organise applications, services, users, policy objects and other resources.

- **Discovery server -** The discovery server maintains a list of devices attached to the cell and regularly sends them an “are you alive” event which it expects to receive a response to. If a response does not arrive, retries are made until a maximum number of attempts is reached. The device is then removed from the list of active devices, and an event is placed onto the bus to inform the domain server that the device should be removed from any domains of which it is a member. New devices are found by the discovery server through a broadcast event that is picked up and responded to by devices that are not attached to a cell. The device is then added to the list that the discovery server maintains, and an event is placed on the event bus for the domain server to be able to add it into the necessary domain(s). The discovery services, could also potentially be used to discover new services, users, and other resources, and act as an “ORB” or “registry” for applications.

- **Context & correlation engine** - The context & correlation engine (CCE) is a supporting component that enhances the usefulness of management policies. In terms of context, the engine regularly updates its own store of information on items such as time and location, as well as direct readings taken from any context sensors that are in range. Devices can request that the CCE provide data for a particular context value or set of values via an event, and the information is returned as an event as well. The correlation functionality allows for **'derived' events**, i.e. events which are based on other events. An example is an event A that occurs if event B occurs at least 5 times within the space of 10 minutes. These aggregate events are also defined by policies; however the CCE is responsible for carrying out the aggregation and raising events when the relevant conditions are met.

Here, we have considered the basic structure of a self-managed cell and its key components. The following sections of this chapter describe specific aspects of the middleware architecture in more detail, including the rationale for design decisions that have been taken.

4.2 Policies and policy management

We will use the Ponder policy specification language (with a few of our own extensions) to specify policies in our system. Whilst the deliverables of this project do not include a Ponder compiler to translate our policies in to a code-based implementation, the notation provides a concise means to capture the policy behaviour.

4.2.1 System policies for device behaviour

The main type of policy in our architecture is the system policy. As described earlier, these policies will be semi-static and the intention is that they will be stored in flash memory as part of the cell management firmware. We use a form of obligation policy to define each policy as follows:

```
inst oblig policyName "{"
        on              e = eventName;
        subject         domainPath ;
        target          t = domainPath ;
        do              actionList ;
        [ when          constraintExpression ; ]        "}"
```

Each policy has a unique name *policyName* and is a member of a group of policies. The group is defined using the same domain structure as we use for devices, however policies have their own hierarchy, e.g. /policies. The policy *subject* refers to the domain to which the policy belongs and the

target refers to the domain on which the actions are to be performed. The *do* value is a list of actions to be carried out, separated by commas. The *when* clause is optional and is used to specify conditions such that the actions are only carried out if those conditions evaluate to true.

The constraintExpression is written using UML's Object Constraint Language (OCL) notation, which is a superset of ordinary Java conditional notation. We can make use of e and t to refer to the event and target respectively. For example:

when (e.status == 1 || e.status == 2) && (t.batteryPercent < 60)

The "e" conditions are evaluated on the event itself and are used to build the subscription that is sent to the events server – this is discussed in the next section. Conditions with a "t" prefix are evaluated on the target and the do actions are only carried out if they evaluate to true. Policies that use context information and correlated events are discussed later in Section 4.6.

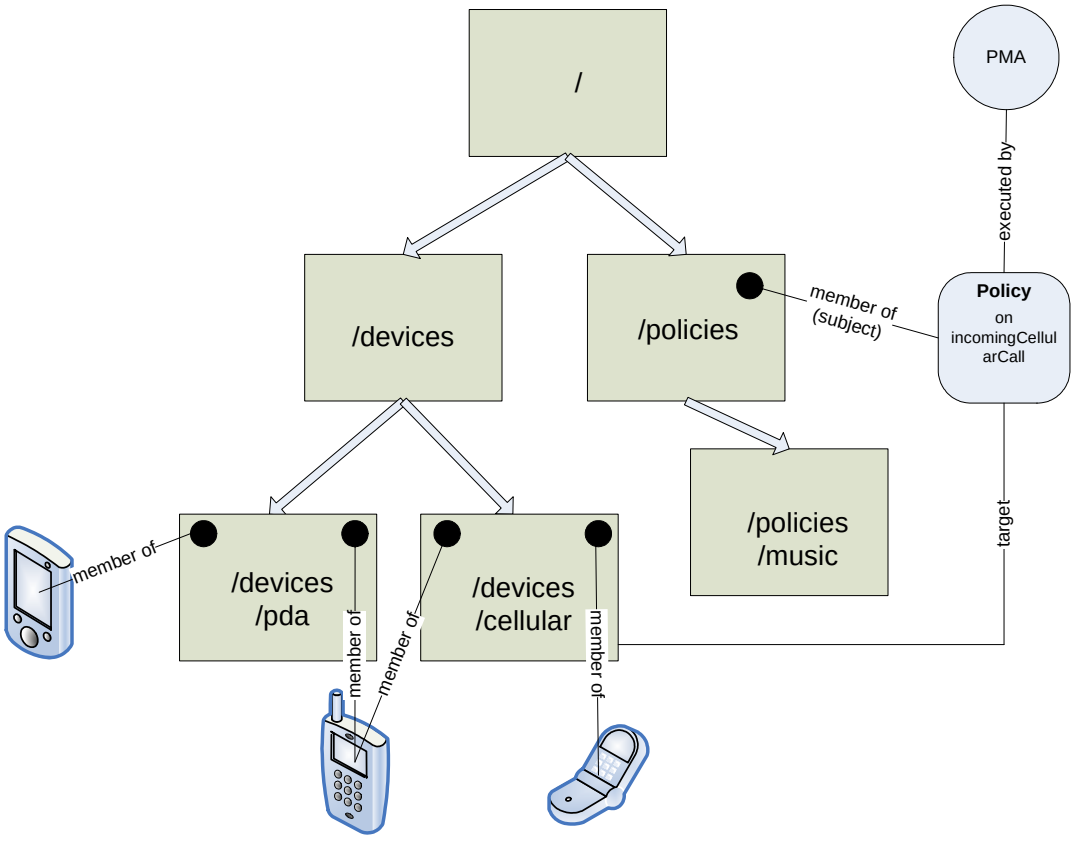

Figure 4.2: A sample domain structure with devices and one policy

Figure 4.2 shows a sample domain structure. The policy is a member of the /policies domain and acts on the /devices/cellular domain. The policy management agent (PMA) collaborates with the events engine and is

notified when the event has occurred. This corresponds to the following policy definition:

```
inst oblig incomingCall "{"
        on              e = incomingCellularCall;
        subject         /policies ;
        target          t = /devices/cellular ;
        do              actionList ;                    "}"
```

In this example, there are no *when* conditions, however if these did exist, the PMA would first evaluate these and then carry out the actions if appropriate.

The policy management agent is not defined in the policy itself and it is the responsibility of the agent to activate and manage any policies that it is in control of. A policy is simply a data object and cannot execute on its own. Most modern policy systems such as future implementations of Ponder will disseminate and/or exchange policies using XML. The reason why we distinguish between a policy and a policy management agent is that the latter can manage more than one policy. A policy management agent is envisaged to be a concurrent process or thread when implemented and it is possible to use this feature to form a trade-off between the number of threads/processes and the level of concurrency. In an environment where it is inefficient to have too many threads, we may choose to have one PMA responsible for a set of policies. However it should be noted that the mapping of PMAs to proceses/threads is not the overriding concern when writing policies. This is more of an implementation issue.

4.2.2 User policies for customisation

User policies allow users to customise the behaviour of the system by replacing policies or defining new ones. However, it would be unwise to allow users to be able to modify all policies, since this may have an adverse impact on the overall system. Positive authorisation policies should therefore be defined using the following syntax:

```
inst auth+ policyName "{"
        subject         domainPath ;
        target          domainPath ;
        action          installPolicy(p) ;
        [ when          constraintExpression ; ]        "}"
```

The *subject* specifies the source of the request, e.g. members of the domain /devices/cellular. The *target* is the domain at which the policy is conceptually enforced, e.g. by members of the domain /policies/music. In this case, we enforce the **installPolicy(p)** action. The *action* is the method that requires an authorisation check on whether it is permitted or

not permitted. The *when* clause can be used to specify additional restrictions. For example, if it should only be possible to override *incomingCallPolicy* or *incomingTextMessagePolicy in the* target domain, then we should specify a when clause of *target.member == "incomingCallPolicy"* || *target.member == "incomingTextMessagePolicy"*.

The syntax for user policies is the same as for system policies discussed in the previous section. The addition of a user policy that has the same name as a system policy causes the system policy to be "shadowed" – the user policy therefore takes precedence.

4.2.3 System configuration policies

Configuration policies are used to define the behaviour of internal aspects of the system, rather than to define interactions between devices. We propose the following syntax:

```
inst config componentID "{"
        ( variable          value ; )*                  "}"
```

Each component in the system should have a unique *componentID* and the policy consists of a series of name/value pairs. The component reads in the policy upon start-up and is configured based on that data.

An example of a configuration policy is:

```
inst config discoveryServer "{"
        pollInterval            30000       -- in milliseconds
        timeToLive              10000       -- in milliseconds
        maxRetries              5                   "}"
```

We specify the configurable variables as part of the documentation for each component.

4.3 Events system

In the background study on middleware (see Section 2.3), we considered object-based, event-based and message-based approaches. The event-based method has been chosen for our architecture since we require a lightweight, asynchronous mechanism for communication between the devices and management components in our self-managed cell. A request-reply scheme may impose a higher overhead, due to the need to acknowledge requests, and less flexibility due to unnecessary blocking of threads/processes. In addition, our primary goal is to minimise the latency of the system, since events that arrive even slightly late are to be of very little use. The queuing of messages, and ensuing in-sequence delivery is not as critical, hence the choice of an event-based approach rather than a message-based one.

The events bus forms the backbone for our distributed cell architecture. Each device and management component can execute independently, and may potentially be implemented in different programming languages and may even run on different hardware platforms. Each device simply requires an adapter that will allow it to subscribe to events, place events onto the event bus and be notified when relevant events occur.

Rather than implement our own event bus and engine, we will use DSTC's Elvin product (see Section 2.3.4) which provides an engine to store and manage subscriptions, and client adapters that can connect to a range of popular programming languages including C, Java and Python.

We make the assumption that Elvin's event delivery system is reliable, however the product does not provide a guarantee of this. The one-shot paradigm and desire to operate as fast as possible may occasionally result in the loss of events.

4.3.1 Subscription mechanism

Figure 4.3 illustrates the subscription mechanism used in our architecture, with 2 policy management agents as an example.

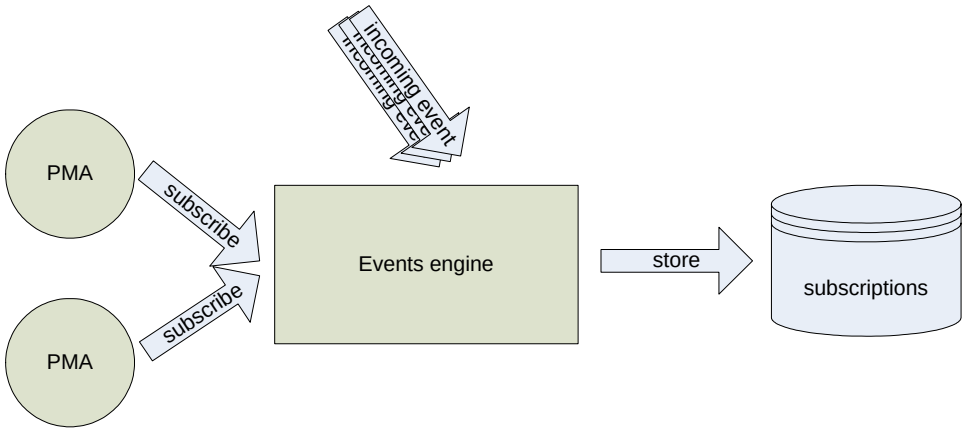

Figure 4.3: PMAs create subscriptions which are registered with the Events engine.

Each PMA object is a consumer of events and translates the policies it is managing into subscriptions. The 'subscribe' action sends the subscription across to the events engine which stores it locally. A consumer can only receive events if it is subscribed to them.

In order to be able to express how a subscription can be made, we first need to consider the format for events. Note that Elvin refers to events as 'notifications'. Each event consists of a mandatory block, containing the *eventName* and a generation timestamp *timestampGen* which is the time at which the event was created. In addition, there are one or more name/value pairs for the event's data:

eventName:	"eventName"	} *mandatory*
timestampGen:	YYYYMMDDHHMMSS	}
dataVariable:	dataValue	} *1 or more*

The translation from policy to subscription is then done by firstly expressing the name of the event that we wish to be notified about:

on eventName ➔ (**eventName** == "eventName")

Next, the *when* conditions in the policy are analysed, and those that refer to the event, i.e. with a prefix of "e" are translated by dropping the prefix. For a simple condition:

e.condition == "value" ➔ (condition == "value")

Logical AND operations (&&) are then inserted between each of the clauses. For example:

(eventName == "onCellularCall") && (status == 1 || status == 2)

Note that conditions on the event are evaluated by the events engine, rather than the PMAs, since these can be included as part of the subscription. The events engine only sends the event across if it matches the conditions specified in the subscription.

4.3.2 Event generation & quenching

Devices that generate events onto the event bus are known as 'producers'. Producers do not have to worry about the destination of an event – it is up to the events engine to pick up the event and work out which subscribers it should be sent to, depending on the conditions specified in the subscriptions. This decouples the consumers from the producers. As described above, an event has a very simple structure.

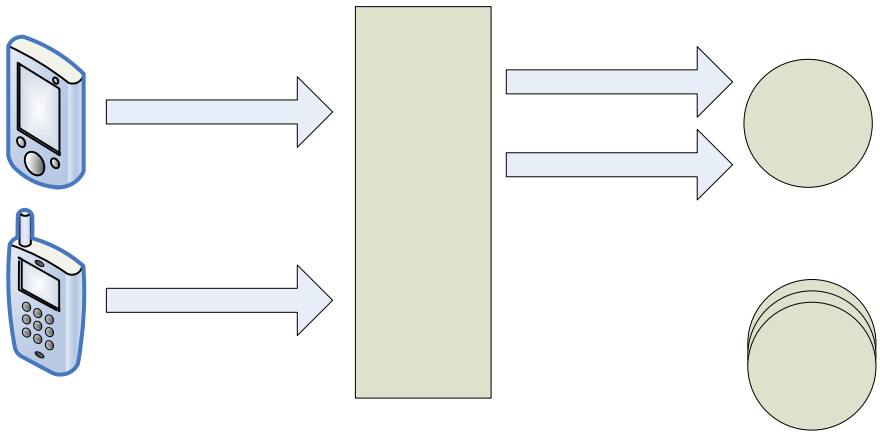

Figure 4.4: The events engine forwards events based on subscriptions.

The only mandatory fields are *eventName* and *timestampGen* which represent the name of the event and the time at which it was generated respectively. Figure 4.4 shows an example of a scenario where two devices both send different events to the events engine. The events engine maintains a list of subscriptions and therefore only forwards those events on to those subscribers – in this case a single PMA. The other PMAs do not receive any notification of these events.

In our requirements for this system, we stated that power consumption must be minimised wherever possible, since the devices in a personal area network are likely to be battery-powered. In order to meet this objective, a 'quenching' mechanism is used in our architecture. Elvin refers to quenching as 'reverse subscriptions'. Devices that generate events receive quench events from the events engine when the interest in that event changes, i.e. when a PMA subscribes or de-subscribes from it. The device maintains a counter for each event that it is capable of generating and updates these based on the quench events. If a counter is zero, then the event is disabled, otherwise events are written onto the bus as normal.

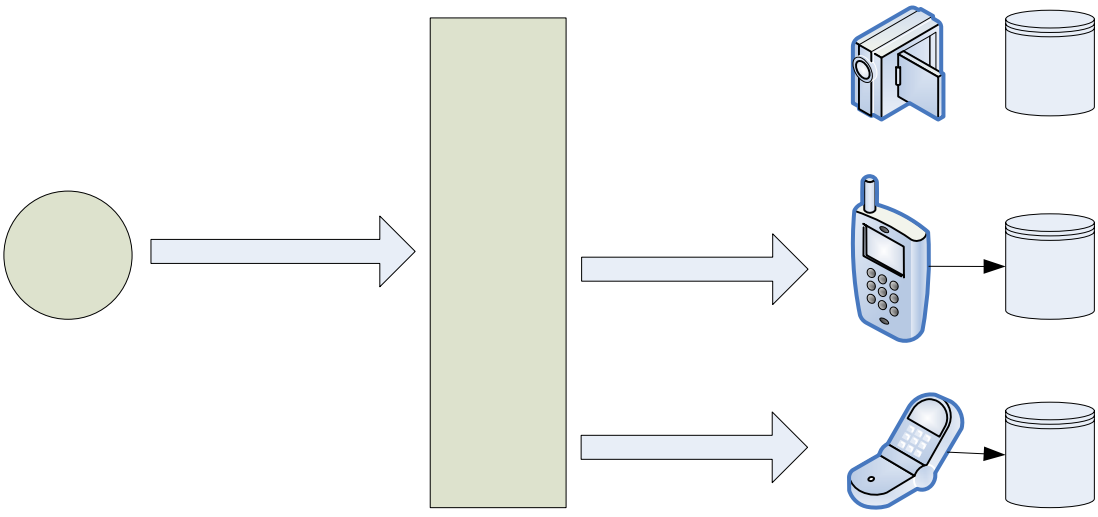

Figure 4.5: A new subscription causes quench messages to be sent to producers of those events.

Figure 4.5 shows an example of quenching. A policy management agent subscribes to an event *textMsgAlert* and this causes quench events to be sent to two devices that are producers of those events. The devices are informed that the number of consumers has increased by 1, and they update this in their own data store. The events engine knows to only send the quench events to those devices since they have set up quench reverse subscriptions upon initialisation, which inform the events engine about the names of events that can be produced by that device.

Quenching is a relatively new feature that is available in more recent versions of Elvin. In the detailed design we consider how this feature can be integrated in to our middleware.

4.4 Domain management agents

The domain structure provides the ability to group policies and devices in a hierarchical fashion. We could also use the domain system to group applications and services etc. Earlier, in Figure 4.2, we observed that devices can belong to one or more domains. If a device belongs to two domains, for example, it will be affected by policies that act on both of the domains. The membership of a domain is stored within its Domain Management Agent (DMA). The DMA is the central point of interaction between other entities and the members of the domain. Actions from Policy Management Agents (PMAs) to a domain are received by the DMA which carries out the action on all members of the domain. Figure 4.6 shows an example, where actions from a PMA arrive at the /devices/cellular DMA and are sent to the two devices within that domain.

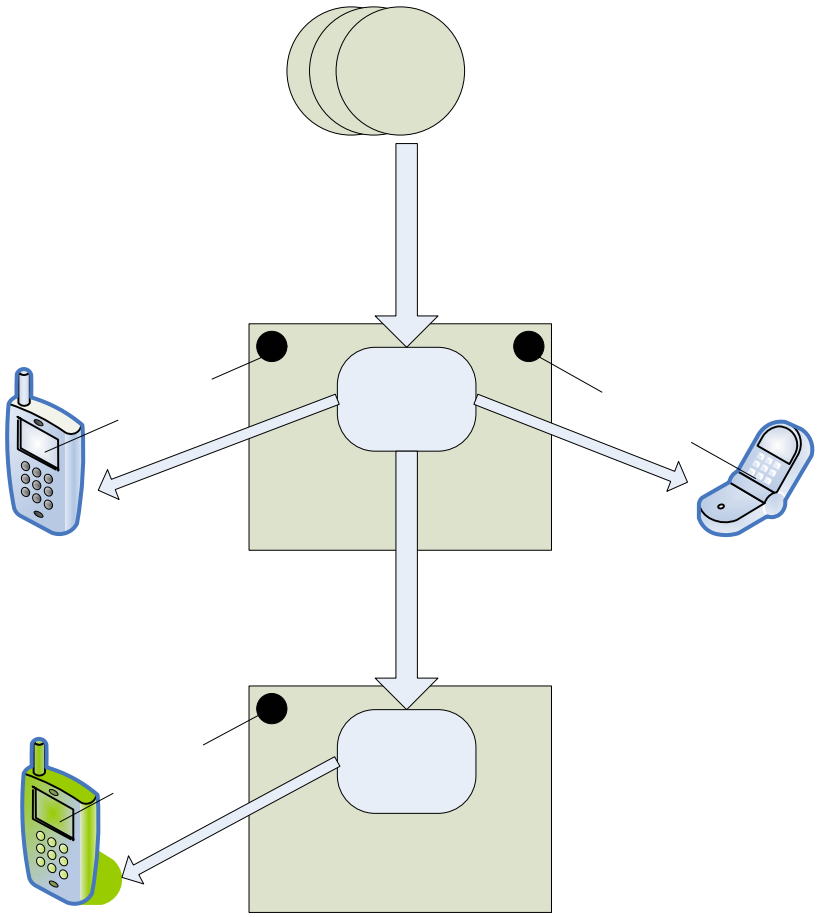

Figure 4.6: An example showing a domain and its sub-domain. Actions performed by PMAs are sent to the devices in the domain as well as to devices in all sub-domains.

In addition, the DMA propagates the action on to its subdomain /devices/cellular/3g, whose DMA in turn carries out the action on the single device contained within that domain. Note that each domain has its own DMA – there is a 1:1 relationship between these objects.

The actions sent between the PMA and a DMA, and from the DMA to its devices are also transmitted using the common event bus, rather than via remote procedure calls or a similar mechanism. We have chosen this design for simplicity. An action is therefore simply an event, but with a slightly different structure:

eventName:	**"action"**	
actionTarget:	devicePath | domainPath/DMA	
actionConditions:	"conditionalExpressionList"	} optional
actionData:	"actionList"	

The system reserved eventName "action" specifies that this event is an action that is to be performed on devices. The actionTarget specifies where the action should be delivered – this is either the full device path, e.g. /devices/cellular/7600PHONE or the DMA, e.g. /devices/cellular/DMA. The actionConditions is an optional name/value pair that is used to specify a conditional expression that should be tested at the device in order to determine whether or not to carry out the action. The actionData specifies a list of actions that should be executed.

Each DMA and device must therefore subscribe to the "action" event in order that they are able to receive these events. An example of such a subscription is:

eventName == "action" && actionTarget == "/dev/cellular/7600PHONE"

A DMA would subscribe in a similar fashion, using "*domainName*/DMA" as the actionTarget.

The actionConditions are only evaluated by devices and not by DMAs. If the actionConditions clause evaluates to false, then the action is simply discarded without being carried out.

The membership of a domain is altered by the use of *addDevice* and *removeDevice* events that are sent by the domain server to a DMA. These simple events contain a *deviceID* that is unique to each device. Further details on the device addition and removal process can be found in Section 4.5.1.

Note that we have assumed that the events system is reliable and that messages are not lost during transmission. However in reality this may not be the case. The DMAs (and PMAs that we considered earlier) may need to be able to operate in the presence of lost events. In addition,

when a DMA carries out an action on a domain, the actions do not take place across the multiple affected target objects in an atomic manner. Just as in the databases arena we aim to adhere to the ACID properties, a possible extension to this work may look at how we can define similar properties for our domain.

4.5 Device discovery

The discovery service stores the following data internally for each device that is within its range:

Variable	Description
deviceID	The unique ID of the device – in a WiFi environment, this will be the MAC (Media Access Control) address.
connectedSince	A timestamp representing the point since which constant communication with the device has been maintained. Format YYYYMMDDHHMMSS.
numRetriesLeft	An integer representing the number of retries remaining to contact this device, before it is considered to be "lost" from the cell. This value is decremented each time a "ping" request is sent and no reply is received.

Figure 4.7 shows the events that are generated and received by the discovery server. Information about new and lost devices is communicated to the domain server via events that the discovery server places onto the event bus.

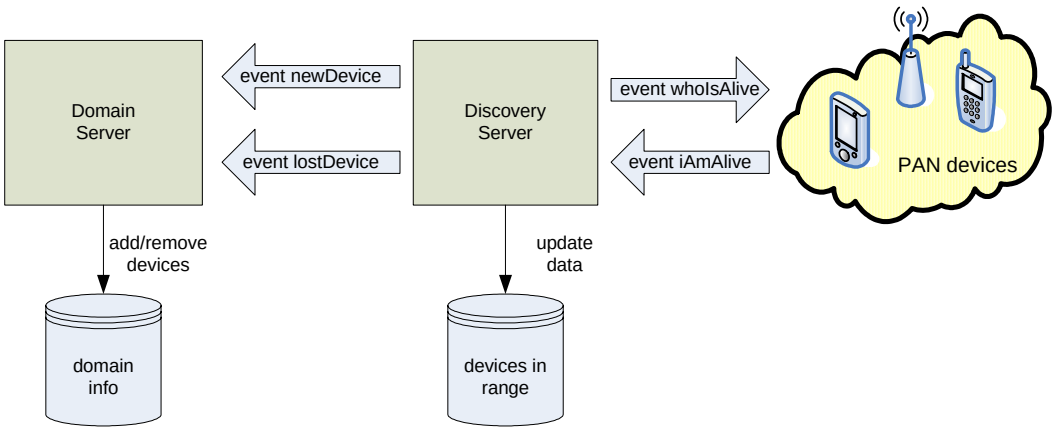

Figure 4.7: The discovery server checks for devices in range and communicates this information to the domain server.

The following pseudo code illustrates the behaviour of the discovery server:

```
until (process terminated) do {
      send event whoIsAlive
      sleep for timeToLive
```

```
        for each event e of type iAmAlive in received
        events {
                if e.deviceID exists in data {
                        // Existing device
                        set numRetriesLeft to maxRetries
                } else {
                        // New device
                        create new data entry for device
                        set deviceID to e.deviceID
                        set connectedSince timestamp to CURRENT
                        set numRetriesLeft to maxRetries
                        // Inform domain server about new
                                device
                        send event newDevice(deviceID,
                                e.profile)
                }
        }
        // Scan through the devices to see if any should be
                removed
        for each device d in data store {
                if d.numRetriesLeft == 0 {
                        // Inform domain server about removal
                        send event lostDevice(deviceID)
                        // Destroy data we hold on this device
                        delete d
                }
        }
        sleep for pollInterval
}
```

The pollInterval, timeToLive and maxRetries variables are defined by a configuration policy, as described in the next sub-section.

4.5.1 Device profiles

Devices should subscribe to the whoIsAlive messages in order that the discovery server can contact them. The iAmAlive event that the devices generate in response should contain the deviceID as well as the profile string. We define a profile string, as a list of profiles that the device supports. For example, a smart phone may have a profile string of:

“pda;phone;music”

The profiles are semi-colon delimited. The profile string is sent to the domain server as part of the newDevice event and it is used to determine which domain(s) the device is placed in. There may be a one-to-one mapping between profiles and domains, however a more complex relationship is possible if required. For example, rather than place the device above in three domains /devices/pda, /devices/phone and /devices/music, the domain server may decide to place the unit in a single domain /devices/smartphones. These decisions are made by a set of rules that are held by the domain server.

Once the domain server has decided which domains the new device should belong to, an *addDevice* event is sent to the relevant DMAs. This event includes the deviceID that is used as a unique reference to the device.

4.5.2 Discovery server configuration policy

The discovery server has three configurable variables. The values for these variables can be defined by modifying the following configuration policy – default values are shown:

```
inst config discoveryServer "{"
        pollInterval            30000       -- in milliseconds
        timeToLive              10000       -- in milliseconds
        maxRetries              5                   "}"
```

These values are read in by the server at start-up – if they are changed, the server should be shut down and brought back up again.

4.6 Context & Correlation

The context & correlation engine provides support to the Policy Management Agents in order to enable more useful policies to be written. The policies that we have looked at so far are relatively simple, in that on occurrence of a particular event, one or more actions are carried out if specified conditions evaluate to true. The actions we have considered are based on the state of the event itself or the target on which the actions are being carried out. Clearly, it would be useful to be able to base our conditional expressions on "external" events such as time or location. In addition, we may only wish to perform an action if a set of events occurs in a particular pattern, rather than the occurrence of a single event. Solutions for these issues are presented separately in the following sub-sections.

4.6.1 Context data

Policies that include a *when* clause can make use of context information by using the reserved word **context**, followed by the required context variable in brackets, as follows:

```
inst oblig policyName "{"
        ...
        when            context(contextVar) == expr ;   "}"
```

The *contextVar* is a variable that is supported by the context & correlation engine. We can compare the value returned against any expression using standard operators such as ==, < and > - the example above tests for equality. In addition, we can use Boolean logic operators such as AND

(&&) to tie sub-clauses together. A *when* clause can also combine sub-clauses based on the event, the target and context variables. For example:

when (e.eventInfo == “ABC”) && (t.deviceStatus == “IDLE”) && (context(currentLocation) == “HOME” || context(currentLocation == “COLLEGE”)

Whilst sub-clauses on the event are used to build a subscription that is sent to the events engine, and sub-clauses on the target are evaluated there before the action is performed, the behaviour for context sub-clauses is slightly different. When an event arrives at the PMA, it determines which context variables are required, and sends a *getContextData* event that is picked up by the CCE. This event contains the *contextVar*. The CCE returns the result in a *sendContextData* event. Whilst this approach does place a burden on the CCE to respond to a potentially large number of requests for context data, the advantage is that only the CCE obtains, processes and stores context data. Personal devices have no need to maintain communication with a multitude of context devices, or to process this data in any way.

The CCE effectively presents devices with a higher-level interface to contextual data. How the data is collected or processed is not of concern to the users of the data. Internally, the CCE distinguishes between context variables that are calculated internally, and those that require some external input, e.g. readings from a sensor. Those that are calculated internally should simply be implemented in code. To speed up the response time to devices, a separate process or thread should carry out the calculations and store the data at defined intervals. A response then simply involves reading the latest value for that variable.

Context variables that require one or more external events involve the CCE subscribing to those events with the events engine. A recalculation of the variable is then done each time an event arrives. Again, the data is stored such that responding to a device’s request simply involves reading the data. An example of context data that involves external events is geographical location. A General Positioning System (GPS) device may send events containing location co-ordinates, however this data is of little value to a device that wants to configure its behaviour dependent on whether someone is in a “home” or an “office” environment. The CCE provides the ability to convert low-level concepts into higher-level ones. In this instance, it would determine the mapping between numerical co-ordinates and abstract locations such as “home”, “office”, “cinema” etc that devices can make better use of.

The recalculation time interval is configurable via a configuration policy:

```
inst config CCE “{”
```

```
        recalcInterval          30000       -- in ms        "}"
```

4.6.2 Correlation (derived) events

In order to provide support for event correlation, a slightly modified type of policy will be used:

```
inst oblig policyName "{"
        on              derived(derivedEventPattern) ;
        subject         domainPath ;
        do              event(aggregateEvent) ;          "}"
```

The reserved keyword **derived** specifies that the expression that follows in brackets is not a single event, but a derived condition based on a number of events. There is clearly considerably scope for providing a wide range of types of rules, however there is a trade-off between flexibility of event correlation and performance. We therefore propose the following limited syntax:

eventA → eventB	true iff event B immediately follows event A
*x * eventA*	true iff x occurrences of eventA

Events are therefore chained using → operators to represent the order in which they occur. The clause *3 * eventA* is of course equivalent to eventA → eventA → eventA. Support for other operators, such as Boolean operators is left as a potential future extension.

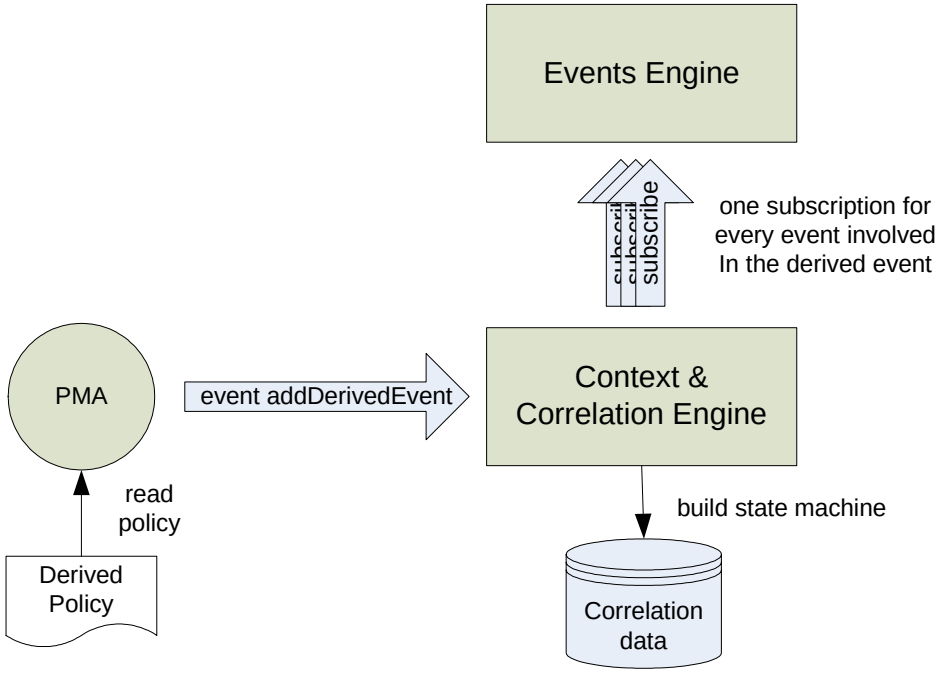

Figure 4.8: A derived event is not sent directly to the events engine – instead the context & correlation engine builds a state machine to represent the derived event and subscribes to the individual component events.

The subject *domainPath* specifies the domain in which the policy resides, e.g. /policies/derived. The do (action) specifies that a new event should be generated when the derivedEventPattern becomes true. This is just like any other event that might be generated by a device. We can then write policies that trigger when this event occurs.

Derived policies are not translated directly into subscriptions by the corresponding PMA; instead, an *addDerivedEvent* event is sent to the context & correlation engine (CCE) containing the *derivedEventPattern* as a string. When the CCE receives such messages, it parses the string and produces a list of individual events that are the dependencies of the derived event. It subscribes to these events in a similar way to which a PMA subscribes to a standard event. Therefore the CCE is acting as a consumer of these events. Figure 4.8 illustrates these interactions.

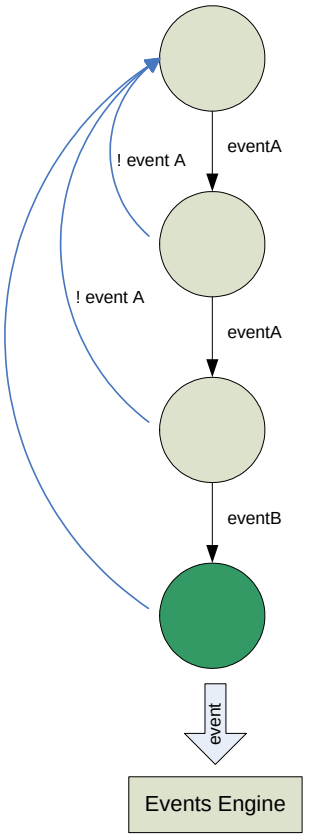

Figure 4.9: State Machine model

The CCE is responsible for generating the derived event when the pattern of events required occurs. It therefore builds a state machine for every derived policy. The state machine splits the *derivedEventPattern* string into transitions. For example, "2 * eventA → eventB" is represented as {eventA, eventA, eventB}. These transitions cause a change in state. Figure 4.9 shows an illustration of a state machine for this example.

Every time the CCE receives an event, it checks each state model that it is maintaining to see if there is a match in the next transition that is waiting to complete. If there is no match, we move back to the initial state. If there is a match, we move to the next state. If the final state is reached (pictured in green in our diagram) the derived event is generated and placed onto the event bus. We then go back to the start node.

4.7 Cell instantiation, standby and shutdown

Whilst we have looked at the various components that make up a self-managed cell in our architecture, together with the major interactions between them, we have not yet considered how a cell can be instantiated or shutdown. All management components and devices that are to exist in a single cell should have a local copy of the following common policy:

```
inst config common "{"
        eventsEngine                urlPath                "}"
```

The urlPath specifies where the events engine for the cell being instantiated is located. The standard syntax for Elvin event servers (routers) is "elvin://hostname:port" where the port is optional. If one is not specified, then a default will be used.

The events engine should be started first since this forms the communications backbone for the cell. The domain and context & collaboration engines should be started next. Following this, the discovery server should be started. We delay starting the discovery engine until we are sure that we can deal with the domain membership of new devices that we discover. Once all of these management components have been loaded, devices can be allowed to connect to the system, and will be discovered by the discovery server.

4.7.1 Standby mode

Since the devices in a cell are likely to remain relatively static over time and we would like to minimise the amount of time taken to start up the system, there is support in our architecture for a **'standby' mode**. When the cell goes into standby, the components write out all data that they hold on to disk. For example, the domain server will write the domain structure including the membership of each domain. The events engine keeps a record of subscriptions. When the cell is brought out of standby, the information is restored from disk and execution can continue as before. If a cell is fully shutdown, rather than placed properly into standby, then it is necessary to reinitialise the system from scratch, by re-reading all the policies, creating subscriptions for the events server and rediscovering the devices for addition into domains.

4.8 Summary

In this chapter we have presented a high-level design overview for our proposed middleware architecture. We have looked at the components that make up a self-managed cell, their functionality and the ways in which they interact in order to support the autonomic management of devices in a Personal Area Network. In addition, we have presented new types of policies that can represent this behaviour.

Our approach towards policy and domain management agents advocates the use of threads to increase the level of concurrency possible. In addition, each object that runs as a thread has direct capabilities to communicate with the asynchronous event bus. We suggest that this paradigm is likely to deliver higher performance than a more traditional

approach where a centralised communications agent handles communication for a large number of entities.

We have also proposed the definition of device profiles as standards. These can be updated over time, and define the functionality that devices adhering to them can provide. By enabling these profiles to be upgraded via firmware updates, users can be provided with enhanced functionality as and when it is agreed by the standards body.

In the following chapter, we translate our high-level design into a detailed architectural design, mapping into software components suitable for an implementation.

5 Detailed Architectural Design

This chapter forms the second part of our architectural design description. We build upon the work described in the previous chapter and present a detailed design that maps the high-level behaviour we have already looked at into a model suitable for implementation in a modern programming language. Whilst the majority of content presented here is language-independent, where it is necessary to explain the behaviour of algorithms and the definition of data structures in specific detail, we make reference to the Java 1.4 API. However mapping our model from Java into similar object-oriented languages such as C++ and C# is unlikely to prove a particularly arduous task, since the overall software structure remains unchanged.

5.1 An object-oriented architecture

The middleware architecture we propose is has been developed using the object-oriented paradigm. Under this approach, software quality is heavily influenced by how we assign responsibilities to objects that compose the system. Responsibilities include knowledge and behaviour. Knowledge refers to knowing about private encapsulated data, about related objects and about things that can be derived or calculated. Behaviour encompasses an object doing something itself, initiating action in other objects, as well as controlling and co-ordinating activities in other objects. By ensuring an appropriate separation of concerns, we can achieve our goals of high cohesion and low coupling between objects.

As part of good software engineering methodology, we advocate and utilise the Gang of Four (GoF) design patterns [DATA02] that suggest recurring solutions to problems that are encountered again and again in software systems.

The design presented in the rest of this chapter makes extensive use of UML (Unified Modelling Language), which is in widespread use in the design of object-oriented software. In particular, we make use of class diagrams to represent the relationships between classes and interfaces.

5.2 Events engine

As described in the high-level design, we make use of DSTC's Elvin event-based middleware to provide our asynchronous communications bus. Elvi is deployed in two components – a router (elvind) that is effectively the events engine and handles subscriptions etc, and a Software Development Kit (SDK) for the chosen flavour of programming language that provides the client libraries for communicating with the router. We use the Elvin Java API (je4) to describe how our components make use of

the features that Elvin provides, however similar APIs are available for C++, Python and the Microsoft COM (ActiveX) environment. Client libraries implement the Elvin client protocol, and provide a mapping from the native data types of the programming language to those used in Elvin notifications.

Our approach is to encapsulate and extend the supplied Elvin API as part of own 'client connection object' that provides a single, clearly defined object for use by all entities to communicate via the asynchronous bus. The sub-sections that follow detail our methodology.

5.2.1 Elvin Router

The Elvin Router supports several platforms at the time of writing, including Microsoft Windows 2000 and XP, Red Hat Linux, Sun Solaris, Mac OS X, FreeBSD and Irix. Microsoft Windows CE and Pocket PC are not supported at the moment, though the scope exists for these platforms to be supported in the near future. Despite this disadvantage, we selected Elvin due to it meeting our requirements in a wide range of other areas, such as performance.

Communication under Elvin takes place using TCP/IP sockets. However this is wrapped by a series of higher-level proprietary protocols that define the URL-based addressing scheme and the specific way in which events are transmitted, transported and delivered. A key benefit of using Elvin is that we can run it over any type of TCP/IP network that we choose. For example, we could choose between a WiFi 802.11b network or one based on the Bluetooth PAN profile – as long as we have a TCP/IP stack, no modification needs to be made to the Elvin installation.

The router can be either referred to either by its hostname or IP address, e.g. "elvin://elvinserver" or "elvin://192.168.2.1". The only requirement from a networking perspective is that the devices must be able to communicate on the TCP and UDP ports that Elvin requires, as defined in the Administrator's Guide [MANT04b].

5.2.2 Core communications functionality

Since all entities in our self-managed cell (including management components and devices) need to be able to send events to and receive events from the asynchronous bus, we have defined a core communications class *CoreComms* that implements common functionality such as integration with the Elvin client components. This eliminates the need for individual entities to have to duplicate these tasks. Figure 5.1 presents a class diagram defining the dependencies between *CoreComms* and the Elvin API. Note that we have only shown the attributes and methods that are relevant to our implementation.

The *CoreComms* abstract class should be extended by all management components and the device adapters. Instantiation takes place by specifying the URL of the Elvin router and a unique entity ID, e.g. "discoveryServer" as parameters. Sub-classes must implement the *eventAction* method that is declared abstract in *CoreComms*. This method defines the actions that should be performed when an incoming event arrives. Note that Elvin handles "notifications" and not "events". An *Event* is defined by us as an extension to Elvin's *Notification* class. This is simply a wrapper for neatness and allows us to define additional attributes such as the name of the event and the time it was generated. The *Event* class handles the translation into a *Notification* before the object is placed onto the bus.

Elvin's notification system is based on the Observer design pattern. The *addSubscription* method is called on the *Consumer* object, specifying an object that Elvin should call with the *notificationAction(event Notification)* method. *CoreComms* implements this interface and therefore any class extending it will have the method called by Elvin to deliver a notification. Since we would like to deal with events rather than notifications, the *notificationAction* method always carries out any common functionality and then calls the *eventAction* method to pass the sub-class an *Event* object.

Upon instantiation of an object that extends *CoreComms*, the connection to the Elvin router will be established, along with the creation of *Consumer* and *Producer* objects that are used for communication with the router. As shown in the class diagram, entities in our architecture should not need to communicate with these objects directly – all outward communication takes place using the *sendEvent* and *subscribeEvent* methods as defined in *CoreComms*.

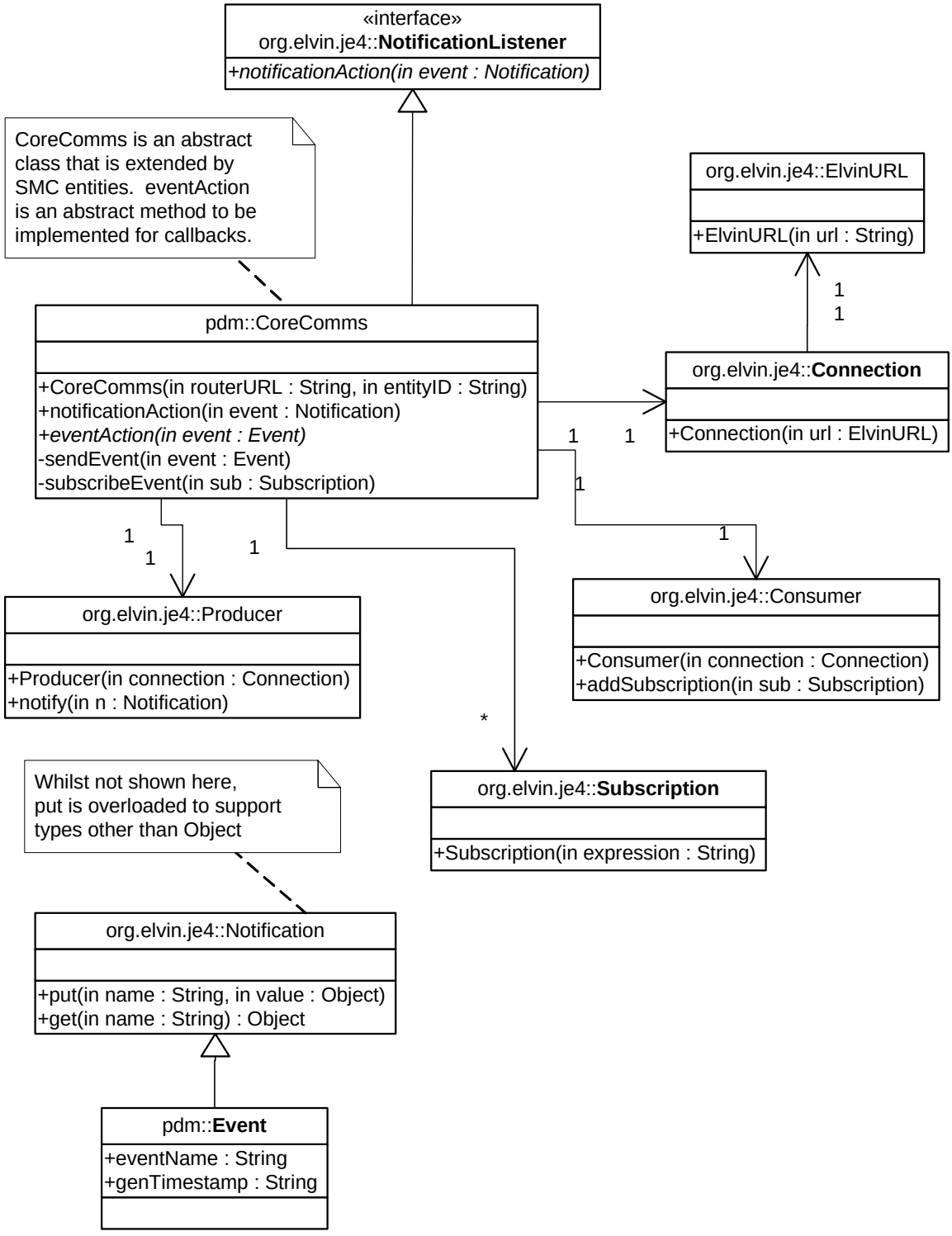

Figure 5.1: Definition of a core communications class to be extended by management components and devices in the self-managed cell.

Note that when a notification arrives, the *notificationAction* method will be called by a thread belonging to Elvin, allowing notifications (and thus events) to arrive and be processed in a concurrent manner. However as stated in the Elvin documentation, it is vital that the time spent executing code upon the callback should be minimised. Therefore our entities that define *eventAction* should make use of their own worker threads to carry out any complex processing of events, rather than doing the work in the Elvin notification thread.

5.3 Policies & Policy management agents

The proposed design for the policy sub-system is presented in Figure 5.2. *Policy* is an abstract class which is extended by two types of policies – *BasicPolicy* and *DerivedPolicy*. All policies have a subject (i.e. the path of the domain where the policy is stored). Basic policies are those which are based on the occurrence of a single event. These policies define the event on which they are triggered, the target domain, and provide a method that can be called to test conditions on an event as well as a method to test expressions based on context data.

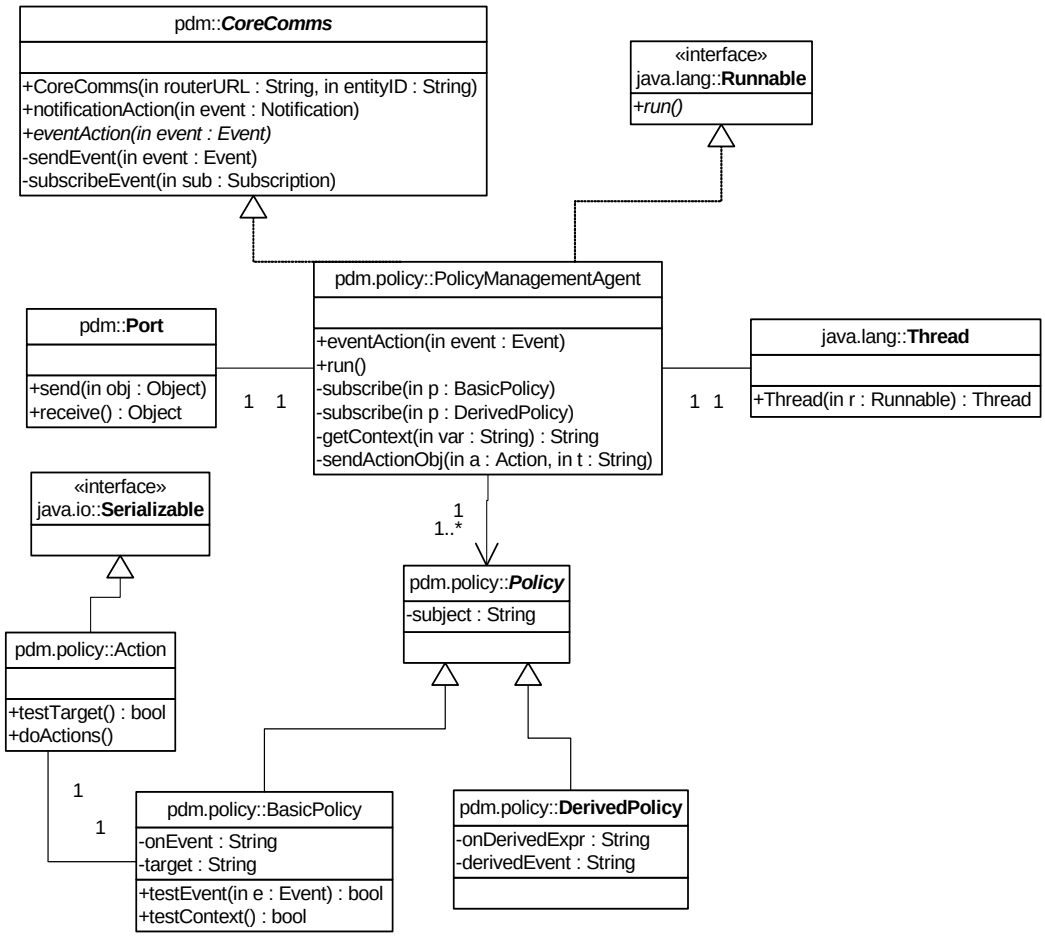

Figure 5.2: Policy and PMA structure and key relationships

When an event arrives at the PMA, it checks the *onEvent* of all basic policies that it is managing and if there is a match, then the *testEvent* and *testContext* methods are called to evaluate any conditions corresponding to the event itself or context data. The *testContext* method results in a call to the PMA's *getContext* method for each variable that is required in order to complete the test. The PMA requests this data via an event, and returns when the data is available. The use of a retry and timeout mechanism is essential to handle the case where the request and/or response become lost during transmission.

5.3.1 Encapsulation of actions

A basic policy has a 1:1 relationship with an *Action* object. This is based on the Command design pattern, where the operations to be carried out are encapsulated in an object and passed to the target which then calls the object's relevant methods. In our implementation, the PMA calls its private *sendActionObj* method which serializes the *Action* object passed in and delivers it to the target domain's DMA by placing an event on the bus with an eventName of "action". The recipient DMA will propagate the action object to its devices and any subordinate DMAs. When the action object reaches a device, the *testTarget* method is called which carries out conditional tests on the device. If the result of this call is true, then the *doActions* method is called to carry out the actions. The action object can then be discarded.

5.3.2 Thread & mailbox model

The PMA itself runs as a thread – note that we have demonstrated how this may be implemented in Java, however several other languages also support threads in a similar manner. Since the PMA responds to incoming events, we make use of a simple mailbox (port) mechanism to store events before they are processed by the PMA. An incoming event (via *EventAction*) causes the event to be "sent" to the port. The PMA thread's *run* object then performs a *receive* on the port and is blocked if there is nothing waiting to be processed. A sample Java implementation of our *Port* class is shown below:

```
package pdm;

import java.util.LinkedList;
import org.elvin.je4.Notification;

public class Port {

      private LinkedList messages;

      Port() {
            messages = new LinkedList();
```

```
        }

        protected synchronized void send(Notification n) {
                // This method is called by the Elvin router
                // via the policy
                messages.addLast(n);
                notify();
        }

        protected synchronized Notification receive()
                throws InterruptedException {
                // Policy thread is blocked if there are no
                // notifications available to collect
                while (messages.size() == 0)
                        wait();
                return (Notification) messages.removeFirst();
        }
}
```

Note from the above that as part of Java's support for building applications that use concurrency, we can use the *synchronized* keyword to ensure exclusive access to a method.

5.3.3 Basic & derived policies

When the PMA is instantiated, it creates subscriptions for each of the policies that it is managing. The *subscribe* method is polymorphic, so that the behaviour can be different for basic and derived policies. Subscribing a basic policy is quite straightforward – the policy's *onEvent* attribute is used. For a derived policy, the *onDerivedExpr* and *derivedEvent* attributes are passed via an event to the context & correlation engine. It is the responsibility of the CCE to parse the derived events expression and to set up subscriptions for the corresponding events that are involved in the derived event. The CCE is informed about these events and generates *derivedEvent* only if the individual events occur in the pattern stated in the derived events expression.

5.3.4 Policy compilation

Note that as stated in the project scope, we have not proposed a mechanism for compilation of policies from the high-level Ponder-like syntax into objects that can be instantiated. Therefore for the purposes of our implementation, the policies are translated "by hand" into the equivalent object representations as shown in Figure 5.2.

5.4 Device adapters

Device adapters provide instrumentation with physical hardware devices. As described in the previous chapter, we make use of a 'quenching' mechanism to ensure that devices only transmit events if there are

consumers that require them. Since devices may have the capability to send out a wide range of events at regular intervals, it would be highly inefficient for all such devices to send every type of possible event to the event bus. However implementing quenching does require that we store additional data in order to keep a track of the number of consumers that are listening to each event that the device has the capability to generate.

The class diagram for device adapters is presented in Figure 5.3. Our *DeviceAdapter* is defined to be an abstract class so that users of the middleware can define their own adapters as necessary by extending it and defining the *processActionObject* method that is called when an incoming action arrives from a policy management agent. Upon instantiation of a device adapter, in addition to the Elvin Router URL, we must pass in a profile string and an entity ID. The profile string is simply a semi-colon delimited list of profiles that the device supports, e.g. "phone;pda". These profiles are sent to the domain server when the device is discovered, so that it is can be added in to the correct domain(s).

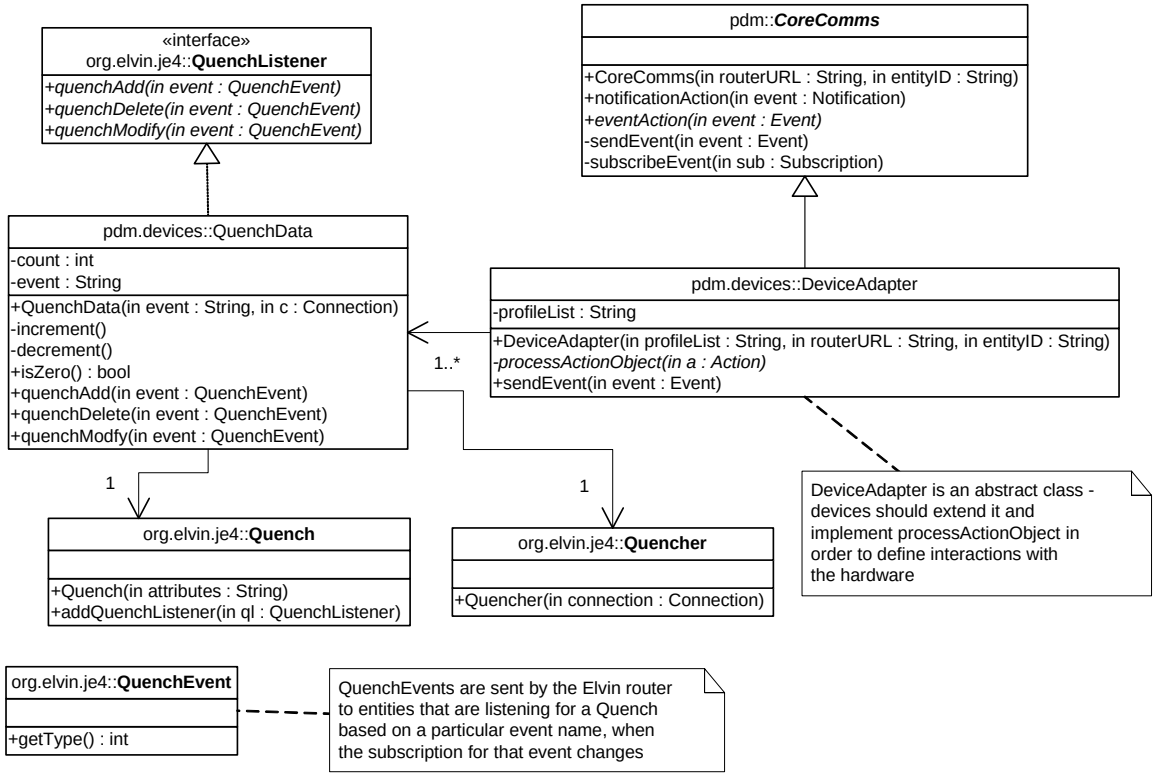

Figure 5.3: Class diagram defining the abstract DeviceAdapter and event quenching mechanism

All *DeviceAdapter* sub-classes must listen out for a common set of events and therefore the constructor of *DeviceAdapter* sets up these subscriptions. In particular, we need devices to listen for "action" events that are directed to them and to the "whoIsAlive" ping messages that are

sent by the discovery server. Appropriate *Subscription* objects should therefore be created and passed to the *subscribeEvent* method. The subscription format for action events is detailed in the high-level design for the action transport mechanism (see Section 4.4).

5.4.1 Event quenching

In addition to subscribing to standard events, it is necessary for the device adapter to instantiate a *QuenchData* object for every event that the device can generate. This class is defined as part of our architecture rather than as part of the Elvin libraries, and provides support for handling changes in subscriptions. The Observer design pattern is used here to register a *QuenchData* object by calling the *addQuenchListener* method on the *Quench* object (there is a 1:1 correspondence between *QuenchData* and Elvin's *Quench* API class). *QuenchData* realizes the *QuenchListener* interface. We also make use of the *Quencher* class from the Elvin API which is equivalent to the *Consumer* and *Producer* classes for standard subscriptions. To instantiate a Quench subscription, we simply provide a string containing the name of the event that the device would like to know consumer information for[1]. As part of the quenching mechanism, Elvin will send the device adapter a *QuenchEvent*, each time the number of consumers subscribed to an event changes. The *getType* method can be called on this object to work out whether it is an addition, deletion or modification of subscription.

Devices should place events on the bus by calling the *sendEvent* method. This method overrides the method with the same time defined in *CoreComms*, since devices require slightly different behaviour to support quenching. Our overridden method calls the *isZero* method on the *QuenchData* object corresponding to the event that is about to be sent on to the bus. If this returns true, then no consumers are listening and the event is discarded. Otherwise the event is transmitted as usual.

5.5 Discovery server

We propose a simple structure for the discovery sub-system, as shown in Figure 5.4. The *DiscoveryServer* component extends our common functionality class *CoreComms* to provide access to the event bus. As part of the instantiation of the server, we take in the configuration variables as specified in the configuration policy. Note that we do not provide a compiler to map from the configuration policy syntax into objects as part of this implementation; this is left as scope for potential future work.

[1] Currently it is only possible to quench based on an attribute name – the condition cannot be based on attribute values.

The behaviour of the Discovery Server is as described in the pseudo code contained in Section 4.5. A *whoIsAlive* call is made at regular intervals to check if the devices that we know about still exist, or if new ones are visible in the cell. We maintain a *DeviceData* object for every device that is discovered. This contains information that is communicated to the domain server via the *genNewDevice* and *genLostDevice* methods for new and lost devices respectively. These methods generate and place the appropriate event on the bus, which is picked up by the domain server.

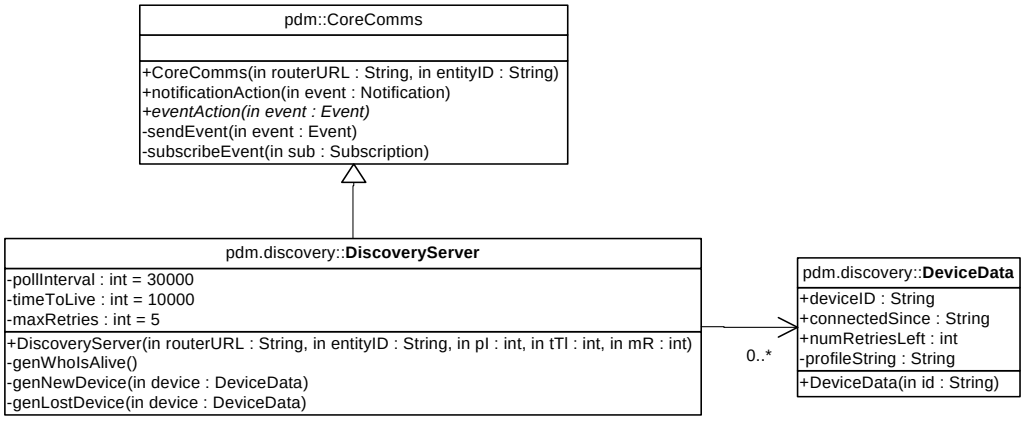

Figure 5.4: Discovery server key components

5.6 Domain server

The domain sub-system is provided for the purposes of being able to group policies and devices and to be able to apply actions to an entire group of objects, together with any sub-groups. In Section 4.4 we presented the structure of our domain system, noting that devices can belong to one or more domains. We now take this high-level design and propose a software design for this sub-system as part of our middleware architecture. As part of the design process, we considered whether or not it should be possible to distribute the domains across several devices. With a distributed solution, the domains could communicate via the asynchronous events bus to propagate actions to sub-domains, however this is likely to impose a significant overhead with large domain hierarchies. Since our intention is to minimise the amount of network traffic between components in our self-managed cell, we concluded that a centralised approach to domains is more suitable for this application. However we have proceeded to design this sub-system in such a way that should distributing domains be perceived to be advantageous in the future, this can be implemented with minimal changes to the code.

A class diagram for the domain sub-system is shown in Figure 5.5. *Domain* objects can have one or more children, as shown by the

relationship between this class and itself. We use a composition operator to indicate that sub-domains can only exist if the parent exists. All domains have a relative name *domainName* and the recursive *getAbsDomainName* method call can be made to obtain the absolute path by recursing up the hierarchy of domains. The *DomainDevice* class is used to maintain references to devices that belong within a domain. A *DomainDevice* object is created using the unique device string (e.g. MAC address) and a *Domain* will hold zero or more of these objects.

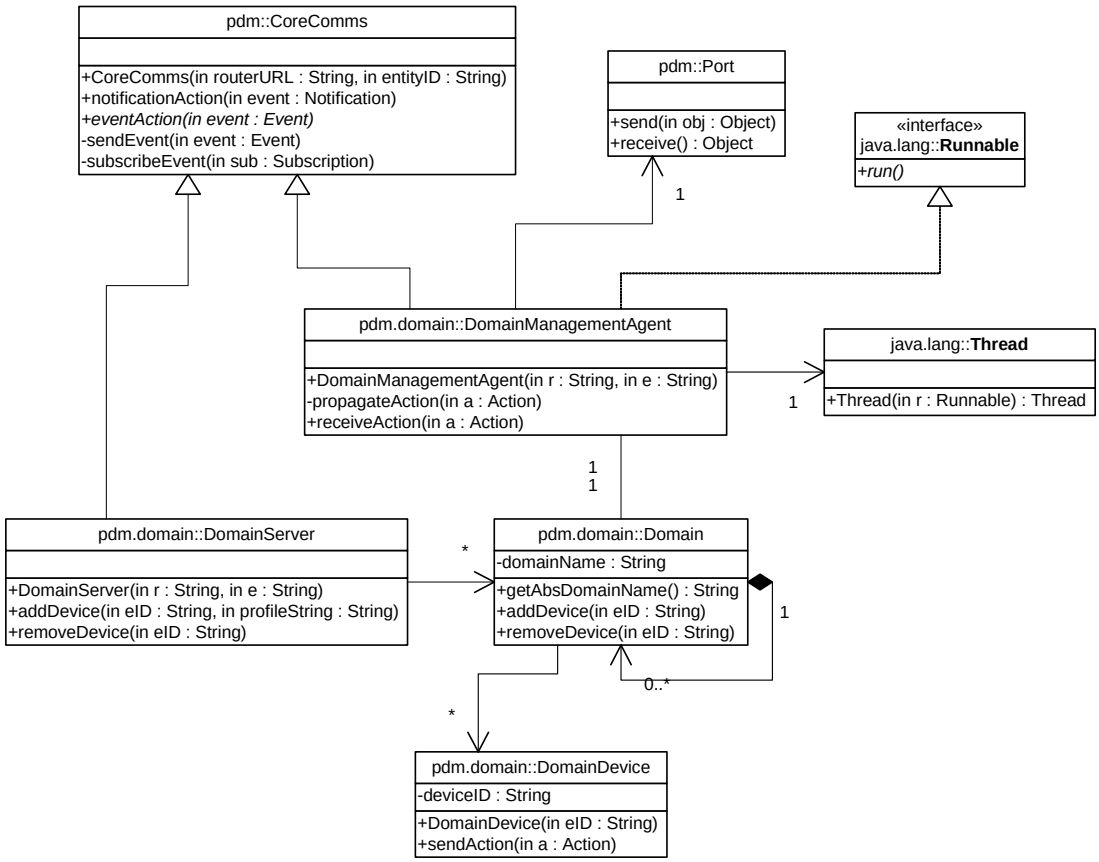

Figure 5.5: The domain sub-system

5.6.1 Mapping from device profiles to domains

The *DomainServer* object is responsible for dealing with the *newDevice* and *lostDevice* events from the discovery server, and registers subscriptions for these with the events server upon instantiation. The occurrence of these events causes the *addDevice* and *removeDevice* methods to be called respectively. When adding a device to the domain system, we need to map a profile string as received from the device via the discovery server into one or more domains that the device will be

placed into. Whilst there are a number of ways in which this might be carried out, our proposed method for doing this is as follows:

```
split the semi-colon delimited profile string into a set
of profiles

scan through our ruleset and narrow down to r = rules
that contain at least one of the profiles

for each rule in r {
      if rule evaluates to true against profile set {
            add device to rule.domain
      }
}
```

As an example, with the ruleset:

phone → /devices/phone
pda → /devices/pda
music → /devices/musicplayers
headset → /device/headset
phone && pda → /devices/phone/smartphone

Our example device with profiles {phone, pda, music} would join the /devices/phone, /devices/pda, /devices/musicplayers and /devices/phone/smartphone domains. The fourth rule will evaluate to false, against our profile set since our device doesn't support the headset profile. Therefore, as part of our approach, if a rule "conflict" occurs, the device simply joins both of the domains and there is no requirement to carry out any prioritisation of rules or to resolve any conflicts. We scan through all rules until we reach the bottom of the ruleset.

An alternative approach might be to follow the mechanism typically used to define rules for a firewall in a network. The rules are evaluated in order until one evaluates to true. We then stop at this stage and no further rules below this one as considered. We may also wish to use a system based on scoring, where we evaluate a profile set against a set of rules, and assign a numerical value against each match. We then add the device to the domain with the highest score. As part of this, we may also need to define a "mandatory" flag, to allow for certain rules to always be used, regardless of the score. The problems for using these types of complex approaches however involve an increase in the amount of processing power required, as well as the increased scope for causing undesirable behaviour by misconfiguration of the ruleset.

5.6.2 Adding devices to domains

The domain server adds a device to a domain by simply calling the *addDevice* method on the relevant *Domain* object, passing in the device

ID. This causes a *DomainDevice* object to be instantiated and attached to the relevant domain. A *Domain* has a 1:1 relationship with a corresponding *DomainManagementAgent*. The two could have been implemented as one entity, however we have separated them in order to encapsulate data related to a domain inside a *Domain* object and functionality related to managing a domain inside the agent. Our domain management agents run as threads and use a mailbox to queue events. The reason for using a mailbox (*Port*) and our own worker thread to carry out work is that we must avoid blocking the incoming Elvin thread for anything more than very simple operations. When an action arrives we need to generate and transmit events to the devices in the domain, however this work is much too intensive to be carried out within the Elvin thread, and could cause Elvin to run out of threads in its internal thread pool.

5.6.3 Handling incoming action objects

The *DomainManagementAgent* subscribes to the "action" event with the events engine such that actions to be performed on events in a domain and propagated to sub-domains can be received from policy management agents. When an action is received, the *receiveAction* method is called and this results in the *sendAction* method being called on each *DomainDevice* object that corresponds to a device in the domain. Through this mechanism, the action is serialized and sent as an event to each of the devices. The action is processed as detailed earlier in this chapter. However, in addition to carrying out the action on devices in that domain, it is also necessary to propagate the action to the DMAs of all sub-domains, if they exist. In our implementation, this is achieved by making a call to the private *propagateAction* method of the DMA that calls the *send* method on the mailbox (*Port*) of the relevant sub-domain. The action is therefore placed in the queue for the sub-domain and processed by its DMA, just as if the event had arrived directly via the asynchronous bus. The advantage of this mechanism is that we provide the ability to move from a centralised to a distributed domain structure with relative ease. In a distributed system, the DMAs would propagate the action objects via the event bus rather than via local method calls.

We note that as part of our design, actions are sent directly to the target domain's DMA, rather than to the domain server and each DMA runs as a thread. The rationale behind this design is firstly that running each domain as a thread provides us with added concurrency. Secondly, and more importantly, sending actions directly to a DMA is more efficient than a domain server having to propagate the action through the domain hierarchy. We would like to minimise network traffic and therefore propagation is only required for sub-domains. Parent domains are not affected in any way. In addition, by allowing each DMA to receive events

directly from the bus, we make it more straightforward to move to a distributed domain model if this is perceived to be of benefit. DMAs could be located on different hardware devices and could still form a hierarchical domain structure.

5.7 Context & collaboration engine

The context and collaboration functional areas were grouped into one sub-system since both of these features enable richer policies and behaviour based on events other than those generated directly by devices. The context & collaboration engine (CCE) takes in one configuration parameter, which defines the interval at which context data should be recalculated. Again, we extend the *CoreComms* common class that provides us with the ability to send events to and receive events from the asynchronous bus.

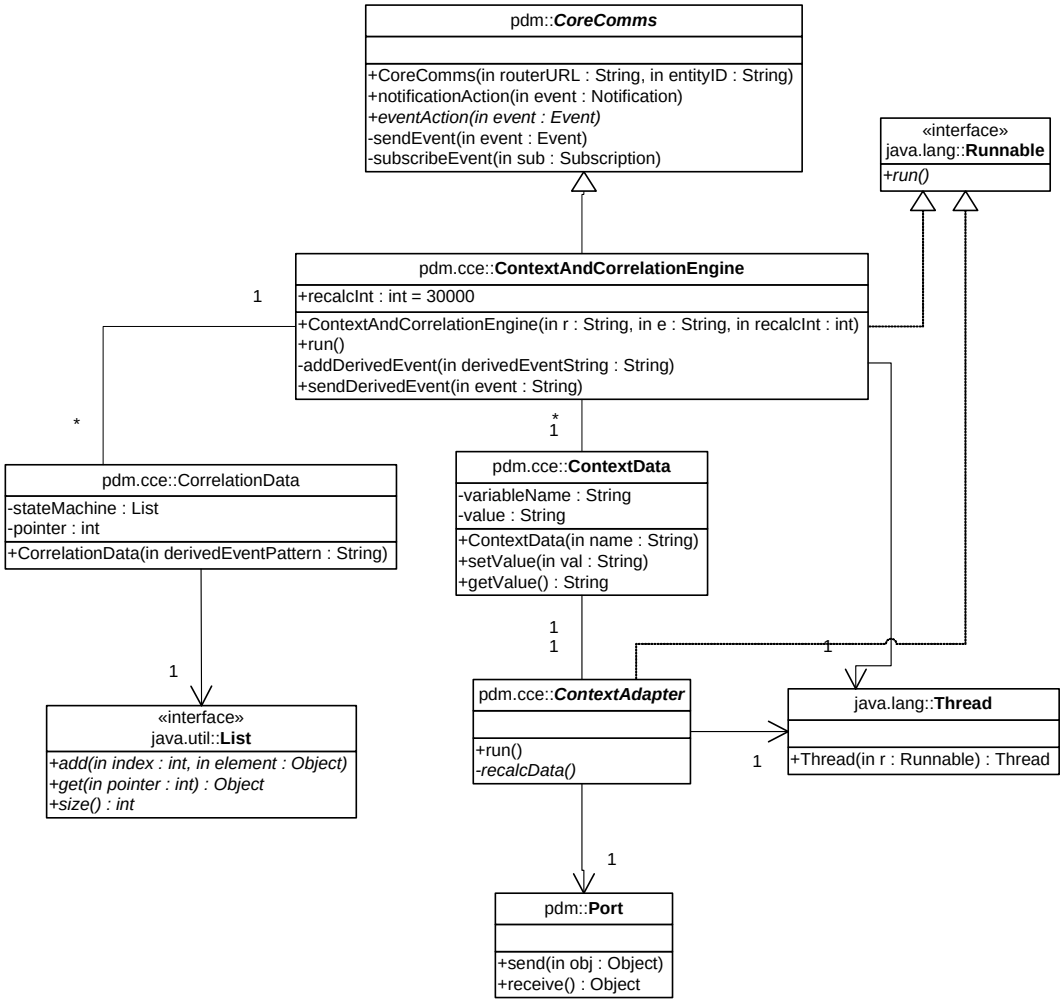

Figure 5.6: Context & Correlation Engine

5.7.1 Contextual data

Firstly, let us look at the components that calculate and provide contextual data. The CCE instantiates a *ContextData* object for every context variable that the CCE can provide to policy management agents. This simple object holds a value against a variable, where both are treated as strings. As shown in Figure 5.6, each context variable has a *ContextAdapter* object which runs as a thread and consists of a mailbox for the storage of events. In a Java implementation, the *run* method is called when the thread is started. Within the method we simply call *receive* method on the *Port* and repeat this infinitely. When an inbound request for context data comes in, we will pick it up from the mailbox in this way. In addition, the CCE itself is a thread and in its *run* method it sleeps for recalcInt, and then calls *recalcData* for each of the context variables it is managing. This process repeats perpetually. Note that the *ContextAdapter* class and *recalcData* method are abstract – adapter implementations should extend this class and define the method to determine how a variable should be calculated.

The CCE should subscribe *getContextData* event which allows it to receive requests from PMAs. In addition, subscriptions will exist for any external events that are needed. In our implementation, we assume that external events are those that are generated by context devices that effectively sit outside of a SMC. They are therefore broadcast to all cells within range. Each time an external event is received that is of interest, we recalculate the relevant variables and update the *ContextData* objects. Internal variables (i.e. those that do not require at least one external data value) are recalculated at regular intervals instead.

An incoming request for context data will be placed in the mailbox via the *send* method on the *Port* object. The request will be serviced by the *ContextAdapter* thread which will call *getValue* on the *ContextData* to obtain the stored value. The result will be returned by placing a *sendContextData* event on the event bus via the main CCE component.

5.7.2 Event correlation

The mechanism for event correlation is based on a state machine approach and is described at a high-level in Section 4.6.2. The CCE should listen for the *addDerivedEvent* event as placed by PMAs on to the event bus. This event includes a “derivedEventPattern” string that specifies the order in which individual events should occur for the derived event to trigger. When the *addDerivedEvent* is picked up by the CCE, it will call its own *addDerivedEvent* method that will instantiate a new *CorrelationData* object with the derived event string as a parameter. The string enables the *CorrelationData* object to build a state machine. Our

approach here is to use an indexed list, where the *pointer* variable starts at zero. The string is parsed and broken into a set of events. These are added in the same order to the list.

The individual events require subscriptions and these are created via the CCE's *subscribeEvent* method. When these events occur, the CCE uses the *get* method on the list to find out the event that each *CorrelationData* object expects to see next. This is compared against the event that has occurred. If there is a match, the pointer is incremented. If the pointer now equals the value returned from a call to the *size* method, then we have reached the end of the state machine, and the derived event should be generated using the CCE's *sendDerivedEvent* method. If no match was found, then the pointer is set back to zero, i.e. we reset the state machine. Note that we are assuming here that the network is reliable and that events arrive in the correct sequence. However due to the inherent "one-shot" communications paradigm used by Elvin, it is possible that events may be delayed and arrive out of sequence. To deal with this situation, we may use timestamps and hold a small buffer (i.e. a sliding window), reordering any out-of-order packets in a similar way to how the Transmission Control Protocol (TCP) provides a reliable transport protocol for networks such as the Internet. The disadvantage of this approach is of course increased overhead and computational complexity.

5.8 General architectural issues

In our discussion above with regards to the various sub-systems that comprise our middleware architecture, we have not looked into system quiescence with data persistence. This is a general design feature that should be implemented in each of the management components. Our suggested approach is to add functionality in the *DiscoveryServer*, *DomainServer* and *ContextAndCorrelationEngine* classes to write out all state data as an XML document. When instantiating each of these components, a check should then be carried out to determine if saved state data exists. We only want to use saved state data if it was written in the recent past, so the timestamp on this file should also be checked. If the XML document is valid, the data is then restored and the sub-system is effectively taken from a quiescent state to an active one.

In terms of carrying out a cold bootstrap on the system, the procedure described in Section 4.7 should be followed. It is imperative that the asynchronous event bus is started first, since this forms the backbone for the whole system.

5.8.1 Package structure

For reference, the package hierarchy that we have used is as follows:

org.elvin.je4	Elvin client libraries
pdm	Core classes and interfaces, events engine
- **pdm.policies**	Policies & Policy management agents
- **pdm.domain**	Domain server
- **pdm.discovery**	Discovery server
- **pdm.cce**	Context & Correlation Engine
- **pdm.devices**	Device adapters

Note that pdm is an abbreviation for “policy-driven middleware”.

5.9 Summary

In this chapter we have presented a detailed architectural design, building on the high-level design described in the previous chapter. We made use of UML class diagrams to describe the structure of each of the sub-systems and provided a description of the key object interactions. In certain instances we made use of parts of the Java API to demonstrate how an implementation might be achieved if Java is selected as the preferred language. However, most of these concepts could be easily mapped into object-oriented languages such as C++ and C#.

6 Case study

In this chapter we present a scenario that may typically occur in a personal area networking environment. We then describe how our middleware solution may be deployed to deliver this functionality, specifying how device profiles, domain structure and policies should be configured. Our intention here is to demonstrate the capabilities of our solution as a tutorial, such that other application developers can make use of the technology by following a similar approach.

6.1 Scenario overview

We'll present a scenario using one of our fictional characters, Bob. Bob's personal area network consists of a standard mobile phone (std_phone), a PDA with built in phone features (pda_phone), a MP3 music player (mp3_player) and a digital watch (watch). Before running these devices as part of a self-managed cell, Bob typically used them as independent devices, however this led to a number of problems:

- Bob used his PDA to store appointments and set alarms to give him advance notice of them, however since his PDA was often kept inside his bag, he frequently missed alarms and was subsequently late.
- Bob's PDA also functions as a mobile phone and he used it to make and receive calls in addition to his standard mobile phone. However since he regularly travels around the country, he found that his PDA often ran out of batteries and therefore he ended up missing important phone calls.
- As an avid movie lover, Bob regularly visits the cinema. However he almost always forgets to switch his phone off, or on to a silent mode, and this often caused him a great deal of embarrassment.
- Whilst listening to his MP3 player, Bob often missed several important phone calls since he was unable to hear his phone ring.

One of the key points we observe from the above is that Bob is having to adapt around the way in which the technology operates, rather than the other way around. The types of problems discussed above are clearly likely to discourage users from making greater use of advances in technology.

Under our self-managed cell middleware architecture, many of the problems that Bob faced can be eliminated, making the technology more useful to him. Here are the specific improvements that we will look at:

- When Bob's PDA wants to alert him about an appointment, in addition to sounding its own internal alarm, it asks his digital watch to sound an alarm as well. Whilst Bob's watch doesn't have the capability to display alphanumeric characters, the type of alert indicates to him that he should pick up his PDA for further details.
- If Bob's PDA has less than 10% battery life remaining, it automatically sets up a call diversion service to forward his calls from the PDA to his standard mobile phone. This is done without Bob's intervention and means that Bob no longer misses important calls. The value of 10% is a default, and Bob can specify his own value if he wishes to do so.
- When Bob comes with proximity of a cinema, his phone and PDA both automatically switch to a silent vibrating mode. Bob no longer needs to worry about turning his phone off. When he leaves the cinema, both his phone and PDA switch back to an audible alert.
- When Bob is listening to music on his MP3 player and a call comes in on either mobile device, the music pauses and he hears a message announcing the phone number of the person calling him. Once the call his over, his music player automatically resumes playback.
- If Bob receives two consecutive missed calls on his phone, his watch bleeps to indicate that someone may be trying to reach him urgently. Bob finds this useful if he accidentally left his phone in a different room and therefore couldn't hear it ringing.

Now that we have defined the type of behaviour that we would like to provide, we will describe how our middleware can be deployed to provide this functionality.

6.2 Defining device profiles

Device profiles should typically be defined as **standards** and implemented by manufacturers in a similar way to Bluetooth profiles. For the purposes of this case study, we define some suggested profiles.

For each profile, we define events and/or actions as appropriate. Events are those generated by devices implementing that profile. The defined actions are the ones that devices supporting that profile can accept. Note that our profiles only define events and actions relevant to our scenario. We also show local conditions where necessary – these are conditions that are tested at a device before an action is carried out.

6.2.1 Profile: common

Events:
batteryLow(percentRemaining)

6.2.2 Profile: pda

Events:
appointmentAlarm(apptText)

6.2.3 Profile: phone

Events:
incomingCall(callerID)
endedPhoneCall()
missedCall(callerID)

Actions:
switchToSilentMode()
switchToAudibleRing()
divertCalls(number)

6.2.4 Profile: simplealertdevice

Actions:
soundAlarm(alarmPattern)

6.2.5 Profile: audioplayer

Actions:
pauseAudio()
playAudio()
playCallerID(callerID)

Local conditions:
isAudioPlaying
pausedForIncomingCall

6.3 A domain hierarchy

6.3.1 Profile to Domain Ruleset

We use the following ruleset in our domain server to map profiles to domains:

common → /devices
phone → /devices/phone
pda → /devices/pda
audioplayer → /devices/musicplayers

phone && pda → /devices/phone/smartphone
simplealertdevice → /devices/alarm

6.3.2 Domain structure

Application of the ruleset above results in the following domain membership:

/devices/phone	{std_phone, pda_phone)
/devices/pda	{pda_phone}
/devices/musicplayers	{mp3}
/devices/phone/smartphone	{pda_phone}
/devices/alarm	{watch}

The common profile applies to all devices, and is linked to the domain /devices, so that there is propagation to all devices. For further details on how our ruleset is applied to determine the domain membership, please see Section 5.6.

6.4 Setting up SMC policies

SMC policies are configuration policies that allow for developers to configure the behaviour of the management components with a cell. The first policy is a common policy that defines the URL to the events engine. This policy must exist on all management components and devices. In this example, we assume that the Elvin Router is running on IP address 192.168.1.1:

```
inst config common {
        eventsEngine                “elvin://192.168.1.1”        }
```

Next, we configure the discovery server. In this example, we would like the discovery server to poll for devices every 10 seconds, waiting 5 seconds for each device to reply, and making a maximum of 3 retries before considering a device to be “lost”:

```
inst config discoveryServer {
        pollInterval                10000        -- in milliseconds
        timeToLive                  5000         -- in milliseconds
        maxRetries                  3                    }
```

The context & correlation engine is also configured via a policy. Here we define that we would like context data to be recalculated every 60 seconds:

```
inst config CCE {
        recalcInterval              60000        -- in ms        }
```

6.5 Composing system and user policies

Next, we will define the system and user policies for our system.

6.5.1 System policies

The first policy specifies that all devices defined as being alarm devices should sound with a specific alarm pattern (we assume pattern_2 is one of the types) when the appointmentAlarm event occurs. Note that we don't make use of the apptText, since these devices cannot display the text:

```
inst oblig appointmentAlarm {
        on              e = appointmentAlarm(apptText);
        subject         /policies/system ;
        target          t = /devices/alarm ;
        do              soundAlarm(pattern_2) ;          }
```

The call diversion functionality is implemented as a user policy, rather than as a system policy and we consider it later. This is because it is only Bob who can specify where he would like his calls diverted to.

Next, we implement a context based policy to switch Bob's phone and PDA devices on to silent mode when he enters a cinema:

```
inst oblig silentMode {
        on              e = epoch60Seconds();
        subject         /policies/system ;
        target          t = /devices/phone ;
        do              switchToSilentMode() ;
        when            context(location) == "CINEMA"     }
```

Note that we assume the event epoch60Seconds()[2] is an internal system event that triggers automatically every 60 seconds and allows us to execute policy at regular intervals. This event is provided directly by the events engine. Therefore every 60 seconds, we check if the location has changed to "CINEMA" – if it has, then the action is carried out. We assume that the context variable "location" is provided by the context & correlation engine.

Similarly, we can define a policy to switch back to an audible ring when appropriate:

```
inst oblig audibleRing {
        on              e = epoch60Seconds();
        subject         /policies/system ;
        target          t = /devices/phone ;
```

[2] This concept was not proposed in our design and is an extension to the architecture.

```
        do              switchToAudibleRing() ;
        when            context(location) != "CINEMA"      }
```

The following policy enables Bob to be informed about incoming phone calls via his MP3 player, if it is playing audio:

```
inst oblig incomingPhoneCall {
        on              e = incomingCall(callerID);
        subject         /policies/system ;
        target          t = /devices/musicplayers ;
        do              pauseAudio() → pausedForIncomingCall
                            = true → playCallerID(callerID) ;
        when            t.isAudioPlaying                    }
```

Note that in the above policy the when condition is based on the target and would therefore be encapsulated without our action data object and evaluated when it reaches each device in the target domain. When the call ends:

```
inst oblig endedPhoneCall {
        on              e = endedPhoneCall();
        subject         /policies/system ;
        target          t = /devices/musicplayers ;
        do              playAudio() → pausedForIncomingCall
                            = false ;
        when            t.pausedForIncomingCall             }
```

The final set of policies implements the missed calls functionality – if 2 missed call events are received consecutively then alarm devices should bleep with pattern_3. Firstly, we define the correlation policy which is sent to the context & correlation engine:

```
inst oblig correlationMissedCalls {
        on              derived("2 * missedCall(callerID)" ) ;
        subject         /policies/system/derived ;
        do              event(twoMissedCalls) ;             }
```

Next, we make use of the *twoMissedCalls* aggregate event in the usual way:

```
inst oblig bleepOnMissedCalls {
        on              e = twoMissedCalls();
        subject         /policies/system ;
        target          t = /devices/alarm ;
        do              soundAlarm(pattern_3) ;             }
```

Our simple context & correlation engine only supports aggregate event generation based on the occurrence of events of a particular name. A more advanced model might be able to allow for the specification of conditions based on the individual events. For example, in this case we

might want to only raise the *twoMissedCalls* event if the *callerID* parameter in both events was the same.

6.5.2 User policies

The policy to enable call diversion when the battery is low is implemented as a user policy – these policies are typically configured by the user or administrator of the cell rather than being embedded in firmware. We use the following policy:

```
inst oblig callDivertOnBatteryLow {
        on              e = batteryLow(percentRemaining);
        subject         /policies/user ;
        target          t = /devices/phone/pda_phone ;
        do              divertCalls("07923919288") ;
        when            e.percentRemaining < 10          }
```

In the above policy, the number to divert to should be configured by the user. Note that our target here is a **single** target, not a domain. We refer specifically to /devices/phone/pda_phone which is Bob's pda_phone device. We could equally have referred to it as /devices/pda/pda_phone since it exists in that domain too.

In order to support this policy, the following authorisation policy needs to exist. This allows for this policy to be added to the system by the user:

```
inst auth+ allowCallDiversionPolicies {
        subject         /devices/phone ;
        target          /policies/user ;
        action          installPolicy(p) ;
        when            p.do == "divertCalls*"           }
```

Note that we do not provide an implementation for authorisation policies as part of this work. However the above policy specifies that only devices in /devices/phone should be able to install a policy in the /policies/user domain, and only if the do action begins with divertCalls. We have assumed that a suitable authorisation server will be able to handle wildcards in this manner.

6.6 Summary

In this chapter we have presented a hypothetical scenario involving a set of devices in a personal area network. We discussed the kinds of problems that a user might face without a middleware platform, and then presented how the situation might change with this extra functionality. We then provided a walkthrough of how our solution could be configured to deliver this functionality, including specification of relevant SMC, system and user policies.

7 Building a prototype

The purpose of this chapter is to present the work that we have carried out as part of the third deliverable of this project, which involves the development of a prototype system based on a subset of the middleware architecture that we have proposed. The intention is to use actual hardware and software components to provide a 'proof of concept' demonstration for several of the key concepts that we have discussed earlier in this report and to analyse the results of this exercise which will feed into our suggestions for potential future directions in this research area.

7.1 Prototype objectives

Our primary objective is to bring together elements from our background study (see Chapter 2) with the middleware architecture and case study. The purpose of the background study was not only to look at developments in the research areas related to this project, but also to investigate the 'state of the art' with regards to hardware and software platforms for mobile devices. As part of this work, we looked at wireless technologies such as Bluetooth and WiFi, and discussed development platforms such as Microsoft's .NET Compact Framework, Java 2 Micro Edition (J2ME) and Symbian OS. These technological developments are of importance, since they effectively define the scope of what is feasible to implement today.

We wish to demonstrate the behaviour of an asynchronous event bus, including event quenching to improve the efficiency of the system. Devices should be able to join a cell and have actions performed on them via the execution of policies. Whilst we are unable to provide a full implementation for the middleware due to time constraints, the prototype is designed to cover the most important areas of the system.

7.2 Choice of implementation environment and tools

In this section we briefly discuss the hardware and software platforms that we are selected for this exercise and a rationale for these decisions where appropriate.

7.2.1 Programming language: Java

We have selected Java as a programming language primarily due to its flexibility and cross-platform capabilities. An increasing number of devices provide some support for Java, either in the form of a virtual machine for Java 2 Micro Edition (J2ME) or in some cases, Java 2 Standard Edition (J2SE). The advantage of Java is that code is compiled into Java bytecode which can be executed on any Java Virtual Machine

(JVM). This allows us to write code that can run on a range of different underlying hardware platforms (e.g. Motorola Dragonball, ARM, Intel) without modification.

We also considered Microsoft's C# language which provides similar features to Java, however this restricts us to running software on devices that support the Compact Framework. Currently, this includes a subset of Microsoft Smartphones and Pocket PC devices. Compact Framework is not available for competitive platforms such as Symbian OS. The advantages of using Compact Framework however are that it provides the developer with a relatively rich API with which to interact with the hardware. Achieving the same behaviour in Java may prove to be a more arduous task.

To achieve maximum possible interactivity with the hardware, the Symbian OS SDK is a good choice, however we discovered that this only supports C++ and has a relatively steep learning curve. In addition, Java provides features such as automatic garbage collection which generally reduce development time.

7.2.2 Development IDE: Eclipse

We have chosen the Eclipse Integrated Development Environment (IDE) which was originally developed by IBM and has recently transferred to the responsibility of an independent organisation known as the Eclipse Foundation [ECLIWWW]. The advantages of Eclipse are that it is open source and operates in a multi-language, multi-platform, multi-vendor environment. For example, we can choose to develop in the same way on a Microsoft Windows machine as we can on one that is running a distribution of the Linux operating system.

An important design feature of the Eclipse software is the ability to 'refactor' code with ease. For example, we can move a class from one package to another, and Eclipse will automatically make the necessary modifications to the code, adding package and import statements where appropriate. This feature is also extremely useful when it is necessary to rename classes. Again, Eclipse has the ability to automatically change all source files so they make use of the new name of the class.

7.2.3 Portable device: Microsoft Pocket PC & Jeode PersonalJava

We make use of HP iPAQ h5550 devices (see Figure 2.7) which are recently-launched Pocket PCs running Microsoft's Pocket PC 2003 software and the .NET Compact Framework. The h5550 has integrated Bluetooth and WiFi (802.11b) and infra-red support, eliminating the need to add these features via expansion cards. In addition, they have fast Intel XScale PXA255 400Mhz processors and 128MB memory which

provides ample storage space for installing our code and supporting tools such as prerequisite client libraries.

The h5550 is bundled with a Java Virtual Machine (JVM) from Insignia Solutions known as JeodeRuntime. This implements Sun's PersonalJava 1.2 specification [SUNM04b]. PersonalJava Is gradually being phased out and replaced with J2ME, however we make use of PersonalJava since it is supplied with the iPAQ and specifically designed to support it. The limitations of PersonalJava is that it is a reduced version of the J2SE v1.1 API, which is quite dated and does not include a large number of classes that we have come to expect from most recent versions of the Java API such as 1.4. In addition, the Swing graphical libraries are not provided in PersonalJava and AWT is the only choice. However for the purposes of our relative simple 'proof of concept' simulation, this is not a significant cause for concern. If implementing this type of middleware for a real application, we would advise the choice of the newer J2ME with the enhanced profiles such as the 'Personal Profile' that provides a richer API.

7.2.4 Event handling: Elvin

As discussed earlier in this report, the Elvin events system is provided as two components – the Router product which is effectively the events engine, and a set of client libraries that allow devices to interact with the event bus. Since our prototype is based on the Java platform, we have selected the Elvin Java SDK which is supplied as a JAR file that simply needs to be installed onto every client device. The advantages of the Elvin Java SDK are that it provides automatic reconnection support. If the network connection is interrupted, reconnection to the Elvin Router will take place automatically and seamlessly. The SDK also supports HTTP tunnelling, which is useful for devices that are behind a firewall, or can only make outbound communication in a restricted manner. The Elvin Java SDK has been fully tested against PersonalJava 1.2 and the two are therefore an appropriate choice.

The Elvin Router currently only supports desktop and notebook platforms. In the future, as the processing power and capabilities of handheld devices increases, support may be added for them. Therefore for the purposes of our simulation we installed the Elvin Router on a IBM Thinkpad 570 notebook computer, running Microsoft Windows 2000. The notebook was equipped with a Netgear 802.11b Wireless Ethernet card. The Elvin Router provides a console window that shows its activity. This is useful for debugging and a screenshot is shown in Figure 7.1. In this example, the Elvin Router receives a subscription request from a device and subsequently sends a quench event to those devices that generate that can generate the event specified in the subscription.

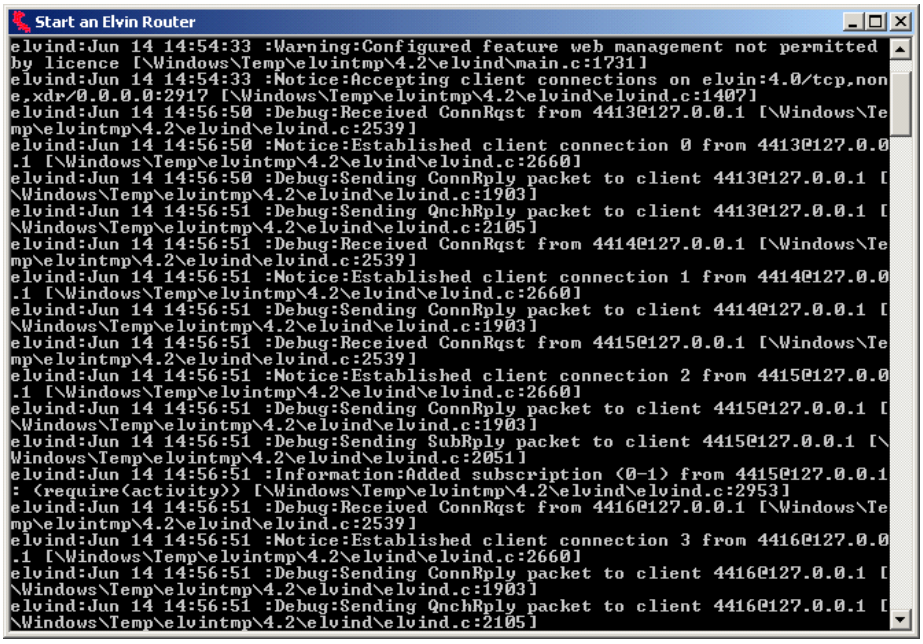

Figure 7.1: Screenshot of an Elvin Router console

Whilst Elvin is now developed by Mantara and is a commercial product, their Academic Program provides academics with free licences without support.

7.3 Standards & development practice

7.3.1 Coding standards

In order to improve the maintainability of our source code, and reduce the effort needed to make future changes, we adhere to a set of development standards based on those that have been widely used for other projects in the last few years. Firstly, for the naming of classes and interfaces we use the standard proposed by Sun Microsystems. All class names should start with an uppercase letter and underscores should not be used to separate words – instead, the words should be concatenated together with the first letter of each word capitalised. All other letters should be lower-case. Therefore *ImportantCoreClass* is acceptable, but notation such as *Important_Core_Class* or *importantCoreClass* is not.

Furthermore, we make use of packages to provide some logical grouping of classes. Our package hierarchy is presented in Section 5.8.1) and attempts to split the code into packages based on a division by sub-system.

Method names also follow the Sun Microsystems standard. Methods must always start with a lower-case letter. Again, underscores should not be used to connect words – instead, the words should be concatenated with

the first letter of each word (except the first) capitalised. All other letters should be lower case. Therefore *getDomainData()* is acceptable, but *GetDomainData()* is not.

We use a similar notation for variable names, with the first letter in lower-case and the first letter of each additional word in uppercase. However, constant values are entirely capitalised, e.g. *CONSTANT*, and class variables (i.e. static variables) are prefixed with two underscores, e.g. *__class_variable*. We do not use any notation to distinguish between instance variables and local variables.

In terms of commenting style, since this is a simulation and not a full implementation we do not use the JavaDoc standard. However the code has been well commented where appropriate using the standard "//" comment prefix.

7.3.2 Source code control

During the development, we made use of the Concurrent Versions System (CVS) [COLLWWW] as a repository for our source code. Whilst we were not carrying out any distributed development, CVS is useful for building a history of file versions. In addition, we can tag versions of files into a release, allowing us to fetch those same set of files again if it is necessary to revert to an old version of the system. To make the administration of CVS easier, we used the WinCVS tool [WINCWWW] which provides a GUI for the Microsoft Windows platform.

7.3.3 Unit & modularised testing

Since our architecture is divided into a set of sub-systems, we use unit and modularised testing in order to reduce the chance of errors when the system is brought together as a whole. Our practice is to test small components in isolation, including the behaviour of a class and then a package. Whilst formal, written test plans were not produced, ad hoc testing are still extremely useful at finding bugs in code at an early stage and reducing the time taken to resolve such problems in the future.

7.4 Prototype detail

7.4.1 Events system

The prototype model we have implemented demonstrates the application of the Elvin event-based middleware product in the domain of personal area networking. Whilst we are currently unable to run the Elvin Router on a handheld device, the Elvin client libraries were successfully installed and tested on a HP iPAQ Pocket PC device. The Elvin JAR file has a very

small footprint of about 250KB, which makes it suitable for installation on a wide range of devices that provide a Java Virtual Machine.

7.4.2 Networking technologies

As part of our tests, we tried three different types of networks:

- **WiFi 802.11b in infrastructure mode** – our iPAQ and notebook computer were connected to a D-Link DWL-700AP Wireless Access Point, and IP addresses were assigned by a DHCP server on the network.
- **WiFi 802.11b in ad hoc mode** – our devices were both set up in ad hoc node and used their own IP addresses in the 169.254.x.x range. In ad hoc mode, a Wireless Access Point is not required.
- **Bluetooth Personal Area Network (PAN)** – we set up a Personal Area Network between the iPAQ and a notebook PC with Bluetooth USB dongle. IP addresses for both devices were configured manually.

Elvin operated successfully over all of these bearers; however the range for Bluetooth was obviously much lower than for the WiFi solution. For the relatively simple tests we carried out, we did not notice any significant performance differences between the three solutions. Clearly, ad hoc networks are of greater interest to us, since in a personal area network, it is undesirable to have to implement components such as a DHCP server to assign IP addresses.

7.4.3 Functionality tests

We implemented a domain server based on a centralised model. Whilst our prototype implementation is relatively simple in design, it successfully demonstrates the ability to group devices and policies into domains, and the propagation of actions from a domain to its sub-domains. In addition, we implemented several policies as Java objects and corresponding policy management agents running as threads. As discussed earlier in this report, we have not developed a compiler to translate high-level policy syntax into Java code, and therefore the compilation is effectively carried out by hand.

Due to the fact that we had a relatively limited set of hardware for our prototype system, and that it was not feasible to develop instrumentation for devices such as MP3 players and other electronic devices, we used the iPAQ as a simulator for other hardware. The occurrence of events was simulated by making appropriate method calls. For example, we simulated an incoming call to a mobile phone, and demonstrated the transmission of this information to the event bus and on to consumers

subscribed to the event. We created several subscriptions to test that events are sent only to the devices that have requested them.

Since we discovered that the current version of Elvin only supports event quenching based on attribute names and not values, we simply added an attribute to each event, with the attribute name equal to the event name. This workaround enabled us to quench based on event names. We implemented several scenarios based on quenching to demonstrate that devices were reliably informed by the Elvin Router about changes to relevant subscriptions. In addition, devices in our personal area network stopped sending events on to the bus, when the number of consumers listening to an event fell to zero.

As part of this work, we were also interested in showing how actions to be performed on a device can be encapsulated inside an object by a PMA and transmitted to a device for execution. We chose this approach in our middleware architecture as an alternative to the approach of making remote method calls on a device. The implementation for this mechanism was carried out in the way in which it was proposed in our detailed architectural design; we provided functionality in the PMA to serialize the action object and sent it across as an event to the domain management agent, for distribution to its device members. The object was successfully deserialized by the devices and the actions carried out.

Due to time constraints, the context & correlation engine functionality was not implemented as part of our prototype. However in terms of providing support for correlated events, the CCE behaves in a very similar way to a policy management agent and subscribes to individual events with the Elvin Router.

7.5 Summary

The prototype phase has served as a useful ‘proof of concept’ demonstration of key parts of our middleware architecture using real hardware and software components. Despite the fact that we did not make use of a range of consumer devices such as mobile phones and music players, it was possible to simulate a range of scenarios using the HP iPAQ Pocket PC devices. We successfully demonstrated a working self-managed cell, composed of key management components and devices. We also showed that it is feasible for policy management agents to package actions within an object, serialize this into a byte stream and send it across to the devices that can reverse this process and carry out the necessary operations.

We did not focus on the aspect of performance testing as part of this work. However worthwhile future work could be conducted in determining how the system behaves when placed under varying levels of

stress. For example, we have assumed that our asynchronous event bus provides a reliable delivery mechanism, however it is quite possible that with much higher loads than we have applied, events may not always be correctly delivered or may be subject to delays.

8 Evaluation and future directions

In this final chapter to our report, we carry out a review of the project and provide an overview of the work that we have carried out. In addition, we attempt to highlight both the key contributions and limitations of our architecture so that we are able to make suggestions as to what we perceive to be the potential future directions for this work in particular and the research area in general.

8.1 A review of our work

We started the first chapter of this report by considering a hypothetical situation involving our two fictional characters, Alice and Bob. The purpose of this was to illustrate a typical type of problem that many of us face today. We own an increasing number of intelligent personal devices that are designed to improve our standard of living, yet we find ourselves spending more and more time configuring and operating them. Unfortunately, the vast majority of consumer devices require a considerable amount of user interaction, and are unable to readily interact with one another in an autonomic fashion to deliver an integrated service to the user. In order to further motivate our work, we present a different scenario in which Alice and Bob experience the benefits of a world in which personal devices are able to collaborate, exchange data and react to one another in an intelligent and seamless manner. Our objective has therefore been to develop a middleware solution that will enable these types of interactions, and minimise the level of input required from the user.

We moved on to a background study, covering a wide range of pertinent research areas such as policy specification, middleware systems, ad hoc networking, and recent developments in technologies and platforms for portable devices. In addition, we looked at relevant research projects such as MIT's Project Oxygen, and AMUSE which is being jointly undertaken by Imperial College and the University of Glasgow. This process assisted us in developing a specification for our project, including the issues that we decided to tackle as part of our proposed middleware architecture.

We then presented our design, based on the concept of a self-managed cell as proposed by the AMUSE project. Our architecture includes a set of management components that can potentially be distributed across devices in a personal area network. Communication between management components and devices takes place via a single asynchronous event bus which is based on the 'one-shot' paradigm rather than a request-reply approach. We make use of parts of the Ponder policy specification language and propose a few extensions as well in

order to describe policies that influence the behaviour of devices and also of management components. We subsequently provide a detailed architectural diagram, breaking our solution into sub-systems and suggesting a software implementation using UML notation.

Our case study was designed to highlight a potential application of the technology that we have developed, translating software design and implementation into real benefits at the application and user levels. In this section, we presented a guide to the kinds of possibilities that our architecture provides, and how the system can be configured to behave in the required way. Finally, we discussed a prototype implementation that was built to test several of the concepts that we have proposed.

8.2 Key achievements & contributions

Our overall objective for this project was to develop a set of middleware that allows devices within a personal area network (PAN) to communicate together in an effective, efficient and appropriate manner, with a view to minimising the amount of input needed from the end of user. Here we consider specific aspects of our work that we consider to have made a positive contribution in this area. We will then move on to look at problems and limitations of our approach in the following sub-section.

8.2.1 A new approach and application domain

The application of policy specification to the area of personal area networks is a relatively new one. In particular, very little work has been carried out to date that is related specifically to the autonomic interaction of personal consumer devices. Whilst the work being carried out at MIT on Project Oxygen advocates a ubiquitous human-centred architecture in which the management functions are effectively hidden from users, the emphasis is on allowing users to interact with the system using advanced speech and vision techniques. In addition, they focus on allowing users to be able to pick up “anonymous” devices and for those devices to adapt to a user’s profile for a relatively short period of time. Our approach is quite different, since we envisage that personal devices in our self-managed cell will typically be owned by a single user. We have therefore focussed quite specifically on defining the structure of a self-managed cell, the behaviour of the various management components within the cell and how we can use policies to define the autonomic interaction of devices.

8.2.2 Our proposed extensions to the Ponder language

We provide a flexible and extensible architecture that uses policies not only to define interaction between devices, but also to configure the SMC – we refer to these as “SMC policies”. We advocate the use of policies for

all aspects of configuration. Whilst we have made use of the existing Ponder policy specification language, we have proposed extensions in a few areas. We use the notation "**inst config**" to define SMC policies. SMC policies would typically be compiled down into code in addition to obligation policies. We have also proposed an alternative type of obligation policy to be used for 'derived' events. These are events are based on the occurrence of a series of other events. We use the "**derived**" keyword and capture the event pattern as a string inside parentheses, e.g. derived("2*eventA → eventB") evaluates to true is A occurs followed by A again, and then B. In addition we use the keyword "**event**" as part of the do condition to indicate that that action involves generating another event, rather than performing any operations on devices.

Whilst Ponder makes use of syntax such as:

when *time == "1600"*

we make use of a reserved keyword "**context**" to specifically indicate that data is being obtained from something external to the cell. E.g.

when *context(time) == "1600"*

8.2.3 Correlated events

One of the key management components we have proposed is capable of dealing with correlated events. In the sub-section above, we looked at the use of the "**derived**" keyword in policies that define these events. The purpose of supporting correlated events is our view that enabling policies based on raw event data alone is not helpful, since in many cases we are much more interested in responding to particular patterns of events.

8.2.4 Event quenching

Our implementation uses a quenching mechanism to improve the efficiency of the system. Since it is undesirable for devices to flood the event bus with events that are not required by any other entity, our scheme uses the Elvin product to pass subscription information on to devices such that events are only transmitted if there is interest in them.

8.2.5 Encapsulation of actions in serializable objects

We take a different approach to the way in which actions are carried out on a 'target'. Rather than the 'subject' making a remote method call on the target, we instead package the action inside an object and send it across to the target for execution. The advantage of this approach is that it enables us to include conditions that can be evaluated against each

target object individually, in the case where the target consists of a group of objects. The actions may therefore apply to some of the objects in the target group, but not others. The disadvantage of this approach is that the onus to execute the actions falls on the target rather than the subject, and of course there are issues of security and trust surrounding this. However since our application domain is a personal consumer network, trust is unlikely to be a significant issue.

8.2.6 Distributed domain structure

We propose a scheme in which each domain is managed by a Domain Management Agent (DMA) which runs as a thread. Our design enables the domain structure to either be centralised or distributed, depending on requirements. In certain circumstances, it may prove beneficial to have parts of the domain hierarchy split across various devices.

8.2.7 Multi-threaded policy management agents

Our policies are managed by policy management agents and each of these runs as a thread. This model not only allows us to distribute the policies across multiple devices, but also means that we minimise the delay between the policy sub-system and other entities. A policy management agent can receive events directly from the event bus and these are queued using the mailbox mechanism that we proposed earlier. We make the assumption that threads are generally extremely lightweight, so running a reasonably large number of threads is not likely to result in performance issues.

8.3 Limitations

Here we aim to provide a self-critical analysis of the project, highlighting limitations of our work and discussing the key problems that were faced. Due to project timescales, a number of desirable features could unfortunately not be designed and/or implemented.

8.3.1 Policy compilation

A significant area that this project has not covered at all relates to the compilation of policies from the OCL-like syntax used by Ponder into an implementation language such as Java. Whilst we have described policies using OCL syntax in the design chapters and in the case study, we have made the assumption that compilation will be carried out by hand. An interesting exercise would be to extend the existing Ponder compiler to be able to generate code for the types of policies that we are using here.

Rather than compiling our policies from OCL into Java for example, a better approach may be to generate an intermediate XML representation from the OCL which can then be translated into code for a language such as Java or C#.

8.3.2 Security & trust issues

Issues of security and trust are generally extremely important in distributed applications, though for the purposes of this project we focussed on the structure of a self-managed cell and component interactions, rather than on the security-related aspects. For example, we have assumed that an action sent to the target will be executed upon arrival. Since no reply is returned to the PMA, we have no way of knowing whether or not the action was actually carried out.

There are a number of issues that should be considered in more depth, including defining who has access to information and how we enforce such a security model. For example, we may only wish to give certain policies the ability to carry out privileged operations on a set of devices. In addition, whilst we distinguished between system policies (i.e. those that are semi-static and tend to be embedded in firmware) and user policies (i.e. those that are configurable by the user), we have not looked at how this would be enforced.

8.3.3 Conflict detection & resolution

There are a number of cases in which conflicts may occur. For example, two policies may trigger at the same time, yet may disagree on what action needs to be carried out. We can divide conflicts into those that can be detected at compilation time, and those that can only be detected at runtime. The latter typically pose more arduous challenges, since we need to be able to detect and resolve these 'on the fly'. We did not consider conflict detection, however methods based on priorities or timestamps are a good starting point for investigation in this area.

To a limited extent, we did however consider the issue of conflicts with regards to deciding which domains a device should belong to. Our solution involved adding a device to both groups in the case of a conflict, but there are clearly a number of other, potentially better approaches to the problem.

8.3.4 Semantics for correlated events

We proposed a simple state machine model for keeping track of correlated events. However there are a number of issues surrounding this that we have not considered in any significant depth. These include determining how events should be consumed when a full or partial match

occurs. In addition, if two consecutive events of the same type occur, yet we have only matched on one of these, we need to decide which event to choose when we retrieve data from the event. We may wish to choose the most historic or perhaps the most recent.

8.3.5 Cell interaction

Our original intention was to spend some time investigating the possible ways in which self-managed cells can interact with each other to co-operate in a layered or peer-to-peer manner, or to compose larger structures. Unfortunately due to time constraints we were unable to investigate these interesting paradigms. However we envisage that one possible model for inter-cell communication may be a “roaming” model, analogous to cellular phones in a GSM network. Devices may be allowed to roam into ‘foreign’ cells for short periods of time and be provided with a restricted level of privileges.

8.4 Suggested directions for future work

By considering the limitations of our work above, we have already described several of the possible areas for future extensions to this project. In our opinion, the implementation of a security model would significantly enhance the usefulness and potential applications of the architecture. Enforcing security policies in an efficient manner is extremely important if cells are to communicate with one another, or devices allowed to overlap more than one cell.

Our prototype model was somewhat restricted by the lack of a suitably rich Java implementation for a portable device. Unfortunately many hardware features cannot be addressed via PersonalJava or J2ME, however developments in this arena are almost certainly likely to increase the possibilities available to us within the next few years. A related area that is also becoming increasingly feasible involves learning from user behaviour. In addition to having policies that are based on events and allowing users to configure their own policies, a more useful system may allow for policies to be generated dynamically and in a seamless manner. This eliminates the need for users to have to learn policy syntax – they should be able to make changes via their device’s graphical user interface.

Finally, we believe there is a significant amount of work to be done to develop standards for device profiles. Just as the Bluetooth profiles were developed by a consortium of key industry players, it may be worthwhile to set up a similar special interest group to investigate and attempt to standardise profiles for device interaction. A profile should define the events that the device is capable of generating, and the actions that it is

capable of handling. There are various issues surrounding profiles, such as maintaining backwards compatibility with older devices when newer versions of profiles become available. In addition, whilst Bluetooth profiles tend to be static from the time of manufacture, we believe that it should be feasible to update profiles via firmware updates in order to ensure that users receive the benefits of new profiles as and when they are released.

A1 Requirements capture

In this appendix we present details of our requirements capture exercise. We carried this out before embarking on the high-level system design. The requirements have been divided into *primary* and *secondary,* where the former category is used to identify those requirements that are considered to be critical. Note that within each category, the requirements are listed in no particular order:

Req ID	Requirement description
PRIMARY REQUIREMENTS	
P1	The system will provide a domain server with hierarchical structured domains that can contain devices and policies.
P2	Devices can belong to one or more domains depending on the functionality they implement.
P3	The system will be able to interpret 'system' policies that define the behaviour of devices.
P4	It will be possible to add, remove and temporarily disable policies.
P5	Policies will also be stored in domains for manageability.
P6	Policy management agents (PMAs) will be provided that, when active, will be responsible for executing one or more policies.
P7	The system will support the dynamic addition and removal of devices from the cell.
P8	Devices will be managed by one cell only.
P9	A discovery service will be provided as part of the management functionality and discovering the presence of devices will be the sole responsibility of this component.
P10	The discovery service will send out a message to registered devices on a regular interval in order to determine if they are still in range.
P11	The discovery service will generate events of its own onto the event bus when devices are discovered or lost.
P12	The system will provide an asynchronous event bus that will be used for all communication between entities.
P13	The "target" for a policy can be either a domain or a specific device. Where it is a domain, the action will be carried out on all of the entities in that domain.
P14	The event service will contain buffers to aggregate events, e.g. look for specific patterns as specified by policies.
P15	The system will support policies that trigger based on aggregated event occurrences, e.g. if an event occurs a certain number of times or if events occur in a specific pattern.
P16	The system will support external events, i.e. those generated by devices that are effectively not part of any cell.
P17	Policies can be based wholly or partly on external events, and can also evaluate conditions based on external devices.
SECONDARY REQUIREMENTS	
S1	Users will be able to specify 'user' policies to customise the behaviour of the system.
S2	Events will inherently contain a timestamp indicating the time at which the event was generated. This information will be

	accessible by the policy management agent.
S3	The system will write cell management data such as domain members to disk upon shutdown of a cell, such that the information can be reinstated at startup.
S4	Network traffic will be kept to a minimum to economise power usage.
S5	The behaviour of the discovery server will be configurable via internal policies, e.g. frequency of “ping” message, number of retries.
S6	The event bus must have minimal latency – messages must pass through and be delivered to their destination(s) as quickly as possible.
S7	The event bus must provide high availability and reliability since communication is asynchronous in nature.

A2 Bibliography

[AFFI04] Affix in a Nutshell: Open Source Bluetooth Protocol Stack for Linux: Usage manual, http://affix.sourceforge.net/affix-doc/c1051.html

[BANT03] Bantz, D.F., et al, "Autonomic personal computing", IBM Systems Journal, vol. 42, no. 1, 1993, pages 165 – 176.

[BUSN03] BusinessWeek: Tech Wave 1: Utility Computing, http://www.businessweek.com/magazine/content/03_34/b3846619.htm

[CAPK04] Capkun, S., Hubaux, J-P., Jakobsson, M., "Secure and Privacy-Preserving Communication in Hybrid Ad Hoc Networks", EPFL-IC Technical Report no. IC/2004/10.

[COLE98] Coleman, D., "A Use Case Template: draft for discussion", available at Bredemeyer Consulting, http://www.bredemeyer.com/pdf_files/use_case.pdf

[COLLWWW] Concurrent Versions System, https://www.cvshome.org/

[CURB02] Curbera, F.; Duftler, M.; Khalaf, R.; Nagy, W.; Mukhi, N.; Weerawarana, S.; Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI, Internet Computing, IEEE, Volume: 6, Issue: 2 , March-April 2002, Pages:86 - 93

[CYBIWWW] Cybiko Inc., http://www.cybiko.com

[DAMI99] The Policy Framework, http://www.doc.ic.ac.uk/~ncd/policies/PolicyFramework.html

[DAMI01] Damianou, N., Dulay, N., Lupu E., Sloman M., "The Ponder Policy Specification Language", Proc. Policy 2001: Workshop on Policies for Distributed Systems and Networks, Bristol, UK, 29-31 Jan. 2001, Springer-Verlag LNCS 1995, pp 18-39.

[DES03] Desai B., Verma V., Helal S., "Infrastructure for Peer-to-Peer Applications in Ad-Hoc Networks", Submitted to 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03), Berkeley, CA, February 2003.

[DMTF04] Distributed Management Task Force (DMTF) – CIM Schema – Policy Model, http://www.wbemsolutions.com/tutorials/CIM/cim-model-policy.html

[DMTF04b] DMTF – Directory Enabled Network (DEN) Initiative, http://www.dmtf.org/standards/den/

[DORS98] Dorsey, J., "A performance comparison of multi-hop wireless ad hoc network routing protocols", Carnegie Mellon University, Slides from the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98), http://www-2.cs.cmu.edu/~wearable/group/slides/ad-hoc-performance/img0.htm

[DSOL03] IEEE Distributed Systems Online: Middleware, http://dsonline.computer.org/middleware/index.htm

[DSTCWWW] Distributed Systems Technology Centre: Elvin content based messaging, http://elvin.dstc.edu.au

[ECLIWWW] Eclipse.org, http://www.eclipse.org

[GROT01] Groten, D., Schmidt, J.R. "Bluetooth-based Mobile Ad Hoc Networks: Opportunities and Challenges for a Telecommunications Operator", IEEE VTS 53rd Vehicular Technology Conference, VTC 2001 Spring, pp. 1134-1138, May 2001.

[HENR02] Henry, P.S., Luo, H., "WiFi: What's Next?", IEEE Communications Magazine, December 2002, pages 66 – 72.

[HOR02] Horozov T., Grama A., Vasudevan V., Landis S., "Moby – A Mobile Peer-to-Peer Service and Data Network", Proceedings of the International Conference on Parallel Processing (ICPP'02), 2002.

[HUBA01] Hubaux, J-P., Buttyán, L., Capkun, S., "The Quest for Security in Mobile Ad Hoc Networks", Proceedings of the ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHOC 2001).

[IBMR96] IBM Almaden Research Center, PAN Fact Sheet: Hi-Tech, Hi-Touch, http://www.almaden.ibm.com/cs/user/pan/pan.html

[JAVA01] JavaMobiles: List of JVMs for PDAs, http://www.javamobiles.com/jvm.html

[JXMEWWW] JXTA for J2ME (CLDC/MIDP), http://jxme.jxta.org/

[LUPU03] Lupu, E., Sloman, M., Dulay, N., Sventek, J., "AMUSE: Autonomic Management of Ubiquitous Systems for e-Health", http://www.doc.ic.ac.uk/~ecl1/projects/AMUSE/

[KORT02] Kortuem G., Schenider J., Preuitt D., Thompson T. G. C., Fickas S., Segall Z., "When Peer-to-Peer comes Face-to-Face: Collaborative Peer-to-Peer

Computing in Mobile Ad hoc Networks", First International Conference on Peer-to-Peer Computing (P2P'01), August 27 - 29, 2001, Lingköping, Sweden.

[MANEWWW] Mobile Ad Hoc Networking (MANet) – IETF Working Group Information, http://protean.itd.nrl.navy.mil/manet/manet_home.html

[MANT04] Elvin Subscription Language Reference 4.0, http://www.mantara.com/support/docs/guides/elvin-sublang/elvin-sublang-4.0.pdf

[MANTWWW] Mantara Software, http://www.mantara.com/

[MOBI03] Mobilni – Internet Magazin, June 2003, http://www.mobilni.co.yu/2003_jun/p_nokia_6600.htm

[NOKI03] Forum Nokia: J2ME & Symbian OS: A Platform Comparison, http://ncsp.forum.nokia.com/downloads/nokia/documents/J2ME_Symbian_OS_1_01.pdf

[OXYGWWW] MIT Project Oxygen, http://oxygen.lcs.mit.edu/

[PREN04] Prentice Hall: Bluetooth and other wireless technologies, http://www.phptr.com/articles/article.asp?p=24265&seqNum=3

[REIL00] Java RMI & CORBA: A comparison of two competing technologies, http://www.javacoffeebreak.com/articles/rmi_corba/

[SCHU04] Schumacher, A., Painilainen, S., Luh, T., "Research Study of MANET Routing Protocols", Department of Computer Science, University of Helsinki, Finland. Available at http://www.cs.helsinki.fi/u/kraatika/Courses/IPsem04s/manet.pdf.

[SLOM02b] Morris Sloman, Naranker Dulay, and Emil Lupu. The PONDER Policy Based Management Toolkit, August 2002. http://www-dse.doc.ic.ac.uk/Research/policies/ponder/PonderSummary.pdf.

[STRE04] Streettech.com: Cybiko, http://www.streettech.com/archives_gadget/cybiko.html

[SUNM04] Sun Microsystems: Deploying Wireless Java Applications, http://developers.sun.com/techtopics/mobility/midp/articles/bluetooth1/

[SUNM04b] Sun Microsystems: PersonalJava Application Environment, http://java.sun.com/products/personaljava/index.jsp

[SUNX01] Sun, J., “Mobile Ad Hoc Networking: An Essential Technology for Pervasive Computing”, Proc. International Conferences on Info-tech & Info-net, Beijing, China, C:316-321.

[SVEN03] Sventek, J., “AMUSE – Autonomic Management of Ubiquitous Systems for e-Health”, PowerPoint Presentation Slides, University of Glasgow.

[WALD03] Waldrop, M.M., “Autonomic Computing: The Technology of Self-Management”, The Future of Computing Project, Woodrow Wilson International Center of Scholars, July 2003, available from http://www.thefutureofcomputing.org/Autonom2.pdf

[WINCWWW] CvsGui: WinCVS, http://www.wincvs.org/

[XMLB04] xmlBlaster: Performance tests, http://www.xmlblaster.org/performance.html

[XMLBWWW] xmlBlaster, http://www.xmlblaster.org

[YUAN04] Comparing .NET Compact Framework with J2ME, http://www.enterprisej2me.com/pages/resources/compare.php

[ZIMM96] Zimmerman, T.G., “Personal Area Networks: Near-field intrabody communication”. IBM Systems Journal, Vol. 25, No. 3 and 4, 1996.