A Middleware for True Mobile Agents in Wireless Sensor Networks

Ciarán Lynch
*Electronic Engineering Department, Cork Institute of Technology, Cork, Ireland.*

Follow this and additional works at: https://sword.cit.ie/allthe

Part of the Electrical and Electronics Commons

**Recommended Citation**
Available at: https://sword.cit.ie/allthe/207*

This Doctoral Thesis is brought to you for free and open access by the Dissertations and Theses at SWORD - South West Open Research Deposit. It has been accepted for inclusion in Theses by an authorized administrator of SWORD - South West Open Research Deposit. For more information, please contact sword@cit.ie.
A Middleware for True Mobile Agents in Wireless Sensor Networks

Ciarán Lynch

CIT LIBRARY
REFERENCE ONLY
A Middleware for True Mobile Agents in Wireless Sensor Networks

Ciaran Lynch, BE
Centre for Adaptive Wireless Systems
Electronic Engineering Cork Institute of Technology

Supervised by Dr. Dirk Pesch

A thesis submitted for the degree of

Doctor of Philosophy

to Cork Institute of Technology, August 2009
Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material to a substantial extent has been accepted for the award of any other degree or diploma of the university of higher learning, except where due acknowledgement has been made in the text.

Signature of Author:

Certified by:

Date: 27/10/09
Acknowledgements

I would like to thank everybody in CIT and elsewhere who have helped me over the last few years. Dr Dirk Pesch for his supervision, encouragement and support. Susan for her strictly professional advice, Alan and Chong for showing me that it is possible to finish (and fouling), and Olivia for always encouraging everyone. Aisling, Dave, Gary, Jian, Mary, Niel, Widy and everyone else in CIT for making it a more pleasant place to work, as well as all the others who have passed through in the last few years.

I would also like to thank Cass for listening to me giving out about this work for years. I couldn't have done it without her, and of course my parents and family for helping me along the way.
List of Publications


Abstract

The application space of wireless sensor networks has evolved considerably from their initial role as an efficient data gathering system. Modern wireless sensor networks are expected to be dynamic, powerful, reliable and flexible, while continuing to operate efficiently and autonomously. The supporting software systems have not kept pace with this development, and currently limits the deployment of complex and robust applications. Mobile software agents present a powerful approach to application design, however no existing middleware for wireless sensor networks supports native mobile software agents.

The main contribution of the work presented in this thesis is the development and evaluation of a middleware supporting powerful mobile software agents for wireless sensor networks. A hybrid memory allocation strategy is developed to encapsulate the mobile agent, a code caching strategy is used to reduce unnecessary transmission and directed movement is used to simplify agents and to allow attribute-based movement and operation. These improvements act to reduce the overall energy consumption of the agent system.

A number of application scenarios are used to evaluate the system performance. The primary defining characteristic of wireless sensor network systems is their low energy consumption, and the proposed system demonstrates lower-power operation than existing mobile agent systems in environments where complex, mobile in-network processing, control or actuation are required, particularly when code caching can be exploited. This is best demonstrated by the reliable control system, that is capable of dynamically adapting to a changing topology, reliably recovering from node and communication failures while
continuing to provide application functionality in unaffected systems, while still leaving enough resources free to allow independant applications to run on the wireless sensor network.
# Contents

## 1 Introduction

1.1 Motivation & Thesis .................................................. 3
1.2 Requirements .......................................................... 5
1.3 Contributions .......................................................... 6
1.4 Definitions .............................................................. 8
1.5 Thesis Outline .......................................................... 8

## 2 State Of The Art

2.1 Challenges of WSN Programming ................................. 11
  2.1.1 Mobility Model and Latency ................................. 12
  2.1.2 Execution Model ................................................... 14
  2.1.3 Security ............................................................... 15
  2.1.4 Reliability ............................................................ 15
  2.1.5 Energy Consumption ............................................. 16
2.2 Programming Models for WSNs ................................. 17
  2.2.1 Data Gathering ..................................................... 19
  2.2.2 Declarative .......................................................... 21
  2.2.3 Publish/Subscribe ............................................... 22
  2.2.4 Functional Programming ....................................... 23
    2.2.4.1 Native Stateless ............................................. 23
    2.2.4.2 Interpreted Stateless ..................................... 26
    2.2.4.3 Interpreted Stateful ..................................... 27
2.3 Application Level ..................................................... 29
  2.3.1 Application Requirements ..................................... 30
  2.3.2 Application Performance ....................................... 32
CONTENTS

2.3.2.1 Mobility Model and Latency .................................... 33
2.3.2.2 Execution Model and Energy Consumption ........ 33
2.3.2.3 Reliability .......................................................... 34
2.3.2.4 Security ............................................................. 35
2.3.3 Existing Agent Systems for WSNs .......................... 37

2.4 Middleware and Mobile Agents ................................................. 38
2.4.1 Middleware .............................................................. 39
2.4.2 Characteristics .......................................................... 41
2.4.3 Self-Stabilisation ........................................................ 42
2.4.4 Existing Agent Systems ............................................ 43

2.5 Native Mobile Agents ............................................................ 44

3 Middleware Architecture .......................................................... 48
3.1 System Architecture ............................................................. 49
3.1.1 Mobile Agent Middleware .......................................... 50
3.2 Operation Manager ............................................................. 53
3.2.1 Agent List and Management ........................................ 54
3.2.2 Inter-Agent Communication ......................................... 55
3.2.3 Communication Architecture ......................................... 56
3.2.3.1 Interface ............................................................. 58
3.2.3.2 Block Writes ....................................................... 59
3.2.3.3 Direct Remote Read ............................................. 59
3.2.3.4 Neighbour Query ............................................... 59
3.2.3.5 Broadcast Remote Read ...................................... 60
3.2.3.6 Conditional Remote Read .................................... 60
3.2.4 Neighbourhood Support ............................................. 61
3.2.5 Routing ................................................................. 62
3.2.6 Agent Termination ..................................................... 64

3.3 Mobility Manager ............................................................... 64
3.3.1 Mobility Model .......................................................... 64
3.3.1.1 Agent Mobility ..................................................... 65
3.3.2 Encapsulated Agent .................................................... 67
3.3.2.1 Agent State ........................................................ 67
CONTENTS

3.3.2.2 Agent Encapsulation ................................................. 68
3.3.3 State Reconstruction ...................................................... 69
3.3.4 Agent Push ........................................................................... 72
3.3.5 Service Pull ........................................................................... 76
3.3.6 Broadcast Push ................................................................. 78
3.3.7 Carrier Move ........................................................................ 81
    3.3.7.1 Move Conditions ......................................................... 83
    3.3.7.2 Termination Conditions ............................................... 84
    3.3.7.3 Examples .................................................................... 84
    3.3.7.4 Default Behaviour ....................................................... 86
3.3.8 Code Caching ....................................................................... 87

3.4 Security Manager ..................................................................... 87
    3.4.1 MD5 .................................................................................. 88
    3.4.2 Secret Key .......................................................................... 89
    3.4.3 Signing Process ................................................................. 91
    3.4.4 Receiver ............................................................................. 91

3.5 Agent Server ............................................................................ 92

3.6 Agent Structure and API .......................................................... 93
    3.6.1 OS Requirements .............................................................. 93
    3.6.2 API Summary .................................................................... 94

3.7 Conclusion .............................................................................. 95

4 Implementation ........................................................................... 98
    4.1 SOS Implementation .......................................................... 99
        4.1.1 SOS application ............................................................. 100
        4.1.2 Scheduler ................................................................. 101
            4.1.2.1 Code ID ................................................................. 102
            4.1.2.2 Module Suspend .................................................... 103
        4.1.3 Memory Management .................................................. 104
            4.1.3.1 FLASH Buffer ........................................................ 105
            4.1.3.2 Local and Remote ................................................... 106
            4.1.3.3 State Snapshot ......................................................... 106
        4.1.4 Module Distribution Protocol ......................................... 107
CONTENTS

4.1.5 Debugging System ................................................. 107
4.2 Programming Restrictions ........................................... 109
  4.2.1 Necessary operations for mobility ......................... 110
4.3 Agent Structure ..................................................... 111
  4.3.1 Middleware Size .................................................. 114
4.4 Hardware Emulator .................................................. 114
  4.4.1 AvroraZ ............................................................... 115
  4.4.2 Physical Simulation Extension .................. 117
4.5 Agent Server ......................................................... 118
  4.5.1 Server Implementation ........................................ 118
  4.5.2 GUI ........................................................................ 119
4.6 Conclusions ............................................................ 120

5 Evaluation .............................................................. 122
  5.1 Evaluation Criteria ................................................. 123
  5.2 Application Scenarios ............................................... 124
    5.2.1 Physical Scenario ............................................... 126
  5.3 Data Gathering ........................................................ 127
    5.3.1 Mobility Model .................................................. 128
      5.3.1.1 Addressing .................................................... 132
      5.3.1.2 Statefulness .................................................. 133
    5.3.2 Energy Consumption ........................................... 134
  5.4 Location Determination ........................................... 135
    5.4.1 Execution Model ............................................... 136
    5.4.2 Energy Consumption ........................................... 138
      5.4.2.1 Network Traffic ............................................ 140
  5.5 Reliable Distributed Control ..................................... 142
    5.5.1 Execution Model ............................................... 148
      5.5.1.1 Limitations .................................................. 149
    5.5.2 Movement Latency .............................................. 150
    5.5.3 Reliability ........................................................ 152
      5.5.3.1 Application-level Reliability ......................... 154
      5.5.3.2 Reliability Measure ...................................... 155
List of Figures

2.1 Application and Lower Layers .............................................................. 12
2.2 Classification of existing middleware ................................................. 19
2.3 Sample TinyDB Query .......................................................................... 20
2.4 Agilla VM (from [1]) .............................................................................. 28
3.1 System Block Diagram .......................................................................... 52
3.2 Detailed System Implementation ............................................................ 52
3.3 Agent Coordination (based on [2]) ........................................................ 56
3.4 Conditional Remote Read ...................................................................... 61
3.5 Conditional Remote Read Example ......................................................... 61
3.6 Example Module State, transfer form and reassembled ......................... 70
3.7 Transfer Block Structure ........................................................................ 71
3.8 Transfer Sending Side ............................................................................. 75
3.9 Transfer Receiving Side .......................................................................... 75
3.10 Transfer Endpoints ................................................................................ 76
3.11 Broadcast Push ...................................................................................... 80
3.12 Carrier Routing Move ........................................................................... 85
3.13 Carrier Gradient Move .......................................................................... 86
3.14 MD5 Block Diagram .............................................................................. 90
4.1 SOS Memory Map .................................................................................. 105
4.2 Example SOS Module State ................................................................... 111
4.3 Example SOS Module Header .................................................................. 112
4.4 Avrora Structure (from [3]) .................................................................... 116
4.5 Agent Server GUI ................................................................................... 120
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Packets by Implementation</td>
<td>142</td>
</tr>
<tr>
<td>5.2</td>
<td>Repeated Runs (7 hops)</td>
<td>143</td>
</tr>
<tr>
<td>5.3</td>
<td>Average temperature difference</td>
<td>148</td>
</tr>
<tr>
<td>5.4</td>
<td>Sample node availability, $P=0.001$</td>
<td>156</td>
</tr>
<tr>
<td>5.5</td>
<td>Average node availability</td>
<td>157</td>
</tr>
<tr>
<td>5.6</td>
<td>Broadcast Push Coverage</td>
<td>163</td>
</tr>
<tr>
<td>5.7</td>
<td>Agent Move Timing</td>
<td>167</td>
</tr>
<tr>
<td>5.8</td>
<td>Agent Execution Timing</td>
<td>174</td>
</tr>
<tr>
<td>5.9</td>
<td>Expected Lifetime, $T_e=0.1$</td>
<td>175</td>
</tr>
<tr>
<td>5.10</td>
<td>Expected Lifetime, $T_e=1.0$</td>
<td>177</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The development of Wireless Sensor Networks has revolutionised the area of embedded systems. Twenty years ago, embedded processors almost always operated in isolation. Each microcontroller was linked to a small set of sensors and actuators and served a single, static purpose. The application code on each microcontroller was simple and tightly coupled to the hardware and was designed for maximum efficiency and reliability.

Simple wired communication protocols opened up some communication channels between microcontrollers. Simple standardised interfaces such as SPI and I2C allowed inter-processor communication, although mostly between microcontrollers and external peripherals.

In 1999, researchers at the University of California, Berkeley attached a 9,600 baud, low-power, ISM-band radio to an Atmel microcontroller and created the first purpose-built Wireless Sensor Node[4], although this drew on the concepts of ambient intelligence initiated by the Smart Matter project at PARC[5; 6]. This allowed direct communication between the microcontrollers of physically separated sensor nodes. The node was no longer a self-contained processing device, but a member of a network of cooperating devices.

Wireless Sensor Networks were initially programmed using embedded C cross-compilers, in the same way as traditional embedded systems[4]. Over time, more complex systems of scripts and precompiler directives were built up to bring the programming of such systems away from the limitations of the hardware and closer to mainstream programming. Eventually these evolved into the nesC
language and the TinyOS operating system[7]. This presents a system of components that are statically linked together to form an application with much of the hardware interface hidden behind the operating system.

Support for dynamically reprogramming the wireless sensor node over the radio was added, by uploading a new program image over the radio and rebooting the node. This did not allow any state to persist over the reboot. More dynamic operating systems such as SOS[8] and Contiki[9] were developed that allowed only parts of the operating system to be replaced, allowing the node to update the program image without rebooting the node. More recent work allowed nodes with different roles in the network to have different program images, however the software image is still fixed at each node[10].

Parallels can be drawn with the development of software systems for desktop computers. Originally, a PC was a stand-alone system, running dedicated software, capable of performing one task at a time. Updates were made to the operating system manually, slowly and irregularly. This evolved into multi-tasking systems capable of switching between different applications but still limited in their connectivity. With the advent of the internet and permanently connected systems, the software image on a modern PC is constantly updated to incorporate security fixes, and to adapt to changing user requirements.

The limitations of this model appear when the software must evolve rapidly to deal with highly mobile or dynamic applications. The update speeds of tens of seconds or more, and the fact that the node can not continue to operate normally during this process mean that the slowly-moving software systems must have software pre-installed for all events that may be encountered. In order to overcome these difficulties, mobile agents were introduced for Wireless Sensor Networks. These were restricted to basic interpreted agents[11; 12], or else more capable systems that required megabytes of storage and sophisticated hardware systems[13; 14]. As the application program exists independently of the operating system, a layer is required that sits between the agent code and the operating system. This is a form of middleware, and supports the agent, allowing it to execute on the target platform, although in some systems it may also be part of the OS reprogramming model.
1.1 Motivation & Thesis

The middleware limits the execution model of the mobile agent. Interpreted agent systems are limited in the amount of state that each agent can carry, by the complexity of code that can be represented in the virtual machine and by the amount of resources and energy consumed by interpreting the mobile agent\[1; 15\]. Native code suffers less from such limitations, and has advantages in terms of power and flexibility, as well as having access to the entire resources of the sensor node.

This thesis presents a novel middleware platform supporting native mobile software agents in wireless sensor networks. This allows self-contained software agents to move around the network, following events of interest. The agent executes directly on the wireless sensor node, without any interpretation or translation. Mobile agent code is a new method of programming sensor networks and the challenges, advantages and limitations of this approach are considered.

1.1 Motivation & Thesis

Existing dynamic or mobile code systems in wireless sensor networks fall into one of two categories – dynamic code updating and interpreted mobile software agents. There are no existing systems that allow true mobile agents.

Dynamic code updating allows the running image on a sensor node to be changed after deployment, without physical access to the node. Algorithms can be used to vary the image assignment according to the role each node will perform in the final network\[10; 16\], however the image at each node is fixed in the short term and uploading a new image is a complex and invasive operation. Although optimisation can reduce the amount of data transferred, the node must be reset to enable the new image, causing all stored state to be lost. Transferring this large program image reliably also consumes a lot of energy.

The SOS operating system\[8\] took one step closer to true mobile agents by supporting dynamic updates to modules. The operating system is composed of a microkernel and a set of modules. Modules can be loaded and unloaded, started and stopped without interfering with other modules in the system. SOS only supports a single system image – viral propagation is used to distribute modules.
1.1 Motivation & Thesis

to each node in the network. The module does not contain any operating state and can not be considered an agent – it is simply an efficient code update mechanism.

Existing mobile software agent systems for embedded wireless sensor nodes are based on virtual machines or interpreted code[12; 17]. Efficiencies are achieved by limiting the size of the agent and by interpreting it at each sensor node, however there is a cost in terms of performance. The limited architecture of an embedded microcontroller is not well suited to interpretation of sophisticated languages. The hardware required to interpret a high-level language such as Java must contain megabytes of storage and a microcontroller operating on the 100s of MHz, such as the SunSPOT[18], consuming orders of magnitude more power than an embedded wireless sensor node – the environment presented in this thesis is necessarily more simple, to allow it to function on an embedded node with kilobytes of storage, operating in the low MHz.

These interpreted mobile agent systems are limited in the scope of application they can express. The simple byte-codes used to describe the agent and the severe limitations on agent state and storage do not allow the implementation of complex algorithms or protocols. Simple mobile agents can not perform their role autonomously in dynamic and unreliable networks, and they require external management and supervision to achieve reliable operation. A mobile agent system that is not constrained by a virtual machine would be able to carry out complex self-monitoring and support, allowing truly autonomous and reliable operation.

Native mobile agents expand the application scope of wireless sensor networks. They enable complex behaviour that can not be provided by interpreted mobile agents subject to the same resource constraints. This allows a mobile agent network to operate reliable and autonomously, providing overall system performance that can not be achieved by statically operating systems without considerable external management. The mobile agent programming paradigm provides a natural and powerful way of implementing distributed and self-managed applications.

The mobile software agents presented in this thesis are written entirely in the C programming language. While the C language is criticised as old-fashioned and minimalist, these features allow it to generate small and efficient code on a limited architecture such as an 8-bit microcontroller. In fact the architecture
1.2 Requirements

of a modern microcontroller is probably closer to the systems on which C was originally created than a modern microprocessor.

The mobile software agent system presented in this thesis presents an efficient and flexible solution to some of the challenges of wireless sensor networks. It represents another step in the evolution of wireless sensor network applications. Originally the application represented the entire software image executing on the sensor node – now the sensor node is part of the infrastructure, and the application executes on the network, not on any particular node. The application runs on the network – it is not the network.

The thesis of this work is therefore:

That native mobile software agents are a viable technology for resource-limited embedded wireless sensor nodes. That the power consumed during execution of a native agent is less than an interpreted agent, while the power and flexibility of the system is greater than a statically deployed system. That the native mobile agent system is capable of complex, reliable and autonomous operation that is not possible using alternative technologies.

1.2 Requirements

The challenges to the development of such a system are, like most issues with wireless sensor networks, related to the limited resources of the node. The system should be as powerful and flexible as possible, while still executing on a standard embedded wireless sensor node. It should be capable of supporting enough and as powerful agents to enable as wide a set of applications as possible, while keeping the system overhead as low as possible.

The research presented here has reviewed the state of the art in wireless sensor network application middleware and support, and identified the benefits a powerful mobile agent system.

The middleware system must operate reliably, with the minimum of overhead. It is capable of moving agents around the network reliably and efficiently – the transfer of an agent either completes successfully, with the agent starting at the
1.3 Contributions

The primary contribution of this thesis is the development and characterisation of a novel application middleware supporting mobile software agents in an embedded wireless sensor node environment. This middleware allows the encapsulation of code and operating state into a moveable mobile software agent that is then resumed at the destination. This operation is not available with any existing middleware for comparable hardware platforms, apart from interpreted systems which suffer from a severely limited execution environment.

In order to support such mobile agents, a novel hybrid memory allocation and encapsulation system was developed, in addition to a reliable transfer protocol and a novel caching system that allows reuse of code wherever possible, generating...
considerable saving in energy consumption. A mechanism and policy was defined to allow an agent to resume operating at a remote device in a consistent state with exact knowledge of which parts of its state it must manage.

An indirect movement management system was developed to allow agents freedom of movement without regard to a particular topology or particular nodes. This, in addition to middleware services serves to reduce the size of the agent and reduce the energy required to move it. A detailed application evaluation was carried out to quantify the reduction in energy consumption generated by these techniques.

In summary the contributions are:

- The development of novel techniques to enable mobile agents in a resource-limited wireless sensor network
- Presentation of a novel middleware architecture supporting and optimising native mobile agents on resource-limited wireless sensor networks
- The implementation of this architecture on a particular sensor node
- Characterisation of this implementation as a standalone system and also in networked scenarios

The proposed middleware expands the state of the art to include for the first time mobile natively executing software in embedded wireless sensor networks. This expands the application space to include mobile applications that require complex processing and management. This allows applications that would previously have required a sophisticated, expensive and power-hungry platform to execute on a cheap, reliable and efficient embedded wireless sensor node.

Mobile software agents are not the solution to every issue in wireless sensor networks. The overhead of transmitting the agent itself means that interpreted agents or specifically targeted database-like systems will outperform mobile software agents in some applications. However a range of areas are identified in which mobile software agents do provide a good compromise of power and flexibility over deployment time, supported by the characterisation established in this thesis.
1.4 Definitions

For clarity, the following definitions are presented here and are used throughout the thesis:

- The Operating System (OS) is the lowest level of code on a wireless sensor node. It interfaces with the hardware and provides simple services to higher layers and applications.

- The Middleware sits between the Operating System and the application code. It provides services to the application, managing and supporting complex operations at the application layer.

- A Code Module is a piece of code that can be started or stopped dynamically at a particular wireless sensor node. It is not mobile and each instance of the Code Module starts with the same Operating State.

- A Mobile Agent is a piece of code that is capable of movement between different nodes in the wireless sensor network, and of bringing its Operating State with it as it moves.

- The Operating State of a Code Module or Mobile Agent is the set of data that the Code Module or Mobile Agent requires to operate locally.

- A Service is a Code Module that can be brought to a particular wireless sensor node if it is not already available.

1.5 Thesis Outline

The remainder of the thesis is structured as follows:

Chapter 2 presents an overview of related work in wireless sensor networks. Existing work in mobile software agents, dynamic code updating and the various programming paradigms are discussed, and how the contribution of this thesis relates to this work.
1.5 Thesis Outline

Chapter 3 discusses the architecture of the mobile software agent middleware that is the primary contribution of this thesis. This includes the support necessary to dynamically link and unlink agents at a sensor node, the protocols used to transfer agents from one node to another and the optimisation, security and support functions implemented by the middleware to reduce agent size.

Chapter 4 presents an implementation of the mobile software agent middleware. This is based on the SOS operating system and MicaZ Wireless Sensor Nodes, and includes the Avrora emulation environment that is used for the evaluation.

Chapter 5 presents an evaluation of the middleware system. The emulation system used is discussed. The middleware system itself is evaluated under a number of criteria, mostly related to time taken or resource usage for various operations. These benchmarks are then used in conjunction with some complete-system scenarios to evaluate the system in comparison to a number of alternative approaches.

Chapter 6 concludes the thesis and discusses some future research directions opened up by this work.
Chapter 2
State Of The Art

In recent years, Wireless Sensor Networks (WSN) as a research tool have matured as a hardware platform. A small number of designs based on common, off-the-shelf technology have emerged as de-facto standards. The IEEE 802.15.4 (Zigbee) standard[19] has unified the RF physical layer and most continuing hardware research focuses on miniaturisation and optimising the power consumption of these platforms, although industrial systems still exist based on custom-built communication layers, using the ISM bands (433/868/900/2400MHz). Embedded wireless sensor nodes as a device class are based on microcontrollers with 2-10kB of RAM, and 64-256kB of program memory. Integrated sensors are generally few and simple, with expansion connectors allowing additional analog or digital sensors to be added.

At the operating system (OS) and supporting software (or middleware) level, this is not the case. Although TinyOS[7] is often cited as the standard WSN operating system, the limitations of its static component structure are becoming apparent as applications evolve from static, single-purpose systems. A modern application consists of a set of flexible, mobile and dynamic modules combining to carry out the application functionality. Extensions to TinyOS exist but they are still based on the underlying static component model.

The traditional concept of Wireless Sensor Network Operating System must change as the application space expands. Where in the past the OS was simply a Hardware Interface Layer (HIL) combined with a simple scheduler, an effective modern WSN OS now includes a sophisticated layer that sits between the
HIL and the application, providing services to support the application. This is known as middleware, and its design is critical to realising efficient and powerful applications.

Mobile applications are crucial to exploiting the potential of WSNs. The application space has expanded from simple distributed data gathering to dynamic event tracking, distributed control and completely autonomous network operation. Dynamically distributed code supports this easily and efficiently, moving between nodes in the network to follow dynamic processes. Existing systems support viral code propagation, however while executing, the code is fixed at a single node. Existing mobile agent systems are either very limited in scope, executing in a small virtual machine with very limited functionality and resources[12] that is not capable of the complex behaviour required by such applications, or far too resource-hungry to execute on an embedded wireless sensor node, requiring a processor executing at 100s of MHz, and MBs of storage space[14; 16], requiring larger devices with larger power consumption than a true embedded wireless sensor node.

The structure of the software layers of a modern wireless sensor network operating system and application are shown in Figure 2.1. The Hardware Level interacts directly with the hardware of the Wireless Sensor Node. The Operating System (OS) Level provides a standardised software base on top of which more complex services are constructed. The middleware level provides support for complex and dynamic operations, reducing or eliminating the application’s dependence on the specific hardware in use at any particular node. Section 2.2 discusses middleware solutions. The Application and Module level is the highest level in the system, consisting of the actual application code. Existing application-level systems are reviewed in Section 2.3.

2.1 Challenges of WSN Programming

Wireless Sensor Networks as a platform have their own particular challenges, related to resource limitations. Middleware and Operating Systems for WSNs suffers from the same limitations as any other application, and in addition, must
2.1 Challenges of WSN Programming

make tradeoffs due to the varying application space. The limited resources of a sensor node will not permit a middleware that is optimal for all tasks.

The challenges of a WSN middleware or programming system are listed in the following sections [20; 21; 22].

2.1.1 Mobility Model and Latency

This must be robust in a dynamic network topology. Dynamics can be slow or fast depending on application. Node failures are expected in the long term and must be managed in some form. Self-configuration and self-maintenance are critical in a mobile environment. The network should manage itself as much as possible and recover from transient failures. Nodes should recover from catastrophic failure to a known good state (such as with no applications running).

The mobility support of the underlying execution environment can be either weak and strong. A strong mobility model allows an agent to be migrated at any time during its execution, without any intervention from the agent itself. A weak
2.1 Challenges of WSN Programming

model only allows mobility at certain controlled points in the flow of execution. Generally, the agent requests that it move itself.

Non-agent systems can be considered a degenerate case of a mobility model – the model transfers executable code only. As the execution context of the code is not involved, it can not be considered weak or strong.

The mobility model can be local or remote. A local mobility model can only move an agent from a node to one of its direct neighbours, while a remote mobility model can move an agent to any other reachable node in the network, over multiple hops if required (subject to the constraints of the system).

These attributes are not directly quantifiable, however an analysis of any middleware system will reveal which category they fall into.

The limiting factor in agent mobility is the time taken to move from one node to another. This movement latency limits how quickly the agent can follow physical events of interest, and how quickly it can react to changes in the environment. Movement latency is the primary limiting factor in any mobile software system's dynamic performance. This is not a factor for static code execution systems, although if they are ever to be moved they must be treated as a dynamic system.

Movement latency involves a number of factors. All of the overhead of moving and beginning execution at the target device are included in this measure – the physical transfer of the data required, any error-checking, retransmissions, acknowledgements, protocol overhead and writing the executable code into non-volatile memory at the destination. If the destination is not an immediate neighbour of the source, then the path to take must be determined.

The size of an agent is a factor in the latency – the larger an agent is, the more data that must be transferred. Movement latency can be measured by timing the movement of an agent under various conditions.

The movement latency is an important factor in the power consumption, as the radio must be powered on continuously during the agent transfer. As this is one of the most power-hungry states the wireless sensor node can be in, this time must be minimised as much as possible.
2.1 Challenges of WSN Programming

2.1.2 Execution Model

The Execution Model should substantially support application development. Typically a tradeoff between abstraction, expressiveness and efficiency must be made. As the application executes on the middleware, this will define the user's interaction with the network (in addition to the underlying OS). It should be capable of dynamically adapting to varying resource availability to improve overall system performance. The restricted hardware resources of a wireless sensor network can be most efficiently utilised by making such application adaptation lightweight and efficient.

Code can either execute directly on the target device, or through some sort of interpretation layer. Directly executing code has clear benefits in terms of power and efficiency, however as it has unfiltered access to the target hardware, it must be carefully written and tested. Interpreted code is slower and the instruction set is much simpler, but it can guarantee correctness and reliability due to its simple execution model.

The time taken for the agent to carry out a particular operation limits how quickly it can respond to an event. The limits placed on the agent by the execution model determine what complexity of operation is possible. It may not be possible to represent complex operations within the limit of hundreds of simple instructions, in a stack-based execution environment of virtual machine environments such as Agilla or WiseMAN[17; 23].

There are two main features of the Execution Model the must be considered – the number and type of operations that are permitted, and the speed at which they execute. The speed can be directly measured by timing certain standardised operations, while the range of operation possible is established by elimination, where analysis of alternative systems shows that certain operations are not possible. A limited execution model does not allow complex and self-managing behaviour, and severely restricts the scope of application that can be created.
2.1 Challenges of WSN Programming

2.1.3 Security

Whether or not security is required depends on the application requirements – some do not require any security, some are not feasible without it. Wireless sensor nodes are simple devices, and the mathematical operations required for modern cryptographic operations place a substantial burden on them. If security is included in a system, it should be possible to disable it to eliminate the overhead introduced, if desired.

Security can be classified at a number of levels. An insecure system implicitly trusts any agent data it receives over the radio. A signature-based system trusts only agent data that has been digitally signed using a secret key known only to authorised users. This does not prevent overhearing of agents and may be vulnerable to replay attacks\textsuperscript{[24]}. An encrypted system uses a secret key to encrypt all communication. This prevents an external user from overhearing or injecting any transmission into the network.

The level of overhead increases with each level. A fully encrypted system requires encryption and decryption of every packet transmitted. This overhead can be measured by the additional time the microcontroller spends carrying out security operations, which results in additional energy consumption.

2.1.4 Reliability

Reliability measures how resilient the system is to failures. Wireless sensor nodes are cheap, battery-powered devices and are vulnerable to physical failure. The unreliable and highly time-varying nature of the low-power radio channel can also introduce potential failures. The system should manage and work around these node failures.

Reliability can be offered at two levels – at the middleware level and at the application level. At the middleware level, the system must be able to provide guarantees about the success or failure of any transfer operation. An agent should either move completely, or not at all. It should not be possible to end up lost or duplicated.

Once this guarantee is provided by the middleware, the application can then provide the next level of reliability, assuming the execution model is powerful
2.1 Challenges of WSN Programming

enough to allow it. A properly designed application should expect node and communication failures as a matter of course, and work around them.

Reliability can be quantified in a given scenario by measuring the average number of nodes actively participating in the system over a given timescale, under stated conditions (such as packet loss or rate of node failures).

Self-stabilisation and Superstabilisation are important characteristics for reliable operation, as discussed in Section 2.4.3. This guarantees that the application will continue to operate, under certain specified conditions.

2.1.5 Energy Consumption

Energy Consumption is a critical factor in WSN systems. General-purpose wireless sensor nodes contain an energy source of up to a few thousand mAh. Once this is depleted, the sensor node will no longer function. Energy scavenging systems may be available, but realistic systems would need to be many times larger than the sensor node itself to achieve power outputs greater than a few $\mu W$.

Energy consumption is dominated by two factors – radio communication and active processing. The radio transceiver of a wireless sensor node operates at a very low output power (0-10dBm), and the transceiver itself dominates the energy consumption, not the radiated RF energy (i.e. they operate at extremely low output power efficiencies). The processing requirements required to perform the convolution operation needed to detect and decode a spread-spectrum signal such as that used by IEEE 802.15.4[19] are much greater than to transmit it, so receiving a transmission consumes as much or more energy as transmitting[25].

Idle listening will dominate the radio energy consumption of a wireless sensor node, and the amount of idle listening that occurs is a function of the MAC layer. The MAC layer can optimise idle listening during periods of lower activity, but the radio must be almost continuously active during a transmission, as large amounts of data must be transferred in a short time. The lower the movement latency therefore, the lower the energy consumption.

The processor is the other factor that determines energy consumption. As long as the processor is active, it is consuming energy. Modern WSN operating systems break up the application and OS into a number of event handlers, with
longer running tasks allowed to execute when no event handler is scheduled. The processor will then sleep when no task or event handler is active. The amount of time the processor is active is directly proportional to the amount of energy consumed, and the amount of time the processor is active is related to the execution model.

Energy consumption is measured by counting the number of packets transmitted and received, and measuring the amount of time the microcontroller spends in active operation. This is highly dependent on the application and protocol choices made, and most analysis is done at some level in terms of energy consumption. As an approximation, the number of packets transmitted can be used to estimate overall energy consumption as radio communication is the most energy-intensive process on a wireless sensor node.

Energy consumption is linked to almost all of the other criteria, and many of the other criteria are inter-linked, so they can not be evaluated entirely in isolation. Complete-system evaluation in realistic applications is required to accurately view the overall system.

### 2.2 Programming Models for WSNs

The growth in WSNs is driven by the development of flexible and powerful software systems to support application development. Since the release of TinyOS 0.51 in 2001, WSN operating systems have evolved from simple hardware interface libraries with a task scheduler to complex hardware abstraction libraries, dynamic linkers and fully-featured networking stacks. All of the WSN operating systems discussed are released under Open Source licenses and borrow liberally from each other when possible (particularly low-level hardware interface code).

The operating system is the second level in Figure 2.1, interfacing with the Hardware Level below it, and the Middleware and Application Levels above. Middleware is the third layer in Figure 2.1. While it has some knowledge of the underlying hardware, it depends on the operating system to isolate it from the lower-level details of the hardware implementation. Hardware specific knowledge is used for efficiency rather than as a necessary part of the middleware functionality.
2.2 Programming Models for WSNs

The programming model of the WSN application determines how a user interacts with the network, and how different pieces of software in the network interact. The programming model determines what type of software can be written and determines the form that this software must take. The programming model naturally constrains the system user, and proper choice of model is critical in the design of an application.

The middleware layer presents a comprehensive, flexible and portable service layer on which to build complex, dynamic applications. In many ways, this is closer to the general view of an operating system than those systems discussed previously. Existing programming models fall into a number of categories. Figure 2.2 shows a broad taxonomy of existing WSN middleware platforms, drawn from a number of studies[20; 21; 22; 26]. At the highest level, the programming model can be divided into four categories:

- **Functional Programming** executes the application on each node of the network directly. The application itself is distinct from the OS and Middleware. Middleware services operate independently on each node in the network. The application itself may be interpreted or may execute natively.

- **Data Gathering** systems fix the role of the sensor network as an information-gathering (and sometimes, in-network processing) tool. They do not support complex, mobile applications and tend to be limited in their application scope once deployed.

- **Declarative** systems allow the user to *Declare* that the application should take a certain form. The middleware is then entirely responsible for generating the appropriate code to perform the required functionality. Declarative middlewares isolate the user entirely from the underlying hardware, presenting the entire wireless sensor network as a single virtual platform.

- **Publish / Subscribe** middleware limits the role of the wireless sensor network to acting as a data gathering tool. Nodes advertise the availability of any information they possess, using an attribute-based naming scheme. An interested user then registers an interest in this information, forming a data gathering link across the network. This operates at a lower level than a
2.2 Programming Models for WSNs

Figure 2.2: Classification of existing middleware

database approach, and the subscriber and publisher can be nodes internal to the network.

2.2.1 Data Gathering

A data-centric network serves simply as a data gathering tool. It is simple a network of sensors without wires, optimised to efficiently and quickly present data to an external user. As they serve only this single purpose, they can do this efficiently. Since the user is dealing with data, rather than nodes themselves, some sort of attribute naming scheme is used. The network topology must either be fixed and known in advance, or self-managing, since the user is only concerned with data and its context, not which node address the data has come from.

While the abstraction provided by such systems is useful, they essentially act
as an even more limited virtual machine where the only possible operations are data gathering and aggregation. Applications with complex data processing requirements will inevitably have to resort to centralised data-gathering and offline processing, or application-specific modifications to the system.

TinyDB\cite{27} is the best-known example of such a system. Data queries are flooded from a base station to the network, and returned using a spanning tree generated by the query. TinyDB presents the network as a database to the user. The set of nodes is a virtual database table and SQL-like queries are used to extract values from this table. This is translated into a query that is sent out to the network using a controlled flood. Any node that has relevant data returns it along the spanning tree that has been generated by the query.

Readings can be combined or aggregated at intermediate nodes according to the query. By default, TinyDB only supports simple aggregation. This can be extended by modifying the TinyDB application, however this application-specific version of the TinyDB application must then be redistributed to every node in the network using some other distribution service. Combinations of values may be used to minimise the transmission of unnecessary data.

A sample TinyDB query is shown in Figure 2.3.

SINA\cite{28} is a more sophisticated data gathering middleware, that includes hierarchical clustering and an attribute-based naming scheme. This improves scalability and efficiency in large networks.

This attribute naming scheme abstracts the physical network even more – for example a user can query for “temperature” with no knowledge of which physical sensor is taking the reading; in fact if the query involves more than one response they could even come from different sensor types. Clearly this requires the sensor nodes to translate their readings into a standard form.

DSWare\cite{29} provides event detection in a distributed system. It also supports distributed decision making and reliable in-network storage to reduce commu-
2.2 Programming Models for WSNs

nication overhead. Readings and the resulting decisions can be read out of the network at a later stage. Data is also cached to improve performance. An SQL-like syntax is used to specify which events should be monitored and stored.

Data gathering middlewares do not have the flexibility of operation for agent-like behaviour. They are inherently centralised and rely on an external program to generate queries and to manage the network.

2.2.2 Declarative

Declarative systems present a more abstract view of the sensor network. Note that while a portion of the middleware executes on the wireless sensor node, most of the substantial processing and management operations take place on a connected, higher-specification device, typically a laptop or a PDA. The aim of a declarative system is to allow the programmer to focus on program outcomes rather than implementation, writing fewer lines of codes, yet with the program still matching the designers intuition on program behavior.

MagnetOS[30] is such a declarative system, although it requires a PDA-scale device. It provides a single system image of a unified Java virtual machine to applications over an ad hoc collection of heterogeneous nodes. The application is automatically and transparently partitioned into components that are dynamically allocated to particular nodes. Energy consumption is reduced by intelligent placement of system components. MagnetOS consists of a static code partitioning service and a local runtime that supports the partitioned applications.

Declarative Sensor Network[31] is a declarative system implemented on TinyOS. This system consists of a programming language called Snlog, a compiler that translates the Snlog to nesC program using components from a set of compiler library generic templates, and a runtime system that is compiled together with the translated Snlog program into a single binary image.

In TinySOA[32], each node exposes its capabilities in the form of a service profile, consisting of a set of services such as sensing and actuation capabilities that it provides, and the quality of service (QoS) parameters associated with those services, such as delay, accuracy or freshness. An application then specifies a generic requirement that is met from the specific deployment according to the...
advertised service profiles. Cross-layer optimization is used wherever possible to improve service performance. Service description at the node level is limited to basic sensing and communication services.

The generated application is limited to the form that can be described by the declarative system. The application scope is limited by what the service description can express, and the resulting applications are simple and generic.

2.2.3 Publish/Subscribe

Publish / Subscribe systems are also designed for data gathering but in a more context-sensitive and distributed way than simple data gathering. Nodes advertise the availability of any information they possess, using an attribute-based naming scheme (This is the Publish operation). An interested user then registers an interest in this information (known as the Subscribe operation), forming a data gathering link across the network. This operates at a lower level than a database approach, and the subscriber and publisher can be nodes internal to the network.

MiRES[33] is a simple attribute-based Publish / Subscribe middleware. It is built on top of TinyOS 1.0, and allows a user to query and extract data from the network. It attempts to optimise multiple queries, exploiting knowledge of periodic queries and overlapping queries, as well as simple in-network processing to reduce data traffic.

TinyCOPS[34] is a more sophisticated Publish / Subscribe middleware based on TinyOS 2.0. Subscribers express their interest in events by injecting subscriptions into the system containing constraints on the properties of the events. Publishers post notification messages to the system based on an attribute naming scheme, and when a notification matches the constraints of a registered subscription, it is delivered to the subscriber.

Metadata is also attached to the query. This contains requirements for the publishers and run-time control information, such as repeated samples or accuracy constraints. The metadata patterns are not binding and a publisher may choose to optimise the request, for example by combining similar subscriptions. The framework is flexible and allows the user a choice of underlying methods and algorithms.
2.2 Programming Models for WSNs

Such systems are limited to data gathering and management, and require external management. The system must be specifically set up for a single purpose, and is limited in the scope of response it can make to detected events.

2.2.4 Functional Programming

Functional Programming supports the execution of an application. This helps the user to program the network, and may allow the user to create more complex applications than would otherwise be possible, however the user requires some specialised knowledge of the middleware platform used, and some awareness of the underlying hardware. The application executes on top of the application middleware, and is aware of the underlying topology and system configuration.

2.2.4.1 Native Stateless

Native stateless systems are the simplest – these function simply as code distribution mechanisms. This is generally provided as part of the basic sensor network operating system. TinyOS[7], SOS[8] and Contiki[9] all include mechanisms to push code updates to a running sensor network.

This update may take the form of a complete system update, in which the node is reset to enable the new image, or a modular approach, where only part of the image is updated. SOS and Contiki are both modular systems and can update part of the operating image without affecting others. TinyOS, including extensions such as FlexCUP[35], does not support this – even if only part of the operating image is modified, the node must be reset and will lose all operating state at this point. A mobile agent system must use a middleware system that supports dynamic code update without resetting the entire operating image.

The programming model of a native system is determined by the underlying operating system.

TinyOS 1.0[7] uses a specially developed language, nesC, to compile the program into a static image. nesC is based on C/C++, with the addition of component and interface specifications. These are superficially similar to C++ classes, and the learning curve is intended to be shallow for an experienced C programmer. It is an event driven operating system with binding interface definitions
used to statically link components together offline, as the image is being created. Efficiency is achieved by statically optimising the monolithic image.

Each component contains a list of interfaces supplied and provided. The interfaces are specified in an interface file, and impose constraints on both sides of the interface. Interfaces contain commands and events. A component providing an interface must implement each command and may signal events to the component using the interface. A component using an interface must provide a handler for each event in the interface, and may call any commands in the interface. This is the only mechanism by which components may communicate. Apart from scheduling and radio communication, operating system functionality is also provided through interfaces.

Interface specification requirements are imposed by the nesC compiler at compile-time. Unless all of the interface methods are correctly implemented, the compilation will fail.

TinyOS 2.0[36] introduces a large three-level hardware abstraction layer (HAL) to provide platform-independent services. This allows applications to be written with no regard for the platform on which they will eventually run, although the performance of an operation will vary enormously depending on whether it is executing directly or through an emulation layer. An application can, if desired, access services through the second or first level instead of the top level, however they then lose the cross-platform benefits of the HAL. The basic structure is the same as TinyOS 1.0, and nesC is still used.

Although TinyOS is the de-facto standard WSN operating system, its poor support for dynamic code makes it unsuitable for this work. TinyOS can not modify the executing program image without replacing large parts of it, as it is statically compiled and optimised. The node must also reset, losing all data in non-volatile storage.

FlexCUP[35] introduces dynamic linking to TinyOS 2.0. Components are compiled independently and stored with their symbol table. The components are then dynamically linked together at the sensor node, with inter-component references patched together by the dynamic linker. This approach considerably reduces the amount of data transmitted to carry out an update of one component.
however it does not preserve state across the update – the node is reset to enable the new image, losing all operating state.

Although dynamic linking is an essential part of a mobile agent system, the node reset makes it unsuitable for mobile agent systems. Resetting the node loses all of the information stored by existing agents on the node.

Contiki[9] supports dynamic loading of modules, however much of the kernel is still compiled into a large static image. The development of Contiki is focused on providing a large, highly capable networking stack. Thread-like constructs[37] aim to ease application development for developers unfamiliar with the state-machine approach generally employed in WSN programming – however a state machine is still used internally and these are not true threads.

Contiki’s dynamic loading approach is powerful and flexible, using Compact ELF (CELF) to represent dynamically loadable modules. CELF is a reduced form of ELF, optimised for small, simple modules[15; 38]. The large networking stack consumes most of the available code and data memory, and does not leave enough space free for a mobile agent system. Contiki aims to control WSNs interactively, with an interactive command shell similar to a UNIX login shell and a sophisticated networking layer, quite different to the autonomous embedded systems proposed by mobile agent systems.

The proposed middleware system is suitable for implementation on Contiki, if either the memory footprint of Contiki could be reduced, or if a platform with more memory was available. The sophisticated Contiki networking stack could provide a useful basis for native mobile agent middleware if the resource limitations were solved.

SOS[8] is an operating system based on dynamic operation from the bottom up. It is a module-based system with a simple code distribution and update protocol. The SOS kernel provides a minimum of services – simple hardware interface, scheduling and MAC functions. All higher-layer functionality is provided by dynamically loadable modules (they may also be statically linked if desired, for simplicity and efficiency).

SOS uses message passing as its primary inter-module communication mechanism, as well as allowing modules to register function pointers that may be
2.2 Programming Models for WSNs

dynamically linked by other modules. SOS uses fixed memory locations to implement kernel functions (i.e. a jump table), and dynamic linking to link used and supplied functions between modules.

Modules are represented using Mini ELF (MELF). Position-independent code is generated as much as possible (using relative location references to reduce the amount of linking required). MELF is also a form of ELF, optimised for small, simple modules[8; 38]. When a module is started, it allocates a block of memory for the module state. A module can not have local variables apart from it’s state and any dynamic memory it allocates. Each module designates a function as a message handler. Messages are routed to the module through this function. Each module is self-contained, and its state is easily accessible and transferable.

2.2.4.2 Interpreted Stateless

A Virtual Machine (VM) may be used to execute the agent code. The VM interprets the code in the form of a simple instruction set. In general, VM-based systems allow simple algorithms to be represented efficiently, injected and distributed cheaply around the network. However they suffer from interpretation overhead during execution.

Dunkels et al.[15] showed that even for a simple application that can be easily implemented in a virtual machine (a vector convolution in this case) the VM was almost one hundred times slower executing than a native implementation. They also consume significant memory resources – Agilla is only able to support a maximum of four agents on any node at any one time.

The original WSN virtual machine was Maté[11]. Maté consisted of one specific VM, tailored for simple data gathering and processing applications. Code is divided into self-contained capsules of 24 single-byte instructions, which can be transferred in one packet, eliminating fragmentation and reassembly issues. Maté execution is stack-based. Instructions include Push Constant (6-bit maximum size), Add, Copy, Call Subroutine (each capsule is a subroutine). A Send Message instruction sends a value over the radio – a certain capsule is executed in response to this packet, if registered at the receiving node.
The network is deployed in an empty state, and agents are pushed out into the network by the user.

Although Maté introduced the virtual machine paradigm to WSNs, it is dated by modern standards. Capsules are limited to 24 instructions; larger programs must be split into multiple, interlinked capsules, increasing the complexity and introducing potential problems if consistent versioning is not maintained.

Maté evolved into ASVM[12]. ASVM defines a library for implementing virtual machines, allowing extension and application customisation of the byte codes used. The original Maté VM was implemented in ASVM as an example. Both of these permit mobile software agents, however the agent is still constrained by the relatively simple VM in which it executes.

ASVM extends the scope of the Maté VM instructions, however the resource constraints are still present. As discussed in the following section, the limitations of the execution environment restrict the scope of interpreted applications in embedded wireless sensor networks.

2.2.4.3 Interpreted Stateful

Agilla[1; 17] is a more sophisticated version of Maté. The network is deployed in an empty state, and agents are injected into it. Agents can move, copy and terminate themselves. Information is propagated among agents and one-hop neighbours using a tuple-space[2] . It is assumed that nodes have knowledge of their physical location and addresses are a function of the predetermined network topology. The Agilla VM is much more powerful than the Maté VM – containing a 150-byte stack, a 12-byte heap (essentially general purpose registers) and three fixed-purpose registers. Although powerful and flexible, the agents are still constrained by the virtual machine in which they operate, while the resources required by the virtual machine implementation itself limit each sensor node to four agents at a time. The total agent size, containing all of the agents in the system is limited to 440 bytes (simple instructions occupy one byte, some two or more).

The limitations of the execution environment restrict the scope of applications that can be generated. Inter-agent communication is only possible indirectly,
using the tuple-space. Agents can not create or monitor other agents, limiting the level of cooperation. The VM language is very simple, close to a typical assembly language, but with restrictions on the amount of storage. Agilla can only store information in a 12 element heap, or on the stack, if the data can be manipulated into a suitable form. The 440 instruction limit is quite restrictive, as the stack-based architecture requires multiple instructions to implement even the simplest operations.

A graphical representation of the Agilla VM architecture is shown in Figure 2.4.

WiseMAN[23] is a more recent development of an embedded mobile script-based agent interpreter. It suffers from similar problems to Agilla - the programming language is obscure and very limited in scope. The agent’s may define variables, operators and rules, and its operations are defined in terms of only these constructs. Detailed performance measurements are not available, however the authors suggest that overall execution and migration times are similar to or slightly worse than Agilla, while expanding the agent scope somewhat by allowing dynamic topologies and self-modification. Only one agent can be processed at a time, and agents are expected to either terminate or migrate to another node when they have completed. Agent rules are discarded as they are matched,
allowing an agent to shrink in size as it fulfils its design goals. Agent sizes are in the range of tens of bytes, with a maximum of 170 bytes.

WiseMAN provides a powerful environment for systems that can be described as a simple set of relational operators and rules, however it operates at an extremely abstract level. It is not capable of manipulating lower-level hardware or performing complex coordination and management tasks without substantial assistance from external software, and its agent size limitations, coupled with its inability to multi-task agents limit its flexibility.

In-Motes[16] is based on Agilla, but introduces a high-level architecture for agent management. This uses a set of behavioural rules to coordinate agent-node interaction. Nodes are assigned roles based on rules and adjust their agent mix appropriately. A tuple-space is used as before for local communication, based on the Agilla equivalent.

The performance of this system was tested on Mica2 motes over a period of months[39]. The dynamics of the tested system were very slow – readings were measured with a period of minutes, and agents were injected even more slowly. The network required complete re-flashing five times over the course of the experiment due to node failures, and ten single nodes failed and were reset.

The agents (for the Virtual Machine) had sizes of 118 bytes and 68 bytes, even for this relatively simple distributed data gathering application that was studied. They rely on external network monitoring and resetting to guarantee reliability. The experiment does however confirm the usefulness of the mobile agent approach in WSNs, and emphasise that node failures should be expected and managed as part of the application design. Even a simple data-gathering application required almost half of the available storage space.

2.3 Application Level

The application level is the highest level of software in the system, as shown in Figure 2.1. The application level can interact with all of the levels below it, but communication is primarily with the middleware and operating system levels. The application layer is the point of contact of the user with the wireless sensor network and contains all of the functionality of the application.
The nature of the application level depends very much on the middleware and operating system underneath it. It can vary from a static application[40], to a virtual machine-based mobile agent[1], to a complete native mobile software agent. The application itself can be written in an attribute-based declarative language[30], virtual-machine byte codes[1], a C-like modular programming language[36], or in standard C[8]. The capability of the programming system determines the capability of the application itself.

2.3.1 Application Requirements

The application scope of WSNs has expanded considerably from its origins in simple data-gathering (Contrast [41] and [42]). The true power of WSNs is only realised by complex distributed processing and control applications.

In general, older application focused on static, externally-managed data-gathering and processing. The limited operating systems and software architectures available at the time did not allow complex and autonomous operation. These were focused almost entirely on data gathering. These included the habitat monitoring study carried out by UC Berkeley[40] in 2002, as well as monitoring a forest habitat[43].

Simon et al.[44] demonstrated how a distributed wireless sensor network could be used to discover the location of and characterise the type of a sniper using measurements of the sound waves produced.

A static application image is the simplest and most efficient form of application, however it is inflexible and extremely costly to modify after deployment. Every wireless sensor network software system contains some portion of the operating image that is statically deployed to every node in the system. This may be a micro-kernel, or a more complex middleware system, allowing modules or agents to be created at the node.

Most wireless sensor network software systems that are not specifically designed for modular and dynamic operation construct an entirely static program image. This is the approach taken by TinyOS[7], FlexCUP[35], TinySOA[32], and any TinyOS-based system.
2.3 Application Level

A static program image’s biggest advantage is its efficiency. As the program call graph is known entirely in advance, the compiler can perform the maximum optimisation, leading to smaller and more efficient code (although the use of pointers and indirect function calls can limit the amount of optimisation possible). As the image is static, resource availability can be guaranteed and the system does not need to handle complex fail-safe situations. The application code is inextricably linked with the operating system, and one can not be modified without changing the other.

The disadvantage of a static application is its inability to respond quickly to changing circumstances. The image can only be changed by generating a completely new image and replacing the old program image. This requires a reset of the sensor node and will lose any stored application state. As the role of each node in the system is fixed by its program image, it is difficult to reliably work around node failures. Over time this has lead to the development of more dynamic and modular applications.

Modular applications[8] are the step between static images and dynamic agents. The application is composed of multiple self-contained software modules. Each module can communicate with other modules in the system, either by calling functions directly, or by passing messages to a fixed message handler. If direct function calls are used, the operating system must link each function call to the correct address, as this address can vary from system to system and is not known in advance at the time of the module compilation[15].

The system is capable of updating part of the operating image without destroying the operating state of the rest of the image. This allows the system to perform multiple tasks, modifying itself in response to changing conditions without interfering with existing operations at the sensor node.

Modular applications provide the system with the ability to reconfigure itself in response to changing operating conditions, however the change is to the locally operating image only – it is only a code distribution system. The resulting modules are fixed to the sensor node at which they start, and must communicate with modules on other nodes to achieve dynamic or distributed computing[8; 9]. Modular applications are an improvement on static systems as the system can now
update itself without reconfiguring the entire operating image, but the software is still tied to one hardware device once it has started executing.

Autonomous monitoring and controlling of dynamic and complex physical phenomena require complex and dynamic sensor network applications. Static applications are slow to respond and reconfigure themselves. Applications are limited by their execution environment, and the limited execution environment of an interpreter capable of running on an embedded wireless sensor node limits the complexity of the algorithms that can be represented in such a system. The application must be robust and reliable, and must be capable of dealing with failures in the sensor network. Failure should be expected and the system should continue to operate even after agent or node failures. This requires a flexible and powerful application platform. Nodes and applications or application modules must self-monitor and cooperate to achieve reliability guarantees. Mobile agent systems meet all of these requirements.

Fire detection and tracking is a common application of Mobile Software Agents. Fok et al.[1] describe how mobile agents are used to detect a fire in progress, and how a chain of agents is set up around the fire, allowing fire fighters to accurately determine the extent of the fire.

In a mobile agent environment, software is continuously moving around the system. This differs from the static and modular approach, where reconfiguration only occurs in occasional, user-initiated operations, and as the software transfers are system initiated, they require additional authentication. The system can no longer blindly trust and communication received. An application platform must be able to guarantee that an unauthorised user can not interfere with the application after deployment.

2.3.2 Application Performance

The various application-level approaches can be evaluated in terms of the application requirements discussed in Section 2.3.1.
2.3.2.1 Mobility Model and Latency

Tripathi et al. [45] describe two types of mobility – weak and strong. A strong mobility model allows an agent to be migrated at any time during its execution. A weak model only allows mobility at certain points – i.e., the agent requests that it move. They conclude that the benefits of a strong mobility model are generally outweighed by the considerable cost in complexity and that “program-controlled migration under weak mobility suffices for the majority of the applications”.

This can be related to multithreading systems in traditional computing – a strong mobility model is like a pre-emptive multithreading system that can interrupt a thread at almost any point, without any input from the thread itself. A weak mobility model is like a cooperative multithreading system in which a thread must relinquish the execution context explicitly.

Mobile software agent systems are the only approach that provides mobility, while dynamic operation is supported by modular systems. A static program image provides very little mobility or ability to operate dynamically.

2.3.2.2 Execution Model and Energy Consumption

Biswas et al. [46] provide a simulation study of mobile agent performance compared to a client/server data gathering model, in a target-classification application. They compare execution time and energy consumption in an ns-2 simulation. Although based on a somewhat unrealistic IEEE 802.11 MAC and with no resource limitations apart from network bandwidth, their general conclusions are useful.

They observe that as the size of the network increases, the execution time scales linearly for the mobile agent system, while it increases exponentially for the client-server system. Energy usage is 20%-30% lower for the mobile agent system, with the saving increasing as the network scales upwards. They vary the data size and verify the intuitive result that as the amount of data transferred increases, the mobile agent system becomes more efficient than the client-server system.

They conclude that when the network is small and the dataset of the application is small, simple client-server applications are the best choice. As the network
scales upwards, and the amount of data involved is increased, mobile agents will outperform client-server systems.

This suggests that mobile agents are a good choice for large, complex applications. Interpreted agents will struggle to represent complex algorithms, and any large-scale processing or storage will require native execution (whether static, modular or agent-based). A static program image will execute efficiently the algorithm that it is programmed to carry out.

2.3.2.3 Reliability

Self-monitoring and cooperation in multi-agent systems is not a new concept. Resnick[47] discusses how a large network of very simple cooperating agents can produce complex, emergent behaviour. He shows how simple adjustments in the application parameters can vary the response from stable, self-reinforcing behaviour to wildly unstable. Bonabeau et al.[48] describe how the concepts taken from cooperating agents in nature can be used in artificial systems. Starting with the natural behaviour of various insects, such as ants and wasps they describe how similar techniques can be applied to networks of simple, distributed, mobile agents, without any central control structure.

Dressler[49] describes techniques for self-organization in sensor networks. They show that more complex control structures are made up of smaller units employing a combination of positive and negative feedback, interactions among individuals and probabilistic techniques. They conclude that self-organizing systems should use local behaviour rules that achieve global behaviour, should exploit implicit coordination, should minimize long-lived state information and should adapt to changes.

Frank and Römer[10] discuss how wireless sensor nodes can be assigned roles in a network in a distributed manner, although the initial specification is still externally controlled. TinyCUBUS[50] provides a similar role-based code distribution protocol. In contrast with the proposed autonomous system, the role assignment is externally driven.

Monitoring and management are crucial – as stated by Han et al.[51] – "Wireless Sensor Networks will never succeed if they require constant maintenance
2.3 Application Level

from an entire IT department". Systems such as SNMS[52] and more recently, Hermes[53] provide some form of management and visibility in a deployed network, however compared to traditional wired networks, post-deployment network management in WSNs is immature.

Cooperation is more easily implemented in a dynamic agent system than in a static system configuration. Mobile agents can move around the network to perform tasks and are not tied to one node in the system. This means that the failure of a single node does not necessarily lead to the elimination of certain activities from the system. The proposed agent-based systems is inherently more capable of reconfiguring to work around problems or failures than existing systems due to its ability to produce more complex self-managing behaviour. While self-monitoring is achievable by any application system, agent-based systems are better able to respond once an issue has been discovered, and native mobile agent systems are best able to deal with complex, dynamic behaviour.

2.3.2.4 Security

Zhou et al.[54] define four requirements of WSN security – confidentiality, authenticity, integrity, and availability. Signatures are an appropriate approach for wireless sensor networks as the signature is only calculated once for each data transmission, and the data itself is not encrypted. This provides the benefits of authentication and integrity without imposing severe resource requirements.

Digital signatures are used to verify the integrity of a piece of data. The simplest form of digital signature is a Private Key scheme[55] in which a private key known to each party in the transaction is used, along with the message data, to generate a signature. The signature is transmitted along with the message data. The receiver performs the same operation, using the same secret key, and if the two signatures do not match, the message is discarded.

The algorithm used to generate the signature is critical[24]. It should be computationally infeasible to reverse the algorithm, as this would allow an eavesdropper to deduce the secret key. It should be infeasible to find two sets of input data that produce the same output (this is known as collision-resistance). Due
2.3 Application Level

to the finite key space, it is always possible to reverse the operation using a simple brute-force search, however with careful choice of algorithm, the computation required to do this is not practical.

MD5\[^{[56]}\] is an extremely popular algorithm for digital signatures. It is a member of a class known as cryptographic hash functions. It produces a 128-bit signature. While weaknesses have been discovered in MD5\[^{[57]}\], as in most hash algorithms, these are specialised methods capable of finding two inputs sets with the same output – this does not help an attacker attempting to discover the secret key. The internal state of MD5 is only 128 bits, making implementation possible on wireless sensor nodes.

MD5 is based on older algorithms, MD2\[^{[58]}\] and MD4\[^{[59]}\]. Although MD2 is specifically designed for implementation on 8-bit systems and MD4 is a simpler version of MD5, both of these algorithms have more serious security weaknesses\[^{[60]}\] that reduce the complexity of a key search considerably (from of the order of \(2^{128}\) to \(2^{100}\) or less).

SHA\[^{[61]}\] defines a more recent set of cryptographic hash functions, however their large internal state and increased complexity require considerably more storage space and processing ability than older functions, making their implementation difficult on a wireless sensor node. Passing and Dressler\[^{[62]}\] evaluated some hash and cryptographic functions on wireless sensor node-level hardware. 1024-byte blocks were hashed or encrypted and the time taken compared. MD5 took 42.7ms to hash 1024 bytes. SHA-1 128.5ms and encryption using AES took 1.67s. This suggests that only simpler hash functions (such as MD5) are realistic on embedded wireless sensor nodes, as the overhead of multiple seconds per agent would be far too great for a mobile agent system.

Secure Deluge\[^{[63]}\] and Sluice\[^{[64]}\] both use incremental hashing to verify each packet of a code update for a static program image. Their design model is based on an architecture in which large static code updates are deployed infrequently from a single source with access to more complex processing and storage requirements to multiple nodes in the network. This ensures that an attacker can not inject malicious code into the network. As code updates are very infrequent, they can tolerate an overheads of multiple seconds per update that would not be acceptable in a mobile agent system.
2.3 Application Level

Any application programming system can be inherently secure or insecure - the system must be designed to use an appropriate security system during deployments or updates. As mobile agent-based systems reconfigure the operating image more frequently, they will suffer more overhead from the introduction of security than a static system.

2.3.3 Existing Agent Systems for WSNs

Agilla has previously been discussed in Section 2.2.4.3. It is a virtual machine-based mobile agent system for embedded wireless sensor nodes, however it suffers from the overhead of interpretation and the resource limitations this imposes.

Impala[13] is a mobile agent system for PDA-scale wireless sensor nodes. It provides support for complex dynamic applications, with a sophisticated and powerful middleware. The Zebranet hardware platform[65] has 1.8MB of storage, a GPS receiver and a sophisticated microcontroller. It uses AgentTCL[66] and has a code size measured in megabytes - much too large to implement on a typical embedded wireless sensor node.

Impala adapts its application makeup according to measured parameters, reacting to a changing environment. Different application modules are selected based on the prevailing conditions, such as network connectivity, battery level or sensor readings. Application modules consist of a set of event handlers. An application updater keeps modules current and consistent across the network.

Modules do not move across sensor nodes, however sensor nodes do move around the physical environment. Impala is based on a wildlife tracking project (ZebraNet[65]). The premise is that sensor nodes will be implanted on tracking collars and placed on animals. As the animals move, communication links will appear and disappear. The mobile agent approach is well-suited to this as agents can operate autonomously while no communication link is available and then use a link when it does appear.

Sensorware[14] is a scripted mobile agent system that is also implemented in TCL[66]. Lightweight control scripts written in TCL are injected into the network. They can migrate, along with their state, to neighbouring nodes. The TCL scripting language is modified to provide most of the common mobile agent
operation such as migration and data gathering. The application has a code size of 180kB and memory requirements above that of an embedded wireless sensor node – the target platform is an iPAQ PDA with 16MB of code memory and an IEEE 802.11 radio.

SensorWare sits above the operating system, between it and the application. The OS is an embedded Linux distribution that provides a multi-threaded operating environment. The use of a scripting language provides compactness and abstraction. The node hardware is accessed through abstract interfaces, shielding the application programmer from the complexities of the underlying OS and hardware. Scripts can migrate to neighbouring nodes, leaving behind the original or moving with its state. The script is essentially a set of event handlers – each script executes in a separate thread, and a set of system threads manage hardware such as the radio, timers and sensing.

The SunSPOT node and its associated Java virtual machine provide a portable Java virtual machine, that allows migration of threads from one node to another. Again, this uses a device much more powerful, and power-hungry than an embedded wireless sensor node. However the flexibility and ease of programming of this system requires more expensive nodes with much greater energy consumption.

All of the existing mobile agent systems, apart from Agilla and its precursor virtual-machining based systems are far too large for implementation on an embedded wireless sensor node. In order for the power and flexibility of the mobile agent approach to transfer to embedded wireless sensor nodes, there is a need for a native mobile agent system, that is efficient enough to be implemented on an embedded wireless sensor node, while powerful enough to create the complex and self-managing behaviour that is shown by existing mobile agent systems for larger-scale platforms.

2.4 Middleware and Mobile Agents

A mobile software agent is a self-contained package of executable code and an execution context that is capable of moving between different nodes in the wireless sensor network.
The critical difference between a mobile agent and a simple modular system that supports dynamic code is that a mobile agent moves the agent state as well as the executable code when it is transferred from one node to another. Existing mobile agent systems for WSNs are either Virtual Machine-based, as discussed in Section 2.2.4.2, or are implemented on hardware platforms with much more resources than an embedded wireless sensor node. These include the HP iPAQ, based on an Intel X-Scale processor and other PDAs, with at least an order of magnitude more memory and storage than an embedded sensor node.

As WSNs are be deployed in hard-to-reach places, embedded into the fabric of devices, they can no longer be a single-application platform. When the sensor network is considered as part of the network infrastructure, and not as an application itself, it must allow users to inject applications into the network after deployment. The more flexible this support is, the more powerful the applications that can run on it.

Another approach to mobile agents, quite different to that taken by this work is to fix the agent on a particular Sensor Node but allow the nodes to move around. This approach is taken by Impala[13] (tracking wildlife with the ZebraNet project) and ElectricCow[67] (tracking herds of cattle), as it fits well with their application model, however this is only applicable where considerable node mobility can be guaranteed and is not generally applicable.

The contribution of this thesis builds on the existing work on mobile agents. It is distinct from VM-based approaches as the agent executes as a native module, with all of the power of the underlying operating system available to it if desired and is not constrained by the limitations of the particular virtual machine in which it operates. It executes on embedded wireless sensor nodes that can be miniaturised and embedded into physical infrastructure, operating at extremely low power levels.

2.4.1 Middleware

While there are a myriad of middleware and mobile agent platforms (see for example [68]), they are targeted at desktop-scale systems or at best, PDA-class
2.4 Middleware and Mobile Agents

devices. Java is widely used, for its cross-platform compatibility. Classical middleware platforms such as CORBA\cite{69} or J2EE\cite{70} are far too heavyweight for WSNs and different design approaches must be taken.

The SunSPOT\cite{18} as noted earlier, although described as a wireless sensor node, runs on a powerful ARM processor with megabytes of storage. This represents a considerable increase in processing and storage capacity over the devices considered in this thesis. This much processing and storage ability is required, even for the minimal implementation of the Java VM and libraries that it supports, suggesting that a fully-featured Java VM can not be implemented on an embedded wireless sensor node. While the additional capabilities of this platform would solve many of the resource constraint problems encountered in this thesis, this must be traded off against the power dissipation of such a device which is at least an order of magnitude greater than embedded devices such as a MicaZ or T-Mote\cite{18; 71; 72}.

The development of middleware and mobile code on less resource-constrained devices occupies a broader space as the middleware can be designed according to the design vision of the creator and not according to the limitations of the device. Resource-constrained designs still have a function in such systems as they will provide the most efficient solutions, however if resources are not a severe limiting factor this is likely to be a less important consideration in the design.

Even in the desktop-scale middleware environment, wireless environments are much less common than wired. Urra et al.\cite{73} observe that PDA-based platforms often create difficulties for implementing mobile agent systems, that bluetooth performs very poorly and that the variable latency of WiFi can cause issues with middleware platforms that implicitly assume that the network is wired.

Due to the shortcomings of existing dynamic programming systems for wireless sensor networks, and the large gap that exists between these and classical desktop-scale middleware, none of these existing solutions could be used on WSNs for mobile software agents. A middleware specifically targeting mobile agents for wireless sensor networks is required, and the development of such a middleware is described in this thesis.
2.4 Middleware and Mobile Agents

2.4.2 Characteristics

Jennings\cite{Jennings95} describes some of the characteristics of mobile software agents:

- They are clearly identifiable entities with well-defined boundaries and interfaces
- They are embedded in an environment over which they have partial control and observability
- They are autonomous
- They are capable of exhibiting flexible problem-solving behaviour in response to their environment

He argues that agent-based technologies, particularly multiple cooperating agents, present a natural and effective means of partitioning, analysing and controlling complex problems and systems.

Mobility and security are also important:

- A Mobile Software Agent is capable of moving from one physical device to another, carrying its state in some form that allows it to resume operation at the destination
- A Secure Agent system contains some authentication mechanism that does not allow agents to enter the system without the correct authorisation

Lange and Oshima\cite{Lange98} give seven reasons why mobile agents are used. Mobile Agents:

- Reduce the network load
- Overcome network latency
- Encapsulate protocols
- Execute asynchronously and autonomously
- Adapt dynamically
2.4 Middleware and Mobile Agents

- Are naturally heterogenous
- Are robust and fault-tolerant

The various features of mobile software agents are discussed in the following sections.

2.4.3 Self-Stabilisation

The concept of reliability as defined here is a particular case of a more general form of *Self-Stabilising* systems. A system may be in various states. A self-stabilising system can be formally defined by the following conditions\(^76\):

- A state is *legitimate* if starting from this state, the algorithm satisfies its specification
- Starting from an arbitrary state, it is guaranteed to converge to a legitimate state within a finite time
- Once in a legitimate state, it is guaranteed to only reach other legitimate states

The scope of the definitions used by Dijkstra and others\(^76; 77\) is quite narrow and relates to provably self-stabilising operations in synchronisation and resource arbitration in low-level hardware networks, specifically excluding hardware failures and topological changes from the algorithm. However the concept translates readily onto the area of reliable distributed software systems, where these are exactly the disturbances that can appear in the system.

In order for a system to be strictly self-stabilising, it must guarantee that it continue to satisfy its specification exactly, in the face of any disturbance. A less restrictive definition is that of a *Superstabilising* system\(^77\). A superstabilising system is self-stabilising in normal operation, but enters a less restrictive *Passage* state when faced with significant disruption. When in a *Passage* state, the system satisfies only a subset of its specification, and once the disruptive event has completed it will eventually return to a *legitimate* state.

Applying this to the reliable control of a wireless sensor network, the network is Self-stabilizing if:
2.4 Middleware and Mobile Agents

- The network is in a legitimate state when all of the nodes in the system are taking part in the control operation

- Starting from an arbitrary agent setup, it is guaranteed to reach a legitimate state

- Once in a legitimate state, it is guaranteed to remain in a legitimate state, unless it encounters node failures or topology changes occur

While the network is also Superstabilizing if:

- If node failures or topology changes occur, some nodes in the system may not take part in the control operation for a period of time

- Once the failures or changes have completed, the system will eventually return to a legitimate state

In terms of autonomously operating wireless sensor networks these are extremely important properties. The network designer must be confident that once the network has been set up, that it will continue to function normally, as long as the system does not exceed its normal operating parameters. Initial efforts at self-stabilisation have focused on algorithmic approaches to mathematically prove self-stabilisation and failure recovery\[78\]. In addition to this, middleware is required that is capable of implementing these algorithms reliably.

2.4.4 Existing Agent Systems

While the idea of using mobile software agents is not a new one, it is only recently that they have begun to be deployed in meaningful, commercial environments. Research in agent-based systems has been carried out for over 30 years\[79\], but only recently have computer networks been large enough, reliable enough and the attached devices been powerful enough to achieve the goals of earlier research systems. Many of the scripting languages used for agent research such as AgentTCL\[66\] and TeleScript\[80\] were developed over this period, while Java\[81\] and various extensions such as Aglets\[82\] and Jini\[83\] are more common in modern agent systems. The Foundation For Intelligent Physical Agents (FIPA) has
standardised a number of high-level specifications for heterogeneous and interacting agents and agent based systems[84].

Mobile Agents have found uses in the internet space, such as for traffic management in wireless networks[85], management of ad-hoc networks[86; 87], cache management for improving the internet browsing experience[88] and to customise the internet interface for mobile wireless users[89]. In these applications, the mobile agent represents the user of the system, commanding or negotiating with the internal technical service providers to improve the experience of the agent's owner.

High-level agent systems have been proposed[90] for optimising various mobile computing services, such as providing users with consistent working environments, no matter how they connect to a network. This is presented at a number of levels, ranging from allowing a user to detach from their working environment, complete with any running processes, and reattach to it from a different location, to dynamic resource binding to allow users and software programs access to resources from any location.

Telecommunications networks have deployed mobile agents to improve their network management systems[91]. Modern digital telecommunication networks consist of massive, distributed computing environments, that seem perfectly suited to mobile agent technologies. MAGENTA[92] uses mobile agents to decentralise network management functions and to allow them to continue even while parts of the network are disconnected. They conclude that there are clear advantages over centralised control systems. Mobile software agents have also been proposed to integrate various services and technologies in heterogeneous telecommunication networks[93] – typically the agent tries to fulfil certain service guarantees at the lowest possible cost to the user. Despite the involvement of such companies as BT and France Telecom in these projects, they have not had widespread deployment in a commercial environment.

2.5 Native Mobile Agents

Wireless Sensor Network Operating Systems are evolving from their original role as scheduling and hardware interface layers. A modern operating system is ex-
pected to provide application support services. This can be viewed as a middleware layer operating in between the operating system and the application. The expanding application space has led to an expansion in the services required of an operating system.

Chief among these is mobility. As long as the application is limited to one physical location, it will struggle to deal with truly mobile events, and may fail in the presence of discontinuous network connectivity. Existing mobile agent systems either severely limit the application by executing it in a Virtual Machine, or their resource requirements do not allow them to run on a resource-limited wireless sensor node.

Mobile agents based on natively executing mobile code offer performance improvements over virtual machine-based or scripted agents. The cost of this is increased code size. Every effort must be made to reduce the size of the mobile agents as this is where they will compare least favourable with alternative approaches.

Embedded sensor nodes as a device class are defined as low-power sensor nodes, operating at an average power dissipation of tens of mW, capable of surviving deployment periods of months to years without battery replenishment. These typically contain 8- or 16-bit microcontrollers operating in the low MHz, with kilobytes of RAM and tens of kilobytes of program memory. Existing mobile agent systems are either too large and complex for implementation on an embedded wireless sensor node, or in the case of the few existing agent systems, too restricted by their execution environment to allow complex, reliable and self-stabilising operation. Most are externally managed, creating a single point of failure and precluding the systems from autonomous operation.

Existing mobile agent systems for wireless sensor nodes therefore do not meet the emerging application requirements for complex behaviour, reliability, efficiency and distributed self-management. The requirement for complex operation of such systems under severe energy constraints is best satisfied by a natively executing software system, and the requirement for reliability, distributed and self-managing operation under the same energy restrictions is best met by mobile software agents.
2.5 Native Mobile Agents

The main contribution of this thesis is the development of a novel mobile agent architecture for embedded wireless sensor nodes, and the implementation and evaluation of this system. A weak mobility model is supported – an agent can move itself, and its operating state to any neighbouring sensor node, and resume execution at the destination node. Code is not executed in a virtual machine – the agents are native code written in standard C. This provides for the first time, a rich and powerful execution environment that is capable of autonomously executing complex control and management algorithms, even on resource-limited wireless sensor nodes.

The middleware system provides simple routing, reliable migration and remote module fetching, neighbour discovery and inter-agent communication. A tuple-space is used, as in Agilla, to provide decoupled inter-agent communication, while direct communication is also provided. This allows agents to communicate with each other, creating reliable networks of cooperating agents. This allows the agents to cooperate, and to execute complex algorithms, allowing the creation of reliable, self-managing, self-stabilising systems that were not possible with pre-existing software systems for wireless sensor networks.

This reflects one of the directions current research in WSNs is taking – towards the original ambient intelligence or “smart dust” role of the WSN[6]. The other, taken by systems such as LiteOS[94] and Contiki[9], is to move WSNs into the user space. They have developed application shells, visualisation tools and thread-like programming models – turning WSNs into a distributed extension of traditional interactive computing platforms.

Smart dust enables true ambient intelligence, where the network operates independently, with no input from people passing through the network. The network can potentially exist without even the knowledge of the people concerned, although this requires further development in miniaturisation. True autonomous operation requires complex self-monitoring and management, to ensure the network continues to operate, no matter what environmental conditions are encountered.

The “smart dust” concept as currently implemented is extended further by this work – instead of static software executing on smart dust, this thesis presents a novel system of “smart software”, which can now take complete advantage of
2.5 Native Mobile Agents

the distributed nature of a wireless sensor network, while intelligently working around its limitations and restrictions.
Chapter 3

Middleware Architecture

The previous chapter presented the state of the art in middleware and operating systems for Wireless Sensor Networks, with particular reference to mobile software agents. This chapter presents a detailed description of the system architecture of the middleware framework that is the primary contribution of this thesis.

Existing modular code systems capable of executing on an embedded wireless sensor node fall into one of two categories – mobile code that is interpreted in some form at the sensor node, or static code that executes directly, but is fixed to one node in the system.

A middleware framework is presented that supports mobile operation of native code modules, commonly called mobile agents. Each agent is a self-contained package of code and operating state that can move from one node in the network to another, executing directly at each node, without interpretation or translation.

The structure of the middleware framework is first described in Section 3.1. The middleware is composed of three semi-independent software managers – Operation, Mobility and Security.

There are significant challenges in implementing such a system. These and the solutions implemented are then described. They are categorised as follows:

- How to encapsulate an agent's code and state and efficiently transmit them across the network? This is accomplished by the Mobility manager, considered in Section 3.3. The method used to encapsulate the agent in a form
suitable for transmission is discussed, as well as how the agent is reconstituted at the destination, and the limitations of the approach used. This includes mechanism used to pull in services and to push agents around the network. The agent carrier system is introduced, which allows the agent to move according to defined rules.

- How to ensure the security and reliability of the system? The Security Manager is responsible for ensuring that only authorised agents are used in the system. The mechanism used to accomplish this is discussed in Section 3.4.

- How to reduce the size of mobile agents by taking common operations out of the agent and into the middleware? The Operation Manager assists local services, such as a tuple-based blackboard system, the neighbourhood support and the routing module. This is described in Section 3.2.

- What user-side tools can be designed to assist in programming the network? The network is deployed without any agents onboard. In order to inject agents, a user-side software tool is used. This agent server is also used to provide a service repository, guaranteeing availability of code providing any service. This tool is described in Section 3.5.

- What structure must an agent take? The programming model imposes some restrictions on the agent itself. This is discussed in Section 3.6.

Section 3.7 lists the programming techniques that have been established as best practice for the development of mobile agent systems in restricted execution environments, summarises and concludes the chapter.

### 3.1 System Architecture

This section provides an outline of the general system architecture. Various parts are described in more detail in the following sections. The fundamental premise of mobile agent systems is that a system composed of multiple, semi-independent parts, can achieve better results than a monolithic system, and the middleware
3.1 System Architecture

itself is no different[68]. The middleware is therefore composed of a number of sub-processes, referred to as "Managers", loosely tied together into a complete framework. This section describes the composition of the system in terms of these constituent parts and how they relate to each other.

3.1.1 Mobile Agent Middleware

Section 2.1 discussed the particular challenges and tradeoffs that must be made by middleware systems in a WSN. These are summarised as

- Mobility Model
- Movement Latency
- Execution Model
- Energy Consumption
- Security
- Reliability

In Section 2.2, mobile software agents have been identified as a suitable programming paradigm for complex, distributed and dynamic tasks. The primary function of the middleware must be to enable and support mobile software agents. Only once this has been achieved can additional functionality be implemented.

Restricted resources and energy consumption are factors in every WSN OS, application and middleware system. The middleware architecture described in this chapter is no different, and it must be aware of the resource constraints imposed by the platform on which it operates. A wireless sensor node such as the MicaZ[71] has 4kB of RAM available in total. This must be shared between the OS, middleware and all executing agents. The middleware must therefore make every effort to conserve RAM. The FLASH memory limit of 128kB is less restrictive, and does allow some flexibility – other resource limitations are much more likely to be reached before the FLASH memory.

A native mobile agent system is capable of more powerful and flexible operation than an interpreted or declarative system that is subject to the same
3.1 System Architecture

resource restrictions, as the application has almost complete access to the microcontroller, without the overhead of interpretation or translation. Any task that can be feasibly carried out on a wireless sensor node can be carried out by a native software agent. This includes communication with devices outside of the network, integration with external data providers and infrastructure, data collection and processing. The mobile agents can interact with the underlying OS as well as the middleware.

Security is an application decision – some applications absolutely require it, some do not. The hardware of the sensor node itself is assumed to be physically secure – additional security is therefore only required during communication. Efficient verification or encryption of every radio transmission is a difficult task that is beyond the scope of this work – only the security of mobile agent transmissions is considered. While the security operation is only applied to agent movement operations, it is sufficiently large and complex to merit its own manager – the Security Manager.

Mobility is the fundamental operation of a mobile software agent, and every other operation in the system depends on this. Without mobility, the system is simply a code distribution protocol. The Mobility Manager is therefore the most complex and most important component in the system.

In order to meet these challenges, a middleware must also be efficient. An Operations Manager is also included. This is not part of the core system functionality, and in fact the system would be capable of operating without it. However, it contains a number of processes to assist mobile agents in their operation, to reduce the size of agents and to optimise their execution.

The Mobile Agent System Architecture is shown in Figure 3.1. The three Managers are shown, and the interaction between the agents and the managers. A more detailed view of the Manager’s relations with each other is shown in Figure 3.2. Each agents execute as a normal OS code module, interacting with the middleware when they require the services it provides.

While the purpose of native mobile agent systems is to allow mobile code to execute directly on the sensor node, there is much common functionality that does not need to be duplicated across every agent. The operation manager contains functions to simplify the agent. It takes some common functionality out of the
3.1 System Architecture

![System Block Diagram](image)

**Figure 3.1: System Block Diagram**

![Detailed System Implementation](image)

**Figure 3.2: Detailed System Implementation**

agent and into the middleware. These include the management of neighbour lists, inter-module communication and the stopping and starting of other agents in the system. This concept is related to the Agilla implementation\[17\], which found it useful to add very specific opcodes to the interpreted language to carry out similar common tasks. While these could have been implemented using the existing language, it would require adding tens of instructions to every agent script that used them. If the operations are common, it is more efficient to implement them once in the middleware than repeatedly in agent scripts. General-purpose operating systems have also used shared libraries for this purpose for many years.
The mobility manager handles movement of agents. Agents can duplicate themselves locally or remotely or move to another node, with or without state. The manager can be used to move an agent over multiple hops, following a pre-defined move condition. It must encapsulate the agent code, together with its operating state, and send it reliably from one node to another. It must guarantee that the agent is always in a consistent state, not duplicated and not lost during transmissions.

The security manager ensures that only authorised agents are allowed to run on the system. A signature is used to sign each agent with a secret key, known only to the nodes in the system and to authorised users of the system.

### 3.2 Operation Manager

The operation manager contains functions to simplify the agent. It takes some common functionality out of the agent and into the middleware. Almost all of the operation manager’s functionality is not required for the operation of the middleware - apart from its core agent management role, it exists purely to improve the efficiency and performance of agent and the middleware. Agilla takes a somewhat similar approach by adding application-specific opcodes so that common operations are carried out by a specific instruction, rather than a set of instructions[1]. Any virtual-machine opcode apart from those required to make the language turing-complete and to enable basic sensing and communication falls into this category, as does almost any non-OS library in embedded software environments such as TinyOS 2.0[36] or Contiki[9].

The inclusion of certain functionality in the operations manager can be a somewhat application-specific decision. Subject to the resource restrictions imposed by the WSN hardware, there is no particular penalty to including functionality that is not used. However the system must strike a balance between this optimisation and moving the entire application into the middleware, in which case it simply becomes a static application. The operations included here can be extended in the future if appropriate functionality is identified.

The operations manager must therefore aim to take common operations, that are likely to be large or costly to implement directly in a mobile agent and take
3.2 Operation Manager

<table>
<thead>
<tr>
<th>Information</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID</td>
<td>Unique identifier for each running instance of each agent</td>
</tr>
<tr>
<td>CID</td>
<td>Identifier for each agent (code) type</td>
</tr>
<tr>
<td>Code Memory</td>
<td>Location of agent code</td>
</tr>
<tr>
<td>Options</td>
<td>Is the agent active, complete, an agent or a service</td>
</tr>
<tr>
<td>Code Length</td>
<td>Size of the agent code</td>
</tr>
</tbody>
</table>

Table 3.1: Stored Agent Information

them into the middleware. In the absence of application-specific information, these operations should be generic, and common to many agents. The mobility manager already manages common mobility operations, so the operations manager deals mostly with the remaining local and communication-based operations.

3.2.1 Agent List and Management

The middleware must keep track of which agents are running, as well as various configuration information associated with each one. The operations manager is the most appropriate place to do this, as it deals mostly with local operations. The minimum information that must be stored for each agent is shown in Table 3.1.

It was discovered that the RAM of a wireless sensor node is too small to store even a partial agent image. The agent must be stored in the ROM of the wireless sensor node, and all of the agent code operations deal with a ROM image. Even a partial agent will consume a large portion of the RAM in an embedded wireless sensor node.

Every agent that exists on the system has an entry in this table. If multiple instances of the agent are executing on the same sensor node, each has a separate entry – the CID will be the same but the PID will be different. Inactive agents also have an entry in the table storing their code memory location, CID and size so that they can be reactivated. As the agents are deleted they are removed from the table, as long as at least one entry per CID remains. If the available code memory is exhausted, the least recently used of these are released in order to free space.

The CID identifies only the agent code and not any particular instance of that agent, while the PID identifies a particular instance of an agent, locally unique.
3.2 Operation Manager

to each node.

Additional memory is used during an agent transfer, however this is temporary and is released when the transfer has completed. The dynamic OS is designed to deal with varying memory requirements, and memory exhaustion will simply lead to a managed failure of the transmission.

The OS stores per-module information for each executing agent, just as it does for any code module. This may include timers, module state and the code memory location of each module. This information is released when the module is deleted. The actual details of the active agents such as state location and state size are not included in this table – these are a function of the underlying operating system and storing this value in two locations could create synchronisation issues. The OS is queried to discover this information when required.

While the OS and middleware information could be combined into a single source, this would couple the middleware tightly to one particular OS. It would also lead to unnecessary complexity for code modules in the OS that are not mobile agents.

An agent may query the middleware to discover which other agents are currently executing in the system. Given a certain Code ID (CID), the middleware will report the PIDs of any agent with that CID (code type) currently executing in the system. The PID uniquely identifies each instance of an executing agent locally.

A novel code caching system was introduced into the Operation Manager. This caches the code image of previously encountered agents, allowing the middleware to provide the agent locally without transferring the code. This provides performance benefits, as shown in Chapter 5.

3.2.2 Inter-Agent Communication

An agent system that does not allow communication between agents restricts the systems it can create to single large agents that carry out an entire complex task independently. This is not efficient or reliable, as there is a single point of failure, and the entire agent must be moved around to perform every task in the network.
Agent communication is necessary to create autonomous networks of cooperating agents.

Cabri et al.[2] describe four forms of inter-agent communication, which can be categorised in two ways – in terms of temporal and spatial coupling. Temporal coupling requires the agents to carry out some action or be in some place at the same time, while Spatial coupling requires the agents to visit or access some physical node or place. Table 3.3 shows the various types of communication between two agents and how they relate to each other.

<table>
<thead>
<tr>
<th>Temporally Coupled</th>
<th>Spatially Coupled</th>
<th>Temporally Uncoupled</th>
<th>Spatially Uncoupled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>Messaging</td>
<td>Blackboard</td>
<td>Linda-Like</td>
</tr>
</tbody>
</table>

Figure 3.3: Agent Coordination (based on [2])

### 3.2.3 Communication Architecture

As both the *Messaging* and *Linda-Like* communication mechanisms require either dedicated communication brokers or complex attribute-based naming and discovery mechanisms, a combination of *Direct* and *Blackboard* communication was chosen. Direct communication provides an efficient and reliable way for agents to communicate when they are spatially coupled, while the blackboard system provides a simple method of temporally decoupled inter-module communication. It allows agents to deposit data for other agents that do not need to know which agent has created or used the information.

This is based on a tuple-space such as that used in Agilla (see [2; 17]). The Agilla tuple system is also accessible locally and to each node’s one-hop neighbours (read-only). Agilla allows tuples to be named using strings, although the string length is limited to three characters, which are encoded onto a 16-bit numerical value. Tuples can be queried using simple wildcard-based patterns, although
3.2 Operation Manager

again the implementation of this in the virtual-machine environment is limited. Reactions can be registered, triggering a piece of code whenever a certain tuple is written (locally).

SPIN[95] also allows attributes to be named, and queried for by name. This requires storing the name corresponding to each value, and algorithms to match the name, partially or completely. It also requires a standard naming scheme to prevent collisions in the namespace. A design decision was made that if sufficient coordination is available to decide offline on a standard scheme, sufficient coordination is available to translate these names into numeric values, with different value sets reserved for different applications.

This functionality was not duplicated in this work – while the simple execution environment of Agilla encourages the developer to build the entire application around the tuple-space and simple reactions to changes in the space, the native mobile agent system presented here is aimed at more complex, self-managing applications. A reaction can be effectively implemented using polling. The wildcard-based matching rules and more complex data types are appropriate in the data-centric programming model that Agilla encourages, but less so in the functional model best supported by a native agent system. Agilla uses the tuple-space as the primary inter-agent communication system, and does not support direct agent communication, while in the native agent system proposed here, this is one small part of a much wider communication space, of which direct, inter-agent communication is the most important part.

The tuple-space as a neighbourhood information space is improved by the addition of conditional reads. This allows an agent to extract certain data of interest from its network locality without enumerating each individual value in the process. This is consistent with the vision of the tuple-space as an information tool that informs the agents, rather than as the primary inter-agent communication tool.

The decision not to include any metadata in the request was taken due to the simplicity of the rest of the scheme – if the identifier values are standardised off-line then the data types should also be. In order to access particular data, an agent must have knowledge of the identifier used for it. If it has knowledge of this, it knows the data type and can handle it appropriately.
3.2 Operation Manager

3.2.3.1 Interface

As the tuple-space is not accessed through a string, a 16-bit ID is used. The agent is free to map any other data type onto this 16-bit space. A three-character string can easily be mapped onto this 16-bit value, allowing rudimentary attribute naming. The simple, numerical type allows fast and efficient access. The block read/write operation allows more complex data to be encoded if necessary. No metadata is stored with the value written – as an agent must have knowledge of the particular 16-bit value involved to access it, it presumably also knows how to decode the information. The operations possible on a node's local blackboard are:

- **Query** – Return true if a value exists, false if not
- **Read** – A 16-bit value is read, or 0xffff if this item does not exist
- **Write** – Write a 16-bit value, overwriting any existing value
- **Clear** – If a value exists, clear it

Local operations return immediately, while remote operations are split-phase. Remote operations are limited to reading:

- **Read** – Returns value and flag stating whether value exists
- **Broadcast Read** – Any sensor node at which the value exists will respond
- **Conditional Remote Read** – Read and return a certain response – see below
- **Ping** – Every node will respond

Blackboard values below 0x0100 are reserved for system applications. Values from 0xE000 to 0xFFFFD are reserved for memory block writes (described below). Values 0xFFFE and 0xFFFF are reserved.
3.2 Operation Manager

3.2.3.2 Block Writes

Block writes are only supported for local accesses. Instead of a 16-bit value, a pointer is written. This pointer must point to a block of memory allocated dynamically. The ownership of this memory block is taken over by the operations manager. When an agent attempts to read one of these values, the memory block is copied and returned to the reading agent. This allows values larger than 16 bits to be written. The reading agent is assumed to know the layout of the block — no metadata is stored.

This allows an agent to store large amounts of sensor data that are not efficient to move around the network, or that are not relevant outside of the physical environment of a single node. This includes sensing histories, calibration values or detailed logging information.

3.2.3.3 Direct Remote Read

Remote operations are split-phase — a request is sent, and a response is only given when a response is received from a remote node. Multiple reads may be outstanding at one time. Remote operations are always best-effort — communication is not guaranteed, however the middleware will always respond to the agent that initiates the read with a status report, even if no remote values are retrieved.

A direct read is targeted at a particular neighbouring sensor node. The node will respond, indicating if the value exists, including the value if it does. If the node does not respond within a certain time period, this is signalled to the requesting agent.

The requesting agent does not need to implement a timeout — the middleware is guaranteed to respond exactly once with either the result or an error condition. The middleware distinguishes between an unreachable node and a node with no relevant data values.

3.2.3.4 Neighbour Query

Neighbour Query provides a simple mechanism for an agent to query for all of a node's neighbours. Any sensor node receiving a neighbour query (or Ping) will
respond. This is used to build up or to refresh the neighbour list maintained at each sensor node.

### 3.2.3.5 Broadcast Remote Read

Broadcast reads are an extension of direct reads. Instead of being targeted at a particular node, any node with a relevant value will respond. These responses are sent back to the requesting agent. The agent will also respond when the time for remote responses has expired, whether any responses have been received or not.

The blackboard will therefore send a response for each value received, and then one more response when the timer has expired. The list of neighbouring nodes is not considered in this operation – the final response always comes when the timeout expires. The requesting agent does not need to use any form of timer or poll, the blackboard will always signal when the operation has completed, even if no values have been returned.

### 3.2.3.6 Conditional Remote Read

A conditional remote read is identical to a broadcast remote read, except only one value is reported back to the requesting agent. This is decided according to a condition in the request – currently two conditions are defined:

- **Greatest** – The largest reported value is returned (values are unsigned)
- **Least** – The smallest reported value is returned (values are unsigned)

As before, the blackboard will always respond, even if no value is returned. The sensor node from which the chosen value was sent is also reported to the requesting agent. The system is shown in Figure 3.4, and an example in Figure 3.5. The response comes directly from the middleware at the remote sensor node – no agent is involved.

Conditional remote reads allow the agent to determine certain attribute values in its locality without the overhead of contacting every node and taking a reading from each node. This simplifies the agent and allows it to devote its resources to using the data rather than gathering it.
3.2 Operation Manager

**Figure 3.4: Conditional Remote Read**

**Figure 3.5: Conditional Remote Read Example**

### 3.2.4 Neighbourhood Support

The manager keeps a list of neighbouring nodes that have recently transmitted packets. The entire list or portions of it may be accessed. Managing neighbour lists and selecting random neighbours are common operations for mobile agents and moving them to the middleware reduces the size of the agent.

This is in common with other agent systems for WSNs. Agilla added specific instructions to their virtual machine to deal with neighbour lists, although these are not as sophisticated as those described here. Agilla is limited to a simple neighbour list, updated automatically, that can be accessed by each agent. The system described here can also select a random neighbour, update the list.
3.2 Operation Manager

dynamically and can dynamically modify some of the neighbour list parameters.

A decision must be taken on how long to keep a neighbour in the list. Whenever a packet from the middleware is successfully received, the timestamp of the sending sensor node is updated. The list is checked periodically and any sensor node that has not been heard from within a certain limit is removed.

The value of this limit is completely network-dependent. In a very dynamic network, setting it too high may result in time being wasted attempting to communicate with sensor nodes that are no longer in the neighbourhood. In a low-traffic, low-mobility network, setting it too low may result in nodes being removed from the neighbourhood list when they are in fact still available. Adaptive or distributed schemes (such as [96]) may provide a good compromise, however they have their own cost in increased communication cost and complexity. The middleware allows this limit to be updated by any agent in the system with knowledge of the local network dynamics, however it does not adapt the limits automatically.

The simple scheme chosen is communication-neutral – it does not introduce any additional traffic into the network. With a suitable choice of timeout it provides a very simple neighbourhood estimation – there is no reason why a more complex scheme could not be implemented as an agent itself, if needed by a particular application.

3.2.5 Routing

Routing represents an entire field of networking research in itself (e.g. See [97]). In order to support some of the node mobility functions described in Section 3.3, the middleware must provide some form of simple routing. This routing is not used as an actual communication channel – it simply provides an indication of a good route an agent may follow between sensor nodes.

In a large distributed wireless sensor network, it is not feasible for a wireless sensor node to hold a complete routing table for all nodes with which it may wish to communicate, as may be possible in a simpler network. The approach taken in the internet is hierarchical – the network is split up into sub-networks, and a route to the particular sub-network is first found[97; 98]. This assumes that address assignment is possible according to this hierarchy, and that once formed,
3.2 Operation Manager

this hierarchy is relatively static. Neither of these assumptions are valid for a large mobile wireless sensor network.

A hierarchy may be formed geographically, however this assumes wireless sensor nodes have knowledge of their physical position. This requires expensive hardware or large communication overhead[99; 100].

Routing must therefore be on-demand and ad-hoc. There are a wide range of routing protocols satisfying these conditions, however the limited resources of a wireless sensor node lend themselves best to simple protocols with little storage requirements. A standard routing interface is provided, and any on-demand routing protocol can be linked into this. If for example, the node has knowledge of its position, a position-based routing scheme can be used, while a standard on-demand scheme can be substituted if this is not available. A reserved blackboard value is used to store the currently active routing module, allowing dynamic switching to be performed.

Many of what are considered routing protocols in WSNs rely heavily on cross-layer information to optimise routing according to application-specific parameters[101]. True, general-purpose routing protocols are generally restricted versions of classical routing approaches.

One such protocol is Ad-Hoc On-Demand Distance Vector[102] (AODV), and a sample implementation is provided using AODV. Routes are dynamically established when needed, and after the initial limited flood, only nodes involved in the route itself play any part. Each node need only store its own one-hop route information (“the shortest route to node 19 is through neighbouring node 5”).

The route are timestamped and expire after a certain time. The reverse route is also established (this assumes the links are symmetrical) – this saves a routing phase in many common scenarios, when an agent wishes to move from one node to another and back. Having an identical forward and reverse route also provides savings in agent transmission since the agent code will already be stored at each node from the forward movement, speeding up the reverse movement. Intermediate nodes involved in the route will also have the same forward and reverse routes established, so that typically only one routing phase is required for an agent move.
3.2.6 Agent Termination

An agent may terminate itself or any other agent that has been started by the middleware. The agent is sent a message telling it that it will be terminated, and it is then removed from the active process list. This assumes that all agents cooperate with each other – in an environment without hardware-based memory protection this is a necessary assumption, as agents can not be prevented from manipulating the memory state block of any other agent, or of the middleware itself. There is no separation between OS, middleware and agent memory spaces at the hardware level, and any piece of code can always write to any memory address.

3.3 Mobility Manager

The mobility manager is responsible for the primary operation of agent mobility. This is the fundamental distinguishing feature of the mobile agent middleware, and the performance of this system is critical to the overall system performance. The Mobility Manager must manage the transmission, encapsulating and transmitting the agent code and state. It must deal with any lost packets, congestion and resource limitations at either end of the transmission. As agent mobility is the primary limiting factor in the dynamic performance of the system, it must take every step possible to optimise transmission.

3.3.1 Mobility Model

The mobility model both limits and characterises the operation of the mobile agents in the system. Without a mobility model, the middleware is simply a code distribution protocol. The model can be classified in terms of the agent, in terms of the network and in terms of addressing. These characteristics determine how powerful the agent mobility is – a powerful mobility model moves much of the considerations of where the agent should move to out of the agent and into the middleware, simplifying the agent. A less well-featured mobility model requires the agent to explicitly determine and specify the details of any movement
3.3 Mobility Manager

operation, and may require the agent to stop and execute on every intermediate node before reaching the desired location.

3.3.1.1 Agent Mobility

Agent mobility can be \textit{weak} or \textit{strong}, depending on whether the agent can be moved in any state or only in certain restricted states or at certain times, usually with the cooperation of the agent itself.

In terms of the agent, the mobility model is one of \textit{weak} mobility. The agent can only be suspended and moved at pre-determined points in its execution path, and the agent must stop execution and be suspended during the move – it can not be moved arbitrarily or without the cooperation of the agent itself.

Strong mobility is extremely difficult to achieve in the limited resources of a sensor node. The scheduler in SOS (in common with all other sensor network operating systems) is not pre-emptive. Each task in the system must voluntarily yield the execution context. This fits well with the typical wireless sensor network application, which is a state machine, with a number of handlers executing a small piece of code in response to various signals and operations. Prolonged processing tasks which would require the interruption of an execution context should be broken up into sub-tasks and should yield (i.e. return to the operating system) internally.

As the operating system is unable to interrupt an executing context, the system can only guarantee a consistent operating state while there are no handlers executing. The mobility model used in the system is one of agent-driven mobility – an executing agent decides itself when it wishes to move, the move is not driven by external forces. For this reason, there is no significant gain in performance by using a strong mobility model, at the cost of reimplementing a much more complex preemptive scheduler which is not well-suited to the architecture of a wireless sensor node.

The mobility model may be \textit{local} if the agent can only move to immediate neighbours, or \textit{global} if it can move to any reachable node in the network.

In terms of the network, the mobility model is \textit{global}. An agent can move to any node that is reachable by the routing system. A local model is also available,
to move to a node’s one-hop neighbours. If the node knows that the destination is local, this eliminates a potential routing phase and may speed up the transmission (although the routing module will bypass most of the routing mechanism anyway if it has recently received a direct transmission from a targetted node).

Each sensor node can only be involved in one movement operation at a time. This means that for the duration of an agent move, the sending and receiving nodes cannot take part in any other transfer. Movement requests received locally at a node while it is occupied with another move are queued and processed in order once the move has completed. Remote requests are not acknowledged – the remote node is free to retry the request or to attempt a different move if desired.

For transfers where code caching can be exploited, with a time-scale of 50–200ms this is not a major issue. For large agent transfers, with code, this means that the node is unable to receive any agents for a number of seconds. The resources required to handle an agent transfer make it very difficult for a node to handle more than one request at a time. The agent must store all of the metadata associated with the transfer. It must receive, process and write to FLASH memory every packet received. It must track which packets have been received and which have been lost, sending negative acknowledgements to fill any holes in the packet list. It must carry out a security check once all of the packets have been received, and link and start up the new agent when it has completed.

In order to minimise the time spent in a transfer, the packets are sent as quickly as possible. The sending node attempts to send on average one packet every 80ms. As discussed in Section 5.7.3, the MAC layer on a typical wireless sensor platform is optimised for random-access packet transmissions, not data throughput, and as the middleware is designed to operate on any suitable wireless sensor network platform it does not modify the MAC layer. An ideal system could transmit data 3–4 times faster than this if an unloaded network could be guaranteed, but this rate would not scale to a random-access network with other applications generating data at the same time. It is likely that a cross-layer protocol capable of switching the MAC mode could achieve higher throughput, however that is outside the scope of this work. The manager must therefore work within the limitations imposed by the underlying operating system, and not monopolise the system resources.
3.3.2 Encapsulated Agent

The difference between a mobile agent and a normal code module is that a mobile agent carries its executing state as well as the binary code. In order to do this, the operating system must be capable of isolating the state of each executing code module. In a static OS, this is extremely difficult as there is no clear separation of OS and module code or memory.

A novel self-management policy for dynamic state was used to efficiently implement the state reconstruction. This allows the middleware to efficiently transfer static state while the agent self-manages more complicated dynamic state.

3.3.2.1 Agent State

The simplest case is an agent with no state. In this case, the system functions as a code distribution protocol. This is a degenerate case of an agent – it is more correctly described as a code module.

In order to produce a true mobile agent, the agent must carry some execution state. The simplest way to achieve this is to allow each agent to define a constant-sized state block. Different agents may have different sized state blocks, but the state size is fixed for each instance of the agent. This approach is easy to manage for the middleware – it allocates a state block for each agent instance, and this is the only state that must be moved with the agent.

While a fixed-size state block has advantages, it also limits the functionality of the agent considerably. The agent is unable to track "open-ended" systems, and must limit the size of any buffer or table. This limits its ability to dynamically discover information. As all data storage is statically sized, the agent must discard values once the static limit has been exceeded. It must also allocate maximum-sized storage even when this is not needed, unnecessarily increasing its resource usage.

The advantages of a fixed-size system are simplicity and a guarantee of resource availability – as the agent only ever accesses stack-based local variables and a single block of state memory, it can be encoded efficiently. The middleware can also guarantee that enough resources are available before accepting an agent transmission – as long as the state block can be allocated, the agent has no
3.3 Mobility Manager

more resource requirements. The middleware does not need to know the internal composition of the state block, only the size. As all memory accesses are to the same block, the code may better allow optimisation.

A fully dynamic system is harder for the middleware to track but allows much more flexibility in the agent. The agent can dynamically allocate tables and buffers, shrinking and growing them as required. This clearly imposes a burden on the middleware, as it must track each piece of allocated memory, and it must provide some mechanism to reconstruct this at the destination after an agent move. There is no fixed bound on the resource utilisation of any one agent, and the middleware cannot guarantee resource availability.

In order for the dynamic memory to be reconstructed at the destination, a memory map must also be transmitted. This specifies the address and size of each dynamic block. This is used at the destination to recreate the dynamic memory. The agent may specify that some memory is only useful at the local node. This prevents unnecessary transmission of data that is particular to the local sensor node.

A novel memory allocation approach was developed for the native mobile agent system. Each agent specifies a fixed-size state block (which may be zero-sized if not needed) and may also allocate dynamic memory. The fixed-size state block is transferred with the agent, and any dynamic memory that has not been marked as local-only, this provides the agent with maximum flexibility – simple agents that do not need dynamic memory need not use it, while complex, dynamic agents have complete freedom to allocate any required data structure.

3.3.2.2 Agent Encapsulation

Agents are therefore made up of up to three parts – Executable Code, Static Module State and Dynamic Memory, as shown in Figure 3.7. The module state and dynamic memory make up the agent state – the executable code must also be transmitted if it is not already available on the destination node.

All of these parts may or may not be present. Module State is transmitted for an agent move or copy and consists of the dynamically allocated per-module
state allocated by the operating system. Dynamic Memory consists of memory that has been allocated dynamically by the module (See Section 3.6).

An example of a module state block is shown in Figure 3.6. In this example, the module has two blocks of memory allocated – a block of size 6 at address A000, and block of size 12 at address B640. This is converted to a single structure complete with a memory map identifying the two dynamic blocks along with their size.

Memory state tracking allows the manager to only send the relevant portion of the dynamic memory. This reduces overhead and unnecessary data traffic. All of the module’s static state is always sent. While it would be possible to specify that parts of the static state should be sent and parts not (at least in simple cases – such as sending the first 10 bytes and not the rest), this would require an extremely intrusive knowledge of the internal layout of the state block. The overhead of specifying which information is present and which must be initialised would likely exceed the saving produced.

### 3.3.3 State Reconstruction

When a mobile agent reaches its destination, the state must be reconstructed, allowing it to continue operation at the new sensor node. In general, the agent can not assume that the same memory addresses are available for all memory blocks. The agent must translate any memory addresses from the old addresses that are only valid at the old sensor node to addresses appropriate to the new node.

For the static memory block this is simple. The static memory block is always accessed through a pointer. This pointer is passed to every message handler during the agent’s operation, so no translation is necessary. The middleware receives a memory map with the old address and size of each dynamic memory block, and this is used to reallocate the actual dynamic memory blocks, keeping track of the old address of each block.

The middleware could attempt to carry out this remapping automatically, however this is extremely difficult to achieve. The agent would either need to specify which addresses in the static memory block contain dynamic pointers, or
3.3 Mobility Manager

Rely on the middleware matching likely addresses based on their value. The C language does not provide detailed information about data types (in the way that Java, for example, would), so this information would need to be specifically provided at compile time. Although this could be automated by a script to extract pointer values from the data definition, it would still considerably complicate the compilation process. It would be extremely difficult to properly manage compound data types, including structures, unions and the various alignment rules that the compiler and hardware platform may impose. Linked lists, null pointers and pointers to pointers would be difficult to reassemble correctly without detailed knowledge of how they are to be used.

Instead of attempting to decode complex data structures, the remapping could be carried out dynamically. This would require storing the dynamic memory
map permanently, and every pointer access would be filtered through a mapping function. Even with hand-optimisation, this would add considerably to the time taken for every pointer access – the RISC architecture of a typical microcontroller does not have the complex indirect memory accessing modes common on desktop-scale processors, and complex memory accesses must be explicitly computed. This overhead is imposed every time the agent performs a dynamic memory access and will cause a significant performance penalty – essentially the agent always uses the memory space of the first node at which each block is allocated, and maps dynamically to the current node’s address space. The memory map must also be held in memory for the duration of the agent’s lifetime.

A novel self-mapping strategy was developed to provide the most efficient memory translation. Each agent must specifically remap each memory block that it uses. The agent calls a function to remap from the address on the old node to the address on the new node, for every dynamic memory block. As the agent knows the layout and inter-dependencies of any data structure it has allocated, it can walk through these, mapping each block as it goes. For a typical agent with a small number of dynamic memory blocks, this translates into a small number of instructions. The size of the code required to remap the agent’s dynamic memory (consisting of a few static function calls) is close to the size required to describe the memory layout in a format sufficiently generic to include all possible data structures. The code is only transmitted the first time an agent reaches each node, and is then stored at the node.
3.3 Mobility Manager

The advantages of this approach are that the memory map can be released once the remapping has completed. The agent, which has knowledge of the exact memory layout of the state, carried out the remapping itself, and any combination of data structures can be supported. No extra information, apart from the memory map must be transmitted, and the size of the code required is similar to the amount of information that would be required to describe the memory layout correctly. Any combination of data structures can be supported.

3.3.4 Agent Push

Agent mobility can take a number of forms. As discussed in Section 2.5, weak mobility is the best choice for embedded systems and WSNs. The mobility manager manages an agent's movement from one node to another. This section considers the simplest case, a move from one node to a neighbouring node.

The agent orders its own transfer, so it is assumed to be in a state ready for transfer. As the agent itself it responsible for the transfer, the middleware need not take any action to prepare it for transmission. The middleware must encapsulate the agent for transmission, as described in Section 3.3.2.2. The middleware must keep the agent in a consistent agent after encapsulating it for transmission.

The agent is not deleted when the transmission commences - the transmission may fail, and the middleware must then resume the agent. If the agent continues executing while the transmission is in progress, it may modify its own state. If the transmission succeeds, any operations the agent has carried out will be discarded as the state image that is sent to the destination is that at the time of the start of the transmission. If the transmission fails, there are then two different state images which must be reconciled. As the most common case is for the agent transmission to complete successfully, allowing the agent's operations to continue during the transmission would result agents frequently executing “in limbo” where the results of the operation will eventually be discarded. Any changes made to the hardware configuration or any communication carried out will not be recorded and the system may not be in a consistent state.

The agent is therefore suspended during the transmission. This provides a clear, distinct thread of agent execution across the various sensor nodes that it
3.3 Mobility Manager

visits. The agent is only ever executing at one node at a time, and the agent state is always resumed in exactly the configuration in which it left the previous node.

While the agent is suspended, it may still receive messages. While the middleware may keep these messages, they are unlikely to be useful to the agent. Either the transmission succeeds, and the messages are discarded, or the transmission fails and the agent will have to deal with a large number of old messages that may no longer be relevant to it. These will relate to the time during which the agent was suspended. Any messages received while the agent is suspended are therefore discarded. This has the same result as if the agent transmission had completed immediately anyway.

The agent specifies whether it wishes to copy or move itself. In the case of a move, it is immediately suspended, and will not receive any further messages. The module state and any dynamic memory owned by that module are identified and copied for transmission (the original must be kept for a move in case the transfer fails - on successfully completing, the original is released). In the case of a copy, it is only suspended for long enough to duplicate the state and dynamic memory (to guarantee that the state is consistent). Since the mobility model is weak, the agent knows that it is being moved and so should prepare its state appropriately.

A negotiation takes place between the two nodes involved in the transmission. This exchange of information includes the size of the agent, the agent’s CID ad whether or not the executable code has been previously cached at the destination. If the destination accepts the transmission, the transfer of data commences. The transfer protocol is simple, to allow its efficient implementation on a sensor node. The executable code is transmitted followed by the module state, the memory map and the dynamic memory blocks. Any of these sections may be omitted if not required.

Operating system overhead, error-checking and physical-layer framing add a substantial amount of data to each packet, so it is more efficient to use larger packets. Hardware limitations do not allow packets greater than 128 bytes[25], and powers of two are preferred as they simplify considerably the management of the received data. The largest available power of two is therefore 64 bytes. Each
3.3 Mobility Manager

packet therefore carries 64 bytes of information; the final packet is padded out to complete the 64 bytes if needed, as shown in Figure 3.7. This improves efficient at all layers – both the security algorithm (see Section 3.4) and the program memory interface (see Section 4.1.3.1) typically operate on even powers of two.

Selective-ARQ with NACKS is used to manage this transfer. On timeout at the receiving end, or if the final data packet is received while there are still outstanding packets, a selective-ARQ NACK response is sent, listing which packets are missing at the destination. The sender then resends the missing packets. This continues until the process completes. Since the size of an agent is generally limited to 10-20 packets, a sliding window is not used. SR-ARQ gives better performance than simple ARQ in a lossy link, without overly complex resource requirements at the receiver. Stann and Heidemann [103] concluded that a combination of selective-ARQ (MAC) and NACK-based transport provide the best solution for reliable transport in an unreliable wireless network (although based on an IEEE 802.11 MAC).

Intermediate nodes do not need to store a complete agent image, allowing them to use RAM to store the partial image. This is the only exception to the general principle that agents are stored in ROM, as a particular intermediate node may not ever have a complete agent image.

A flow chart of the sending side is shown in Figure 3.8, the receiving side in Figure 3.9.

A close is sent from each end to end the connection, and the service is started at the destination node. While this protocol is described in terms of multihop communication, it serves equally well over a single hop. In this case, there are no intermediate nodes and all of the protocol packets are sent directly from one endpoint to the other. The various timeouts involved are reduced when the protocol is functioning over a single hop to allow more efficient operation when there is no additional latency being introduced by intermediate nodes. As the communication layers on the embedded sensor node are very thin, this is easily achieved. The ARQ protocol does not need complete knowledge of the transport details, only whether the communication is one-hop or multi-hop.

By default, the middleware attempts each stage of the move a fixed number of times, before giving up. The number of attempts is a tunable parameter.
defaulting to five times. As long as reasonable responses are received from the other endpoint of the transmission, it will continue — the transmission will give up when 5 attempts have been made to send a particular piece of data, where a response is expected, and none has been received. The agent may specify that the agent should keep trying the transmission, even if this limit is exceeded. In this case, the middleware will continue to attempt to send data indefinitely. This should be used with care — if a node moves or fails during a transfer, the sending node will be trapped in the sending loop. A better solution is usually to increase the number of retries.

Figure 3.8: Transfer Sending Side

Figure 3.9: Transfer Receiving Side
The mechanism used to ensure reliability is shown in Figure 3.10. The sending node will continue to repeat Finish messages until a FinishACK is received. The receiver will acknowledge a Finish message as long as the previous transfer completed successfully (otherwise if the last FinishACK was lost, the system would be in an inconsistent state).

![Figure 3.10: Transfer Endpoints](image)

### 3.3.5 Service Pull

Self-initiated transfers are needed for mobile agent-like operations, however there are situations in which a node needs to request a particular agent. A generic agent of this type is created – not a specific instance of the agent. This contains the default initial operating state of the agent. This is called a *Service* to distinguish it from a mobile agent with encapsulated state.

The middleware must identify which service is desired. This identification can be carried out using a unique identifier (the approach originally taken by Active Messages[104]), or via an indirect service description, such as that used by UPnP[105], Jini[83] and SLP[106]. If a service description is used, the middleware
must also have a description of each service provided, and a method of matching between the desired service specification and the provided service specification. This requires matching generic capabilities across different modules, and in all but the most simple cases will require a translation layer between the generic interface and the actual code module. All of the systems mentioned above are implemented on much more powerful devices than an embedded wireless sensor node.

The aim of wireless sensor network programming software from the beginning has been to avoid such methods for accessing services. In TinyOS[7] and TinyOS 2.0[36] this is achieved by compiling a (possibly generic) interface description into a static function call, while SOS[8] and Contiki[15] patch up function calls to statically defined program locations to avoid dynamic call generation. The benefits of WSNs are only realised by efficient code, and attempting to dynamically provide a generic interface to specific and complex code will not be efficient[4]. In an environment where even the biggest agent contains only a few hundred bytes of code, a suitably complex description language will require a substantial portion of the agent’s size just to express its functionality, in addition to the code required to execute operations through this interface.

Efficient WSNs require tightly coupled software agents and services. The potential benefits of providing a comprehensive service description and discovery system are negated by the fact that the agent requires detailed knowledge of the service implementation in order to usefully interact with it. For this reason it was decided not to implement any service description or discovery. In this case, there is no benefit in identifying services by anything other than their unique Code ID (CID), identifying the executable code module. Even purpose-built sensor network service discovery systems such as TinySOA[32] provide only very basic service description and discovery – the node enumerates available sensor hardware, and higher-level constructs are built around this outside the sensor network.

A node initiates a Service Pull. The service is identified only by its Code ID (CID). The service request is flooded out through the network, up to a limited number of hops from the initiator. Any sensor node willing and able to provide the service responds along the reverse route. While the routing module could be
used to establish a path, this would require two requests – one to establish which node can provide the service and another to establish a route to this node. As the return path is available in the route request anyway, it is more efficient to simply use this path than to establish a new one.

The requesting sensor node then chooses one of the responses (the lowest number of hops in the current implementation) and sends a request to this node, along the path established by the service request and response. The sending node then begins sending data packets, until the transfer has completed.

Once the transfer has started, the sending node sends packets regularly. The transfer protocol described in the previous section is also used, using SR-ARQ with NACKs to ensure a complete and reliable transmission.

### 3.3.6 Broadcast Push

Broadcast Push is a combination of a Service request and an Agent Push. Instead of pushing to one sensor node, it is pushed to the entire network. This is equivalent to the standard module distribution mechanism of the OS. It serves to propagate module updates through the network. This is not mobile agent behaviour, however it is still a useful operation, and is easily included in the middleware. With middleware support for this, any existing module distribution code can be eliminated completely from the kernel, saving space and memory. The process is shown in Figure 3.11.

- This is based on the Trickle algorithm\cite{107}, however the mechanism for avoiding broadcast flooding is slightly different. Trickle suppresses advertisements where an equivalent advertisement is overheard, while the Broadcast Push system always advertises, but uses a random time delay in the advertisements to ensure that the advertisements do not collide. As each node will respond to the first advertisement it receives, nodes in a broadcast area will tend to respond to the same advertisement. This provides a faster completion time than Trickle as all nodes advertise new data, all of the time, at the expense of higher packet rates under high packet loss. Trickle is intended for infrequent updates to large code images, so it favours efficiency over completion time, whereas the Broadcast Push
3.3 Mobility Manager

The Mobility Manager system is intended to inject small code images into the system frequently. It is therefore optimised to deliver the data as quickly as possible.

The same transfer protocol is used as for all other transfers - the only difference is that only NACKs are sent – there is no acknowledgement of a successful transfer. A sensor node wishing to perform a Broadcast Push sends out a push request (essentially an advertisement). If any sensor node responds to this request, it begins sending, using the same protocol described in Section 3.3.4. It sends the entire agent code, and then waits for a NACK. If none is received, it finishes. If one or more NACKs are received, each will contain a list of packets not received. These are combined to make a list of all of the packets not received by any neighbouring node, and any packet that has not been received by at least one sensor node is resent. It then waits for a NACK again. This continues until eventually all neighbouring nodes have received the module correctly. In the presence of a very poor radio link this procedure may result in a large amount of NACKs being generated, however any protocol will struggle under these conditions.

The system is simplified by using almost the same transfer protocol for every type of transfer. Even the largest software agent is only made up of 10 to 15 packets, ruling out any complex transfer protocol as the number of extra protocol packets would be greater than the amount of data packets. Point-to-point transfers are not used, as this does not exploit the benefits of broadcast overhearing, allowing a single transmission to send data to every node within reception range. In a densely deployed network this will provide substantial benefits in terms of the time taken to deploy an agent to every node and the power needed[107].

When a node receives a broadcast NACK while it is not sending a Broadcast Push, but for which it does have the relevant code, it will send the first unreceived packet from the NACK, without going to a sending state. This ensures that the transmission will eventually succeed, even if the originally sending sensor node is unable to complete it, as long as some sensor node within range has the relevant data. This is simpler than restarting a new dedicated transfer, just to fill in a few lost packets. The system is designed to minimise the overhead of large-scale agent transfers. As only a single packet is sent, there is no overhead due to session establishment, however packets are effectively being acknowledged one at a time.
and multiple sources may respond. This is a last-resort guarantee that the system will complete eventually and is not intended to be the primary transfer operation.

On completion, the receiving node then sends a Broadcast Push request and the process continues until all nodes in the network have the correct information.

There is no guarantee in this scheme that all nodes will receive the module eventually. Periodic advertisements could be used (such as the viral propagation used in Trickle[107]). This imposes a continuous traffic penalty, even when no agents are actually being moved. The advertisement could be repeated a fixed number of times, to increase the probability of reception, without actually guaranteeing it. As the number of retransmissions increases, the probability approaches one. In practice, it was found that in a typical medium-density network with a modern radio interface, a single advertisement, with a small random delay (so that two neighbours receiving the same transmission do not transmit the advertisement at the same time) provides a high probability of success. This could be adapted based on the network topology - more retransmissions may be required in a sparsely-connected network than in a dense network. Section 5.7.2 demonstrates the system working to push an agent out to every node in a network.

A stateful transfer could be used to speed up the filling of gaps in the received packet data, however this would then create a point-to-point link and remove the benefit of overhearing the agent data. In practice, as a number of neighbouring nodes generally have the relevant data, gaps are quickly filled.

Figure 3.11: Broadcast Push
3.3 Mobility Manager

3.3.7 Carrier Move

Carrier move is a mechanism to allow agents to move to a point of interest in a network without having to stop at every node along the way. Intermediate nodes will still receive a copy of the agent, however they will not execute it unless it has reached the end of the carrier path. The code is stored at the intermediate nodes for potential execution in the future. Each intermediate node must have enough space to store the agent code, even if it is not executed – there is generally not enough RAM at a sensor node to temporarily store the node image. As with a direct agent transfer, if the code storage space is almost exhausted, the least recently used agent images are deleted to create space.

This is a general principle that was developed for agent transfer – it is more efficient for the middleware to determine the target of an agent move, and to manage the transmission than for the agent to explicitly manage each hop of the movement, as intermediate nodes do not need to execute the agent or even to have a complete agent image.

Instead of signalling a move to a particular node number, a condition is defined for the move, based on certain node attributes. The condition consists of two parts, a move condition and a termination condition. Attribute-based actions are used in existing Publish-Subscribe systems such as TinyCOPS[34] and TinySOA[32], as well as data-centric systems such as TinyDB[27], which allow the selection of data based on specified attributes. Code mobility based on attributes is not supported, only data selection. Agilla[17] allows multihop movement based on the Node ID only, which is also supported by the Carrier Move system. The use of generic attributes for code mobility control in agent systems for wireless sensor networks is a novel application, although based on concepts that have been previously presented in research on multi-agent systems[47; 48] in larger scale networks.

This mechanism takes the logic of following move conditions out of the agent and into the middleware. This reduces the size of the agent, and speeds up the operation, as the agent is not executed at intermediate sensor nodes. This is only possible when the desired behaviour can be specified as a simple condition. If the movement condition is too complex to be represented, the agent must execute
at each node and explicitly decide where to move next. For example, if sensor readings are used to make the decision, or if a combination of factors must be considered. The carrier move conditions are not intended to take the place of complex agent decisions.

Addressing can either be node-based, where the node address is the only method of identifying a node, or it can be attribute-based where the nodes are identified using some other configurable attribute. In a node-based addressing scheme, the agent must first determine which node ID it wishes to move to, and then carry out the move, while an attribute-based scheme allows the agent to simply specify a movement condition and allow the middleware to find out which node matches this condition.

The mobility model implemented is a hybrid approach. Moves can be specified as purely node-address based, however the agent carrier system also allows conditional movement. In conjunction with the blackboard system, which allows nodes to configure publicly viewable attributes, this allows simple attribute-based movement. Only attributes within the local neighbourhood can be used – attributes are not viewable across the entire network.

A set of simple move conditions must be included, on top of which more complex conditions can be built, with the partial intervention of the agent itself. It is not beneficial to include very complex move conditions, as these are very application-specific (unless it is known in advance that such applications will be common).

An alternative approach, used in some of the evaluation applications discussed in Chapter 5 is to send a small “pilot” agent to find the appropriate path to the destination, using Carrier Moves where possible and executing directly to make decisions where not, and then to summon a larger and more powerful agent to the destination when reached. This avoids the overhead of moving and executing the larger agent at each node while searching for the destination. This fits in with the general principle that networks of small single-purpose cooperating agents will be more flexible and powerful than large multi-purpose agents[108].
3.3.7.1 Move Conditions

Complex movement conditions are not included in the standard middleware as these are likely to be highly application-specific – they can easily be added if it is known in advance that a certain operation will be common. The mobility manager includes only generic conditions that are likely to be used by many agents. This includes four conditions:

- Move towards – use a routing module to move towards a target node
- Move along gradient – use the blackboard (see Section 3.2.2) remote read interface to move either up or down a gradient of a particular blackboard value – it will move to the neighbour reporting either the highest or lowest value.
- Move towards sink – a special case of routing – move towards the closest node identifying itself as a gateway or sink.
- Random Walk – move to a random neighbour

Node ID, the blackboard system and the presence of external connectivity are the only generic attributes that are available to the middleware. Every node has an ID, and this is the primary method of identifying nodes. Every node has a blackboard, and each blackboard contains whatever attribute values have been written into it, while every node that is externally connected also has knowledge of this fact.

It is assumed that the nodes in the system are randomly deployed – i.e. that there is no pattern to the Node ID distribution. The Node ID therefore only serves to identify the node, and its numerical value is not significant. In a system where the identifier is only considered as a token, with no numerical significance, the only relevant operation is to locate a particular token (and move towards it).

Connectivity is similar – nodes identify themselves as either connected or not, so the only operation an agent can take is to move towards a connected node.

For the blackboard, a single numerical value is available at every node in the neighbourhood locality, so only a simple comparison operation is possible. The agent can either move up or down the gradient of an attribute.
Random walk is a special case in that no attribute information is used at all—a random neighbour is selected. This is a common agent operation and can be added to the carrier move system at almost no cost.

### 3.3.7.2 Termination Conditions

In addition to specifying how the agent should move, the condition must specify when the agent should stop moving and begin execution. These are simple conditions, related to the movement conditions. Five carrier termination conditions are currently defined:

- **Execute at node** – execute when a certain node ID is reached
- **Execute at peak** – when the top or bottom of a gradient is reached, execute
- **Execute at value** – execute when a non-zero value of a particular blackboard value is reached
- **Execute at sink** – execute at a gateway or sink node
- **Execute now** – make one move along the carrier path and then execute

As before, these use only the generic attribute information that is available to every node in the system—The Node ID, the blackboard system and the presence of external connectivity. The termination conditions are closely related to the movement conditions, and generally specify when the condition specified in the movement description has been achieved.

*Execute Now* is the only exception—this makes one carrier move and then executes. This allows an agent to follow a carrier-based path but to still execute at every node along the path.

### 3.3.7.3 Examples

Two simple examples of carrier moves are shown in Figure 3.12 and Figure 3.13.

Figure 3.12 shows a simple move. An agent at Node 0 issues a carrier move, with the conditions “Move to 11” and “Execute at 11”. The agent is suspended and its state captured pending transmission. The routing module is used to find
a route, and once it has moved successfully from Node 0, the suspended agent is terminated permanently at Node 0. The agent passes through Node 1 and Node 6. The agent code is cached for future use at these intermediate nodes but not executed. On reaching Node 11, the agent is started, with the state that has transferred from Node 0.

![3.3 Mobility Manager](image)

Figure 3.12: Carrier Routing Move

Figure 3.13 shows a gradient move. This assumes that the blackboard has already been setup to contain an attribute, numbered 240, with a numerical value at each node. The conditions are Move up Gradient of Blackboard Attribute Number 240 and Execute at Peak. The diagram shows the Node ID with the value of this attribute shown below each Node. As before, the agent is suspended at Node 12. The blackboard remote read operation is used to find the values of this attribute at the neighbours of node 12. The value 7 is the largest encountered, at Node 8, so the agent moves to this node. As before, the agent code is cached but not executed. This process is repeated at each node along the route followed, moving up the gradient to Node 5 and then Node 6. At Node 6, the values read from its neighbours (17, 20, 23, 12, 24, 6, 2, 3) are all less than the value at Node
6 (25). The termination condition has therefore been satisfied, and the agent is started with the state that has been transferred from Node 12.

3.3.7.4 Default Behaviour

When an agent signals a carrier move, the mobility manager takes account of the move condition and decides where the agent should be moved to next. On receiving a carrier moved agent, the manager first determines if the termination condition has been met. If so, the agent is executed on this node. If not, the move condition is examined and the next destination for the agent is established.

If the node initiating the move cannot determine where to move next, the move will fail. An agent should always monitor the result of a transfer. If a node receiving a carrier move cannot determine where to move next, the agent is executed at the current node (which may tell it to continue trying).
3.4 Security Manager

3.3.8 Code Caching

The executable code of a native mobile agent is typically much larger than the operating state. The middleware assumes that the executable code of the agent associated with a particular CID does not change during the lifetime of the system (if a new version of an agent is introduced it must use a different CID). Transferring the executable code every time an agent moves represents a substantial overhead, and the system avoids this as much as possible.

A cache is maintained at each node of the executable code of every agent that has previously visited that node. This code must be written into the permanent storage of the node anyway in order to begin execution, so there is no cost involved in leaving it there afterwards. As long as there is enough storage available, these code images are retained indefinitely. If a new agent arrives, and there is not sufficient storage, the least recently used code images are discarded, until enough storage has been released to allow the new agent to execute.

This provides substantial performance benefits – reducing the size of subsequent agent transfers substantially. Section 5.7.3.1 demonstrates reductions in total packets transferred due to code caching of 74–83\% for a multihop transfer, and in time taken of 78–83\%.

The middleware can be instructed to flush this cache, forcing all new transfers to transfer fresh executable code, or for debugging purposes.

3.4 Security Manager

The security manager ensures that only authorised agents are allowed to run on the system. Section 2.3.2.4 lists the four possible characteristics of WSN security: confidentiality, authenticity, integrity and availability.

Availability is almost impossible to guarantee in a wireless medium without significant hardware support. Saturating the wireless medium will make communication impossible and there is nothing that can be done using standard hardware apart from switching radio channels in the hope that another channel will have less interference.
Confidentiality is computationally intensive to achieve. In order to prevent overhearing of the transfer process, every packet must be encrypted. Encryption is a difficult task for a wireless sensor node, and mobile agents are intended to move around frequently, consuming a lot of the available resources on the sensor node.

Authenticity and integrity are therefore targeted. This is achieved by appending a digital signature to each agent transmission. A block-based hash is used in conjunction with a secret key known only to the nodes in the system and to authorized users to create the digital signature. This does introduce significant overhead to the system and it can be disabled if not needed. Each node chooses whether or not to enable security – an unsecured node will accept any transfer, signed or not, while a secured node will only accept a correctly signed transfer.

The process of signing a block of data is well-established[24]; however there are alternatives in how it is implemented. Secure Deluge[63] and Sluice[64] use per-packet signatures, however this introduces a significant overhead in each packet. The advantage is that an invalid transmission is detected at the start of the transmission and not with the last packet. This is more significant for large transmissions (these systems are targeted at whole-system updates, including the operating system) than for the small single-agent transmissions that are prevalent in this work. Per-transmission signatures are more appropriate for the frequent transmissions encountered by mobile agents.

3.4.1 MD5

A hash algorithm is required to generate the digital signature. The MD family of functions are designed for implementation on less sophisticated hardware. MD2[58] and MD4[59] are both more easily implemented on a wireless sensor node, MD2 being specifically designed for implementation on 8-bit systems, however both have serious security vulnerabilities[60; 109]. The MD5 algorithm is a more advanced block-based hash function. It takes an arbitrary number of fixed-size input blocks and transforms them into a hash value. In the case of MD5, the input blocks are 512 bits (64 bytes) in size and the output hash is 128 bits in size. The hash function should not be invertable – given a certain hash
it should not be possible to determine what input generated that hash, and it should be collision-resistant – given a certain input (and output) to the function it should be infeasible to find another input that gives the same output. SHA[61] defines a newer hash function, however it is more complex and there is no evidence that it is more secure, and it is considerably more complex than the MD functions, while modern elliptic-curve cryptography and public-key cryptography such as RSA[110] or AES[111] – designed for implementation on desktop-scale processors, are far too complex for implementation on a wireless sensor node – public-key cryptographic operations can take real-time seconds to complete even on a desktop-scale computer, and substantial storage is needed to store a set of keys and to carry out encryption operations.

Clearly, a brute-force search will eventually yield collisions, however in theory this should be the only way. Research has shown some weaknesses in the scheme, however these are mostly theoretical. It is possible to find two input blocks that produce the same output block, but in this application that will not aid an attacker attempting to deduce the secret key. Although this could potentially allow an attacker to inject malicious code, it is highly unlikely that an essentially random block would be in the correct loadable MELF format.

MD5 is the algorithm chosen for use in this implementation, as MD5 is a standard, powerful hashing algorithm that is widely used, but that is simple enough to be implemented on an embedded wireless sensor node. Any block-based hash capable of operating on 512-bit blocks can be easily substituted, as long as it can be implemented efficiently on sensor node hardware. The MD5 function itself requires 128 bits of temporary state during the calculation. The code to carry out the algorithm is reasonably large and it does introduce transfer overhead – this is discussed in Chapter 5.

### 3.4.2 Secret Key

Central to the signature scheme is a secret key, known to each sensor node and to authorised users and not known to anybody else. This key is assumed to have been placed into each sensor node at the time of deployment. The key is randomly generated, and is 512 bits in length. Without this key, it is impossible
A single key, common to the whole system is used. While multiple keys would allow multi-level and multi-user authentication, in practice any software agent, once started, has complete access to the entire sensor node and there is little practical benefit from distinguishing between them. Once an agent has managed to get into the system it would then have access to any keys currently on the wireless sensor node. Each key occupies 64 bytes (512 bits) of storage. While the operating system can take steps to make it difficult for an agent to read data out of the code memory, the lack of memory protection in a typical embedded microcontroller means that this can not be guaranteed without hardware support.
3.4.3 Signing Process

The signing process is simple – the agent is first split into 64-byte blocks (in practice it is already split by the agent manager for transmission). The MD5 function is first initialised. The first block sent through is the secret key. This is then followed by the agent information, as transmitted over the radio. This includes the agent code, agent state and dynamic memory. This is shown in Figure 3.14.

The signature is regenerated for every transmission – this is necessary since the agent state may have changed since the agent was received and this would change the signature.

Putting the secret key at the start of the process forces an attacker to go through the complete MD5 transformation in an attempt to discover the key – if the key was at the end, the attacker could pre-calculate most of the state and would only need to carry out the transformation for the last block.

3.4.4 Receiver

The receiver also goes through the same process. The packets must be processed in the correct order, even if not received in this order. At the end of the transmission, the signature generated at the receiver is compared with the transmitted signature – if these do not match, the agent is discarded and the transmission fails.

The digital signature does not prevent an attacker from wasting system resources. As the signature is only checked at the end of the transmission, an attacker can still carry out a large transmission and waste time and resources on the receiving sensor node. Although any resources will be released when the signature check fails, the sensor node will not be available to receive or send any other agents while this is in progress.

In a low-power, shared-medium system where packet filtering is carried out at the software level anyway, it is very difficult to avoid such denial-of-service attacks. An attacker can easily prevent the network from functioning usefully in a certain neighbourhood by continuously sending packets. As long as the packets
are valid at the physical layer the radio must still receive and store them before determining whether they are valid higher-level packets\cite{112}.

While spread-spectrum and frequency-hopping protocols can help avoid jamming attacks, the simple hardware of a wireless sensor node is limited in the complexity of approach it can take. The radio controllers used in wireless sensor nodes can only monitor one RF channel at a time and switching channels takes significant time. A sensor node may be able to avoid attacks by devices of similar capabilities but it can not defend against an attacker that is able to continuously transmit a high power, wide-band interference signal. Research indicates that the only viable defence of resource-limited wireless sensor nodes against unconstrained physical-layer attackers is to shut down affected nodes and attempt to route traffic around the area of interference\cite{113}. The normal agent routing system will route around unreachable segments of the network, however an intelligent attacker with sufficient capability to analyze packet headers and then jam certain packets could still allow routing packets to pass, jamming only agent transfers.

3.5 Agent Server

In order for agents to enter the system initially, they must either be pre-deployed on a node in the system or be provided externally. An agent server provide a library of agents and sends agents into the network as required. In addition to this, the agent server may allow an external user to push agents into the sensor network.

The agent server could be employed in a network-monitoring role, particularly if it is directly connected to a sensor node – however in this case, this node can no longer take a normal part in the sensor network’s operation. The decision to implement a simple agent server capable of operating without external input reflects the native mobile agent system’s design philosophy of autonomous, reliable operation.

When a node requests a service that is not currently available, and that cannot be supplied by another node in the network, the Agent Server will respond to the request. The agent server may run on a laptop or PDA, connected to a sensor
node in the network, or on a super-node with improved storage and power supply requirements. It stores a list of services and agents and can send them out to any node in the network. If running on a sensor node, it can communicate directly with the sensor network. If not, it uses a sensor node as a bridge to the sensor network, over a dedicated serial port link. This node also runs the middleware as normal, and the directly connected agent server is simply treated as another network neighbour.

The agent server may also push an agent into the network. This may be a simple push, in which case the agent is started on the node connected to the server, and moves itself to where it needs to be in the network, or a broadcast push, in which case the agent is pushed out to every reachable node in the network. Broadcast Push simply moves the agent to the connected sensor node, and then initiates a Broadcast Push, as described in Section 3.3.6. This provides a more efficient method of transferring an agent to all nodes on the network (this method of viral propagation is equivalent to Deluge in TinyOS and the existing SOS distribution protocol).

3.6 Agent Structure and API

The precise structure of an agent depends on the underlying operating system. All of the major WSN operating systems ultimately describe code modules (and therefore agents) as event handlers. Handlers are expected to complete quickly, while long-running operations are split-phase, and can be passed off to interruptible tasks. The mobile agent system does not impose any particular restrictions on the agent code – it relies on the capabilities of the OS to deal with normal operation.

3.6.1 OS Requirements

The middleware is not restricted to any particular OS, however it does impose some requirements on the OS. In general, it requires a dynamic and flexible OS. The requirements are summarised as follows:
3.6 Agent Structure and API

- Dynamic Modular Updates – The OS must be capable of loading a piece of code at run-time, without resetting or interfering with the operation of existing code. In effect this requires a modular execution and loading system. Multiple instances of a particular module must be capable of executing at the same time.

- Dynamic Memory – The OS must be capable of allocating and freeing memory at run-time. Mobile agents arrive and leave nodes frequently, and a static memory model is not sufficiently flexible.

- Code Memory Access – The middleware must be able to write to the code memory of the microcontroller on the wireless sensor node. The OS may require the middleware to pre-allocate the code memory and only write to certain sections, but it must have direct access.

- Native Execution – Security operations require complex processing, and will not execute in a reasonable time if executing in an interpreted system. The code must execute natively on the target hardware.

3.6.2 API Summary

In order to maintain efficiency, the agent relies on the underlying OS for most common operations. Only the functionality directly relating to agent-like behaviour accesses the middleware directly. Some of the most common interfaces are presented here, for reference.

```c
int8_t agent_move(uint32_t target_id, uint8_t pid, uint8_t options,
                  uint8_t responsemsg);
int8_t agent_copy(uint16_t target_id, uint8_t pid, uint8_t options,
                  uint8_t responsemsg);
uint16_t agent_utility(uint16_t number, uint16_t param);
uint8_t agent_remoterequest(uint8_t pid, uint8_t resptype,
                             uint8_t options);
uint8_t blackboard_write(uint16_t key, uint16_t value);
```
3.7 Conclusion

This chapter has described the structure of the proposed native mobile agent middleware. While the middleware draws ideas from some existing agent systems in wireless sensor networks, the capability to create mobile agents that execute natively is a novel contribution.

A hybrid memory allocation approach as well as a code caching system provide novel methods of reducing the amount of data transferred in the area of mobile
agents for resource-limited wireless sensor networks, while the transfer protocols provide a fast method of transferring agents to one or all agents in the system.

Mobility is the primary operation of this middleware, and represents its largest part. The mobility manager allows the system to operate autonomously and reliably, as any operation can be guaranteed to complete successfully or to signal a failure. This allows the development of truly autonomous, distributed and self-managing systems, which were not previously possible in embedded wireless sensor networks, as described in Chapter 5. The use of directed conditional mobility or carrier movement takes complex movement decisions out of the agent, allowing the middleware to efficiently direct mobile agents as desired. This is a new approach for agent mobility in WSNs which is evaluated in the following chapters.

Security on the other hand is not always required. In the restricted execution environment of a wireless sensor node, complex security operations impose a large overhead, but one that must be accepted in certain application spaces. The security manager implements a reliable authentication mechanism using a well-known industry-standard hash function. This allows a wireless sensor node to only accept agents from trusted sources, preventing the injection of malicious agents. The policy of existing mobile agent systems, that trust any information received over the radio link, while acceptable in a research environment, is far too permissive for an industrial or commercial system.

The operations manager assists the agents in their normal operation. This includes a remotely-accessible attribute space. A limited version of this is implemented, as the attribute space is intended as an aid to non-colocated agent’s communication, and not as the primary method of inter-agent communication. Neighbour support and agent management are included in this manager, all of which exist to speed up and reduce the size of the agents.

While the network is designed to operate autonomously once started, the agents must be introduced into the system from some external source. An agent server provides a library of agents and is capable of pushing agents into the system if needed. The agent server is simply another node in the network with more storage and power, and could in fact run on a suitable sensor node.
3.7 Conclusion

A number of general techniques have been established in this chapter, that are useful for mobile programming and mobile agent systems in limited execution environments in general. These are as follows:

- Hybrid memory allocation, encapsulation and reconstruction policies allow the middleware to efficiently manage simple state transfers, while the agent can self-manage more complex transfers. The middleware does not then restrict the type of state that an agent can use.

- The middleware can more efficiently manage a multihop transfer than an agent. If possible, the agent should use directed transfers, allowing the middleware to manage the per-hop transmissions, however explicit transfers should still be supported for cases that are not considered in the middleware design.

- A transaction-based transfer is needed for large agents. Per-packet acknowledgements are slow and inefficient for all but the simplest agents.

- Agent code should be stored in FLASH memory. The RAM is too small and must be shared between OS, middleware and all of the agents. Using FLASH memory allows the middleware to cache code.

- Code caching provides considerable performance benefits when agents are expected to revisit nodes. ROM storage allows a large code cache to be maintained (even in an interpreted environment), reducing subsequent data transfers.

This chapter has presented the architecture of the mobile agent middleware, and highlighted the contribution of this thesis to the state of the art in mobile agent systems for embedded wireless sensor networks. The following chapters present a suitable implementation of this middleware architecture, and an evaluation of the system in realistic application scenarios, demonstrating the power and flexibility of the system, and its suitability for modern dynamic applications.
Chapter 4

Implementation

The previous chapter described the architecture of the proposed mobile agent middleware suitable for implementation on wireless sensor nodes. This chapter describes an implementation of this system on embedded sensor node-scale hardware.

The SOS operating system is a dynamic and flexible OS for WSNs, and it was chosen as a suitable base on which to implement the mobile agent middleware. While SOS is a dynamic and flexible sensor network OS, some modifications were necessary to allow it to support mobile software agents. Section 4.1 describes SOS and the reasons why it was chosen as the basis for this particular implementation.

The aim of the mobile agent system is to allow the programmer to use completely the power of the underlying wireless sensor node. Programming the agents in standard C gives this flexibility, however due to the extra requirements of a mobile agent, there are some restrictions on the code that can be used. Section 4.2 describes these limitations and why they are necessary, and discusses the structure of an actual agent, with reference to a simple example.

It is extremely difficult to develop and debug such a system on actual hardware. The physical constraints of a wireless sensor node make it almost impossible to carry out such development. While hardware testing should be used to verify the correctness of the emulation, it is impossible to extract detailed statistics from a wireless sensor node without significantly affecting its operation. This is particularly true under a heavily loaded network, where the node is being pushed
to the limits of its performance capabilities – exactly the situation in which performance measures are most valuable. For this reason, the primary evaluation was carried out using AvroraZ[114], an extension of the Avrora sensor network emulator[3] that allows accurate emulation of the MicaZ hardware platform.

AvroraZ is described in Section 4.4. AvroraZ contains a detailed model of the microcontroller on the MicaZ node, allowing it to directly execute the exact same program as the physical hardware, down to the instruction and register level. It reproduces precisely the constraints of an embedded wireless sensor platform. This allows very accurate emulation of complex applications, complete with the timing and memory constraints of the physical system. Wireless sensor node applications provide services by interacting with the physical environment. In order to accurately emulate such applications, the emulator must be capable of providing a model of the physical environment and allowing the sensor nodes to interact with this model. The modular nature of AvroraZ allows this, and a temperature model was created for use in the application evaluations. The complete network emulation environment used, in addition to the physical scenarios are described.

Section 3.5 described the general architecture of an agent server, capable of providing agents to the system. Section 4.5 describes an implementation of such a server, executing on a computer connected directly to a wireless sensor node in the system.

Section 4.6 concludes this chapter.

4.1 SOS Implementation

The only WSN operating systems that are sufficiently stable, powerful and available for the class of hardware described in this work are SOS, TinyOS and Contiki. TinyOS’s static component interface makes it entirely unsuitable for mobile agent development. Previous work based on TinyOS such as Agilla[17] and TinyDB[27] builds a dynamic application layer on top of TinyOS, however to do this for a mobile agent system would require duplicating most of the functionality of the operating system, which would be very inefficient.

Contiki[9] also suffers from a large static program image. While loadable modules are supported, the system is not intended to support small, dynamic
and low-cost agents. Contiki's vision is of sensor network interfaces moving away from their embedded and autonomous computing roots and towards interactive, user-driven computing.

SOS[8] is a dynamic operating system for embedded wireless sensor nodes developed at the University of California, Los Angeles. It consists of a small, statically compiled kernel and a set of dynamically loaded modules. Module loading, in common with most of the SOS kernel, is lightweight and efficient. SOS compiles to a small program image, leaving much of the sensor node's resources available for the middleware.

For this reason, the middleware architecture described in this thesis was implemented in SOS. SOS is well-suited to this work as its simple kernel and modular system supports loading and unloading of self-contained software modules on a range of modern WSN hardware, while leaving most of the sensor node's resources available for middleware and application use, however it is not an integral part of the middleware and any suitably dynamic operating system could be used.

### 4.1.1 SOS application

The middleware is made up of a statically compiled SOS application. An SOS application is composed of one or more modules interacting with each other and the kernel via three methods:

- **Direct Kernel Calls** – A limited number of system calls are provided at fixed addresses. These functions may be modified with a kernel recompilation, but this requires redistributing the kernel to all nodes.

- **Dynamic Functions** – A module may register dynamic functions. Other modules may then call these functions, and the function call is dynamically linked into the calling module. The kernel handles all possible failure cases and ensures that only valid links are made.

- **Message Passing** – This is the simplest and most common communication method. Messages are passed from one module to another, either locally or to a one-hop neighbour. Messages are identified by a target Process ID (PID) and Message Type. If a handler exists, the message is passed to it.
4.1 SOS Implementation

If not, it is silently discarded. Local message delivery is guaranteed, remote delivery is not. Messages are queued, with kernel messages taking priority over application messages.

A static application is compiled directly into the kernel image. This allows the middleware to efficiently access the kernel resources.

Dynamic memory allocation is widely used. Messages are dynamically allocated and released when processing has completed, and modules allocate dynamic memory for storage and processing tasks. Hardware memory management units are not available on embedded sensor node-scale platforms so all memory management is software-based and there is no memory protection. Apart from the kernel, all code executes inside a module. Each module is identified by an 8-bit PID.

SOS has been ported to the Mica2, MicaZ, T-Mote (and the various MSP430 / CC2420 platforms functionally equivalent to the T-Mote), XYZ and I-Mote platforms. The software is released under an Open Source licence permitting modification and redistribution. The software tools required to build the system are also available under Open Source licences.

In this work, the basic SOS kernel has been modified to improve the integration of the middleware framework. The modifications made are described in the following sections.

4.1.2 Scheduler

SOS applications are composed of a set of modules, interacting using the methods described above. Each module is self-contained, and the kernel manages inter-module communication. Modules are primarily identified by their eight-bit Process ID (PID). PIDs below 128 are reserved for kernel modules, internal and platform specific modules. PIDs above 223 are reserved for threads. Threads in SOS are simply modules that can be instantiated more than once. The space from PID 128 to PID 223 is therefore available for application modules, and 224 to 254 for threads (PID 255 is reserved). More than one instance of a mobile agent of a particular type may be active on a sensor node at the same time, so mobile agents must be created as threads. This imposes a practical limit of 31
agents executing at any one time. While this could easily be increased, it is likely that memory restrictions would be reached before the thread limit.

The PID uniquely identifies each particular instance of an executing agent. Implicit in this system is the assumption that each PID corresponds to exactly one module, and that each module is always active. While it is necessary to have a PID identifying each executing module, this is not sufficient in a mobile agent system when there may be multiple instances of a particular agent executing at one node at the same time – alternatively a module may be physically present in the system but may not be active at that time.

4.1.2.1 Code ID

The scheduler was modified to include an extra piece of information with each module – an 8-bit Code ID (CID). In the case of a normal SOS module, or a middleware service, the CID is simply equal to the PID. In the case of a mobile agent, the CID identifies the type of agent, while the PID is different for each instance. The CID is used to manage agent-specific information by the middleware – this is described in Section 3.2.1.

CIDs provide a globally-unique ID for mobile agents. Agents whose CIDs are equal are guaranteed to have the same binary code. While executing on a sensor node, the PID of an agent is dynamically allocated (from the thread PID pool) and may vary as the agent moves from one sensor node to another, while the CID always remains the same.

No attempt is made to coordinate the CID between different agents. The CID space must be externally managed – the middleware assumes that CIDs have been correctly allocated. Increasing the size of the CID, or modifying it to include some source information as well as a numeric identifier could allow multiple classes of agents to operate in the same network without external coordination, however this information would then need to be stored for every agent. As discussed in Chapter 3, each agent communicates using a specific interface, tied tightly to the operation of that agent – generic interfaces result in heavyweight and inefficient translation layers. Agents wishing to communicate with each other require reasonably detailed knowledge of the interfaces used. This requires a
large transfer of information at the design stage, and it is reasonable to assume that CIDs can also be coordinated at this point to avoid collisions. The eight-bit space is wide enough to allow some managed sub-division, however this must be carried out by the user and not by the middleware.

This system has limitations – the CID list must be known to all users of the network (a collision in the CID space would be catastrophic for the agents as one agent could then receive the state of a completely different agent). A naming system such as a hierarchical namespace could be used to guarantee global uniqueness, however this would introduce significant overhead.

The proposed middleware is targeted towards networks with a low level of external connectivity, in relatively controlled environments. For this reason, an externally managed namespace such as the CID list is a good compromise. Multiple independent users of the system can still be supported as long as some external mechanism is used to coordinate usage of CIDs. This could involve reserving portions of the CID space for each user, or the users communicating their usage of CIDs using some external tool.

The CID (and associated PID) are widely used to identify agents and processes – widening the CID will increase the size of many internal data structures and tables, to no advantage as the system could not begin to support even the 256 different agents that are possible with an eight-bit CID. More sophisticated hardware could support a hierarchical agent namespace (such as that used by Java classes) or some distributed synchronisation mechanism such as those used on the internet but the considerable overhead of such a system is not justified in the proposed middleware system.

4.1.2.2 Module Suspend

Agents must transfer reliably from one sensor node to another. If an agent initiates a transfer, the transfer must either complete successfully, and the agent begins execution at the target sensor node, or fail, with an error at the initiating sensor node – the agent always resumes at exactly one of the endpoints.

While agents are being transferred they must be suspended to maintain consistency of state. The transfer may take multiple seconds and were the agent to
continue executing for this time the state could diverge significantly from that sent. An agent can be effectively suspended for a short period of time by entering an atomic section (i.e. disabling interrupts on the microcontroller) however this is not feasible for long transfer operations and would likely interfere with hardware interfaces such as the radio and timers. The agent can not simply be deleted because if the transfer were to fail, the agent must be capable of resuming on the initiating node.

SOS was modified to keep a list of suspended modules. Messages intended for the suspended module are filtered out and silently discarded, as they would be after the transfer has completed anyway. Messages that are already in the queue intended for a suspended module will be discarded. Timers, sensor readings and any other system resources will continue to exist and function normally but any messages they attempt to send to the suspended module will be silently discarded.

More complex systems are possible. Initially, a separate queue was kept of messages for suspended modules. This was found to grow in size unsustainably. The queue would quickly consume too much of the dynamic memory, particularly with long or slow transfers – even with small messages, the 22 byte overhead of each message quickly exhausted the available 2kB of dynamic RAM. If the transfer completes successfully, these messages would then be discarded in any case. If not, the messages serve little purpose after resumption of the agent anyway – for example an agent with a 100ms timer running, attempting to move to a neighbouring node for 5s before giving up would then have to process 50 out-of-date timer messages on resuming. Failed transfers run for much longer than successful transfers, as various timeouts and retransmissions must occur before the middleware aborts the transfer.

4.1.3 Memory Management

Memory in SOS is either static or dynamic. Only kernel modules may use static memory. As the SOS kernel is ultimately compiled as a C program, static variables are simply the global variables available to any C program. A large block is reserved for use as the dynamic memory heap – once all of the static memory has been allocated, and sufficient space is left for the execution stack (which need
not be very large as the message-passing architecture of SOS tends to process everything serially, with only one context active at a time), all of the remaining memory should be made available for dynamic allocation.

4.1.3.1 FLASH Buffer

The design of FLASH memory does not allow random-access writes. A 256-byte page must be erased and rewritten in its entirety. Loading a module involves writing a number of values directly into the program code, in order to fix up addresses. A cache of one page is kept to ensure that physical writes are minimised. In a standard SOS kernel module updating occurs infrequently so this buffer is dynamically allocated. In the mobile agent system, updates to the code memory are a frequent operation so it is more efficient to statically allocate it. This also reduces memory fragmentation due to frequent allocation of the memory block.

With frequent active transmissions, the actual available dynamic memory is only reduced by 0.25kB and fragmentation is reduced slightly. The typical SOS memory structure is shown in Figure 4.1.

![SOS Memory Map](image)

Figure 4.1: SOS Memory Map
4.1 SOS Implementation

4.1.3.2 Local and Remote

In the standard SOS kernel, all dynamic memory is allocated equally, although kernel modules may specify a “Longterm” option. Longterm memory is not expected to be freed, or certainly not frequently, so it is all allocated together at the end of the dynamic memory to reduce fragmentation.

This is modified to introduce an extra flag into the memory map – “Local-Only”. The default memory allocation is not local-only. The distinction is only significant for agent modules - Local-only memory is guaranteed not to be moved during an agent state transfer. It is used for data tables or storage that are only useful on one sensor node, such as neighbour lists or sensor parameters that would need to be regenerated if moved to a different node.

A memory block is classified at allocation time. Subsequent resizing will preserve the flag – it must be freed and reallocated to change the status. Although allowing existing blocks to be marked dynamically might be useful, it is difficult to imagine a scenario in which this would be common and would require the overhead of an extra function call.

4.1.3.3 State Snapshot

The functionality described in the previous sections combines to produce the snapshot of an executing module’s state required by the mobile agent middleware. This is done with interrupts disabled and the module suspended (see Section 4.1.2) to guarantee consistency. The management and reassembly of the state snapshot is discussed in more detail in Section 3.3.2.

The snapshot for a particular agent is copied into a buffer for transmission. It consists of three parts:

- Module State
- Memory Map
- Dynamic Memory

The module state is the dynamically allocated SOS module state. An SOS system call exposes this for a particular PID and it is copied into the buffer.
The memory map identifies each block of dynamic memory allocated by that PID, apart from its module state and any memory blocks marked Local-Only (as discussed above). The memory location of the block and its size are stored in the memory map. This is used to reassemble the module state at the destination sensor node (see Section 3.3.2).

This is followed by the dynamic memory. This is a copy of all of the memory blocks, in the same order in which they appear in the memory map. SOS does not track the actual size of memory allocated, just the number of memory blocks it uses. Blocks are eight bytes in size, with three bytes of overhead per allocation. The memory blocks (and the sizes in the memory map) are therefore rounded up to the nearest multiple of eight, minus three. For example, if a module allocates a block of 15 bytes, 21 bytes will actually be transmitted – 15 bytes + 3 bytes of overhead = 18 bytes which requires 3 8-byte blocks. These 24 bytes, minus the 3 bytes of overhead, makes up the 21 bytes transmitted.

The overhead of tracking the actual allocated block size would exceed the average 3.5 byte per block overhead introduced.

### 4.1.4 Module Distribution Protocol

The module transfer mechanism described in Chapter 3 is much more powerful and flexible than the existing SOS module transfer protocol and exceeds its capabilities in every aspect – the old protocol is therefore no longer required and it was removed from the SOS image used by the middleware.

### 4.1.5 Debugging System

Debugging sensor network applications after deployment is difficult. Debugging strategies are either extremely simple, such as flashing LEDs, or extremely invasive. Naive strategies such as sending out a packet containing debugging information or sending data to the serial port are simple and can be effective on lightly loaded systems, but will still consume significant battery power. As the system loads increases, the overhead of sending packets or communicating with the serial port will begin to significantly interfere with the system being debugged. Toggling LEDs in a certain sequence can provide debugging information with very
little interference with the running system, however the amount of information that can be conveyed is very limited. Protocol analysers exist for standard systems such as IEEE802.15.4 [19], however these will not aid debugging of a higher level system and the state of the sensor node must still be inferred from the data it is generating.

A debugging system was created, based on a monitor that was added to the AvroraZ emulator (see Section 4.4 for more details on AvroraZ). A 64-byte buffer in the microcontroller RAM is reserved for debugging. The monitor traps writes to the first byte of this buffer, and prints the contents of the 64-byte buffer as a debug message, formatted as a null-terminated string.

A DEBUG macro is then used in the SOS module code. This uses a simple sprintf call to fill the buffer with a variable length string (up to the maximum 63 bytes), and then writes a zero byte to the start of the buffer. Avrova traps this and prints the rest of the string to the output of the emulator. The debugging output is specifically flagged as such and can be filtered and monitored to observe the system's operations. The free-form string gives the programmer complete flexibility in what output is required, although the longer the string and the more variables are included, the longer it will take.

There is still potential for interference with the system – clearly any inline debugging will have some effect on the system being debugged. As the string is only written to local memory, this is considerably less than sending a packet or writing to the serial port. However with a typical microcontroller only operating at a speed in the low MHz, and with multiple instructions required in the RISC instruction set to carry out a FLASH-to-RAM copy, outputting a debugging string can still take tens to hundreds of μs. This is enough to interfere with code in the critical timing path, such as low-level hardware interface code and care must be taken not to place debugging statements into timing-critical sections or frequently-repeated loops.

Care should also be taken to keep the debugging strings as short as possible. Each printf call consumes some FLASH memory to store the string, so debugging should only be enabled for modules that are being developed – if hundreds of debugging statements are enabled the available storage will quickly be consumed.
4.2 Programming Restrictions

For accurate measurement of timing-critical sections, and to generate the execution time measurements used in the following chapter, a less invasive method is used. An unused general-purpose I/O pin is toggled high at the start of the measurement and low at the end of the measurement. This requires only two instructions, or 0.25\(\mu\)s when operating at 8MHz. The timing of these outputs is extracted by the emulator, allowing accurate measurements to be made.

4.2 Programming Restrictions

There are some operations that, while allowed by the C language, will not work correctly due to the nature of the mobile agent. These restrictions are imposed by the SOS module structure and by the limited C library provided by a typical embedded compiler.

Static variables will not function correctly. The only static variable allowed is the module state block. This includes local static variables inside functions. Dynamic memory must be allocated using sos_malloc and must respect SOS block ownership.

Floating-point mathematical libraries, while available, are too large to co-exist with the middleware and operating system. Complex integer operations are not well supported by the 8-bit microcontroller and will likely compile into code that is large and inefficient. In particular, there is no hardware divide operation so this will run entirely in software unless it can be optimised to a simple bit-level operation.

Static constant variables stored in FLASH memory (such as large constant tables) will also not function correctly. This includes constant array initialisers to local variables, most commonly string variables. In normal static code, these are read from the FLASH memory, however the dynamic linker can not reliably patch up the address due to a limitation in the avr-libc library. The library uses a 16-bit pointer to point to constant memory locations (i.e. code memory), however the microcontroller used has 128kB of memory. For static code, the compiler simply ensures that all constant initialisation is placed into the lower half of memory, however for dynamically linked code this is not possible.
Most C standard library functions are unavailable. The ISO C standard distinguishes between hosted and freestanding implementations — embedded C compilers are termed freestanding implementations (as there is typically no underlying operating system) and are not required to implement all of the C standard libraries. Many library functions do not apply to embedded systems (for example, interactive functions and file I/O) while others are too complex to be implemented on such a simple system. Only the standard C language, the operating system and middleware functionality, and simple library functions such as string and memory manipulation will be available.

4.2.1 Necessary operations for mobility

The weak mobility model employed puts some restrictions on the agent when it moves from one node to another. While all allocated memory is automatically transferred, the agent must remap this to addresses on the new node (this is described above in Section 3.3.2, and should inform the mobility manager when the remapping is completed, by calling the remapping function with NULL arguments. Even if dynamic memory is not being used, it is good practice to perform a NULL mapping anyway as this will release any memory being used to store transferred state.

Any timers, sensor readings or any other hardware interface operations are not transferred between the nodes. Due to the potential variation between nodes, and the likely requirements for setup operations at a new sensor node, it is not a trivial operation to transfer the hardware state. On arriving at a new node, the agent must therefore restart any timers it uses and re-register any sensor interfaces or other hardware devices. Clearly, any handlers or references to hardware devices remaining from the previous sensor node will no longer be valid after moving.

Although a standard SOS module, a mobile agent is not yet completely loaded when started at the SOS level. The agent state is not set up and the middleware may not yet have entirely completed its loading operations. While it is safe to call any SOS kernel functions, however the middleware functions should not be called. It is safer to ignore the SOS module start and finish messages, using instead the mobile middleware versions.
4.3 Agent Structure

The SOS module unload message is used to inform the agent that it is to be terminated (including during a successful agent move). The agent should not perform any split-phase operation from this handler – as soon as the handler returns, the agent is terminated and the PID is no longer valid. This gives the agent a chance to shut itself down gracefully.

4.3 Agent Structure

The mobile agent executes as a normal SOS module, and must follow the structure of an SOS module to allow it to be loaded and function correctly. This is described in more detail in the SOS literature and website[8; 115].

The only static local variable a module may have is its module state block. This is dynamically allocated by the OS before the module is started and is fixed for the lifetime of that module. Typically, a structure is defined to describe the structure of this block (this is only for the benefit of the programmer – the OS needs only to know how much memory to allocate). Figure 4.2 shows a sample module state structure, taken from the Blink tutorial on the SOS website[115]. It has a size of two bytes, and defines two variables, pid and state. The two bytes will automatically be allocated by SOS and the address of this block will be passed to each message handler, allowing it to address it as if it were a static variable.

```c
typedef struct {
    uint8_t pid;
    uint8_t state;
} app_state_t;
```

Figure 4.2: Example SOS Module State

The module must then contain a module header. This must be present in every module – a module will not compile without one. An example is shown in Figure 4.3. Most of the fields are standard and can be left to defaults. Values that may need to be changed are as follows
mod_id and code_id – For a standard SOS module, this is the PID. For an agent, this represents the CID (these may be the same in the case of a service). These must be unique and should be the same. The mod_id is in the local numerical format, the code_id is in a network-neutral format (to avoid endianness issues). Values below the predefined constant DFLT_APP_ID0 are reserved.

state_size – This is the amount of state that must be allocated. If an app_state_t structure has been defined, simply using sizeof() as shown will automatically compute the correct size.

ever_sub_func, num_prov_func – These relate to dynamic function linking – these are not used by the agent framework and should be left at zero unless the module uses them for some other purpose.

platform_type, processor_type – These identify the hardware platform, so that code for one platform will not attempt to execute on another. The defaults use the current compilation target platform which is almost always the desired operation.

module_handler – This is the function used to process all messages passed to this module.

```
static const mod_header_t mod_header = {
  .mod_id = DFLT_APP_ID0,
  .state_size = sizeof(app_state_t),
  .num_sub_func = 0,
  .num_prov_func = 0,
  .platform_type = HW_TYPE,
  .processor_type = MCU_TYPE,
  .code_id = htonl(DFLT_APP_ID0),
  .module_handler = blink_msg_handler,
};
```

Figure 4.3: Example SOS Module Header
4.3 Agent Structure

The module handler handles all messages passed to this module. SOS messages are a standard structure with a dynamically allocated payload. Each message contains information about how it should be handled – for example, the source address and module, the message type, the message size and any payload data. Apart from the pre-defined system messages, the module is free to define and handle any other messages it wishes. The most common system messages are:

- MSG_AGENT_START – This is sent by the operation manager any time an agent or service is started. By the time this message is sent, any mobile state has been loaded and the agent can commence normal operation. The payload of this message is a single byte representing the options used in the transmission. In particular, the least significant bit of this option status is set if the agent is being started freshly (i.e. the module state must be initialised), and clear if the agent has moved to this sensor node from another (i.e. the module state has been copied and is already initialised).

- MSG_FINISH – This is sent to a module before it is terminated. As soon as the message handler returns, the module is terminated. It should release any resources that require cleanup.

- MSG_TIMER_TIMEOUT – This message is posted whenever a timer expires. The payload gives the timer number, as specified when the timer was started.

- MOD_MSG_START – This is the first general-purpose message identifier available to the module. This and any higher message identifier (MOD_MSG_START + 1 etc.) may be used for the module’s internal purposes, and to communicate with other modules or agents.

- MSG_DATA_READY – Posted when a sensor reading is available.

Some sample agents are shown in Appendix A. The functionality of these agents and how they translate into executable code is discussed in Chapter 5. For more discussion of this example and much more information about writing SOS modules, see the SOS documentation[8; 115].
4.4 Hardware Emulator

<table>
<thead>
<tr>
<th>Kernel</th>
<th>FLASH (kB/%)</th>
<th>RAM (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOS</td>
<td>48.0 (37.5%)</td>
<td>3.349</td>
</tr>
<tr>
<td>SOS with middleware</td>
<td>64.0 (50.0%)</td>
<td>3.401</td>
</tr>
<tr>
<td>SOS, middleware, security</td>
<td>77.8 (60.8%)</td>
<td>3.403</td>
</tr>
</tbody>
</table>

Table 4.1: Kernel Sizes

4.3.1 Middleware Size

The middleware has been designed to fit into as small as memory footprint as possible. Wireless sensor nodes are resource-limited devices, and a mobile agent middleware is useless if it does not leave enough resources for functional mobile software agents. The storage and RAM usage for various configurations are shown in Table 4.1. The memory usage is given in terms of the The ATMega128L, with 128kB of FLASH memory and 4kB of RAM. 2kB is reserved for dynamic memory in both cases. Other nodes such as the Tyndall nodes[116] contain up to 8kB of SRAM, which would leave 6kB of dynamic memory free for mobile agents.

The three values listed are for the standard SOS kernel with all of the hardware interface functionality required, SOS with the mobile agent middleware, but without the Security Manager and SOS with the middleware including the complete Security Manager. The middleware fits comfortably into a resource-limited, embedded wireless sensor node environment such as the ATMega128L. This is a significant result as existing middlewares of similar functionality such as Sensorware[14] requires 180kB of storage, while the SunSPOT[18] Java middleware requires 848kB. Sensorware also equires a device capable of running embedded Linux, such as a 32-bit StrongArm processor operating at 206MHz[14], while the SunSPOT uses a similiar 180MHz 32-bit ARM920T processor[18].

4.4 Hardware Emulator

Developing, debugging and evaluating applications in a live Wireless Sensor Network is extremely difficult. The authors of one of the first and most well known large-scale outdoor deployments of a Wireless Sensor Network[40] concluded that node reliability in a real deployment is much worse than in a typical workbench
4.4 Hardware Emulator

deployment. Most wireless sensor node hardware is cheap and not particularly robust, and the physical environment is likely to affect system performance as much as any protocol or architecture. Physical failures and software errors causing battery depletion are common, requiring manual intervention to reset or reconfigure the network.

For this reason, emulation is a crucial tool in the development and evaluation of complex wireless sensor network systems and applications. Emulation provides a virtual microcontroller, complete with all of its constraints. The same program image is used on the emulator as on the physical hardware. The flexibility of varying operating conditions under the complete control of the user, combined with the ability to non-intrusively inspect the operation of the application at a detailed level simply cannot be replicated in a physical testbed. In order for the emulation to be useful, the emulator must be accurate enough that the program image used is identical to that of a physical sensor node. As wireless sensor networks are so sensitive to low-level hardware performance, this requires cycle-accurate emulation of the execution environment. This includes both the communication aspects of the application, as modelled in a traditional network simulator, and the low-level hardware access, direct code execution and inter-device communication within the sensor node itself.

4.4.1 AvroraZ

The AvroraZ emulator was used in the evaluation presented in this chapter. AvroraZ[114] is an extension of the Avrora[3] emulator. Avrora is an emulation and analysis framework that allows cycle-accurate emulation of the ATMega128L microcontroller[117], and the CC1000 radio interface chip[118] used on the Mica2 wireless sensor node[119]. Avrora emulates the hardware performance precisely, down to the register and instruction level.

AvroraZ extends Avrora to include support for the CC2420 radio interface chip[25] – an IEEE 802.15.4 (Zigbee) compliant radio transceiver[19]. This allows emulation of the MicaZ wireless sensor node, manufactured by CrossBow[71]. All of the features used by the sensor network operating system are supported, in-
4.4 Hardware Emulator

Figure 4.4: Avrora Structure (from [3])

including address recognition, clear channel assessment (CCA) and the link quality indicator (LQI) based on an accurate indoor radio model.

Avrora is written in Java [81], with a hierarchical, object-oriented class structure allowing easy extension and expansion. Each device in the system is represented by an object, and standard interfaces are defined, reducing the work required to add new devices. Each sensor node operates in a separate thread, synchronisation is performed between threads only when required due to communication. Avrora can support network sizes up to thousands of nodes [120] (although the Java virtual machine will require a lot of memory to do this). The structure of the Avrora emulator is shown in Figure 4.4.

The radio model used is a probabilistic model, based on measurements taken on a physical testbed. This is described in much more detail in [114].

As Avrora emulates the AVR hardware directly, down to the assembly language instruction level, the code that is run on Avrora is identical to that run on the real hardware. This removes a common source of errors in wireless sen-
4.4 Hardware Emulator

sensor network simulation – that the code run on the simulator is only a model of the actual application. The disadvantage is that a complete application must be written from top to bottom before it can be simulated – protocols can not be simulated in isolation.

The approach taken by both TinyOS[121] and SOS[8] is to add an emulated hardware platform. This generates a PC application that emulates some of the functionality of the sensor network application. Hardware interface operations are replaced by an idealised model and detailed timing analysis is not possible. While this serves as a useful tool for rapid prototyping, it can not be considered a full simulation of the application.

MSPSim[122] is an emulator for the Texas Instruments MSP430 family of microcontrollers, similar to AvroraZ.

At the other end of the simulation space, traditional network simulators such as ns-2 and OPNET are beginning to be adapted for wireless sensor network simulation. However they are only intended to model the network in terms of communication protocol and application models, assuming that the application layer is independent of the network. The tight cross-layer coupling of wireless sensor nodes is not well represented. They are essentially protocol simulators rather than network device simulators.

4.4.2 Physical Simulation Extension

The modular architecture of AvroraZ allows it to be easily extended. In order to test meaningful complete applications, an extension was written for AvroraZ to model a physical environment. This created a temperature map across a multi-room environment. Each node is located in this environment by its physical position (the same position that is used for the radio model), and one ADC channel is connected to an idealised temperature sensor. The dynamics of the temperature sensor are not considered, but the ADC and its timing characteristics are modelled accurately by AvroraZ.

Some nodes in the system are connected to heaters – each heater’s output is controlled by two of the otherwise unused general-purpose output pins of the
microcontroller. Parts of the system can be subjected to other temperature constraints – for example, doors and windows can be held at a certain temperature profile, either constant or time-varying.

Internally, this is represented as a 100x100 grid of temperature points. Intermediate temperatures are found by interpolation. 10 times per second, the system updates each point with a weighted average of its neighbours. While this is not a perfectly physically accurate model, it is quite sufficient to evaluate a control application. The primary concern is to provide meaningful temperature values and to observe the control effect rather than to precisely model a real temperature system.

4.5 Agent Server

Section 3.5 described the necessary functionality for a server capable of supplying agent code to the wireless sensor network. This section described an implementation of such a server, running on a PC connected directly to a wireless sensor node using a serial cable (or USB interface).

4.5.1 Server Implementation

The design of SOS allows this to be easily achieved. While primarily targeted at sensor node-scale devices, it can also be compiled for a PC target. This means almost exactly the same middleware code can be used for the agent server as is used for the sensor network version. This eliminates a potential source of problems due to inconsistencies between two different codebases. Any changes to the distribution protocols or packet formats require only a recompile of the server application, greatly improving reliability. The mobile agents themselves can not execute on the server – it is simply a repository of agent code. The binary mobile agent code is compiled for a microcontroller and not a PC. The server is given a list of agent files, which are read to determine the characteristics of each agent.

When the server receives a query (transparently translated by a dedicated bridge between the serial interface and the agent server itself), it checks the list of agents. If it can fulfill the query, it responds using the same mechanism as
any other node in the network, which is translated back through the bridge. If an agent request is then sent, it reads the agent code out of the file system and transmits it using the same code as the sensor node version. The bridge can also be connected to the virtual serial interface of a wireless sensor node in an AvroraZ simulation.

The only issue with connecting to the AvroraZ emulator is one of timing – large simulations will run at slower than real-time. The agent server must therefore slow down the transmission of packets, or they will appear at the simulated nodes too quickly to be processed. This is achieved by introducing a scaling factor to the timeouts in the agent server – this is adjusted to produce reasonable behaviour. The normal feedback mechanisms of the transfer protocols (retransmission timeouts, backoffs and NACKs) ensure that the data is managed correctly.

An adaptation mechanism was introduced to help in this process – if retransmissions are encountered (meaning a timeout has expired without any packets being received), the agent server speeds up its transmissions, if NACKs are received (meaning packets have been lost – since the link is a dedicated serial link this generally means that the packet has been sent too fast or the destination is too busy to process it), it slows down.

### 4.5.2 GUI

A GUI allows the user to manage and view the status of the agent server (shown in Figure 4.5). This was developed (in GLADE[123], an XML-based cross-platform GUI creation library) to allow interaction and to display the status of the application.

The user may use this interface to push agents into the network. Agent pushes are either Unicast or Broadcast. In the case of a Unicast push, the agent code is transferred to the wireless sensor node that is directly connected to the server, and started there. The agent is then responsible for moving or duplicating itself. In a broadcast push, the agent is transferred to the directly connected node, which then attempts to copy it to all of its neighbours, and on to every connected node in the network, using the broadcast push mechanism described in Section 3.3.6.
4.6 Conclusions

This chapter has presented an implementation of the middleware architecture for mobile software agents that was presented in the previous chapter, based on the SOS operating system. Some changes were made to the OS to support mobile agent operations, particularly in the area of memory and process management. The middleware executes on top of the SOS operating system, as a normal SOS module.

While the mobile agents are written in standard C, some operations are not possible, due to the structure of the mobile agent and the requirement for mobility. These are relatively minor restrictions and a suitable choice of programming model can easily avoid them. A simple example of a software agent was also presented, to demonstrate how easily agents may be created.

In addition to the middleware itself, a suitable agent server must be created to introduce agents into the system. An example of an agent server was presented.
4.6 Conclusions

operating on a PC directly connected to a node in the wireless sensor network. This server includes a user interface allowing a user to interact with the agent system.

More complex examples of agents and applications will be given in the following chapter, along with a detailed performance evaluation of the middleware system implementation presented in this chapter, in some typical application scenarios.
Chapter 5

Evaluation

The previous two chapters have presented a middleware architecture and a particular lightweight implementation supporting mobile software agents on resource-limited wireless sensor networks. This chapter contains a detailed evaluation of the architecture and its implementation and operating characteristics. Analysis of the middleware performance, combined with detailed application evaluations demonstrate the viability of native mobile software agents and the advantages of this novel lightweight native mobile agent middleware in power and flexibility compared to existing static or interpreted systems for similar hardware platforms. Autonomous and reliable operation is demonstrated that is not possible using any existing technology for mobile agents in comparable embedded wireless sensor networks.

An evaluation can not be carried out without first defining the criteria under which the system is to be evaluated. These have previously been established in Section 2.3.1, and are restated in Section 5.1. These criteria vary in form, from easily quantifiable values that can be directly compared between one system and another, to less specific concepts that require analysis and can not always yield a direct comparison. The issues relating to these criteria are discussed, and how they should be treated in any analysis. The rest of the chapter is structured around the criteria established in this section.

Only a complete application presents a true picture of how the system performs. Wireless sensor network applications are too tightly coupled to the hardware on which they operate, with too many cross-layer linkages to allow them
to be accurately split up into independent layers or sub-systems. A range of applications are presented, with varying performance benefits resulting from their implementation as native mobile agent systems. These applications are presented in Section 5.2, along with the relevant performance results.

The three application scenarios identified are presented in Sections 5.3, 5.4 and 5.5, along with some tests that do not involve an application in Section 5.7, and considered according to the criteria established in Section 5.1.

While direct comparisons between systems with widely varying functionality are difficult, there is sufficient data to conclude that native mobile software agents provide a powerful and flexible alternative to static code distribution, and open up application spaces that are not available to interpreted mobile software agents. Complex mobile software agents are particularly powerful when operated as a network of cooperating agents in reliable, autonomous and dynamic applications. Section 5.8 discusses this and concludes the chapter.

5.1 Evaluation Criteria

The work presented in this thesis distinguishes itself from the state of the art in that it provides mobility to natively executing code, or alternatively, native execution to mobile agent code. The benefits of mobile software agents are well established, both in larger systems[90] and in wireless sensor networks[17; 108]. Application scenarios for wireless sensor networks are a well studied field[1; 16; 26; 41], and it is sufficient to evaluate the proposed middleware in the areas in which it differs substantially from existing systems.

Compared to existing code distribution protocols for natively executing code, the proposed middleware adds mobility. The characteristics of mobility must therefore be evaluated – both the model, determining when and how agents may move, and the latency, determining how long these moves take. The latency is also closely related to the energy consumption.

Compared to existing mobile agent systems for wireless sensor networks, the proposed middleware adds native execution. The execution model must therefore be evaluated. This model determines how code is executed, and also contributes to the power consumption.
Security is also added to existing mobile agent systems, and the overhead introduced by the security system must be evaluated, in terms of both time (movement latency) and energy.

The central thesis of this work is that reliable and autonomous behaviour is enabled by this native mobile agent middleware, that was not previously available in resource-limited wireless sensor networks. While autonomy is a function of application design, although enabled by the execution model, reliability also requires support from the middleware. The resulting reliability is measured in complete application scenarios, in order to analyse how much the increase in reliability is offset by worsening of performance in other areas.

All of the criteria are tied together by the energy consumption. In order to properly evaluate this, the entire system must be considered in realistic, complete application scenarios.

In summary, this chapter evaluates and compares the proposed middleware system with the state of the art using the following criteria:

- Mobility Model
- Movement Latency
- Execution Model
- Energy Consumption
- Security
- Reliability

5.2 Application Scenarios

Only a full-application evaluation captures the complexity and resource limitations that are encountered in real application development. In order to present a complete evaluation of the middleware system, a number of application scenarios were implemented. Survey articles on Wireless Sensor Networks[41; 42; 124] identify numerous applications, however these fall into one of a few categories:
5.2 Application Scenarios

- Remote Data Gathering
- Event Monitoring and Localisation
- In-Network Processing
- Distributed Control and Actuation

In order to provide a comprehensive evaluation of the proposed mobile agent middleware, three applications were chosen that cover all of these categories. These are typical of the application space of wireless sensor networks. Mobile software agents are best suited to mobile and dynamic applications, however they can also find use in more traditional contexts.

- Data Gathering is the simplest of the applications. Sensor readings are to be taken from a remote wireless sensor node, and transmitted back to an externally connected base station, where they will be transferred out to an external application. Data gathering is a relatively simple task for a mobile software agent, and the overhead of the agent system will be considerable. However it forms a base on which more complex applications may be constructed.

- Location Determination combines data gathering with Localisation and In-Network Processing. The agent must locate a particular mobile node, then determine its location based on the quality of the received signal from various fixed neighbours.

The algorithm used for location determination in this scenario is loosely based on one used for pedestrian detection in an automotive scenario [125].

- Distributed Reliable Control requires a combination of all of the areas described. The application must manage a set of wireless sensor nodes, with varying sensing and actuation capabilities. It must discover the initial topology, and respond to changes in the topology. It must deal with node failure and communication failure, while continuing to maintain the best possible service. The resulting system was found to operate autonomously and reliably even in the face of complete node failure.
5.2 Application Scenarios

5.2.1 Physical Scenario

All of the applications use a similar physical scenario. The significant features of the sophisticated native execution model are best observed in a complex application scenario. Measurement, control and actuation are the most common such applications and these most commonly occur in an indoor environment. The size represents a medium-sized office or industrial building – the typical target area of current wireless sensor network deployments.

The radio range of a wireless sensor node can reach up to 100m outdoors, however in an indoor environment, reflections and interference from various electrical and radio-frequency devices will reduce this substantially. Typical indoor radio range is closer to 20m, with a wide variance. Each node has between 3 and 8 neighbours, with an average connectivity of 5.76 (assuming a square grid).

25 nodes was chosen as this represents a large building of approximately 100m squared, according to measured indoor propagation characteristics at 2.4GHz, or 0.5km squared outdoors[126]. The technology is scalable to larger networks, however it is difficult to extract meaningful statistics on specific operations in a very large deployment, and there is little value added to the simulation by expanding the size of the network. The agent and carrier move operations communicate only with the node’s one-hop neighbours, and will scale upwards easily. The resulting system currently uses AODV – an alternative routing implementation would most likely yield better results for a very large network. Similarly, the agent service request uses a DSR-like mechanism to discover local sources of agent code – an alternative mechanism more suited to very large-scale networks would probably yield better results. 25 nodes is a large enough network to observe significant multihop effects, while the size simplifies instrumentation and monitoring of the system performance.

The physical scenario is fixed for all of the tests. The fixed, regular node layout removes potential distortion of the results due to irregularities in the node deployment. An irregular deployment would be expected to demonstrate higher variability for all systems, without adding to the results. The fixed, regular grid provides a standard, repeatable setup for comparison. The agents in the network do not have knowledge of the topology and must discover it every time.
5.3 Data Gathering

The neighbour density is also a contributing factor. Very well connected networks will suffer from substantial overhearing effects, reducing the available network bandwidth. The simple MAC protocols used by WSNs will suffer in a highly-contended network region, and nodes should adapt to this by reducing their radiated power to reduce the connectivity level. The increased contention created in a dense network will lead to lower overall throughput and slower transmissions in general, if the MAC layer does not manage this. An adaptive MAC layer would still maintain the same performance and throughput, however the number of nodes that are suppressed during the transmission would be greater in a densely deployed network.

Each node starts up at a randomly selected time during the first second of the simulation, in order to eliminate artificial synchronisation effects. This also randomises the order in which nodes appear in each other's neighbour tables, which is the primary method used to control local agent movement, so that agents do not always take the same movement path.

The control application uses the temperature model discussed in Section 4.4.2 - the other two applications do not need the temperature model and do not use it.

5.3 Data Gathering

Data gathering is one of the most common tasks that sensor networks are used for. In the simplest form, the value of a sensor reading at some node in the network must be brought out of the network to a suitable gateway, for the use of the external user. Data gathering is a relatively simple task for a mobile software agent, and the overhead of the agent system will be considerable. However it can serve as a base on which to construct a more complex solution, based on in-network processing or decision making.

Data gathering systems are by their nature centralised – the sensor network serves to get data out of the network to an external user, who uses the data for whatever purpose they require. The speed and reliability with which this data is taken out of the network is critical – particularly if highly dynamic data is being monitored. Critical factors are the time delay from the data leaving the
5.3 Data Gathering

At which it is gathered at the externally connected node, and the amount of data that is contained in each reading, as well as the total data traffic and energy consumption. Single-purpose data gathering systems are deployed for long deployments and must operate at a very low power dissipation.

Two alternative applications are used – Repeated Data Gather and Random Data Gather.

**Repeated Data Gather**  
Repeated Data Gather repeatedly gathers a value from the same node in the system. An external user pushes a data gathering agent into the system. The data gathering node discovers the location of the target node, moves to it, takes a sensor reading and moves back. This is repeated every 5 seconds.

The data gather agent can take a single reading, or a set of readings – the agent can easily transfer either. This evaluates the performance of the system where the agent code is highly likely to remain cached for almost all of the transfers. The performance of the system as a pure data gathering tool is therefore evaluated, without the overhead of repeated code transfer.

**Random Data Gather**  
Random Data Gather uses the same system, except instead of gathering a value from the same node every time, a random node in the system is selected. This forces the middleware to transfer the code to every node in the system. This is the worst-case scenario for a mobile agent as it can not rely on caching of the agent code.

5.3.1 Mobility Model

Agents can move directly to a neighbouring node, or use the carrier move system to move to more remote nodes in the network. There is no penalty for using carrier move – the neighbour list is first checked and if the node is directly reachable, a direct transfer is used internally anyway. Moving over multiple hops requires the determination of the best route to the destination. How this is carried out is a function of the network dynamics – if the topology is regular and fixed, this is a trivial operation, while irregular topologies must determine each route once.
Dynamic, irregular topologies require constant route establishment and updates, introducing overhead into each agent transfer.

The overhead of this mechanism is primarily determined by the routing mechanism. There is a tradeoff between packets transmitted, time to establish a route and memory required to store cached routes. The target application scenarios consider networks with a width of 5-10 nodes, with only slow mobility – nodes are expected to move around the network but slowly enough that each node will remain within transmission range for the duration of a transfer operation (a maximum time of 1–2s). For a 1s transfer, assuming a node starts the transfer 5m away from the destination node and moves directly away from the destination, with a 20m transmission radius, the transfer will complete before the node leaves the transmission radius as long as the node is moving at less than 15m/s, or 54km/h – as the nodes are assumed to be connected to people and not cars or fast-moving devices, this is more than sufficient.

Even if the node leaves the transmission range, the initial transmission will fail, and the system will retry using the remote carrier move mechanism. As nodes are expected to remain in the same physical area for tens of seconds, the routing module caches routes for 30 seconds. If the route is not stored in the cache, a request is flooded out through the network, up to a specified maximum number of hops. A node that is the destination, or that has a route to the destination replies immediately with this response. The routing module waits 500ms for responses – this means that any multihop move over a route that has not previously been established will take at least this long to set up. The evaluation assumes that the MAC layer can provide reasonably low-latency communication. If the MAC protocol incorporates long sleeping periods during which communication is not possible, this will impact on the routing performance. Low-power routing in dynamic networks is an area of active ongoing research and is beyond the scope of this thesis.

The overhead of the routing system can be observed in Table 5.1, containing the timing results for the Data Gather application. The Repeated Read scenario moves an agent to and from the same node in the network repeatedly, while the Random Read scenario selects a random node out of the 20 in the network each time. The values are summarised in Table 5.3. These show the number of packets
## 5.3 Data Gathering

<table>
<thead>
<tr>
<th>Repeat Reading</th>
<th>Packets (First)</th>
<th>Transmitted (Cached)</th>
<th>Time (First)</th>
<th>Taken (s) (Cached)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>114.33</td>
<td>50.15</td>
<td>3.45</td>
<td>0.55</td>
</tr>
<tr>
<td>Per Hop</td>
<td>14.29</td>
<td>6.27</td>
<td>0.43</td>
<td>0.069</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Random Reading</th>
<th>Packets (First)</th>
<th>Transmitted (Cached)</th>
<th>Time (First)</th>
<th>Taken (s) (Cached)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20.11</td>
<td>18.14</td>
<td>0.43</td>
<td>0.44</td>
</tr>
<tr>
<td></td>
<td>57.33</td>
<td>40.60</td>
<td>1.08</td>
<td>0.84</td>
</tr>
<tr>
<td></td>
<td>74.33</td>
<td>37.21</td>
<td>1.77</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td>102.25</td>
<td>16.72</td>
<td>2.83</td>
<td>0.89</td>
</tr>
<tr>
<td>Per Hop</td>
<td>12.39</td>
<td>8.36</td>
<td>0.28</td>
<td>0.17</td>
</tr>
</tbody>
</table>

Table 5.1: Data Gather Characterisation (security enabled)

<table>
<thead>
<tr>
<th>Value</th>
<th>Repeat Reading</th>
<th>Random Reading</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average</td>
<td>S.D.</td>
</tr>
<tr>
<td>Packets (First)</td>
<td>114.3</td>
<td>2.05</td>
</tr>
<tr>
<td>Transmitted (Cached)</td>
<td>50.2</td>
<td>1.27</td>
</tr>
<tr>
<td>Time (First)</td>
<td>3.45</td>
<td>0.14</td>
</tr>
<tr>
<td>Taken (s) (Cached)</td>
<td>0.55</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Table 5.2: Overall Statistics for Data Gather

<table>
<thead>
<tr>
<th>Repeat Reading</th>
<th>Random Reading</th>
</tr>
</thead>
<tbody>
<tr>
<td>Per Hop</td>
<td>Per Hop</td>
</tr>
<tr>
<td>6.27</td>
<td>8.36</td>
</tr>
<tr>
<td>Time per hop</td>
<td>Time per hop</td>
</tr>
<tr>
<td>0.069</td>
<td>0.17</td>
</tr>
<tr>
<td>Number of route operations</td>
<td>17 / 72</td>
</tr>
<tr>
<td>Increase (%)</td>
<td>33</td>
</tr>
<tr>
<td>146</td>
<td></td>
</tr>
<tr>
<td>165</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Routing Overhead
5.3 Data Gathering

transmitted and the time taken (in seconds) for each complete transfer from this application. In this table, the First value refers to the first time the data gather agent visits each node, requiring the complete transfer of all of the agent code. Cached refers to subsequent transfers, where the code will be cached. The values given are from a complete round-trip, from the base to the target and back again – a transfer across 4 nodes therefore represents 8 hops in the round trip.

Random Reading moves the agent to a random node each time, while Repeat Reading always takes the reading from Node 19, which is 4 hops away from the base. Values for individual nodes are not given – the average values for all of the nodes within a certain network distance are given.

The standard deviation is calculated across each set of values, and averaged to give a representative figure, shown in Table 5.2. For the repeat reading, the standard deviation is between 1.8% and 4.2%, while for the random reading it is much greater, at up to 42.4%. This is expected, as the order in which readings are made is hugely significant for the Random Reading system. A node may already have the code cached if it is on the transfer path to a node used for a previous reading, or one of its neighbours may have the code cached, requiring less than the estimated transfer length. While this causes a huge variance in the readings, this is what would be expected in a real deployment.

Randomly selecting nodes almost triples the amount of routing operations that must be carried out, increases the amount of packets generated per hop by 33% and the average time taken to travel one hop by 150% (the actual transfer time is not affected, but the routing operation must complete before the movement operation can take place). The random read operation selects one node every 5 seconds, out of a random selection of 19. Routing is not required for transfers to direct neighbours, leaving 16 nodes that require routing. With a cache expiry time of 30s, a new route must be discovered approximately \( \frac{2}{3} \) of the time, as observed.

Data gathering systems establish the query path as the query is propagated, in a very similar way to that used here. TinyDB floods the query out through the system, using application-specific information if available to optimise this flood. The flood contains the path it has taken to reach the target nodes, and
5.3 Data Gathering

this path is used for the return transfer (this is effectively Dynamic Source Routing or DSR[127]). Directed diffusion[128] uses a similar mechanism, however a reinforcement mechanism is used to promote good quality routes and remove poor routes. This reinforcement is not applicable to the proposed system, as the routes are only used for a single agent transfer and not for a repeated data transfer. Directed diffusion relies on a continuous stream of data to reinforce routes, and will perform no better than an Ad-Hoc system for single, randomly distributed data events.

The attribute-based specification of these data gathering systems has some similarities with the carrier move system—both allow the target of the operation to be specified by description, rather than by specifying exactly which node should be targeted. Database queries can be considered a very simple type of VM, in which the only operations are to retrieve data.

This overhead is contained in every multi-hop move at some point in the move and is an important factor in the performance of the overall system. This operation is not relevant to Agilla, as it assumes a fixed and regular node topology. The route to any node can be calculated from its node id. In order to add support for dynamic, mobile nodes, some form of routing protocol would be needed, with similar overhead to that described here. This would also add considerable overhead to the Agilla transfer protocol, as it can no longer assume that the shortest route from any node to another is immediately available. As shown in Table 5.3, this represents a significant overhead to the system that must be assumed in order to allow the system to handle dynamic networks.

In comparing the native mobile agent system with alternative systems, this is an important consideration—the native agent system makes no assumption about network dynamics or topology, while many alternative systems require a highly constrained physical node setup.

5.3.1.1 Addressing

The mechanism used to address nodes also determines how agents can move around the system. It is assumed that the addressing scheme in terms of node IDs is random. This is a requirement if mobile nodes are to be supported. Agilla
5.3 Data Gathering

requires the network topology to be regular, and pre-set on each node at the time of deployment. The address of each node identifies its position in a regular grid, and routing becomes a trivial operation, allowing agents to move to any node in the system. Clearly, mobile nodes are not supported in any way by this system – this is a severe limitation on the application performance.

The carrier move system removes the dependency on node IDs – an agent can move around the network without ever needing to know the ID of the node it currently occupies. Node attributes are used to control the agent’s movement, however only attributes within the node’s one-hop neighbourhood are available. This is related to the directed diffusion system, where data is moved around the network according to attributes[128].

The fact that only locally available attributes can be used can be limiting. An agent can become trapped in a local minimum or maximum. It is difficult to see how the agent system could overcome this without becoming a complete network-wide data searching system, which is well outside the scope of a mobile agent system. Query-based systems avoid this problem by taking all of the data out of the network for processing, allowing global trends to be observed, at the cost of considerable amounts of data traffic. The node ID in this case serves only to identify where each piece of data has come from. If the node’s location is available, and is coupled with the data, the node ID is not relevant, and does not even need to be globally unique.

Protocols exist to dynamically assign addresses to wireless sensor nodes, using centralised, server-based[129] schemes. An alternative is to only ever assign addresses temporarily[130] – for example, in a query-based system, the address only needs to remain valid for the duration of the query. In any such system, the address of a particular node may change over time. This is only possible in applications in which uniquely identifying any node in the system is not a requirement.

5.3.1.2 Statefulness

The mobility model can either transfer the state of an operating agent or not. If the mobility model is stateful, the executing state must be captured in some
way. As discussed in Section 3.3.1.1, the agent is suspended during a move. This means that only the static and dynamic state must be captured - there is no execution context as the agent is not executing. The middleware does not attempt to capture any hardware configuration such as timers or sensor readings. These are hardware-specific, and cannot easily be translated from one physical device to another. As the middleware cannot prevent the agent from interacting directly with the hardware of the sensor node, this state cannot be transferred directly. While the middleware could attempt to create a seamless transfer in some simple cases, the agent would still be required to deal with cases where this was not possible. Responsibility for communicating with the hardware is therefore left to the agent. The agent must reestablish any hardware operation after moving.

Stateful operations allow the agent to carry its operating state around with it. Without stateful operation, the middleware is simply a code distribution protocol, and separate routing and data transfer protocols are required to move data around the system. Encapsulating the operating state with the agent code simplifies the execution model significantly.

Virtual machine-based systems are better at transferring state from one device to another, as the agent can not interact directly with the hardware. The virtual machine completely encapsulates the agent's state, allowing the system to guarantee complete state capture. As the virtual machine instructions are quite simple the data that must be captured is much easier to store and transfer. This must be balanced against the cost of execution. As shown in Section 5.5.1, many operations that easily implemented in a native software agent are difficult and inefficient to implement using a virtual machine subject to the same resource limitations. Dunkels et al. [15] showed that even for a simple application that can be easily implemented in a VM (a vector convolution in this case) the VM was almost one hundred times slower executing than a native implementation.

### 5.3.2 Energy Consumption

Execution times were measured using minimally-intrusive emulation instrumentation for the Data Gathering application, described in Section 5.3. This simple
agent moves from one node to a target node, takes a sensor reading and moves back. The execution time recorded at the initial node and the destination node are given in Table 5.4.

<table>
<thead>
<tr>
<th></th>
<th>Execution Time at Source (ms)</th>
<th>Execution Time at Destination (ms)</th>
<th>Total Energy (μJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeat Reading</td>
<td>0.421 (S.D. 3.78%)</td>
<td>0.131 (S.D. 8.41%)</td>
<td>11.04</td>
</tr>
<tr>
<td>Random Reading</td>
<td>0.403 (S.D. 3.71%)</td>
<td>0.119 (S.D. 18.1%)</td>
<td>10.44</td>
</tr>
</tbody>
</table>

Table 5.4: Execution Time and Energy

5.4 Location Determination

A common application scenario involves fixed infrastructure sensor nodes working with mobile, user-carried nodes. A key problem in such systems is discovering the physical location of a mobile node as quickly and efficiently as possible. Location determination requires some significant processing and decision making that is not easily implemented in an interpreted agent system with a simple byte-code instruction set. The amount of data required to make this determination and the fact that the initiator of the operation does not know the location of this information means that a lot of data transfer would be required to implement it using a remote-querying system in which data is moved from the point of origin back to the initiator so that it can make the decision. If the target node is highly mobile, the data may also no longer be relevant by the time it has been transferred back to the initiator, particularly if the network is large and the data will have to pass over many hops between the two nodes.

The operation at each node is relatively complex. The decision is taken in the network, to minimise data transfer. The larger the network, the more power and time is saved by not moving all of the data out of the network. Critical factors are the amount of data transferred, the time taken to make the decision and the total energy consumption. Carrying out the decision at the located node removes the requirement that the initiating node have a continuous communication link to the target nodes during the operation, and allows mobile and dynamic node topologies.
5.4 Location Determination

Two agents are used to process the data and to make the decision:

- **Person Location Mobile Agent (PersonLoc)** – The location mobile agent contains the logic to determine the target node’s location. It summons a helper agent for each neighbouring sensor node and sends the helper to that node. The helper sends back the estimate of that node’s location. The location agent then uses those estimates along with the signal strength or quality from the data transmission to estimate its own location.

- **Person Location Helper Mobile Agent (LocHelper)** – The location helper mobile agent is sent a target on starting. It then moves to that target, determines the location estimate of that target sensor node and sends it back to the sensor node that it started on. It waits for confirmation that the estimate has been received and then terminates.

### 5.4.1 Execution Model

The Location Determination Agent was implemented in the Agilla VM. 104 Agilla instructions were required to implement a subset of the actual tracking agent’s functionality.

**Limitations** The Agilla execution environment does not allow the implementation of all of the functionality of the native mobile agent. The limitations of the Agilla agent are:

- Agilla assumes that the node topology is regular and that nodes are addressed according to their position. This effectively eliminates the routing phase, as the optimal route to any node can be determined from its address. The system cannot handle dynamic or irregular topologies and cannot react to node failures en route. The agent assumes that all of the nodes in the system are numbered linearly from 0 upwards, with no gaps (this is effectively a requirement of the Agilla topology setup, and is much less flexible than the native agent). This is a fundamental limitation of the Agilla environment – it does not support dynamic or irregular topologies.
5.4 Location Determination

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move from 19 to 25</td>
<td>850ms</td>
</tr>
<tr>
<td>Move to and from each Neighbour</td>
<td>450ms</td>
</tr>
<tr>
<td>Move from 25 to 19</td>
<td>850ms</td>
</tr>
<tr>
<td><strong>Total Movement</strong></td>
<td>5300ms</td>
</tr>
<tr>
<td>Execution at 19</td>
<td>1.26ms</td>
</tr>
<tr>
<td>Execution at each neighbour</td>
<td>5.45ms</td>
</tr>
<tr>
<td>Execution at 25</td>
<td>29.21ms</td>
</tr>
<tr>
<td><strong>Total Execution</strong></td>
<td>74.07ms</td>
</tr>
</tbody>
</table>

Table 5.5: Agilla Operation Timing

- Agilla can not create and destroy subsidiary agents. The only method of inter-agent communication is the tuple-space, which is not designed for imperative communication such as native agents are capable of. All of the functionality of the system must therefore be contained within one agent, or separated into multiple, almost entirely independent agents that communicate only through the tuple-space.

- Agilla does not have access to the signal strength of the communication between the sensor nodes. This operation is approximated by assuming a constant signal strength for each neighbour.

- The Agilla execution environment does not support string processing. Each room is identified only by a 16-bit identifier (the native implementation uses a 10-byte string).

- The Agilla agent is limited to a maximum of 5 rooms, as 2 heap locations are taken up by each room’s information, while 1 is required for temporary storage during the agent execution.

**Performance** The time this agent will take to perform even this subset of the native agent’s functionality can be determined from the timing information provided. Movement in Agilla takes 225ms per hop. Each instruction takes between 75μs and 330μs, depending on its complexity and how much of the stack it must access.
5.4 Location Determination

<table>
<thead>
<tr>
<th>Operation</th>
<th>Execution at 19 (ms)</th>
<th>Execution at target (ms)</th>
<th>Execution at each neighbour (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raw Value</td>
<td>0.146</td>
<td>13.669</td>
<td>1.306</td>
</tr>
<tr>
<td>Adjusted for helpers</td>
<td>0.146</td>
<td>6.99</td>
<td>1.306</td>
</tr>
<tr>
<td>Agilla</td>
<td>1.26</td>
<td>29.21</td>
<td>5.45</td>
</tr>
<tr>
<td>Increase</td>
<td>763%</td>
<td>318%</td>
<td>317%</td>
</tr>
</tbody>
</table>

Table 5.6: Agilla Agent Location Timing

The total execution is summarised in Table 5.5, assuming the operation is started at node 19, moves to node 15 which has eight neighbours and moves back afterwards. 5.3s is taken up by movement for every location operation (even after the first), while 74ms is taken up by agent execution.

The execution time of the native mobile agents was also measured. This is shown in Table 5.6. The Raw Value is the actual total time measured. However, the native version creates a helper agent to move to each neighbour, instead of moving itself. Agilla does not have an equivalent operation, and the limited functionality of the Agilla agent does not duplicate this. The time taken to create the helper agents is excluded from the native timings, giving the adjusted value.

Even though the native agents have much greater functionality, they are still substantially faster than the Agilla agents – the interpreted agents are between 317% and 763% slower. This demonstrates the performance benefit from using the proposed native mobile agent system, even for simple tasks, and particularly for computationally-intensive tasks.

5.4.2 Energy Consumption

Table 5.7 shows the total time spent and the total number of packets transmitted for each repetition of the location determination operation. Location determination involves creating a mobile agent and moving it to each of the node’s neighbours, to evaluate its location and signal strength. On average, each node has 6.3 neighbours in its neighbour table at any time, and there are agent transfers per location operation. Note that the times recorded are not just for the transfer operation – they are for the entire transfer, operation at the destination, sending results back and terminating the helper agent. The operation completes 5.9 times
5.4 Location Determination

<table>
<thead>
<tr>
<th>Operation</th>
<th>Total Packets</th>
<th>Total Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Run</td>
<td>185.1 (S.D. 8.7%)</td>
<td>5.78 (S.D. 15.2%)</td>
</tr>
<tr>
<td>Subsequent</td>
<td>32.1 (S.D. 15.4%)</td>
<td>0.99 (S.D. 24.2%)</td>
</tr>
<tr>
<td>Difference</td>
<td>83%</td>
<td>83%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Agilla Packets</th>
<th>Agilla Time (s)</th>
<th>Packet Time Difference</th>
<th>Time Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Run</td>
<td>226.8</td>
<td>2.8s</td>
<td>+22.5%</td>
<td>-51%</td>
</tr>
<tr>
<td>Subsequent</td>
<td>226.8</td>
<td>2.8s</td>
<td>+606.5%</td>
<td>+183%</td>
</tr>
</tbody>
</table>

Table 5.7: Location Application Operation Results

more slowly when code must be transferred than when the code has already been cached. The standard deviations are given for the measured values. Due to the large number of separate operation involved, there is considerable variance in the values observed. Routing operations in particular, are only carried out occasionally and add 0.5s to the time taken and generate up to 50 packets. This accounts for much of the variance in observed values.

The previous section discusses an alternative implementation of this application in Agilla in more detail. Ignoring the time spent actually executing the agent, the Agilla version would require 12.6 one-hop agent movements at 225ms per hop, taking 2.8 seconds. Agilla transfers its program in 22-byte blocks of code. It also uses one packet for the register state, one packet for every 4 heap values and one packet for every 4 values on the stack, acknowledging every packet as it is received. Assuming a minimal value of one stack and one heap packet, the 104-instruction Agilla program requires 18 packets per one-way transfer.

The Agilla version that does not implement the entire agent functionality, and neglecting execution time therefore is 51% faster and uses 32% fewer packets on the first operation, but is almost 3 times slower and uses 7 times as many packets for subsequent transfers. The values for the first and subsequent operations in Agilla are the same as Agilla does not cache agent code.

The native version also includes in these totals the overhead of transferring the helper agent from the agent server to the area of interest in the first totals. This is measured as taking 1.356s, and using a total of 61 packets. As the Agilla version does not contain an equivalent operation, the calculations of Table 5.7 are
5.4 Location Determination

<table>
<thead>
<tr>
<th>Operation</th>
<th>Total Packets</th>
<th>Total Time (s)</th>
<th>Agilla Packets</th>
<th>Agilla Time (s)</th>
<th>Packet Difference</th>
<th>Time Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Run</td>
<td>124.13</td>
<td>4.42</td>
<td>226.8</td>
<td>2.8s</td>
<td>+82.7%</td>
<td>-37%</td>
</tr>
<tr>
<td>Subsequent</td>
<td>32.06</td>
<td>0.99</td>
<td>226.8</td>
<td>2.8s</td>
<td>+606.5%</td>
<td>+183%</td>
</tr>
</tbody>
</table>

Table 5.8: Location Application Adjusted Results

repeated in Table 5.8 with this part of the transfer excluded. The Agilla version now uses twice as many packets, and is only 37% quicker for the first operation, with the same results for subsequent operations.

For example, using the values previously established, then if the system moves once per second, and has an execution time for the native agent of 5ms per node, with an average packet length of 2.5ms per packet, with the native agent executing in 17.8% of the time of the interpreted system, then the native system will on average consume 0.617mW of power, while the interpreted will consume 1.387mW - 2.25 times as much. Alternatively, if an agent moves every 10 seconds and the native version requires 20ms of execution time, the native version will consume on average 91.6μW, while the interpreted version will consume 307.3μW - 3.35 times as much. This does not include idle listening at the MAC layer.

5.4.2.1 Network Traffic

In addition to the time spent executing, the number of packets generated is another indicator of energy consumption. The total number of packets required to implement the various methods of carrying out the location determination operation was calculated, as a function of the number of hops between the initiator of the request and the target, and shown in Figure 5.1.

The number of packets is one measure of the power consumed during the transmission, although the actual value depends on the MAC protocol. The number of packets generated is an indicator of the expected network congestion caused by the operation. Congestion causes backoffs and collisions which greatly increase the time spent carrying out operations and increase the amount of energy consumed.
5.4 Location Determination

For anything over one hop, the native agent implementation is the most efficient, once the code has been cached. Agilla is substantially higher, and the first native agent move higher again. As expected, the TinyDB model is the most efficient for small networks, but as the size of the network increases, this becomes less and less efficient. Above 11 hops, Agilla out-performs TinyDB.

As long as an agent is expected to visit each node more than once, the native software agent therefore provides the most power-efficient implementation for all but the simplest of networks. Figure 5.2 shows the number of packets per visit, assuming a network size of 7 hops. If the operation is only to be carried out once, Agilla or TinyDB generate fewer packets, and so are more power efficient. For two visits, all of the approaches generate a similar amount of packets, while as long as the agent is expected to visit each node more than two times, the native approach generates the least traffic.

In order to calculate this, a number of assumptions were necessary. The packet values for the native and Agilla version have already been established in previous sections. For the TinyDB implementation it was assumed that 6.3 neighbours are accessible on average, as before. It is assumed that one flood of the entire network to discover nodes is required, and after this that communication with each neighbour node of the target is required, and that each neighbour can report all of the data it must send in one packet, and that the various routes required are cached after the initial network flood, for the duration of the query. It is assumed that the network size is 9 times the number of hops (i.e. 27 nodes for the original 3-hop scenario, 90 nodes for a 10-hop scenario).

In general, network traffic increases localised congestion. If no other operation is taking place this may not be significant as the role of each endpoint in the transfer is well-defined and they should never both attempt to transmit at the same time, however if multiple agents are in the system, or traffic is also being generated by other sources, this becomes important. The more traffic that is generated, the higher the likelihood of packet collisions, as the radio medium is occupied for longer. This increases the expected time taken taken for any communication operation, increasing the expected energy consumption.

The native mobile agent system generates the least traffic for large, dynamic systems. Under these conditions it is best able to cope with multiple agents.
and applications, reducing the overall system energy consumption when multiple processes are attempting to use the network.

5.5 Reliable Distributed Control

Distributed control is an area well-suited to mobile agent systems, in wireless sensor and actuator networks. Distributed control systems must operate autonomously and reliably, across a varying network topology. They must communicate with potentially complex sensors and actuators, and must carry out whatever data processing is required to generate the control response.

The alternative is an externally managed system. The amount of data transfer (and energy) required to move all of the data out of the system, and to send control decisions back into the network grows exponentially with network size.
An externally managed network also requires a continuous communication link from the external management system to all of the sensor nodes in the system. The interface between the sensor network and the external management system acts as a single point of failure.

The distributed system is autonomous and reliable, and will continue to perform with reduced functionality, even after complete node failures. This allows a user to start the control system and then to leave the system unattended, knowing that it will continue to function despite all but the most catastrophic failures. The agent functionality necessary to implement such a system is complex, and requires a sophisticated execution environment.

The native mobile agent system provides such an environment, coupled with low energy consumption. The flexibility of the native mobile agent system allows such a system to be implemented in a number of different ways. This gives
the application developer the freedom to target the application at the desired performance point. In order to guarantee reliability, each agent monitors other agents in the system. Mobile agents expect to encounter static agents at certain nodes in the system. If they do not find a static agent in the correct place, one is created. Conversely, static agents expect to be visited by mobile agents periodically. If a period elapses, large enough to rule out any normal operating delay, a new mobile agent is created that rediscovers the system. This also allows the system to bootstrap itself from an initial state with no knowledge of the network setup.

A reliable, autonomous system capable of carrying out a reasonably complex operation such as this would be difficult to implement efficiently using interpreted or scripted mobile agents, and would suffer in terms of energy consumption. A static system could perform some of the control operations, but would provide little benefit in terms of energy consumption once a protocol was established to reliably transmit data around the network, and would not have the same benefits in terms of reliability and decentralisation.

Complete local failures result when the wireless sensor node is no longer able to usefully communicate with the rest of the network. This may stem from hardware misconfiguration or resource exhaustion. In either case, the sensor node is unlikely to be able to usefully recover from this situation and should reset itself, allowing other agents in the system to restore it to its correct role. This is carried out using a watchdog timer.

Mobile Agents The system is implemented in three different ways, to evaluate the flexibility of the mobile agent middleware and to evaluate the relative merits of each method. The reliability guarantees of each system are discussed in Section 5.5.3. Every 10s a control decision should be made using fresh data from every node in each room. The aim of the control system is to keep the system as close as possible to the desired setpoint. The desired setpoint for each room is stored by each agent – this can be changed using the blackboard system, and is set to 20 degrees for this evaluation.

The agent systems are arranged in order of increased reliability. Reliability is measured as the ability of the system to continue functioning after the loss
of nodes or agents. A quantitative measure of the reliability of the system under certain conditions and over a given time period is the average availability. This can be obtained by taking the expected value of the number of nodes participating in the control process over that time period. Section 5.5.3.2 plots the reliability of each agent system over 10,000 seconds for varying node failure rates.

Reliability is improved by distributing the decision-making process, by separating the application functionality into multiple agents and by giving the various agents knowledge of the expected behaviour of other agents in the system, allowing them to recover each other if the expected behaviour is not observed.

The simplest system uses only a single agent. If this agent, or the node on which it is running fails, control operations cease. This system consumes the least resources as it is the simplest, but performs the worst in terms of reliability, as decision-making is centralised, and only a single agent is used. The reliability is improved by separating the control operation from the network discovery and data gathering operations. As three separate agents are used, they are able to restart each other if the expected agent activity is not observed. However, the control operation is still centralised at certain nodes. The final system is the most reliable. The control operation is completely decentralised and each node with a sensor can function on its own, without any other agents visiting. Mobile agent recovery is randomly distributed, removing centralised points of failure. This requires the most resources, as the system is more complex, but also performs best faced with node and agent failures.

- Single Agent

  - Single Control – A single agent carries out all of the control and management operations. This agent moves around the network, dynamically discovering all of the nodes in the network. It visits each node in turn, keeping a list of which nodes have been visited already. If any neighbouring nodes are unvisited, the agent moves to them first. If not, the first unvisited node is visited, until all have been visited, at which point it starts again. At each node, it takes a sensor reading and makes a control decision if it is connected to a heater.
5.5 Reliable Distributed Control

The agent stored the sensor reading at each node. Whenever it reaches a node connected to a heater, it makes a control decision. A sum of the difference between each temperature reading and the setpoint is taken. If the resulting sum is positive (i.e. the temperature is greater than the setpoint), the heater is switched off. If it is between 0 and 1.5° below the setpoint, the heater is set to 'Low', between 1.5° and 3° it is set to 'Medium' and at more than 3° it is set to 'High'. This proportional control is a standard algorithm for linear systems with slow dynamics, however more complex algorithms can be implemented, subject to the processing constraints of the target hardware[131].

- **Active Collection**

  - Active Discovery – The *Active Discovery* agent moves slowly around the network, discovering the network topology. If an unknown node is discovered in the neighbour list of a visited node, it is visited and added to the system if possible. The *Discovery* agent starts a *Static Control* agent at each node connected to a heater, and updates the node list for the *Static Control* agent at each visit.

  - Static Control – The *Static Control* agent has two purposes – it runs the control operation at each node, and it ensures the *Active Discovery* agent continues to operate. Every 10 seconds, a *Data Gather* agent is created and given a copy of the layout of the current room. This gathers sensor readings, which are used to make a control decision. The time between visits of the *Active Discovery* agent is monitored. If 60 seconds pass without a visit, a new *Active Discovery* agent is created, which begins a new network discovery operation. The separation of the discovery and the static control operation means that the control algorithm continues executing even if the discovery agent is not available. This provides increased reliability.

  - Data Gather – The *Data Gather* agent is given the layout of a particular room in terms of node IDs by the *Active Discovery* agent. The *Active Discovery* agent updates this as the topology changes. It moves
around each node in the room, takes a sensor reading from each, returns to the initiating node and terminates. This provides separation of the operations of data gathering and control operation, simplifying the two agents. If a move to any node fails, that node is ignored for the current operation.

- Distributed

  - Distributed Discovery – A Distributed Discovery agent is used, almost identical to the Active Discovery agent used in the previous scenario. It discovers the network layout and starts a Static Monitor at every node it visits. Each Static Monitor is told which nodes contain heaters in the room it occupies. This is updated at each visit.

  - Static Monitor – Static Monitor agents are not mobile. They create Decision Package agents periodically, with a uniformly distributed interval, averaging ten seconds. A control decision is carried out at the current node, and the result of this is conveyed to the Decision Package agent. The control operation is therefore completely distributed – the loss of any node in the system will only result in that node being excluded, and all other nodes will continue to function as before.

    A random selection of Static Monitor agents act as reinforcement agents, with a probability of \( \frac{1}{5} \) per node. If the Static Monitor agent is configured as a reinforcement agent, it monitors the time since it has been visited by a Distributed Discovery agent, and creates a new one if 60 seconds have elapsed.

  - Decision Package – The Decision Package agent moves to the heater node, and implements the decision that has been made at its initiating node. It then terminates. The net result of a decision package arriving at a random time from each node in the room, is an average heater setting of the average decision value, assuming the decision function is linear. If any move attempt fails, the agent terminates.

The control system executed correctly – a sample output is shown in Figure 5.3. This plots the average temperature difference from the setpoint, across
5.5 Reliable Distributed Control

The 25 sensor nodes over the duration of the 360 seconds simulated for one particular run. The initial large error can be seen, decreasing as the controller activates the heaters. It then goes into a controlled state, with the heaters activated periodically as the temperature drops below the setpoint. Slight disturbances to the average temperature can be seen at 120 and 240 seconds as the door opens and closes, causing the temperature in the area to drop.

![Total Temperature Difference](image)

Figure 5.3: Average temperature difference

5.5.1 Execution Model

The control agent was also implemented as an Agilla agent. As before, there were a number of limitations in the system that could be implemented due to the restrictions of the Agilla execution environment. 157 instructions were required to implement this subset of the native agent functionality – this is over \( \frac{1}{3} \) of the total instruction capacity of a node running Agilla.

The time this agent will take to perform the functionality it is capable of is again determined from the Agilla timing information.

The total execution is summarised in Table 5.9, for each visit to each node, 225ms is taken up by each one-hop move, while the average hops per move is 1.6, giving an average movement time per visit of 360ms. The average execution time is 9.27ms (this assumes that on average, each loop within the agent completes half of its iterations).
5.5 Reliable Distributed Control

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Movement per hop</td>
<td>225ms</td>
</tr>
<tr>
<td>Average hops</td>
<td>1.6</td>
</tr>
<tr>
<td>Total Movement</td>
<td>360ms</td>
</tr>
<tr>
<td>Total Execution</td>
<td>9.27ms</td>
</tr>
</tbody>
</table>

Table 5.9: Agilla Agent Location Timing

The benefits of the native mobile agent system are clearly demonstrated here. The limitations on the system implementation that are imposed by the interpreted system considerably reduce the functionality of the application, and yet even such a limited subset of the native agent functionality requires over $\frac{1}{3}$ of the resources available to the virtual machine. Interpreted systems on such limited devices are restricted in their functionality and are not suitable for implementing complex, reliable and autonomous applications.

5.5.1.1 Limitations

The limitations of the Agilla implementation were as follows:

- As before, Agilla assumes that the nodes are in a regular grid. It can not deal with an irregular or dynamic network topology. The agent assumes that the entire system it is controlling is located in a linear node address space starting at 0, and the agent must be given the size of the network in advance.

- Agilla can not use subsidiary agents, so all of the functionality must be provided in a single agent. Agilla agents can not communicate with each other (apart from indirectly) and can not create other agents. This is a serious limitation as it constrains the ability of Agilla to develop complex systems of multiple cooperating agents.

- Due to the limited storage ability of an Agilla agent, the agent can not keep a list of which nodes have been visited and the sensor reading from each node. The agent simply keeps a moving average of the last 10 sensor readings. At each node visited, this is reduced by 10% and the value at that node is added. This acts as a low-pass filter – in particular, it will be slow
to respond to rapidly changing conditions (as a portion of each previous reading will continue to be contained in this total for a long time after each reading has been taken). As the agent has only a 12-element storage heap, and must deal with multiple rooms and nodes this is a necessary compromise.

- The agent is limited to four different rooms, again due to the storage restrictions of an Agilla agent.

- The agent can not dynamically follow the network topology – it always moves in a linear sequence through the node ids. It can not optimise this sequence by preferentially visiting neighbours. The computation required to determine which node to visit is too complex for the Agilla execution environment, and as the agent can not track where it has been it can not store this information to inform itself that a certain node has been recently visited.

- In general, the error handling of the Agilla agent is limited. If a particular transfer fails, the agent will continue to retry the transmission until it eventually succeeds – complex error and failure handling in a simple environment such as the Agilla execution environment will substantially increase the agent size and slow down the agent’s execution.

5.5.2 Movement Latency

Movement rate is an important factor in tracking dynamic and fast-moving processes. The required movement rate depends on the physical density of the sensor nodes and on the movement rate of the process being tracked. For example, if the sensor nodes are deployed 10m apart, the maximum movement rate at 6 moves per second is 60m/s, or 216km/hr. Allowing a 33% margin of error, the system can not reliably follow a physical event moving faster than 40m/s, or 144km/hr. The radio on the MicaZ node has a theoretical outdoor range of over 100m, suggesting a maximum rate with a widely deployed network of 1440km/hr, although this is unlikely to be observed in practice. This is sufficient for all but the most
5.5 Reliable Distributed Control

<table>
<thead>
<tr>
<th>Operation</th>
<th>Single Agent</th>
<th>Active Collection</th>
<th>Reliable Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup Time (s)</td>
<td>44.4 (SD 2.0%)</td>
<td>51.2 (SD 12.9%)</td>
<td>82.36 (SD 4.0%)</td>
</tr>
<tr>
<td>Total Packets</td>
<td>11083 (SD 1.7%)</td>
<td>22941 (SD 9.0%)</td>
<td>23667 (SD 7.5%)</td>
</tr>
<tr>
<td>Discovery Visit Period (s)</td>
<td>7.82 (SD 7.3%)</td>
<td>6.51 (SD 17.7%)</td>
<td>5.87 (SD 10.4%)</td>
</tr>
<tr>
<td>Discovery Exec Time (ms)</td>
<td>1.21 (SD 2.1%)</td>
<td>0.88 (SD 1.0%)</td>
<td>1.07 (SD 17.5%)</td>
</tr>
<tr>
<td>Static Exec Time (ms)</td>
<td>n/a</td>
<td>0.61 (SD 12.3%)</td>
<td>0.56 (SD 0.78%)</td>
</tr>
<tr>
<td>Gather / Package Exec Time (ms)</td>
<td>n/a</td>
<td>1.31 (SD 11.6%)</td>
<td>0.56 (SD 0.8%)</td>
</tr>
<tr>
<td>Utilisation (%)</td>
<td>96.3 (SD 0.4%)</td>
<td>95.3 (SD 10.3%)</td>
<td>98.4 (SD 5.8%)</td>
</tr>
</tbody>
</table>

Table 5.10: Distributed Results

Esoteric applications and supports the thesis that mobile agent systems are a viable technology on wireless sensor networks.

Movement times for the control agents are given in Table 5.10. _Start Time_ measures the time the system takes to discover all of the nodes and incorporate them into the control system. This varies from 44.4 seconds for the simplest, _Single Agent_ system where only one agent is active, to 82.36 seconds for the _Distributed_ system – which must visit each node twice – once to establish the static monitor and the second time to report the room layout. _Utilisation_ measures how many sensor readings are taken into account for each control decision at a node connected to a heater. This is expressed as a percentage of the total number of nodes.

This system is considerably more complex, and multiple agents contend for the radio resources, however a movement rate of between 3.2 and 4.25 hops per second is observed. This allows a monitoring agent to visit each node in a 25 node network every 5.9–7.8 seconds. As the primary duty of the monitoring agent is to monitor node failures, which are assumed to be relatively uncommon occurrences, this is more than a sufficient rate. If the energy consumption of this monitoring agent is a concern, it can be instructed to wait a certain time at each node, reducing the average movement rate and therefore the energy consumption.

Alternative virtual-machine based approaches have a larger movement latency.
5.5 Reliable Distributed Control

Agilla states a value of 225ms per hop, which is slightly faster than the first transmission but much slower for subsequent transmissions, once the executable code has been cached. Agilla does not separate the agent code from its state, and so does not benefit from code caching at intermediate and previously visited nodes. It does not use the permanent storage available on the node, but instead stores the entire agent in RAM. The values measured for the Data Gather agent also include the security manager – Section 5.7.3 shows that a 10%-45% speedup could be generated by disabling security.

The movement latency for code-based transfers is the most significant limiting factor in the performance of the native mobile agent system. Code caching eliminates this for subsequent transfers, and in applications where agents are expected to revisit each node many times, this is only an issue the first time. This is considered in more detail in Section 5.4.2.1.

5.5.3 Reliability

At the middleware level, each agent transmission is guaranteed to either succeed or fail completely. Timeouts and retransmissions are used in the transmission protocol to ensure reliability. If either endpoint aborts the transmission or if the communication link fails before the transmission has completed, it will eventually time out. A three-way handshake is used at the end of the transmission to ensure that the agent does not start at the destination until it has been stopped at the source.

This is an integral part of the transfer operation and it is difficult to quantify the effect it has on the middleware performance. The primary source of inefficiency is on the timeouts before retransmission. The middleware waits between 80 and 160ms for a response before retransmitting a packet. Due to the latency introduced by the MAC and physical layer at each end of the connection, it does not perform well with smaller values than this. However in the absence of competing traffic and interference, an entire transmission can take place in under 80ms. As interference is often caused by a competing transmission, shorter retransmission periods risks saturation of the physical medium.
5.5 Reliable Distributed Control

An agent that tries to move to another node is only suspended at the target node. This allows the middleware to resume the agent if the transmission fails. A message is sent to the agent informing it that the transmission has failed, and any timers or readings that that agent had started are resumed. Any messages generated during the move attempt are lost. While it is possible to keep these messages, it was found that after a prolonged transmission attempt the agent was forced to deal with a large number of meaningless messages (particularly timers firing) at startup time, considerably delaying its return to normal activity. The memory overhead of storing all of these messages is also considerable.

Virtual Machine-based systems can easily suspend and resume agents, as each operation is handled directly by the middleware anyway. As less data is transmitted, there is much less chance of a transmission partially completing. Maté[11] avoided this problem by restricting agent sizes to the maximum that could fit into one packet. Agilla was not able to do this, as agents can reach hundreds of bytes[1]. A simple control protocol is used. Agents are only transferred one hop at a time, and per-packet acknowledgements are used. Timeouts and retransmissions are again used to guarantee the operation completes in a consistent state, although they do tolerate duplicate agents in some cases. WiseMAN[23] uses a similar approach to Agilla, breaking the transmission into multiple packets and using per-packet acknowledgements and timeouts to manage the transmission. This is a design choice and could be eliminated at the expense of more protocol overhead at the end of the transmission.

As the proposed native middleware requires some initial negotiation between nodes to establish whether or not the agent code has been cached, it must adopt a transaction-based transfer system. This is extended to cover the completion of the transfer, requiring an acknowledgement of the transfer before the agent is started at the destination. Agilla and WiseMAN do not need such a complex agent transfer, as the entire agent is always transmitted. They can simply transmit a stream of data packets, waiting only for a simple acknowledgement of each packet before continuing.
5.5 Reliable Distributed Control

5.5.3.1 Application-level Reliability

Application level reliability uses the middleware-level reliability in conjunction with the agent and application design to ensure reliability. This is shown by the Reliable Distributed application example. This presents three different implementations of the same system.

The simplest implementation, Single Agent uses a single agent to carry out all of the work in the system. This agent is created in the system at the start of the simulation, and there are no external monitoring agent. This is vulnerable to node failures. There is no guarantee of reliability in this system – if the agent or the node containing the agent fails, all operations in the system will stop, and will not resume until restarted by some external entity.

The Active Collection system is inherently reenforcing. Once setup, the static Static Control agents monitor and reenforces the operation of the Discovery agent, while the Discovery agent monitors and reenforces the Static Control agents. The loss of either agent type will be recovered by the other. Data Gather agents are temporary and only exist long enough to take a set of readings – the loss of one will only compromise the current set of readings. The next agent will gather new readings.

If a Static Control agent is lost, or the node containing it resets, control operations will cease until the next visit of the Discovery agent. If a Discovery agent is lost, control operations will continue but the network topology will not be updated until the Discovery agent is recreated and rediscovers the network layout.

The Distributed system is very reliable. If any node in the system is lost, only that node’s effects are removed from the system. New nodes are quickly added to the system, and will immediately begin acting (within 15s, as measured in Table 5.11). The reinforcement of the mobile agents is constant, randomly distributed and moves around the network as the mobile agents operate. The heater nodes are the most critical – however no agent operates permanently at the heater node, so that even if it resets it will continue to operate in the system. The loss of a Decision Package agent only loses the single decision value contained in that agent.
The variability and complexity of these systems shows how the native mobile agent system extends the scope of features that can be provided over existing systems. Previous deployments of mobile software agents\cite{1, 16, 23} have only been able to produce simple, single-purpose agents. The complexity of the native agents demonstrated here is a novel development in the programming of embedded wireless sensor networks and enables reliable and autonomous operation that was not previously possible.

5.5.3.2 Reliability Measure

This reliability can be measured by the average availability of nodes over time, using the different applications, in a numerical simulation. Node failures are assumed to be independent poisson processes (i.e. exponentially distributed). The probability of node failure during each second is varied from 0.01 to 0.0001. It is assumed that mobile agents can move 5 times per second, and that a failed node can reset itself and is available for reprogramming after 10 seconds. A system of 25 nodes is simulated, for a total of 10000 seconds.

For comparison, a static system is also simulated. This system starts out operating on every node in the system, and once a node fails, the static system is no longer available.

A sample run of each application is shown in Figure 5.4, for a probability of failure of 0.001. This shows the total number of nodes available at any time, for each application. As expected, the static application decays away until all nodes have failed. The monolithic (single agent) system has almost all nodes available, until eventually the agent is killed by the node on which it resides failing, after which no nodes are in the system (as there is no control agent). The reliable and distributed systems keep almost all of the nodes available – the big drops for the distributed control application occur when a static control agent is killed – it is eventually recovered by the mobile discovery agent.

The results for repeated runs of each application, for varying values of failure probability are shown in Figure 5.5. The availability is calculated as the total number of nodes actively participating in the control system at any time. The maximum value is 25, while a value of 0 means that the entire system has stopped.
5.5 Reliable Distributed Control

Node Availability
over Time, Pf=0.001

Figure 5.4: Sample node availability, P=0.001

working. Secondary effects are not modelled – for example if 50% of the nodes have failed in a sparsely deployed network it is possible that there will be no communication path between the remaining nodes.

The standard deviations of the values vary considerably. The static system is a random decaying process, with an average standard deviation across the various values of 13.3%. The Monolithic (single agent) system is highly variable, as it depends on a single random value. Once the agent is at a node that fails, the availability immediately drops to zero. The standard deviation is as expected a very high 75.1%. The reliability and self-stabilisation of the Datagather and Distributed systems make them much more consistent, as the system is isolated by the self-recovery from the random effects of node failures. The average standard deviations are 0.27% and 0.12% respectively.

Table 5.11 shows this recovery process operating. The node setup was altered from the standard setup for a number of simulation runs, to test the resilience
5.5 Reliable Distributed Control

![Availability vs. P(failure)](image)

**Figure 5.5:** Average node availability

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Late Start</td>
<td>15.3s</td>
</tr>
<tr>
<td>Reset Test</td>
<td>15.9s</td>
</tr>
</tbody>
</table>

**Table 5.11:** Failure Recovery

and self-management capability of the system. Nodes 6, 17 and 19 were kept shut down and not started until 3, 4 and 5 minutes respectively.

This demonstrates that cooperative agent-based reliability mechanisms are powerful and effective. Previous work with more powerful nodes and agents demonstrated such effects on larger systems[49; 74; 108], however the design and deployment of such systems in agent-based embedded wireless sensor networks is a novel development.

### 5.5.3.3 Self-Stabilisation

With reference to the definition of stabilisation presented in Section 2.4.3, the distributed control system is self-stabilising as long as a dynamic discovery agent
remains available somewhere in the network. The distributed system is the easiest to examine, but similar results hold for the others. Once it has settled into its operating state, it is self-stabilising under the following condition:

- At least one static agent at a node is in a reinforcement state. Reinforcing agents are selected with a $\frac{1}{8}$ probability, and constantly updated by the discovery agent, and any reinforcement agent is capable of recreating a discovery agent that will recover the entire system, so this reduces to:

- All of the nodes containing reinforcement agents are not in a failed state simultaneously.

For example, if the probability of a node failing in a given second is $10^{-5}$, and failure recovery takes 15s (as measured in Section 5.5.3.2), then the probability of all three reinforcing agents failing at the same time, assuming the failures are independent, is approximately $2.25^{-8}$. As the network size increases and more reinforcing agents are added, or if the proportion of reinforcing agents was increased, this would decrease considerably.

5.5.3.4 Reliability Overhead

Each move is only attempted once in all of the above agents (internally, the middleware sends the move requests a number of times and resends packets as necessary, however once this reaches a limit, the move is aborted). In a dynamic application such as this, it is better to ignore and later return to a node that is uncontactable than to spend a long time trying to reach it at the expense of other nodes.

The results of the distributed control system are shown in Table 5.10. The movement times, as expected, are significantly higher for a system where there is real contention for the radio resources. Start Time measures the time the system takes to discover all of the nodes and incorporate them into the control system. This varies from 44.4 seconds for the simplest, Single Agent system where only one agent is active, to 82.36 seconds for the Distributed system – which must visit each node twice – once to establish the static monitor and the second time to report the room layout.
5.5 Reliable Distributed Control

As established in Section 5.5.3.2, distributed systems with multiple reenforcing agents are reliable, and can recover the system from even large-scale failures. This requires each agent to be aware of and duplicate some of the operations of others. It also requires more agents than a simple, unreliable system. The cost of this reliability can be seen from the *Total Packets* measure – the reliable systems generate approximately twice as many packets over the simulated time period as the unreliable single agent. This is a measure of the total number of packets generated by all of the nodes in the system combined. This must be traded off against the required reliability – highly reliable and recoverable systems necessarily require higher energy consumption. The optimal level is a design choice and depends on the application requirements – if a utility function is specified then it is possible to work out exactly the optimal tradeoff between expected utilisation and energy consumption.

The reliability that is introduced by such a system has not previously been demonstrated in embedded wireless sensor network platforms, without centralised management systems operating outside of the sensor networks. In-Motes[16] demonstrated a reliable agent-based system, however the network was externally managed and required physical intervention a number of times during the deployment. The SelfWISE project[132] demonstrated algorithms for self-stabilisation, however this work is focused on the algorithmic level and has not been demonstrated on actual wireless sensor nodes. Autonomous, self-management is an important property that is required as sensor networks move into long-term deployments in hard or expensive to access locations.

*Utilisation* measures how many sensor readings are taken into account for each control decision at a node connected to a heater. This is expressed as a percentage of the total number of nodes. As no interference is being introduced and nodes do not fail in the simulation, the utilisation is high – above 95% in all cases. The only reason for a sensor not to be considered is if localised congestion does not allow an agent to visit it at the appropriate time. The *Reliable Distributed* agent has the highest utilisation at over 98% – as the decision making and distribution is completely decentralised, even localised congestion can only remove a small number of nodes from the system.
This overhead is introduced to provide reliability - this is shown by the two tests of Table 5.11. In one case, a node recovers from being reset, in the other, a node is started up late and introduced into the system. In each case, after just over 15 seconds, the node has returned to full functionality in the network.

Data-centric query systems cannot be compared to mobile agent systems, as they are neither expected to operate autonomously nor capable of autonomous operation. The complexity of the system required to implement this makes it impossible to implement in any existing interpreted agent system for comparable hardware platforms. The interpreted agents of such a system do not contain sufficient storage, they cannot express suitably complex decision algorithms efficiently, they cannot create, monitor and communicate with other agent instances, and they cannot operate completely autonomously, as the reliable agents in this system do. Reliability such as this has not previously been established in mobile agent systems for embedded wireless sensor networks.

5.6 Agent Characteristics

The sizes of the various agents described in previous sections are given in Table 5.12. The code size of the compiled agent, the static state size and the number of lines of code are given. The executable code may be cached at any node in the system, the static state in addition to any dynamic state is transferred every time. The number of lines of code required, while not an exact measure, is representative of the complexity of the agent.

The number of lines of code given in Table 5.12 is the total number of lines in the file excluding formatting, comments and state declaration - it represents only the number of lines of code that will actually be compiled into executable code.

5.7 Single-Agent Tests

Some aspects of the system are best observed without any application executing. This provides a baseline performance metric on which to base application analysis.
5.7 Single-Agent Tests

<table>
<thead>
<tr>
<th>System</th>
<th>Agent</th>
<th>Code Size (bytes)</th>
<th>State Size (bytes)</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Gathering</td>
<td>Data Gather</td>
<td>390</td>
<td>8</td>
<td>36</td>
</tr>
<tr>
<td>Location</td>
<td>PersonLoc</td>
<td>858</td>
<td>10</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>LocHelper</td>
<td>370</td>
<td>6</td>
<td>35</td>
</tr>
<tr>
<td>Distributed Control</td>
<td>Control</td>
<td>684</td>
<td>43</td>
<td>68</td>
</tr>
<tr>
<td>Single Agent Reliable</td>
<td>Single Control</td>
<td>1246</td>
<td>16</td>
<td>106</td>
</tr>
<tr>
<td>Active Reliable</td>
<td>Active Discovery</td>
<td>1220</td>
<td>14</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>Static Control</td>
<td>644</td>
<td>9</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>Data Gather</td>
<td>916</td>
<td>13</td>
<td>88</td>
</tr>
<tr>
<td>Distributed Reliable</td>
<td>Distributed Discovery</td>
<td>1190</td>
<td>14</td>
<td>109</td>
</tr>
<tr>
<td></td>
<td>Static Monitor</td>
<td>428</td>
<td>10</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>Decision Package</td>
<td>300</td>
<td>4</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 5.12: Agent Sizes

5.7.1 Carrier Move

In order to measure the saving in agent size created by the carrier move system, an agent was implemented both with and without using the carrier move system. Some of the functionality of the carrier move system was moved into the agent itself. This does not replicate the complete carrier move system in terms of reliability and error-checking – it is a basic movement system. The Routing Move agent uses a routing module to find the best path to a destination, duplicating the carrier Move Towards functionality. The Blackboard Move agent uses the blackboard system to follow a particular blackboard gradient to its peak.

The sizes of these agents are shown in Table 5.13. All values are in bytes. The two agents with the carrier are the same size – only the move condition is changed. The increase in agent size without the carrier is 244 and 78 bytes – the blackboard conditional remote read allows the agent to easily find the highest value in the region, reducing the overhead for the Blackboard Move agent.

<table>
<thead>
<tr>
<th>Agent</th>
<th>With Carrier (bytes)</th>
<th>Without Carrier (bytes)</th>
<th>Increase (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing Move</td>
<td>614</td>
<td>858</td>
<td>244</td>
</tr>
<tr>
<td>Blackboard Move</td>
<td>614</td>
<td>692</td>
<td>78</td>
</tr>
</tbody>
</table>

Table 5.13: Test Agents
5.7 Single-Agent Tests

The carrier move system therefore reduces the size of agents that wish to perform non-local moves considerably, as well as reducing the execution time (and therefore energy consumption) as the agent does not execute on each intermediate node. This saving in agent size translates directly into a saving in energy consumption due to agent code transfer, while the use of the carrier to perform multi-hop moves is more efficient than performing it one hop at a time, reducing the energy consumption of later agent moves.

5.7.2 Broadcast Move

The proposed middleware system improves on existing module distribution protocols for embedded WSN operating systems, and in order to do so it must duplicate their functionality. The middleware is capable of pushing an agent out to every node in the system, while also validating the agent image with the Security Manager, as described in Section 3.3.6. This performs the same task as Sense[133; 134] and Sluice[64], for the smaller agent-based data set.

Figure 5.6 demonstrates the time taken to push three agents of varying sizes out to 25 nodes, arranged into a regular grid, with each node within communication range of its immediate neighbours. The node in one corner of the grid is connected to an agent server which initiates the transfer, so the agent must travel a maximum of 4 hops. The average total time elapsed from the start of the operation to the agent starting on every node within a certain range is shown. The standard deviation of the values ranges from 0.15% to 7.3%, with an average of 2.1%. As the size of the agent and the network are significant factors, broadcast performance is measured using a measure of seconds taken per kilobyte of code transferred per hop (s/kB/Hop). The agent sizes chosen represent the range of agent sizes encountered in the agent evaluations, ranging from a few hundred bytes to approximately a kilobyte.

Fitting a line to these graphs as a function of the agent size yields a fixed overhead per hop of 0.222s, with a size-based component of 1.134s/kB/Hop. The data given fits this line with a standard deviation of 6.04%. This compares very favourably with Sluice[64] and Deluge[63]. These systems are optimised for much larger code images, however the observed programming rate in a network of the
same size is 17.15s/kB/Hop for Deluge and 19.21s/kB/Hop for Sluice. Sense
does not report overall programming time, however it is also based on Deluge so
similar results would be expected.

![Node Coverage Time](image)

This demonstrates that the proposed native mobile agent middleware at least
matches existing network-wide reprogramming systems, and in fact exceeds their
performance for the small updates encountered in module- and agent-based pro-
gramming. While the system is capable of considerably more complex behaviour,
existing wireless sensor network applications and operations can still take place,
with the same functionality that is provided by existing network reprogramming
systems.
5.7.3 Movement Latency

In order to generate a best-case transfer scenario, a simple scenario was used to eliminate multi-hop and interference effects. Five nodes are evenly spaced in a straight line, with each node only within the transmission radius of its two neighbours. The propagation model is the default AvroraZ model[114] which models path loss and fading in a typical indoor environment, using the calculated total SNR at the receiver to calculate whether or not a transmission succeeds.

The simulation was run for 360 seconds – at measured transfer times of 40–100ms, this represents multiple thousands of transfers, giving a reliable estimate of the average. The initial startup delay and fetch of the mobile agent from the agent server takes 14 seconds – the system is therefore active for 346 seconds. This simulation was run both with and without the security manager enabled, to measure the overhead of security.

In order to measure the performance in a medium suffering contention, interference was generated by causing each node to send useless packets periodically. These packets contain 20 bytes of payload (42 bytes total). The packet transmissions follow a Poisson distribution with \( \lambda \) varying from 90 to 1440 (or the average inter-packet time varying from 0.25s to 4s). The number of times the agent is started is measured over 346 seconds. As the agent performs no action other than moving to each neighbour, this represents the minimum possible transfer time.

The average time taken by each move is shown in Figure 5.7. The limiting factor is the capacity of the radio transmission and processing system. For example, with the security manager enabled and 4 packets per second per node of interference generated, each node sends 4 packets and receives 6.4 packets of interference (the end nodes receive less interference). The standard deviation of the values in the graph varies between 0.16% and 4.4%.

In the absence of retransmissions, each agent move requires 6 packets to be transmitted (according to the protocol defined in Section 3.3.4). If a packet action is defined as the act of sending or receiving a packet, as this represents the time the node is unable to perform any other radio-related task, each agent move generates at least 12 packet actions at each endpoint. At the measured value of 2.456 moves per second per node (each move involves two nodes), this gives
5.7 Single-Agent Tests

a minimum of 39.8 packet actions per second per node. This does not include retransmissions due to collisions – it is a lower bound for the system.

In fact, a value of 42 packet actions per second was measured. Although the physical layer radio interface can handle this data rate, the operating system and microcontroller in the sensor node struggles to deal with this rate, while also allocating memory, performing code memory writes and all of the other operating system tasks it must attend to. In addition, the measured value does not include packets that were so badly corrupted that they were not identified as valid packets by the radio – only packets that at least begin to be received, and subsequently succeed or fail are recorded.

This is supported by the increase in movement rate for lower values of interference. Without interference, 4.10 moves per second per node were measured, giving a minimum packet action rate (without collisions) of 49.2 packet actions per second. In fact, 51.9 packet actions per second were measured. This is an increase in the number of packets of only 23%, despite a 67% increase in the number of successful transmissions.

The measured value of 51.9 packets sent or received by each node per second, with packet sizes of between 42 and 88 bytes plus a 10 byte preamble, gives an overall datarate of between 21.6kbps and 40.8kbps.

While the raw data rate of IEEE802.15.4 is 250kbps[19], a wireless sensor node will never achieve this. After the packet is received at the physical radio layer it must be transferred to the microcontroller over the SPI interface. SPI operates at a clock rate of 4MHz, however one interrupt is triggered per byte. Each interrupt has a latency (including hardware and software) of approximately 7μs, so an 88 byte packet, with 10 byte preamble (which is not transmitted over SPI) requires

\[
\frac{98x7}{250000} = 3.136\text{ms at the radio physical layer, followed by } 88\times \left( \frac{8}{1000000} + 7 \times 10^{-6} \right) = 0.792\text{ms to transfer the packet to the microcontroller.}
\]

The minimum time per packet at the microcontroller interface under these conditions, for packet sizes of 42 and 88 bytes is therefore 1.664ms + 0.378ms = 2.042ms and 3.136ms + 0.792ms = 3.928ms.

The MAC protocol will send the packet immediately if the radio medium is free, however if it is not, it backs off for a minimum of 5ms, with an exponentially increasing backoff if the radio medium does not become free. Even if a packet
is successfully sent, there is no guarantee the packet has been received as hidden terminals may still cause collisions. If it is not received, the protocol timeouts of at least 40ms (described in Section 5.7.4) must expire before a retransmission occurs. Apart from the data transfer, all of the protocol packets are sequential – each endpoint must wait for a packet to be received before taking any action. This produces worst-case behaviour as any packet loss results in large protocol delays.

This processing time at the physical/MAC layer is matched by the processing that must be carried out at the middleware layer. At a minimum, memory must be allocated and a handler called for each packet, at both sender and receiver. As the packets involved are protocol packets in a relatively complex agent transfer, these handlers are large and involved multiple memory and hardware accesses, taking hundreds of μs. A FLASH memory write must be performed at least once per transfer, to start or stop an executing agent. This takes 4.1ms. Assuming a 1ms per packet processing overhead and distributing the FLASH memory access over all 12 packets to achieve an average, gives an assumed 1.342ms per packet processing overhead.

The resulting times per packet are therefore 4.282ms and 6.998ms, giving an overall best-case throughput, assuming no contention or collisions of 78.468kbps or 100.600kbps. The measured rates of 21.6kbps and 40.8kbps represent 27% (for a 42-byte packets) and 40% (for an 88-byte packet) utilisation. This is reasonable in a simple CSMA MAC protocol.

Attempting to increase the interference rate beyond 4 packets per node per second quickly causes the transfer rate to drop as nodes are overwhelmed by the amount of data traffic and exhaust the available memory. As the nodes detect that they are close to memory exhaustion, they will refuse any agent transfers, and the agent will remain on whichever node it last reached. As detailed above, the nodes are operating at between 27% and 40% of the maximum system throughput with no interference. A value of 12 42-byte interference packets per node per second (i.e. 4 packets sent and 8 received), which are discarded after they are received, consumes only $2.94ms \times 12 = 35ms$ of the available transmission and processing time, however the resulting collisions and backoffs introduce timeouts at the MAC layer and the middleware layer of between 5 and 40ms, per contention or collision.
Agilla suffers from the fact that it uses smaller packets and acknowledges each packet in turn. Larger packets could speed this up, however this would require more complex management. The Agilla transfer management system would require considerable modification to support a more efficient transfer protocol. When the application code has been previously cached, there is no interference and security is disabled, the transfer time is as low as 42ms per hop. This is much faster than Agilla (at 225ms per hop[1]). The radio and MAC layer requires about 5ms to accept, transmit and process a packet – so even with just a single packet acknowledgement, 10ms per hop is the minimum achievable value by a static distribution protocol. The increase over this minimum is more than justified by the rich execution environment created by the mobile software agent system.

This demonstrates that native mobile agent systems are a viable technology
5.7 Single-Agent Tests

In embedded wireless sensor networks, as discussed above, assuming a hybrid MAC protocol, the primary energy consumption is due to idle listening during agent transfer. Assuming both nodes involved in the transfer use the CC2420[25] are in receive mode for the 42ms duration of the transfer, this gives an energy consumption of 2.37mJ at each end, or a total of 4.74mJ per one-hop agent transfer.

5.7.3.1 Code Transfer

While the mobility model constrains the type of movement possible, the movement latency is a fundamental limit on how fast an agent can move around the network. In addition to the basic latency, a mobility system may be restricted by limits on simultaneous operations.

The lowest recorded value for movement latency in Section 5.7.3 is 42ms, the lowest value with security enabled is 61ms. This is in a very simple scenario, with no other operations taking place in the network, with an agent that does nothing but move between neighbouring nodes.

This includes all of the tasks involved in moving – from carrying out routing if required to setting up the move, transferring and verifying data, writing the state, linking the agent, starting the execution of the new agent and closing the connection. This also includes any timeouts and retransmissions. The recorded rate is between 6 and 14 hops per second. This includes the overhead of routing which can be considerable. The increase in time and packets for the random read compared to the repeated read is mostly made up of the routing phase.

5.7.4 Transfer Parameters

There are three main parameters involved in the agent transfer – Agent Send Timeout, Agent Receive Timeout and Agent Protocol Timeout. In order to allow implementation on a wide range of hardware and software, the values chosen are generic and are not tied to any particular networking system or hardware. These timeouts should be large enough that they are not unnecessarily triggered when a packet is only delayed by congestion or OS delays, but small enough that the system does not wait much longer than necessary when packets are lost. They
should also be large enough that the system and the physical radio medium are not overwhelmed by the rate of packet generation.

- **Agent Send Timeout** is the period at which the sending node generates data packets. This must be large enough that the MAC layer is not overwhelmed by the amount of data that is generated. Each data packet has a 64-byte payload, with 3 bytes of protocol overhead, 13 bytes of OS overhead and at least 6 byte of MAC overhead, giving at least 86 bytes. Queueing such large packets will quickly consume available memory, and the system must guard against this. The sender will automatically drop packets if the available memory approaches exhaustion, relying on the feedback mechanism to later recover these packets. The receiver must also verify the packet, including any security check and write the data packet to the FLASH memory within this period.

- **Agent Receive Timeout** is the time that the receiver waits to receive a data packet before sending a NACK. Clearly this can not be less than the **Agent Send Timeout**. Values close to the **Agent Send Timeout** will create unnecessary NACK packets, as any long MAC-layer delay will result in the timeout being exceeded.

- **Protocol Timeout** is the time the receiver waits before retransmitting a protocol packet. Every packet apart from data packets are considered protocol packets. In general, protocol packets should be immediately acknowledged, so the delay should reflect a maximum expected value of MAC and OS-level delay. Note that as the timer is started at the sending node as it sends a protocol packet, the timeout should reflect the maximum expected time to send the packet, including all MAC-level delay at the transmitter, to receive and process the packet, and to transmit a response back, including all MAC-level delay at the receiver.

The **Agent Send Timeout** is only significant when the code has not been cached at the receiver, or when the agent state exceeds 64 bytes. Each packet of agent data received requires 4ms at the physical layer (from Section 5.7.3), a potential...
5.7 Single-Agent Tests

code memory erase and write (as an example, this requires 8.2 ms on an AT-Mega128L microcontroller[117] and a security operation requiring at least 7.5 ms, but measured at up to 18.9 ms (from Section 5.7.3), on top of all the management operations carried out by the middleware. With no other process active in the network, each packet therefore requires 31.1 ms, in addition to the middleware processing time. The value of 80 ms is 2.5 times this minimum value, to allow for other agents consuming microcontroller resources and for network traffic delaying the transmission. While a small transmission will spill into the packet queues and eventually clear the system at a less aggressive rate, this does not scale to larger agent transfers, where the agent size is a significant portion of the available memory.

Part of the design rationale of the middleware system is that it should coexist with other applications on the nodes of the network, including applications that have no knowledge of the existence of the middleware or of mobile agents. Using 100% of the system's resources for the transfer would severely degrade the performance of such applications, and the middleware avoid this wherever possible.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Send Timeout</td>
<td>80</td>
</tr>
<tr>
<td>Agent Receive Timeout</td>
<td>160</td>
</tr>
<tr>
<td>Agent Protocol Timeout</td>
<td>80</td>
</tr>
</tbody>
</table>

Table 5.14: Protocol Parameters

The values chosen for normal execution are shown in Table 5.14. 80 ms is the shortest period at which packets can be reliably moved through the system, including all MAC delays, OS delays, security check and writes to the FLASH memory. The protocol timeout is also kept low, as there is less processing to do for protocol packets and the remote system should respond immediately.

A random jitter of 10% is applied to the timeouts. For example, the Agent Send Timeout is randomly distributed between 76 and 84 ms, with a new value chosen for each transmission. This is to prevent artificial synchronisation between competing transmissions. The CSMA-MAC protocol used in this evaluation is optimised for random packet arrival times – if two sources generate packets with
5.7 Single-Agent Tests

exactly the same period, this can lead to sub-optimal performance. Randomising the transmission period slightly is a simpler solution than exploiting cross-layer information to deal with unusual network conditions.

The protocol timeouts are primarily determined by the underlying MAC layer. The system aims to be independent of lower-level considerations, and so must use a course-grained timeout approach, instead of more accurate cross-layer information. A particular implementation could be modified to consider such information if it was available. There are two primary considerations – not to overload the underlying MAC and radio layers, and not to create a packet flood that will reduce the overall throughput in times of high contention. The values chosen have been shown to create a stable and reliable transmission system, with good transfer rates in times of low congestion (as low as 40ms), while maintaining acceptable performance in times of high congestion.

The adaption of these parameters is more important than the starting protocol parameters. The initial parameters represent a good choice for a lightly-loaded system, so the system adapts primarily by backing off in the presence of high losses or retransmissions. This reduces congestion, allowing multiple devices to contend for the physical medium and creates greater overall throughput. Backoff is only carried out per-transmission. Congestion state is not stored across transfers, and cross-layer information is not available. One of the design assumptions of the system is that the network topology is random and dynamic, meaning that stored information quickly becomes obsolete. There is also a strong feedback mechanism that will confuse congestion estimation – transferring a mobile agent over tens of packets per second will turn a lightly congested area into an area of high congestion.

The transmission layer is highly dependent on the underlying MAC layer, however in the absence of cross-layer feedback, the transmission layer must necessarily take a coarse-grained approach to data transfer, based on realistic expected performance, with simple adaptation to manage congestion. Cross-layer information will improve the performance of the system, however implementing this in a standard manner would require a complex networking stack that is outside the scope of this work. The approach taken is simple, allowing implementation on a wireless sensor node without consuming excessive resources.
5.7 Single-Agent Tests

5.7.5 Security

As shown in Figure 5.7, for low levels of interference, where each transfer executes without retransmissions, the security system imposes an overhead of 45%. The time per move is 42ms, giving an overhead of 18.9ms. As the interference increases, the security overhead remains fixed. At 4 packets per node of interference per second, the security system imposes an overhead of 10% on a transmission time of 93ms, or 9.3ms. With such a high interference rate, the time taken to calculate the signature becomes less relevant, as the signature will often be retransmitted anyway. The amount of data in such a transfer is quite small, however the signature calculation must process at least 3 64-byte blocks per signature – consisting of the secret key, the application data (using a standard padding mechanism to pad it out to a multiple of 64 bytes), and a final standard closing block.

Passing and Dressler measured the time taken to perform various cryptographic operations on wireless sensor nodes[62]. They recorded a value of approximately 40ms per kB for the MD5 hash operation. This would give an expected time of 7.5ms on a sensor node with nothing else running. In addition, the data must be read out from the FLASH memory, and the signature packet transmitted. Sense[133; 134] and Sliice[64] report similar overheads for verification of received code images.

The generated signature is embedded into the packet used to close the connection – increasing the size of this packet by 64 bytes. In terms of power, the overhead is this additional 64 bytes of packet data, adding approximately 2ms of active time to the radio transmission, and the 18.9ms of additional processing time measured above. At 20mW active power at the microcontroller and 33mW active power at the radio, this is an additional 444µJ of energy per transmission.

5.7.6 Execution Model & Energy Consumption

The AtMega128L microcontroller operates at a clock speed of 8MHz, with each instruction taking between 1 and 3 clock cycles to complete (125–375ns). Each instruction occupies two bytes, so a mobile agent of hundreds of bytes such as those used in this evaluation contains a few hundred instructions.
5.7 Single-Agent Tests

In order to evaluate the time spent executing an agent, the system was instrumented to record the time at which an agent starts and stops executing. This is only an approximation of the execution time, as the agent does not have exclusive use of the microcontroller. Due to the message-queueing architecture of SOS, the agent has not actually started to execute anything when the measurement starts – the agent starting message has been placed into the queue and will be executed when the middleware returns control to the scheduler, if no higher-priority messages are awaiting execution. This means that the measured value is not a precise reflection of the actual time spent executing, however it is representative of the time a real agent will take in a normal system.

The only actions the agent takes is to record which node it is executing on and which direction it is travelling in. It determines when it should move to next, and carries out the move. Once the move request is received, the agent is suspended for the move and the measurement completes.

The distribution of the execution time is shown in Figure 5.8. Execution times fall within a tight range from 69μs to 80μs, with an average of 72μs, which is also the median value. The figures are normalised to give a total of 1.0.

An equivalent agent was written in Agilla. The minimum execution time was 2.625ms, 36 times the native version, and the agent required 35 instructions. Actively executing at 8MHz, the ATMega128L, a typical microcontroller on a wireless sensor node, consumes approximately 20mW of power. This gives an average energy consumption per agent of 51.5μJ for the Agilla version, and 1.41μJ for the native version.

Interpreted schemes such as Agilla require between 75μs and 300μs per operation, in contrast to the measured times of hundreds of μs measured for entire agents in Table 5.4. Each instruction is limited to simple stack manipulation and input/output operations – complex operations must be built up from multiple instructions. A firetracker example required two agents, one of 50 bytes, one of over 100[1], at hundreds of μs per instruction. Table 5.6 shows a comparison of execution times for similar agents written natively and in Agilla – the native version is between 3 and 7 times faster (and therefore lower power), even though it has substantially more functionality than the interpreted agent. WiseMAN[23]
5.7 Single-Agent Tests

Execution Time Distribution

![Graph showing execution time distribution with bars for different execution times.]

Figure 5.8: Agent Execution Timing

does not give a detailed energy evaluation, but states that the performance is similar to Agilla.

This demonstrates that the native system has clear performance benefits over all existing mobile agent systems for the same class of hardware. The more complex the application, the more processing time is required, and the greater the performance benefit from using a native mobile agent over an interpreted system.

5.7.7 Long Term Lifetime

Energy consumption is the primary factor in the lifetime of a wireless sensor network application. The energy limitation of an embedded device is what distinguishes it from normal application programming. All of the factors considered above feed into the energy consumption. Energy is consumed by the hardware
on which the agent and middleware operates – it is impossible to quantify energy consumption without referring to specific hardware. The MicaZ is a common wireless sensor network platform, and it is used as the basis of any energy calculations presented here, as it is representative of the state of the art in wireless sensor nodes. Similar results could be expected from any other contemporary platform. The MicaZ contains an ATMega128L microcontroller, operating at 8MHz. At this speed, the microcontroller dissipates 20mW of power while active. The sleeping power dissipation is 15µW. The CC2420 radio dissipates 33mW while transmitting at -10dBm, 56.4mW while receiving, 1.2mW while idle and 60µW while in power-down.

![Lifetime vs. Movement Period](image)

**Figure 5.9: Expected Lifetime, Te=0.1**

Other peripherals such as external sensors may also dissipate some energy
5.7 Single-Agent Tests

- however this is system-specific and is not a function of the system software. Assuming the system is sufficiently capable to manage the peripherals correctly, the energy required to take a single reading is constant, no matter what software system is in place.

The relative energy consumption of different systems is therefore a function of the radio activity and the active processing time. Radio activity is a function of the number of packets generated – this is considered in Sections 5.3.1 and 5.7.3.1. Processing energy consumption is a function of the execution model and the application programming. Three systems are compared – the proposed native mobile agent system, Agilla and SensorWare. The energy consumption values used are those established in the previous section for the native system and Agilla, and the values given by the designers of SensorWare[14] of 430mW active energy consumption and 0.47mJ per remote script transfer. It is assumed that the radios of the various devices are idle in between transfers, and that during an agent transfer, the radios of both nodes involved are continuously in receive mode, and the processors are active. This is an approximation, however receive mode power dissipation is greater than transmit mode, so this will tend to over-estimate the energy consumption slightly. Execution times are those previously measured, of 5ms for the native agent, 28ms for Agilla and 15ms for SensorWare[14], representing a reasonable complex operation.

The energy storage is provided by a 2400mAh battery, there are 40 nodes in the system which are visited sequentially, in a repeating sequence. The agent performs its execution task periodically at the current node, either once per second or ten times per second, and moves to the next node at a rate which is varied from once every second to once every 25s. The first move of the native system to each node requires a code transfer, subsequent moves do not. It is assumed that all other energy consumption in the system is included in an average 0.5% active duty cycle[7]. In particular, it is assumed that there is a mechanism to wake up the nodes from sleep in order to initiate a transfer. This will likely require a low-power MAC protocol, capable of switching to a high-throughput mode for the agent transfer, however the energy consumed by this will be equal across every platform. The overhead of the underlying operating system is assumed to be 0.5% – SensorWare is likely to have a much greater overhead as the embedded
Linux system on which it is built is not designed for ultra low-power operation in the same way as a WSN operating system.

The expected network lifetimes, before the batteries begin to be exhausted for an execution period of 0.1s and 1s respectively are shown in Figures 5.9 and 5.10. The influence of the execution time can clearly be seen. In both cases, SensorWare quickly exhausts the available supply. Its powerful processor quickly depletes the available energy.

For the system that only executes once per second (Figure 5.10, for large values of the movement period, the system becomes dominated by idle power dissipation, and both the native system and Agilla return similar lifetimes. For faster movement, as expected, the more efficient native system has over twice
the lifetime. For the system that executes 10 times per second (Figure 5.9), the difference in execution time is much more prevalent. The native system has 3 times the expected lifetime for a fast moving agent, and still has almost double the lifetime for a slowly moving agent.

5.8 Conclusions

This chapter has shown that the native mobile agent middleware is a viable and powerful system for dynamic and mobile application development. While the overheads of the system are considerable, they are more than compensated for by the power and flexibility of the system, enabling complex, dynamic and autonomous applications.

5.8.1 Mobility Model

The mobility model supported by the proposed middleware architecture is a reliable, weak, network-aware, node- or attribute-based stateful model. This can be compared to existing agent and reconfiguration systems.

The agent transfer protocols are designed to move agents as quickly as possible, without saturating the physical radio medium. This allows the network to spend as little time as possible actively transmitting or receiving, and as much time as possible in a low-power sleeping state. The mobility model allows agents to take movement decisions out of the agent, and minimises the transfer latency and the number of packets generated. This reduces overall energy consumption and speeds up the transfer time, allowing the agent to follow and manage quickly changing events. In a location determination scenario, the native system was proven to reduce overall network traffic by 18% without code caching and by 83% with code caching, reducing contention and retransmissions. The system was shown to support systems with tens of agents executing simultaneously on each node – an application requiring more agents than this will probably benefit from a more capable hardware platform.

The middleware system was shown to compare favourably with existing solutions in a representative set of application scenarios. As the network becomes
larger and the required operations more complex, the native mobile agent system emerges as a clear leader in energy consumption, both with respect to data processing and radio packet generation. This demonstrates that the native agent system is a powerful and flexible application platform. It out-performs existing mobile agent systems for embedded wireless sensor nodes when the network is large, dynamic and the application is required to be autonomous and reliable – exactly the target application space for mobile agent systems.

The proposed native mobile agent middleware is not the best solution for simple, static applications executing with the assistance of external programs or users (although it can fulfil such a role if needed). The overhead and complexity of creating a system based on mobile agents is not justified when the target application is not mobile or is not intended to operate autonomously. The native mobile agent system is a multi-purpose system, and it can not expect to out-perform purpose-built data gathering systems in that role, nor can it expect to compete in processing ability with off-line data processing using more powerful hardware. However if the application requires in-network processing, mobility and flexibility and an ability to operate autonomously, then the proposed native mobile agent system provides an execution environment and supporting middleware that is more powerful than any existing system for mobile agents in embedded wireless sensor networks, and that is capable of reliable and autonomous operations that are not possible in such hardware devices using any existing mobile software solution.

5.8.2 Movement Latency

Agent transfer between nodes is dominated by the communication characteristics – the most time-consuming operation in any transfer is the actual transmission of data between different nodes. While the physical radio interface operates at up to 250kbps, the management protocol, processing and storage requirement of the complex agent transfer operation add to this considerably.

The middleware architecture is designed to operate on a wide range of platforms, however there is an assumption that the MAC layer can provide relatively low-latency communication (of the order of 10s of ms). The implementation would
5.8 Conclusions

require some tuning to function in a communication environment with potentially long communication delays (100s of ms to seconds). A hybrid MAC will provide a good compromise – the initial agent transfer request can still be transmitted using the standard high-latency, random-access communication. Once all nodes involved have accepted the transfer request, they can then switch to a high-power, low-latency MAC for the duration of the agent transfer. As the agent transfer has a clearly defined endpoint, they can switch back to low-power listening once this has been achieved. Idle listening is the major source of energy consumption in most wireless sensor network applications. The less time the system spends in a high-power, high throughput MAC mode, the less energy is consumed.

The time spent listening (i.e. the movement latency) is a more important factor than the amount of data transmitted – for example, the CC2420[25] 2.4GHz radio transceiver, a common device used in wireless sensor networks, dissipates between 25.5mW and 52.5mW while transmitting, depending on the output power, and 56.4mW while in receive mode. Transmitting a packet in fact consumes less energy than receiving, and the primary factor determining how much energy is consumed during a transfer is how long both nodes involved must spend listening for packets.

5.8.3 Execution Model

A number of factors are evaluated in order to quantify the differences in execution model between the proposed middleware system and existing systems for wireless sensor nodes. The execution model is always subject to the constraints of the hardware platform, and all of the analysis here is carried out with respect to an embedded wireless sensor node. The ability of a system to carry out particular operations depends both on the speed and capability of the system. Raw execution speed constrains the ability of the system to perform complex operations, while the limitations inherent in an execution system can prevent the system from carrying out certain operations at all.

The execution model presented here is powerful and flexible. This allows complex operations that can not be efficiently implemented in existing mobile agent systems for wireless sensor networks. The use of this powerful, native
execution model with mobile software agents is the primary improvement made by
the work presented in this thesis over existing mobile agent systems in embedded
wireless sensor networks, and it enables the implementation of the reliable and
autonomous systems described in later sections.

In general terms, programming virtual machines such as Agilla is difficult. The
instruction set is extremely simple, and programming it is very similar to
assembly language programming, but with an even smaller register file than all
but the simplest microcontrollers. While the stack-based environment allows
simple operations to be efficiently represented, more complex applications are very
tedious to represent. In theory, any expression can be translated into a stack-
based representation, however in practice the number of instructions required
quickly becomes very large. For example, 43 out of the 104 instructions in the
Location Determination Agent, or 41% of the agent, is made up of instructions
that do nothing but manipulate the stack – either pushing constants, swapping
or copying values on the stack or popping values off the stack.

Structured programming is almost impossible. Conditional and explicit jump
instructions are the only way to implement flow control, while a return address
can be explicitly pushed into the stack to implement a function call. Loops, condi-
tional blocks and general program flow control must be explicitly implemented
by the programmer using multiple instructions. Conditional jumps offsets are
limited to within 15 instructions of the jump instruction – an expression that
can not be expressed within this limit must be split into a complex multi-part
operation. While a higher-level language could be translated into VM instruc-
tions, the limitation on the number of instructions will severely limit such an
implementation.

The 12-element program heap is the only storage space available, apart from
extremely complex stack-based storage which is only possible in limited circum-
stances. This limits the amount of information that an agent can carry around
with itself – it is forced to infer its operating conditions from local state rather
than from stored information. For example, the control agent can not carry
around a list of nodes to visit in a particular room, or store which nodes have
previously been visited as this information would be too large for the 12-element
heap.
Atomicity can not be guaranteed across multiple instructions. This applies particularly in neighbour lists and sensing – for example, there is an instruction to read the number of neighbours, and to read out a particular neighbour, but the middleware can not guarantee that a new neighbour has not been detected in between the operation of the two instructions. This is also an issue when multiple cooperating agents are used – the problems of attempting distributed operations without a guarantee of atomicity are well-known[135; 136]. The addition of atomic guarantees would complicate the interpreter considerably, and the simple instruction set would require a worst-case, coarse-grained implementation. The native execution model allows fine-grained control of atomic sections and concurrency.

Handling node failures, system failures or unexpected operations is very difficult without hugely increasing the agent size. Multiple instructions are required to check a return value, and a simple retry counter combined with a test of the return value will consume up to twenty instructions. Adding such a check to every operation will massively inflate the agent size.

Similar problems were encountered by the developers of SensorWare[14], who found it very difficult to generate a non-trivial system that could be implemented in both existing interpreted systems for embedded wireless sensor nodes and in their TCL-based scripted system. Sensorware provides an execution environment that is comparable in power and flexibility to that of the native system. Its scripted system and high-level programmability make it easier to develop in, however this comes at a cost of much greater power consumption.

WiseMAN[23] simplifies the programming model by not even attempting to be a general-purpose programming system – the agent is specified as a collection of rules, operators and conditions. The language used to express these is difficult to manage and the size of each agent is severely restricted. WiseMAN agents are restricted to a certain class of problems that can be easily expressed by a set of simple conditional operations.

In the microcontroller used in this evaluation with 128kB of FLASH memory, at least 50kB of FLASH memory is available for agent storage. As the largest agents encountered in testing are just over 1kB, this is enough space for 40 of these agents. Each agent type only occupies this storage space once – multiple
instances of the same agent do not require additional space. Agents that are stored but not executing still take up space in the FLASH memory – these can be released if the memory usage grows too large. Resource constraints of this magnitude are common across WSN platforms and any device with substantially greater resources will be constrained by its operating power dissipation.

The biggest limitation on executing agents is the available RAM. The basic operating system and middleware consumes a total of about 2.6kB of RAM, including static and dynamic allocation. The ATMeta128L microcontroller used in this evaluation has 4kB of RAM, leaving approximately 1,400 bytes available in a normal deployment, with no agents running. Other embedded microcontrollers have similar memory sizes. The memory required by all of the operating agents must therefore fit into this space. This includes all allocated memory, including agent state, dynamically allocated memory and packets queued for transmission. If this approaches 100% utilisation, it may not be possible to allocate memory to send or receive packets. The middleware will stop accepting incoming agents if the memory utilisation exceeds 90% in order to prevent complete memory exhaustion which could compromise the node’s operations. The middleware also requires 15 bytes per agent of metadata and agent information.

Tests have been run with up to 20 agents per node executing simultaneously without any problems – as long as the memory limit is not exceeded, the only effect is that the processing time is split between 20 agents instead of one. A system requiring hundreds of agents to coexist on a single node will not be possible, and would require a more capable hardware platform. The limitation on the number of agent types coexisting could limit the ability of the system to function in an environment where very many agents of varying functionality exist, however if so many large and varying agents are required, the requirements of such an application might well require a more capable hardware platform anyway. The mobile agent system is capable of supporting a system with tens of agents executing simultaneously. This is the maximum level of multi-tasking possible on this hardware platform – Contiki, SOS and TinyOS are all similarly constrained.

In order to guarantee reliable and autonomous operation, an agent must be capable of communicating directly with other agents in the system in order to determine their status and it must be able to create new instances of itself or any
other agent. It must have sufficient storage to carry enough state information to be able to determine whether or not a certain portion of the network is functioning correctly, and what action should be taken if it is not. It must be large enough and must execute in an environment powerful enough to express the algorithms and code necessary to carry out such decision-making, and it must be able to move around the network to gather enough information to allow it to make the decision. In order to work around failed nodes and recover from failures, it must not use a fixed network topology, but must be capable of dynamically adjusting its view of the network topology. It must be able to do all of this, while also carrying out its primary application role.

The limitations of an Agilla agent can clearly be seen from Section 5.5.1. Even for the relatively simple agents considered in this section, the Agilla agent can not reproduce the functionality of either of the native agents. The programming model is tedious and limited, and the resource restrictions of the execution environment quickly become apparent. The actual execution time of the native agent is of the order of 1ms, while the Agilla agent requires a minimum of tens of milliseconds per node to carry out a limited subset of the functionality. WiseMAN[23] displays similar performance to Agilla and an even more restrictive execution environment.

### 5.8.4 Energy Consumption

Energy consumption is the critical limiting factor in the useful lifetime of a wireless sensor network. It turns out that the native mobile agent middleware system in fact reduces energy consumption and increases reliability compared to a data gathering or virtual machine-based agent system in all but the simplest scenarios. It was demonstrated that transfer times increase slightly compared to an interpreted system without code caching, but when code caching can be exploited, reductions in transfer times of up to 80% are achieved. This reduces idle listening, which is the primary determinant of energy consumption in an embedded wireless sensor network.

The expected network lifetimes for a set of sample applications clearly show the influence of the execution time. For a system that only executes once per
5.8 Conclusions

second, for large values of the movement period, the system becomes dominated by idle power dissipation, and both the native system and Agilla return similar lifetimes. For faster movement, the more efficient native system has over twice the lifetime. For the system that executes 10 times per second, the difference in execution time is even more pronounced. The native system has 3 times the expected lifetime for a fast moving agent, and still has almost double the lifetime for a slowly moving agent.

This demonstrates that the native mobile agent system is a viable alternative to existing mobile agent systems for both embedded wireless sensor networks and larger devices, when the available energy is limited. This is an important result, as it demonstrates that the system can be deployed in application areas where it is difficult or expensive to access the nodes. The overhead of native mobile agents is largely eliminated by code caching, and the efficient transfer protocols allow it to out-perform existing systems in dynamic applications.

5.8.5 Security

The native mobile agent system uses a signature-based security mechanism. This ensures that only authorised agents can enter the system. There is an overhead to this system, and it can be disabled if not needed.

Existing mobile agent systems for wireless sensor networks do not incorporate security in any way - they completely trust any correctly-formatted data received over the radio interface. While signature-based verification is not novel in wireless sensor networks and systems such as Sense[133; 134] and Sluice[64] have incorporated similar systems for large, whole-image updates, it has not previously been implemented for frequent, agent-like transfers. This requires an efficient implementation, and the overhead introduced is established in the following section. This allows the application designer to decide whether or not the overhead of security is justified by its benefits.

This system ensures that a malicious user can not inject an agent into the system, however they can still consume significant system resources. The system is unable to process transfers from other nodes while it is receiving the malicious agent, and a malicious user can effectively isolate a node from the network. A
challenge-response system could be introduced into the initial agent transmission request using the secret key to pre-authenticate the transmission to prevent this, however a malicious node can still effectively isolate a node by flooding the physical radio medium – in a regulated, low-power environment with a small and limited radio-frequency bandwidth there is no way to avoid such an attack.

Sense[133; 134] and Sluice[64] are reprogramming systems for whole-system images that include incremental, per-packet signatures. This eliminates the protocol-based denial-of-service vulnerability of network transfers, but at the cost of an overhead of multiple seconds per transfer – this can be tolerated as the system is being completely replaced and rebooted in the update operation anyway, but is extremely excessive for a mobile agent system where frequent transfers are expected.

The secret key used by the Security Manager is installed on each node at the time of deployment. If this key is compromised, a new key must be physically delivered to each node in the system. The key is stored in the FLASH memory of each sensor node and can not be read out of that memory without the cooperation of the operating system. The only way to read the key is to create an agent to read it out, which can not be introduced into the system without knowledge of the key. The only way the key is likely to be compromised is by the person using the network (who requires the key to introduce agents) – however no system design can prevent this.

Clearly it is not possible to use a standard unencrypted code-update process to update the program image when the secret key is part of this program image. The image must either be encrypted when transmitted over the radio, physically distributed to the sensor node, or the key distributed to the sensor node after deployment, using a secure key distribution protocol – however this is outside the scope of this thesis.

The security check also serves as a last-resort integrity check for the received agent. As the transmitted data is written directly into the code memory of the destination sensor node, any errors that were introduced by previous software or communication failure would most likely corrupt or crash the node. With the security manager enabled, the received data would then be different to that used
5.8 Conclusions

to calculate the transmitter's signature, causing the signature check to fail, and the transmission to be restarted.

5.8.6 Reliability

If wireless sensor networks are to become a true autonomous platform, reliability is critical. The middleware provides reliability guarantees to each transfer, while the application design ensures that these guarantees translate into application-level reliability.

In order for an application to be reliable, it must deal with node failure. Wireless sensor nodes are cheap and mechanically vulnerable – the application can not assume that every node in the system will survive for the entire deployment. In addition, software failure and resource exhaustion can cause a node to fail recoverably.

Virtual machine-based mobile agent systems are the only other system that has been proposed for wireless sensor networks that could demonstrate such autonomous reliability. The mobility of a mobile agent system makes it inherently more robust than a static application – the mobile agent expects to execute on multiple different nodes over its lifetime and its design and execution model assume this. In order to manage a complex system, multiple interlocking agents can reinforce each other. This complex agent management requires more resources than are available in the limited execution environment of an embedded virtual machine.

In the case of a complete hardware failure, the watchdog timer should reset the wireless sensor node. This is an independent hardware timer, running separately to all other system clocks that automatically resets the node if it is not signalled periodically. The period if the watchdog timer is variable and dependent on a number of factors, however it is typically in the range of tenths of a second to two seconds. The timer is reset by the scheduler loop, so that if the node becomes deadlocked it will automatically reset. The system may however end up in a state where the normal scheduler loop is still operating but due to hardware misconfiguration or resource exhaustion it is unable to usefully function in the network.
5.8 Conclusions

In order to detect such scenarios, the sensor node must know what its role in the network is expected to be. This is highly application dependent, however some common failure scenarios can be defined. The system can consider itself to be in a failure state if:

- No communication is received for $N$ seconds
- No mobile agents have visited the sensor node for $N$ seconds
- Sensor readings are in an impossible state

These are not common system occurrences and the failure thresholds should be set high enough that they will not be triggered during normal operation.

The node will reset with a blank agent image - the middleware will be active but any agents that were active on the node when it crashed will be lost. This ensures that whatever caused the node to crash will not be restarted.

If the node can be sure that in the case of a reset it will eventually be re-integrated into the system, it does not need to worry about operating agents or state - any failure results in a reset, and the system relies on the surviving agents elsewhere in the network to recover the reset node. If the system is self-stabilising, this can also be used for the initial system setup - the control agent starts at a single node in the network and is guaranteed to discover the rest of the network and eventually reach normal operation, from any starting condition.

A reliable system was created using an autonomous network of self-reinforcing mobile software agents. A mobile agent can seamlessly move around the network, taking its functionality to the place at which it can best be used, while also checking that all other parts of the system continue to function. The proposed native agent system can self-manage - autonomously locating, working around and recovering from errors and failures. It can dynamically discover the network topology, constantly adapting to changing conditions and environments, while all the time providing a guarantee that as long as a sufficient sub-set of nodes survives, the complete application functionality will be restored. The overhead of an interpreted agent system restricts the available resources too much to allow such complex verification, while static systems require central management. The reliability and self-stabilising abilities of the system were shown by simulation in
a high node-failure environment – the reliable agent system was shown to provide
a high level of utilisation long after other systems have failed. While it is difficult
to quantify the power and flexibility of a programming system and an execution
environment, there is sufficient evidence to conclude that the native mobile agent
system represents a clear step forward over any existing system.

5.8.7 Necessary Conditions for Reliability

In a shared wireless network, reliability can not be guarantee under all circum­
stance and any guarantee is necessarily conditional. The conditions under which
reliability can be guaranteed in this system are as follows:

- Radio Communication must be possible. While the system can continue
to operate in a limited form, using historical sensor data if communication
is temporarily unavailable, gathering fresh data and maintaining reliable
operation require communication. This is a feature of wireless systems in
general and not of the agent model.

- The nodes must not fail so quickly that no monitor agent survives. The
proportion of nodes that contain monitoring agents is a design decision and
increasing this will increase the tolerance of the system to failure, but any
system will not survive if all nodes in the system fail simultaneously. While
failures are expected in the system design, they are still considered rare and
independent events.

- An individual node must be able to restore itself to a known good state on
detection of a failure. This requires each node to store a “gold” image in the
external storage and restore this image on startup. This is easily achievable
using current WSN self-management systems. Physical hardware failures
are not recoverable without physical intervention.

- Agent code must available somewhere in the network. As each node in
the network receives copies of all of the agents at different stages in the
operation and is likely to cache this code, this is less important than the
previous conditions, and would only be an issue faced with a complete
system failure.
Chapter 6

Conclusions & Outlook

This thesis has presented a novel native mobile agent middleware system for wireless sensor networks. This mobile agent system has been implemented and evaluated, using accurate emulation of wireless sensor networks and nodes. While mobile agent systems exist for wireless sensor networks, native mobile agents capable of executing directly on the hardware of a wireless sensor node without interpretation have not previously been implemented on embedded sensor node-scale hardware, as they were considered too large, complex and power-hungry.

The implementation demonstrated in Chapter 5 shows that native mobile agent systems are a viable system in resource-limited embedded wireless sensor networks. A complete control system was implemented, including redundancy, reliability and dynamic discovery in less than 10% of the available storage space on the hardware platform discussed. This meets the requirement that the mobile agent system should be able to coexist in a network with other applications that are not agent-aware.

This is an important requirement if wireless sensor networks are to be used as general-purpose computing platforms. If the sensor network must only ever fulfil one role from the time of deployment, the application can be designed to use 100% of the network resources. This allows the system designer to give more specific performance guarantees, however the designer must attempt to predict all possible usage scenarios and potential future requirements. A dynamic programming system allows the system designer to modify the network's operation after
deployment, and a modular dynamic system allows multiple application systems to co-exist.

Mobile software agents impose an overhead on the network, compared to a static program. This is traded off against the improved flexibility and dynamic behaviour that is possible from a mobile agent system. The overhead of the mobile agent system is least onerous in situations where the benefits of the mobile agent system are the most applicable. These are systems where the physical point of decision making is mobile or distributed, and where the application functionality changes over time. In general, a dedicated, static software program will out-perform a multi-purpose agent system when the application functionality is fixed and static.

The improved execution model allows the implementation of algorithms that were not previously possible in mobile agent systems for equivalent hardware platforms. This expands the application scope of such devices, allowing more complex control, management and actuation systems to be developed. The combination of complex behaviour with mobility in very small, low-power devices merits further investigation as it has the potential to substantially change the operating space of such devices.

Not all applications will transfer well to such a platform. The system is limited in fast transfers, when the agent code can not be provided from a local cache. In a shared, low-power wireless medium it is impossible to guarantee that an agent transfer can always take place - the radio medium is a shared resource among an unknown number and type of devices, of varying sophistication. This limits the applicability of the system in systems that require hard real-time timing or availability guarantees - the restriction is imposed by the characteristics of the radio medium rather than the agent system particularly, and applies to any system requiring radio transfer.

Ultra-low power operation in wireless sensor networks is determined primarily by the MAC-layer performance. In order for the agent system to achieve low-power operation, it must have support from the MAC layer, and in order for the MAC layer to achieve reasonable agent transfer times it must have support from the agent middleware.
6.1 Contribution

Code caching and reuse are very important in mobile agent systems in general. For anything more than a trivial agent, the executable code will be many times the size of the agent state. Eliminating the transfer of this code as much as possible provides considerable performance benefits. In order for this caching to be possible, the system must be able to identify an agent’s code without reference to a specific instance of that agent. This requires some sort of identifier, service directory or signature for each agent.

The system is designed to be as scalable as possible, however some issues require resolution before the system could operate in a massively large-scale environment. The service discovery request is currently flooded out through the network, and the routing (as part of the carrier move system) is implemented as a simple AODV protocol. The implementations of these functions are not fundamentally important to the middleware system and alternative are easily available. These operations aside, each node only ever communicates with the one-hop neighbours. As long as addresses are unique and consistent in this space, they need not be globally unique. As the network size increases, the number of agents in the system will likely also increase. There is no limit on the number of agents in the network in general, however each node can only support a limited amount of executing agents at one time. While this limit is hardware- and implementation-dependent, the class of devices considered is unlikely to support more than tens of agents at any time. Again, this is a limitation imposed by the limitations of the hardware platform and the potential availability equals or exceeds comparable systems in similar hardware platforms.

6.1 Contribution

In summary, the primary contribution of this work is the development, implementation and evaluation of a novel native mobile software agent system for embedded wireless sensor networks. This brings together previous work in dynamic update of mobile wireless sensor nodes, with basic mobile agent systems. This provides a new platform on which to develop wireless sensor network applications, and opens up new areas of research in complex, highly mobile applications.
6.1 Contribution

The execution model of the native agent was proven to be much more flexible than any existing system for embedded wireless sensor nodes. It proved impossible to implement some simple sample applications in an interpreted system, due to the limitations of the architecture. The native mobile agents were shown to execute between 3 and 30 times faster than a sample interpreted system, even with the interpreted agents operating with reduced functionality. For an agent sufficiently simple to allow direct comparison, the entire native agent executed in the time taken to interpret one instruction of the virtual-machine based version.

A hybrid memory allocation approach is used for maximum efficiency. Both fixed and dynamic memory are available to mobile agents, and dynamic memory can be treated as local or mobile agent state. This allows simple agents to operate efficiently with only fixed memory, while complex agents can create any data structure they require. A novel, simple self-management system is used to reconstruct the memory state across the transfer to provide maximum flexibility.

In addition to the basic middleware enabling mobile agents, functionality has been added to reduce the size of the mobile agents. This takes some tasks that are common to many mobile agents and takes them out of the agent and into the middleware. Chief among these is the agent carrier system. This takes movement decisions out of the agent, allowing the agent to move according to a specified condition without concerning itself with the details of the topology or attributes involved. While this draws from existing work on attribute-based data collection, no such system has been implemented for agent mobility in sensor networks. A signature-based security mechanism was also added to allow the middleware to authenticate arriving agents.

A protocol was developed to allow reliable transfer of mobile agents, over single or multiple hops. This can be integrated with a system-specific routing system if the underlying hardware or operating system provide one, or with a generic protocol if they do not. A code caching system was implemented to optimise code transfer when the executable code was previously available at a node. The time for a complete agent transfer in the absence of interference was measured as 430ms per hop for a 390 byte agent without code caching, and 69ms per hop with caching, including a complete signature verification of the agent code.
6.2 Discussion

Evaluation of this system has been carried out, using a highly accurate wireless sensor node emulation system. This allows detailed instrumentation and evaluation of every phase of the operation, without interfering with the system being monitored. The performance was shown to exceed that of existing mobile code and mobile agent systems on the same class of hardware devices. As expected, the initial code transfer is a limiting factor in the operation of the system in a highly dynamic environment. After the initial code transfer, the application can move around the network quickly and reliably, with all of the power of a native system.

The evaluation was used to calculate expected system lifetimes, subject to some typical operating conditions. This was used to prove that for dynamic, mobile systems in large networks, the proposed native mobile agent system will out-perform existing technologies, while providing a base on which reliable, autonomous and self-stabilising applications can be designed.

A number of general conclusions were established for mobile agent systems in resource-limited devices in general, and particularly for larger and more complex agents. These are listed in more detail in Section 3.7, but in summary:

- FLASH memory storage and code caching improve performance considerably. RAM is not sufficient to store agents.
- Transaction-based agent transfers improve reliability
- Hybrid memory tracking and state reconstruction techniques provide efficiency for simple cases while still allowing complex state structures
- Directed movement is generally more efficient than explicit, per-hop movement.

6.2 Discussion

Software systems for wireless sensor networks are gradually evolving. The first wireless sensor network systems used an entirely static monolithic software image containing both the entire operating system and all of the application code. The two were statically linked together, and were indistinguishable at the operating
6.2 Discussion

Updates were performed on the entire program image, and no state was preserved across these updates.

Module-based systems provide some improvement. Code is still static, however the executing environment is broken up into a micro-kernel and dynamically loadable modules. The update of one module affects only that module, and not the operating state of the rest of the system. Modules do not move from the node at which they are started. The module code is executed directly on the node hardware.

At the other end of the reprogramming spectrum, more complex dynamic tasks are carried out by interpreted execution environments. This ranges from complete stack-based virtual machines to attribute-based data gathering systems. The application is represented by a query or simple agent, executing through a heavy interpretation layer. The query or agent is distributed around the network, discovering the appropriate place to execute. It is managed and controlled by an external application. The limitations of the interpreted execution environment when faced with sophisticated application demands were demonstrated in Chapter 5, where even relatively simple applications exceeded the available programming system and resources.

The work presented in this thesis bridges the gap between the two. The agents are mobile and powerful, and they do not suffer from the overhead of an interpretation layer. Agents can choose which nodes in the system to visit, either directly or conditionally. They have all of the power of dynamically loadable modules, executing directly with all of the node’s resource available, with the additional benefit of mobility and middleware support.

While WSN hardware is improving in power and performance over time, according to Moore’s Law, there is an alternative direction in which this can be taken. Improving technologies can either be used to expand the capabilities of the hardware, at constant cost and energy consumption, or can be used to make existing designs smaller, cheaper and less power-intensive. This represents the two directions that research is taking in hardware for WSNs. One is towards more featured, powerful devices capable of high-level integration, but suffering from high energy consumption, while the other is towards smaller, highly efficient devices. This represents the original WSN vision of pervasive “smart dust.”
nodes — simple, ubiquitous devices, simple at an individual level but capable of complex operation when combined.

This represents an evolution from static wireless sensor networks, with intertwined operating system and application code, that were fixed and infrequently updated, to a system where a separate wireless sensor network and controlling middleware provide a complete platform, on which the application executes. The static software on the sensor node enables and supports the agents. The application agents are mobile, dynamic and autonomous, using their mobility to best manage the limited resources of the wireless sensor node. Agents are injected dynamically into the network, where they operate autonomously, performing their desired task to completion, then terminating. The sensor nodes in the network are no longer the application themselves — they are the infrastructure on which the application operates. The application runs on the network — it is not the network.

Evaluation of WSN performance is not as well-defined as in higher-level systems. There are no standard benchmarks or test suites, and any evaluation must aim to reproduce as accurately as possible real-world performance. The applications used in the evaluation must correspond to the intended application space of the technology being considered. As benchmarks are not standardised, it is very difficult to generate simple "side-by-side" comparisons, and any evaluation must be based on a range of criteria.

The performance of the native agents while executing has been shown to be superior to equivalent interpreted systems. While difficult to compare directly in all but the most trivial applications, the natively executing system executes comparable instructions in a fraction of the time of an interpreted system. This shows that native mobile agent systems are a viable and powerful technology even in the limited execution environment of an embedded wireless sensor node, with substantial energy consumption benefits during execution over existing systems.

In order for mobile agents to be viable, they must have sufficient mobility. Code transfer is the key limitation in native agents, as the executable code can be ten or more times greater in size than the agent's operating state. The initial visit of an agent is substantially slower than subsequent visits, due to the need to transfer the executable code. The system therefore is most efficient in systems
where the agents are highly mobile, making multiple visits to each node in the system. Once the initial transfer has taken place, code caching ensures that later transfers occur within tens of milliseconds, which combined with execution times of less than one millisecond, allow the creation of highly dynamic and mobile systems of mobile agents.

In a system where code caching can be exploited, the native mobile agent system can in fact out-perform interpreted systems and compete with static systems in terms of data transfer, while providing a robust and secure environment to encapsulate the transferred data. The encapsulation of a mobile agent removes the need for the application developer to consider issues of authentication and state transfer.

A self-managing and self-healing networks of mobile agents was demonstrated. This system displayed the ability to recover from node failures, and restore system operation. As long as there are multiple agents in the system, and they are sufficiently mobile, the chances of all of the agents being destroyed or isolated simultaneously by failing nodes or communication problems are very small. The absence of one agent is noticed by the others, which take steps to recover the lost agent. This, combined with self-stabilising algorithms that discover the operating environment dynamically allow true autonomous and distributed operation in a way that has not been previously possible in embedded wireless sensor networks.

This self-management comes at a price in terms of terms of complexity and energy use – twice as much traffic was generated, and the use of three agents instead of one in the sample application increases the energy consumption, however good design practice and the use of efficient transfer protocols and code caching will keep this to a minimum. If wireless sensor networks are to develop into viable autonomous control and management systems, this price is well worth paying for the autonomous, reliable and self-managing behaviour that it produces.

6.3 Future Work

Native mobile agents are only beginning to be investigated in embedded systems. The work presented in this thesis shows that such systems are feasible and powerful, however a number of areas of potential improvement remain.
6.3 Future Work

Simultaneous Operations

The middleware system is limited to one transfer at a time. This is very restrictive, particularly when transferring the code of a large agent, which can take multiple seconds. The sensor node is unable to send or receive any other agents while this is in progress. Future work should investigate allowing at least a send and receive operation to take place simultaneously. It is likely to be advantageous to allow many simultaneous transfers – the resulting contention for the radio medium will result in a reduced overall throughput. Access to the program FLASH memory is buffered through a 256-byte page (FLASH writes are internally carried out by a bulk erase and rewrite, at the page level). If multiple agents are read and written simultaneously, this caching ability will be lost, increasing the frequency of FLASH physical operations from once per page to once per packet.

Cross-layer integration

The middleware system does not attempt to interact with the MAC layer – packets are simply presented for transmission and after reception. It is assumed that an unreliable MAC protocol with low overhead is in operation (such as the simple CSMA-CA MAC used in SOS). In particular, a hybrid MAC protocol with basic cross-layer integration is required to achieve the full performance benefits of the native mobile agent system. This must be capable of switching between low-power, high-latency operation for occasional, random-access traffic to a high-power, low-latency mode during agent transfers. This allows the system to minimise idle listening, and minimise power dissipation. As the mobile agent middleware can determine the exact start and endpoint of the agent transfer, this can be used to switch the MAC layer between the various modes of operation.

It may be possible to vary some of the parameters of the system dynamically in response to information provided by the MAC layer. This requires more complex interaction between both endpoints of the transmission, as they can no longer assume that they have the same transmission timeouts.
6.3 Future Work

**Carrier Move Operations**

Only a simple set of carrier move operations are implemented. Extension of this will allow more complex movement operations to take place, without direct agent intervention. This allows the agent to move around the network without stopping at each node, and takes the required code out of the agent, reducing the agent size and speeding up the transmission. This has the potential to become very application-specific, however if it is known in advance that a particular move operation will be common, there are substantial performance benefits from taking it out of the agent.

**Application Space**

This work has only begun to investigate the scope for mobile software agent systems in wireless sensor networks. Mobile agents have the potential to open up new areas of research and new application spaces. This requires a movement away from the static application-based programming paradigm that has been inherited from higher-level programming systems, and a move towards dynamic networks of small, cooperating, self-monitoring and reenforcing agents, each with a simple task which when combined, carry out the required operations reliably and autonomously.

**Reliability and Self-stabilisation**

The control system described in Chapter 5 demonstrates that mobile agent systems can autonomously work around node failures and resets, allowing the network to self-manage and self-heal. This allows the system to operate in environments where physical access to individual nodes is difficult or expensive. These environments, where nodes are embedded into infrastructure or deployed in hazardous conditions, are also where transient and permanent failures are common. Sophisticated agent systems allow complex operations to be performed reliably, even in these difficult conditions.

The guarantee of reliability should be examined in terms of existing work on self-stabilisation. It is possible to analyse the operating conditions and establish
6.3 Future Work

the necessary conditions required to prove that self-stabilisation can be guaranteed, subject to certain conditions. A network of cooperating agents can then be designed to be robust and reliable, capable of surviving node failures, working around topology changes and adapting to any combination of operating conditions, allowing the truly autonomous and self-managing operation that has never been fully realised on wireless sensor network platforms.
Appendix A

This Appendix contains some of the agents discussed in Chapter 5, implemented as both native software agents, and as Agilla agents.

A.1 Location Native

A.1.1 Location Agent

#include <sys_module.h>
#include <string.h>
#include <malloc.h>
#include <agentfw_mod.h>
#include "lochelp/lochelp.h"

#define PERSONLOC_PID (DFLT_APP_ID + 1)

typedef struct {
    char name[10];
    int6_t total_rssi;
    uint8_t num_responses;
} room_info_t;

typedef struct {
    room_info_t *info;
    uint6_t *responded;
    uint6_t origin;
    uint8_t num_rooms;
    uint8_t num_nbs_called;
    uint8_t num_nbs_sent;
    uint8_t num_responded;
} app_state_t;
A.1 Location Native

static int8_t location_msg_handler(void *state, Message *msg)
{
    app_state_t *s = (app_state_t*)state;

    switch (msg->type)
    {
        case MSG_AGENT_START:
        {
            /* Need to find the location of THIS node */
            uint8_t *data = (uint8_t *)(msg->data);
            if(((*data) & AGENT_OPTIONS_NOSTATE))
            {
                sys_agent_remap(NULL);
                memset(s, 0, sizeof(app_state_t));
                s->origin = sys_id();
            } else if(sys_id() == s->origin) {
                sys_agent_utility(2, 0x0000);
            } else {
                sys_agent_remap(NULL);
                sys_agent_remoterequest(0x95, MOD_MSG_START+4, AGENT_OPTIONS_AGENT | AGENT_OPTIONS_KEEPTRYING);
                s->num_calls = 1;
            }
            break;
        }
        case MOD_MSG_START:
        {
            uint16_t target = ((uint16_t*)((uint32_t*)msg->data));
            sys_agent_move(FROM_CARRIER(CARRIER_EXATNODE, target, CARRIER_MVTOWARDS, target), 0x00,
                         AGENT_OPTIONS_CARRIER | AGENT_OPTIONS_AGENT | AGENT_OPTIONS_KEEPTRYING, MOD_MSG_START + 2);
            break;
        }
        case MOD_MSG_START+4:
        {
            agent_response_t *resp = (agent_response_t*)msg->data);
        }
if(resp->success)
{
    /* An agent was successfully loaded */
    sys_post_value(resp->pid, MOD_MSG_START, ((uint32_t)sys_agent_utility(1, s->num_nbs_sent)) << 8, 0);
    s->num_nbs_sent++;
    if(sys_agent_utility(1, s->num_nbs_sent) != 0xffff)
    {
        sys_agent_remoterequest(0x85, MOD_MSG_START+4, AGENT_OPTIONS_AGENT | AGENT_OPTIONS_KEEPTRYING);
        s->num_nbs_called++;
    }
}
break;
}

case MOD_MSG_START+5:
{
    sys_post_net(msg->sid, MOD_MSG_START+2, 0, NULL, 0, msg->saddr);
    lochelp_msg_t *loch = (lochelp_msg_t *)(rasg->data);
    if(s->num_responded)
    {
        uint8_t i;
        uint16_t n = s->responded;
        for(i=0;i<s->num_responded;i++,n++)
        {
            if(*n == msg->saddr) && (*n != sys_id())
            {
                return 0;
            }
        }
        s->responded = sys_realloc(s->responded, sizeof(uint16_t) * (s->num_responded + 1));
    } else {
        s->responded = sys_malloc(sizeof(uint16_t));
    }
    s->responded[s->num_responded] = msg->saddr;
    s->num_responded++;
    if(loch->name[0] != 0)
    {
        room_info_t *r;
        if(s->num_rooms)
        {
            uint8_t i;
            for(i=0, r=s->info;i<s->num_rooms;i++, r++)
            {
                char *a = r->name, *b = loch->name;
                uint8_t j=0;
            }
        }
    }
}
A.1 Location Native

```c
while(j++ < 10)
{
    if(*a != *b)
        goto skip_loop;
    if(*a == 0)
        break;
}
r->total_rssi += msg->cor;//rssi;
r->num_responses++;
goto breakout_loop;

skip_loop:
r++;
}
s->info = sys_realloc(s->info, sizeof(room_info_t) * (s->num_rooms + 1));
r = s->info + s->num_rooms;
else {
    s->info = sys_malloc(sizeof(room_info_t));
r = s->info;
}
memcpy(r->name, loc->name, 10);
r->total_rssi = msg->cor;//rssi;
r->num_responses = 1;
s->num_rooms++;
}
breakout_loop:
if((s->num_msgs_sent == s->num_responded)) {
    /* Move back */
    uint8_t i;
    uint16_t max_so_far = 0;
    room_info_t *cur_room = s->info;
    for(i=0;i<s->num_rooms;i++, cur_room++)
    {
        if(cur_room->total_rssi > max_so_far)
        {
            max_so_far = cur_room->total_rssi;
            if(i != 0)
                memcpy(r->i<s->info, cur_room, sizeof(room_info_t));
        }
    }
    if(s->responded)
    {
        sys_free(s->responded);
    }
    s->info = sys_realloc(s->info, sizeof(room_info_t));
sys_agent_move(FORM_CARRIER(CARRIER_EXATNODE, s->origin, CARRIER.MVTOWARDS, s->origin), 0x00,
```

204
A.1 Location Native

AGENT_OPTIONS_CARRIER | AGENT_OPTIONS_AGENT | AGENT_OPTIONS_KEEPTRYING, MOD_MSG_START + 2);

break;
}
}
return SOS_OK;
}

A.1.2 Helper Agent

#include <sys_module.h>
#include <agentfw_mod.h>
#include <string.h>
#include "lochelp.h"
#define LOCHelp_PID (DFLT_APP_IDO + 5)

typedef struct {
  uint16_t orig;
  uint16_t dest;
  uint8_t respmsg;
  uint8_t resppid;
} app_state_t;

static int8_t lochelp.msg.handler(void *start, Message *e);

static const mod_header_t niod_header = {
  .mod.id = LOCHelp_PID,
  .state.size = sizeof(app_state_t),
  .num_sub_func = 0,
  .num_prov_func = 0,
  .platform.type = HW_TYPE or PLATFORM.ANY * /
  .processor.type = MCU_TYPE,
  .code.id = ehtons(LOCHelp_PID),
  .module.handler = lochelp.msg_handler,
};

static int8_t lochelp.msg_handler(void *state, Message *msg)
{
  app_state_t *s = (app_state_t*)state;

  switch (msg->type) {
    case MOD_MSG_START:
    {
      lochelp_move_t *move = (lochelp_move_t *)(msg->data);
      s->orig = sys.idO();
      s->respmsg = move->respmsg;
      s->dest = move->dest;
      s->resppid = msg->sid;
      sys_agent_move(s->dest, 0x00, AGENT_OPTIONS_AGENT, MOD_MSG_START+3);
  
  return SOS_OK;

  break;
  }
}
case MOD_MSG_START + 3:
{
    char *repmsg = sys_malloc(10);
    *repmsg = 0;
    sys_post(s->resppid, s->respmsg, 10, repmsg, SOS_MSG_RELEASE);
    sys_post(msg->did, MOD_MSG_START + 2, 0, 0, 0);
    break;
}

case MSG_AGENT_START:
{
    sys_agent_remap(NULL);
    uint8_t *data = (uint8_t *)(msg->data);
    if(((*data) & AGENT_OPTIONS_NOSTATE))
    {
        return 0;
    }
    sys_timer_start(0, /*2048*/523, TIMER_REPEAT);
    //break;
}

case MSG_TIMER_TIMEOUT:
{
    void *nodeloc = (void *)sys_blackboard_read(0x1000);
    char *repmsg = sys_malloc(10);
    *repmsg = 0;
    if(nodeloc != ((void *)0xffff))
    {
        memcpy(repmsg, nodeloc, 10);
    }
    sys_post_net(s->resppid, s->respmsg, 10, repmsg, SOS_MSG_RELEASE, s->orig);
    break;
}

case MOD_MSG_START + 2:
{
    /* Die */
    sys_agent_utility(2, 0x0000);
    break;
}
return SOS_OK;
A.2 Location Agilla

// Read destination from tuple-space "dst"
pushn dst
in

// Move to destination
RE_MOVE_0:
copy
smove
cpush
pushc 1
cneq
rjumpc RE_MOVE_0
pop

// Get neighbour list and add it to stack
copy
pushn dst
out
aid
numnbrs
NEIGHBOUR_LOOP:
copy
pushc 0
cceq
rjumpc FINISHED_LIST

copy
getnbr
swap
dec
rjump NEIGHBOUR_LOOP

FINISHED_LIST:
pop
aid

// Visit each neighbour in turn
REVISIT_LOOP:
swap
RE_MOVE_1:
copy
smove
cpush
pushc 1
cneq
rjumpc RE_MOVE_1
pop

// AT neighbour, TOS is <node id> of managing node

// Read room location from sensor 9
pushc 9
sense

setvar 11
pushc 0
// Search room list for this room
ROOM_SEARCH_LOOP:
copy
copy
getvars
copy
pushc 0
cceq
rjumpc CHECK_ROOM_VAL
swap
setvars
inc
pushc 1
swap
setvars
rjump FINISHED_ROOM_LIST

CHECK_ROOM_VAL:
pushc 11
getvars
cceq
pushc ROOM_SEARCH_LOOP
jumpc
pop

// Found in room list
inc
getvars
inc
A.1 Location Native

swap
inc
setvars

FINISHED_ROOM_LIST:
// Move back to managing node
copy
smove
cpush 1
cneq
rjmpc FINISHED_ROOM_LIST
pop

swap
pop

swap
setvar 10
ceq
rjmpc DECISION_TIME
getvar 10
aid

// Not finished, go on to next neighbour
pushc REVISIT_LOOP
jumps

// Decide which room we are in
DECISION_TIME:
pushc 1
setvar 11
pushc 0
DECISION_LOOP_START:
copy
getvars
pushc 0
ceq
pushc MOVE_BACK
rjumps
pop
inc
copy

getvars
copy

getvar 11
getvar
cgte
rjmpc SKIP_SET
setvar 11
SKIP_SET:
pushc DECISION_LOOP_START
jumps

MOVE_BACK:
pushn dst
in
smove
getvar 11
getvars
pushn loc
out
halt
// 104 instructions total
A.3 Control Native

```c
#include <sys_module.h>
#include <agentfw_mod.h>
#include <pin_defs.h>
#include <bitsop.h>
#include <string.h>
#include "agentfu/staticcontrol/staticcontrol.h"
#include "agentfw/statmonitor/statmonitor.h"

#define MONOCONTROL_PID 0x88
#define RETRY_THRESHOLD 2
#define CHECK_CRAWLER_MASK 7
#define SETPOINT 0x51
#define ONE_THRESHOLD 0
#define TWO_THRESHOLD -10
#define THREE_THRESHOLD -20

#define SET_HEATER(x) do { if((x) & 0x01) { SETBITHIGH(PORTC, 0); } else { SETBITL0W(PORTC, 0); } 
       if((x) & 0x02) { SETBITHIGH(PORTC, 1); } else { SETBITLOW(PORTC, 1); } } while(0)

/* Crawl around network, make sure each node is alive and is running a monitor agent */

typedef struct {
    uint8_t node_reached;
    uint8_t *room;
    uint8_t node_failed;
    uint16_t *reading;
    uint16_t num_nodes;
    uint16_t num_failures;
    uint16_t target;
    uint8_t temp_pid;
    uint8_t retries;
} app_state_t;

static int8_t personmob_msg_handler(void *start, Message *msg) {
    app_state_t *s = (app_state_t*)start;

    static int8_t personmob_msg_handler(void *start, Message *msg) {
        app_state_t *s = (app_state_t*)state;
```
switch (msg->type){
    case MSG_AGENT_START:
        {
            uint8_t *data = (uint8_t *)(msg->data);
            if(*data & 0x01)
                /* Fresh start — wait for instructions */
                s->node_reached = sys_malloc(l);
                s->room = sys_malloc(l);
                s->node_failed = sys_malloc(l);
                s->reading = sys_malloc(l);
                s->num_nodes = 0;
                s->num_failures = 0;
            else
                s->node_reached = sys_agent_remap(s->node_reachQd);
                s->room = sys_agent_remap(s->room);
                s->node_failed = sys_agent_remap(s->node_failed);
                s->reading = sys_agent_remap(s->reading);
                sys_agent_remap(NULL);
        }
        //break;
    case MSG_MSG_START:
        {
            uint8_t this_room = (uint8_t)sys_blackboard_read(0x3000);
            uint16_t id = sys_id();
            uint16_t max_nbs;
            uint8_t *nl;
            uint8_t *pa,*pb;
            max_nbs = id;
            if(s->num_nodes)
                {
                    max_nbs = s->num_nodes - 1;
                }

            agent_neighbour_response_t *nbs = sys_agent_neighbourlist();
            for(i=0,nl=nbs->neighbour;i<nbs->num_neighbours;i++,nl++)
                {
                    if(*nl > max_nbs)
                        max_nbs = *nl;
                }

            if((max_nbs + 1) > (s->num_nodes))
                {
                    s->node_reached = sys_realloc(s->node_reached, max_nbs + 1);
                    s->room = sys_realloc(s->node_reached, max_nbs + 1);
                    s->reading = sys_realloc(s->reading, (max_nbs * 2) + 2);
                    for(i=s->num_nodes,pb=&(s->node_reached[i]),pa=&(s->room[i]),nl=&(s->reading[i]);
A.3 Control Native

```c
for (i = (max_nbs + 1); i++, pb++, pa++, nl++)
{
    *pb = 0;
    *pa = 0;
    *nl = SETPOINT;
}
s->num_nodes = max_nbs + 1;

s->node_reached[id] = 1;
sys_free(nbs);

sys_sensor_get_data(l);
brent;
}

case MSG_DATA_READY:
{
    MsgParam *data = (MsgParam *)(msg->data);
    uint16_t id = sys_id();
    s->reading[id] = data->word;

    uint16_t this_room = sys_blackboard_read(0x3000);

    if(sys_blackboard_read(0x3001) == 1)
    {
        /* Make control action */
        uint16_t running_total = 0;
        uint8_t i, *rml, num_nodes = 0;
        uint16_t *rdl;
        for (i = 0, rml = s->room, rdl = s->reading; i < s->num_nodes; i++, rml++, rdl++)
        {
            if (*rml == this_room)
            {
                running_total += *rdl;
                running_total -= SETPOINT;
                num_nodes++;
            }
        }
        running_total /= num_nodes;
        num_nodes = 0;
        if (running_total < THREE_THRESHOLD)
        {
            num_nodes = 3;
        } else if (running_total < TWO_THRESHOLD) {
            num_nodes = 2;
        } else if (running_total < ONE_THRESHOLD) {
            num_nodes = 1;
        }
        SET_HEATER(num_nodes);
    }
```
A.3 Control Native

```c
sys_timer_start(0, 102, TIMER_ONE_SHOT);
break;
}

case MSG_TIMER_TIMEOUT:
{
    uint8_t found = 0, i, *pa;
    for (i = 0, pa = s->node_reached; i < s->num_nodes; i++, pa++)
    {
        if (!(*pa))
            found = 1;
    }
    /* Reset list if full, start another round */
    if (!found)
    {
        for (i = 0, pa = s->node_reached; i < s->num_nodes; i++, pa++)
            *pa = 0;
    }
    /* Move on to neighbour if possible */
    agent_neighbour_response_t *nbs = sys_agent_neighbourlist();
    uint16_t nl;
    for (i = 0, nl = nbs->neighbour; i < nbs->num_neighbours; i++, nl++)
    {
        if (s->node_reached[nl] == 0)
        {
            uint16_t nb = *nl;
            sys_free(nbs);
            s->retries = 0;
            s->target = nb;
            sys_agent_move(FORM_CARRIER(CARRIER_EXATHODE, nb, CARRIER_MVTOWARDS, nb), 0,
            AGENT_OPTIONS_AGENT | AGENT_OPTIONS_CARRIER, MSG_MSG_START + 2);
            return 0;
        }
    }
    sys_free(nbs);
    /* Not in neighbour list */
    for (i = 0, pa = s->node_reached; i < s->num_nodes; i++, pa++)
    {
        if (*pa == 0)
        {
            s->target = i;
            s->retries = 0;
            sys_agent_move(FORM_CARRIER(CARRIER_EXATHODE, i, CARRIER_MVTOWARDS, i), 0,
```
A.3 Control Native

AGENT_OPTIONS_AGENT | AGENT_OPTIONS_CARRIER, MOD_MSG_START + 2);
    return 0;
}
}
break;
}

case MOD_MSG_START + 2:
{
    if(s->retries++ > RETRY_THRESHOLD)
    {
        s->node_reached[s->target] = 1;
        s->room[s->target] = -1;
        s->num_failures++;
        sys_post_value(msg->did, MOD_MSG_START, 0, 0);
    } else {
        sys_agent_move(FORM_CARRIER(CARRIER_EXATNODE, s->target, CARRIER_MVTOWARDS, s->target), 0,
                       AGENT_OPTIONS_AGENT | AGENT_OPTIONS_CARRIER, MOD_MSG_START + 2);
    }
break;
}
return SOS_OK;
}
A.4 Control Agilla

// Given number of nodes, move around one by one
// Each node knows which room it is in (tuple)
// Keep average for each room and add in new values
// [This is a low-pass filter of actual values]
// Assume 10 values per room [LP]
// Make control decision at decision nodes
// Memory space:
// heap[11] = number of nodes in system
// [must be numbered 0 -- (N-1) ]
// heap[10] = number of rooms
// heap[8] = scratch
// heap[9] = scratch
// heap[0] = room id 0
// heap[1] = room average 0
// heap[2] = room id 1
// heap[3] = room average 1
// heap[4] = room id 2
// heap[5] = room average 2
// heap[6] = room id 2
// heap[7] = room average 2
// MAX 4 rooms -- no need for loop
// quicker to check directly
// sense 0 = sensor at node
// tuple rom = room value
// tuple htr = 1 if heater here
// tuple htv = heater setting

START:
pushn rom
inp
rjumpc SKIP_JUMP_SKIP_NODE
pushc SKIP_CONTROL
jumps
SKIP_JUMP_SKIP_NODE:
copy
getvar 0
cneq
rjumpc SKIP_0
getvar 0
pushc 0
cneq
rjumpc SKIP_0_2
setvar 0
pushc 0
sense
pushc 10
mul
setvar 1
pop
pushc 0
pushc SETVAL
jumps
SKIP_0_2:
copy
getvar 2
cneq
rjumpc SKIP_1
pop
pushc 2
pushc SETVAL
jumps
SKIP_1:
getvar 2
pushc 0
cneq
rjumpc SKIP_1_2
setvar 2
pushc 0
sense
pushc 10
mul
setvar 3
pop
A.3 Control Native

pop
pushc 2
pushc SETVAL
jumps
SKIP_1_2:
copy
getvar 4
cneq
rjumpc SKIP_2
pop
pushc 4
pushc SETVAL
jumps
SKIP_2:
getvar 4
pushc 0
cneq
rjumpc SKIP_2_2
setvar 4
pushc 0
sense
pushc 10
mul
setvar 5
pop
pushc 4
pushc SETVAL
jumps
SKIP_2_2:
copy
getvar 6
cneq
rjumpc SKIP_3
pop
pushc 6
pushc SETVAL
jumps
SKIP_3:
getvar 6
pushc 0
cneq
rjumpc SKIP_3_2
setvar 6
pushc 0
sense
pushc 10
mul
setvar 7
pop
pushc 6
pushc SETVAL
jumps
SKIP_3_2:
SETVAL:
inc
copy
getvars
copy
pushc 10
div
sub
pushc 0
sense
add
setvars
CONTROL:
pusha htr
inp
rjumpc CARRY_ON
pushc SKIP_CONTROL
jumps
CARRY_ON:
getvar 8
pushc 200
sub
copy
pushc 100
clt
rjumpc SKIP_SET_HIGH
pushc 3
push SET_CONTROL
jumps
SKIP_SET_HIGH:
copy
pushc 50
clt
rjumpc SKIP_SET_MED
pushc 2
push SET_CONTROL
jumps

SKIP_SET_MED:
copy
pushc 0
clt
rjumpc SKIP_SET_LOW
pushc 1
push SET_CONTROL
jumps

SKIP_SET_LOW:
SET_CONTROL
pushn htv
outp
pop

SKIP_CONTROL:

aid
inc
copy
getvar 11
cneq
rjumpc SKIP_RESET
pushc 0
setvar 11
pop
pushc 0
SKIP_RESET:
copy

smove
cpush
pushc 1
cneq
rjumpc SKIP_RESET
pop
pushc START

// 157 instructions total
Bibliography


[65] Philo Juang, Hide Oki, Yong Wang, Margaret Martonosi, Li-Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet. In Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), San Jose, CA, USA, oct 2002. 37

[67] D. PaliiG., G. James, and P. Corke. Electriccow: a simulator for mobile sen-


[69] Open Management Group. Corba common object request broker architec-
ture: Core specification, version 3.0.3. Mar 2004. 40


[72] MoteIV Corporation (now known as Sentilla). T-mote datasheet, August


[74] Nicholas R. Jennings. An agent-based approach for building complex soft-

[75] Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile

*Communications of the ACM. 17(11). 1974*. 42

[77] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic


