<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th>The SIXTH Middleware: sensible sensing for the sensor web</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Authors(s)</strong></td>
<td>Carr, Dominic</td>
</tr>
<tr>
<td><strong>Publication date</strong></td>
<td>2015</td>
</tr>
<tr>
<td><strong>Publisher</strong></td>
<td>University College Dublin. School of Computer Science and Informatics</td>
</tr>
<tr>
<td><strong>Link to online version</strong></td>
<td><a href="http://dissertations.umi.com/ucd:10051">http://dissertations.umi.com/ucd:10051</a></td>
</tr>
<tr>
<td><strong>Item record/more information</strong></td>
<td><a href="http://hdl.handle.net/10197/6851">http://hdl.handle.net/10197/6851</a></td>
</tr>
</tbody>
</table>
The SIXTH Middleware:
Sensible Sensing for the Sensor Web

Dominic Carr, B.Sc

The thesis is submitted to University College Dublin in fulfilment of the requirements for the degree of Doctor of Philosophy in the College of Science

April 2015

UCD School of Computer Science and Informatics

Head of School: Professor Pádraig Cunningham

Academic Supervisors: Professor Gregory M. P. O’Hare & Dr. Rem Collier
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>List of Listings</td>
<td>xi</td>
</tr>
<tr>
<td>Statement of Original Authorship</td>
<td>xiv</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>xvii</td>
</tr>
<tr>
<td><strong>I Introduction</strong></td>
<td></td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Preface</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 Middleware</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Objectives</td>
<td>5</td>
</tr>
<tr>
<td>1.3.1 Primary Objective</td>
<td>5</td>
</tr>
<tr>
<td>1.3.2 Secondary Objectives</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Contributions</td>
<td>5</td>
</tr>
<tr>
<td>1.5 Outline</td>
<td>6</td>
</tr>
<tr>
<td><strong>II Background</strong></td>
<td></td>
</tr>
<tr>
<td>2 Sensor Middleware</td>
<td>8</td>
</tr>
<tr>
<td>2.1 Preface</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Related Areas</td>
<td>10</td>
</tr>
<tr>
<td>2.2.1 Ubiquitous Computing</td>
<td>10</td>
</tr>
<tr>
<td>2.2.2 Mobile &amp; Wearable Computing</td>
<td>10</td>
</tr>
<tr>
<td>2.2.3 The Internet of Things</td>
<td>11</td>
</tr>
<tr>
<td>2.2.4 The Sensor Web</td>
<td>11</td>
</tr>
<tr>
<td>2.3 Core Domain Concepts</td>
<td>12</td>
</tr>
<tr>
<td>2.3.1 Sensor</td>
<td>12</td>
</tr>
<tr>
<td>2.3.2 Sensor Node</td>
<td>13</td>
</tr>
<tr>
<td>2.3.3 Wireless Sensor Network</td>
<td>14</td>
</tr>
<tr>
<td>2.4 What Is Middleware?</td>
<td>18</td>
</tr>
<tr>
<td>2.5 WSN Middleware Classification</td>
<td>20</td>
</tr>
</tbody>
</table>
IV Evaluation

7 Case Studies

7.1 Preface .......................................................... 120
7.2 Satisfaction of Desiderata ....................................... 120
7.3 Cyber Sensing With SIXTH .................................... 121
  7.3.1 Application of Pipeline Programming ................. 121
  7.3.2 Utilities .................................................. 121
  7.3.3 Implemented Adaptors .................................. 122
  7.3.4 Ubiquitous Mapping ..................................... 122
  7.3.5 Tracking Wildfires in Tasmania ....................... 123
  7.3.6 Discussion .............................................. 123
7.4 System of Systems ............................................. 124
  7.4.1 SNoMAC ................................................ 124
  7.4.2 Agent Oriented-Programming Bridge for Sensor-rich Environments ................................................................. 125
  7.4.3 Discussion .............................................. 126
7.5 Application Scenarios ........................................... 127
  7.5.1 Sensor-based Waste Disposal Tracking ................ 127
  7.5.2 Energy Management in a Home Context ............... 129
  7.5.3 Discussion .............................................. 132
7.6 Semantic Modelling with SIXTH ............................... 132
7.7 Extending Android With SIXTH ................................. 133
  7.7.1 Citizens Observatory WEB (COBWEB) .................. 133
  7.7.2 Portable Object Discovery Service (PODS) ............ 133
  7.7.3 Commercialisation of SIXTH ......................... 134
  7.7.4 Discussion .............................................. 134
7.8 Conclusions ................................................... 135

8 Evaluation .......................................................... 136

8.1 Preface .......................................................... 136
8.2 What Is Software Quality? ..................................... 137
8.3 Middleware Evaluation ......................................... 137
8.4 Software Quality Metrics ..................................... 139
  8.4.1 Metrics Utilized ......................................... 139
  8.4.2 Metrics Software Selection ............................. 142
  8.4.3 SIXTH Metrics Results ................................ 142
  8.4.4 Comparison .............................................. 143
8.5 Practitioners Survey ............................................ 145
  8.5.1 Identification of Usability Assessment Survey .......... 145
  8.5.2 Survey Design ........................................... 145
  8.5.3 Evaluation Procedure ................................... 146
  8.5.4 Results .................................................. 147
  8.5.5 Usefulness of Abstractions .............................. 147
  8.5.6 Free-text Feedback .................................... 148
  8.6 Discussion ................................................ 150

V Coda .......................... 152

9 Critique & Future Work ......................................... 153
Abstract

Governments, multinationals, researchers, and enthusiasts are presently weaving the planet’s “electronic skin” (Gross, 1999) via miniature, wireless, low-power sensor technologies. However, the control and interconnection of these diverse heterogeneous devices remains difficult, tedious, and time consuming.

The thesis proposes and develops a novel sensor-domain middleware permissive of any data source which espouses flexibility, domain modelling, design patterns, extensibility, and simplicity. This thesis provides an extensive review of the state of the art in middleware for sensor technologies. In doing so, a set of shortcomings is identified which form the basis for a desiderata for future sensor network middleware. In line with these aspirations the SIXTH middleware is designed, implemented, and evaluated thoroughly.

The design of SIXTH is true to the domain directly mapping virtual representations to real-world artifacts. The design incorporates the abstractions prevalent in low-level domain middleware such as logical grouping aggregates, and queries. SIXTH advances the state of the art by providing improvements over the form and function of its near neighbours. A concrete implementation has been delivered using OSGi as its basis. This implementation is evaluated through its usage in published case-studies, a survey of the developers utilizing the framework, and through objective code metrics.
## List of Figures

1.1 The Roadmap for the Thesis ................................................. 6

2.1 An IR Sensor ................................................................. 12
2.2 A Computational Sensor ....................................................... 13
2.3 Generic Sensor Node Architecture ........................................... 13
2.4 Diversity of Sensor Platforms ............................................... 14
2.5 A Wireless Sensor Network ................................................... 14
2.6 WSN Coverage Example ....................................................... 17
2.7 A Wireless Sensor Network with a Middleware Layer ................. 19
2.8 Middleware Support Levels (Bröring et al., 2011) ..................... 24
2.9 Taxonomy of WSN Middleware Solutions ................................ 26
2.10 Database Middleware High-Level View .................................. 31
2.11 Logical Groupings ............................................................. 31
2.12 Virtual Machine Middleware High-Level View ....................... 33
2.13 Agent Middleware Architectures .......................................... 35
2.14 Generic Service-Oriented Architecture ................................ 37
2.15 Generic Publish-Subscribe Mechanism ................................ 38
2.16 WSN Tuple Spaces ............................................................ 39

4.1 Iterative Development of the Mona Lisa (Cockburn, 2008) .......... 62
4.2 Brokers in SIXTH .............................................................. 63
4.3 The SIXTH Middleware Architecture ..................................... 65
4.4 Hierarchical Abstractions in SIXTH ...................................... 65
4.5 Sun SPOT Sensor Platform ................................................. 66
4.6 Sensor Node Model ............................................................ 66
4.7 Sensor Node Abstraction ...................................................... 66
4.8 Sensor Abstraction .............................................................. 68
4.9 SIXTH Query ................................................................. 70
4.10 Two views of SIXTH Adaptor Creation ................................ 73
4.11 SIXTH Deployment .......................................................... 75
4.12 Distribution in SIXTH ....................................................... 75
4.13 Tasking Service ............................................................... 77
4.14 Sensor Data Receiver ........................................................ 79
4.15 SIXTH Deployment-receiver Communication .......................... 80

5.1 Open Services Gateway Initiative System Architecture ................ 83
5.2 UML Class Diagram of the Provider and Descendants ............... 85
5.3 UML Sequence Diagram Depicting Interaction With the Node Provider 86
5.4 Specification Design Pattern UML Class Diagram ..................... 87
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Meta-analysis of WSN Middleware Taxonomies</td>
<td>21</td>
</tr>
<tr>
<td>2.2</td>
<td>Comparison of Middleware Platforms</td>
<td>25</td>
</tr>
<tr>
<td>2.3</td>
<td>Comparison of Operating Systems</td>
<td>29</td>
</tr>
<tr>
<td>2.4</td>
<td>Comparison of Macro-programming Middleware</td>
<td>30</td>
</tr>
<tr>
<td>2.5</td>
<td>Comparison of Database Middleware</td>
<td>32</td>
</tr>
<tr>
<td>2.6</td>
<td>Comparison of Virtual Machine Middleware</td>
<td>34</td>
</tr>
<tr>
<td>2.7</td>
<td>Comparison of Agent-Based Middleware</td>
<td>36</td>
</tr>
<tr>
<td>2.8</td>
<td>Comparison of Service-Oriented Middleware</td>
<td>38</td>
</tr>
<tr>
<td>2.9</td>
<td>Comparison of In-network Middleware</td>
<td>41</td>
</tr>
<tr>
<td>2.10</td>
<td>Comparison of Gateway-side Middleware</td>
<td>48</td>
</tr>
<tr>
<td>4.1</td>
<td>SIXTH Satisfaction of Desiderata</td>
<td>58</td>
</tr>
<tr>
<td>4.2</td>
<td>Design Patterns within SIXTH</td>
<td>59</td>
</tr>
<tr>
<td>7.1</td>
<td>SIXTH Case Studies Satisfaction of Desiderata</td>
<td>121</td>
</tr>
<tr>
<td>8.1</td>
<td>Middleware Evaluation Survey</td>
<td>138</td>
</tr>
<tr>
<td>8.2</td>
<td>Code Metrics Comparison: SIXTH, GSN, and WSNWare</td>
<td>143</td>
</tr>
<tr>
<td>8.3</td>
<td>System Usability Scale (SUS) Results</td>
<td>148</td>
</tr>
<tr>
<td>8.4</td>
<td>Self-identified Participant Expertise</td>
<td>149</td>
</tr>
<tr>
<td>8.5</td>
<td>Participant Answers to Bespoke Questions</td>
<td>150</td>
</tr>
<tr>
<td>8.6</td>
<td>Usefulness of Key SIXTH Abstractions (Q16 of Bespoke Questionnaire)</td>
<td>151</td>
</tr>
<tr>
<td>10.1</td>
<td>Comparison of SIXTH &amp; Gateway-side Middleware</td>
<td>158</td>
</tr>
</tbody>
</table>
# List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Creating a SpatialView (Ni et al., 2004)</td>
<td>30</td>
</tr>
<tr>
<td>2.2</td>
<td>Example TinyDB Query</td>
<td>32</td>
</tr>
<tr>
<td>2.3</td>
<td>Mate Script to Flash LEDs (Levis and Culler, 2002)</td>
<td>33</td>
</tr>
<tr>
<td>2.4</td>
<td>Agilla Code for Fire Tracking Agent (Fok et al., 2009)</td>
<td>36</td>
</tr>
<tr>
<td>2.5</td>
<td>Region Creation and Update in Abstract Regions (Welsh et al., 2004)</td>
<td>39</td>
</tr>
<tr>
<td>2.6</td>
<td>GSN Virtual Sensor XML Definition</td>
<td>44</td>
</tr>
<tr>
<td>4.1</td>
<td>Generics in the C++ Standard Template Library (STL)</td>
<td>60</td>
</tr>
<tr>
<td>5.1</td>
<td>IAdaptorProvider Interface</td>
<td>85</td>
</tr>
<tr>
<td>5.2</td>
<td>Configuring Sampling Frequency</td>
<td>89</td>
</tr>
<tr>
<td>5.3</td>
<td>GSN network reconfiguration code framework from AbstractWrapper.java</td>
<td>90</td>
</tr>
<tr>
<td>5.4</td>
<td>AbstractSensingNetworkAdaptor Configuration Method</td>
<td>95</td>
</tr>
<tr>
<td>5.5</td>
<td>Lookup Function &amp; Dynamic object Creation</td>
<td>95</td>
</tr>
<tr>
<td>5.6</td>
<td>Adding ISensorData to an Adaptor</td>
<td>95</td>
</tr>
<tr>
<td>5.7</td>
<td>Sensor Node Production of Valid ISensorData</td>
<td>99</td>
</tr>
<tr>
<td>5.8</td>
<td>Creating a Sensor Node Aggregate from a Query</td>
<td>100</td>
</tr>
<tr>
<td>5.9</td>
<td>Adding Data to a Sensor</td>
<td>101</td>
</tr>
<tr>
<td>5.10</td>
<td>SensorData Constructor</td>
<td>101</td>
</tr>
<tr>
<td>5.11</td>
<td>Mosen SensorReading Constructor</td>
<td>102</td>
</tr>
<tr>
<td>5.12</td>
<td>RESTful URL Schema for the SIXTH Server</td>
<td>105</td>
</tr>
<tr>
<td>6.1</td>
<td>Skeleton of the Dummy Adaptor</td>
<td>111</td>
</tr>
<tr>
<td>6.2</td>
<td>Dummy Sensor</td>
<td>112</td>
</tr>
<tr>
<td>6.3</td>
<td>Creating the Dummy Sensors</td>
<td>112</td>
</tr>
<tr>
<td>6.4</td>
<td>Tasking the Dummy Adaptor</td>
<td>113</td>
</tr>
<tr>
<td>6.5</td>
<td>Receiving Messages from the Sun SPOT</td>
<td>113</td>
</tr>
<tr>
<td>6.6</td>
<td>Sun SPOT Translator</td>
<td>114</td>
</tr>
<tr>
<td>6.7</td>
<td>Reddit User Sensor</td>
<td>116</td>
</tr>
<tr>
<td>6.8</td>
<td>Subreddit Sensor</td>
<td>117</td>
</tr>
<tr>
<td>6.9</td>
<td>User Data Receiver</td>
<td>118</td>
</tr>
<tr>
<td>7.1</td>
<td>SNoMAC RDF /nodes/:nodeID</td>
<td>125</td>
</tr>
<tr>
<td>7.2</td>
<td>ASTRA Data Receiver</td>
<td>127</td>
</tr>
<tr>
<td>7.3</td>
<td>DeviceAdaptor Tasking Code</td>
<td>130</td>
</tr>
<tr>
<td>9.1</td>
<td>Eligibility Method sans Functional Programming</td>
<td>156</td>
</tr>
<tr>
<td>9.2</td>
<td>Eligibility Method Using Functional Programming</td>
<td>156</td>
</tr>
<tr>
<td>9.3</td>
<td>Example of Java 8 Optional</td>
<td>157</td>
</tr>
<tr>
<td>A.1</td>
<td>SIXTH adaptor Activator</td>
<td>175</td>
</tr>
<tr>
<td>A.2</td>
<td>Skeleton for Adaptor Implementation</td>
<td>175</td>
</tr>
<tr>
<td>A.3</td>
<td>Threaded Adaptor Implementation</td>
<td>176</td>
</tr>
<tr>
<td>A.4</td>
<td>Code for Fetching Mouse Position</td>
<td>177</td>
</tr>
<tr>
<td>A.5</td>
<td>The run() Method for the Example Adaptor</td>
<td>177</td>
</tr>
</tbody>
</table>
Statement of Original Authorship

I hereby certify that the submitted work is my own work, was completed while registered as a candidate for Doctor of Philosophy and I have not obtained a degree elsewhere on the basis of the research presented in this submitted work.

The thesis work was conducted from September 2010 - January 2015 under the supervision of Professor Gregory O’Hare at University College Dublin.

The research was conducted with several collaborators. The collaboration was as follows:

- **Dr. Sean Russell**
  Development of the WAIST platform, implementation of Query Language parsing and assistance in developing Sun SPOT adaptor.

- **Dr. Olga Murdoch**
  Focused upon development of extended SIXTH layer for Cyber-sensing applications, provided impetus for additional design abstractions. Dr Murdoch is also a superb proof-reader and editor.

- **Barnard Kroon & Dr. Levent Gorgu**
  Development of AndroSIXTH extension of SIXTH for Android devices, Barnard Kroon provided assistance in refining the SIXTH design to encapsulate message delivery so as to make such an extensible component, which enables his work on Sensor Network ontologies.

- **Dr. David Lillis**
  Wrote the initial version of the SIXTH user guide from my notes. Additionally Dr. Lillis was an exemplary Internal examiner of this work, with whom I credit many improvements in the final version.
Contributions

The following publications derive in part or entirely from the research conducted in this thesis. Publications both finalised and in preparation for imminent submission are listed.

Published Work


In Preparation / Press


Software

The core of the SIXTH middleware as well as several implemented adaptors and services have been made available for download at http://sixth.ucd.ie.
Acknowledgements

I wish to firstly thank my supervisors Professor Gregory O’Hare and Dr. Rem Collier for their help across my tenure as a PhD student. This work would not be possible without you.

To my family, my parents, my brother Shane, and my fiancée Nula; I would not be here without all of you and your support and encouragement.

I would also like to thank F.D.C. Willard, Noir Carr, and Pudding O’Gara for their useful contributions to the discussion.

Finally, I dedicate this thesis to my departed grandmother Theresa Mc Brearty. I submit this work on the 2nd anniversary of your passing. You were loved, and you are missed. It has been said that a person dies twice “One time when you stop breathing and a second time, a bit later on when someone says your name for the last time” (Banksy, 2010) - that second death shall be not be for a long time yet for you live in the hearts of your children, your grand-children, and in all subsequent generations.
Part I

Introduction
1 | Introduction

“If we had computers that knew everything there was to know about things using data they gathered without any help from us we would be able to track and count everything, and greatly reduce waste, loss and cost….The Internet of Things has the potential to change the world, just as the Internet did. Maybe even more so.”

– Kevin Ashton, That 'Internet of Things' Thing (Ashton, 2009)

1.1 Preface

Humanity lives in a world of sensors; devices capable of the detection and measurement of physical quantities such as temperature, sound level, or humidity. Humans possess a vast array of sophisticated biological sensors without which daily life would be an impossibility.

In recent years, there has been a rapid acceleration in sensor device development; this has given rise to a much greater usage of sensors in everyday life. Sensors are rapidly becoming smaller, more computationally capable, more fully subsumed into the environment, and quite crucially abundant. Many people carry a sensor platform everywhere they go, in the form of smart-phones, tablets, and in the near future wearable technology. Such devices can sense location, temperature, humidity, velocity, orientation, pressure, and heart-rate. This sensor-laden environment has the potential to bring reality closer to the Ubiquitous Computing vision of Mark Weiser (Weiser, 1991).

“Ubiquitous computing names the third wave in computing…the age of calm technology, when technology recedes into the background of our lives.”

Society is still a great distance from such a vision wherein computing and technology disappear into the flow of our lives. Consider, for example, our notification spouting modern smart-phones that often act as interruptions and distractions rather than as an augmentation of daily life. Individual sensor platforms can deliver significant utility to the user; However, it is a fully sensor saturated environment which has the capability to become a trans-formative force in daily life. It is prudent therefore to consider seamlessly connected networks of sensors; most powerfully realized through a wireless connection medium.

Culler (Culler et al., 2004) heralded Wireless Sensor Networks (WSNs) as one of the 21st Century’s most important technologies. Hill (Hill et al., 2000) defined a WSN as a network of many spatially distributed autonomous platforms collaborating to achieve an objective. The uses of WSNs are numerous, diverse, and well documented. Examples of typical usage include environmental monitoring (Polastre et al., 2004), energy consumption monitoring (Ceriotti et al., 2009), supporting Ambient Assisted Living (Celler et al., 1994), battlefield monitoring (Winkler et al., 2008), and forest fire detection (A. S. Tanenbaum et al., 2006).

The Sensor Web (SW) vision (Delin et al., 2001) promotes the integration of local area Sensor Networks in a manner that ensures interoperability and obscures heterogeneity. The Sensor Web considers individual
sensing platforms, or sensor nodes, as complex computational entities capable of autonomous interaction and behaviour modification in the absence of external instruction. These nodes combine to form a cooperating amorphous network. The Sensor Web encompasses sensor access, control, and observation publishing via the World Wide Web (WWW).

The Sensor Web ideology can be extended to encompass diverse information streams coming from non-traditional sensing platforms such as real-time WWW-based weather reporting services. This information gathered from cyber sensor networks can be fused with directly sampled physical sensed data to augment the richness of the data-set. Within this thesis a cyber sensor network is defined as a collection of software components which monitor and retrieve real-time, or archived, information which is accessible through the WWW. In the Sensor Web vision it is possible to reconfigure remote sensing resources without knowing the particulars of the underlying routing network or communication format. Reconfiguration is paramount as the needs of consumers change over time; reconfiguration of such devices needs to be achievable through uniform software interfaces. This capability will be delivered through a set of well-defined standardized interfaces and appropriate programming primitives.

There exist many barriers and challenges which need to be addressed before the vision of the Sensor Web can be delivered in its entirety. The vision cannot become manifest without sufficiently robust mechanisms for device connectivity, control, and representation. Issues exist in hardware diversity, incompatibility, communication heterogeneity, and messaging formats. There exist a plethora of privacy related concerns (Chan et al., 2003; Karlof et al., 2003) such as device security (Perrig et al., 2004), denial of service (D. R. Raymond et al., 2008), the provenance of the observations (Ledlie et al., 2005; H.-S. Lim et al., 2010) and the instructions issued. Another major concern lies with provisioning unique addresses for sensor nodes.

Internet Protocol Version 6 (IPv6) (Deering et al., 1998) has the potential to eliminate sensor addressability concerns over the Internet Protocol. The Sensor Web vision is described (Botts et al., 2008) as an enabler for The Internet of Things (IoT) (Ashton, 2009); which envisages all smart devices connected to the internet. In the IoT all devices, numbering in the billions, are uniquely addressable and capable of interacting autonomously through machine to machine (M2M) protocols (Wu et al., 2011). In this vision devices are capable of vastly distributed collaboration, computation, and coordination. Standard device operation is to be autonomous and disappear into the background of daily life.

1.2 Motivation

Data collected from sensors is crucial for decision making and constructing an up-to-date model of the environment. In typical usage it is necessary to gather samples of diverse phenomena through a set of heterogeneous sensors; consider the plethora of sensors required to effectively instrument the home environment. Sensor observations provide the concrete building blocks of context awareness which, in turn, underpins confident decision making.

Such sensing devices possess a plethora of disparate properties such as their battery life, communication medium, messaging formats, reconfiguration options, modality, sampling rate capabilities, and hardware limitations. It is a combination of this heterogeneity and the inherent difficulties of low-level device programming which fuels the requirement for a software layer to sit between these networks and their users. Low-level concerns include the presence of bug-prone device level code which is difficult to test due to poor feedback mechanisms such as LEDs. An often utilized solution to this problem is to insert a middleware layer (Campbell et al., 1999; Emmerich, 2000).
1.2.1 Middleware

Middleware is an often nebulous term and within the literature its specific meaning and the resultant features are unclear (Campbell et al., 1999; Emmerich, 2000; Al-Jaroodi et al., 2012). In this thesis there is agreement with the definition of Bakken (Bakken, 2001) that states that a middleware platform is akin to the plumbing systems which exist in every home and office. Middleware provides an essential connective tissue which remains as transparent as is possible while still being open for extension and modification. This definition is suitably non-exclusionary prompting exploration of diverse approaches. The design of a middleware framework within any problem domain is primarily concerned with:

- Remaining as invisible as possible so as not to hamper or constrain the user;
- Provisioning the interconnection of multiple complex systems, while disguising this complexity. In this way application developers focus on development at an application logic level (Medvidovic et al., 2003).

Mottola (Mottola et al., 2011) notes that one major issue holding back more widespread adoption of WSN technology is the necessity to labour at low-level sensor network programming. These low-level concerns often result in a duplication of developer effort for each new project. The management of a WSN is a non-trivial task as there are many unique concerns including the efficient utilization of resources to facilitate long-term operation and accommodating network dynamicity. Such concerns should not be within the purview of a developer attempting to utilize sensor observations within their application. Middleware is the solution. There is a need for middleware platforms capable of autonomous control and modification in response to shifting topology or resource availability. Mottola (Mottola et al., 2011) identifies that while middleware is widely utilized in other problem domains most sensor network focused software is developed on a per deployment basis without the aid of a predefined middleware framework; this is despite the development of many sensor-focused middleware efforts.

With this in mind, this thesis describes the design and implementation of a new middleware platform for Wireless Sensor Networks, the Sensor Web and The Internet of Things. The approach adopted by this middleware is to provide a unified framework featuring a consistent interface to disparate sensor resources. The primary goal is to free an application level programmer from the low-level concerns of heterogeneity. Raising the level of support for the application programmer has the effect of freeing those practitioners from non-application specific tasks. This can be likened to the evolution of programming languages which has led to high-level languages (HLLs) which do not require the programmer to be concerned with notoriously fault-prone issues such as memory management or pointer arithmetic.

It is important that the system be suitably flexible to support traditional sensor networks and on-line data streams such as Really Simple Syndication (RSS) (RSS Board, 2006) feeds or micro-blogging platforms such as Twitter. The primary challenge here is the balance of sufficient generality and domain specificity within a cohesive development framework.

In treating all information sources in the same way data fusion becomes simpler. Extensibility is another key consideration of this thesis allowing for the rapid integration of support for new information providers or the replacement and augmentation of middleware defaults. The reconfiguration of a sensors operational parameters is necessary to support dynamic application demands. Such reconfiguration is key to the longevity of the network and is paramount in providing Quality of Service (QoS). A model in which data can only be consumed and actuation is not supported is not tenable for the demands of modern applications. There must exist the capability to tweak all aspects of the network where possible. Energy constrained devices by necessity must adapt their operation in terms of what and when they sense.
1.3 Objectives

This thesis conducts a thorough and comprehensive study of pre-existing middleware solutions in the area of Wireless Sensor Networks. This literature analysis informs the design and realization of the SIXTH middleware platform.

1.3.1 Primary Objective

There is necessity for a generic sensor domain middleware implementation which is strongly focused on the mapping of the domain abstractions into object-oriented representations in a generic manner which is permissive and supporting of application development. This zeal to which this goal is pursued must be tempered with the necessity to balance suitable generality with sufficient specificity. Such features are absent, or insufficient in flexibility or generality, in the existing approaches surveyed.

1.3.2 Secondary Objectives

The secondary objectives of this thesis are as follows:

1. To provide a comprehensive review and assessment of the state of the art in sensor network middleware and related research areas and in so doing determine the shortcomings of the various classifications of approach;
2. To identify the form and associated functionality of a middleware which will address many of the shortcomings identified in 1.
3. To design and implement a new sensor middleware framework, entitled SIXTH, which will accommodate diverse information streams, permitting;
   - sensor-driven applications to be abstracted from sensing resources;
   - data fusion from heterogeneous sources;
   - ease of integration and extensibility;
   - support for user applications and intelligent agent management;
   - auto generated GUI interaction components;
   - an extensible model for sensor network re-configuration;
   - concept-level extensibility of domain abstractions;
   - a robust model decomposing device connection and message translation;
4. To evaluate the efficacy of the middleware approach through a series of real-world case studies; each of which assesses the suitability of differing aspects of SIXTH.
5. To undertake an evaluation of system usability and quality through a practitioner’s survey and the application of quantitative code metrics.
6. To design a middleware such that design patterns will be adapted together with best design practice facilitating simplicity, clarity, extensibility, and flexibility.

1.4 Contributions

SIXTH is the primary contribution of this thesis, this middleware provides an enhancement of domain representation, and has successful usage in diverse scenarios which is indicative of ease of use and sufficient generality.

The secondary contributions of this thesis are as follows:
• A comprehensive literature review, and the identification of a desiderata for an idealized middleware solution to act as an impetus for the development of SIXTH.

• The design and implementation of the SIXTH middleware. This middleware offers advancements in a number of key areas:
  – A system in which the view of the sensing domain is relaxed to encompass all data producers. All information streams are treated as equal including observations gathered from WSNs and those obtained by cyber sensors.
  – A lightweight and extensible model for reconfiguration.
  – The communication model decomposes incoming and outgoing message translation.
  – High-level, Object-Oriented, and extensible query functions.
  – Provision of domain decomposition features which offer a unique conceptualization including brokers, providers, and logical grouping constructs lifted from the problem domain and low-level in-network approaches.
  – A model for interaction with any data source. Pre-configured adaptors facilitate rapid connection with heterogeneous data sources.
  – Support for dynamic functionality addition during runtime; replacing the default functionality.
  – An environment model for connection with intelligent agent systems which can manage a sensor network autonomously.
  – A peer-to-peer design which facilitates interconnection and scalability, underpinned by a strong Proxy Pattern.

• An evaluation of the SIXTH middleware is provided by means of published case-studies, quantitative industry standard quality metrics, and a results-proven practitioner questionnaire.

1.5 Outline

Figure 1.1: The Roadmap for the Thesis

Figure 1.1 provides an illustration of the roadmap and structure that will be adhered to in the remainder of this thesis. Chapter 2 provides an analysis of related work in the area of Wireless Sensor Network and Sensor Web middleware. This chapter provides a system comparison which is broken down across middleware implementation methodology and abstraction level. The analysis given in Chapter 2 informs the identification in Chapter 3 of a middleware desiderata. This feature set is enumerated making clear
the necessity for each in an ideal system. **Chapter 4** outlines the design of the SIXTH middleware in furtherance of the previously identified characteristics. **Chapter 5** discusses the implementation of the middleware in the Java language underpinned by the service-oriented OSGi modularity framework. **Chapter 6** provides a walk-through of the development of three SIXTH network adaptors. **Chapter 7** describes case studies which exercise the SIXTH middleware platform functionality. **Chapter 8** looks at the evaluation of SIXTH through a set of code metrics and reports from developers using the platform to develop a diverse set of applications. **Chapter 9** provides a critique of the middleware solution and discusses prudent future work. The thesis concludes in **Chapter 10** which provides an retrospective on the contributions of the thesis. **Appendix A** provides a user guide for the SIXTH middleware and **Appendix B** presents the user survey utilized in **Chapter 8**
Part II

Background
2 | Sensor Middleware

"Those who cannot remember the past are condemned to repeat it."

– George Santayana, The life of Reason (Santayana, 1905)

2.1 Preface

Chapter 1 established the importance of sensors and sensor observations as the core building blocks for context awareness and situational inference. The aforementioned are critical in delivering the vision of ubiquitous computing. Networks of diverse sensors connected over a wireless medium have been identified as the likely enabler of ubiquitous computing due to the ease of integration offered over an expensive wired solution. The more sensors, and the more diverse their observations, the more information that can be provided to derive knowledge, drive decision making and autonomic management. These sensors might be local to the environment or external; in the latter such remote information may illuminate the context of local observations. This chapter examines research done to date in the field of Wireless Sensor Networks (WSNs), related areas, and specifically middleware. Middleware is chosen for the goals of this thesis as it is a well accepted solution for managing the inherent complexity and heterogeneity of distributed systems; of which WSNs constitute a specialization.

A vast array of research exists which has explored WSN technology. Much of this research is summarized in the following literature reviews which reflect the growth of the field (Abbasi et al., 2007; Akkaya et al., 2003; Akyildiz et al., 2002a; Anastasi et al., 2009; Arampatzis et al., 2005; D. Chen et al., 2004; Demirkol et al., 2006; Huang et al., 2013; Al-Karaki et al., 2004; M. A. M. Vieira et al., 2003; K. Yang, 2014; Yick et al., 2008). The focus of this chapter is on providing an in-depth review of Wireless Sensor Network middleware. WSN middleware has been a keen focus for researchers in the field almost since its inception. The evolution of this research area is charted through a range of survey papers, notably (Chatzigiannakis et al., 2007; Delicato et al., 2014; Hadim et al., 2006; Henricken et al., 2006; Radhika et al., 2012; Romer, Kasten, et al., 2002; Sain et al., 2011; Sugihara et al., 2008; M.-M. Wang et al., 2008).

The primary objective of this thesis, asserted in Section 1.3, is the development of a new middleware platform which will encompass WSN technology as the primary information provider. This objective strongly prompts for a comprehensive review of the state of the art in sensor-domain middleware and the associated convergence of concepts from neighbouring domains. Consequently, this chapter examines the existing work in the domain.

The research conducted herein informs the design principles which underpin the proposed system. In Chapter 3, consequent to this background research, middleware platform design requirements are identified and discussed. This desiderata informs the design and implementation of a new middleware solution which is the subject of Chapter 4 and Chapter 5.
This chapter begins with an overview of related research areas (Section 2.2) and subsequently examines the key concepts of the WSN domain (Section 2.3), thus, providing clarification as to the role of sensors, sensor nodes, and the network itself. Having defined the problem domain the chapter moves into a discussion of the applications of such technology (Section 2.3.3) and the problems which hamper its usage (Section 2.3.3). As discussed in Section 1.2.1 middleware is often used to mitigate against issues of heterogeneity and low-level resource management and as such it is key to meeting the challenges identified for WSN usage. The thorny topic of what constitutes a middleware is discussed (Section 2.4) before moving on to identifying a WSN middleware classification (Section 2.5). A middleware classification is identified, and representative systems are considered under a set of criteria; the advantages and disadvantages of each approach are discussed (Section 2.5.5 - 2.5.3).

2.2 Related Areas

The following sections discuss research areas which are directly related to, or overlapping with, Wireless Sensor Network research.

2.2.1 Ubiquitous Computing

Ubiquitous computing (Weiser, 1991), represents a vision of the future wherein the world is saturated by electronic devices that are sensitive and responsive to the changing requirements of people and the applications they use. The goal is the development of non-intrusive environments whereby computers are embedded in everything, yet to the human mind they are not anywhere unless needed. The primary aims of ubiquitous computing systems are to be minimal, non-intrusive, aware of their environment, and omnipresent (Muldoon, 2007; Weiser, 1993a,b). O’Grady (O’Grady et al., 2006) presents the signpost analogy of ubiquitous computing having the central tenet that people do not log in to signposts. The interaction is almost unconscious, fluid, and perfectly subsumed into life. The provision of such seamless integration requires rich streams of information which builds a context model of the individual and the environment. Context is paramount for ubiquitous computing. Context can be defined as the set of circumstances or facts which surround an event. A lack of context compromises understanding. Multiple information streams provide context for each other; and a richer data set can dramatically change the perception of an event. Dey (Dey, 2001) provides an operational definition of context as: “any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant”. The definition emerged from dissatisfaction with earlier overly specific definitions such as those given by (Salber et al., 1999; B. Schilit et al., 1994; A. Schmidt et al., 1999). It is understood that sensors provide the basic building blocks of ubiquitous computing by providing streams of raw data about individuals, their surroundings, events nearby, environmental factors etc. which can be combined to infer context. Consequently, the availability of widespread wireless sensing technologies will contribute significantly towards the realization of ubiquitous computing.

2.2.2 Mobile & Wearable Computing

Mobile computing (G. Chen et al., 2000; B. N. Schilit et al., 1993) seems set to become the dominant paradigm of human-computer interaction (Preece et al., 1994). This has been fuelled by the fairly recent popularization of smart-phones, tablets and, to a lesser but growing extent, wearables. Such platforms are all sensor-rich providers of contextual information. Satyanarayanan (Satyanarayanan, 1996) identified four key constraints on mobile computing: unpredictable quality of the network, reduced robustness, finite energy resources, and a resource differential when compared to non-mobile computing artefacts. These
concerns, while somewhat mitigated, are still present in everyday usage. These qualities are all present in WSNs, the topic of WSN constraints is discussed at greater length in Section 2.3.3. Wearable computing (Mann, 1997; Starner, 1996), defined as the placement of computational devices on the body or clothing, research emerged from the perception that mobile computing was insufficient in delivering an implicit interaction paradigm. It is considered a separate research area which shares common issues with Wireless Sensor Networks and Ubiquitous Computing such as battery-life, ad-hoc networking, heterogeneity, and the need for context.

2.2.3 The Internet of Things

“In the next century, planet earth will don an electronic skin. It will use the Internet as a scaffold to support and transmit its sensations”

The above is how Gross (Gross, 1999) described the Internet of Things. This vision is of a world saturated with uniquely addressable sense-capable devices connected via IP; this has been appropriately conceptualised as providing the world with a “central nervous system” (Hartwell, 2011). Sterling (Sterling et al., 2005) characterised the vision as a movement away from the time monopolizing “Internet of Screens” and towards more implicit and indirect human interaction on the internet. When the idea first achieved popular uptake RFID was seen as a necessary enabler. Ashton (Ashton, 2009) posited that if everything is catalogued, inventoried, and queried then automatic management is greatly simplified. At present IPV6 is seen as the primary enabler for direct device connection to the IoT. Section 2.2.1 and Section 2.2.2 discussed ubiquitous, mobile, and wearable computing. These related research strands are enablers of the IoT. It is apparent that an abundance of sensorized environments and their subsequent ubiquity would provide sufficient building blocks for ubiquitous computing in the macro. Over time the IoT vision has expanded to encompass contributions from Wireless Sensor Networks research (Mainetti et al., 2011; Zorzi et al., 2010). WSN resources are controllable over IP through frameworks and middleware such as GSN (Aberer et al., 2006b) and LSM (Aberer et al., 2006b). Additionally WSN research has fuelled IoT through proposals for sensor networks to communicate over internet Protocol (Dai et al., 2004). Notable is the 6LowPan standard (Kushalnagar et al., 2007) which espouses that “the Internet Protocol could and should be applied even to the smallest devices” (J. W. Hui et al., 2008). The future of WSNs is likely to feature IP for both intra and inter-network communication. However, this view is not universally accepted, some researchers have pondered the necessary and applicability of total WSN-IoT integration (Alcaraz et al., 2010) citing unresolved security issues, unnecessary risk, and a lack of necessity.

2.2.4 The Sensor Web

The Sensor Web as defined by the Open Geospatial Consortium (OGC) (Botts et al., 2008) refers to “web accessible sensor networks and archived sensor data” which can be accessed over standard communication protocols and API’s. Delin (Delin et al., 2001) defined a sensor web as a WSN architecture of hardware equivalent sensor nodes which performed collaboration and coordination. In this thesis the term refers to the former definition. However, the goals of the latter are admirable, lofty, and relevant. As such they feed into the enhanced in-networking reasoning and M2M collaboration goals of the IoT. The Sensor Web concept overlaps with The IoT as sensor web architectures bring sensing resources into the IoT. Section 2.2.3 commented that the likely future direction of sensor networks will focus upon the direct connection of sensor platforms into the WWW through IP, consequently it is likely that much of the Sensor Web platform responsibility will be in the integration of non-compliment and legacy devices, however devices unsuitable for IPV6 and incapable of internet connection will persist for a long period of time.

---

1This is what celebrated English broadcaster Charlie Brooker called the “Black Mirror”

2Radio-frequency identification
2.3 Core Domain Concepts

In order to provide some contextual grounding for this background research the following sections describe sensors and sensor nodes in detail. This description is followed by a definition and discussion of Wireless Sensor Networks. From this basis sensor-driven applications are explored motivating a discussion of the challenges intrinsic in realising such applications with as little friction as possible.

2.3.1 Sensor

In the physical world context a sensor is a hardware component which is capable of measuring a physical, biological, automotive, acoustic, navigational or chemical quantity, e.g. temperature, electrocardiography (ECG), galvanic skin response, CO$_2$ level, or three-axis acceleration. A sensor converts its observations into a unit readable by the observer, for instance, a thermometer is easily readable by a human observer.

Figure 2.1 shows an example of a simple infra-red sensor. Such sensors typically convert their observations into electrical signals.

![An IR Sensor](http://en.wikipedia.org/wiki/File:Infrared_Transceiver_Circuit.jpg)

Lifton (Lifton et al., 2002) defines a sensor as a device that “transduces physical quantities from the real world into a machine-readable digital representation.” For the purposes of this thesis a more inclusive definition of a sensor is considered: a sensor is an entity that captures information from an arbitrary source and can deliver that data (augmented, transformed, stripped, or untouched) to a consumer element.

Figure 2.2 illustrates this computational model of a sensor. Therein the sensor receives information from one or more data sources, through push or pull mechanisms, once received the data is transformed, homogenised, and made available to associated data consumers. Other work from the literature (Coyle, Neely, Stevenson, et al., 2007; T. Gu et al., 2005; Le-Phuoc, Quoc, et al., 2011; Westlin et al., 2014) has furthered the cause of an inclusive sensor definition. Related work (Baqer, 2010; Baqer and Kamal, 2009; Breslin et al., 2009) make the case for the integration of sensor observations with social networks and the information produced therein.

Sensors are heterogeneous in a multitude of dimensions; and present in all shapes and sizes. An intelligent sensor is defined as having the capability to perform actions in response to their input stream, for example a clap activated light controller. Other sensors are capable of actuation but this is performed in response to given instruction rather than sensory input. Sensors may be mobile entities which move through an environment (mobile phone), or fixed unable to move unless explicitly reorganized by human intervention, consider an appliance monitoring smart-plug. Sensing may be passive, as with a strain gauge, or active, as with an accelerometer. Sensors and their host platforms, are often subject to finite energy resources, connectivity issues, and hardware failure. Computational sensors are also diverse; the level of actuation and feedback provided by their underlying data sources varies significantly. As traditional sensors have finite battery resources a computational sensor can also be so afflicted with API call limitations, lack of internet connectivity, resource downtime, or changing data representations. A computational sensor may be capable of resource actuation but this is not guaranteed.
2.3.2 Sensor Node

A sensor node, sometimes termed a mote, is a single device within the larger sensor network, which is capable of sampling sensory information from one or more inbuilt sensors. Processing is performed by a sensor node to determine when to sample data, when to communicate, and whether to modify sensed data through aggregation, filtering, removal, adjustment, or other processes. A sensor node communicates with other nodes within its network for a variety of collaborative purposes. In typical operation, information will be passed to one or more sink nodes connected to a base station e.g. a desktop computer.

Figure 2.3: Generic Sensor Node Architecture

Software programs execute on the microcontroller and interface with the sensors through an analogue-to-digital converter (ADC). External memory can be harnessed to store information such as routing tables or historical observations. The power source is typically a scarce resource such as a battery possibly augmented with an energy gathering solar panel (Paradiso et al., 2005). The transceiver facilitates communication between nodes and other devices; this is the resource that consumes the most power.

Figure 2.4 depicts a wide ranging collection of sensor node platforms. This provides some clarity on the diversity of sensor platforms which have begun to be woven into everyday life. Moving from left-to-right the example platforms are: Oracle Sun SPOT, Shimmer, Arduino Leonardo, Mica2, Tyndall Sugar Cube, HTC One M8, Chevrolet Impala, Ubisense, WiSense, Nest, Apple Watch, Smart Vest, and a Samsung Smart Fridge. These platforms possess wildly divergent power sources, computational capabilities, connectivity options, sensory input streams, and user interaction mechanisms. Figure 2.4 depicts sensor platforms which are physically manifest, within this thesis, the notion of a computational sensor node is utilized as a host for one or more computational sensors.
2.3.3 Wireless Sensor Network

Hill (Hill et al., 2000) describes a wireless sensor network (WSN) as a computer accessible network of many spatially distributed autonomous sensor nodes collaborating to achieve the overall systems objective. Figure 2.5 depicts a generic WSN situated in a art gallery. Therein a gateway sensor node interfaces with one or more external devices. This connection encompasses both the distribution of instructions and the delivery of sensor data. In Figure 2.5 the arrows denote bi-directional, over the air (OTA), inter-node communication. OTA communication is typically conducted using a standardised protocol such as Zigbee (ZigBee-Alliance, 2006), Wi-Fi, Near-field Communication (NFC), Ultrawideband (UWB), or Bluetooth (Haartsen, 2000).

The origins of WSN development can be traced back to Smart Dust (Kahn et al., 1999, 2000; Warneke et al., 2001). This hypothetical system envisaged a world saturated with tiny wirelessly connected sensing devices.

---

5 As a caveat, the systems objective may be composed of multiple composite application objectives.
6 The Tehran gallery of contemporary art is used for this fictional set-up.
devices which are sprayed onto the environment. Such devices would cooperate to service the needs of people and industries. A WSN is a form of distributed system. Tanenbaum (A. Tanenbaum et al., 2007) identified a distributed system as comprising “a collection of independent computers that appears to the user as a single coherent system.”, it should be noted that in WSN development the aforementioned goal is often aspirational, and is the ultimate goal of WSN research. A WSN shares the generic distributed system qualities identified by Coulouris (Coulouris et al., 2005):

- **Concurrent operation**: Different processes on separate machines can operate simultaneously.
- **Scalability**: In theory system capability can be augmented by the addition of new resources e.g. sensor nodes, sinks, gateways, or actuators.
- **Fault tolerance**: A distributed system is inherently tolerant of component failure as the design lacks a central point of failure. Service quality can be degraded as components fail or falter, for example sensing granularity suffers as nodes fail but some level of service can be provided as long as there is connectivity.
- **Openness**: Standard protocols allow for the combination of hardware from disparate sources. However, this may have been what Sun Tzu referred to when he remarked that “no battle plan survives contact with the enemy” (Tzu, 1910) as in practice this is often fraught with complexity.
- **Sharing**: Resources can be shared between components. For example processing tasks can be offloaded to more computationally capable devices.

Several qualities render WSNs distinct from generic distributed systems (Akyildiz et al., 2002b; Culler et al., 2004; Lewis, 2004; Romer and Mattern, 2004; Tubaishat et al., 2003). Such qualities overlap significantly with the Mobile Computing constraints identified by Satyanarayanan (Satyanarayanan, 1996), and discussed in Section 2.2.2. These qualities of WSNs are:

- **Scarcity of power**: A WSN has very limited computational resources; energy is a scarce resource and routing is dynamic in contrast with static routing in traditional distributed systems (Yu et al., 2004). Energy resource availability is an issue of critical concern in the WSN domain. Sensor nodes are typically powered by a perishable source such as a battery. Often WSNs are deployed in remote, hazardous locations and as such it is undesirable to replace batteries. Even in an ideal scenario expending person-hours manually replacing batteries is inefficient. Much effort has been expended to mitigate this issue through efficient limitation of radio communication and the identification and cycling of sufficient network subsets (Anastasi et al., 2009; Tian et al., 2003; Ye et al., 2002).
- **Fault prone devices**: Sensor devices are prone to failure. The causes include programmer error, inaccurate sensors, and faulty communication hardware.
- **Unreliable communication medium**: OTA communication is lossy, some packets may never arrive at their destination. This prompts a need for redundancy and the interpolation of observations.
- **(More) programming complexity**: The determination that programming is an intrinsically hard task has long been established in the research community. (Brooks Jr., 1956; D. C. Smith et al., 1994). The addition of resource constrained devices, the requirements for energy efficiency, and the inherently distributed application scenario make programming sensors more difficult (Bucur et al., 2011; Fok et al., 2005b; Welsh et al., 2004).
- **Poor debugging functionality**: Programming complexity is compounded by the difficulty of debugging a distributed code-base executing on, possibly, thousands of nodes.
- **Limited computational capability and storage**: Limits on computational power and storage capacity restrict which operations are appropriate to perform in-network.
• **Dynamic Network Topology**: Typically a WSN is set up to support *multi-hop routing* where intermediate nodes act as a transport layer for messages from origin to recipient. A network in which such a system is absent is termed a *single hop network*.

WSN middleware is useful in abstracting issues of energy management, efficient routing, duty cycling and issues with arise such as node failure and disconnection (Younis et al., 2014). Such abstraction empowers the application developer to work at a desirable high level. The following section provides a discussion of the application domain of Wireless Sensor Network technology.

**Applications**

Wireless Sensor Networks are utilized in a vast range of applications (Kuorilehto et al., 2005). They have been harnessed for numerous military applications (Bekmezci et al., 2009; Hussain et al., 2009; Lamont et al., 2011; Lee et al., 2009) including the detection of enemy forces, chemical weapons warning systems, remote target surveillance and terrain analysis.

In healthcare, WSNs monitors the physiological conditions of patients and their environment (Y. Chen et al., 2010; Chung et al., 2008; Huo et al., 2009). This technology can be applied to the identification of mobility in the elderly (Zijlstra et al., 2007). The relatively unobtrusive surveillance offered can enable such persons to live in their homes for a longer period; these systems are key to realizing the vision of Ambient Assisted Living (H. Sun et al., 2009).

Environmental monitoring is another area suitable for WSN technology as a WSN can be left unattended in inhospitable environs for long periods of time. This has motivated their utilization for diverse tasks. Examples include aquatic (Alippi et al., 2011), volcanic (Tan et al., 2010), infrastructural (Xu et al., 2004) and earthquake monitoring (Cheng et al., 2009), flood detection (Basha et al., 2008), and animal tracking (Abdelzaher et al., 2004).

A WSN is a necessary underlying technology for the development of home automation and energy management suites (Kazmi et al., 2013). To address energy reduction sensors monitor energy consumption and the state of devices. Sensors are coupled with actuators to effect appliance management. Further details of work in the energy management domain associated with the contributions of this thesis is given in Section 7.5.2.

WSN technologies have been applied to the field of waste disposal, wherein sensors are utilized to ascertain if tampering has occurred (Russell et al., 2013). An example of such an application underpinned by this thesis is discussed in Section 7.5.1. WSN applications have numerous requirements for the underlying network such as longevity of operation, a resistance to node failure, guaranteed sampling rates, and appropriate coverage. The forthcoming section provides a focussed discussion on a number of issues which hamper the usage of Wireless Sensor Networks.

**Usage Challenges**

The unique qualities of Wireless Sensor Networks discussed in Section 2.3.3 make programming a WSN a *non-trivial and often tedious task*. A WSN node possesses limited computational ability and finite energy; it is often prohibitively expensive or undesirable to replenish energy resources. These energy constraints impose restrictions whereby WSN applications should be lightweight and reduce network communication and sampling redundancy. Poor network longevity is regarded as limiting the adoption of WSNs (Mottola et al., 2011).

A wide range of approaches are employed to conserve energy in WSNs; herein some notable approaches are discussed. Topology control (TC) (Y. Wang, 2008) involves finding the optimal subset of nodes to
guarantee total connectivity within the network. Connectivity is achieved by exploiting node redundancy; which occurs when two or more nodes service the same region. The minimal subset is left active while the remaining nodes are put into sleep mode. TC has been shown to provide a threefold increase in longevity. TC approaches can be subdivided into location-driven and connectivity-driven approaches. Figure 2.6 depicts an example scenario in which the the area sensed by the central node is sufficiently observable by the neighbouring nodes. As such this node is surplus to requirements and is powered off.

In a location-based approach nodes are awoken at specific times based on known location. In a connectivity driven approach the network is configured to provide complete sensory coverage or connectivity. Protocols controlling sleep/wake cycles complement TC; active nodes can be placed in low-power mode when there is no network activity. Scheduled rendezvous (E.-Y. Lin et al., 2004), asynchronous wake (Y. Sun, Gurewitz, et al., 2008), and on-demand schemes (Y. Sun, Du, et al., 2008) represent different approaches to sleep/wake duty cycling.

Communication is very expensive in a wireless sensor network. Pottie (Pottie et al., 2000) determined that a node can execute 3 million instructions for the same power cost as transmitting 1kb over 100m. As such reducing data transmission is paramount. Data reduction (Anastasi et al., 2009) schemes reduce the samples transmitted. In-network processing is performed at intermediary nodes to limit the number of samples sent. These processing techniques are often application-specific. Data compression schemes (Kimura et al., 2005) are useful in reducing the size of the data. Data prediction schemes (Lu et al., 2004) use models to predict sensor samples within an error range. If the required accuracy is satisfied the model can be used to evaluate a query.

An alternate approach is to reduce data acquisition; this is often achieved by exploiting the temporal and spatial correlations of sensors and observations. Mobile sensor nodes are employed to move through the network and collect samples; this is only prudent when the cost of mobility is less than communication cost. The node itself may be mobile or attached to a mobile entity such as an animal (Abdelzaher et al., 2004).

Routing and topology in a WSN are dynamic (Al-Karaki et al., 2004); changes in routing occur as nodes expire or are duty cycled. Routing issues should be handled transparently from the point of view of the application. As network size increases it becomes a necessity to adopt efficient routing algorithms to spread the load in regard to data transmission. Debugging WSN applications is notoriously difficult as the programmer is often faced with only blinking LEDs to determine the error in their code (Krunic et al., 2007; J. Yang et al., 2007). Often it is necessary to re-purpose an already deployed network. Without a proper reconfiguration framework this may not be possible.
In Section 2.3.3 WSN application scenarios were discussed. Often in such application areas there is a need to utilize a diverse set of incompatible sensing devices. To facilitate communication between such platforms a middleware layer is often utilized. Indeed, middleware is an often used solution to all issues presented in this section, as well as in mitigating wider issues of heterogeneity and incompatibility. In the following section the question of what a middleware is shall be addressed.

2.4 What Is Middleware?

The origins of the term can be traced to a report on the NATO 1968 Software Engineering conference (Naur et al., 1969). Therein middleware is defined as a layer between service routines and user applications. Middleware achieved widespread usage in the 1990s as a means of interconnecting complex systems; however precise definition can be divisive.

Campbell (Campbell et al., 1999) defined middleware as any software layer that is placed between a distributed systems infrastructure and the application level. Bakken (Bakken, 2001) provided an identification of the primary function of middleware; which is to manage the complexity and heterogeneity inherent in a distributed infrastructure. This support is given through the provision of a common set of domain specific abstractions that remain consistent when underlying elements differ. Hadim (Hadim et al., 2006) defines middleware as a “software infrastructure that glues together the network hardware, operating systems, network stack, and applications.” The common thread among these definitions fits with the inclusive definition that middleware sits between, both separating and connecting, two or more pieces of hardware or software. Such a definition would not be troublesome to many researchers; the devil lies in the detail, promoting disagreement over what lies in the middleware layer, what belongs below, and what are the application level concerns (Campbell et al., 1999; X.-H. Sun et al., 2004; Vinoski, 2002).

Sugihara and Gupta (Sugihara et al., 2008a) discuss the particular need for WSN middleware to achieve and accommodate heterogeneity. The diversity of sensor platforms, and underlying Operating Systems (OSes), make this task difficult. There is a compelling need for one or more layers of abstraction to reduce friction and enable seamless tasking of diverse sensors. However, it is prudent to consider that too much abstraction brings its own difficulties (Bowen et al., 1995). In particular this can be a detriment to flexibility. These sentiments are echoed by Romer (Romer, Kasten, et al., 2002) who also identifies criteria and challenges particular to WSN middleware such as supporting development, maintenance, deployment, and execution. (Romer, Kasten, et al., 2002) states that WSN middleware should extend to the devices, and the networks connected to the WSN. Therein the importance of abstractions for dealing with heterogeneous sensor nodes is noted. Molla (Molla et al., 2006) is in agreement with the preceding criteria while also noting the importance of scalability and interoperability across hardware platforms.

Hadim (Hadim et al., 2006) remarks that a WSN middleware should include data aggregation services, enact management policies, and prolong system lifetime. Anastasi (Anastasi et al., 2009) meanwhile suggests that middleware shields the developer from sub-standard bespoke low-level design and provides built-in efficiency. Horre (Horré et al., 2007) suggests that middleware should provide support for manual, semi-automated, and automated sensor network management. Figure 2.7, derived from Figure 2.5, illustrates the broad role of WSN middleware. Therein, middleware is placed between the sensor network and the computational device which is controlling the network and consuming the produced observations. Optionally a middleware layer is also placed within each sensor node providing an abstraction of the nodes operation.

---

7North Atlantic Treaty Organization
Traditional Middleware systems such as the Distributed Component Object Model (DCOM) (Redmond, 1997) and the Common Object Request Broker Architecture (CORBA) (Ben-Natan, 1995) are considered too heavyweight (Souto et al., 2004) for use within WSNs. As noted by Mottola (Mottola et al., 2011) middleware can be an even more nebulous term in WSN development as “layers blur and blend together, to the point that placing a middleware layer in a WSN design becomes difficult, even conceptually”, the authors conclude that traditional middleware functionality is often present but not explicitly identified; they also derive from a literature review that most WSN deployments do not utilize middleware.

The term framework is often used in WSN literature in conjunction, or in place of, middleware. Truyen (Truyen et al., 2001) and Schmidt (D. C. Schmidt and Buschmann, 2003) define a framework as a “semi-complete” software platform in which the most variable components are extended to form a solution; thereby augmenting or overriding default behaviour. Middlewares are considered examples of frameworks however the converse does not always apply. Wolfgang Pree (Pree, 1994) posited that frameworks consisted of frozen and hot components. Frozen elements are static and define the relationships between components and their organization; what is commonly referred to as a system architecture. Hot spots are overridden by developers utilizing the framework to provide custom behaviour e.g. user interface event handlers. In general, a middleware provides hot spots but their utilization is discretionary. Frameworks are distinguished from traditional software libraries in several ways:

- **Inversion of Control (IoC):** In this paradigm the framework is in charge of execution flow. The inversion is relative to traditional invocation based interaction with software libraries (Fowler, 2004). IoC is often called the Hollywood Principle, “Don’t call us, we’ll call you”. This is the model utilized in event-driven middleware; in a sensing context this may be a user-defined handler for sensor node failure or unusual sensory observations.

- **Extensibility:** Elements of the framework can be overridden to derive application specific behaviour. For example this includes sub-classing elements of the framework.
The framework should provide default behaviour; as such the framework should be functional if not entirely complete.

The above has discussed a myriad of definitions and qualities of middleware. As noted in Chapter 1 this thesis embraces generalized definition of middleware as any software stack placed between an underlying architecture and user applications which provides an interaction simplification obscuring heterogeneity and complexity, however it should manifest. The following sections provide a classification of middleware approaches as applied to Wireless Sensor Networks.

### 2.5 WSN Middleware Classification

A need exists to classify a taxonomy of the approaches adopted in delivering middleware services for Wireless Sensor Networks. The classification identified is distilled from an examination of the research literature which has previously surveyed middleware in the area such as: (Chatzigiannakis et al., 2007; Hadim et al., 2006; Henricksen et al., 2006; Horré et al., 2007; X. Li et al., 2014; Molla et al., 2006; Radhika et al., 2012; Sain et al., 2011; Sugihara et al., 2008b; Tong, 2009). The forthcoming sections decompose middlewares into several categories. These categories reflect differing approaches to providing WSN middleware. Many distinctions are present at the approach level including how close the abstraction brings the application developer to the sensor network and the degree of control given to that developer.

The identified categories are as follows:

- **Sensor Databases**: View the sensor network as a distributed database.
- **Virtual Machines**: Application of virtual machines to WSNs to abstract hardware heterogeneity and aid reprogramming.
- **Agent-Based**: Agent-oriented programming (AOP) applied to WSN control and coordination.
- **Macro-Programming**: View the network at a high level; focus on programming the global behaviour.
- **Service-Oriented**: Compose systems from a set of loosely coupled services.
- **In-Network Middleware**: Miscellaneous approaches linked by a set of common concepts.
- **Gateway Middleware**: Primarily concerned with connecting with and managing multiple sensor networks.
- **Operating Systems**: Lowest level of middleware; provides messaging and sensing abstractions.

The remainder of this section examines previous survey work carried out in the area. This research has resulted in Table 2.1 which summarizes the approaches identified by each group of researchers. The following is an examination of these survey papers which details the reasoning of this thesis for the judicious utilization of their findings.
Table 2.1: Meta-analysis of WSN Middleware Taxonomies

<table>
<thead>
<tr>
<th>Survey</th>
<th>DB</th>
<th>VM</th>
<th>Agents</th>
<th>Macro</th>
<th>OS</th>
<th>Other</th>
<th>Tuple</th>
<th>Event/Message</th>
<th>Service-Oriented</th>
<th>Gateway</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Henricksen et al., 2006</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chatzigiannakis et al., 2007</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radhika et al., 2012</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Molla et al., 2006</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hadim et al., 2006</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sugihara et al., 2008b</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X. Li et al., 2014</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sain et al., 2011</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tong, 2009</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Horré et al., 2007</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>x</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Terfloth and Schiller, 2005</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Masri et al., 2007</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rubio et al., 2007</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delicato et al., 2014</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pignaton de Freitas, 2008</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8Referred to as modular programming
9Explicitly identifies the group-level approaches
10Referred to as modular code
11Identified as component-based, deemed to be a distinction without a difference
Hendricksen (Henricksen et al., 2006) identifies several categories of approach: (i) database abstraction (ii) tuple-space model (iii) event-based (iv) service-based. The database abstraction is present in the taxonomy of this thesis and is discussed in Section 2.5.5. Tuple-space models and event-based subscription are not considered as distinct approaches within this thesis but are discussed as common abstractions in Section 2.5.9. Service-based approaches are discussed in Section 2.5.8 and, even more so than others, features of this approach are present across the continuum. This phenomenon was discussed previously in the literature (Mottola et al., 2011). Tuple spaces are a messaging model that are very useful in asynchronous communication, but it is not useful to consider as an approach rather it should be considered as a communication implementation methodology. Event-based subscription is an important subcomponent of a middleware solution, and it typifies the role played by applications and sensors in the sensor application, wherein the sensors are producing information that is consumed by the listening applications.

Chatzigiannakis (Chatzigiannakis et al., 2007) provide another taxonomy as follows: (i) database abstraction (ii) virtual Machine (iii) agent-based (iv) network management tools (v) other approaches. The database abstraction was identified previously (Henricksen et al., 2006). The taxonomy of this thesis includes virtual machine and agent-based approaches; these are discussed in Section 2.5.6 and Section 2.5.7 respectively. The term network management tools is used to refer to systems supporting visualization and utilization of sensor data; herein these are not considered as middleware systems. (Chatzigiannakis et al., 2007) note that such taxonomies do not cater for middleware which exists partly or wholly outside the sensor network; such systems are discussed in Section 2.5.10. Further compounding previous discussion (Henricksen et al., 2006; Mottola et al., 2011), Chatzigiannakis also noted that difficulties exist in placing middlewares in a single category as features often overlap.

Horre (Horré et al., 2007) identify a concise taxonomy: (i) back-end (ii) gateway (iii) in-network. In-network approaches are those which operate within the sensor network, for example as an abstraction over the OS. Gateway systems operate on a traditional desktop machine and connect to the WSN through gateway nodes. A back-end is identified as a storage element and data viewer. This thesis breaks up the discussion of in-network middleware into several approaches in an effort to address key ideas and abstractions which are of importance to the high-level. The gateway middleware approach discussed in Section 2.5.10 encompasses the features of both Horre’s “gateway” and “back-end” systems.

Radhika (Radhika et al., 2012) provides a taxonomy sharing commonalities with both (Chatzigiannakis et al., 2007) and (Henricksen et al., 2006); they identify: (i) database (ii) virtual Machine (iii) modular Programming (iv) message-Oriented (v) application-Driven. Modular programming refers to the modularization of in-network code. The goal being to reduce traffic when performing updates the code running in the network. Modular code update is better thought of as a feature than an approach; notably it is a core component of many virtual-machine and agent-based middleware implementations. Message-Oriented refers to the use of the asynchronous publisher/subscriber paradigm; in this thesis this approach is seen as a common abstraction and is discussed in Section 2.5.9. The application-driven approach is broadly similar to the service-based approach discussed previously in (Henricksen et al., 2006); in such scenarios the middleware is concerned with providing desired QoS to client applications.

Molla (Molla et al., 2006) offer a familiar classification: (i) database (ii) tuple-space Model (iii) event-based (iv) service-based. All of which have been discussed previously.

Hadim (Hadim et al., 2006) promotes a rigorous classification with features a high-level separation into two distinct approaches as follows:

- **Programming Support:** (i) Virtual Machine (ii) Agent-based (iii) Database (iv) Application-Driven (v) Message-Oriented
- **Programming Abstractions:** (i) Macro-programming (ii) Local Behaviour
The highest level is classified as either programming support or abstraction. Programming support approaches provide systems to make programming the WSN easier, for example, reliable code update, or safe code execution. Virtual machine, agent-based and database abstraction are included in the taxonomy of this thesis. As previously discussed application-driven and message-oriented approaches are seen as common concepts and are discussed in Section 2.5.9.

Programming abstractions provide a layer over the network which changes how it is viewed; abstractions of sensor data are also provided. As with prior survey authors, (Hadim et al., 2006) acknowledge some feature overlap in their classification. Local behaviour refers the grouping of sensor nodes into logical regions; this is considered as a common construct in Section 2.5.9. Macro-programming refers to programming the network as a whole; this approach is considered in Section 2.5.4. (Rubio et al., 2007) identify broadly the same set of approaches to programming WSNs as the already discussed in (Hadim et al., 2006).

(Sugihara et al., 2008b) split the discussion of WSN programming approaches into two levels:

- **Low-level**
  - Operating Systems (OS) / Node-level languages
  - Virtual Machine
- **High-level**
  - Group-level
    * Neighbourhood-based
    * Logical Group
  - Network-Level
    * Database
    * Macro-programming

Low-level approaches include the use of virtual machines and operating systems featuring sensor programming languages and abstractions; both of these are included in the taxonomy of this thesis. The high-level approach is subdivided into group and network level solutions. Group level solutions utilize sensor node grouping strategies which are considered to be a common construct of macro approaches (see Section 2.5.4). Network level approaches encompass the database metaphor, see Section 2.5.5 and macro-programming discussed in Section 2.5.4.

Figure 2.8 from (Bröring et al., 2011) depicts a classification of sensor middleware which is subdivided across three layers and four approaches. The lowest layer is for sensors; this is where heterogeneous platforms perform their tasks; it is arguably more appropriate to term this the sensor network layer. The sensor web layer (see Section 2.2.4) provides a link between sensors and their high level users. The application layer is where clients directly interact with resources.

The four approaches are summarised as follows:

- **Sensor Network Management**: This class deals with the internal management of the network including issues summarised in Section 2.3.3 such as routing, coverage, and connectivity. In this thesis Sensor Databases and Virtual Machines fit into this classification.

- **Sensor Web Infrastructures**: This is a higher-level layer, as depicted in Figure 2.8, it exists entirely above the sensor layer. Sensors can be made available over the web to applications. The details of sensor networks such as communication medium are abstracted and commonly sensing resources are presented as services. In this thesis Gateway Middleware approaches fit within this paradigm.

- **Sensor Web portals**: These systems provide high level access to sensors through web resources. These are discussed in Section 2.5.10 as higher-level gateways. Most act merely as viewers not offering actuation capability.
• **Internet Of Things (IoT):** The IoT vision demands the incorporation of all “Things” into the internet. The objectives of this classification are achievable through sensor web middleware and the primary differences stem from metaphor. Arguably pure IoT approaches mean that all sensory devices should be directly connected to the web and not via a mediator middleware.

• **Web Of Things (WoT):** This approach encompasses work from the IoT espousing the same goals but provides explicit application layer support structures. Where the IoT brings physical devices into the Internet, the WoT seeks to support their integration into the WWW.

### 2.5.1 Final Taxonomy

Figure 2.9 showcases the middleware taxonomy used within the remainder of this chapter. The taxonomy is based on the detailed analysis of related research presented above. However, this taxonomy is somewhat divergent from the state of the art in related work, this is owing to the view that some approaches advocated in the literature are more cross-cutting features that can be applied to different types of WSN middleware and not distinct approaches. An illustration of this is the utilization of logical regions and shared memory models. Similarly, QoS considerations, typified by *application-driven* approaches, form an important part of a middleware solution, and it is counter-productive to position such as a distinct approach as it should be present across all approaches.

Table 2.2 provides a summary of extensive comparison of middleware platforms which is expanded in the following sections [Section 2.5.5-2.5.3]. These sections describe existing WSN middleware under the classifications identified at the beginning of [Section 2.5]. Each section describes the high-level approach and examines key implementations. Therein various tables are provided comparing systems within a given approach, these expand the binary $\times / \checkmark$ feature identification given in Table 2.2. In the forthcoming tables a $\times$ denotes that a feature is absent; $\checkmark$ denotes that a feature is supported and additional $\checkmark$ usage denotes enhanced support. The scale is, theoretically, unbounded. Firstly the following section discusses a set of criteria for comparing middleware implementations.

### 2.5.2 Comparison Criteria

Section 2.5 identified a middleware classification. Consequently, this section describes a set of criteria for use in the comparison of middleware platforms. These criteria are selected by systematically examining previously identified comparison methods from the literature.

In designing BISNET (Boonna et al., 2007), further described in [Section 2.5.7], the authors identify three broad design features of WSN support software:
<table>
<thead>
<tr>
<th>Name</th>
<th>Heterogeneity</th>
<th>Reconfiguration</th>
<th>Extensibility</th>
<th>Energy Awareness</th>
<th>Scalability</th>
<th>Intelligence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cougar</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SINA</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>TinyDB</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>DSWare</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>KSpot</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>XQuery</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Sensewe</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>DbManager</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Virtual Machine</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MagnetON</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Mate</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>VMSTAR</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SensorWare</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SmartMessages</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>PushPin</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SwistQM</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Agent-Based</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Agilla</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>In-Motes</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>MAPS</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SensioMax</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Impala</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>IrisNET</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>BISNET</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Service-Oriented</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mires</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Oasis</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>MidSN</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Macro-Programming</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Regiment</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Karios</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Abstract Regions</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>In-network</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TinyLIME</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>teenyLIME</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>TinyCube</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>MASTAQ</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>GRA</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Facts</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>FamiWare</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MiLan</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Funf</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Hood</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Pushpin</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Pogo</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>MoSen</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TS-Mid</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Gateway-Side</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WSNWARE</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Atvoly</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GSM</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LSM</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ThingSpeak</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IrisNet</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>SenseWeb</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TinySOA</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Smart-M3</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>MISTIEN</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Hourglass</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Hi-Fi</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SStreamWare</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Construct</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Operating Systems</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contiki</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TinyOS</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SunSPOF</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>T2</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Figure 2.9: Taxonomy of WSN Middleware Solutions

- **Autonomy**: The ability to operate for long periods of time without human intervention and to extend the lifetime of the network without direct intervention.
- **Scalability**: The ability to adequately react to growth in some dimension of the problem domain e.g. increase in data volume, user applications etc.
- **Adaptability**: Adapt what is performed, when it is performed and which nodes perform the actions.

Sugihara (Sugihara et al., 2008a) arrived at similar conclusions identifying the need for scalability and energy efficiency. The goals of autonomous operation were decomposed into failure resistance and collaboration concerns.

Low-level systems are found to provide capability for all criteria as the developer has freedom, however support is lacking, and implementation is difficult. The lack of support is, if not purposeful, at least fitting as low-level systems are engineered for use by skilled domain developers and freedom is necessary at this lowest development tier. Consequently, this increases the complexity of development with regard to investment in initial coding, bug testing, and program validity. Mid to high-level systems obscure components which address efficiency; this can be at times problematic if the developer requires flexibility. It is identified that there are often no capabilities to take control over collaboration or define new management policies.

The design goals proposed by Romer (Romer, 2004) have been influential in the literature (Andreou et al., 2011; Avilés-López et al., 2009). While the authors echo the above discussed a distinct set of important considerations is identified:

- **Abstraction**: This refers to how far removed the user is from the low-level sensor network; it is widely felt that users should be able to specify what is to be performed through high-level mechanisms. However as described by Sugihara (Sugihara et al., 2008a) this can be problematic at times as it sacrifices flexibility for ease of use. There is an important distinction between *hiding low-level detail* such as hardware configurations and *obfuscating* it.
- **Programmability**: Consequent to the previous item programming of the network(s) needs to be
flexible providing for easy configuration of a diverse set of properties.

- **Integration:** This item acknowledges the importance of having an open system which can be both extended and integrated with other platforms and tools.

In designing KSpot (Andreou et al., 2011) (see Section 2.5.5), further to the design goals of Romer, the authors are in agreement with regard to energy efficiency, autonomy, and scalability. However, these scholars describe several complimentary goals which warrant explicit distinction:

- **Modularity:** The architecture of the middleware platforms should be modular; such that different component implementations substitutable and new applications composed from existing services and libraries.

- **Accuracy in the presence of failure:** As identified in Section 2.3.3 WSNs are routinely subject to myriad usage challenges such as lossy communication, malfunctions, and node failure. Consequently, it is important to provide accurate readings despite these issues.

(Hadim et al., 2006) also include energy efficiency, scalability, and heterogeneity in their criteria. In addition, they also discuss the importance of aggregation and appropriate application knowledge. One of the most salient points that they raise is that there exists a balancing act between generality and application specificity. This is a key concern for middleware development. (Hadim et al., 2006) acknowledge the no system ticks all the boxes, and find that providing such a system is an open research question. They conclude that it is “unclear if network management and programming abstractions will stem from the known paradigms, or if all-new abstractions and approaches must emerge to specifically meet WSN goals.”

(Hadim et al., 2006) also include the following criteria:

- **Ease of Use:** This is defined as how far the application developer is abstracted from the sensor network; with the acknowledgement that for some applications there can be too much abstraction.

- **Dynamic Topology and Node Mobility:** It is necessary for middleware to provide abstractions to deal with shifting network topology which arises from node failure, duty cycling, and node mobility.

- **Network Organization** This deals with the spread of relevant information about the network to all nodes. This broadly encompasses dealing with dynamic resources e.g. bandwidth and energy.

- **Security:** Addresses the issues of denial of service (Wood et al., 2002), packet injection (Roman et al., 2006), eavesdropping (Chan et al., 2003) etc. This has to be handled in a WSN specific fashion as heavy security mechanisms are unsuitable in an energy conscious context. Applications need to know the provenance, and authenticity, of sensor observations, and often data should be private.

- **Quality of Service (QoS):** This criteria is determined to be very application specific. It is most often characterised as **connectivity, coverage, and granularity**. QoS mechanisms should be designed based on trade-offs between throughput, data delivery delay, and energy consumption.

To structure a middleware comparison the following comparison criteria is proposed. Many of the above discussed considerations from the literature have been discarded for the purposes of this thesis. **Quality of Service** and **Ease of Use** are not included explicitly as they are regarded as cross cutting issues which are addressed from various perspectives through explicitly named criteria. Issues of **security** are inherently complex and require substantial independent assessment beyond the scope of this work. Issues such as autonomy extend to the application layer; explicitly considered criterion such as reconfiguration capability, energy awareness, and the capacity for intelligent reasoning are in support of such concerns.

The following presents the criteria adopted in the remainder of this chapter:
• **Heterogeneity:** Heterogeneity in this context refers to the wide variety of sensor platforms that exist and the disparate protocols and means involved in interfacing with these networks. Not only is this manifest in sensor platforms, but also in the individual sensors wherein two sensors may employ different units of measurement, levels of accuracy, or support varying frequencies of observation. (Sugihara et al., 2008a) rightly assert that heterogeneity presents in a multitude of dimensions including hardware (sensors, platforms, and actuators), hierarchical network organisation (tiering, clustering) and distinctions in how the network is viewed and decomposed. It has been identified that middleware needs to interface with diverse platforms in a manner invisible to the user (Hadim et al., 2006; Molla et al., 2006).

• **Reconfiguration:** Reconfiguration refers to the ability to alter the behaviour of the whole system, or some elements thereof, during runtime. There is a spectrum of reconfiguration that runs from simple parameter tweaking such as sampling rate modification (Aberer et al., 2006b) to the deployment of a new application to the entire network (Chlipala et al., 2003). In-between these extremes tasking can be performed such as altering the routing algorithms (Pantazis et al., 2013; Souto et al., 2004), setting nodes sleep/wake cycles, and configuring in-network aggregation solutions (Madden, Hellerstein, et al., 2002).

• **Extensibility:** It is obvious that heterogeneity demands extensibility through the need to engender the introduction of diverse sensor platforms and sensor types (Aberer et al., 2006b; Perera, Jayaraman, et al., 2014). However, this is only part of the picture; WSN middleware must also be designed to permit the introduction of a diverse range of services that are required by applications, for example, customizable data storage, aggregation services or the application of semantic knowledge extraction algorithms (Matheus et al., 2012). Such extensibility is closely related the need for modularity (Andreou et al., 2011), indeed modularity is a necessary pre-condition for extension.

• **Energy Awareness:** Energy awareness necessitates that there is some awareness of power constraints and some mechanism by which to reduce energy usage. Energy reduction may be achieved through the reduction of radio usage, duty cycling, intelligent coverage, in-network aggregation, or varying the routing. In particular reduction in radio usage is very important (Pottie et al., 2000). Application requirements such as accuracy and latency must be considered and balanced with the need to enhance system lifetime. A generalized framework to support application specific QoS with respect to energy is needed; consider the ramifications of earthquake detection with high latency contrasted with the acceptability of latency in home management. In the former thousands of lives hang precariously in the balance, and in the latter a momentary delay in the automatic activation of your television will slightly annoy you.

• **Scalability:** Scalability refers to how the system operates when additional resources are added such as a huge network of nodes. (Hadim et al., 2006) describe the need to maintain acceptable performance as the network scales; there is an implicit acceptance that slowdown is unavoidable and granularity may be sacrificed. Scalability concerns in WSNs are primarily associated with bandwidth efficiency (Yu et al., 2004), as such this is tied into energy awareness concerns. There is also a need within a sufficiently large system to increase the amount of data aggregation to reduce information overload. It is also prudent to consider the shifting topology of the network as nodes fail and network size increases.

• **Intelligence:** Intelligence refers to autonomous control of the network in all aspects of its operation and objective fulfilment such that it can be maintained without human intervention. This efficient automatic management is critical in promoting longevity (Marsh et al., 2004).
2.5.3 Operating Systems

Operating systems for sensor nodes such as TinyOS (Levis, Madden, et al., 2005), MANTIS (Bhatti et al., 2004), Sun SPOT (R. B. Smith, 2007), LiteOS (Cao et al., 2008), SNACK (Greenstein et al., 2004), T2 (Levis, Gay, et al., 2005), InceOS (Harvey et al., 2012), TinyGALS (Cheong et al., 2003) and Contiki (Dunkels, Grouvall, et al., 2004) are also representative of middleware. Sensor network operating system have been augmented through multi-threading extensions such as Y-Threads (Nitta et al., 2006), TinyThread (McCartney et al., 2006), and Protothreads (Dunkels, O. Schmidt, et al., 2006).

TinyOS, an influential early development in the field, was concerned with code portability across sensor platforms through interface abstraction. Devices supported include: Telos, Mica2, and MTS130. To a large extent this was successful, but in some situations program code requires alteration, for example, if the temperature sensor of a node were of a different type. The NesC (Gay et al., 2003) language of TinyOS receives justifiable criticism for burdening developers with complexity; however, it is a very flexible development platform which places few restrictions on implementation. This freedom typifies operating system level approaches. Reconfiguration of TinyOS programs is expensive as the OS only supports full-image flashing; to address this concern both FlexCup (Marrón, Gauger, et al., 2006) and Dynamic TinyOS (Munawar et al., 2010) have provided modifications to TinyOS to allow for modular updates akin to those used in Virtual Machine approaches. SOS (C.-C. Han et al., 2005) is as an advancement upon TinyOS providing additional abstraction and management including dynamic memory allocation and a module system. The SOS kernel facilitates module management, transparent network messaging, and memory allocation. The modularity is energy cost effective when compared with TinyOS. Application programming is done in C; having the advantage of familiarity over NesC and is, subjectively, more coherent. C is also utilized in LiteOS (Cao et al., 2008). LiteOS is notable for utilizing UNIX-esque abstractions in a wireless sensor network. The sensor network is mapped onto an UNIX-like file-system, LiteFS. WSNFuse (Filipponi et al., 2009) middleware also takes this approach of mapping the network to a file system; this use of a familiar metaphor echoed the database metaphor.

Some academics reject operating systems as middleware layers (Horré et al., 2007); herein they are viewed as lower-level middleware. Operating systems provide programming abstractions to enable the creation of sensing applications. Within TinyOS, the programmer is shielded to a significant degree from the complexity of sensor management by an event-driven programming model. The details of radio communication and persistent storage access are abstracted. This thesis is in agreement the assertions of (Sugihara et al., 2008a) that operating systems and node-level programming abstractions provide flexibility and efficiency; however they are difficult to program and undesirably low-level.

<table>
<thead>
<tr>
<th>Name</th>
<th>Heterogeneity</th>
<th>Reconfiguration</th>
<th>Extensibility</th>
<th>Energy Awareness</th>
<th>Scalability</th>
<th>Intelligence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contiki</td>
<td>3</td>
<td>4 / 4</td>
<td>1</td>
<td>1 / 1</td>
<td>4 / 4</td>
<td>4 / 4</td>
</tr>
<tr>
<td>TinyOS</td>
<td>3</td>
<td>4 / 4</td>
<td>1</td>
<td>1 / 1</td>
<td>4 / 4</td>
<td>4 / 4</td>
</tr>
<tr>
<td>Sunspot</td>
<td>3</td>
<td>4 / 4</td>
<td>1</td>
<td>1 / 1</td>
<td>4 / 4</td>
<td>4 / 4</td>
</tr>
<tr>
<td>T2</td>
<td>3</td>
<td>4 / 4</td>
<td>1</td>
<td>1 / 1</td>
<td>4 / 4</td>
<td>4 / 4</td>
</tr>
</tbody>
</table>

2.5.4 Macro-Programming

This is a focus on programming the global behaviour of the network at a high level; to program the network as a whole. The behaviour of each node will be determined by the macro-programming framework to deliver upon the high-level demands. Sugihara (Sugihara et al., 2008b) noted this approach treats the entire network as if it were a “single abstract machine.” In general there is difficulty in generating efficient low-level code from macro-programming frameworks; this is offset by a much reduced complexity for application developers.
Listing 2.1: Creating a SpatialView (Ni et al., 2004)

```java
spatialview sv1 = camera @ Home10 % 100
visit each camera : sv1 {
    Picture picture = camera.getPicture();
    ...
}
```

In **Kairos** (Gummadi et al., 2005) the programmer specifies global behaviour and the back-end system uses the available resources to accomplish that goal. This is done by generating all the local behaviours and interactions. In Kairos, the developer writes a single program in a Python extension. From this node specific code is compiled, deployed, and executed. Three abstractions govern this paradigm; node data type; neighbourhood list and remote variable reading. Kairos utilizes a shared memory model similar to TS-Mid. **SpatialViews** (Ni et al., 2004) are logical groupings of virtual sensor nodes defined by a service e.g. temperature sensing at a specific location. The virtual sensor nodes are mapped to physical counterparts. The language, an extension of Java, defines an iterator construct to traverse the nodes in a SpatialView providing instructions on what actions should be performed. These instruction are migrated to each node and executed. Listing 2.1 depicts the creation of a spatial view containing all cameras in “Home10”. The iterator traverses these devices and gathers imagery. Additional macro-programming platforms include Abstract Task Graph (ATaG) (Bakshi et al., 2005), Pleiades (Kothari et al., 2007), Regiment (Newton et al., 2007), and Semantic Streams (Whitehouse, Zhao, et al., 2006). Regiment is a functional sensor programming language. Therein, logical region constructs, see Section 2.5.9, are used to identify the relationships between sensor nodes.

### Table 2.4: Comparison of Macro-programming Middleware

<table>
<thead>
<tr>
<th>Name</th>
<th>Heterogeneity</th>
<th>Reconfiguration</th>
<th>Extensibility</th>
<th>Energy Awareness</th>
<th>Scalability</th>
<th>Intelligence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regiment</td>
<td>✗</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>Kairos</td>
<td>✗</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>Abstract Regions</td>
<td>✗</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗ ✗ ✗</td>
</tr>
</tbody>
</table>

Logical Regions

A logical region is a grouping of sensor nodes that share a commonality for example grouping by geographical location. This is a specific type of macro construct, which straddles the line betwixt feature and approach, many of its adherents are discussed in Section 2.5.9. Grouping is often done based on device capability e.g. all nodes which have a temperature sensor. Figure 2.10 provides an example of location-based grouping and grouping by capability. Groups may have a variable membership which changes as devices lose capability or are physically relocated. This concept is applicable to the problem domain as users often require information tied to a specific region. The members of the region work in aggregate reducing data transmissions over wide areas of the network and minimising task duplication. Logical regions provide an intuitive resource for programmers suitable for capturing collaboration (Sugihara et al., 2008a).
The forthcoming Section 2.5.5 discusses Sensor Databases which are considered to be a specialization of a Macro approach. These are discussed separately as the approach is very prevalent in the domain.

### 2.5.5 Sensor Databases

In this approach the fundamental abstraction is to envision the entire sensor network as if it were a virtual distributed database. Figure 2.11 depicts a high-level view of a generic WSN database middleware. The distributed database, i.e. the sensor network, can be interfaced with by posing queries to it in the same way as with a standard database management system (DBMS). Interfacing with the network is done through Structured Query Language (SQL) type languages (Chamberlin et al., 1974); these dialects are used to formulate information extraction queries. Queries are routed to the appropriate sensor nodes which can satisfy the query. Typically such platforms support in-network data aggregation and optimize the system configuration in response to what is being queried.

**TinyDB:** The most widely known implementation of this approach is TinyDB (Madden, Hellerstein, et al., 2002; Madden et al., 2005), which provides the capability to query the entire WSN using SQL-like
syntax that is parsed, optimised, and used to actuate sensing within the network. Listing 2.2 defines a TinyDB query for light, temperature, and the node ID. This query is run on all suitable devices, and sampling is performed once per second for a period of ten seconds.\textsuperscript{12} It is evident from this example that such syntax is easy for non-technical end-users to utilize. TinyDB supports operators such as SUM, AVG, and MAX for in-network aggregation; however as with the majority of database-driven approaches there is no means to implement custom collaboration. There is also support for advanced SQL queries using joins and projections.

**Listing 2.2: Example TinyDB Query**

```sql
SELECT nodeid, light, temp
FROM sensors
SAMPLE PERIOD 1s FOR 10s
```

**DSWare** (S. Li et al., 2003) focuses upon efficient data storage. DSWare uses SQL-syntax to specify event detection parameters such as temperature in excess of 100°C. Efficient storage of data within the network is seen as a key enabler for reducing communication cost and provisioning in-network processing. Consequently, data replication guards against node failure or communication issues. This flow of data can optimize application queries as a node in closer proximity may have suitably fresh data originating at a leaf node to answer the query. The cost of duplication is considered, and a balance needs to be maintained. As in systems such as TS-Mid sensor nodes are grouped logically. Where event detection is subject to multiple readings a confidence function takes into account value agreement and responses from members, this is similar to the functionality of MASTAQ (see Section 2.5.9). **KSpot** (Andreou et al., 2011) is an energy conscious middleware providing actuation and sensory observations through SQL-like network queries. KSpot features workload balancers which prolong network lifetime by synchronising duty cycles and through the creation of efficient topologies. A limit based query processor reduces energy usage using advanced declarative query operators such as group-by and top-k. Other approaches include SINA (Srisathapornphat et al., 2000), Senseive (Hermann et al., 2008), COSMOS, (Kim et al., 2008), and Cougar (Yao et al., 2002). (Meena et al., n.d.) present a somewhat divergent approach which supports network querying using an XML query language: XQuery. XQuery is user entered and transformed into SQL internally. The embedded intelligence of these systems lies in query optimization which reduces network throughput. TinyDB and Cougar are capable of collection from a single WSN and are tied to hardware implementation. (Bagula et al., 2009) presents a gateway side usage of the paradigm in which a mySQL database represents the network on the gateway. The system described is minimal, but highlights the prevalence of the paradigm in the research literature.

**Table 2.5: Comparison of Database Middleware**

<table>
<thead>
<tr>
<th>Name</th>
<th>Heterogeneity</th>
<th>Reconfiguration</th>
<th>Extensibility</th>
<th>Energy Awareness</th>
<th>Scalability</th>
<th>Intelligence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cougar</td>
<td>×</td>
<td></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>SINA</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>TinyDB</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>DSWare</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>KSpot</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>XQuery</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Senseive</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

\textsuperscript{12} the sampling rate and duration can be modified subject to the accuracy required by the end-user

### 2.5.6 Virtual Machines

Within this middleware approach a Virtual Machine (VM) is placed on sensor nodes providing a layer of abstraction over the Operating System (OS). Virtual Machines are defined as software packages, or runtimes, which emulate an Operating System. The Java VM (JVM) (Venners, 1996) is illustrative of the
successful application of this approach. The JVM allows programmers to write software on one machine
and be assured that it will operate equivalently on other JVM platforms; this is referred to as sand-boxed
software execution. In essence a set of sensor programming primitives is provided to the programmer;
this offers a degree of hardware independence as scripts can be deployed to any device supporting the
VM (Kuorilehto et al., 2005).

Developers construct applications in small, separate, modules which are interpreted by the VM. This
interpretation is subject to some performance penalty due to the overhead introduced but is attractive as
it provides a common abstraction. Figure 2.12 depicts the high-level design of a generic sensor node VM
middleware. The two applications, samples from Mate (Levis and Culler, 2002) and SwissQM (Mueller
et al., 2007), are sand-boxed in the nodes VM and given access to node services such as sensors and
storage via programming primitives.

Reprogramming is the primary goal of the VM approach; to alter the behaviour of an individual node,
or group, the user injects a script to run on each VM node. Typically module distribution is cognisant of
energy and is tailored to reduce consumption. A VM can also be a useful insulation against performing
unsafe commands on the host machine.

Mate is notable VM implementation (Levis and Culler, 2002). Therein, a spectrum of network tasking
is possible. Tasking is defined as a modification of network or node behaviour. For instance, operational
parameters, e.g. sampling rate, may be modified on one or more nodes. To achieve the behavioural
shift messages are sent into the network from the gateway. Full re-programmability is also achievable
wherein a new script is injected into the network; this replaces the entire code base executing on the
nodes. Listing 2.3 depicts a mate script to show the bottom 3 bits of the counter on the sensor nodes
LED display. SwissQM (Mueller et al., 2007) is a similar system however, in contrast it features greater
support for adaptability, extension, and optimization. SwissQM utilizes a relation database querying
model for sensor networks (see Section 2.5.5); this is an example of the feature bleed which is common
in WSN middleware. The programs developed for SwissQM and Mate are syntactically similar. It is
uncontroversial to state that both are unintuitive and difficult to program for developers accustomed to
Java, C++, Ruby etc.

Listing 2.3: Mate Script to Flash LEDs (Levis and Culler, 2002)

```plaintext
1  pushc 1  # push one onto operand stack
2  add     # Add the one to the stored counter
3  copy    # Copy the new counter value
4  pushc 7
```
SensorWare: (Boulis, C. Han, and Srivastava, 2003) provides an abstraction of the sensor nodes OS which acts as an execution environment for scripts. Scripts are mobile moving through the network. Static applications are also present and are used to provide solutions for WSN needs such as node discovery. Scripts are written in an extension of Tcl (Ousterhout et al., 2009). The extensions define hooks into sensors and commands by which event based reasoning such as the wait command are supported; this is illustrative of the event-based paradigm discussed in Section 2.5.9. The runtime controls translation of messages to and from the scripts. SensorWare features data aggregation capabilities. Another similar effort is SmartMessages (Ravi et al., 2005) which encapsulates program code and execution state as a SmartMessage; these programs are routed to nodes of interest based on properties such as location or sensing capability. Program code may be cached on intermediate nodes to reduce network load. A shared memory model is used. VMSTAR (Koshy et al., 2005) generates resource efficient VMs for the Mica platform. These VMs are generated by a Java interpreter based upon application requirements and the capabilities of the platform. This system supports runtime updates for deployed applications and the VM. In a similar vein of work MagnetOS (Barr et al., 2002) attempts to abstract the WSN as a single JVM. The user writes a standard Java program which is then translated into components to be executed on specific nodes. This offering is indicative of “feature-bleed” as it could just as easily be placed among the macro-programming approaches discussed in Section 2.5.4.

DAVIM (Horre et al., 2008) models the sensor network as a minimal service platform and desires a strong separation of services from applications and concurrent applications from each other. The DAVIM middleware supports the concurrent execution of multiple VMs with the middleware isolating VM executions. The instruction sets of each VM may differ; instructions are mapped to a subset of DAVIM services such as messaging or storage. Pushpin (Lifton et al., 2002) views each node as a self-sufficient entity and within the system coordination between nodes should happen automatically. In contrast to the initial assumptions of TinyOS driven research of its contemporaries, in which all communication was bound for the gateway, much communication happens locally between nodes. Pushpins programming model is heavily influenced by paintable computing (Bove Jr et al., 2002); this model is governed by the notion of algorithmic self-assembly. Therein simple processes, involving local interactions with neighbouring nodes, combine to form complex global behaviour in service of application objectives. (J.-Z. Sun et al., 2009) specifies a Virtual Machine architecture for Communicating Finite State Machines. The goal is to empower network tasking through CFSM program code; each sensor node has a script interpreter.

Table 2.6: Comparison of Virtual Machine Middleware

<table>
<thead>
<tr>
<th>Name</th>
<th>Heterogeneity</th>
<th>Reconfiguration</th>
<th>Extensibility</th>
<th>Energy Awareness</th>
<th>Scalability</th>
<th>Intelligence</th>
</tr>
</thead>
<tbody>
<tr>
<td>MagnetOS</td>
<td>x</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Mate</td>
<td>x</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>VMSTAR</td>
<td>x</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>SensorWare</td>
<td>x</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>SmartMessages</td>
<td>x</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>PushPin</td>
<td>x</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>SwissQM</td>
<td>x</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

2.5.7 Agent-Based

In every day living an agent is defined as someone who is authorised to act on behalf of another; it follows that a software agent is a code bundle which performs functions for the user. Wooldridge & Jennings
(Wooldridge et al., 1995) formally define an agent as “An encapsulated computer system, situated in some environment, and capable of flexible autonomous action in that environment in order to meet its design objectives”.

Agents are responsive to environmental stimuli, provided via sensors. Through effectors an agent can take action. Agents control their internal state and behaviours. Such agents correspond to the weak-notion of agency proposed Wooldridge & Jennings (Wooldridge et al., 1995). The definition encompasses autonomy, interactivity, responsiveness to external stimuli, and proactive capability. For example, the sampling rate may be increased in response to unusual readings, or the agent may migrate to another node after a certain period. Figure 2.13 depicts the high-level system architecture of two agent-based WSN middleware solutions. This is included to underscore the commonalities present in how the authors present their architectures visually. Therein it is shown that these middleware mechanisms provide a node-level layer for the execution of one or more agents; agent communication and migration are also evident.

![Agent Middleware Architectures](image)

**Figure 2.13: Agent Middleware Architectures**

**Agilla** (Fok et al., 2005a, 2009) is based upon Mate (Levis and Culler, 2002). Herein, mobile agents are injected into the network rather than executable scripts as in systems such as SensorWare (Boulis, C. Han, Shea, et al., 2007). Listing 2.4 shows program code for an Agilla agent which detects fires. In Agilla multiple agents can reside on one node; agents can migrate. Communication is facilitated by remote access to the tuple-space of other nodes, this concept is discussed in Section 2.5.9. Consequently, Agilla is another example of the “feature-bleed” across approaches. This has been noted as hampering classification (Delicato et al., 2014).

**In-Motes** (Georgoulas et al., 2006) addresses perceived shortcomings of systems such as Agilla. Therein, mobile agents travel through the network via agent migration performing application-specific behaviour. All aspects of the application are encapsulated within agents. Facilitator agents are responsible for managing communication between agents; the nodes on which the facilitators reside is determined by a selection algorithm based on hop-distance. Facilitators receive job agents, if a node is not too busy the agent is run. If a node is too busy, the agent is passed to the next facilitator, to improve network longevity and spread the load. A shared memory model is used for asynchronous agent communication.

**Impala** (Liu et al., 2003) is modelled on the event-based programming paradigm. In Impala, the event filter agent propagates events to other components. These events include a packet being received, the successful sending of a message, sensor data, or device failure notification. An application must implement event handler hooks; these are invoked by the filter when a matching event occurs. The application adaptor agent is designed to receive and process events and if necessary adapt application behaviour. Intelligent routing management prompts changes in protocol in stances of node failure. In keeping with the application update functionality discussed in Section 2.5.6 Impala employs an update module to install new program code modules. Applications are designed in a modular fashion so that
transmission of updates and components will be considerably less costly than sending complete code. Impala agents are mobile; moving from node to node as necessary. **BISNET** (Boonma et al., 2007) applies biological principles to WSN design objectives such as autonomy, scalability, and adaptability. Each TinyOS node is modelled as a Multi-agent System (MAS) mapping agent behaviour to biological processes. As regards decentralisation and autonomy this mapping is direct as the aforementioned are core qualities of agents. The agent-driven gathering of sensor data is likened to food gathering and storage behaviour of animals. A lack of sensor observations, which are “energy”, results in the eventual demise of agents. Agents emit pheromones which encompass data for example an agent in transit (migration) emitting a pheromone displaying this intent. **MAPS** (Aiello et al., 2009) is another Agent-based solution. Within MAPS, agents run on a mobile agent execution engine. Application level agents are engaged with sensor monitoring and event detection. Middleware agents control discovery and data fusion. Network level agents are responsible for routing. Components are defined to control agent naming, asynchronous agent communication, migration, hardware access, and timing. **Sensomax** (Haghighi et al., 2013) and **Agent Factory Micro Edition (AFME)** (Muldoon et al., 2006) are two examples of other agent-based middleware which operate on Sun SPOT nodes. Midgard (Araújo et al., n.d.) also targets the Sun SPOT platform but presents as a modular service-oriented middleware (see Section 2.5.8). While agent technologies have been used at the fringes of the sensor network higher level frameworks also utilize the approach to provide control and co-ordination. One example is **IrisNet** (Gibbons et al., 2003). Therein the use of agents is two-fold. One set of agents manages the collection of sensor data while another set is responsible for the storage and organization of data across a distributed database. This has obvious echoes of the database abstraction described in Section 2.5.5. It should be noted that the agent-based approach is quite similar to the virtual machine model and can be viewed as a further restriction or abstraction.

<table>
<thead>
<tr>
<th>Name</th>
<th>Heterogeneity</th>
<th>Reconfiguration</th>
<th>Extensibility</th>
<th>Energy Awareness</th>
<th>Scalability</th>
<th>Intelligence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agilla</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>In-Motes</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>MAPS</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>SensoMax</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>Impala</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>IrisNET</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>BISNET</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
</tbody>
</table>

### 2.5.8 Service-Oriented

In this model sensor nodes are viewed as providing services to applications, users, and other nodes. This is, essentially, the application of Service-Oriented architectures to WSNs. Figure 2.14 conveys a generic SOA; therein clients register interest in a service through a broker. The broker informs clients of services matching their requests and the two then interact. The architecture espouses a loose coupling of services
and consumers such that implementations are substitutable. Some researchers believe the future lies in viewing “sensing as a service” (Perera et al., 2014). This is in line with the view espoused to deem all software as a service (Banerjee et al., 2011). (Mohamed et al., 2011) provides a thorough survey of SOA middleware in the sensors domain which identifies abstraction, transparency, reconfiguration, and interoperability as key for the domain.

Figure 2.14: Generic Service-Oriented Architecture

Adapted from http://www.w3.org/2003/Talks/0521-hh-wsa/slide5-0.html

Mires (Souto et al., 2004) is described through the lens of the publisher/subscriber design pattern. A sensor node advertises a set of services e.g. sampling of various phenomena. Applications register their interests, and only the specified phenomena are sampled, and the results returned to the sink nodes. Mires features multi-hop routing support and data aggregation services. The middleware has been extended with a group management component which introduces logical grouping functions (M. S. Vieira et al., 2005); groups are definable for diverse scenarios such as fault detection and object tracking. USEME (Caete et al., 2008) is an in-network SOM which composes WSN applications from lightweight services deployed on the sensor platforms. Services interact through a binding mechanism conceptualized as a port. Each service provides a port which can be bound by remote applications. All aspects of typical operation, e.g. data publishing, are included as services. MidSN (Cecilio, J. Costa, et al., 2013; Cecilio and Furtado, 2012) is an implicitly SOM whose composition is described as “lego-like”. Heterogeneous sensor nodes are overlaid with a homogeneous command-driven API. Implementations have been realised, as device drivers, for Contiki and TinyOS sensor nodes, and generic gateways (PCs). The messaging model for communication and configuration supports (i) command (ii) acknowledgement (iii) task completion (iv) sensory observations. Gateway-side components are implemented to link the sensor networks with root nodes using the generic messaging protocol.

SenSer (Paulino et al., 2011) provides access and control of sensor networks over the WWW. The functionalities of the networks are virtualized as services. The system architecture is composed in a three-tier model. The data layer is responsible for device connectivity and persistent storage. The logic layer, provides filtering, notification, and query management to the client facing layer. The presentation layer is separated into two parts, the first for standard user interaction and the latter for administrative control of the network. OASIS (Kushwaha et al., 2007) is a SOM running on Mica2 motes. The design is grounded in firm support of the separation of concerns. Core functionality is written by domain experts and bound to application specific behaviour to form node-level code. OASIS offers a set of services such as node management and data flow. A discovery service utilizes broadcast request messages to gather the details of services which are offered by other nodes. The flow of information is managed through composers that specify input and output at each stage; the node management service handles network communication. Interactions are decoupled through the service model and well-known interface layers. Related work (Mainland et al., 2008) presents Flask which provides a dataflow programming model. Applications are constructed by chaining operators which control the flow of information across multiple
nodes; this may define filtering, aggregation, and context specific forwarding. As is symptomatic of feature bleed and the convergence of approaches several of the solutions discussed subsequently in Section 2.5.9 and Section 2.5.10 espouse service-orientation and loose coupling of functional components.

Table 2.8: Comparison of Service-Oriented Middleware

<table>
<thead>
<tr>
<th>Name</th>
<th>Heterogeneity</th>
<th>Reconfiguration</th>
<th>Extensibility</th>
<th>Energy Awareness</th>
<th>Scalability</th>
<th>Intelligence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mires</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Oasis</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MidSN</td>
<td>✓ ✓</td>
<td>✓ ✓ ✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>

2.5.9 In-Network Middleware

In this section, a discussion is given of in-network middleware which do not fit precisely within the above-identified abstractions. The section begins by describing a set of common abstractions which appear in many WSN middleware.

Common Concepts

- **Publisher/Subscriber**: Utilizes the publisher/subscriber design pattern to support asynchronous communication typically between data producer and recipient. Figure 2.15 depicts the generic pub/sub mechanism highlighting the decoupling of publisher and receiver. As noted by (Eugster et al., 2003) decoupling may manifest temporally, in the removal of explicit knowledge or in removing synchronisation concerns. This event-based paradigm is natural for the problem domain and promotes a loose-coupling through asynchronous callbacks; in contrast a request/response paradigm is wholly unsuited to WSN sensing applications.

![Figure 2.15: Generic Publish-Subscribe Mechanism](image)

Adapted from “The Many Faces of Publish/Subscribe” (Eugster et al., 2003)

- **Tuple-Space**: This is a shared memory model used in distributed computing applications and widely applied in WSN middleware. This is also known as the “blackboard” model and is often applied to agent communication (De Nicola et al., 1997; Hildum et al., 1997); agent middleware such as Agilla and In-Motes utilize tuple-spaces. The tuple-space is shared among some set of entities that must share information. Figure 2.16 depicts the concepts as applied to WSN nodes; therein nodes share their observations through the transient tuple space. Information can be posted to and read from the tuple-space. Tuple-spaces are useful in decoupling communication concerns. It is typical for a consumer to register their interest in tuples that match a pattern.

![Figure 2.16](image)
• **Event-Driven:** An event-driven approach is closely related to the publisher/subscriber paradigm however it is an appropriate metaphor for sensor-based applications inherently focused on event detection. Event detection can involve determinations made from several related sensor observations. Event detection schemes are often built upon pub/sub with a further layer utilized to filter messages for those meeting a criteria e.g. Temperature greater than 35 degrees.

• **Application-driven / Quality of Service:** This common feature set allows the application-level to tune the operational parameters of the network so as to best fit it’s own needs. Some authors argue this can lead to too-tight a coupling (Delicato et al., 2014), whereas database and macro-programming approaches are considered too loosely coupled in their execution of requirements. There is need for mediation in this layer to align the competing QoS desires of multiple applications. MASTAQ and MiLan, discussed below, feature this property strongly.

Having considered these common domain concepts a review of in-network sensor middleware offerings is now provided.

**TinyCubus** (Marrón, Lachenmann, et al., 2005) provides an abstraction over TinyOS. Therein, a data management layer weighs network longevity concerns, application needs, and system properties to define suitable behaviour for subcomponents. A cross-layer framework facilitates modularity by acting as a mediator in communication enabling dynamic callbacks to components. The configuration engine is responsible for the distribution of program code and the specification of a nodes function which is chosen based location and capability. **TAG** focuses upon sensor data aggregation (Madden et al., 2002) resolving gateway queries within the WSN. **GridMap** (Jiang et al., 2009) defines a logical region for posing queries over a virtual region which corresponds to a physical section of the sensor network. for instance, by selecting sensors which are located within a threshold distance of the specified interest points. **Hood** (Whitehouse, Sharp, et al., 2004) groups nodes using logical-regions. Therein the user defines criteria for the formulation of groups. Hood handles the region management including supervision of neighbourhood lists and information sharing. It is evident that this partitioning provides a powerful abstraction which is understandable to the end-user. **TS-Mid** (Cassia Acioli Lima et al., 2008), **SPIDEY** (Mottola et al., 2006) and **Envirotack** (Abdelzaher et al., 2004) are also logical grouping systems. Envirotack is concerned with tracking; a typical group encompasses sensor nodes attached to, or near, an animal. For example such a logical group would be capable of monitoring several parameters of the animals health e.g. heart rate, blood pressure, location, and level of movement. **Abstract Regions** (Welsh et al., 2004) explicitly defines sensor node grouping by network topology and geographical distance; such groups are sometimes distinguished from logical groups as “neighbour-hood groups”. **Listing 2.5** depicts the creation of a region in Abstraction Regions; when a region has been created local variables can shared in the tuple-space.

```
region = k_nearest_region.create(4);
```

**Figure 2.16: WSN Tuple Spaces**
MASTAQ (Hwang et al., 2005) provides application with a desired Quality of Information (QoI). This QoI is based upon the observation that in many application scenarios data is combined from diverse sources to facilitate event detection. There is a trade-off between accurate detection and the energy resources consumed. An application poses a query with a modality, location, and duration. An application specifies its QoI requirements on the basis of accuracy; those demands dictate how many sensors are necessary to retrieve the information. Energy is managed intelligently to utilize only the necessary subset and perform duty cycling. In a similar vein Milan (Heinzelman et al., 2004) was designed to provide Quality of Service (QoS) and utilize proactive adaptation. As in MASTAQ, an application specifies a necessary data quality, and the network is adapted to achieve this goal. The system attempts to strike a balance between lifetime and data quality.

Active Messages (Buonadonna et al., 2001) provides an abstraction of low-level radio communication for TinyOS; the implementation is decomposed in to three separate modules. Differing module implementations are chosen by developer to service specific aspects of the network. The message model is an asynchronous Publisher/Subscriber solution. Each Active Message contains a data payload and is cognisant of the handler, residing on the target node, which is to be invoked on arrival. Handlers fold messages into the nodes computation and, optionally, fire a response.

Generic Role Assignment (GRA) (Romer, Frank, et al., 2004) targets the issue of large-scale network reconfiguration. The assertion is that in a large network the user cannot perform task assignment on each node. As such the sensor nodes tune behaviour autonomously. A node’s behaviour is informed by its state and the state of its neighbours. Each node exposes its capabilities and remaining resources; this information facilitates co-ordination to accomplish the application goals. This autonomous collaborative communication echoes the messaging paradigm of Agent-based systems (see Section 2.5.7).

Facts (Terfloth, Wittenburg, et al., 2006) is designed with minimalism and generic applications in mind. The system is governed by the notion of information or “facts” that are stored in a node repository. A fact can trigger the execution of a rule. A rule actuates behaviour such as increased sampling frequency. This model can be seen as an extension of tuple-spaces that explicitly defines actuation in response to “facts”. Also utilizing tuple-spaces are TinyLime (Curino et al., 2005) and TeenyLIME (P. Costa et al., 2006) which are based upon LIME (Murphy et al., 2001). TinyLime provides support for mobile devices such as PDAs to connect to nearby base stations and collect the sensor data. A key differentiation of TeenyLime is the focus on moving reasoning capabilities into the network. Generic tuples, which advertise capabilities and data samples, such as ("Temp", ?int, ?float) can fire a reaction when a matching sample is added to the tuple space. An energy-cost analysis determined the constraint that each node should only share its tuple-space with one-hop neighbours.

The FamiWare (Gámez et al., 2011) middleware supports several hardware platforms through a Software Product Line approach. The software can be deployed on several device types while providing a common API. The middleware is designed to be highly modular. The deployment of code is supported by a middleware generator which can produce a system with a custom set of services. Implementations for Android and TinyOS are given as an example of this approach in action. Funf (Moturu et al., 2011) also targets Android devices allowing the user to develop a tailored data collection application for their device. The information gathered is then transmitted to the Dropbox[15] account of the user. Funf enables the collection of many modalities including GPS, cell tower ID, SMS logs, and battery level. In a similar vein POGO (Brouwers et al., 2012) supports Android, however, POGO also operates within traditional personal computer architectures; the intention of the platform is as an enabler for large-scale scientific data collection utilizing the sensing infrastructure of the modern mobile device. Ubiquitous devices such

as smart-phones are seen as a gateway toward large-scale high-resolution sensing. POGO exposes a 11
method API for JavaScript client applications. This approach was chosen over more explicit specification
as it is difficult to provide explicit methods for every sensor. A Publisher/Subscriber paradigm is adopted
for data delivery; intelligent optimizations are employed such as ceasing sampling when the subscriber
expires. XMPP, originally defined for instant messaging, is used as a communication protocol between
devices encompassing peers in a “buddy” list. As with POGO, MoSen (Bakhshi et al., 2013) has been
developed for mobile devices, and features a single sensor access API replicated across platforms. MoSen
is modular; a sensor module for each resource e.g. the GPS or the proximity detection modules. A
sensor manager exposes sensors to client applications. Event-based data delivery is supported under a
subscription paradigm as seen previously in Mires. TinyMQ (Shi et al., 2011) presents sensor nodes
cast in the dual roles of publisher and subscriber. TinyMQ builds upon previously efforts in the area;
TinyDDS (Boonma et al., 2009), PUB-2-SUB (D. A. Tran et al., 2009), and Mires. TinyMQ also provides
an abstraction of the sensor network through an overlay in which each node is uniquely addressable.
Nodes may be logically linked in the overlay as in the previously discussed logical grouping concept. The
publisher/subscriber layer provides routing and capability mapping for subscription and transmission in
the physical network.

Middleware platforms have been developed specifically to assist AAL including SAM (Wolf, A. Schmidt,
and Klein, 2008), OASIS (Amundson et al., 2006), GAL (Eichelberg et al., 2010), and openAAL
(Wolf, A. Schmidt, Otte, et al., 2010). openAAL is a flexible component-based middleware showcasing
variant behaviour based upon installed bundles. The openAAL work-flow moves from event detection
to an invocation of a service controlling actuation in response to an identified situation. GAL is a
service-oriented middleware which integrates BAN sensors with environmental sensors and combines
the information from these sources to provide context to identified events. The SOPRANO Ambient
middleware (SAM) shares a common goal of providing a loop from sensory input to actuation. The
OASIS project is an agent based offering attempting to facilitate the sharing of content between services
in domains relevant to elderly patient care. Middleware architectures are also an important consideration
for a Body Area Network (BAN); consider two such relevant examples: Personal Wireless Body
Area Network (PWBAN) (Waluyo et al., 2009) and Self-Managed Cell (SMC) (Keoh et al., 2007). The
PWBAN offering is designed to acquire sensed data and deliver it to user-level applications. Elements
of security, node tasking, and resource detection are supported. SMC describes an architecture akin to
a software agent in which each SMC is autonomous and is reactive to current user activity. A discovery
module maintains contact between neighbouring SMCs. A policy service governs the SMC reaction to
identified events.

Table 2.9: Comparison of In-network Middleware

<table>
<thead>
<tr>
<th>Name</th>
<th>Heterogeneity</th>
<th>Reconfiguration</th>
<th>Extensibility</th>
<th>Energy Awareness</th>
<th>Scalability</th>
<th>Intelligence</th>
</tr>
</thead>
<tbody>
<tr>
<td>TinyLIME</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TeenyLIME</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TinyCubus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MASTAQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GRA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Facts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FamiWare</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Milan</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Funf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hood</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pushpin</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pogo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MoSen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TS-Mid</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.5.10 Gateway Middleware

Gateway-Side middleware platforms represent a higher-level abstraction than in-network middleware and their major system components exist outside of any particular sensor network. The middleware interfaces with the network through one or more gateway sensor nodes. This approach reduces coupling with the underlying network and assists in utilizing multiple networks simultaneously.

Services provided by this approach may include web publishing and access, data access, WSN monitoring, and autonomous actuation. Such systems typically allow for visualisation and reprogramming of the network. This class of system can utilise networks programmed using middleware discussed in preceding sections by interfacing with that middleware layer; this system-of-systems approach is found to be advantageous (Bröring et al., 2011).

Lower-Level

**Sensor Network Services platform (SNSP):** (Sgroi et al., 2005) defines an abstraction of the underlying physical network. Application subcomponents, termed controllers, communicate with sensors, actuators and each other via this abstraction. Communication is subdivided for querying and issuing commands. A notion of virtual sensing is defined as a data source collating information from other sources and optionally applying some transformation. A virtual sensor is defined to encompass connection to external data sources such as on-line weather forecasts; virtual sensor also represent aggregation of multiple raw sensor values. Services are assigned responsibility for global synchronization and network capability accounting. A peer-to-peer approach enables an application using one SNSP instance to gather data from another.

**Sensor Web Enablement (SWE)** The SWE framework (Botts et al., 2008), developed by the Open Geo-spatial Consortium (OGC), provides a set of web service APIs and protocols for communication, which abstract the details of heterogeneous interactions with and between sensor networks. The 52 North Sensor Web framework (Jirka et al., 2009) and ICTA Open Sensor Web Architecture (NOSA) (Kobialka et al., 2007) both implement the core SWE standards such as SensorML and Observation and Measurement (O&M). SensorML provides a model and XML encoding for sensing processes. Therein a sensor is defined as a process capable of observing and conveying that observation. The Observation & Measurement (O&M) standard is used for interpretation of sensor observations and integration. The Sensor Collection Service is used to gather observations; planning and notifications standards are also implemented. **Sensor Web 2.0** (Mandl et al., 2008) merges SWE with a RESTful architecture. The use of SWE standards has been limited, in some part due to their verbosity and the associated overhead. **TinyREST** (Luckenbach et al., 2005) exposes sensing and actuation resources through REST. Recently the OGC has defined a candidate standard termed **SensorThings** (Jazayeri, 2014), for the specification of sensor interaction through RESTful means, this new standard is decidedly more lightweight than SensorML and O&M. The OGC acknowledges the process view espoused in O&M is not entirely compatible with the “Things” view advocated in the IoT vision; SensorThings is more suitable in this way. (Rouached et al., 2012) propose the use of a more minimalist JSON representations of SWE concepts through a RESTful API as a solution to SWE verbosity. The authors demonstrate a reduction in description length, and an increase in readability. In support (Nurseitov et al., 2009) presents a study of the merits of JSON with respect to XML concluding that JSON is more resource-efficient and provides increased readability.

**Global Sensor Networks (GSN)** (Aberer et al., 2006a,b) is developed as a peer-to-peer architecture. A key abstraction is the virtual sensor (VS). Virtual sensors are abstractions of either data streams from physical sensors or a stream produced, or derived, from many other virtual sensors. This abstraction

---

16JavaScript Object Notation
enables the creation of aggregation nodes which can in turn be used by other VS instances to facilitate data processing encapsulation. A virtual sensor is specified in XML, wherein its inputs, addressing information; outputs and processing are specified. The virtual sensors in GSN share commonalities with the Kabadayi (Kabadayi et al., 2006) and VNLayer (M. Brown et al., 2007) definitions. These approaches, particularly VNLayer, also have a strong relationship to grouping abstractions. A virtual node in VNLayer provides stability by representing inputs from several related physical nodes. Kabadayi (Kabadayi et al., 2006) and GSN describe virtual nodes as potentially having multiple inputs mapped to a single output. This convergence within the literature suggests the applicability of these abstractions.

The goals of GSN are simplicity, adaptivity, scalability, and resource conciseness. A minimal set of configurable resources supports simplicity, wrappers support adaptivity to new data sources, the P2P architecture supports multiple publishers and consumers. The implementation is designed for a standard desktop machine in line with resource conciousness. Queries are specified in SQL. A manager service is responsible for the routing of data streams from heterogeneous sources to the virtual sensors. Listing 2.6 depicts the XML configuration of a virtual sensor gathering location data from an external device; this sensor has a single input stream, and a singular output format. In GSN, a wrapper is defined for connecting diverse data sources, for example, through serial connections, over HTTP or IMAP. As GSN is decoupled from the data source it is possible to define wrappers for non-traditional sensors such as a stream of emails. MOSDEN (Perera, Jayaraman, et al., 2014) was designed with GSN as its basis. The platform runs on “resource constrained” Android devices. Plugins are utilized to connect sensor networks to the MOSDEN core. The major architectural change, from GSN, is the addition of plug-in management and the plug-in layer. As with the developers of GSN, MOSDEN researchers espouse a “zero-programming” approach. This builds upon the GSN model by removing the need for users to write declarative XML virtual sensors, of which Listing 2.6 is an example. It would appear that in doing so flexibility is reduced and ease of use increased (provided use stays within supported parameters). As is evident throughout the authors research this trade-off frequently recurs. The Context Aware Sensor Configuration Model (CASCoM) (Perera, Zaslavsky, Compton, et al., 2013) system implements a semantic-driven data retrieval model for non-technical users within GSN.

\[\text{Listing 2.6}\]

\[\text{As an aside, it would be prudent to design a metric to measure what is and is not resource constrained}\]
Construct (Coyle, Neely, Rey, et al., 2006; Coyle, Neely, Stevenson, et al., 2007; Dobson et al., 2007) is a smart-home middleware solution for fusing multiple information streams from diverse sensory inputs and making the data available to user applications. Construct is explicitly permissive of observations gathered from web-resources in addition to traditional sensor devices. Five core services are provided (i) Discovery (ii) Management (iii) Sensing (iv) Actuation (v) Distribution. The middleware promotes a peer-to-peer architecture in which multiple instances of construct exchange sensory information such that each instance has a local view of the global data set. Service resolution is facilitated via an implementation of the zero-conf protocol (Guttman, 2001). RDF triples are used to model the sensory observations. The implementation is light-weight and the service advertisement model, via WSDL, allows sensors to dynamically establish contact with construct instances. Within SenseWrap (Evensen et al., 2009) virtual senses are representative of any physical sensor. This is somewhat restrictive when compared with self-identified nearest neighbour Construct which advocates representation of all data sources. As in Construct sensor discovery is powered by zero-conf. Owing to limited computational capabilities and the battery issues, earlier identified as plaguing the domain (see Section 2.3.3), the full stack uses gateways to house the virtual representations. The adaptor pattern is employed on the client-side for interaction over diverse protocols. The protocol adaptors interface with services which act as wrappers around a single virtual sensor. This is presumed to service a separation of concerns but seems odd from an architectural standpoint.

Atlas (King et al., 2006) is a supporting framework for pervasive computing environments. As in GSN the architecture is intended to decouple applications from physical sensors. Implementation is based upon OSGi; OSGi is a plug-in and modularity framework for Java, which can be seen as an enabler of service oriented architecture concepts in Java. OSGi allows for run-time extensibility and explicit modularisation through bundling and resource sharing via the service bus. Modularity promotes the
decomposition of large programs into bundles which are maintainable and reusable. The node layer
encapsulates sensor platform connectivity and creates a service to represent each real-world node. OSGi
explicitly supports service oriented architecture concepts. **SStreamWare** (Gurgen et al., 2008) also
utilizes OSGi as its basis. SStreamWare obscures heterogeneity from applications via query based services.
SStreamWare is capable of interaction with a dynamic set of sensor platforms which are interacted with
through adaptor services. The system presents a distributed database view of the network, as previously
discussed in Section 2.5.5. Client interactions with SStreamWare is restrictive in that it is performed
through a predefined GUI. **Sensor Node Plug-in System (SNPS)** (Di Modica et al., 2014) features
an architecture which utilizes virtual sensors composed of many raw inputs as previously discussed (Aberer
et al., 2006b; Kabadayi et al., 2006). OSGi is utilized for the same purposes as in Atlas and SStreamware
but is also extended for remote communication through R-OSGi. The data model of SNPS is based on
a restricted version of the SensorML OGC standard (Botts et al., 2008); JSON is used to represent data
as it was found to be less verbose than XML. **WSNWare** (Viani et al., 2013) Further cementing the
applicability of OSGi is its usage herein. The WSNWare framework is designed to support heterogeneous
platforms. Support is delivered by information source connection modules. WSNWare does not provide
any device code for its supported platforms; the application developer must define WSN messaging and
interpretation. WSN-ware lacks any query based functionality, approach to security, or incorporation of
cyber sensing. WSNWare is a minimalist framework with a solid object-oriented domain representation
and a cohesive code-base. **UniversAAL** (Ram et al., 2013) aims to provide a common framework for
Ambient Assisted Living (AAL) technologies. UniversAAL represents a consolidation of previous projects
including SOPRANO (Klein et al., 2007), Persona (Tazari et al., 2010), and OASIS (Kushwaha et al.,
2007). The architecture promotes a familiar adaptor model to incorporate heterogeneous devices. OSGi
is utilized to facilitate application and device transience without platform restart. Like GSN individual
deployments communication peer-to-peer; herein a multicast technology, jGroups (Ban et al., 2002), is
used for peer discovery. OSGi is deployed on capable devices, where this is not achievable an adaptor
facilitates connection.

**ManySense** (Westlin et al., 2014) is designed as a unifying framework for Android wearables so as to
provide a rich-model of an individuals context. As noted in Section 2.2.1 context is a key enabler for
Ubiquitous Computing. Consequently and cognisant of competing hardware, communication models, and
standards, the middleware is intended to be extensible. The **adaptor design pattern** is employed to fit
the incompatible interfaces of wearable sensors into the framework[19]. In this model adaptors exist on
a per sensor basis rather than in GSN, SStreamWare, or WSN-WARE wherein an adaptor exists for a
network of devices. ManySense advocates the same flexible and relaxed view of a sensor as advocated in
Section 2.3.1 Section 7.7 describes a adaptation of the thesis contributions which places the developed
middleware in Android in the same broad manner as ManySense[20].

**LinkSmart** (Badii et al., 2010) assists in the integration of sensing devices and ambient systems. The
two main modules are responsible for context management and data acquisition. Context is defined for
devices, semantic context such as environment and location, and application specific context. Linksmart
takes a web services approach to connecting heterogeneous systems. **TinySOA** (Avilés-López et al.,
2009) provides a Service-Oriented architecture (SOA) for WSNs; abstracting the heterogeneity of sensors
by encapsulating their capabilities as services. The functionality, which is both in-network and external,
is delivered through four primary components: 1. Node 2. Gateway 3. Registry 4. Server. A Node is
the SOA software which runs on each sensor node in the network; node to node communication is also
conducted through a service metaphor. A Gateway is the management bridge between sensor networks
and external applications. This is typically deployed on more computationally capable devices such as
desktop computers. The registry stores information about all known gateways and associated networks.

[19] Indeed, ManySense is in the small minority in explicitly framing design decisions using patterns
[20] Headless background android services with publish/subscribe interaction
The registry is utilized by the server to locate the sensor networks; thereafter the server provides access to each network via a web service.

**SenseWeb** (Santanche et al., 2006) provides an uniform representation of sensing resources that can be shared across applications. Its central component is the coordinator that is made up of a tasking module and a database. The tasking module receives application requests and attempts to fulfill them. The database pushes data to applications. Intelligent caching is utilized to reduce network overhead by determining if an applications request may be solved, at least partially, by cached data. Sensor gateways connect to the physical network and provide a uniform abstraction to those devices. Transformer units are defined to perform data analysis, feature extraction, and uniformity. Such units can convert between unit or perform more complex processing such as feature extraction from a video stream. **SensorMap** (Nath et al., 2006) is a web-portal which displays sensor data published via SenseWeb.

**The Linked Sensor middleware (LSM)** (Le-Phuoc, Quoc, et al., 2011) endeavours to harness the qualities of Linked Data (Bizer et al., 2009) for real-time sensed data, or Linked Stream Data (Sequeda et al., 2009) to provide context awareness. LSM employs modular wrappers to connect with data sources and transform the raw data feeds into linked stream data. Wrappers are defined for direct connection to data sources and for coupling with an intermediary for example connecting to another middleware such as GSN. Querying is performed through two separate engines; a linked data query processor and a CQELS (Le-Phuoc, Dao-Tran, et al., 2011) engine which enables real-time persistent querying. LSM contains functionality to allow users to input static XML data sets which are lifted to linked data, during the process the user may further annotate the data, for instance, specifying units used in data collection e.g. Celsius or Fahrenheit. Within the wrappers a time independent semantic layer specifies the components of sensing devices and their relationships, as well as their capabilities. These properties are then tied to shifting values over time. Recently other middleware platforms targeting the IoT have espoused enriching raw sensor observations with semantics, one such example is the CA4IOT system (Perera et al., 2012) which was deployed on a university campus sensor network.

**Hourglass** (Shneidman et al., 2004) is an internet-scale middleware focusing on pulling data from heterogeneous sensor networks. The flow of data is seen as a circuit specifying the path between producer and consumer; this follows a publish-subscribe paradigm. Communication with a sensor network is provided via services; other services make-up elements of the data flow such as buffering, filtering, and aggregation. The circuit model resembles that of pipe programming particularly in reference to the transformation services included in intermediate stages of data delivery.

**Hi-Fi** (Franklin et al., 2005) is designed for high fan-in networks. High fan-in networks are edge heavy as with sensor networks having hundreds of nodes. Stream-query processing, described in SQL syntax, is used at every level of the system. Queries on data are performed continuously and utilized for event detection. Query complexity is envisaged as increasing from the edge in e.g. from a simple single modality threshold to complex multi-modal comparison and selection across a large data set.

**Smart-M3** (Honkola et al., 2010) describes an information brokering framework which targets smart spaces, which are likely to be delivered and controlled through WSN technology. Therein, the scenario described is of a mobile smart phone controller which receives data from, and actuates, local area sensor devices. Devices are abstracted as *knowledge processors* which communicate via the semantic information broker. This communication is in the vein of a publish/subscribe paradigm, and the broker presents akin to the tuple-space model discussed in Section 2.5.9 (Laukkarinen et al., 2012) to make the judgement that Smart-M3 is more of a in-complete scaffolding framework (see Section 2.4) than a complete middleware solution.

**Perla** (Schreiber et al., 2012) is both a SQL-like language and a middleware architecture. The language defines how data should be gathered and processed, and the middleware provides a runtime architecture. Connection with heterogeneous devices is achieved through the provision of XML-based description files.
The XML is passed to a factory that constructs an appropriate \textit{Functional Proxy Component} to manage device connection through a homogeneous interface. Perla supports low-level queries that define a single devices behaviour and higher-level queries which act upon the streams produced by the lower levels.

\textbf{Lamses} \cite{Jeong.2010} describe a gateway middleware for context-aware application interaction with any sensor network. As is a common theme in middleware of this ilk the heterogeneity of the network is masked, through a homogeneous interaction layer. The architecture has been described as complex by other scholars \cite{Laukkarinen.2012}. The primary architectural elements manage the network, meta-data, state, and the context awareness engine. As in GSN there is a heavy usage of XML, for the description of queries and the storage of sensor data.

\section*{High-Level}

The following section provides a discussion of a number of higher-level gateway-side middleware frameworks. There are characterised as primarily being WWW portals.

\textbf{Xively} \cite{K.-p.Yang.2013} is a minimal IoT middleware offered through the web; sensor data is pushed to the servers via a RESTful API. All public sensor feeds are accessible via the API or website. Numerical data is automatically graphed by Xively. The data sources found on Xively are diverse ranging from Twitter feeds to reports on backyard gardens. \cite{K.-p.Yang.2013} details the usage of Xively as an underlying framework for a car parking application. Similarly, \textbf{ThingSpeak} \cite{Team.2013} is an online sensor database platform. ThingSpeak users can register, create a channel, and upload sensor data. As in Xively graphing functions are provided, and Google Maps is integrated to display the location of sensor resources. \textbf{ThingSquare} \cite{http://thingsquare.com} mirrors the aforementioned and aims to provide a cloud back-end for smart sensors and their Smartphone controller applications. This supports reconfiguration of devices that are not targeted in other solutions. Other solutions in this area include OpenSense \cite{http://open.sen.se/} and SicsthSense \cite{http://sense.sics.se/explore}.

\textbf{Onion} \cite{https://onion.io/} provides both hardware and a cloud framework for connecting sensors and actuators to the internet. \textit{Onion connect} is a collection of firmware that allows devices to connect to the \textit{Onion Cloud}. The cloud services acts as a conduit for device interaction and a platform to which expensive computation can be delegated. Finally, an application development kit is provided to allow developers to build applications that interact with Onion devices. Commercial concerns such as Gimbal and Sensaris are manufacturing hardware and accompanying smart phone applications which can gather data from and control their sensor devices. Revolu is a commercial middleware solution for smart home technologies which provide control of many third party devices through a single smart phone application.

\section*{2.6 Conclusions}

This chapter has provided an overview of the core concepts within the area of sensor networks; subsequent to this a review of sensor middleware approaches was conducted in which key conceptual divisions between systems were identified. A taxonomy of the different approaches was presented. The background knowledge presented here informs the identification of desirable features that are given in Chapter 3. This literature review serves to illustrate that the research community has not converged on an optimal approach to sensor middleware, inside, or outside, the network.

\textit{Operating systems} are viewed as the lowest level of middleware in a wireless sensor networks context. Operating systems provide an abstraction for communication and sensor access. In most \textit{in-network}
middleware a layer of further abstraction is placed on top of the operating system in line with goals of that middleware. An operating system level abstraction is unsuitable for the demands of a Sensor Web middleware.

**Sensor Databases** provide a familiar and easily understood metaphor for working with sensors however systems of this kind such as TinyDB (Madden, Hellerstein, et al., 2002) are tied to a specific set of hardware. There is no capability to extend these systems to add new information sources or features such as data aggregation or prediction. Sensor databases are best suited to applications interested in querying the network externally; they are unworkable in scenarios requiring fine-grained node-level control.

Virtual machine and agent-based approaches provide a good model for changing the behaviour of the network or individual nodes. Within the virtual machine approach new code can be dispatched to run on each nodes virtual machine. Virtual machines are tied to specific hardware and require significant implementation time to be reconstituted for new platforms. Agents provide a good approach for reasoning both within the network as a non-human external controller.

In Section 2.5.9 many disparate middleware platforms were discussed each of these has its own merits and limiting factors. Architectural concepts from such systems were found to be useful in designing the middleware that is the subject of this thesis. These include the usage of logical regions and the application of familiar design pattern philosophies.

The view taken in this thesis is that a middleware platform needs to exist at the level of multiple networks this approach is typified by gateway-side middleware as discussed in Section 2.5.10. This allows for the most-diverse set of information sources that gives the ability to fuse that information. Client applications of such systems can receive updates from many sources and configure multiple networks. Whilst higher level gateway-side solutions exist, these are typically web services such as Xively (K.-p. Yang et al., 2013) and ThingSpeak (Team, 2013), which consume information and provide graphing applications. These services are not of the kind to be developed herein. Indeed these may be thought of simply as indirect information sources for consumption by another middleware service. Platforms such as GSN (Aberer et al., 2006a), WSNWare (Viani et al., 2013), LinkSmart (Badii et al., 2010), and OASIS (Kushwaha et al., 2007) are representative of the lower level of gateway middleware that connect directly with sensor networks and provide actuation. Such systems are typically more rich in features, and better placed to assist in a wider range of application scenarios. It is such a lower-level gateway middleware that this thesis seeks to produce. In the forthcoming Chapter 3 more details are given on the set of features such a middleware should possess.
3 | A Middleware Desiderata

“There is no virtue like necessity.”
– William Shakespeare, Richard II (Shakespeare, 1595)

3.1 Preface

In Chapter 2 a taxonomy of WSN middleware approaches was identified comprised of Sensor Databases, Virtual Machines, Agent-Based, Gateway Middleware, Macro-Programming, Operating Systems and other miscellaneous approaches. Each approach offers its relative strengths and limitations. The implementations underpinned by these approaches differ in terms of sensor platform support, reconfiguration capabilities, abstraction level, and extensibility. In the following sections, a high-level overview of the qualities of these approaches is provided.

Sensor database approaches, typified by TinyDB (Madden et al., 2005), provide a familiar metaphor and a suitably high-level syntax, extended from SQL, to empower the user to query the network. The notion of a query is a powerful abstraction and is suitable at all middleware levels. Sensor databases are tied to specific hardware and are somewhat restrictive in fine-grained reconfiguration; the addition of new configuration options requires a significant re-write of the system.

Virtual machine and agent-based approaches, typified by Mate (Levis and Culler, 2002) and Agilla (Fok et al., 2009), provide an adequate level of abstraction from the underlying hardware. These systems allow for the compartmentalisation of code updates such that programs are decomposed into smaller modules. Systems in this class tend to be tied to specific hardware or an intermediate OS. Gateway-Side middleware, typified by GSN (Aberer et al., 2006b) and WSNWare (Viani et al., 2013), exist at a higher-level. Most components of such systems exist outside the underlying network. Such systems are much more capable at interacting with heterogeneous hardware. However, at times this can mean that the cumbersome task of lower-level sensor programming is left to the end user. Low-level sensor programming is seen as difficult, and error prone, as the limited outputs of sensing devices render debugging problematic. Within this approach, the need for an end-to-end solution which supports an endless array of sensing platforms is necessary for the applications of the future.

Each approach represents an attempt to solve the larger issues of WSN middleware design. A gateway-side system provides a more appropriate mechanism to support heterogeneous information sources and can be seen as a higher-level abstraction providing uniform resource access. In agent-oriented approaches, the prominence and support for intelligent reasoning are evident. Support for intelligent algorithms is highly beneficial to network longevity, primarily in reducing the usage of radio communication through data reduction, estimation models, variable routing, and other schemes. Typically within WSN middleware the interconnection of other external data sources such as web service application programming interfaces (API) are not treated as first-class citizens. The term first-class refers to the manner in which interaction and configuration are handled for the primary information sources. As the realization of the “Internet of
Things’ vision takes shape it is important to provide this interconnection so as to place all information streams in a unified interface. Middleware implementations such as the linked sensor middleware (Le-Phuoc, Quoc, et al., 2011) try to address this problem through the provision of wrappers for such resources.

These desiderata are viewed as a decomposition of the primary thesis objective given in Section 1.3.1; this goal identified the need for appropriate abstraction and domain modelling, this is echoed in the desiderata. The remainder of this chapter provides a breakdown of a set of desirable middleware qualities which should be present in any platform of the sensing domain. This desiderata was arrived from the background research discussed in Chapter 2; these represent our Holy Grail. Brooks identified a desiderata (Brooks Jr., 2010) as an essential set of “secondary objectives” for the design of software systems. Brooks felt that such were always implicitly present in the process but should be made explicit. It should be noted that this listing of desirable qualities is by no means complete and is focused toward the goals and research areas which are of interest to this thesis.

3.2 Heterogeneity of Source

It is a necessity that a middleware platform be capable of the collection of information from many heterogeneous sources. The necessity of this has been discussed at length within the literature notably by (Aberer et al., 2006b; Chatzigiannakis et al., 2007; Hadim et al., 2006; Romer, Kasten, et al., 2002). Interoperability functionality is much easier to achieve through an approach which sits on the fringes of physical networks such as GSN (Aberer et al., 2006b) or LSM (Le-Phuoc, Quoc, et al., 2011). New connections can be added into the system through extension points.

Within in-network approaches, such as Agilla (Fok et al., 2009) or In-Motes (Georgoulas et al., 2006), it is more difficult to expand onto other hardware as this necessitates the time-consuming creation of runtime facilities for the hardware. For this reason, in-network middleware has tended to be functionally tied to a single platform such as TinyOS (Levis, Madden, et al., 2005).

It is often true that data sources which should be fed through the middleware are unknown at the time of creation. Consequent to this the middleware should facilitate the addition, during runtime, of new contact mechanisms which are configured for the translation of sensory information and the injection of reconfiguration requests into the newly attached network. It is prudent also to define the information translation mechanism as modular so as to support the reception of disparate data formats over the same connection medium.

This thesis advocates the relaxed definition of a sensor that encompasses other information producing entities such as streams of Twitter data, RSS feeds, or content from static HTML pages. It is prudent to incorporate data acquisition and reconfiguration functionality for such content streams as a first-class citizen alongside physical sensor networks. This approach is advocated in other work such as LSM (Le-Phuoc, Quoc, et al., 2011) and is possible in other network fringe middleware such as WSNWare (Viani et al., 2013) or ThingSpeak (Team, 2013).

As these information streams can be considered as simply data from another source; a desirable middleware solution needs to accommodate them. It is paramount that a uniform interface be presented to facilitate reconfiguration of and acquisition from both physical and cyber-based resources. It is noted that the constraints that govern the usage of, and access to, diverse data sources are similarly varied. For instance, the usage of a cyber resource can be constrained by rate limitations of an API as with the Twitter API. Physical sensor concerns such as battery life have a strong bearing on their utilization.

1From the lesser used definition “a thing which is eagerly pursued or sought after”
It is the view of this thesis that heterogeneity is not going away (Gorgu et al., 2013; Gurgen et al., 2008; O’Grady et al., 2013; Viani et al., 2013). If all academic obstacles are removed and a homogeneous interface standardization for devices is attained the realities of the commercial economy will ensure seamless device interoperability remains difficult without a third party interfacing layer. There is too much invested in the “walled garden” ideology (Anderson et al., 2010), and this will only grow in the future as companies such as Apple continue to strengthen the links between their proprietary offerings to the detriment of interoperability.

**Heterogeneity of Source** is delivered upon by the network adaptor abstraction layer which is further discussed in Section 4.4.7. The adaptor abstraction shields the application developer from the low-level details of resource interaction such as a sensor network or a web API. To leverage previous work which is fit for purpose other middleware can be integrated using a connection mechanism in much the same way as a WSN or a cyber resource is handled.

### 3.3 Abstraction & Uniformity

In Section 3.2 consideration is given to the importance of connecting with multiple information sources. Building upon this, it is necessary that the complexities of dealing with the underlying network be abstracted from the user applications which are indirectly connected to them. This has been discussed within the literature (Terfloth and Schiller, 2005) as a driving force for sensor middleware development. Indeed it is apparent that this is a cornerstone feature of middleware across all disciplines (Bernstein, 1996).

This abstraction frees the application developer from concerns regarding the connection to the sensor network and the collection of sensor information. The entities that exist within a network and their properties should be represented through a set of consistent abstractions. Such entities typically include sensors, sensor nodes or cyber platforms, and observations.

The application facing interface for all sensor networks should be consistent, and it should be possible to execute some standard set of commands on all networks in a consistent manner. In particular reconfiguration commands should be consistent and the mechanism supporting this should provide some feedback to the invoking application. This feedback indicates if the request was carried out successfully, as some networks may exist in a read-only fashion or be rendered temporarily unable to comply.

Middleware platforms such as MoSen (Bakhshi et al., 2013) and FamiWare (Gámez et al., 2011) provide useful insights into the advantages of providing a consistent interface to the middleware services and the network. Uniform and abstracted representations of networks, sensor nodes, sensors, and sensory information are described for SIXTH in Chapter 4. Also defined therein are providers and brokers which define a consistent API for accessing information streams and historical data for any resource.

### 3.4 Extensibility & Scalability

While discussing the necessity for heterogeneity of source in Section 3.2, it was noted that extensibility is desirable in catering for the inclusion of new network connections. This need for extensibility permeates into all levels of middleware. For the sake of illustration consider that a developer providing a new link mechanism should be able to provide extended sensor data representations while maintaining interoperability with clients to which the extension is foreign. Extensibility, defined as “designing for change”, has long been understood as a useful design principle (Parnas, 1979). SIXTH will need to provide specification and connection as interfaces and allow for a developer to specify extensions to the default implementations (see Chapter 5).
It is also desirable to facilitate the dynamic injection of service level functionality into the middleware. Examples of such components include sensor data processing services, network discovery modules, data persistence handlers, and information retention policy handlers. The middleware should contain a rich set of default behaviour so as to not burden the developer with the implementation of middleware level concepts. However, where possible, it should be trivial to override the default behavioural set and instantiate application-specific behaviours.

With regard to scalability it is clear that the middleware should allow for an increase in the scale and information production of sensor networks. Indeed some researchers (Johansson et al., 2009) consider scalability as a subcomponent of and necessity for extensibility. An intelligent management layer may enact policies to deal with heightened throughput. Instances of middleware separated geographically should be interconnected so that applications can leverage the information sources of all instances. Section 4.5.1 details a high-level view of a P2P model for the interconnection of multiple instances of a middleware framework. In Section 5.7.1 a RESTful approach is used for representation and communication between middleware instances. An approach to adding a new service level extensibility is discussed in Section 5.6.3.

3.5 Ease of Use

Ease of use is an often used and somewhat nebulous term; herein this is understood to mean the how easily a user can learn how to operate the middleware and how complete its features are. To put it another way the doctrine of least surprise should be upheld such that features and abstractions which are expected are available. This criterion of this desiderata is intermingled with all others as everything informs and supports ease of use. The middleware abstractions should be understandable at a conceptual level after limited exposure. In Chapter 2 it was understood that the usage of metaphors, such as the abstraction of the sensor networks as a database, supported a users understanding of the system and simplified how they thought about their tasks. Other examples of these metaphors are found in “blackboard” messaging, and feature based grouping.

In the broader context of software development the explicit usage of design patterns is often likened to a “common vocabulary” which is utilized a shorthand for common concepts and techniques for solving problems. In Chapter 2 it was seen that patterns such as publisher-subscriber and observer are often applied to the domain. This is important as the identified user-base will have exposure, implicit or explicit, to design patterns. As such explicitly framing the appropriate design elements using patterns will aid understanding.

Previous attempts at providing a high degree of ease of use have been fraught with difficulties and unforeseen complications. In particular macro programming approaches (see Section 2.5.4) and its prominent specializations (see Section 2.5.5) have been criticised for removing flexibility. In essence these systems can over-abstract the problem domain such that the developer cannot act in a manner not explicitly foreseen in the middleware design.

3.6 Application Support

Within the software development community there exists a long held pearl of wisdom regarding the purpose of a software application - “software is written for the user”. In the middleware domain, the users are application developers. To facilitate the creation of applications built on top of a middleware framework some fundamental considerations must be remembered. The interface should be well-designed such that it is both intuitive and complete. Common tasks should be performed with great ease, and
uncommon usage should also be frictionless. The creation of an API that can be described as 'clean' is a balancing act between completeness and brevity.

It is appropriate to consider the necessity for application hooks to accommodate information dissemination. In software engineering a hook is a component that allows the invoking application to register itself or a particular function to be informed of the event. In the sensor network domain, it becomes apparent that hooks are a prudent mechanism for an application to register for updates of sensor samples, node and sensor status and resource discovery, among others. This kind of functionality can be considered as an applications active link to the information streams. In Section 4.3.2 and Section 4.6 the consideration for providing hooks for connected applications is discussed and the design used in SIXTH is explained.

Within this domain, an essential component of allowing application control of the networks is to facilitate Runtime Reconfiguration which is discussed in Section 3.8. It is also clear that the provision of Abstraction and Uniformity as discussed in Section 3.3 is intermingled with the above-identified need for a clean and consistent interface into all areas of the middleware.

As discussed frequently in the literature (Hadim et al., 2006; Molla et al., 2006; Yu et al., 2004) a desirable middleware platform ought to be capable of supporting multiple applications. Issues arise in the management of each application, perhaps conflicting, QoS demands. Intelligent reasoning technologies integrated into the middleware platform is identified as a key enabler in achieving autonomic management of these shifting needs.

### 3.7 Embedded Intelligence Support

In Section 3.6 the notion of intelligent reasoning mechanisms making decisions regarding QoS parameters was discussed. One paradigm of intelligent reasoning is that of software agents previously discussed in Section 2.5.7. Agent middleware such as Agilla (Fok et al., 2009) enable decision-making without direct user input. This thesis advocates the usage of such a paradigm with all levels of the problem domain. For instance, agents may exist at the node-level performing tasks cognisant of the energy limitations of the WSN components. Agents should also be leveraged as managers on the gateway side; to make decisions regarding global network behaviour.

Support for intelligent agents has much crossover with application support in Section 3.6 as the same hooks are provided for information access and autonomic network reconfiguration should occur in a similar fashion. A service level bridging layer as discussed in Section 3.4 is an appropriate mechanism through which to connect an intelligent reasoning extension into a middleware. The bridging layer handles the management of bi-directional conceptual mapping.

### 3.8 Runtime Reconfiguration

It is of critical importance that the behaviour of the network is modifiable. This modification is a necessity for many aspects of sensor programming from ensuring network connectivity, enabling longevity with adaptive cycling, and altering sampling frequency in line with the application demands. Reconfigurability is a key enabler for both embedded intelligence and application support.

The configuration of networks share similarities, for example, in almost all instances there exists some notion of sampling frequency. However, reconfiguration is also heterogeneous and the mechanism supporting it from an application level needs to encapsulate diverse demands such as the re-specification of an API key in a cyber-sensing context. This is a particular, but important aspect of achieving Abstraction & Uniformity.
It is also foreseeable that some information sources are not reconfigurable. For example read-access may be provided to a sensor observation database. There is a necessity to mark this for the application developer. In such a way, the system may reconfigure the queries being made, or the time-period over which sampling occurs. In Section 4.3.5 and Section 4.4.6 a design for facilitating reconfiguration requests is proposed and implemented.

3.9 Conclusions

This chapter has identified and discussed a set of requirements for sensor middleware. This desiderata was shaped and informed by the research reviewed within Chapter 2. The need to support heterogeneous sources of information was noted; this need is intrinsically tied to the demand to facilitate abstraction from the information sources and in doing so furnish the user with uniform interfaces through which the information sources can be manipulated. In a related concern, appropriate hooks into the middleware must be provided such that integration is easy, and all information can be accessed easily. Furthermore, the importance of allowing reconfiguration of dynamic networks at the behest of applications or through decisions reached by intelligent software agents acting as monitors for large-scale networks is evident.

The overarching requirements, present implicitly in each of the desiderata, are the necessity for flexibility and the mitigation of extraneous complexity. Following from Brooks’ (Brooks Jr., 1956) seminal discussion of the “essential complexity” of software development this thesis espouses that the perfect middleware layer removes all complexity which is not intrinsic to the concerns of the application developer.

This desiderata is the basis underpinning the design of a new middleware platform that is the subject of detailed discussion in Chapter 4. Subsequently, the implementation of this middleware is discussed in Chapter 5. Subsequent chapters, in particular, Chapter 7 and Chapter 8 relate back to the desiderata and implicitly the primary thesis objective given in Section 1.3.1 which encompasses the desiderata.
Part III

Development
4 | Design

“We're building something here. We're building it from scratch. All the pieces matter.”


4.1 Preface

Chapter 3 described a desiderata for sensor-focused middleware; this serves to isolate the functional requirements of the middleware platform. This chapter discusses the design of a new middleware for Sensor Web (SW) applications which aims to deliver upon the desiderata. The platform is termed SIXTH; the name stems from the capability to add sensing modalities to applications. SIXTH, as per the taxonomy described in Chapter 2, is primarily a Gateway Middleware. However, in service of flexibility SIXTH is designed such that the entire stack could be placed on computationally sophisticated sensor nodes. This in-network porting is revisited in detail in Section 7.7.

Gateway Middleware is the best option when supporting heterogeneous hardware, legacy systems, and other software and hardware not under our control. The vast majority of existing WSN middleware fails to support heterogeneous hardware and cannot, fully, support networks not programmed using that system. It has been noted (Mottola et al., 2011) that within industry in-network middleware is rarely used and systems are developed on a per deployment basis; our approach allows connection with such systems. Similarly this approach allows for interoperability with networks using existing middleware technologies.

Each of the components enumerated in the following sections plays a role in delivering one or more of the desiderata: heterogeneity of source, runtime reconfiguration, extensibility and scalability, abstraction and uniformity, application support, and support for embedded intelligence. Table 4.1 summarizes how the design of SIXTH delivers upon the desiderata through a listing of the SIXTH components that are key to the satisfaction of each requirement. The realisation of this desiderata contributes significantly to the achievement of the primary thesis goal (see Section 1.3.1) to advance the state of the art.

The overall goal is to deliver a system that is easy to utilize and feels complete, but not weighed down with the extraneous. The intended end users of the middleware are application programmers working with sensor networks or any data source which can be conceptualized in a sensor metaphor. This identification of the end-user informs the design and implementation strategy. Interaction with external applications is performed through an API rather than a GUI or terminal script. All communication happens through the API with minimal usage of external descriptors such as XML or JSON; this is in stark contrast to some nearest neighbour middleware such as GSN in which XML is a core middleware component in the specification of control flow and sensor observation requirements.

The overarching philosophy which informs the inclusiveness of the design composition and that which is absent, is to explicitly model the domain. Furthermore, these core abstractions should be scaffolded by a
minimal set of additional constructs for information consumption, tasking, or querying. Consequently the middleware core will be minimal, flexible, and open to extension. For instance the core will not include a default persistence layer rather such is deemed to be in the application scope as such information services their objectives.

This middleware will support the creation of connection mechanisms for heterogeneous sensor networks abstracted through a unified API. Such mechanisms ought to be reusable. Once one developer has written a sensor device wrapper, it can be reused by all. Support is provided for extensible object-based querying of all sensor network components. Uniform representations will be given for entries such as sensors, sensor nodes, and networks.

The core of this chapter begins with a description of the guiding design principles (see Section 4.2) utilized in designing the middleware. These principles are themselves grounded in the objectives given in Section 1.3.2 related to the adoption, and espousement of, design patterns and best practice.

Thereafter a discussion of the compositional components (see Section 4.3) of the architecture is given. These compositional components form the basis of other components; core components such as sensors may be viewed as, or possess, several compositional elements. Subsequently, an examination of the core abstractions (see Section 4.4) of SIXTH is provided. The core abstractions represent either an entity abstracted from the problem domain, e.g. a sensor; or they represent a solution to communicating with or reasoning about a sensor network in an abstract and extensible manner. These abstractions form the spine of SIXTH. Following this the description moves up a level of abstraction to discuss the higher level components. Such components deal with the connection of multiple SIXTH instances and extensible service provision. Finally, a description of the in-built services which enable an application to receive information from the middleware is provided.

### 4.2 Guiding Design Principles

The following sections describe the guiding principles which shape the development of SIXTH. The impetus for this is the acknowledgement that software development is a difficult task. Reid is quoted by Frenkel (Frenkel, 1987) as stating:

“Computer Science is the first engineering discipline ever in which the complexity of the objects created is limited by the skill of the creator and not limited by the strength of the raw materials.”

Brooks termed this the “essential complexity” of software development (Brooks Jr., 1956). What follows is a distillation of some mechanisms which mitigate this complexity. These could be considered as the emergent “genes of culture” for software development (Dawkins, 2006). The following sections describe design patterns, generic programming, and the programming philosophy and development methodology utilized. These elements underpin and inform the design of SIXTH which is subsequently described.
4.2.1 Design Patterns

Design Patterns research can be traced to the architectural work of Christopher Alexander (Alexander, 1979), in which he describes "The Timeless Way of Building". Alexander studied solutions repeatedly applied to the same architectural problems and termed these patterns. His work received only limited acceptance in its native field but in the 1990’s this work formed the foundation for a vast amount of influential research in software engineering.

The primary architects of this movement the Gang of Four (GoF) (Vlissides et al., 1995), described design patterns as a common vocabulary for shorthand communication between the programming literate. A design pattern is defined as an often used solution to common software design problems. Patterns are acknowledged as inherently partial solutions, what (Kerievsky, 2005) termed “half-baked”, but malleable; alterations are necessary to fit the solution to the intricacies of specific problems. For example the Adaptor pattern addresses the problem of program communication by masking an incompatible interface with a recognizable one; this is exactly what an electrical socket adaptor does. This common vocabulary can aid in the comprehension of a system architecture.

In this chapter the SIXTH system design is, where appropriate, explicitly framed in the patterns metaphor. For instance the SIXTH network adaptor which encapsulates heterogeneous network interaction is an example of the adaptor pattern. (Jansen et al., 2005) endorse framing an architecture in patterns; the authors refer to their application as encouraging a “pre-defined design decision”, having the benefit of being well accepted and understood. (D. C. Schmidt, 1995) remark that patterns enabled a leveraging of pre-existing design expertise, reduce perceived risk in implementation and promote code reuse. They note that pattern application should be judicious and that while it can aid understandability, implementation time may be increased. Mc Connell (McConnell et al., 2004) remarks that pattern misapplication can compound development problems and increase complexity; which highlights the need for measured application. Table 4.2 summarizes the use of design patterns in SIXTH.

Table 4.2: Design Patterns within SIXTH

<table>
<thead>
<tr>
<th>Design Pattern</th>
<th>Component</th>
<th>Why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptor</td>
<td>Network Adaptor</td>
<td>Mask incompatibility</td>
</tr>
<tr>
<td>Factory method</td>
<td>Network Adaptor, Sensor Node</td>
<td>Encapsulate creation of sensor nodes</td>
</tr>
<tr>
<td>Composite</td>
<td><em>Many</em>¹</td>
<td>Treat multiple resources e.g. sensors as a single unit</td>
</tr>
<tr>
<td>Façade</td>
<td>Multiple Uses</td>
<td>Unified interface to multiple interfaces</td>
</tr>
<tr>
<td>Specification</td>
<td>Query</td>
<td>Encapsulated boolean criteria testing</td>
</tr>
<tr>
<td>Publish/subscribe</td>
<td>Multiple Uses</td>
<td>Decouple receivers and publishers</td>
</tr>
<tr>
<td>Mediator</td>
<td>Tasking &amp; Brokers</td>
<td>Decouple interacting classes via an intermediary</td>
</tr>
<tr>
<td>Proxy</td>
<td>P2P communication</td>
<td>Make the remote look local</td>
</tr>
</tbody>
</table>

Anti-Patterns  In consciously choosing to view the software design through the lens of patterns it is prudent to consider how to avoid anti-patterns. (W. J. Brown et al., 1998) defined an anti-pattern as “a commonly occurring solution to a problem that generates decidedly negative consequences.” Common examples of this phenomenon include; optimization at the expense of readability, over abstraction, prohibitively large interfaces, loss of cohesion (big ball of mud), and object over-responsibility (god object). Adherence to the software design maxims of Section 4.2.3 acts as an implicit defence against anti-patterns.

4.2.2 Generic Programming

(Musser et al., 1989) defined generic programming as the specification of algorithms for later specified data types. (Stroustrup, 2007) expressed the notion elegantly as “Lifting algorithms and data structures

¹Adaptor, Sensor Node, Deployment, Service-layer
from concrete examples to their most general and abstract form”. A notable example is the C++ Standard Template Library (STL) ([Stepanov et al., 1995](#)). Therein, an iterator operates independently of the data structure. Listing 4.1 depicts C++ STL code, wherein an `int` data type is stored in a generic vector container.

Listing 4.1: Generics in the C++ Standard Template Library (STL)

```cpp
vector<int>::iterator current = values.begin();
int high = *current++;

while (current != values.end()) {
    if (*current > high)
        high = *current;
    current++;
}
```

Generic programming is applicable in SIXTH in areas where a concept is repeatedly utilized. In Section 4.4.5 the design of a query abstraction is defined based on functionality used in related work from Chapter 2. A generic query is defined for its subject which can be any resource e.g. sensor, node, or network. In Java syntax a generic query parametrized for a sensor is given as `Query<Sensor>`.

The GoF warned that “highly parameterized software is harder to understand”, and as such a balance must be struck in its application. It is worth noting that when this statement was made, generics had not been so widely applied in mainstream languages. Comprehension has likely increased in the intervening years as generics form a core part of undergraduate data structure courses and have found themselves integrated into mainstream languages. The judicious application of generics is discussed in later sections and in Chapter 5.

### 4.2.3 Programming Philosophy

Raymond ([E. S. Raymond, 2003](#)) distils the philosophy of software development into seventeen rules which are reflected in the literature (Foote et al., 1997; McConnell et al., 2004; Pressman, 2001; Saltzer et al., 2009). The following sections are an enumeration and discussion of the axioms which most inform the design of SIXTH.

- **Clarity & Transparency**: It is better to be clear than it is to be clever. Software should be developed carefully and designed as if the most important communication is with a human reader, such as the maintainer, and not with the machine. This rule is an accompaniment to the rules of least surprise and separation of concerns. Though unintuitive at first too much cleverness in program code can result in succinct implementations which obscure intent, disrupt extension, and reduce maintainability.

  Transparency refers to designing such that your development process and intentions are clear to future users and maintainers. Raymond writes that “Robustness is the child of transparency and simplicity.”. In creating programs which are simple, or composed of smaller simpler programs, transparency and testing becomes easier.

- **Composition**: This can also be considered as decomposition. A project should be broken down into small and uncomplicated classes. These programs can then interact and be composed in a manner which lends itself to comprehension, reuse, and single-responsibility.

- **Separation & Modularity**: This is the separation of what is to be done from how it is done; “mechanism from policy”. In object-oriented programming an interface is a specification of what
functionality is provided. When “programming to the interface”, the developer need not concern themselves with how something is done.

Modularity refers to breaking apart system functionality into smaller subcomponents. Consider the re-implementation of a monolithic C program into an Object-Oriented set of cooperating classes; the latter is generally considered more readable, maintainable, and understandable. Pressman (Pressman, 2001, p.100) identifies modularity as enabling separation of concerns in software development. In similar fashion Martin (Martin, 2003) defined the “Single Responsibility Principle” which acts as a guideline for modularity. The principle states each entity should have only one responsibility to increase overall system robustness.

- **Simplicity:** Keep things simple where possible; as Beck put it “do the simplest thing that could possibly work” (Beck, 2000). Complexity cannot be ignored, this occurs in all non-trivial development, however every effort should be made to mitigate and limit complexity. Separation and modularity are two such methodologies which mitigate complexity.

- **Least Astonishment:** (Saltzer et al., 2009) stated that “people are part of the system. The design should match the users experience, expectations, and mental models.” The design of the software should make sense to a developer. From an OO perspective this means the objects, their interactions, and composition should be understandable, appropriate, and uncomplicated.

- **Repair:** This is akin to the principle of autonomic management (Kephart et al., 2003); the system should attempt to repair any faults and be capable of self-repair. However it is stressed that if recovery is impossible, the system should fail loudly and early.

- **Economy:** Conserve the time of the programmer by making the code do more; The argument being that machine time is now inexpensive and developer time is precious. This can be achieved through sufficient abstraction, complete intuitive interfaces, simplicity of interaction, and clarity of design.

- **Optimization:** Knuth stated “premature optimization is the root of all evil”. In development this is understood to mean that working software is better than non-existent software. Optimization should be performed when it is known to be necessary to do so beforehand detracts from design.

- **Diversity:** Raymond (E. S. Raymond, 2003) likens this axiom to the “wisdom of open software” which allows for customization. Such customization may relate to user specific requirements for optimization, presentation, or storage. The rule is an appeal to humbleness, stating that no individual can optimize for all eventualities.

- **Extensibility:** Programs should be extensible such that new functionality can be added to a developed architecture. Babar (Babar et al., 2013) notes the importance of “flexibility points” in large software projects. Flexibility points define what is open for extension acknowledging that it is difficult to know how a component will be extended. Pressman (Pressman, 2001) acknowledged this stating “be open to the future” and design software to solve general problems which can be applied to specific instances. Extensibility was explicated identified as a functional requirement in Chapter 3.

The rules are further reduced to the one iron law, “Keep it Simple, Stupid” (KISS) (Pressman, 2001; E. S. Raymond, 2003). The above discussed design principles relate strongly to the functional desiderata from Chapter 3 as such there is a strong correlation between what is to be acheived and the manner in which development is structured in service of those objectives.

### 4.2.4 Development Methodology

When developing any non-trivial software project it is prudent to utilize a software development methodology to structure the work. A software development methodology partitions development into
distinct, possibly repeating, phases which solve different elements of the overall problem. (Boehm, 1988) states that a methodology is primarily concerned with ordering the development stages and establishing the necessary transition requirements. This can be extended to incorporate rolling back over several stages, for example to facilitate re-factoring.

The development methodology used herein is Agile (Beck et al., 2001), particularly with respect to “responding to change”. The methodology is more explicitly described as a combination of iterative and incremental development as described by Cockburn (Cockburn, 2008). Larman (Larman et al., 2003) supports such a methodology and notes the long standing, and fruitful, concurrent usage of these agile cornerstones in development processes distinct from software engineering.

Iterative development refers to the allocation of time to the evaluation and subsequent re-factoring of the software. The goal being to improve the overall work. The concept is deftly illustrated in Figure 4.1 wherein the approach is applied to the Mona Lisa.

![Figure 4.1: Iterative Development of the Mona Lisa (Cockburn, 2008)](image)

Incremental development is the creation of the overall system through the successful development of working sub-programs. Therein each development phase extends the system with new functionality. (Cockburn, 2008) illustrate this as a three dimensional cube, divided into bricks, in each phase more bricks solidify and are integrated. Iterative & incremental are useful in consort as iterative approaches explicitly define reflection on progress and improvements. It was felt that applying a traditional waterfall development model (Royce, 1970) to such a large project would prove unsuccessful, as reflection, re-implementation, and re-factoring were foreseen as inevitabilities.

Section 4.3 describes the compositional elements of SIXTH. These entities form the basic building blocks to compose richer objects such as the core abstractions. It is the core abstractions and the manner of their composition which is the primary provisioner of the desiderata, which specifies our functional requirements, enumerated in Chapter 3.

### 4.3 Compositional Elements

The following sections discuss the compositional elements of SIXTH. Many of the core abstractions have either a has-a or an is-a relationship with one or more compositional elements. For example a sensor is-a provider of sensor observations, and a sensor has-a descriptor. Compositional elements are key to delivering the Abstraction & Uniformity desideratum described in Section 3.3. It should be formally stated that the intention of all components is to support the overarching, and sometimes oppositional, goals of Flexibility (see Section 3.9) and Ease of Use (see Section 3.5).

#### 4.3.1 Provider

A provider is some resource that provides access to a set of other connected resources. This concept exists within the problem domain; for instance, a sensor node provides access to a set of sensors. In turn, a set
of sensor data is provided by the sensor. Within the SIXTH abstraction, an adaptor provides access to a network of sensor nodes. The provider is capable of supplying a restricted result set when invoked with a query (see Section 4.4.5). To the authors knowledge SIXTH is the first sensor middleware to explicitly adopt this domain concept as a design element.

4.3.2 Broker

A broker is a delivery mechanism for some resource such as sensor data, sensors, or nodes. A broker is the subscription mechanism for a subscriber to define its interests to the publisher. When resources are to be disseminated a broker acts as an intermediary for the publisher and its recipients. A broker is often used in the publisher/subscriber Design Pattern to further decouple sender and receiver and allow for extension and scalability. Figure 4.2 shows a generic broker element interfacing between three producers and two consumers. Consumer 1 is interested in what is produced by producer 3; consumer 1 registers interest with the broker and is subsequently streamed information which matches its interests. This brokering model is broadly similar to the broker design pattern utilized in distributed systems design.

![Figure 4.2: Brokers in SIXTH](image)

Compositional elements such as Providers and Broker and other SIXTH components such as Queries provide concept level extensibility in the SIXTH architecture. These components can be defined in terms of any object including those not explicitly in the original design. This is achievable through the use of generic programming constructs.

4.3.3 Descriptor

A descriptor is an immutable specification of information and meta-data regarding some resource i.e. a sensor or a sensor node. The descriptor is not a taskable component, as discussed in Section 4.3.5, as such it can be provided in lieu of a taskable resource in a security consensus approach.

4.3.4 Aggregate

An aggregate is a collection of related entities that are accessed and controlled as one unit. This was identified as a common software design pattern in (Vlissides et al., 1995). An aggregate of sensor nodes is akin to the grouping notion of Kairos (Gummadi et al., 2005), HOOD (Whitehouse, Sharp, et al., 2004) and other WSN Middleware offerings. In the literature this concept is referred to as Logical Regioning; its application in pre-existing middleware solutions is discussed in Section 2.5.9. These groupings are generated using a query and can represent a logical group, sharing some characteristic or a neighbourhood group, existing in the same geographical region.
4.3.5 Taskable

A taskable entity is one which can alter its behaviour based on some input requests specified in a generic format. Reconfiguration requests issued to taskable components are given in the form of tasking messages as defined in Section 4.4.6. Many SIXTH entities are taskable, the following is a listing of the most utilized taskable components: (i) Sensors (ii) Sensor Nodes (iii) Sensor Networks (iv) Network Adaptors (v) Tasking Service. Taskable entities are crucial in supporting the desideratum of Runtime Reconfiguration (see Section 3.8) and Embedded Intelligence autonomous network control (see Section 3.7).

The forthcoming section describes the core abstractions of SIXTH. As discussed in Section 4.1, the above discussed compositional elements form the building blocks of these core domain abstractions. Often this is manifest through an is-a or a has-a relationship, for example a sensor node is-a taskable component and has-a descriptor.

4.4 Core Abstractions

Figure 4.3 illustrates the high-level core architecture of SIXTH. By their nature, the compositional elements (see Section 4.3) are mostly hidden. The data broker which is an important broker instance is shown and is described in Section 4.5.3.

Examining in Figure 4.3 from the bottom up the lowest level elements are information networks. The examples given are an Android smart-phone network, a Sun SPOT WSN, and Twitter. The access to, and control of, these networks is performed via sensing network adaptors described in Section 4.4.7.

Sensing network adaptors create and maintain network resource representation such as nodes and sensors (see Section 4.4.2 and Section 4.4.3). Sensor nodes are encapsulated within sensor network representations, an adaptor typically has a single network but may represent any amount. Figure 4.3 illustrates that sensor observations are pushed from the sensing network adaptors to the appropriate broker mechanism which is responsible for dissemination to client applications. The dissemination process, described in Section 4.5.3, involves client registration using one or more queries, see Section 4.4.5.

Sensing network adaptors register themselves with the discovery service; a collection of adaptors is linked to a SIXTH deployment. A SIXTH deployment is a representation of all components connected to a single host machine running the SIXTH middleware stack. A connection to a remote instance of SIXTH is represented as another SIXTH deployment. The discovery service maintains references to all deployments and offers a set of services that operate across all known deployments. The discovery service is further discussed in Section 4.5.3. Sensing network adaptors communicate status information with a set of application hooks; these hooks are enumerated in Section 4.6.

It is the contention of the author that SIXTH offers a very explicit abstraction of the sensor networks problem domain which is not as well specified in other similar middleware offerings such as GSN (Aberer et al., 2006b), SStreamWare (Gurgen et al., 2008), or WSN-Ware (Viani et al., 2013). The following sections will discuss the representational entities used in SIXTH. These include sensors, nodes, and networks as well as important communication abstractions such as tasking messages.
4.4.1 Sensor Network

Chapter 2 described a sensor network as encompassing many spatially distributed, but connected, sensor nodes which communicate to service application objectives. Figure 4.4 describes the hierarchical encapsulation of the design. Therein an adaptor, as seen in Section 4.4.7, can encompass many networks; in turn a network has many nodes and so on. The adaptor is an active entity which controls the population and regulation of sensor networks and their resources.

4.4.2 Sensor Node

As discussed in Chapter 2 a sensor node is a single hardware device made up of multiple sensors capable of sampling diverse phenomena. The SIXTH sensor node model represents such a platform wherein the single node encapsulates multiple sensors in program code. An example of a commonly used sensor node is the Sun SPOT shown in Figure 4.5. Its most recent iteration features three-axis acceleration,
temperature, and light sensors. Sun SPOTs also feature eight tri-colour LED outputs often utilized for low-level debugging of sensor applications.

Figure 4.5: Sun SPOT Sensor Platform

Figure 4.7 depicts the software abstraction of a sensor node. This generalisation can represent either a single physical sensing device such as the Sun SPOT or provide an interface layer for a cyber sensor node. An example of such a cyber sensor node is a configurable current affairs monitor for the social media content aggregation platform Reddit (Lerman, 2006). The domain specific code for accessing the Reddit API is wrapped in the sensor node interface. This generalization is depicted in Figure 4.6.

Figure 4.6: Sensor Node Model

Figure 4.7: Sensor Node Abstraction

From Figure 4.7 it can be seen that the sensor node contains a listing of its sensory capabilities encapsulated as virtual sensors (see Section 4.4.3) a listing of one-hop neighbours, and a mapping of its hardware properties. Additionally the sensor node provides access to an immutable descriptor, defined in Section 4.3.3 to be distributed to interested receivers for informational purposes.

A sensor node is an example of a taskable resource (see Section 4.3.5). In Figure 4.7 a reconfiguration request is issued to the sensor node. These requests may refer to the operation of one or more of the sensors existing within the node or the node as a whole. Received tasking messages are parsed, and if the request is appropriate and achievable, steps will be taken to achieve that goal within the underlying network or cyber-platform. A boolean return value indicates whether the node will execute the request. A sensor node is an example of the provider model (see Section 4.3.1), provisioning access to its set of
sensors. The node can be queried for its full sensor complement, or that set can be filtered through the use of a query mechanism (see Section 4.4.5). The design draws on the benefits of the Object-Oriented paradigm particularly encapsulation and extensibility. In defining a general interface to the sensor node abstraction it is possible to limit constraints on the possible implementation internals; this topic is revisited in Chapter 5. This high-level abstraction has been leveraged in allowing the sensor node construct to be representative of both physical sensing platforms and Cyber-Sensing code bundles connected to web-based resources. The explicit use of cyber-sensors is discussed in Section 7.3.5.

In Figure 4.4 the usage of hierarchical encapsulation in the design of SIXTH is highlighted. In this high level view it is apparent that an adaptor contains sensor networks, networks encapsulate nodes, sensor nodes encapsulate sensors, and a sensor contains a collection of observations. The concerns of representation and management are split between these components in keeping with the division of responsibility that is paramount to encapsulation and core to object-orientation in the avoidance of God Objects (Riel, 1996).

In early prototype implementations of SIXTH there was no explicit notion of a sensor node; this was implicitly represented as a set of sensors. This design was flawed as it did not account for properties pertaining to the entire node and made tasking more conceptually difficult. The hierarchical encapsulation utilized in the final design is more appropriate to the application domain that the initial prototype design.

In the subsequent section software sensors, which are nested within the sensor node, are discussed.

### 4.4.3 Software Sensor

In Section 2.3.1 a sensor was defined as some entity capable of the observation of phenomena and the subsequent conversion of that measurement into human readable data. In Section 4.4.2 a sensor node was identified as containing one or more sensors. Each sensor is capable of sensing a different phenomena e.g. temperature or humidity. The SIXTH notion of a virtual sensor is distinct from the view offered in GSN (Aberer et al., 2006b); therein a virtual sensor often referred to observations from multiple sources. In SIXTH the mapping is one-to-one and an aggregate sensor is used to model multiple sources in a composite pattern.

In the relaxed view advocated by this thesis the range of these observations is extended into information which may be retrieved by traditional computer programs. Examples include a RSS feed of new articles from Reddit, an email, or a user’s last 20 tweets from their Twitter. The term cyber-sensor is used to refer to some mechanism that retrieves information from web-based resources. The design is intended to allow for a cyber-sensor to be utilized indistinguishably from a physical sensor wrapper.

Figure 4.8 showcases a high-level perspective of the sensor abstraction. It can be seen that the sensor maintains a store of its produced observations (see Section 4.4.4). For this reason the sensor can be modelled as a provider (see Section 4.3.1) for sensed data.

Section 4.5.1 discusses the data retention policy which acts upon the sensor to determine which sensed data samples to maintain; this selection is done through a user determinable criteria. Additionally the sensor provides its descriptor, defined in Section 4.3.3, implements a device monitor, and contains a mapping of its hardware properties. A device monitor may refer to the monitoring of the physical counterpart of a virtual sensor or to the monitoring of the state of the associated cyber environment. The device monitor is responsible for the detection and reporting of an apparent sensor time-out. Time-out occurs when a new reading has not been received within a specified threshold of the sensors known sampling rate.
The hardware properties stored in the sensor are those which are unique to the sensor and distinct from those found in the sensor node. These include pertinent information such as the sensors maximum sampling rate, for instance, the maximal sampling rate for the accelerometer within the Sun SPOT platform is 160hz.

Figure 4.8 shows the sensor receiving a reconfiguration request; the intention of which is to modify the sampling frequency. The sensor is a taskable (see Section 4.3.5) component which interprets tasking messages (see Section 4.4.6).

Sensors are grouped together to form an aggregate as defined in Section 4.3.4. The query mechanism discussed in Section 4.4.5 is leveraged as an inclusion criteria for the aggregate. This abstraction is utilized to reconfigure a group of sensors concurrently.

The descriptor (see Section 4.3.3) obtained through the sensor is designed to encompass all pertinent information about the device as defined within the sensor itself.

The following section provides a discussion of the particulars of the sensor data abstraction format utilized by SIXTH and stored within a virtual sensor.

4.4.4 Sensor Data

Sensor data typically refers to the result of sampling performed by a sensor. For example, the observation that the temperature in the kitchen is 35 °C is a piece of sensor data. Other kinds of information are produced by a WSN in typical operation such as network topology information or battery level indications. This abstraction is not used to convey this sort of information; other representational entities are utilized to convey such meta-data; an example would be hardware properties held in sensors and nodes.

Sensor data is important for any application dependent on a sensor network. These observations and the insights which can be gleaned from them are the crux of the matter. To make a decision it is often necessary to utilize multiple pieces of sensor data from diverse networks.

From this, a determination is made that it is pertinent to lift sensor data from each different source into a single unifying data format, as such this is the approach taken within SIXTH. The mechanisms used by network adaptors to lift sensor data from heterogeneous formats are discussed in Section 4.4.7.

In the following sections, the data fields of the abstraction format are enumerated, and the rationale for their inclusion is given.

- **Time-stamp**: The time at which the observations were made by the sensor. This field can be utilized to order the samples as messages can arrive out of order. Where time is consistent the time-stamp can be used to compare samples from different sensors; where the comparison is across the same modality this can enable determination of a faulty sensor or facilitate event detection.
The grouping of temporally indistinct samples enables the user to see a fuller picture of the sensed environment.

The field is specified as a long data type as per (Schildt, 1990) holding a value $-2147483647 \leq x \leq 2147483647$. The value $x$ is intended to be in unix-time; the number of seconds since 00:00:00 on the 1st, January 1970.

- **Network**: This field indicates the sensor network to which the observation belongs. The value of this field maps to the network adaptor discussed in [Section 4.4.7](#) which encapsulates the connection to that underlying network. No restriction is placed on the format of the string network identifier.

- **Node Identifier**: This field represents the identifier of the sensing platform which produced this observation. SIXTH only requires that node ID be unique within the sensor network. The format of the node identifier is the same as described for the time-stamp.

- **Sample Type**: This field signifies the circumstance under which this sample was obtained. In SIXTH two types of sample are explicitly defined. A periodic sample is one obtained in accordance with the sensors sampling rate. A requested sample is given in response to a one time request. This model is extensible to allow the definition of custom sample types. The sample type is to be held as a free form string, of which there will be common known values. The direction here is to provide short human readable strings.

- **Modality**: This field corresponds to the phenomenon which is being sensed; this directly maps to a sensor. A sensor is uniquely identified by the triumvirate of network, node ID, and modality. The network and node identifiers are sufficient to uniquely identify a sensor node. The modality is a more complex object than the components described previously. As a modality may have many sub-parts this relationship is represented as a map of key-value pairs.

- **Data values**: This field holds the payload of the sensor data which are the observations of the sensor. This may be a single value as with a temperature observation in Celsius. The payload may also be a set of connected readings as with acceleration data where we have a connected set of x, y, and z axis observations. In the preceding example it is evident that the acceleration values should be stored together; for this reason this flexibility has been factored into the design. The values are stored in a Map data structure (Lafore et al., 2003). This enables the storage of any data in string format alongside its unique key.

This is flexible, however for further ease of use, domain wrappers, such as for acceleration values, could be created which extend the base abstraction by providing specific methods for their known data points e.g. `$getY()$`. This has been done in WAIST which is described in [Section 7.5.1](#). Such domain wrappers are explicitly provided in the core of WSNWare and GSN; however in SIXTH the decision was made to omit such from the middleware core in service of flexibility.

### 4.4.5 Query

Information consumers have diverse interests with regard to sensor data; one application might be interested in all aspects of the network while another may be only in monitoring the temperature in some subset of the network. This is heavily linked to the previously identified goal of providing Application Support. Interest may be conditional upon a significant shift in the observations; for instance a fire alarm application may only wish to know when the temperature has increased by 20% over the average observation. The key is to alleviate the consumer from being flooded by minor fluctuations.

The query concept is most explicit in database metaphor middleware such as TinyDB (Madden, Hellerstein, et al., 2002) which were discussed in detail in [Section 2.5.5](#) this has proven as a successful and influential paradigm for in-network middleware and it deemed prudent to lift the concept to an
object-oriented level for the SIXTH gateway middleware. GSN (Aberer et al., 2006b) lacks an object-oriented query function as all queries are expressed on database tables automatically generated for received samples. WSN-Ware (Viani et al., 2013) is the most similar in its approach defining a Validator which denotes rejection or acceptance of messages; this is tied explicitly to sensor observations and sensor nodes and there is no method to easily compose queries.

For the purposes of filtering the real-time sensor data stream and accessing historical data a query abstraction is defined. This query mechanism is extended so that a query may be defined for other resources such as sensors or sensor nodes. This encapsulation of boolean criteria satisfaction tests has been formalised as the Specification pattern (E. Evans, 2004). An application registers interest with a broker (see Section 4.3.2) using a query, the broker then streams matching information to the consumer as it becomes available. In similar fashion when accessing a resource store through a provider (see Section 4.3.1), a query can be supplied. Figure 4.9 illustrates the operation of a sensor data query whose criteria is modality. All sensor data is rejected except where the modality is temperature. For Extensibility a query is a concept-level construct which can be defined for unforeseen components not explicit in the design.

Figure 4.9: SIXTH Query

This abstraction defines a purposefully simple interface which notifies of acceptance or rejection. A query may define any criteria of acceptability. More complex queries may wish to filter across multiple criterion such as time-stamp and modality. These simple criteria are combined through aggregate and conjugate queries, and combinations of both. An aggregate query matches a resource only if all the sub-queries match. A conjugate query matches if any of the sub-queries match. As becomes evident in Section 5.3.2 a query is any object which implements the interface for acceptance or rejection. Additionally Paragraph A.2 describes how a query can be expressed in SQL syntax through the SIXTH Query language.

4.4.6 Tasking Messages

In Section 3.8 runtime reconfiguration was identified as a key cornerstone in provisioning a flexible middleware platform for sensor networks.

It is for this reason that the tasking message abstraction is defined to encapsulate reconfiguration requests which are sent from the application layer to the adaptor modules, or its sub-components. This abstraction allows for the formulation of typical sensor network reconfiguration requests such as sampling rate modification while being flexible to allow for more complicated or network specific requests. In Section 4.3.5 taskable resources were defined and it has been shown that sensors, nodes, networks, and adaptors are taskable. All taskable resources accept tasking messages; when received these requests are either executed or rejected.

Tasking messages were absent in earlier mock-ups of SIXTH. Tasking was performed through explicit methods such as setFrequency(long). It was quickly apparent that such a system was not scalable or adaptable to the needs of diverse sensing networks. A controllable feature of one network such as sampling frequency may be absent or beyond middleware control in another. As previously discussed in Chapter 2
Horre (Horre et al., 2007) suggests that middleware should provide support for manual, semi-automated, and automated management of sensor networks. This thesis is in agreement, as such be the control through user intervention or autonomic management software the required actions can be conceptualized as tasking messages. Section 7.4.2 describes the creation of an automatic management interface for SIXTH.

In the following section a breakdown of each part of a tasking message is detailed as well as the rational for the inclusion of each. Firstly it is prudent to understand the command type which is included in each tasking message as a definition of its basic behaviour. Command type is an enumerated type which is a data type consisting of several named values which are essentially symbolic.

**Command Type**

- **Set frequency**: This denotes that the tasking message is intended to set the sampling frequency for a resource. This command type is also used to turn a sensor on or off, where off is defined as a frequency of 0.
- **Request value**: This denotes a desire for a single observation of a specified modality.
- **Request information**: This is issued to ask the resource to provide specification information. For a sensor node this may refer to sampling rates, battery level, capabilities or its neighbour list.
- **Task**: This command type is specified for more complicated reconfiguration requests which may be network specific. A heterogeneous helper object known to the message target is utilised.
- **Create**: Create some resource.
- **Inform**: Relay pertinent information.
- **Update**: update some resource i.e. update the user name and password used by a cyber sensor.
- **Assign**: Assigns some component to a resource, i.e. assign a Message Translator to an adaptor.

The command type is akin to the message headers prescribed in MidSN (Cecilio, J. Costa, et al., 2013; Cecilio and Furtado, 2012). Therein the authors included: (i) command (ii) acknowledgement (iii) task completion (iv) sensory observations. Those identified for SIXTH differ significantly; it would seem that create, inform, update, and assign would fall under the generic command header. It was felt prudent to provide our further decomposition to aid clarity and to acknowledge tasking of adaptors, services, and SIXTH itself through the same construct. Acknowledgement is included explicitly as a return value from network communication; this then actuates SIXTH to update the Virtual Sensor state. The SIXTH approaches also differs from MidSN in that SIXTH does not envisage always controlling the network, consequently the design does not necessitate the use of our command types within the networks. Obviously this means that message translation is not as clean, however the view advocated is that heterogeneity must be embraced. For a richer, cleaner, integration, Command Type usage in-network would be advantageous.

- **Tasking Object**: This is the object used to express the behavioural changes specified by the tasking message. When modifying a sampling frequency, or enabling a sensor, a frequency object is specified as the tasking object. When a more specific request is to be made a network specialised object is used for tasking.
- **Sensor**: This denotes the specific sensor which is targeted for reconfiguration. The omission of this parameter denotes that the request cover all sensors which exist on the platform specified by the Node ID. This is most appropriate in specifying a common sampling frequency for all of a nodes sensors in a single request.
- **Node Identifier**: This is the ID of a targeted node; this too can be omitted when a tasking message is intended to effect all the network nodes.
• **Adaptor Identifier:** This is a descriptor [Section 4.3.3] which fully captures all the necessary addressing information for the target adaptor.

WSNWare ([Viani et al., 2013](#)) features a similar abstraction in allowing *Reconfigurable* elements to be modified with a dictionary of properties; this lacks the extensibility of the SIXTH approach which allows any object to be used for unbounded extension, thereby avoiding the need for complicated multi-part dictionary keys. In WSNWare, no addressing information is used to allow a service to route requests. As such the request must be issued directly.

The following section provides a discussion of the SIXTH abstraction for network management, the adaptor. Within this description there is a discussion of the manner in which tasking message are dealt with as regards their translation and in-network execution.

### 4.4.7 Network Adaptor

In [Chapter 3](#), an identification of middleware requirements was given; two key parameters identified therein were *heterogeneity of source* and *abstraction and uniformity*. The network adaptor layer is the component which bears the most responsibility for delivering upon these criteria. Each network adaptor is responsible for encapsulating the connection between SIXTH and an information source such as one or more WSNs or a web service API. Each adaptor provides uniform access to its resources and multiple adaptors can deliver upon the necessity for heterogeneous data sources. In keeping with *extensibility*, it is also of importance that the middleware be capable of easily adding to its repertoire. To achieve a suitable *heterogeneity of source* a middleware platform must be capable of connecting to diverse information sources such as sensor networks, linked data sources, databases, text files, and web-based information. By definition an Adaptor is a “hot spot” within the framework. As described in [Section 2.4](#) this describes a component which is overridden by developers utilizing the framework to provide custom behaviour.

As discussed in [Section 3.8](#), runtime reconfiguration is an important consideration for sensing resources. This is central for the middleware as a whole in that the addition, removal, and update of components such as adaptors must be dynamic. The implementation of this mechanism through enabling technologies is discussed in [Chapter 5](#).

Figure 4.10a illustrates a high-level overview of the role played by network adaptors in the middleware architecture. It can be seen that each sensor network connects through the specified adaptor; the method of connection is network specific. The information received from a network is translated by the adaptor and sent to the application layer via the brokers [Section 4.3.2](#). Applications pass their requests, in the form of tasking messages [Section 4.4.6](#), through the tasking service [Section 4.5.3](#) to an adaptor which propagates the message to the affected network resources. The network adaptor utilizes a sensor node provider (see [Section 4.3.1](#)) to represent the network of nodes to which it is connected. Figure 4.10b depicts the logical decomposition of an adaptor into three closely related co-operating functional units: wrappers, translators, and connectors. This decomposition provides a clearer separation of concerns in keeping with core Object-Oriented design principles. In practice, it can be difficult to untangle these concerns but where possible these components should be disconnected. As such redevelopment and modification is compartmentalized. For example, consider if it is necessary to change the connection methodology then it is immaterial to the message translation components. This decomposition was identified after initial implementation of prototype adaptors.

The following sections discuss the logical components which make up an adaptor.

**Wrapper**

A network adaptor is taskable [Section 4.3.5](#) component, and the translation of tasking messages [Section 4.4.6](#) into a network specific data format which is understood by the underlying network is
the purview of the wrapper. In a cyber-sensing scenario a wrapper is performing an interpretation of the tasking message and consequently executes some other code within a cyber sensor node.

**Connector**

The connector establishes and maintains communication with the network through using some connection medium to which the network is attached. This may be across HTTP, over Bluetooth, or via a serial connection to a Zigbee USB dongle. When a wrapper has transformed a tasking message the result is passed to the network via the connector. Similarly, the connector forwards all data received from the network to the translator.

**Translator**

The translator module is responsible for the transformation of messages received from the network into a SIXTH compatible data format. Most often, the received messages correspond directly to the observations of sensors, and as such these are converted into sensor data (Section 4.4.4) objects. Messages may also correspond to alerts regarding sensor or node state, causing an update to be performed on their virtual representations, and this information will be propagated to interested clients. If the translator is unable to parse the message, an error is raised, and the adaptor may respond by cycling through a set of translators. Where there exists some heterogeneity within a single network, e.g. where an incompatibly programmed Sun SPOT is added to a homogeneous deployment, multiple wrappers and translators facilitate communication using several data formats. In combination Wrappers and translators are analogous to the MessageSource component utilized in WSNWare for inserting and collecting messages from the network. However, it is felt that the decomposition in the SIXTH design better aids development following a separation of concerns.
The following sections provide a discussion of the operational particulars of physical sensor and cyber-based network adaptors. These are the two broad classifications of network adaptor instances.

**Sensor Network Adaptors** Typically a WSN adaptor interfaces with the whole WSN via a gateway-node. Messages from other network resources are passed via the gateway and are transformed by the translator into sensor data instances. For each different sensor platform custom code must be written to accommodate tasking and message passing in line with SIXTH standards. Network adaptors have been created for platforms including Sun SPOT, TelosB, Mica2, Arduino, Shimmer, WaspMote, and Tyndall iWSN.

**Cyber-Adaptors** For each web-based environment to be monitored, a cyber sensor adaptor is defined; this component may produce multiple cyber sensors. Cyber sensors can be defined to monitor a different sub-component of the platform; for instance a sensor may be created to monitor a particular user. Cyber-sensors reside on the desktop machine running the middleware framework. As such all sensing and processing is performed within each sensor nodes program code.

A cyber-adaptor provides the same reconfiguration and sensor access mechanism as their physical sensing counterparts. Utilizing these features cyber-sensors can be dynamically created, destroyed, or reconfigured during runtime as befits the shifting demands of the consuming applications. As an example we consider a reconfiguration request to restrict a Twitter sensor to a latitude-longitude bounding box.

The following section will discuss the higher level components of the SIXTH middleware; many of these operate by utilizing the core abstractions.

### 4.5 High-Level Components

Previously in Section 4.4, the core representational entities of the SIXTH design were discussed. Herein a set of higher level components are enumerated. These higher level components are closely related to the core abstractions. For example a SIXTH deployment encompasses many sensor network adaptors and the brokering mechanisms of the discovery service deliver information regarding and from sensors and sensor nodes.

#### 4.5.1 SIXTH Deployment

A SIXTH deployment represents a logical grouping of all resources which are connected with or operating directly under a single host machine. This is in keeping with the goals of scalability, extensibility, and abstraction as multiple SIXTH instances are connected in a scalable peer-to-peer manner. One SIXTH instance sees an abstracted representation of others. The mechanisms envisaged for connecting running instances are extendible and replaceable. Figure 4.11, borrowing some informal UML syntax, establishes that a SIXTH deployment comprises zero-to-many network adaptors (Section 4.4.7), service factories, and modular runtime services (Section 4.5.2).
The encapsulation of all middleware resources allows for a simple conceptualization in dealing with peer-to-peer (P2P) connections. When the middleware communicates with another instance of SIXTH, a virtual representation of that deployment is maintained. In this way a user can act on resources such as a sensor attached to a remote network in the same manner as with a local sensor. This design model is shown in Figure 4.12 therein it can be seen that each SIXTH deployment mirrors the networks which are attached to the other. As the usage of SIXTH expanded to the P2P paradigm, the deployment model was developed as an appropriate object-oriented representation of remote resources. This approach meant minimal changes to the existing design and promotes a seamless interaction with remote resources.

This representation of the remote resources through a shadowing interface layer is an example of the GoF Proxy Pattern (Vlissides et al., 1995). The goal of the pattern is to reduce complexity. To facilitate high system flexibility for implementation a pluggable addressing model approach is utilized for the identification of remote SIXTH deployments. This model can support all data sources from simple static listings to a shared web registration portal. A deployment utilizes a set of providers (Section 4.3.1) which are utilized to provide access to resources such as network adaptors.
Retention Policy Manager

From Figure 4.11 it can be seen that each SIXTH deployment has its own retention policy manager. The manager is responsible for the management of sensor data retention (Section 4.4.4) across all subcomponents of a deployment which hold caches of sensor data.

In some sensing application scenarios a large quantity of sensor samples are generated quickly and the retention of this data in memory is neither feasible or desirable. The manager removes sensor data from memory based on specific criteria. It is important to maintain flexibility in this regard as applications have disparate requirements for what is to be retained.

The criteria for retention is specified as a query (see Section 4.4.5). The manager traverses all available sensor nodes and performs the filtering on all the stored sensor data. Only data which meets the conditions is retained. A default implementation treats all sensing modalities equally and works on the basis of freshness. Such treatment of data is advantageous in many scenarios but is particularly applicable when the middleware is operating on a constrained device.

A custom policy should be enacted when specialized data treatment is needed. For example the preferential retention of certain data modalities may be desirable; one data type may be retained for ten minutes and another may only necessitate the retention of five samples. The only requirement to define a policy is that a filtered sensor data set be returned after it is enacted. This relates back to the requirements for application support, ease of use, and extensibility.

4.5.2 Modular Services

The SIXTH design allows for the dynamic addition of adaptors. As such, it is also prudent to facilitate the same functionality for modular services. This facilitates the goal of extensibility.

Through this mechanism, new functionality can be incorporated into SIXTH without modifying the core. Functionality appropriate for implementation in this manner includes sensor node discovery mechanisms such as Bluetooth and serial port scanners, logging services, and database connections. Modular services are distinct from services generated by service factories. A modular service is a singleton; when brought into the runtime it is loaded and operates itself.

4.5.3 Discovery Service

In Section 4.5.1 it was shown that a SIXTH deployment encapsulates all resources existing under a single host machine. In complement to this it is within the discovery module that all the known SIXTH deployments are maintained and managed. The discovery service is therefore the access point for applications wishing to engage with these resources. There is only one discovery service; however access to this is masked beneath a proxy layer, as typified by the proxy design pattern. This proxy is presented to the user upon request and is made cognisant of the callers access credentials. This opens the possibility to seamlessly limit access to resources on that basis.

The discovery service is decomposed into a set of services comprising the inbuilt SIXTH service layer. These services act across all registered SIXTH deployments. A description of these services in given in the following sections.

Tasking Service

The tasking service is responsible for the delivery of tasking messages (Section 4.4.6) to the resource specified by a given message. Figure 4.13 depicts the usage of the tasking service. Therein, a tasking
message is built by a user application and configured with meta-data in the form of a network adaptor descriptor (see Section 4.3.3) and the message is sent to the tasking service. The descriptor is then examined and matched to one or more network adaptors (see Section 4.4.7). Subsequently the message is delivered to the appropriate adaptor and acceptance or rejection is echoed to the caller. Messages which are not properly formed will be be rejected and an error raised. Finally, the message is passed out into the network to the targeted node(s).

![Tasking Service Diagram](image)

Figure 4.13: Tasking Service

The creation of this single point of contact for resource reconfiguration is intended to bolster ease of use; as a consequence of this design the client does not need to perform multiple calls to locate and reconfigure the network adaptor in one of many SIXTH deployments.

**Sensor Data Broker**

As discussed in Section 4.3.2, the role of a broker is to act as a delivery mechanism for some resource. The data broker functions as dissemination controller for sensor data (see Section 4.4.4) producers, which are typically entities existing within the network adaptor layer. Sensor data is passed to this broker from the producing entity, analysed, and disseminated to interested clients. Clients are instances of data receivers as discussed in Section 4.6.1. A client registers its interest with this service using one or more queries (see Section 4.4.5) which typify the interests of that client. Optionally a client can passively register to receive all sensor data.

The sensor data broker was added to SIXTH to split the responsibility for dissemination from the network adaptors into a standalone component. An appropriate division of responsibility is oft prudent in software design. Barnard Kroon has extended the default data broker to perform automatic tasking based upon the queries issued by clients and to build a semantic model of the network. This extension is discussed further in Section 7.6.

**Sensor and Node Brokers**

Brokers are defined for the provision of updates regarding sensor nodes (Section 4.4.2) and sensors (Section 4.4.3). As with the data broker, clients register their interest in updates regarding these resources through a query. As sensor nodes and sensors are taskable (see Section 4.3.5) resources their descriptors...
Providers

As discussed in Section 4.3.1 the role of a provider is to allow access to a set of resources. These constructs are utilized for manual access of resources in contrast to a broker that acts as a publisher. The network adaptor provider grants access to the set of adaptors which exist across all the connected SIXTH deployments. All providers allow for the selection of resources using a query described in Section 4.4.5. Providers are also defined for sensor nodes and data which also operate across all deployments. At the discovery service level the providers work essentially as aggregations of lower level providers. For instance each network adaptor is a sensor node provider; the discovery level sensor node provider exploits this by collecting all the sensor nodes from each network adaptor. The sensor data provider draws its information from the sensors which are sensor data providers.

Deployment Access

The discovery service provides methods for accessing SIXTH deployments (see Section 4.5.1). Additionally through this interface a deployment can be registered; registration is performed by a connector which has identified a remote deployment.

4.6 Application Hooks

As previously discussed in Section 4.4.4 applications which are built on top of sensor networks are dominated by the need for sensor data and network information. In line with the fundamental goal of application support identified in Section 3.6 it is paramount that the middleware platform provide a simple means of connecting into the sensor data stream.

4.6.1 Receivers

In furtherance of this goal of simplicity the delivery of information follows the well-worn publisher-subscriber design pattern. A provider, see Section 4.3.1, is cast in the role of publisher. The application defines some component to act as the recipient. SIXTH receiver components allow clients to selectively register for updates. This separation of concerns shields clients from information overload. Such detachment was absent in early builds of SIXTH and it quickly became apparent that typically a client utilized one method of the multi-method receiver and implemented the remainder as stubs. In the following sections, a discussion of the various kinds of information receivers is given.

These receivers are examples of Inversion of Control mechanisms previously discussed in Chapter 2; Queries are also representative of this paradigm as in both cases the middleware calls application-specific code which is responsible for handling an event. The following receiver instances are specific extensions of the receiver concept which are defined such that the client can receive different kinds of information about problem domain.

- **Sensor Data Receiver**: A sensor data receiver is defined for the purposes of receiving a stream of sensor data (Section 4.4.4). This receiver can be registered using a query, see Section 4.4.5, or the client can opt to receive the entirety of the unfiltered data stream. Figure 4.14 depicts the registration of a query with the data broker and the subsequent streaming of sensor data to the application. Section 5.6.2 discusses how these ideas are achieved through composition with generics.
Network Adaptor Receiver: A network adaptor receiver is informed of status information regarding the network adaptors (see Section 4.4.7). These notifications include the initialization of a new adaptor, the removal of an adaptor, or changes to its underlying network which render the adaptor inoperable.

Sensor Node Receiver: Sensor node receivers are passed notifications relating to sensor nodes (see Section 4.4.2). Information dissemination to this receiver is triggered by events such as node discovery, failure, or time-out.

Sensor Receiver: A sensor receiver is provided with information relating to the state of sensors (see Section 4.4.3). These changes in status include the initial detection of the sensor, notification, reconfiguration, sensor time-out, and reconfiguration.

Deployment Receiver: This receiver is streamed updates regarding the set of SIXTH deployments (see Section 4.5.1). When a SIXTH deployment is connected a notification will be issued to this receiver. This serves as a key alert to the new resources which are available through the newly connected SIXTH deployment. Similarly, a notification is posted if there is a loss of connectivity to a deployment.

Tasking Receiver: This class of receiver will be notified of reconfiguration by means of a tasking message (see Section 4.4.6). This enables an application to know of an upcoming change in the behaviour of a taskable component. As a result, applications can make decisions or alter behaviour based on a pending changes in the associated network.

Event Receiver: This receiver was conceived largely as an extension point which can receive generic event notifications distributed by the middleware framework. These events are those for which no formalism exists within SIXTH. It is in this way that some consideration can be given for the unforeseen.

In the receiver components access to taskable (Section 4.3.5) resources is restricted. When it is deemed appropriate by the disseminating broker (see Section 4.3.2) a descriptor object, as discussed in Section 4.3.3 is dispatched in lieu of the taskable entity. For example. This variable resource delivery is achieved through a flexible interface design which allows for multiple message types. The implementation of these receiver components is discussed in Section 5.6.2.
4.6.2 Resource Monitors

For the purposes of efficient monitoring, decoupling, and message routing within SIXTH a collection of receiver monitors is defined to maintain references to an associated set of receivers. Resource monitors work in consort with brokers as a monitor has knowledge of the receivers that desire to access the unfiltered streams. [Figure 4.15](image) provides a high-level view of how these resource monitors operate. A network adaptor is shown issuing an update to a resource monitor which is then disseminated to all the appropriate receivers. It is desirable that a receiver not need to directly register with the resource monitor rather it should be found through some asynchronous detection mechanism. Issues relating to such registration are implementation specific and are discussed in [Chapter 5](#).

In earlier instantiations of SIXTH resource monitoring was contained within the network adaptors. This functionality was split off in an effort to divide responsibility and increase code reuse, as other components beyond a network adaptor may have need of the same monitoring services.

![Image](image)

Figure 4.15: SIXTH Deployment-receiver Communication

4.7 Conclusions

This chapter has provided a detailed overview of the system design of the SIXTH sensor middleware showcasing the core concepts and how these are interwoven in service of delivering upon the design desiderata given in [Chapter 3](#).

Modelled on the Adaptor Design Pattern the SIXTH network adaptor delivers upon the goal of Heterogeneity of Source. The network adaptor provides an extensible and flexible method of adding new information sources into SIXTH. This component is logically decomposed into interchangeable wrappers, translators, and connectors.

Abstraction & Uniformity were identified as key middleware objectives. The entire design of SIXTH is cognisant of this goal but its realization is most clear in the core abstractions which provide abstracted and uniform representations of domain concepts such as sensors, nodes, networks, result filtering and reconfiguration.

SIXTH is designed to be extensible in many respects. Modular services define extension points by which to add new services into the core middleware. New adaptors are added to extend the suite of information sources available. The abstract representations of domain concepts are intended to be overridden and extended by developers. Network configuration is recognized as inherently heterogeneous, as such reconfiguration options support the inclusion of runtime defined reconfiguration utilities. This is also in service to the goal of runtime reconfiguration. There is a notion of concept level extensibility for
queries, providers, and brokers in allowing these abstractions to be defined in terms of as yet unknown objects. The peer-to-peer design of the middleware supports scalability.

The application hooks, queries, and tasking message components provide application support to allow for easy integration of applications with SIXTH in a lightweight fashion. The above discussed contribute to the overarching goals of flexibility and ease of use; the design of SIXTH is explicitly intended to allow for flexible usage in a myriad of scenarios. This is evidenced in Chapter 7 in which a diverse set of SIXTH-driven case studies are explored. In Chapter 5 the design that was described within this chapter is utilized to build the reference implementation of the SIXTH framework.
5 | Implementation

“The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures.”

— Frederick Brooks, The Mythical Man-Month (Brooks Jr., 1975)

5.1 Preface

Chapter 3 outlined a middleware design desiderata which provided a roadmap for the design of the SIXTH middleware which was the subject of Chapter 4. Therein, it was made manifest how such a design would service the goals of the thesis given in Section 1.3. This chapter provides a detailed description of the implementation of the SIXTH middleware so as to make concrete the achievement of these objectives.

In discussing SIXTH it must first be established that SIXTH is both a specification and a consequent default implementation. This conscious uncoupling of the interlocking interfaces of the API and the implementation is a deliberate gesture. Uncoupling allows for the implementation to be written against the interface specification in keeping with the common wisdom that one should “program to the interface” (Vlissides et al., 1995). This approach attempts to ensure that interaction with any implementation need not be concerned with internal particulars of how functionality is delivered. Additionally, this allows SIXTH and its governing design principles to be reconstituted in multiple programming languages or within constrained platforms. This decision echoes the ethos of the desiderata from Chapter 3 in particular with regard to extensibility and the overall goal of flexibility. The following section provides an overview of the technology stack which underpins the development of SIXTH.

5.2 Technologies

This section provides a discussion of the programming language and modularity framework selected to underpin the implementation of the SIXTH middleware.

5.2.1 Programming Language

The default SIXTH implementation contributed by this thesis is implemented in the Java programming language version 1.7. Java was chosen for implementation due to its cross-platform nature, static typing, maturity, and strong Object-Oriented essence. Chapter 4 considered a decomposition of the problem domain elements akin to an Object-Oriented Programming (OOP) exercise; the implementation of those components is discussed herein. The widespread adoption of Java is generally attributed to the “write
once, run anywhere” (Kramer, 1996) nature of program code developed for the Java Virtual Machine (JVM). Java has been utilized for in-network sensor programming on the Sun SPOT platform; as such this presents the opportunity for a common code-base for device and gateway-side development. The incorporation of Java in Android provides a vehicle to situate SIXTH entirely on a sensor platform; work carried out to this end is discussed in Section 7.7.

5.2.2 Modularity Framework

As per the desideratum identified in Chapter 3, SIXTH is envisaged as a modular framework into which new functionality is injected with ease so as to enable extensibility and scalability and middleware level runtime reconfiguration. As no such modularization model is present in the Java language it was necessary to utilize a framework for this purpose. The Open Service Gateway initiative (OSGi) (Marbles et al., 2001) component framework is utilized to deliver modularity.

OSGi was chosen over other modularization frameworks such as the Java Plug-in Framework (JPF) (Cheyer et al., 2005) for its maturity and proven success in large scale projects and the provision of a decoupled service-oriented architecture. One such notable project is the Eclipse Integrated Development Environment (IDE) (desRivieres et al., 2004) which has proven itself as an extensible and reliable platform. OSGi delivers upon the need for a dynamic, modular, and fault tolerant service platform for development. OSGi has been utilized within the domain of sensor network middleware development previously, in systems such as WSNWare (Viani et al., 2013), Atlas (King et al., 2006), and SStreamware (Gurgen et al., 2008).

As per the design requirements outlined in Chapter 3 SIXTH must be capable of runtime functionality addition and graceful component failure. This is facilitated, in part, through the bundle management components of OSGi. Bundles are core OSGi components; A bundle is a JAR (Java ARchive) containing meta-data detailing the bundles imports, exports, and requirements. This meta-data defines the bundles relationship with OSGi and other bundles. This meta-data is stored in the manifest.mf file. Typically an activator class is specified to provide a hook into the framework and to start the execution of a bundle. Figure 5.1 illustrates the high-level system architecture of an OSGi platform implementation; three bundles are shown communicating with the framework and each other.

![Figure 5.1: Open Services Gateway Initiative System Architecture](http://commons.wikimedia.org/wiki/File:Osgi.svg)

The core of SIXTH is split into two bundles, one for the API, and the other for the implementation. All other components such as individual adaptors (Section 4.4.7) are implemented in individual bundles.
Bundles are interconnected through the OSGi service bus. For example when a data receiver (see Section 4.6.1) becomes active it registers itself on the service bus; the bus is observed by a resource monitor (see Section 4.6.2) which forwards sensor data to that receiver.

In the remainder of this chapter, the major components of the SIXTH middleware implementation are examined in turn. In doing so particular attention is paid to the transformation of design into an implementation contract and subsequent concrete implementation.

The realization of the design from Chapter 4 is underpinned by a set of interfaces which abstract and decouple typical behaviour. In this context an interface is defined as a specification of what operations a software component can perform without saying anything about how this is to be achieved. As an example the method \texttt{public int getAge(String name)} is defined, this states that such a method can be executed with a string input, and an integer will be returned. Such a method will adhere to this operational contract in all implementations, but the method of operation will vary.

The following is a detailed overview of the implementation of the core SIXTH components. Section 5.3 provides a discussion of the generic interfaces and implementations utilized in SIXTH. Section 5.4 describes the implementation of several compositional elements. Such components form parts of the representational objects which are the subject of Section 5.5. Section 5.6 discusses application level services which facilitate the control of networks and the receipt of sensory observations. Having described the core of the implementation Section 5.7 describes several systems for interaction with SIXTH core, these include a graphical user interface and RESTful web services. Section 5.8 concludes this chapter drawing a comparison of SIXTH and earlier middleware efforts.

### 5.3 Generic Interfaces

Section 4.2.2 defined generic programming and discussed its applicability within software design. This section provides a description of several key generic interfaces in SIXTH. These are examined in turn highlighting the purpose of each and the reasons for applying generality to the component. Section 4.3 discussed the application of generics to the SIXTH design as providing concept level extensibility; this point is furthered clarified herein where it can be seen that the concepts of querying, brokering, and providing are applicable to many resources. This includes those not present in the core design.

As discussed in Section 5.2.1 Java is the chosen implementation language for SIXTH. In Java 5 generic programming facilities were incorporated into the language; generics served to reduce verbosity and enhance static checking capability. (Bracha et al., 1998) discusses the technical challenges of bringing generics into Java and the limitations of the approach. Among these limitations is that due to the erasure of type information during compilation a class cannot implement a parametrized interface more than once. For instance, a single object cannot define itself to be both an \texttt{IProvider<ISensorNode>} and an \texttt{IProvider<ISensor>}. In the main this is not problematic as it promotes a healthy separation of concerns and necessitates composition over inheritance, however it can be restrictive.

The naming convention for interfaces in SIXTH follows the Hungarian notation (Simonyi, 1999). Each interface name is prefixed by ‘I’ to denote the nature of the class. This is the only application of the notation in SIXTH; it was so chosen to aid readability and separate interface and implementation classes at a glance. The following sections describe the generic interfaces of the SIXTH Middleware.

#### 5.3.1 Provider

Section 4.3.1 discussed the concept of a provider as an entity which allows access to a set of associated resources. The concept is formalized in the \texttt{IProvider<T>} interface. Genericness means
that an \texttt{IP}{}rovider<\texttt{T}> can be parametrized as \texttt{IP}{}rovider<\texttt{IS}{}ensor>. This parametrised implementation facilitates access to a set of sensors; a sensor node is an example of this kind of provider.

\textbf{Figure 5.2} depicts the UML class diagram for \texttt{IP}{}rovider<\texttt{T}> and its subclasses. \texttt{IS}{}ensor\texttt{Node} and \texttt{IS}{}ensor both, directly or indirectly, extend the \texttt{IP}{}rovider<\texttt{T}> interface. These interfaces are described in Section 5.5.4 and Section 5.5.5 respectively.

The \texttt{IP}{}rovider<\texttt{T}> interface defines three methods: (i) \texttt{getAll()} (ii) \texttt{getMatches(IGenericQuery<\texttt{T}>)} (iii) \texttt{exists(IGenericQuery<\texttt{T}>)} . The following explains each methods role:

The invocation of \texttt{getAll()} will return a list of resources of type \texttt{T} accessible to the provider. For example consider a sensor node which is an \texttt{IP}{}rovider<\texttt{IS}{}ensor>; the list constitutes the sensor complement of the node. \texttt{getMatches(IGenericQuery<\texttt{T}>)} returns the subset of resources which match the selection criteria of the \texttt{IGenericQuery<\texttt{T}>} argument. The \texttt{exists(IGenericQuery<\texttt{T}>)} returns true if a matching resource exists; otherwise the result is false. The implementation of queries is discussed in Section 5.3.2.

The following sections describe the adaptor, data, and dual provider subclasses of \texttt{IP}{}rovider<\texttt{T}> as depicted in \textbf{Figure 5.2}.

\textbf{Adaptor Provider} The \texttt{IAdaptorProvider<\texttt{T}>} interface defines an extension to the provider. As \textbf{Listing 5.1} depicts, genericness is constrained to classes which implement \texttt{IAdaptor} which is the formalism of the adaptor pattern in SIXTH. \textbf{Figure 5.2} conveys that \texttt{IAdaptorProvider<\texttt{T}>} defines a single additional method. The \texttt{get(String)} method facilitates the retrieval of an adaptor via its unique \texttt{String} identifier.

\textbf{Listing 5.1: IAdaptorProvider Interface}

\begin{verbatim}
1  public interface IAdaptorProvider<T extends IAdaptor> extends IProvider<T> {
2      T get(String name);
3  }
\end{verbatim}

\textbf{Data Provider} \texttt{IDataProvider} is not a parametrised interface, rather, it extends \texttt{IP}{}rovider<\texttt{IS}{}ensor\texttt{Data}> to serve as an access mechanism for sensory observations. \texttt{IDataProvider} is implemented by any SIXTH component which can provide sensor data. The provision of the data may be direct, in relation to the source, as with a sensor (see Section 4.4.3) or indirect as in a network adaptor (see Section 4.4.7). \textbf{Figure 5.2} conveys the addition of a single method; \texttt{getMap()}, which returns a \textit{Map}<\textit{IM}{}odality,
List<ISensorData>>. The keys of which are sensing modalities and each value is a list of corresponding observations.

**Dual Provider**  
IDualProvider<T, U> is a provider for two associated types where U is a descriptor of T. Three methods are defined by the interface. `get(U)` returns an instance of T which is described by the method argument. `getDescriptors()` returns a list of descriptors for all resources of type T. `getDescriptors(IGenericQuery<U>)` returns all descriptors which satisfy the criteria.

Figure 5.3 illustrates the interaction between a consumer and a provider. SensorNodeProvider is a subclass of IDualProvider<T, U>, ProviderConsumer is a simple client class the implementation of which is immaterial. The ProviderConsumer invokes `getMatches(IGenericQuery<T>)`; consequently the query is run to select the matching resources; finally a list is returned to the ProviderConsumer.

![Figure 5.3: UML Sequence Diagram Depicting Interaction With the Node Provider](image)

**5.3.2 Query**

Section 4.4.5 formalised a query as a means of expressing a selection criteria. This section discusses the implementation of queries within SIXTH. The implementation of queries is grounded in the Specification design pattern (E. Evans, 2004). Specification defines the encapsulation of boolean determinant business logic within an object. Figure 5.4 depicts the UML class diagram of the generic Specification. Therein the ISpecification interface specifies four methods; `isSatisfiedBy(Object)` contains the business logic or criteria, a boolean truth value is returned when the argument is satisfactory. The remaining methods specify a means of logically composing multiple specifications. (E. Evans, 2004) determined this approach as appropriate for maintaining logic within the problem domain, the author advocated the utilization of the pattern under the following conditions:

- To determine if an object fulfils a need;
- Provide an object selection criteria for filtering.

Both of these conditions directly map to the manner in which queries are intended for use in SIXTH. It was decided to maintain “query” as a naming convention as it domain appropriate.

A query is formalised in the IGenericQuery<T> interface. Figure 5.5 depicts the UML class diagram for this interface, an abstract subclass, and a concrete implementation. The interface therein depicted
is an almost exact match for the Specification pattern depicted in Figure 5.4, as such this is quite a literal application a design pattern with no major changes needed to mould the concept to the problem domain. By virtue of generic programming a query can be defined for any class. For example $IGenericQuery<ISensorData>$ and $IGenericQuery<ISensorNode>$ define queries in respect of sensor data and nodes respectively. In this thesis support is explicitly provided for the resolution of queries pertaining to sensors, nodes, and data. Selection criteria can be complex or simple; the only restriction on implementation is to fulfil the contract of the $IGenericQuery<T>$ interface.

The $IGenericQuery<T>$ interface defines five methods. $match(T)$ returns true if the argument matches the selection criteria. $filter(List<T>)$ filters the input list based on the criteria and returns it; in (E. Evans, 2004) the second scenario for application of the Specification was in filtering for this reason it was deemed prudent to extend the defined interface of Specification to allow a list to be passed in directly. The remaining three methods specify a means of combining queries using boolean logic (and, not, and or).
Figure 5.5 depicts the AbstractGenericQuery\(<T>\) class which extends from IGenericQuery\(<T>\). The purpose of this class is to provide a default implementation of filter(List\(<T>\)); this operates based on match\(T\), the implementation of which is left for concrete subclasses. All other methods are implemented in AbstractGenericQuery\(<T>\); the manner of implementation is described below. ModalityQuery shown in Figure 5.5 is an example of such a subclass, parametrized for ISensorData.

Aggregate Query  An AggregateQuery is composed of many sub-queries; the selection criteria is the satisfaction of all sub-queries. In the Structured Query Language (SQL) (see Section 2.5.5) this is represented by “AND”.

Conjugate Query  In a similar fashion a ConjugateQuery is synonymous with an “OR” operator. If the resource matches any of the sub-queries it is accepted by the conjugate. An AllQuery accepts everything, this is akin to “SELECT ALL”. A NegationQuery works as a “NOT” operator. This is constructed from another query and selects all resources which do not match to the source query. The IQuery interface specifies a non-parametrised query which operates on ISensor, ISensorNode, and ISensorData. This should be utilized when the selection criteria is appropriate for all three. Conjugate and Aggregate queries are how the AbstractGenericQuery\(<T>\) class defines and and or combinations.

In WSNWare (Viani et al., 2013) the concept most similar to queries is a Validator. This too defines acceptance or rejection of an object. It is explicitly defined for nodes and sensor observations as static nested classes e.g. Node.Validator. As this approach does not use generics a validator defined for another object type cannot be seamlessly used in the framework and would need to be created and integrated by an extending developer. The Validator implementation lacks any functionality for the composition of its instances as described above for SIXTH. It is the opinion of the author that utilizing static nested classes in this instance is to the detriment of comprehension for the sensor network developer.

5.3.3 Tasking Message

Section 4.4.6 defined a tasking message as an abstraction for the representation of reconfiguration requests. Section 5.4.1 introduces the ITaskable interface which is implemented by all configurable SIXTH components; these components accept tasking messages. In this section the implementation of tasking messages is discussed.

![Figure 5.6: UML Diagram of the ITaskingMessage](image)

The ITaskingMessage\(<T>\) formalises the packaging of reconfiguration requests; Figure 5.6 provides UML for this interface, its concrete implementation, and the CommandType enumerated type. An enumerated
Listing 5.2: Configuring Sampling Frequency

```java
public static ITaskingMessage<Frequency> setSamplingFrequency(long nodeID, String sensor, Frequency frequency) {
    return new TaskingMessage<Frequency>(nodeID, sensor, frequency, CommandType.SETFREQ);
}
```

type, or enum, is a restricted set of distinct identifiers. 
CommandType identifiers define the intent of a tasking message.

The TaskingMessageFactory class simplifies the creation of tasking messages through a set of static methods to instantiate configuration requests of differing intents. This is an example of the Factory creational pattern identified by the GoF (Vlissides et al., 1995, p99). Listing 5.2 depicts a method for the creation of a tasking message to modify the sampling frequency of a specific sensor on a given node. A Frequency object encapsulates the specification of sampling frequency. The instruction type is SETFREQ.

Tasking can be performed by invoking the `task(ITaskingMessage<T>)` of an ITaskable component. To reduce complexity the tasking service is defined to route a request to the intended resource. The service itself implements ITaskable; when a request is received the AdaptorDescription is examined to determine the target and subsequently relay the message.

Figure 5.7 depicts a UML sequence diagram for a typical tasking operation. The NodeConfig, a component of the GUI suite discussed in Section 5.7.2, invokes the TaskingService. The service determines the intended recipient; the Sun SPOT network adaptor. The adaptor accepts the request and dispatches it to the SunspotMessageWrapper for dispatch to the network.

![UML Sequence Diagram for Tasking Interaction](image)

Listing 5.3 depicts the program code skeleton provided by the AbstractWrapper class in GSN to support network reconfiguration. The sendToWrapper method has two variants, the first taking an Object and the latter a action type, set of parameter names and corresponding values.

This approach is compared and contrasted with the usage in SIXTH of the TaskingMessage abstraction.

- In the first case where an Object is used all information regarding the task to be completed has to be conveyed by that object. In SIXTH the TaskingMessage holds all relevant addressing, and general tasking information for the reconfiguration request. Only the very specific information is held in a specialised Object class descendant.
• As evidenced in Listing 5.2 SIXTH utilizes common classes such as Frequency to encapsulate common requests. In GSN such is done with less intuitive byte arrays.

• In the latter sendToWrapper method, there is more of a movement toward the approach used in SIXTH. It is seen that a specific field conveys the action to be performed. This is akin to the enumerated COMMANDTYPE used in TaskingMessage.

• The use of a separate array for parameter names, and values is seen as confusing. In such a case SIXTH would use a key-value map. This approach is also espoused in WSNWare.

• The boolean return value is useful, this was identified for the design of the SIXTH ITaskable interface.

• Consequent to the previous item, SIXTH provides ITaskable as a dedicated interface for reconfigurable elements which may be sensors, sensor nodes, services, or any at present unknown quantity. Such a unifying abstraction is absent in GSN, as such there is a tighter coupling between the client and the Wrapper.

• Section 4.5.3 defined the Tasking Service which decouples clients from the elements being reconfigured. From our review of the GSN code-base such a separation is absent.

5.3.4 Broker

Section 4.3.2 defined a broker as the intermediary in an asynchronous publisher/subscriber paradigm. This section discusses the implementation of this generic concept in SIXTH. IBroker<R,U> specifies the broker concept; therein U is associated with R most often as a descriptor. Figure 5.8 depicts the UML class diagram for this interface. The registerInterest(IGenericQuery<R>, IQueryReceiver<R,U>) method is used by a consumer to register its interests. The query typified the topics of interest and the consumer implements the IQueryReceiver<R,U> interface. A second registration method includes a boolean parameter; if true the broker to dispense historical query matches.

Three brokers are explicitly defined in the SIXTH architecture, however this is intended as an extensible concept applicable in a multitude of scenarios. The explicitly defined brokers operate for sensors, nodes, and data. The following describes the data broker as an example.

Data Broker The data broker is parametrized as IBroker<ISensorData, String>. This broker accepts subscriptions for the sensor data producers. When a producer, such as a sensor, supplies data it is forwarded to the data broker. The observation is compared against the query-based subscription parameters and when a match is found the sensor data is forwarded to the subscriber.
Figure 5.8: Broker UML Class Diagram

Figure 5.9 depicts a UML sequence diagram showing the interaction between the DefaultDataBroker and an information consumer, DataReceiver. It is observed that the DataReceiver registers interest and in response the broker begins to stream.

Figure 5.9: UML Sequence Diagram for Broker and Consumer Class

5.4 Compositional Components

Section 4.3 defined a compositional component as forming part of another component e.g. a sensor has-a device specification. This section discusses the implementation of several compositional components. It is noted that a provider is a compositional component however it was decided to place the discussion of it in the context of other generic interfaces.

5.4.1 Taskable

Section 4.3.5 defined a taskable component as any reconfigurable resource; the configuration of which is performed by tasking messages. The ITaskable interface specifies this concept for all SIXTH components. Explicitly defined taskable entities include network adaptors, nodes, and sensors.

Figure 5.10 depicts the UML class diagram for ITaskable and four implementing interfaces. Three of these implementers specify interaction with adaptors, sensors, and nodes. The fourth is the tasking service which was described in Section 4.5.3.

Figure 5.10 conveys the minimalism of ITaskable defining three methods. The central method is task(ITaskingMessage<?>) which is responsible for the examination and fulfillment of reconfiguration requests. A boolean return value denotes acceptance or rejection of a request.

Section 5.3.3 established that each tasking was instantiated with a CommandType as a general specification of intent. As a means of preliminary filtering a ITaskable entity expresses the acceptable CommandType identifiers. This is specified by acceptedCommands() and accepts(CommandType).
acceptable command type does not guarantee that a request is achievable as it may specify unavailable resources, or unobtainable sampling frequencies.

### 5.4.2 Observer

$I\text{Observer}<T>$ is a specification for any component which explicitly requires an awareness of another components creation or modification. This is an example of the observer design pattern. Figure 5.11 depicts the UML class diagram for the observer pattern. Owing to the inherently malleable nature of pattern application the implementation herein differs in form but not function. In some instances $I\text{Observer}<T>$ registers with a resource monitor, conceptually described in Section 4.6.2 which acts as a broker. Adaptors which are instantiated as $I\text{Observer}<I\text{CredentialedReceiver}>$. $I\text{CredentialedReceiver}$ is the base class for information consumers. Sensor nodes are parameterized as $I\text{Observer}<I\text{Sensor}>$ denoting that the observables are sensors which inform the sensor node of their state; this more closely follows the pattern definition, however both schemes adhere to the intent.

![UML Class Diagram of the Observer Design Pattern](http://upload.wikimedia.org/wikipedia/commons/8/8d/Observer.svg)

### 5.4.3 Device Specification

Section 4.4.2 discussed the design of a virtual sensor node abstraction and noted the necessity to provide a mapping to the heterogeneous hardware properties of the device. This notion is also applicable to sensors, which may have distinct properties. The interface to this property mapping is provided by the $I\text{DeviceSpecification}$ interface; its default implementation acts as a wrapper over a key-value data store. Individual properties are represented by an instance of $\text{Property}$ discussed below.
Property

The lightweight *Property* class provides access to an identifier, the value, and the last update time. One such instance corresponds to the following tuple ⟨“state”, “ON”, 1392163200⟩. This denotes that the device was “ON” and the observation was recorded at 1392163200 UNIX time.

5.4.4 Device Monitor

The *IDeviceMonitor* interface describes an entity which is monitoring the state of another component. For example virtual sensors monitor for the time-out of their physical counterparts based on known sampling rates and adjustable reporting thresholds. The interface allows for the toggling of state monitoring.

5.5 Representational Objects

This section describes the components of the implementation which are representative of the problem domain.

5.5.1 Adaptor

The primary use for SIXTH necessitates the abstraction of communication with sensor networks by masking heterogeneity through the presentation of uniform interfaces. Towards the goals of extensibility and flexibility the *IAdaptor* interface is defined as the bottom level of API compliance for all services in SIXTH.

This specification provides a unified means of accessing service identification. Figure 5.12 depicts the UML class diagram for *IAdaptor*; it can be seen that all adaptors are taskable components through the extension of *ITaskable*. The interface defines two accessor methods; the first returns an adaptor descriptor and the other the SIXTH deployment. Mutator methods are defined to unregister the adaptor and modify the deployment.

![Figure 5.12: UML Class Diagram of Adaptor Interface](image)

SIXTH is a modular framework in which bundles may be removed from the runtime environment. This unregister method is called by the class controlling removal to allow the resource to perform any necessary unhooking from the framework. In the context of the current implementation technology, OSGi, this means properly removing all service registrations from the service bus.

The following section discusses the implementation of sensor network adaptors which are extended from the above discussed *IAdaptor* interface.
5.5.2 Sensor Network Adaptor

Section 4.4.7 presented the concept of a sensor network adaptor as a uniform interface to heterogeneous networks of sense-capable devices. The encapsulation offered by this solution allows new networks to be gracefully integrated. The following sections discuss the interfaces and classes which compose the default implementation of sensor network adaptors in SIXTH. An adaptor may encompass multiple sensor networks which are represented independently as discussed in Section 5.5.3.

Figure 5.13: UML Class Diagram for ISensorNetworkAdaptor

**ISensorNetworkAdaptor**

The **ISensorNetworkAdaptor** interface formalizes interaction with a sensor network. Figure 5.13 depicts the UML class diagram for the **ISensorNetworkAdaptor** and its antecedents, **IAdaptor** and **IObserver<ICredentialedReceiver>**. The interface also provides accessor methods for sensor node and data providers, which supply all known sensor nodes and data. The following section discusses the default implementation of **ISensorNetworkAdaptor** which realises the API described by **IAdaptor**, **IObserver<ICredentialedReceiver>**, and **ISensorNetworkAdaptor**.

**AbstractSensingNetworkAdaptor**

The design of SIXTH is formally defined by its interfaces and distinct from any implementation. This facilitates a high degree of flexibility as no defaults are enforced. A developer is free to implement an adaptor from the API specification alone. However, for the vast majority of developers this is neither desirable nor advantageous.

In the interest of code-reuse and reducing development overhead an abstract sensor network adaptor is defined. For clarity abstract, in this instance, means a class which cannot be instantiated. A “concrete” subclass must be defined. The subclass often bears the responsibility for completing the implementation of the interface specification; in Java this is all methods marked abstract in the inheritance hierarchy. The abstract adaptor, termed **AbstractSensingNetworkAdaptor**, forms the basis for all sensor network adaptors discussed in this thesis.

**AbstractSensingNetworkAdaptor** handles low-level interaction with other middleware components. Listing 5.4 depicts the **discoveryRegistration(String networkName)** method which is responsible for the registration of the adaptor with the SIXTH Deployment, the creation of a descriptor, and the dissemination of a creation notification.
Listing 5.4: AbstractSensingNetworkAdaptor Configuration Method

```java
protected void discoveryRegistration(String networkName) {
    UUID id = UUID.randomUUID();
    discoveryService = SIXTHFramework.getDiscovery(new Credentials(networkName,
        IAdaptorDescriptionSENSOR_ADAPTOR, id));
    SIXTHDeployment deployment = discoveryService.getLocalDeployment();
    this.adaptorDescription = new AdaptorDescription(id, networkName,
        IAdaptorDescriptionSENSOR_ADAPTOR, deployment.getDescription());
    credentials = new Credentials(adaptorDescription.getName(), AdaptorDescription.
        SENSOR_ADAPTOR, id);
    this.adaptorDescription.setLocal(true);
    deployment.register(this);
    notify(getDescription(), DeviceStatus.NEW.toString());
}
```

Listing 5.5: Lookup Function & Dynamic object Creation

```java
protected ISensorNode sensorNodeLookup(long id) {
    ISensorNode sensorNode = sensorNodeNetwork.getSensorNode(id);
    log.info(sensorNode + "");
    if (sensorNode == null) {
        sensorNode = new SensorNode(id, getDescription().getNetworkName(), this);
        addSensorNode(sensorNode);
    }
    return sensorNode;
}
```

AbstractSensingNetworkAdaptor also provides default sensor node creation as a result of sensory observations or explicit identification. Listing 5.5 depicts the program code responsible for creating virtual sensor nodes when given a node ID.

When a piece of sensor data, represented by the ISensorData interface, is produced the abstract adaptor is responsible for configuring meta-data such as sensor node description, storing the data within the virtual sensor, signing the data to guarantee provenance and arranging dissemination through the broker. The code which achieves this responsibility is illustrated in Listing 5.6.

From an examination of the code bases of both GSN and WSNWARE it has been determined that neither middleware provides an appropriate separation from sensor observation consumers and the producers e.g. wrappers and message-sources. Listing 5.6 has demonstrated that in SIXTH the Adaptor is not cognisant of the consumer entities. Such responsibility is left to the broker, as befits a separation of concerns. In GSN the output format of an adaptor, wrapper in GSN parlance, is fixed. As one may ascertain from Listing 5.6 this is not the case in SIXTH. This is because is some cases the format of the data may not be known when the adaptor is defined for example dynamic conversion of XML or JSON to key-value

Listing 5.6: Adding ISensorData to an Adaptor

```java
public void addData(ISensorData data) {
    ISensorNode node = sensorNodeLookup(data.getNodeID());
    data.setNodeDescription(node.getDescription());
    ISensor sensor = node.sensorLookup(data.getSensorDescription());
    sensor.addData(data);
    data.sign(priv, signature);
    discoveryService.getDataBroker().send(data);
}
```
pairs. This lack of rigidity is an advantage in SIXTH over GSN.

Given the basis of AbstractSensingNetworkAdaptor it is necessary for a developer to implement Connector and Translator components for the data sources of concern. These components were described in Section 4.4.7; implementation of the aforementioned is network specific. When this is complete the adaptor is capable of producing ISensorData instances in compliance with the API. The method depicted in Listing 5.6 is invoked passing in the sensor data. At this point virtual nodes and sensors are generated and the sensor data is disseminated by the broker. In the method body the TaskingMessage is examined and if an achievable request is specified then the wrapper functions translate the intention into a form understood by the sensor network; this is then dispatched by the Connector.

Figure 5.14 depicts a high level overview of the composition of an adaptor which is extended from AbstractSensingNetworkAdaptor. Therein the obligations of the adaptor to provide bi-direction message translation and data source connectivity are illustrated. Figure 5.15 is an expansion of Figure 5.14 augmented to include code fragments from the Sun SPOT and Twitter network adaptors. The code fragments capture the message wrapping, data connection, and communication aspects of each adaptor. This serves to underscore the applicability of common operational principles even when the information sources are radically different in several dimensions.

The following section provides a discussion of the implementation of sensor nodes which play a central role in the above described behaviour of AbstractSensingNetworkAdaptor. The interface specification and the default implementation are considered.

### 5.5.3 Sensor Network

Section 4.4.1 commented on the inclusion of an explicit network object in the domain representation. Each sensor network is owned by an adaptor and possesses a has-many relationship with sensor nodes. Figure 5.16 depicts the UML class diagram for ISensorNetwork and the default implementation. The default implementation is a simple data container which provides a separation of concerns for the adaptor in providing an abstraction of the network. In general practice an adaptor has one network, however in the interests of flexibility an adaptor can have many networks. This may be particularly advantageous in
Figure 5.15: Applying the SIXTH Design to Heterogeneous Data Sources
the cyber sensing context where the connected environment may be best represented as more than one collection of entities.

Figure 5.16: UML Class Diagram for ISensorNetwork

5.5.4 Sensor Node

Section 4.4.2 described a real-world sensor as a hardware platform which contains one or more sensors which sample diverse phenomena. In the design sensor nodes are split from their sensors. This section discusses the implementation of this design. In Section 5.5.2 the sensor network adaptor was modelled through a composition of interfaces. This model is also applied to sensor nodes. Adaptors and nodes share a common antecedent in their inheritance hierarchies, ITaskable.

ISensorNode provides the interface for interacting with sensor nodes. [Figure 5.17] depicts the UML class diagram of this interface and the others from which it is composed. Section 5.3.1 described the IProvider<T> interface and its role as an accessor for resources of type T. A sensor node implements this interface parametrized as IProvider<ISensor> denoting its usage in allowing access to the virtual sensors.

The following sections provide a detailed description of the ISensorNode interface and the consequent default implementation.

Figure 5.17: UML diagram of the SIXTH Sensor Node Implementation
Listing 5.7: Sensor Node Production of Valid ISensorData

```java
public ISensorData produceData(IModality modality, Map<IComponentModality, String> values, String type) {
    ISensorData data = new SensorData(getAdaptorDescription(), getID(), modality, values, type);
    getAdaptor().addData(data);
    return data;
}
```

ISensorNode

The ISensorNode interface extends from those above discussed to form a virtual representation of a generic sensor node. Figure 5.17 conveys that a sensor node implements its own descriptor; a method is also provided to return a standalone SensorNodeDescription for dissemination in a security concious scenario. SensorNodeDescription defines accessor methods for (i) node ID (ii) type (iii) network ID and (iv) associated adaptor.

The SensorNode class is the default implementation of the sensor node interface. Listing 5.7 provides the program code, from SensorNode, for the produceData method; this method aids in the generation of valid ISensorData.

The ISensorNode interface contains two methods pertaining to the nodes neighbours. A neighbour may refer to nodes one-hop distant in a routing scheme or an associated cyber-sensor. getNeighbours() provides a list of ISensorNodeDescription for all neighbouring nodes. getRelationshipProps(SensorNodeDescription) returns a map of key-value pairs specifying relationship properties between two nodes. Typically such information includes message propagation details, physical distance, and shared regions.

Sensor Node Group

Section 4.3.4 defined a group as a collection of related entities accessed and controlled as single unit. The SensorNodeGroup class defines an aggregation of sensor nodes which can be tasked as if they were a single node. Figure 5.18 depicts the UML for SensorNodeGroup which is parametrized as IProvider<ISensor> and IGroup<ISensorNode>.

Figure 5.18: UML Class Diagram for SensorNodeGroup
Listing 5.8 depicts the program code for a method which generates a \textit{SensorNodeGroup} instance from a given list, filtered through a query.

\begin{lstlisting}[language=Java]
public static SensorNodeGroup generate(List<ISensorNode> nodes, IGenericQuery<
  ISensorNode> query) {
  List<ISensorNode> nodesList = query.filter(nodes);
  return new SensorNodeAggregate(nodesList);
}
\end{lstlisting}

In similar fashion a group construct, \textit{SensorAggregate}, is defined for sensors. The following section discusses the implementation of virtual sensors in SIXTH.

\subsection*{5.5.5 Sensor}

Section 2.3.1 defined a sensor as a device which captures observations of phenomenon such as temperature, humidity, light level, or pressure. A broader definition was also acknowledged and welcomed; in this wider vision a sensor is anything which can produce data. This section discusses the implementation of a virtual sensor abstraction to represent any sensor and function as an accessor for observational data.

Section 5.5.4 stated that a sensor node was a sensor provider, formalized as \textit{IProvider<ISensor>}. As such a sensor node acts a container for sensors. Section 5.5.2 and Section 5.5.4 described the implementations of adaptors and sensor nodes as being composed of several interface specifications.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{sensordiagram.png}
\caption{UML Diagram for SIXTH Sensor Interface}
\end{figure}

As in the realization of other core elements the sensor interface is the combination of several interfaces. Many of these interfaces are common to the implementation of the network adaptors, discussed in Section 5.5.2 and the virtual sensor node seen previously in Section 5.5.4. Figure 5.19 establishes that the same is true for the sensor interface given as \textit{ISensor}. In SIXTH a sensor is-a:

\begin{itemize}
  \item \textit{IDeviceMonitor} (Section 5.4.4)
  \item \textit{ISensorDescription}
  \item \textit{IDataProvider} (see Paragraph 5.3.1)
  \item \textit{ITaskable} (see Section 5.4.1)
\end{itemize}

\textit{ISensor} combines these interfaces to specify interaction with a sensing device. The \textit{Sensor} class provides the default implementation of the \textit{ISensor} interface. Instances of this class are used by the default \textit{SensorNode} implementation, discussed in Section 5.5.4, to create virtual sensors when new sensor data is generated. Listing 5.9 provides the program code invoked when new sensor data is added to the virtual sensor. The receipt of sensor data causes the sensor to disable the time-out detection task and reset the timer. If no data is received for \textit{samplingFrequency + threshold} the timer task will execute and notify the sensor node of possible time-out.
Listing 5.9: Adding Data to a Sensor

```java
@override
public synchronized void addData(ISensorData data) {
    sensorDataList.add(data);
    setLastReply(data.getTimestamp());
    if (!isTimeout) {
        timeoutTracker.cancel();
        timeoutTracker = new Timer();
        timeoutTracker.schedule(new Task(this),
            (long) (samplingFrequency.getFreqMS() + timeoutThreshold));
    }
}
```

5.5.6 Sensor Data

Section 4.4.3 defined sensor data as a representation of one or more observations made by a sensor. Observations are specified by the `ISensorData` interface. Figure 5.20 shows the UML class diagram for `ISensorData` drawing focus to its attributes and operations.

![UML Diagram of ISensorData](image)

The `SensorData` class provides the default implementation of this interface. Listing 5.10 shows one of the constructors of the sensor data class. An `AdaptorDescription` is required to fully specify the associated adaptor. The identifier of the sensor node is given as a long. The sensing modality is encapsulated in a modality class. The observations are supplied in a map. The final two arguments specify the sample type, and the time-stamp of the data in UNIX time. The `ISensorData` interface specifies methods to interact with this information.

Listing 5.10: SensorData Constructor

```java
public SensorData(IAdaptorDescription adaptorDescription, long id, Modality modality, Map<ComponentModality, String> values, String type, long time) {
    ...
}
```

The provenance of sensor data can be determined using public-key cryptography (Nechvatal, 1991) via the `verify(PublicKey key)` method. The public key of the assumed producer, the network adaptor, can
be used to verify that the data originated from it. Before the sensor is passed to the data broker for dissemination it is signed using the private key of the network adaptor.

Listing 5.11 provides the program code for the `SensorReading` constructor from the sensor middleware Mosen (Bakhshi et al., 2013), which was previously discussed in Section 2.5.9. The approach taken here differs from SIXTH in a number of ways:

- *SensorReading* does not implement an interface, as such any desired extension to the class requires sub-classing. This is known to reduce flexibility in development (Canning et al., 1989).
- In *SensorReading*, all sensor data must be numerical as evidenced by the values parameter of type double. The SIXTH approach is more flexible, and consequently permissive of diverse information streams.
- The observation type is recorded as an integer. In SIXTH this information is represented by a *Modality* which can contain extra contextual information such as the structure of the observations constituent elements and an external URL holding an ontological description of the modality.
- *SensorReading* does not capture the *type of observation* which was made. In SIXTH this concept refers to the circumstances under which the sensing was done i.e. periodic, request etc. It is felt that capturing such information is compelling.
- A *SensorReading* does not know to what sensor it belongs, only the observation type i.e where 1 is temperature. This is acceptable in a single node environment but difficult to justify when sharing data in a network.
- *SensorReading* features a single constructor, this is certainly clean and easy to utilize. It could be argued that the current SIXTH model features too rich a set of constructors which might pose initial confusion to the programmer. However, it is difficult to sacrifice any of the current parameters as they all provide necessary information.
- There can be no defence for the flagrant abuse of camel case attribute conventions (Høst et al., 2009).

### Listing 5.11: Mosen SensorReading Constructor

```java
1 public SensorReading(int SensorType, long TimeStamp, double ... values) {
2         this.SensorType = SensorType;
3         this.TimeStamp = TimeStamp;
4         Readings = new ArrayList<Double>();
5         for (int i=0; i<values.length; i++) {
6             Readings.add(new Double(values[i]));
7         }
8     }
```

https://github.com/nishant-jain/Mosen/blob/master/trunk/Android/src/sensorLayer/SensorReading.java

5.5.7 SIXTH Deployment

In Section 4.5.1 a SIXTH deployment was identified as a representation of all resources that are directly connected to the host machine. In this section, the implementation of this component is described.

Figure 5.21 illustrates the UML class diagram of the *ISIXTHDeployment* interface. Methods are shown for the registration and removal of adaptors and services. The *ISIXTHDeployment* provides its descriptor and a set of provider methods to access the adaptors contained within the deployment. The management of deployments is carried out by the Discovery service which is the subject of Section 5.6.1.
5.6 Application Services

This section describes the application facing components of SIXTH which facilitate the control of sensor networks and the interception of sensory observations. The design of these aspects was described in Section 4.5 and Section 4.6.

5.6.1 Discovery

The discovery service provisions access to all SIXTH deployments. A SIXTH deployment must register itself with the discovery service. From the UML class diagram shown in Figure 5.22 it can be seen that the discovery services provisions access to several other services, such as the data, sensor, and node brokers and the tasking service. Discovery also grants access to a set of providers that operate across all SIXTH deployments.

5.6.2 Receivers

In Section 5.3.4 it was shown that each subscriber implemented the \( IQueryRecevier<R,U> \) interface, this is the specification for one class of receiver within SIXTH. Figure 5.23 provides the UML class diagram for this interface and its ancestry. \( IQueryRecevier<R,U> \) defines only two methods each to be utilized by a broker to forward information to the receiver. Each subscriber can hold multiple subscriptions. This is why the \( IGenericQuery<T> \) is included in callbacks to make clear which criteria has been satisfied.
For example an \texttt{IQueryReceiver<ISensorNode,ISensorNodeDescription>} receiver interacting with the associated broker, when a new sensor node is registered the broker will determine whether to pass the taskable \texttt{ISensorNode} instance of the descriptor \texttt{ISensorNodeDescription} to the client.

Another class of receivers is defined, these are implicit entities which do not register with a broker or specify any queries. The appropriate broker disseminates all information to them. We take as an example the \texttt{IDataReceiver} which also extends from the \texttt{ICredentialedReceiver} interface and is given all public sensor data.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{uml.png}
\caption{UML Class Diagram for the IQueryReceiver}
\end{figure}

5.6.3 Services

SIXTH defines a set of services which implement the \texttt{IService} interface. \texttt{IService} is marker interface which extends for the \texttt{ICredentialed} and \texttt{ITaskable} interfaces.

A service is defined to serialize sensor nodes and other resources in XML and JSON formats. This service is utilized in the RESTful SIXTH components specified in Section 5.7.1. Resources can be easily de-serialized back into Java objects. XML serialisation is provided via the XStream library\textsuperscript{5}. JSON is performed through the Google GSON library\textsuperscript{6}.

The SIXTH database service receives all generated sensor data by implementing the data receiver interface. Received data is then inserted into a database. This functionality has been implemented to store data in a SQLite\textsuperscript{7} database. An adaptor can be defined to interface with data stored in a database.

Other services are defined to monitor Bluetooth and the serial ports for devices and inform interested network adaptors through tasking messages.

5.7 SIXTH Interaction

This section describes two means of interaction which have been implemented for SIXTH. The section begins by describing a RESTful web services for communicating SIXTH peers and clients.

\textsuperscript{5}http://xstream.codehaus.org/
\textsuperscript{6}https://code.google.com/p/google-gson/
\textsuperscript{7}http://www.sqlite.org/
Listing 5.12: RESTful URL Schema for the SIXTH Server

```java
public Restlet createInboundRoot() {
   // Create a router
   final Router router = new Router(getContext());
   router.attach("/deployments", SIXTHDeploymentsResource.class);
   router.attach("/", SIXTHDeploymentResource.class);
   router.attach("/adaptors", AdaptorsResource.class);
   router.attach("/adaptor/{adaptorid}/sensornode/{sensornodeid}/sensor/{sensorid}", SensorResource.class);
   router.attach("/adaptor/{adaptorid}/sensornode/{sensornodeid}/sensors", SensorsResource.class);
   router.attach("/push", SIXTHServerResource.class);
   router.attach("/task", TaskResource.class);
   return router;
}
```

5.7.1 RESTful SIXTH Interaction

Representational state transfer (REST) is an architectural style for specifying the connection between applications. REST was introduced in (Fielding, 2000) and expanded upon within (Fielding and Taylor, 2002). REST is popular as a paradigm for connecting systems together, and in the conclusion of this thesis Fielding remarks that “The modern Web is one instance of a REST-style architecture”. The software architecture of REST defines three units; components, connectors, and data. A component is a piece of software that provides a transformation of data. A connector is an intermediary that facilitates cooperation and coordination among components. Data is some piece of information held in a component and transported via connectors. Among the constraints imposed by the RESTful methodology there is to be a clear separation of client and server, and communication between those components should be stateless. In a typical web-based REST system a Uniform Resource Identifier (URI) (Berners-Lee et al., 2005) is given to each resource.

SIXTH implements a P2P interaction between SIXTH deployments using REST. This functionality is underpinned by the RESTlet platform (Louvel et al., 2009) which enables the creation of custom web APIs from Java code. A set of URIs are defined for both the client and server, though a SIXTH deployment can be both.

**SIXTH REST Server**

Listing 5.12 shows the definition of the SIXTH servers RESTful API, URIs are defined to access the known deployments, adaptors, nodes and sensors. The final two URIs are invoked to POST client connection information and tasking messages. The server streams information such as sensor data to connected clients. Data is served in a JSON (Crockford, 2006) representation that is serialized into Java objects by GSON.

**SIXTH REST Client**

A SIXTH client becomes aware of a server through a decoupled discovery component and issues a POST to the /push URIs and resource descriptors are forwarded to it. The RemoteSIXTHDeployment and RemoteSensorNetworkAdaptor classes are defined to represent the remote resources as if they were local. 105
A client may also function as a server. During registration the client will POST its URL, port and a boolean signifying if it is also a server. A class implementing the IServerProvider interface informs a client extending the IClient interface of new server instances. This service detection can be done through any means from a simple CSV list to a multicast DNS system such as JmDNS. SIXTH uses JmDNS (Hoff, 2009) for automatic service discovery. JmDNS facilitates the registration and discovery of services, and provides the starting point of communication. Functionality has been developed, as a plug and play OSGi bundle, to allow remote sensor reprogramming and data access. This functionality is provided within a wireless network via JmDNS.

5.7.2 SIXTH Graphical User Interface Suite

As an extension to SIXTH a suite of Graphical User Interface (GUI) functionality has been developed which can be used to easily develop SIXTH applications. This package includes a mapping component, dynamic widgets for adaptor configuration, data presentation views, data querying views, and a sensor network view. A prototype sensor control dashboard application which is formed from these components is utilized in the Ubiquitous Mapping application discussed in Section 7.3.4. The following sections provide a description the components of the suite. Figure 5.24 shows the Sensor Data Graph that graphs generic numerical sensor data generating a chart for each modality of all sensor nodes. Statistical information such as highest, lowest, average values and standard deviation are also displayed. The underlying graphing framework is provided by the SWTChart library. SWTChart was chosen over other libraries such as JChart2D, SWT XYGraph and JFreeChart due to its simple programming interface and flexible interface configuration options. The sensor data graph registers a query with the middleware to select all numerical data. Figure 5.25 shows a table based view of sensor data; in this view columns are dynamically injected for new modalities.

Figure 5.24: Line Graph of X-Axis Acceleration Sensor Samples

Figure 5.25: Table-based View of Sensor Data

http://jchart2d.sourceforge.net/
https://code.google.com/p/swt-xy-graph/
http://www.jfree.org/jfreechart/
Building upon a SWT wrapper\footnote{http://mappanel.sourceforge.net/swt/} for the Open Street Map project (Haklay et al., 2008), the Data Mapping component, illustrated in Figure 5.26, allows for the display of sensor data on a map. The data mapper registers a query with the data broker; this query selects only data includes a latitude and longitude.

![Map-based view of Sensor Locations](image)

**Figure 5.26:** Map-based view of Sensor Locations

Figure 5.27a shows the reconfiguration view which allows for the tasking of sensor nodes. The UI provisions the user to turn each of the nodes sensors on or off, or alter its sampling frequency. The UI is dynamically generated by specification information provided by each network adaptor. When the user clicks task, their requests are translated into one or more tasking messages, dispatched to the tasking service and delivered to the appropriate resource.

![Network list view](image)

**(a) Network list view**

![Adaptor Configuration](image)

**(b) Reconfiguration panel for a single sensor**

**Figure 5.27:** Network Organization View and Sensor Node Reconfiguration Panel

Figure 5.27b shows the reconfiguration view which allows for the tasking of sensor nodes. The UI provisions the user to turn each of the nodes sensors on or off, or alter its sampling frequency. The UI is dynamically generated by specification information provided by each network adaptor. When the user clicks task, their requests are translated into one or more tasking messages, dispatched to the tasking service and delivered to the appropriate resource.

![Network view](image)

**(a) Network list view**

![Logical network view](image)

**(b) Logical network view which depicts the network with each sensor node connected to its one hop neighbours.**

**Figure 5.27a** shows the network view which provides a hierarchical listing of SIXTH resources. The hierarchy is as follows; deployment → network adaptor → sensor nodes → sensors. This element is constructed as an implementation of an ITreeContentProvider for traversing our resources and using a StyledCellLabelProvider to provide styled text for each cell. **Figure 5.28** illustrates the Logical network view which depicts the network with each sensor node connected to its one hop neighbours. This is facilitated by the listing of neighbours held each virtual sensor node as discussed in Section 5.5.4.
5.8 Conclusions

This chapter has provided a detailed overview of the implementation of the SIXTH middleware which is the concrete delivery of the primary objective given in Section 1.3.1.

Within SIXTH cyber-sensing is incorporated as a first class citizen alongside physical WSNs access. This allows for a unified data collection and sensor tasking model such features are not present in other middleware offerings. The efficacy of such an approach in service to the pervasive computing vision has been previously identified in the literature (Rosi et al., 2011). It is agreed that such a combination allows for richer context-awareness in applications and enables a higher level of certainty when identifying events.

SIXTH provides support for heterogeneous sensor network platforms both cyber and physical and can facilitate connection with multiple networks simultaneously. These features are absent from earlier middleware efforts which were tied to specific WSN platforms such as Mate (Levis and Culler, 2002), In-Motes (Georgoulas et al., 2006), and TinyDB (Madden, Hellerstein, et al., 2002). Extensibility and openness are key to the SIXTH ethos allowing for developers to enhance the provided middleware through incorporation of additional sensor network adaptors and system services.

SIXTH provides intelligent reasoning through the use of software agents at the gateway within the WSNs. This functionality is provided through integration with AOP frameworks including AgentFactory and ASTRA. The full BDI models of agency supported therein separate these agents from examples in other offerings which provided a weaker, more reactive, agent as seen in (Fok et al., 2005a) and Impala (Liu et al., 2003).

In stark contrast to other middleware which target the sensor web SIXTH has a focus on providing actuation within the network rather than simply acting as data receiver. Through a generic interaction component and graduated support layers we provide this functionality across all supported sensor networks. The goal of this is to allow the developer to (re)task sensors absent concern for the specific network to which it is attached. Middleware platforms such as LSM are not concerned with physically interfacing with sensor platforms and networks, rather sensors published to the web are treated as physical devices. In this way the goals of SIXTH are complimentary to this kind of platform as SIXTH can deliver diverse streams of information for these frameworks; in the case of LSM the streams would then be augmented semantically. Section 7.4.1 describes the use of SIXTH in consort with another semantic enrichment framework.

The OSGi framework upon which SIXTH is based, provides the basis to support dynamic change the structure of the entire system during runtime, new adaptors may be added, services removed and so forth. This may be done to conserve resources or to load updated functionality into long running deployments. The SIXTH framework provides program code for network nodes; within platforms such as the Tyndall
iWSN this forms a crucial part of the ethos in delivering intelligence through AFME in-network agents. However, SIXTH is a higher level middleware than many earlier offerings in which the focus was entirely on the network internals; examples of this include MiLan (Heinzelman et al., 2004) or TinyLIME (Curino et al., 2005). SIXTH is in line with offerings such as GSN and WSNWare; the middleware operates with any network and nothing needs to be known except for the connection mechanism and message format.
6 | Adaptor Development Walk-through

6.1 Preface

Chapter 5 described the implementation of SIXTH thus rounding out the pure development chapters which begin with the design in Chapter 4. Herein, the SIXTH implementation is explored through the development of three network adaptors. The intention of this walk-through is to, at a reasonably low-level to, illustrate how to use SIXTH in a set of varied scenarios. How SIXTH is used, and what it brings to the development process, is a theme which echoes through the evaluation detailed in Chapter 7 and Chapter 8. This chapter provides an illustration of the achievement of several of the secondary objectives, given in Section 1.3.2 such as the accommodation of differing information streams, resource abstraction, and problem decomposition.

Section 6.2 launches the chapter by describing the development of a simple adaptor which produces synthesized sensor observations. The simplicity of this example enables us to focus on the concepts and development process in the absence of the complexities of heterogeneous device connection. This section can be conceptualized as our take on Hello, Sensors. subsequently, Section 6.3 builds on the development process shown in Section 6.2 and describes the key elements of a network adaptor for the Sun SPOT sensor platform. This example serves to illustrate the key concepts of wrappers & translators (see Section 4.4.7 and Section 4.4.7) in a concrete manner. Finally, Section 6.4 details the creation of a third adaptor. In this instance the adaptor is defined for a web-based information stream: the social media driven news & current affairs platform Reddit. This section illustrates how to develop a SIXTH adaptor for non-WSN resources, thus illustrating the uniformity of representation and the ease which which diverse information sources can be integrated.

6.2 Building A Simple Adaptor

For this first case study, in the interests of simplicity, an adaptor which produces synthesized sensor observations is implemented. This adaptor is referred to as the Dummy adaptor.

Listing 6.1 presents the skeleton code for the adaptor which extends from the AbstractSensorNetworkAdaptor (see Section 5.5.2). This superclass provides virtual sensor creation and information dissemination functionality. In Listing 6.1, it can be seen that the task(ITaskingMessage) method is a stub, to be filled later. The constructor passes the network name to the superclass.

Listing 6.2 depicts the complete program code for the DummySensor. This class extends the default implementation of the Sensor class (see Section 5.5.5). The definition of the DummySensor constructor begins on line 6. The constructor takes in two parameters; a Modality and a SensorNode. The SensorNode is the container for the virtual sensor and the Modality object is a specification of what is sensed.

This refers to the standard first program developed when learning a new programming language which outputs “Hello, world!” to the standard output.
These two parameters are passed to the superclass. On line 8 of Listing 6.2 the DummySensor invokes task(ITaskingMessage) to set its own sampling frequency. The ITaskingMessage is created via a factory method from TaskingMessageFactory (see Section 5.3.3). The DummySensor implementation of task(ITaskingMessage) begins on line 30. Therein, the CommandType of the ITaskingMessage is examined and when appropriate the sensors sampling frequency is modified. Otherwise the request is rejected.

DummySensor implements Runnable; an instance of Runnable can be run by a thread. This is a necessity as the lack of multiple inheritance restricts the developer from extending Thread. The run() method from Runnable is implemented (line 15 of Listing 6.2). Therein, a while loop is run which will execute until the thread is interrupted. In the loop body a random value integer is generated, subsequently, produceData(String, String) is invoked to produce valid ISensorData from the observation and disseminate it via the Data Broker. Following production the thread sleeps for a period defined by the sampling frequency.

Listing 6.3 depicts createNodes() from DummyAdaptor. This method creates dummy sensors and nodes. Therein an sensory modalities, number of sensor nodes, and the sampling frequency are defined. In yhe outer loop (line 9) a ISensorNode is created via sensorNodeLookup(int) which determines if a matching instance exists and instantiates if absent. The inner loop (line 11) creates a sensor for each modality. Each sensor is added to its node, and its sampling frequency is set. Each sensor node is assigned a DummyMapSensor which produces faked latitude and longitude pairs. At this juncture a functional dummy adaptor has been created.

At this point it is prudent to implement task(ITaskingMessage) for DummyAdaptor; this is depicted in Listing 6.4. Therein, the tasking message is inspected for the recipients identifier. The recipient is determined and if it exists the message is passed. Thus concludes the development of a simple dummy adaptor for SIXTH. In forthcoming sections the development of network adaptors for physical and cyber platforms are detailed.

6.3 Connecting to a Physical Sensor Platform

The Sun SPOT is a popular sensor platform developed by Sun Micro-systems. Sun SPOT applications are written in Java and executed by the Squawk JVM. This section provides a discussion of the key components of the network adaptor which interfaces with Sun SPOT nodes; providing functionality to alter their behaviour and interpret their observations.
public class DummySensor extends Sensor implements Runnable {

    private static final long serialVersionUID = 1497670152920825390L;
    private Thread thread;

    public DummySensor(Modality modality , ISensorNode sensorNode) {
        super(modality , sensorNode);
        this . task(TaskingMessageFactory .
            setSamplingFrequency(getModality () . getType() ,
                new Frequency(500.0 , false ))) ;
        thread = new Thread(this);
        thread . start () ;
    }

    @Override
    public void run() {
        while (! Thread. currentThread () . isInterrupted () ) {
            int val = (int) (Math.random() ∗ 20);
            produceData(val + "", ISensorData .TYPE_PERIODIC);
            try {
                Thread. sleep ((long) (getSamplingFrequency () . getFreqMS() ));
            } catch (InterruptedException e) {
                e . printStackTrace () ;
            }
        }
    }

    @Override
    public boolean task(ITaskingMessage<?> message) {
        if (message .getCommandType() == CommandType.SETFREQ) {
            samplingFrequency = (Frequency) message . getTaskingObject () ;
            return true ;
        }
        return false ;
    }
}

private void createNodes() {
    dataTypes = new String [] {
        Modalities.TEMPERATURE, Modalities.HUMIDITY,
        Modalities.ACCELERATION_ABS};
    numSensorNodes = 4;
    speed = 400;

    for (int nodeID = 0; nodeID < numSensorNodes; nodeID++) {
        ISensorNode node = sensorNodeLookup(nodeID);
        for (int j = 0; j < dataTypes . length ; j++) {
            DummySensor sensor = new DummySensor(new Modality(dataTypes[j]) , node);
            node . addSensor(sensor);
            sensor . task(TaskingMessageFactory .
                setSamplingFrequency(dataTypes[j] , new Frequency(speed , false ))) ;
        }
        node . addSensor(new DummyMapSensor(new Modality("location") , node));
    }
}

private void createNodes() {
    ISensorNode node = sensorNodeLookup(100);
    node . addSensor(new SystemPropertiesSensor(node));
}
## 6.3.1 Message Translation

Listing 6.5 depicts program code utilizing an `IMessageTranslator<E>` (see Section 4.4.7) to transform its input into units of `ISensorData`. If the input cannot be parsed a `SensorMessageTranslationException` is thrown. The `ISensorData` instance are passed to `addData(ISensorData)` which handles sensor creation and message dissemination.

Listing 6.6 provides the code for `translate(Datagram)` method from `SunspotMessageTranslator`; this method was invoked in Listing 6.5. Conditional logic (line 12) is applied to differentiate between single and multiple value observations. If the modality is `MessageConstants.ACC_ALL` three values are extracted for x, y, and z acceleration; otherwise a single value is read. If no `ISensorData` can be produced a `SensorMessageTranslationException` exception is thrown. This exception is caught in Listing 6.5.

## 6.4 Interfacing with the Social Web

In this section an adaptor is implemented for a web platform; Reddit\(^2\) Reddit is a user-driven content sharing and discussion platform. All content on Reddit is supplied by its users who post links to external websites that they find relevant to some subcomponent of the community. Reddit is subdivided, by topic, into subreddits. Each subreddit is based around some interest such as comic books (/r/comicbooks) or computer programming (/r/programming).

Section 6.4.1-6.4.2 describe the adaptor and its cyber-sensors. When the implementation is complete Section 6.4.3 discusses a receiver which filters the sensor data for images to download. The Reddit adaptor utilizes the JReddit\(^3\) API wrapper.

---

1. `www.reddit.com`  
2. `https://github.com/karan/jReddit`
@Override
public List<ISensorData> translate(Datagram datagram) throws SensorMessageTranslationException {
    List<ISensorData> list = new Vector<ISensorData>();
    ISensorData data = null;
    try {
        byte type;
        double value;
        long time;
        String typeString = null;
        int id = datagram.readInt();
        byte modality = datagram.readByte();
        if (modality == MessageConstants.ACC_ALL) {
            type = datagram.readByte();
            Map<ComponentModality, String> values = new HashMap<ComponentModality, String>();
            values.put(Modalities.ACCELERATION_X, "" + datagram.readDouble());
            values.put(Modalities.ACCELERATION_Y, "" + datagram.readDouble());
            values.put(Modalities.ACCELERATION_Z, "" + datagram.readDouble());
            time = datagram.readLong();
            typeString = MessageConstants.replyTypeMap.get(type);
            if (typeString != null) {
                data = new SensorData(adaptor.getDescription(), id, Modalities.ACCELERATION_ALL, values, typeString, time);
            }
        } else {
            // omitted for brevity
        }
        if (data == null) {
            throw new SensorMessageTranslationException();
        }
    } catch (IOException e) {
        e.printStackTrace();
        list.add(data);
        return list;
    }
}
6.4.1 User Sensor

This section implements a user sensor for the retrieval of a Redditor's private messages. [Listing 6.7] depicts the complete UserSensor class. It is evident that UserSensor shares commonalities with DummySensor, as implemented in Section 6.2. Both classes extend Sensor and implement Runnable. Within run() (line 17), the ten most recent messages are retrieved and passed to generateMessageData(Message) for transformation and dissemination. The loop and the usage of the sampling frequency are familiar from DummySensor. In the loop all unread messages are fetched and passed to generateMessageData(Message). This method (line 52) extracts the pertinent information from the Message and creates ISensorData.

6.4.2 Subreddit Sensor

This section describes the implementation of a subreddit sensor which monitors new content submissions for the given subreddit. Listing 6.8 presents the complete SubredditSensor class. Much of the structure SubredditSensor is consistent with UserSensor from Section 6.4.1, as such discussion focuses on the unique. In Listing 6.8 (line 17) a more complex task(ITaskingMessage) a more complex task(ITaskingMessage) is given. Therein the sensor interrupts the thread when the Frequency is 0. If the sensor is inactive, i.e., no thread exists, a new thread is started. In run() (line 50) a check determines if the URL is a JPEG image. If so, encode() (line 66) stores the image in the values map.

6.4.3 Sensor Data Receiver

In Section 6.4.2 images retrieved from Reddit submissions were stored in ISensorData which is disseminated, via the Data Broker, to registered receivers. This section describes the implementation of such a receiver. The program code of the receiver is given in Listing 6.9. Therein (line 3) an IGenericQuery<ISensorData> (see Section 5.3.2) is defined to specify the properties of interest. The query only accepts sensor data containing an image component modality. The query is registered with the data broker (line 10) which then streams all matching data to receive(ISensorData). The image payload is passed to decodeToImage(String, String) which saves the image to the hard disk under its reddit title.

6.5 Conclusions

This chapter has provided a discussion of three network adaptors for the SIXTH platform. In so doing it has provided an insight into the development process involved in utilizing and extending the sensory capabilities of SIXTH.

The adaptors were developed for synthesized dummy data (see Section 6.2), a physical WSN platform (see Section 6.3), and a cyber-space information sharing platform (see Section 6.4). The User Guide which is the subject of Appendix A provides a more detailed implementation guidebook for those so inclined.

This chapter has provided the final part of the development phase of this thesis as such Chapter 7 begins the evaluation of SIXTH through its usage in set of case studies arising from research outputs. Therein, in the same vein as this chapter, many insights are given into the flexibility of the SIXTH framework.
package ie.ucd.sixth.adaptor.reddit;

public class UserSensor extends Sensor implements Runnable {

public UserSensor(Modality modality, ISensorNode sensorNode, User user) {
    super(modality, sensorNode);
    this.user = user;
    new Thread(this).start();
}

private static final long serialVersionUID = -8832475750082931756L;

private User user;

@override
public void run() {
  Messages message = new Messages();
  List<Message> messages = message.inbox(user, 10);
  for (Message message2 : messages) {
    System.out.println(message2);
    generateMessageData(message2);
  }

  while (!Thread.currentThread().isInterrupted()) {
    try {
      user.connect();
      message = new Messages();
      List<Message> unread = messageunread(user);
      for (Message message2 : unread) {
        generateMessageData(message2);
      }
    } catch (Exception e) {
      e.printStackTrace();
    } try {
      Thread.sleep((long) getSamplingFrequency().getFreqMS());
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

private void generateMessageData(Message message2) {
  Map<ComponentModality, String> values = new HashMap<ComponentModality, String>();
  values.put(new ComponentModality("author"), message2.getAuthor());
  values.put(new ComponentModality("body"), message2.getBody());
  values.put(new ComponentModality("subject"), message2.getSubject());
  produceData(values, ISensorData.TYPE_PERIODIC);
}
package ie.ucd.sixth.adaptor.reddit;

public class SubredditSensor extends Sensor implements Runnable {

    private static final long serialVersionUID = 248816719443801790L;
    private String subreddit;
    private User user;
    private Thread thread;

    public SubredditSensor(Modality modality, RedditNode sensorNode, String subreddit, User user) {
        super(modality, sensorNode);
        this.subreddit = subreddit;
        this.user = user;
    }

    @Override
    public boolean task(ITaskingMessage<?> message) {
        if (message.getCommandType() == CommandType.SETFREQ) {
            System.out.println("received tasking command " + getModality() + " " + message);
            samplingFrequency = (Frequency) message.getTaskingObject();
            if (samplingFrequency.getFreqMS() <= 0.0) {
                thread.interrupt();
                thread = null;
            } else if (thread == null) {
                thread = new Thread(this);
                thread.start();
            }
            return true;
        }
        return false;
    }

    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            try {
                user.connect();
                try {
                    for (Submission submission : Submissions.getSubmissions(subreddit, Population.NEW, Page.FRONTPAGE, user)) {
                        Map<ComponentModality, String> values = new HashMap<
                                ComponentModality, String>;
                        values.put(new ComponentModality("title"), submission.getTitle().getString());
                        // common code omitted for brevity
                        if (submission.getHostURL().endsWith(".jpg")) {
                            encode(values, submission.getHostURL());
                        }
                        produceData(values, ISensorData.TYPE_PERIODIC);
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    private void encode(Map<ComponentModality, String> values, String url) {
        // omitted for brevity
    }
}
public RedditImageReceiver(BundleContext context) {
    id = UUID.randomUUID();
    IGenericQuery<ISensorData> data = new AbstractGenericQuery<ISensorData>(){
        @Override
        public boolean match(ISensorData item) {
            return item.hasA(new ComponentModality("image"));
        }
    };
    SIXTHFramework.getDiscovery(getCredentials()).getDataBroker().registerInterest(data, this);
}

@Override
public void receive(ISensorData data, String info, IGenericQuery<ISensorData> query) {
    decodeToImage(data.getValue(new ComponentModality("image")), data.getValue(new ComponentModality("title")));
}
Part IV

Evaluation
7 | Case Studies

7.1 Preface

The primary objective of this thesis (see Section 1.3.1) is to provide a middleware system which is generic, permissive, and conducive to application development. For the assessment of this goal Section 1.3.2 established the need to assess the middleware through a set of real-world case studies. This chapter presents these case studies, with a view to demonstrating, through their variety, that SIXTH achieves this primary objective.

SIXTH has been utilized within seven research projects, consequently the majority of the case studies emerge from peer-reviewed publications. In this chapter, a concise overview of each project is given, in doing so specifying the role of SIXTH with an emphasis on software development support. The case studies are diverse in many dimensions; some are SIXTH extensions while others are sensor-driven applications or ports of the system. The following is a high-level overview of the remainder of the chapter.

Section 7.3 describes a SIXTH extension providing an additional support layer for interfacing with cyber-sensing resources. This serves to illustrate the extensibility, ease of use, domain abstractions, and homogenisation support present in SIXTH. Section 7.4 describes the amalgamation of SIXTH with SNoMAC and CArtArgO. The former is a semantic reasoning engine for diverse sensor streams. The latter is an Agent-Oriented Programming (AOP) environment interface which connects agent platforms to sensory streams. This showcases the capability to utilize SIXTH in pre-existing problem domains and to support intelligent reasoning (as per Section 3.7). Section 7.5 describes research applications which harness SIXTH as an enabling framework. These systems target energy management in the home and the correct disposal of waste in Ireland. This serves to illustrate application support, usability, and reuse within SIXTH.

Section 7.6 discusses the semantic extensions to the SIXTH core which are performed through the replacement of core components such as the Data Broker. This shows that SIXTH is open for modification, and extension. Section 7.7 describes AndroSIXTH; an extension, and porting, of the SIXTH middleware to the Android platform. This demonstrates the flexibility of SIXTH as the entire architecture of the gateway middleware is fitted seamlessly into an in-network scenario.

7.2 Satisfaction of Desiderata

Chapter 3 explicitly identified a set of desirable middleware features which underpinned the design of SIXTH (see Chapter 4). Herein Table 7.1 relates the case studies, subsequently examined, back to this desiderata. In this way the fulfilment of these goals is illustrated. In many cases each case study provides illumination on the achievement of many desideratum, as such Table 7.1 indicates the case study which exemplifies the achievement of each desideratum.

\(^1\)In the computer science sense this means the movement of code from one platform to another
7.3 Cyber Sensing With SIXTH

This section describes the cyber sensing extensions added to SIXTH. This work was done as part of a doctoral thesis by Dr. Olga Murdoch (Murdoch, 2014). The following is an examination of the work which most clearly relates back to the subject of this Thesis. Section 7.3.1 describes the use of pipeline programming. Section 7.3.2 discusses a set of tools for aiding in the development of cyber adaptors and finally Section 7.3.3 describes a set of implemented network adaptors. Having established a frame of reference, beginning in Section 7.3.4 two case studies are examined which utilize this work in consort with the SIXTH core.

7.3.1 Application of Pipeline Programming

The SIXTH cyber extension applies pipeline programming for the modification, filtering, amalgamation, and extraction of Sensor Data. Pipeline programming refers to the chaining of processing components wherein the output, if generated, of the Nth processor is input to N + 1th. Five broad classes of pipes were identified:

- **Filter**: Prevents an unsuitable piece of sensor data from being passed to the next pipe. This is analogous to the Query mechanism described in Section 4.4.5 and has also been implemented using a wrapper for the query interface.

- **Transformer**: Converts sensor data components into different formats. For example it may be desirable to convert a named location into a latitude longitude pair.

- **Fuser**: Augments sensor data with derived knowledge. For example a Tweet can be augmented with the Klout score of the user. Nothing in the data is lost, or modified. This is distinct from the transformer.

- **Learner**: Gains knowledge from the sensor data which passes through it e.g. observe fluctuations in temperature from a single data source as filtered by a query.

- **Aggregate**: Represents a pipe composition; The use of Aggregates in SIXTH was described in Section 4.3.4. An aggregate pipe can modify the execution flow of its sub-pipes based on their outputs.

7.3.2 Utilities

The SIXTH core was augmented with numerous utility functionality which reduces friction whilst working with web resources. The SIXTH GeoManager provides a synthesis of the Yahoo Geoplanet and place-finder API’s and bespoke code to facilitate conversion between place names, WOEIDs, coordinates, and

---

Table 7.1: SIXTH Case Studies Satisfaction of Desiderata

<table>
<thead>
<tr>
<th>Desideratum</th>
<th>Case Studies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heterogeneity of Source</td>
<td>All Section 7.7</td>
</tr>
<tr>
<td>Abstraction &amp; Uniformity</td>
<td>Section 7.5.1</td>
</tr>
<tr>
<td>Extensibility &amp; Scalability</td>
<td>Section 7.5.1</td>
</tr>
<tr>
<td>Ease of Use</td>
<td>Section 7.3</td>
</tr>
<tr>
<td>Application Support</td>
<td>Section 7.5.1</td>
</tr>
<tr>
<td>Embedded Intelligence</td>
<td>Section 7.4.2</td>
</tr>
<tr>
<td>Runtime Reconfiguration</td>
<td>Section 7.5.1 &amp; Section 7.5.2</td>
</tr>
</tbody>
</table>

2Reputation score
The SIXTH web scraper, underpinned by HTMLUnit[^3], allows for easy information extraction from HTML tables through user specified templates. This allows the user to specify instructions such that large tables can be modelled through a minimal template (e.g., describe a single table row using instructions for the pattern with which each of the other rows should be generated). Code is provided to allow intelligent meta-data to be supplemented (e.g., using GeoManager to add geographical coordinates to the data scraped).

The SIXTH TimeManager manages the acquisition of time and conversion between formats as necessary for configuring sensors and/or supplementing sensor data.

### 7.3.3 Implemented Adaptors

Cyber adaptors have been developed for the API's or websites of three Irish electricity supply companies; Airtricity[^4], Bord Gáis[^5], and the Electricity Supply Board[^6]. This set of adaptors was utilized in the ABLE project described in Section 7.5.2. For the incorporation of environmental and weather information a suite of adaptors was developed for numerous web resources including Met Eireann[^7] rainfall data[^8], Ireland’s Weather[^9], proprietary data from the Smart Coasts project, and the Yahoo[^10], YR[^11] and Wunderground[^12] weather services.

Social Media platform Twitter[^13] was integrated into the framework. This adaptor facilitates the creation of several types of cyber-sensor for information retrieval on the basis of keyword (or hashtag), user, trends, or location.

Additionally support has been provided for the sensor devices connected to the WWW through Sensor Web Portals (as described in Section 2.5) such as Xively and ThingSpeak (see Section 2.5.10). This is demonstrative of the interconnection of multiple sensor data publishing frameworks.

The following sections describe two case studies from published work which utilized the SIXTH core and the cyber sensor extension.

### 7.3.4 Ubiquitous Mapping

(O’Hare et al., 2012) describes a ubiquitous mapping demonstrator for SIXTH. This system was built using the GUI development tool-kit discussed in Chapter 5. Therein cyber sensor observations from diverse sources are mapped in real-time. Sources include Xively, Twitter, Met Eireann, and Yahoo! Weather. This interface allows for the dynamic adaptation of sampling rates and user topical interest. These configuration elements are dynamically generated from sensor descriptors. Figure 7.1 depicts the user interface for this demo application.

[^4]: [http://www.airtricity.com](http://www.airtricity.com)
[^6]: [https://www.esb.ie/esbcustomersupply/residential/home/index.jsp](https://www.esb.ie/esbcustomersupply/residential/home/index.jsp)
[^7]: [http://www.meteireann.ie](http://www.meteireann.ie)
[^8]: Met Eireann website live rainfall imagery was scrapped by Dublin City University researchers who provided a JSON API.
[^10]: [http://weather.yahoo.com/](http://weather.yahoo.com/)
[^11]: [http://www.yr.no/](http://www.yr.no/)
[^13]: [http://twitter.com](http://twitter.com)
7.3.5 Tracking Wildfires in Tasmania

Cyber Sensor Networks underpinned by SIXTH have been used to track wildfires in Tasmania. This system is presented through the lens of cyber-journalism. A journalist who is geographically impaired from covering an event can collate sources of information in real-time and present their perspective on the unfolding events for their readers. Cyber sensors were employed to collect five categories of information: satellite imagery, local weather, news, fire service and social media via Twitter. The cyber sensors were configured to filter incoming data streams using a set of appropriate terms, for example, “fire”, “Tasmania” and “bush fire”. Figure 7.2 depicts a collage of all the information retrieved by the cyber sensors. Therein data gathered from Twitter users reporting on bush fires compliments the notification from the Tasmanian bush fire alert sensors and the weather data.

7.3.6 Discussion

The SIXTH Cyber Sensor extension highlights extensibility, portability, and ease of use. The aforementioned are three of the desideratum described in Chapter 3. Additionally this extension builds support for heterogeneity of source, provides a further layer of desirable abstraction, and makes use of the uniform interfaces of SIXTH. The implementation of this extension was able to utilize the vast majority of the code-base for example the AbstractCyberSensorAdaptor base class extends from AbstractSensingAdaptor as such reusing the common functionality provided. This highlights favourably the reuseableness of the generic SIXTH architecture.

Section 7.3.5 and Section 7.3.4 illustrate the diverse usage of the cyber-sensor extension and the underlying SIXTH core. It has been previously demonstrated (Section 6.4) that it is possible to develop cyber-resource connection adaptors in vanilla SIXTH. It may well prove prudent to more tightly fuse the contributions of this thesis with the cyber core with a view toward “A More Perfect Union” (Obama, 2008).

It is observed that working with cyber-resources poses some unique challenges. The adaptors have, at times, been pronounced victims of software atrophy\textsuperscript{14}. The code stops working as the external APIs, upon which the code relies, are modified, deprecated, updated, or deleted.

\textsuperscript{14}This is the application of the biological term, atrophy, to software.
7.4 System of Systems

7.4.1 SNoMAC

SNoMAC (Matheus et al., 2012) is an information management framework which lifts network specific data into a semantic representation that is grounded in the high-level NetCore ontology. The system permits the automatic discovery of new devices, the monitoring of device state and the invocation of device actions in a generic fashion that works across network types, including non-telecommunication networks such as social networks. Figure 7.3 illustrates the integration of SIXTH and SNoMAC via a network Adaptor; it can be observed that UPNP (UPNP, 1999) and TR069 devices are integrated via separate adaptor modules. Sensor Data and Node representations are lifted into the NetCore ontology. SIXTH interfaces with SNoMAC through a custom RESTful interface defined in line with the SNoMAC representational model. This approach forms a system-of-systems by joining SIXTH and SNoMAC.

Listing 7.1 showcases a SNoMAC compliment representation of a node in RDF via JSON-LD. The SIXTH REST interface performs translation from the internal virtual plain old Java object (POJO) representation of a sensor node into RDF when SnoMAC requests device information. The information conveyed in Listing 7.1 represents standard sensor attributes in large part; ncStatus refers to a model of sensor health based on a standard traffic light configuration (red, yellow, and green) and ncState is a hash of standard and device specific properties and their current values. These states were envisaged to include power-level, performance (e.g. drop-calls), or fault (e.g. connection-loss). States may be configurable or static.

---

16 This is a specialization of the SIXTH REST interface defined in Section 5.7.1
17 http://json-ld.org/
Listing 7.1: SNoMAC RDF /nodes/:nodeID

```json
{
  "@id": "n1",
  "@type": "upnp: mediaServer",
  "rdfs:label": "My Media Server",
  "ncStatus": "yellow",
  "ncState": {
    "upnp:active": true,
    "upnp:battery": "high"
  }
}
```

- **GET /nodes**  The URL returns a JSON-LD RDF representation of all known nodes; this representation conveys the node identifier, their RDF type as per the network specific ontology, and the current value of the nodes “status”.

- **PUT /nodes/push_changes?urlWebServer**  SnoMAC issues a request to the adaptor via this URL to specify a web server to which the adaptor should disseminate updates regarding sensor nodes. In the SIXTH adaptor this results in the spawning of a set of receivers which are pushed device updates to disseminate to the newly registered client.

- **PUT /nodes/stop_pushing?urlWebServer**  Stops the SIXTH receivers from publishing updates to the server specified in the URL parameter.

- **GET /nodes/:nodeID/states**  Returns the available state variables for the node specified (by nodeID) e.g. ["upnp:active","upnp:battery"].

- **GET /nodes/:nodeID/states/:stateID**  Returns the value and the time-stamp for the specified state.

- **PUT /nodes/:nodeID/states/:stateID?value**  Sets the specified state variable to the provided value; returns "success" (HTTP 200) or "failure" (HTTP 404).

- **PUT /nodes/:nodeID/states/:stateID/push_changes?webserver**  Assigns the adaptor to push changes which occur on the node to the web service at the specified URL.

---

### 7.4.2 Agent Oriented-Programming Bridge for Sensor-rich Environments

(Lillis et al., 2013) proposed a prototype system which linked the SIXTH middleware with an intelligent Agent-Oriented Programming (AOP) runtime platform thereby providing intelligent agents with an abstraction of a physical environment. The canonical definition of software agency (Wooldridge et al., 1995) considers the environment as an essential component of Multi Agent Systems (MAS); this has acted...
as an impetus for research involving the provision of suitable agent environment abstractions. The work was later extended within the WAIST project (Russell, 2014). WAIST is discussed in Section 7.5.1; the extension incorporated both generic SIXTH and WAIST specific components.

The agent platform utilized was ASTRA (Dhaon et al., 2014) an Agent Programming Language (APL) and runtime environment developed in University College Dublin in recent years. ASTRA is the successor to Agent Factory (Collier et al., 2003). The Common Artifacts for Agents Open Framework (CArtArgO) (Ricci et al., 2007) provides the model of the environment.

Figure 7.4 illustrates the interconnection between SIXTH and ASTRA as facilitated by CArtArgO. SIXTH components such as sensor nodes, observation streams, services, and adaptors are represented as artifacts. ASTRA agents are presented with information about the environment through the artifacts.

For the purposes of example it can be seen that the tasking artifact is invoked by an agent causing a request to be issued to the middleware via the artifact. The stream artifact is responsible for consuming the sensor data stream from the data broker and delivering this information to the agent layer. This system-of-systems is key to delivering upon the goal of providing support for embedded intelligence within SIXTH.

Figure 7.4: SIXTH-ASTRA Bridge (Lillis et al., 2013)

Listing 7.2 provides the program code, with the extraneous omitted, for the ASTRA sensor data receiver Artifact. The DataStreamArtifact extends the Artifact base class from CArtArgO and implements the IQueryDataReceiver interface from SIXTH to facilitate data capture. In the init() method set-up code is provided which registers a query for all data with the DataBroker. The receive(ISensorData, IQuery) method passes the data to an internal operation which makes the data available to the agents. The DataStreamArtifact is an implementation of the Stream Artifact shown in Figure 7.4.

7.4.3 Discussion

SNoMAC (Section 7.4.1) and the ASTRA-SIXTH Bridge (Section 7.4.2) demonstrate that SIXTH can be composed with other software platform in so doing forming a system-of-systems (SoS). Such a SoS is more capable than any of its constituent components.

In Section 3.7 Supporting Embedded Intelligence was identified as an element of the desiderata for SIXTH. Section 7.4.2 and (Lillis et al., 2013) demonstrate that this desideratum has been delivered upon.
### 7.5 Application Scenarios

#### 7.5.1 Sensor-based Waste Disposal Tracking

To demonstrate the benefits of developing applications using SIXTH, this section describes one such application: *Waste Augmentation and Integrated Shipment Tracking* (WAIST). The WAIST project (Russell et al., 2013) utilizes Wireless Sensor Network technology in the detection of illegal waste disposal by individuals and organizations involved in the supply chain of waste products, in particular while the materials are in transit. This project is funded by the Irish Environment Protection Agency (EPA) which highlights the importance, and prevalence, of this issue in an Irish context. A complement of sensor devices are employed to facilitate the goals of WAIST. GPS devices are used to track the route used by the disposal vehicle; this is conducted so as to determine if any unscheduled stops were made or if a suspicious route was used. Accelerometers were attached to waste containers to enable the monitoring of unusual isolated movement. Event detection and handling is performed by ASTRA agents which interface with SIXTH through a refined version of the systems bridge described in Section 7.4.2. WAIST is designed as three main software components: the in-situ component for deployment in the target vehicle, a server component for processing and data storage, and a visualisation utility. Each of these component are instances of SIXTH, which together combine to form the entire application.

**In-situ** This component is designed to manage a small network of sensors and forward relevant data to the central server. The In-situ component can be run on any sufficiently powerful computing device capable of supporting OSGi e.g. Linux plug computers or Android devices, both of which are suitable for the projects demands. This requires connectivity to transmit data to the server. Data is collected through the use of any WSN technology supported by SIXTH. The design of adaptors within SIXTH, and more specifically the translator within the respective WSN Adaptor, ensures that the application is agnostic as to the data source. Figure 7.5a shows the architecture of an example in-situ deployment. Therein, two differing types of sensors are utilised to gather data and as such each sensor type has its own adaptor. The data is then forwarded to central server, using the P2P mechanism detailed in Section 4.5.1 and a local database.

**Server** The heart of WAIST is the server. It is the destination point for all data sensed by the in-situ deployments and herein the data is analysed. Figure 7.5b shows the simplified architecture of the server:
data is received and disseminated throughout the middleware where a number of actions may occur. All received data is stored within a database on the server to maximise the admissibility of the data within a court of law. Data is also analysed in a number of ways to generate events within the system. Simple events such as sharp increases in light level and lack of motion by a target are generated within the Events bundle.

More complex events are generated within separate bundles, for example motion data captured through the use of triaxial accelerometers is used to classify the state of the waste container (motionless, in transit, falling, tilting, dragging, lifting, carrying or rolling). Feature extraction techniques and classifiers are used to increase the accuracy of the classifications. Events are generated for each individual classification as well as more complex comparisons of the individual events, such as disagreement between classifiers on the same sample or disagreement of classifiers on two containers within the same truck. Each classifier/extraction technique is contained within its own bundle, which allows the removal, introduction and updating of the tools and techniques used without compromising the running system. Data may also be streamed live or viewed historically using the visualisation suite, which connects through the Playback bundle.

Visualisation The final component WAIST is the visualisation suite. These tools were built using the Eclipse Rich Client Platform (RCP). As the RCP platform is built using OSGi, this allows SIXTH to be the core of the visualisation suite which again eases the transmission and distribution of data. Data can be visualised in a number of ways: any numerical data can be graphed using tools provided within SIXTH and location based data can be plotted on maps powered by OpenStreetMap. Figure 7.6b shows the realtime location of a target vehicle as a blue truck icon. Figure 7.6a conveys the presentation of acceleration data in two ways; its effect on the rotation of a 3D representation of its real-world waste container and a simple line chart.

Every aspect of WAIST is built from a SIXTH component; it is a fully integrated SIXTH application. Information produced by sensor nodes is continuously analysed using custom pattern recognition algorithms to determine if any tampering has occurred. Figure 7.5 illustrates the system architecture of WAIST wherein SIXTH provides the network adaptors. All the defined receivers are instances of the SIXTH information receivers. For example in Figure 7.5 a database receiver is used to populate an application level database, and a database adaptor is utilized for information retrieval. The remote adaptor connected to the WSN connector is based upon to the RemoteSensorNetworkAdaptor discussed in Section 5.7.1. The evaluation of WAIST was conducted in two parts. The first was a user trial in which students wrote sensor programs using SIXTH, WAIST, and ASTRA agents. The second component

---

**Figure 7.5: WAIST System Architecture (Russell et al., 2013)**

18 [www.openstreetmap.org](http://www.openstreetmap.org)
involved a real-world trial of waste routing, disposal, and the identification of suspicious movement patterns.

### 7.5.2 Energy Management in a Home Context

This section describes two research projects which have utilized the SIXTH middleware as an enabling framework for the control of sensor networks and the consumption and storage of the networks observations.

**Autonomic Home Area Network Infrastructure (AUTHENTIC)**

The AUTHENTIC (O’Sullivan et al., 2014) project is a multi university, in consortium with multiple industry partners, initiative aiming to deliver a Home Area Network (HAN) platform capable of empowering decision making in relation to energy management in the home context. Autonomic management of energy usage is facilitated by intelligent agents driven by the Agent Factory Micro Edition (AFME) platform (Muldoon et al., 2006).

This project utilizes both physical sensors in the home and external cyber-sources such as on-line weather data. A web-based viewer offers real-time streaming of sensor data from the home environment; this is visualized in an intuitive, user friendly, HTML5 Web-application and a native Android application. The user-focused applications communicate with the AUTHENTIC server through a well-defined RESTful interface.\(^1^9\)

\(^{19}\)This is distinct from the SIXTH REST interface defined in Section 5.7.1

---

**Figure 7.7** conveys the role of SIXTH in the AUTHENTIC project. The *Communications Module* interfaces with sensor devices, such as SmartPlugs and kettles, over Zigbee (ZigBee-Alliance, 2006), Bluetooth (Haartsen, 2000), and 6LowPan (Mulligan, 2007) standards and conveys the sensor observations to the *SIXTH Communications Adaptor*. The adaptor transforms the information into the SIXTH standard data format. The information is then disseminated to the two defined subscriber components by the *Sensor Data Broker*. The first subscriber is the *database manager* which stores the sensor data in the database. The second subscriber is the *AFME Agent Layer* which forwards data to the agents which are managing the home. Listing 7.3 depicts the program code for the `task(ITaskingMessage)` method of the `DeviceAdaptor` inherited from the `AbstractSensingNetworkAdaptor` base class. This method accepts `TaskingMessage` instances which with a `RETASK CommandType` and helper objects for toggling device detection on and off.
Listing 7.3: DeviceAdaptor Tasking Code

```java
@Override
public boolean task(TaskingMessage arg0) {
    int deviceID = (int) arg0.getNodeID();

    if (arg0.getCommandType().equals(TaskingMessage.COMMANDTYPE.RETASK) && arg0.getTaskingObject() instanceof OnOffCommand) {
        boolean on = ((OnOffCommand) arg0.getTaskingObject()).isOn();
        if (on) {
            this.deviceDetector.deviceActuatorOn(deviceID);
        } else {
            this.deviceDetector.deviceActuatorOff(deviceID);
        }
        return true;
    } else if (arg0.getCommandType().equals(TaskingMessage.COMMANDTYPE.RETASK) && arg0.getTaskingObject() instanceof LevelCommand) {
        int level = ((LevelCommand) arg0.getTaskingObject()).getLevel();
        this.deviceDetector.deviceActuatorLevel(deviceID, level);
        return true;
    } else {
        return false;
    }
}
```

The AUTHENTIC agent layer performs actuation through the device agnostic Tasking Service; upon reception of a request the service forwards the message to the adaptor which constructs network specific instructions conforming to the requests intention. The semantic adaptor feeds semantically enriched information into SIXTH using an ontology specified by the Web Ontology Language (OWL).

This project constitutes the deployment of the SIXTH Middleware in five Irish homes and a test suite in the Tyndall National Institute. This concretely demonstrates the usage of SIXTH in real-world scenarios thus espousing its stability and usefulness.

![Figure 7.7: AUTHENTIC System Architecture (O’Sullivan et al., 2014)](image)

Autonomous Balancing of Load Energy (ABLE)

The ABLE project (Kazmi et al., 2013) is a further example of SIXTH usage in an energy management scenario. The ABLE ambient intelligence framework is designed to capture, quantify, and present energy consumption patterns within the home organized via spatial and temporal dimensions. This provides

---

20this will potentially rise to ten homes in 2015
support for advantageous use of dynamic pricing schemes and injection of energy surplus back into the electrical network.

ABLE utilizes SIXTH for the retrieval of sensor observations from heterogeneous physical platforms (TelosB, Episensor, Sun SPOT) and web-based cyber-sensor platforms (ESB tariff information, weather forecasting). Figure 7.8 depicts the structure of ABLE. Therein SIXTH is present within the Web Crawler, Device Control, and Monitoring components. This illustrates the flexibility of SIXTH and the value inherent in a unified information capture framework.

ABLE requires observation of environmental parameters such as luminance level, temperature, CO2 level, occupant behaviour, weather forecast, energy prices, and energy usage, etc. To collect this information both physical sensors and cyber sensors are required.

Physical Sensors - To obtain information on luminance level, temperature, and humidity the SunSpot 21, TelosB 22, and Tyndall 23 physical sensing platforms were utilized. To obtain energy consumption readings we’ll use EpiSensor 24 which uses clip-on technique to measure electricity usage.

Cyber Sensors - To obtain dynamic pricing information from electricity suppliers we use SIXTH cyber sensors. A cyber sensor has been implemented for each major Irish supplier i.e. ESB, Airtricity, and Bord Gais. The scenario also needs access to weather forecast data and for this we incorporate a cyber sensor programmatically linked to the Yahoo Weather API. Section 7.3 provided a discussion of the Cyber-sensing core which underpins these adaptors.

Figure 7.9 provides a layered view of the ABLE system in operation. Therein, the data received from the sensors, the data processed by SIXTH, the decisions taken by the agents, and the operation of the computation component are all conveyed. There is a network of eight sensors (two cyber and six physical) measuring and retrieving data on different parameters. These parameters include weather forecast, energy pricing tariff, PIR, temperature, humidity, light, and energy consumption readings. Figure 7.9 also convey the visualisation of the data after it is being processed by each system layer.

---

21 SunSpot: http://www.sunspotworld.com
22 TelosB: http://www.memsic.com
23 Tyndall: http://www.tyndall.ie
24 EpiSensor: http://www.episensor.com
7.5.3 Discussion

WAIST (Section 7.5.1), AUTHENTIC (Section 7.5.2), and ABLE (Section 7.5.2) are three application scenarios in which SIXTH has been utilized. This is demonstrative of the capability to utilize SIXTH as the observation provider in applications which require contextual information for their operation.

7.6 Semantic Modelling with SIXTH

The work of Barnard Kroon is concerned with providing a semantic model of sensor networks. For this work SIXTH is utilized as a sensor resource connector. Additionally core functionality for this extension is implemented as a replacement for the IDataBroker (see Section 5.3.4). This is owing to the semantic layer functionality to reconfigure many aspects of the sensor networks in line with consumer demands. Semantic queries are constructed from IGenericQuery<ISensorData> instances provided by Data Broker subscribers. Figure 7.10 illustrates the environment model from the perspective of Node-1. The node has 1 local sensor and is cognisant of three remote sensors which belong to its neighbours. In this model when a query cannot be satisfied by the node it is dispensed to suitable neighbours. If no such neighbour is found the query may be broken down if possible and the reasoning process repeated with each sub-query.
7.7 Extending Android With SIXTH

AndroSIXTH (Gorgu et al., 2013) is both an extension and a porting of the SIXTH framework. The core of SIXTH, which was developed as a gateway-side middleware solution, is re-purposed as in-network solution on computationally capable Android devices. This re-use of sensor middleware frameworks harkens back to adoption of the GSN (Aberer et al., 2006b) code base for the MOSDEN (Perera, Jayaraman, et al., 2014) middleware, which also targets Android devices. The aforementioned was discussed in Section 2.5.10.

Figure 7.11 depicts the high-level architecture of AndroSIXTH and its interaction with Android Client Applications. SIXTH and the Android extensions are bundled within the Felix OSGi implementation and placed within a Background Service. Therein the AndroidAdaptor interfaces with the sensors native to the host device; this is performed via the Android sensor manager. The SIXTH service declares its binding interface through the Android Interface Definition Language (AIDL). Client applications can register their data interests through this API. The clients interact with this library through a single Sensor object. AndroSIXTH has been used in Head Mounted Displays (HMDs), for the provision of augmented reality, and Smart Phones. Section 7.7.1 describes the usage of SIXTH in an EU project. Section 7.7.2 examines the discovery layer developed for AndroSIXTH. In Section 7.7.3 some notes are providing as regards ongoing work on the commercialization of the SIXTH middleware.

7.7.1 Citizens Observatory WEB (COBWEB)

AndroSIXTH is utilized within the The COBWEB project. COBWEB is an effort to develop a back-end infrastructure for crowd-sourcing of environmental data through mobile devices. Android SIXTH is utilized to share sensory observations between nearby devices in a M2M model, and to underpin the participatory sensing application which dispatches relevant observations to the COBWEB server.

![AndroSIXTH Architecture](image)

7.7.2 Portable Object Discovery Service (PODS)

PODS is a heterogeneous network discovery framework for wireless MESH networks which has been woven into the AndroSIXTH extension of the SIXTH Middleware. The primary aim of PODS is to incorporate diverse discovery protocols into a cohesive framework in furtherance of dynamic and seamless sensor resource discovery and interactions. Figure 7.12 depicts the PODS system architecture. Therein, the discovery functions are decomposed into interaction and service layers. The interaction layer is
more tightly coupled with SIXTH and handles all bi-directional communication between the two. PODS utilizes a superset of the SIXTH sensor abstractions defined in Chapter 4. The SIXTH SensorDescription is wrapped and augmented with additional discovery related fields.

Figure 7.12: PODS Architecture

7.7.3 Commercialisation of SIXTH

SIXTH is being used by Mesh Mobile Networks in a mesh networking application. The mobile application uses SIXTH to create a mesh network that can share GPS, or any observation the device can capture, and text messages. The development of this application was funded by Climate Kic, which provided €20000 funding. The application aims to conserve energy by allowing users to share GPS information and bypass existing networks to use low powered decentralised techniques to send and receive messages.

7.7.4 Discussion

ISO25010 (ISO, 2011) includes portability as one its eight characteristics of software quality, wherein portability is so defined:

Portability is the degree of effectiveness and efficiency with which a system [...] can be transferred from one hardware, software or other operational or usage environment to another.

AndroSIXTH demonstrates the portability and extendibility of the SIXTH middleware by fully harnessing its functionality and abstracted domain representations and repositioning them inside dynamic MESH networks of computationally rich nodes. The movement for gateway to in-network represents a significant change in usage environment. AndroSIXTH is ground-breaking as it represents the movement of a middleware designed as a gateway solution into the network internals. Objectively it must be stated that this is somewhat a function of growing computational capability within sensor devices. However, this is still a testament to the flexibility of the design. In the research literature the author is aware of only one other such re-purposing that of MOSDEN employing the GSN architecture on Android.

Some weaknesses have been identified within the approach. The entire system encompasses many layers of abstraction which results in a heavy software stack. Android development is error prone and the depth of abstraction has further compounded these problems.

7.8 Conclusions

In this chapter a diverse set of projects that utilize SIXTH were discussed. These applications highlight the usage of SIXTH in disparate scenarios such as the monitoring of industrial waste, autonomic management of the home environment, and the tracking of wildfires. AndroSIXTH highlights the extensibility of SIXTH by extending the core functionality to deliver a solution for Android devices. SNoMAC illustrates the use of SIXTH with another sensor processing platform to form a system-of-systems and avoid any duplication of effort in developing connections with information streams.

As a macro-level observation SIXTH has served to prevent, in the context of these case studies, the reinvention of the wheel with regard to low-level abstractions and domain representations. The developer has less work to do, and crucially, less work outside their specific interests. No major hurdles were encountered in fitting the needs of the application to the conceptual framework of SIXTH. In retrospect it may prove fruitful to examine how SIXTH effects the structure of sensing applications built above it.

This successful diverse usage provides a strong indication that SIXTH delivers an appropriate level of abstraction so as to be useful in divergent application scenarios.

In the upcoming Chapter 8 the second component of evaluation for the SIXTH middleware is discussed.
8 | Evaluation

“Not everything that can be counted counts, and not everything that counts can be counted.”

– William Bruce Cameron, *Informal Sociology* (Cameron, 1963)

8.1 Preface

Chapter 7 empirically demonstrated the extensibility, flexibility, and usability of SIXTH through its use in diverse research projects; of particular note is the deployment of SIXTH in four Irish households as part of a home energy management solution. These case studies serve as the first element in the evaluation of SIXTH.

The evaluation of SIXTH presented herein takes a two-pronged approach to evaluation scaffolded by the previous evaluative case studies. Firstly, suitable metrics are used to conduct a comparison between SIXTH and two leading competing gateway middleware offerings. Secondly, a survey of SIXTH users is conducted in order to evaluate the usability of SIXTH, its impact on development life-cycle and the effectiveness of its abstractions. These objectives were established in Section 1.3.2 as foundational elements of the primary objective.

Section 8.2 establishes the basis for the evaluation framework through the lens of software quality assessment examining the lack of consensus on the meaning of such a term and the various models proposed for quality assessment in a myriad of dimensions.

Section 8.3 conveys a comparison table of the evaluation strategies applied to over thirty middleware platforms native to the sensor domain. The vast majority incorporate some aspect of performance testing, this is often superficial, and some influential figures within the literature have expressed scepticism as regards the importance of such results. These issues are examined.

Section 8.4 presents the first element of the evaluation strategy. SIXTH is evaluated through the application of code quality metrics. The extracted statistical data regarding the project structure is compared to two other prominent sensor middleware implementations.

Section 8.5 outlines the assessment of SIXTH from a usability and utility perspective via the pre-existing developer user-base which have developed both industry and research-focused projects with SIXTH. The particulars of the practitioners survey are given and subsequently the results are examined. Feedback received in such an exercise is a key determinant in identifying what features need to be extended and how best to improve ease of use.

Section 8.6 rounds out the chapter by providing a discussion and summary of the preceding work which is then leveraged within the forthcoming Chapter 9 in which there is a critique and identification of future work.
8.2 What Is Software Quality?

This section considers the complex area of assessing the quality of a software product. Such considerations are fraught with difficulty and are subject to ongoing research. Pressman, in his well-regarded software engineering text (Pressman, 2001), defined software quality as follows:

“An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.”

Pressman acknowledges any definition as subject to endless debate. Indeed, many other non-oppositional respected definitions have been proposed in the literature including: (Edwards, 1986; Feigenbaum, 1956; Juran, 1962) and (Kitchenham et al., 1996).

Kitchenham (Kitchenham et al., 1996) was influenced heavily by the work of Harvard business professor David Garvin from his influential text “What does ‘product quality’ really mean?” (Garvin, 1984). Therein Garvin concluded that, generally speaking, “quality is a complex and multifaceted concept”. Consequently, Garvin contributed an identification of five general approaches to quality assessment; these are described as follows:

- **Transcendental**: This refers to quality as something recognizable but which escapes definition. (Kan, 2002) likens this view to Justice Potter Stewart’s infamous: “I know it when I see it” (Stewart, 1964). By definition transcendental software quality cannot be assessed objectively. It is as Wittgenstein remarked, on conceptions with defy language despite understanding, “Whereof one cannot speak, thereof one must be silent” (Wittgenstein, 1994).

- **User-View**: This refers to “highly subjective”, but essential, viewpoint offered by a user. Herein this view is assessed through a practitioners survey detailed in [Section 8.5](#).

- **Manufacturing**: This view assesses quality as how well the resultant product conforms to its specification. Does the product do what it set out to do? This conformance of SIXTH with its design goals is a cross-cutting issue. This factor is assessed in the case studies presented in [Chapter 7](#) and [Chapter 10](#) there is a measured reflection on how SIXTH achieved the goals set out in [Chapter 1](#).

- **Product**: This approach stresses quality as a measure of the inherent qualities of a product. (Kitchenham et al., 1996) note that in software this often refers to the advocacy of code metrics applied as internal quality indicators. The reasoning espoused asserts that internal quality will reflect in “quality in use”. This characteristic is assessed by the software metrics based evaluation of SIXTH conducted in [Section 8.4](#).

- **Value-based**: This view assesses quality as how much a customer is willing to pay for the product; In software development this is most often zero.

8.3 Middleware Evaluation

This section provides a discussion of how middleware platforms have been evaluated previously in the domain of Wireless Sensor Networks. The evaluation of complex software systems, particularly those which are intermediaries between other complex systems, is non-trivial. In an influential and in-depth literature review Mottola identifies that answering the question “how good is my middleware?” is one of several key research questions for the sensor networks domain (Mottola et al., 2011).

---

1Potter was making his ruling in a legal matter concerning obscenity

---
Mottola is critical of both the over-reliance on performance testing and the manner in which it is typically conducted i.e. via simulation. This thesis is in agreement with the assertion that “the impact on programming effort is overlooked”. Ensuring programmer productivity and the quality of the software is, as Mottola put it, the defining consideration.

Table 8.1 conveys a summary of the manner in which thirty-five sensor domain middleware platforms have been evaluated. Thirteen systems had as part of their evaluation a case studies element. Through this mechanism the authors attempt to demonstrate the value and efficacy in, variably, quasi-real-world application scenarios. This has proven useful in demonstrating value and in the identification of the shortcomings of one’s own work. Assured, by the prevalence, of the suitability of such a model a case-studies based evaluation was described in Chapter 7.

Table 8.1 shows a overwhelming focus on performance. Sixteen systems were evaluated in this way. In doing so systems were measured for processing time, introduced overhead, routing time, robustness, memory usage etc. Following the comments of Mottola discussed above, in the evaluation of SIXTH software quality was chosen over performance metrics. This is also prudent as performance is a lesser consideration for gateway, and lesser constrained in-network, middleware of which SIXTH is representative.

Table 8.1: Middleware Evaluation Survey

<table>
<thead>
<tr>
<th>System</th>
<th>Methodology</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSN (Aberer et al., 2006b)</td>
<td>Performance-based</td>
</tr>
<tr>
<td>UniversAAL (Ram et al., 2013)</td>
<td>Implementation of reference model</td>
</tr>
<tr>
<td>ManySense (Westlin et al., 2014)</td>
<td>Performance-Based Case Studies</td>
</tr>
<tr>
<td>Mosden (Perera, Jayaraman, et al., 2014)</td>
<td>Performance-based</td>
</tr>
<tr>
<td>SStreamWare (Gurgen et al., 2008)</td>
<td>Case-studies</td>
</tr>
<tr>
<td>Senseive (Hermann et al., 2008)</td>
<td>Performance-based</td>
</tr>
<tr>
<td>WSNWare (Viani et al., 2013)</td>
<td>Real-world Deployment</td>
</tr>
<tr>
<td>Agilla (Fok et al., 2005a)</td>
<td>Performance-based &amp; Case Studies</td>
</tr>
<tr>
<td>SenseWrap (Evensen et al., 2009)</td>
<td>Performance-based</td>
</tr>
<tr>
<td>SNPS (Di Modica et al., 2014)</td>
<td>Case-studies</td>
</tr>
<tr>
<td>TinyDB (Madden, Hellerstein, et al., 2002)</td>
<td>Performance-based &amp; Efficacy</td>
</tr>
<tr>
<td>Servilla (Fok et al., 2012)</td>
<td>Performance-based Case-studies</td>
</tr>
<tr>
<td>Construct (Coyle, Neely, Rev, et al., 2006)</td>
<td>Case-studies</td>
</tr>
<tr>
<td>Kairos (Gunnadi et al., 2005)</td>
<td>Case-studies &amp; Performance-based</td>
</tr>
<tr>
<td>Impala (Liu et al., 2003)</td>
<td>Performance-Based</td>
</tr>
<tr>
<td>SpatialViews (Ni et al., 2004)</td>
<td>Performance-based</td>
</tr>
<tr>
<td>Active Messages (Buonadonna et al., 2001)</td>
<td>Qualitative &amp; Performance-based</td>
</tr>
<tr>
<td>Atlas (King et al., 2006)</td>
<td>Case-Studies</td>
</tr>
<tr>
<td>BISNET (Boonma et al., 2007)</td>
<td>Performance-based</td>
</tr>
<tr>
<td>DAVIM (Horre et al., 2008)</td>
<td>Performance-based</td>
</tr>
<tr>
<td>DSWare (S. Li et al., 2003)</td>
<td>Performance-based</td>
</tr>
<tr>
<td>Envirotrack (Abdelzaher et al., 2004)</td>
<td>Case Studies &amp; performance</td>
</tr>
<tr>
<td>FAMIWARE (Gámez et al., 2011)</td>
<td>Case Studies &amp; performance</td>
</tr>
<tr>
<td>FACTS (Terlloth, Wittenburg, et al., 2006)</td>
<td>Performance-based</td>
</tr>
<tr>
<td>In-Motes (Georgoulas et al., 2006)</td>
<td>Case-study</td>
</tr>
<tr>
<td>KSpot (Andreou et al., 2011)</td>
<td>Example application</td>
</tr>
<tr>
<td>MAPS (Aiello et al., 2009)</td>
<td>Programming example &amp; Performance-based</td>
</tr>
<tr>
<td>MASTAQ (Hwang et al., 2005)</td>
<td>Performance-based</td>
</tr>
<tr>
<td>MidSN (Cecilio, J. Costa, et al., 2013)</td>
<td>Performance-based</td>
</tr>
<tr>
<td>Mires (Souto et al., 2004)</td>
<td>Example application</td>
</tr>
<tr>
<td>Perla (Schreiber et al., 2012)</td>
<td>Case-study &amp; Performance-based &amp; Qualitative</td>
</tr>
<tr>
<td>POGO (Brouwers et al., 2012)</td>
<td>Case-study &amp; Performance-based &amp; Qualitative</td>
</tr>
<tr>
<td>OFEN (Guo et al., 2011)</td>
<td>User-survey</td>
</tr>
<tr>
<td>WASP (Bai et al., 2011)</td>
<td>User-survey</td>
</tr>
<tr>
<td>USEME (Caete et al., 2008)</td>
<td>Example Application</td>
</tr>
</tbody>
</table>
8.4 Software Quality Metrics

Software metrics are quantitative measures of some feature of a software component (Fenton et al., 1998) such as the number of lines in the source file, or the complexity of a method. One of the advantages of metrics evaluation is that it is eminently repeatable and the results are easily understood.

The remainder of this section is decomposed as follows. Section 8.4.1 establishes the software metrics which are to be utilized, this selection is primarily based on the work of Martin (Martin, 1994) & Halstead (Halstead, 1977). Section 8.4.3 details the utilization of these metrics on the SIXTH Middleware in isolation. This was performed on the core SIXTH bundles only, as specific component implementations are considered separate from their underlying abstractions. Following from this Section 8.4.4 compares SIXTH with nearest neighbour offerings. As SIXTH is a gateway middleware, those chosen arise this category (GSN and WSNWare). As per the individual evaluation, a “core” was identified and used for evaluation so as to provide a like-for-like comparison in so far as is possible.

8.4.1 Metrics Utilized

This section discusses the metrics used in the evaluation of SIXTH and in comparing it with WSNWare and GSN. The full listing of the metrics is included in Table 8.2. Where the metric name is sufficient explanation the author shall not repeat.

- Martin’s Software Package Metrics (Martin, 1994)

The following are the package analysis metrics defined by Martin (Martin, 1994). These are so included as they are useful for package level analysis of large projects, provide useful insights into the code structure, are widely accepted, and routinely used.

  - Abstractness: The ratio of abstract classes and interfaces to the total number of types. Given as \( A = \frac{T_a}{T} \).

  - Afferent Coupling (Ca): This is the measurement of the number of external classes which depend, through use or extension, on one or more of the classes in the package being measured (Martin, 1994). This is also known as fan-out dependency.

  - Efferent Coupling (Ce): This metric is associated strongly with afferent coupling. It is defined as the count of the number of dependencies exhibited by the measured elements on outside types e.g. the count of how dependent package A is on package B. A high efferent coupling is understood to increase brittleness and exacerbate the ripple effect created by changes in external packages.

---

1 Processing time, scalability in query processing
2 Compared with direct use of sensors, “code complexity”
3 Addition of new adaptor and integration with existing exergame
4 Discussion of ease of use provided through familiar SQL interface
5 Memory footprint, service binding and invocation
6 Routing, localization
7 Impala in evaluated with respect to the overhead introduced; and its effect on energy-management, routing, and robustness
8 Assessing the case of the introduced abstraction layers in comparison to plain coding and other middleware
9 Reduction of communication and accurate event detection
10 Implementations for TinyOS, Android
11 Memory usage, overhead, and scalability
12 Developer time reduction
13 Suitability of the model, complexity, case of use
14 Users attempted to create a program using OPEN in 60-80 minutes. Subsequently, a user-survey was completed.
15 Assess suitability of WASP by assigning 28 novice programmers to complete tasks in 4 comparison languages and with WASP.
16 Warmly known as Uncle Bob
17 from another package
Low coupling relates to cohesion, defined as the “degree to which the elements of a module belong together” (Yourdon et al., 1979). Cohesion has been long considered as correlated with good software design. This relates to both afferent and efferent coupling. For example, in the efferent case high coupling (to external elements) may indicate that a class belongs in another package.

- **Instability:** This is a measurement of the ratio of the efferent coupling to the sum of the efferent and afferent couplings. This is utilized as a measure of how insulated a package is to change. It is given as follows:

\[
I = \frac{C_e}{Ca + Ce}
\]  

(8.1)

- **Distance:** This metric is an indicator of the balance in a package between abstraction and instability. It is measured as the distance to the main sequence shown in Figure 8.1.

![Figure 8.1: Abstraction-Instability Graph (Martin, 1994)](image)

- **Block Depth:** A block is a grouping of code; in the Java language this is a grouping between balanced {} braces. Methods which include nested blocks are considered detrimental to code readability and maintainability. For instance block scope means variables which are functionally separate can have the same name; when re-factored this can lead to adverse consequences. For this metric a low number is a desirable result.

- **Average Cyclomatic Complexity:** Measurement of the number of linearly independent execution paths in a method (McCabe, 1976). High complexity begets more complicated methods which are more difficult to test. Consequently it is desirable that each method have a low cyclomatic complexity, which yields a low complexity for the class, the package, and the project.

- **Depth of Inheritance Hierarchy:** This metric examines the overuse of inheritance. This can prompt further analysis of whether the inheritance relationships are sensible. This is a very useful metric when dealing with OO code-bases which SIXTH, GSN, and WSNWare are all instances of. Desirable results are low in this case, but it is difficult to say what is acceptable or otherwise. A high result > 5 ought to prompt a re-examination of the code structure. For instance some is-a relationships may be unsuitable and should be re-constituted as has-a relationships.

- **Lines of Code:** A class with high LOC is indicative of over-responsibility (e.g. a “God Object”), thus indicating a need for re-factoring. Low overall SLOC correlates strongly with maintainability. LOC is ubiquitous and well-understood however, its usefulness is debatable. LOC can give a skewed perspective in judging accomplishment and the programmer effort required. For example, a LISP program is concise, however the equivalent Java program might be produced with lesser effort and be more maintainable.
Some research (Prechelt, 2000) has shown correlations between program length and time-based
development effort. Ballmer and Wozniak (Cringely et al., 1996) are critical of the dogmatic
application of LOC measures arguing that they discourage simplicity and elegance.

- **Number of Constructors Per Type:** This metric is useful in measuring some facets of class
  inconsistency. As the number of constructors increases the likelihood of defects in object state
  increases. Too many constructors may also decrease ease of use as the developer may become
  confused.

- **Number of Parameters** This is a measure of the number of arguments a method takes. In Clean
  Code (Martin, 2008) Martin stresses that the optimum number of arguments is 0, and that more
  than 3 should be avoided. More parameters means more state and more things which can go wrong.
The number of parameters is strongly correlated with method over-responsibility.

Consequently, there is a danger of invalidating the crux of the Unix philosophy (Schwartz et al.,
2001), Curly’s law (Atwood, n.d.) and the intent of the single-responsibility principle (Martin, 2003),
which espouse an essential aspirational axiom - “do one thing and do it well”.

- **Halstead Metrics:** This influential suite of metrics, commonly referred to as software science,
  was developed by Halstead (Halstead, 1977) in an effort to bring more more scientific evaluation
  into computer science in a language independent manner. All of the metrics defined are functions
  of the following building blocks: (i) count of unique operators (n1) (ii) count of unique operands
  (n2) (iii) total operators (N1) (iv) total operands (N2).

  These metrics are useful because they are implementation independent, well understood, and
  accepted, if somewhat criticised.

  - **Length:** The length, N, is calculated as follows:
    \[ N = N_1 + N_2 \]  
    (8.2)

  - **Vocabulary:** The vocabulary, denoted n, is similarly calculated from n1 and n2:
    \[ n = n_1 + n_2 \]  
    (8.3)

  - **Volume:** The volume, V, of a program is calculated as follows:
    \[ V = N \times \log_2 n \]  
    (8.4)

    This is understood to be a reasonable measure of the, language independent, size of an
    algorithm.

  - **Level:** This metric defines the ratio of the volume of the program to the its most compact
    representation. This is given as follows:
    \[ L = \frac{2}{n_1} \times \frac{n_2}{N_2} \]  
    (8.5)

  - **Difficulty:** This is defined as the inverse of the level, given as \( D = 1/L \). As such a longer
    implementation of a given function receives a higher difficulty score.

  - **Effort:** This metric refers to the effort which is required to create or comprehend a measured
    program.
    \[ E = D \times V \]  
    (8.6)
8.4.2 Metrics Software Selection

The metrics analysis was facilitated by the Google CodePro Analytix\(^{19}\). This tool provides a rich-set of metrics, is easy to use, and is fully integrated with the Eclipse IDE. The Metrics\(^{20}\) suite in which the author was experienced no longer plays nicely with Eclipse and was disregarded. Other tools such as Java Metrics\(^{21}\) and JHawk\(^{22}\) lacked maturity and supporting documentation.

8.4.3 SIXTH Metrics Results

Table 8.2 presents the combined results of running the Google CodePro Analytix metrics suite on \textit{ie.ucd.sixth.core.interface} and \textit{ie.ucd.sixth.core.impl}. The former Java project contains the SIXTH interfaces and the latter comprises the default implementation. Section 5.1 discussed this deliberate decoupling which aims to make it easier to program to the interface, provide your own implementation, and make the concepts behind the system clear. Where there is an \(\checkmark\) no package, class etc. of SIXTH violated the metrics threshold. \(\times\) denotes a violation of the threshold by some, possibly just one, sub-component.

Our discussion will focus in the areas where a threshold has been violated to assess to degree of the infraction and whether the metric has identified a real problem. A violation does not necessarily mean something is wrong; Martin cautioned against being too dogmatic in the application of metrics.

**Cyclomatic Complexity**  The Cyclomatic Complexity threshold (6) is violated by a nested class contained with within \textit{ComparisonQuery}. The \textit{ComparisonOperator} class is concerned with parsing operators for queries generated by the query language interpreter. The violation herein is marginal (6.5).

**Number of Constructors**  The \textit{SensorData} class violates the threshold for instance constructors. The class has six constructors. This was intended to give flexibility in instantiation. In hindsight, this could result in confusion from a user perspective. Section 8.5.5 identifies dissatisfaction with the sensor data format by some of those surveyed; this abundance of constructors is a likely factor in this expression.

**Number of Methods per Type**  This metric threshold was violated to the greatest extent. \textit{AbstractSensorNetworkAdaptor} serves as an example for us to consider the implications and mitigating factors in this instance. As per Table 8.2 the average SLOC is a favourable 3.94. Thus, we can infer that the design is very modular. Functions have been broken down into smaller helper functions to aid understanding and debugging. However, approaching the analysis from a API perspective \textit{AbstractSensorNetworkAdaptor} inherits fifteen of its twenty-eight methods from its ancestors. The majority of the remainder are \textit{protected} and \textit{private}. It is felt that the metric may require modification to score such methods less harshly. Some suggestions on correcting this defect are discussed in Section 9.2.

**Efferent Coupling**  The \textit{query} package presents with a high efferent coupling. This is not an intrinsic negative as it is not wrong to reference files external to a package. The reason for the high coupling is that queries, by their nature, are defined for different objects existing in other packages. This could be resolved by, for instance, moving concrete queries related to sensors to that package, rather than maintaining a dedicated package in the implementation project. This would also service functional cohesion (McConnell et al., 2004).

---

\(^{19}\)https://developers.google.com/java-dev-tools/codepro/doc/
\(^{20}\)http://metrics.sourceforge.net/
\(^{21}\)http://www.monperrus.net/martin/java-metrics/
\(^{22}\)http://www.virtualmachinery.com/jhawkmetrics.htm
**Number of Subtypes** AbstractGenericQuery exceeds the threshold for inheritance as it has 10 subtypes. As with Paragraph 8.4.3 there is nothing inherently negative about inheritance. However, this does mean that 10 classes are directly effected by changes in a single class which has obvious knock-on effects for maintainability.

Table 8.2: Code Metrics Comparison: SIXTH, GSN, and WSNWare

<table>
<thead>
<tr>
<th></th>
<th>SIXTH</th>
<th>WSNWare</th>
<th>GSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstractness</td>
<td>46.40%</td>
<td>-</td>
<td>43.10%</td>
</tr>
<tr>
<td>Afferent Couplings</td>
<td>51</td>
<td>11</td>
<td>196</td>
</tr>
<tr>
<td>Avg Block Depth</td>
<td>0.7</td>
<td>1.01</td>
<td>1.18</td>
</tr>
<tr>
<td>Avg Cyclomatic Complexity</td>
<td>1.14</td>
<td>2.03</td>
<td>2.56</td>
</tr>
<tr>
<td>Avg Depth of Inheritance Hierarchy</td>
<td>1.75</td>
<td>2.05</td>
<td>2.66</td>
</tr>
<tr>
<td>Avg Lines Of Code Per Method</td>
<td>3.94</td>
<td>✓ 6.65</td>
<td>✓11.83</td>
</tr>
<tr>
<td>Avg No. of Constructors Per Type</td>
<td>0.68</td>
<td>✓ 1.12</td>
<td>✓ 0.48</td>
</tr>
<tr>
<td>Avg No. of Fields Per Type</td>
<td>1.45</td>
<td>✓ 1.69</td>
<td>✓ 3.43</td>
</tr>
<tr>
<td>Avg No. of Methods Per Type</td>
<td>4.73</td>
<td>✓ 6.74</td>
<td>✓7.06</td>
</tr>
<tr>
<td>Avg No. of Parameters</td>
<td>0.62</td>
<td>✓ 0.94</td>
<td>✓ 0.96</td>
</tr>
<tr>
<td>Avg No. of Subtypes</td>
<td>0.67</td>
<td>✓ 0.74</td>
<td>✓ 0.31</td>
</tr>
<tr>
<td>Comments Ratio</td>
<td>7.10%</td>
<td>-</td>
<td>14.40%</td>
</tr>
<tr>
<td>Difficulty</td>
<td>126.07</td>
<td>✓ 175.76</td>
<td>✓224.53</td>
</tr>
<tr>
<td>Distance</td>
<td>0.12</td>
<td>✓ 0.32</td>
<td>✓ -0.31</td>
</tr>
<tr>
<td>Efferent Couplings</td>
<td>96</td>
<td>✓ 79</td>
<td>✓ 297</td>
</tr>
<tr>
<td>Effort</td>
<td>41,303,713.21</td>
<td>✓ 40,564,398.91</td>
<td>✓ 407,160,506.74</td>
</tr>
<tr>
<td>Instability</td>
<td>0.65</td>
<td>✓ 0.87</td>
<td>✓ 0.6</td>
</tr>
<tr>
<td>Lines of Code</td>
<td>3.978</td>
<td>✓ 6.042</td>
<td>✓ 36.598</td>
</tr>
<tr>
<td>No. of Characters</td>
<td>167,956</td>
<td>✓ 352,544</td>
<td>✓ 2,122,219</td>
</tr>
<tr>
<td>No. of Comments</td>
<td>285</td>
<td>✓ 872</td>
<td>✓ 3,012</td>
</tr>
<tr>
<td>No. of Constructors</td>
<td>85</td>
<td>✓ 107</td>
<td>✓ 169</td>
</tr>
<tr>
<td>No. of Fields</td>
<td>246</td>
<td>✓ 289</td>
<td>✓ 1,746</td>
</tr>
<tr>
<td>No. of Lines</td>
<td>6,740</td>
<td>✓ 9,400</td>
<td>✓ 57,123</td>
</tr>
<tr>
<td>No. of Methods</td>
<td>592</td>
<td>✓ 641</td>
<td>✓ 2,443</td>
</tr>
<tr>
<td>No. of Operands</td>
<td>7,156</td>
<td>✓ 15,052</td>
<td>✓ 93,145</td>
</tr>
<tr>
<td>No. of Operators</td>
<td>2,054</td>
<td>✓ 6,096</td>
<td>✓ 43,948</td>
</tr>
<tr>
<td>No. of Packages</td>
<td>38</td>
<td>✓ 18</td>
<td>✓ 41</td>
</tr>
<tr>
<td>No. of Semicolons</td>
<td>2,086</td>
<td>✓ 3,201</td>
<td>✓ 20,912</td>
</tr>
<tr>
<td>No. of Types</td>
<td>125</td>
<td>✓ 95</td>
<td>✓ 346</td>
</tr>
<tr>
<td>No. of Unique Operands</td>
<td>823</td>
<td>✓ 1,884</td>
<td>✓ 9,541</td>
</tr>
<tr>
<td>No. of Unique Operators</td>
<td>29</td>
<td>✓ 44</td>
<td>✓ 46</td>
</tr>
<tr>
<td>Program Length</td>
<td>9,210</td>
<td>✓ 21,148</td>
<td>✓ 137,093</td>
</tr>
<tr>
<td>Program Vocabulary</td>
<td>852</td>
<td>✓ 1,928</td>
<td>✓ 9,587</td>
</tr>
<tr>
<td>Program Volume</td>
<td>89,656.67</td>
<td>✓ 230,785.78</td>
<td>✓ 1,813,310.42</td>
</tr>
<tr>
<td>Weighted Methods</td>
<td>777</td>
<td>✓ 1,522</td>
<td>✓ 6,728</td>
</tr>
</tbody>
</table>

8.4.4 Comparison

Section 2.6 concluded that the focus of the thesis should be upon the development of a gateway-side middleware. In Section 1.3.2 we established the goal of evaluating the work undertaken through code metrics. It is pertinent therefore to compare SIXTH with nearest-neighbour gateway systems using these metrics.

GSN (Aberer et al., 2006b) and WSNWare (Viani et al., 2013) were chosen as suitable representative middleware to compare with SIXTH. Since GSN began life in 2006 the project has remained under active development and has been continually refined alongside research in WSN development. Consequently, the project is still relevant and routinely updated. GSN is popular, having been used by other research as the basis of other middleware (Perera, Jayaraman, et al., 2014). As a practical consideration both...
GSN and WSNWare are available as open source software, making the comparison achievable at the coding level. WSNWare while only recently appeared in published research has existed as an open source project since 2011. It exists in the same space as GSN - a gateway-side, WSN interpreter and controller. However, WSNWare is structured in a much more object-oriented manner in line with the sensibilities of the author. As such it was deemed to be a good system with which to compare. Higher-level comparisons with GSN and WSNWare are peppered throughout Chapter 5.

By virtue of functionality and availability Mosen (Bakhshi et al., 2013), Midgard23 and TinySOA (Avilés-López et al., 2009) would have also been suitable choices for comparison. However, these systems are less fully realised, of lesser academic impact, feature-poor, and more tightly bound in undesirable ways. Consequently, these were eliminated from contention.

GSN is a very large system. the standard download from the projects GitHub account has 177 external jars upon which the project is dependant. Much of what is contained therein does not, it is felt, belong to the core middleware architecture. It would appear that it is so presented to ease installation. Such extraneous components include WEKA modules, specific connection wrappers (tinyOS, XBee, IEE1451), utilities, UI elements, and domain modelling frameworks. In SIXTH such elements would be confined to separate OSGi bundles.

Consequently, and for the purpose of fairer comparison the GSN code-base was pruned to the best of our ability such that the remainder comprised the core abstractions and a set of elements which would require re-factoring to remove. This process is one fraught with potential error and underscores the difficulties of software comparison even when utilizing the same metrics suite on projects in the same programming language. The process to prepare WSNWare was comparatively simple. The core and common projects were converted to eclipse OSGi projects the metrics suite was run. The results conveyed by Table 8.2 provide a basis for the comparison of SIXTH with these nearest-neighbour systems.

**Abstractness:** The results show similar levels of Abstractness in SIXTH and WSNWare. This illustrates that both system designers felt it was appropriate to separate policy from mechanism, in line with standard software engineering thought. GSN is decidedly less abstract, from a review of the code-base no interfaces are defined. For example this means that the implementation of a Wrapper in GSN must always be done by extending AbstractWrapper. Consequently, flexibility is lost.

**Afferent Coupling:** As defined in Section 8.4.1 this measures external reliance on a package. WSNWare has very low coupling while GSNs average is much higher.

**Cyclomatic complexity:** The average for all three systems was excellent. As described in Section 8.4.3 there were classes in SIXTH which marginally exceeded the threshold. In GSN many outliers exceed the threshold significantly.

**Number of Constructors:** The comparison systems had similar issues with regard to parameter rich constructors, as was earlier discussed in Section 8.4.3. From an examination of the code-base this problem seems to frequently stem from the intent to provide flexibility. Undoubtedly in all cases a re-factoring, perhaps some re-design, could improve these foibles.

**Number of Fields per Type:** GSN and WSNware components violate this threshold. This issue is symptomatic of over-responsibility and/or too much mutable state in a single object. All SIXTH components passed on this criterion.

**Number of Parameters:** SIXTH had the lowest average number of parameters while WSNWare never violated the threshold.

---

23 https://github.com/fenrir/midgard

24 perhaps too tightly woven
8.5 Practitioners Survey

This section serves to describe and discuss the evaluation of SIXTH from the perspective of its user-base. The use of user surveys is a standard industry practice which can produce valuable results in a timely fashion. However, there are several known defects to such approaches with regard to subjectivity, and a lack of statistical significance when faced with a limited user-base or a lack of responsiveness from a larger grouping. This evaluation mechanism focuses upon users with significant, but varied, experience utilizing the middleware in the development life-cycle of both research projects and commercial applications. The bulk of this work is reflected in the case studies examined in Chapter 7. The remainder of this section begins by identifying the appropriate assessment framework.

8.5.1 Identification of Usability Assessment Survey

There exist a vast array of generic system usability and utility assessment surveys; each possessing its own strengths and shortcomings. In initial evaluation discussion consideration the Software Usability Measurement Inventory (SUMI) (Kirakowski et al., 1993) was considered. SUMI is a proven methodology for software quality assessment acknowledged in ISO 9241 (ISO, 2010). However, SUMI encompasses fifty questions placing a burden on the participants. Additionally some SUMI questions are specifically tailored toward unskilled users utilizing a GUI and are unsuitable to assess programmers utilizing a middleware. There exist many other well regarded software evaluation surveys: (Brooke, 1996; Davis, 1985, 1989; H. X. Lin et al., 1997). Ultimately, the decision was made to utilize SUS owing to its minimal question set (10 items), Likert scale integration, and its long standing successful application in related domains (T. Lim et al., 2007; Nithyanand et al., 2010). The 5-point likert scale (Likert, 1932) which captures the degree of user opinion has been identified (Coolican et al., 1994) as advantageous. Users felt it was “more natural”, it has proven valid and reliable, and is useful in capturing temporal shifts in user responses. Recent work (Brooke, 2013) considers SUS to be as reliable as other tools, some of which place greater burden on the user through a larger question set. Researchers have argued that SUS operates well with smaller sample sizes providing useful results with 8-12 participants (Tullis et al., 2004).

8.5.2 Survey Design

The bespoke survey, which incorporates SUS, is designed to gather and evaluate user opinion with regard to:

- The usability and learn-ability of SIXTH;
- The impact of SIXTH on the development life-cycle;
- The efficacy of the problem domain abstractions.

With regard to usability the definition given from ISO25010 (ISO, 2011) is used as a scaffolding framework for this evaluation component:

> “[Usability is the] degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.”

Within ISO25010 usability is further decomposed into a number of key issues (i) to what degree does the user recognize the software as appropriate? (ii) How easy is it to for the user to learn the software? (iii) to what degree does the software have attributes which make it operable? (iv) how is the user protected from errors? (v) to what degree is the interface intuitive? (vi) and how can the software be used by diverse users?
Given the above, it is prudent to restate that the users under consideration herein are programmers, of various skill levels and backgrounds. It follows therefore that consideration (vi) be restricted to those with appropriate pre-existing skills. The structure of the survey is as follows:

- **Demographic Information**: This section captures demographic data from the user-base. Sex and age were recorded.

- **Experience**: This section captures the self-reported experience levels with related technologies and disciplines. Users were asked to rank themselves, from novice to expert, on a 5-point likert scale with regard to the following: (i) Java (ii) Eclipse (iii) OSGi (iv) Sensor Networks (v) Generics (vi) Middleware (vii) Object-Oriented Programming (OOP) (viii) Programming (ix) SIXTH. Java, and to a lesser extent Eclipse and OSGi are the very concrete pre-requisites for working with SIXTH. It was felt a participant would be better placed to utilize the middleware having pre-existing domain experience (sensor networks) and/or experience with other middleware systems and concepts. It was decided to include OOP, general programming, and generics in the experience survey to gauge if difficulties in using SIXTH could be linked to inexperience with these items.

- **System Usability Scale (SUS)** As discussed in Section 8.5.1 SUS was chosen as a well-understood, reliable, and industry standard survey mechanism. Below the full set of ten SUS questions are included:

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

The full compliment of questions assesses usability however it has been found that Questions 4 and 8 provide a reliable assessment of learn-ability.

- **Bespoke Questions** The following bespoke questions were included as a more fine-grained augmentation of SUS. Questions 11-15 were scored on a 5-point likert (strongly disagree to strong agree). Questions 13 and 14 had additional free-text answer space and Question 17 was entirely free-text based. Each subcomponent of Question 16 was scored on a 5-point likert (not useful to very useful).

11. SIXTH provides appropriate abstractions.
12. SIXTH makes sensor application development easier.
13. I would be inclined to use SIXTH in appropriate research scenarios.
14. I consider a bespoke application to be superior to utilizing SIXTH.
15. The SIXTH architecture is easy to understand.
16. Rank these features of the middleware: (i) Adaptor (ii) Receivers (iii) Sensor Nodes (iv) Sensor Data Format (v) Tasking Service (vi) Data Broker
17. In what ways could the SIXTH middleware be improved?

### 8.5.3 Evaluation Procedure

The survey, included in its entirety in Appendix B was disseminated electronically and in person to the SIXTH user-base. No guidance was given except that the user be thorough, thoughtful, and critical in
their assessment. The surveys were left with the participant such that they could be filled in their own time. The administrator later returned to collect the completed surveys and offer thanks in chocolate form. Electronic distribution was a necessity as several users were overseas or working remotely.

Upon receiving the completed evaluation surveys from users this data was entered into a bespoke excel spreadsheet and results were extracted. These are discussed in the forthcoming section.

8.5.4 Results

In total the survey was administered to fifteen SIXTH users. Those surveyed included research assistants, undergraduate and masters students, PhD candidates, Post-doctoral researchers, and university lecturers in UCD. By design all of those surveyed had previously used SIXTH in an unstructured, unmonitored context to build the computational sensor infrastructure required as part of their research or coursework.

The unstructured approach was chosen over a guided task-based assessment because it was felt it would better reflect how people learn to use systems in the real-world i.e. alone, at differing pace, for different reasons, and in the case of large software utilizing different facets. (Tullis et al., 2004) found that for their survey conditions SUS provided reliable and accurate results at a sample size of 12. However, it is better to have a higher sample size approaching 20 if possible. For this type of pre-existing user survey we are limited by the number of real users rather than the number of willing volunteers as would be the case in a task-based, start from scratch survey.

Table 8.3 depicts the results of the SUS component of the user survey for all fifteen participants. Individual questions are scored from 1-5 and the SUS formula is used to produce the SUS score. Also calculated are the mean, mode, median, and standard deviation for each question and the SUS results. Figure 8.2 provides a bar chart which depicts the SUS scores for each participant. The horizontal line represents the average SUS score, as determined by (Sauro, 2011) from a survey of SUS results, which is 68. In 11 of the 15 surveys performed SIXTH scored above the average, and in 3 cases the results were within 3%.

The average SUS score obtained was 77.8%, 9.8% above the average determined by (Sauro, 2011). In the letter grade mapping determined by (Bangor et al., 2008) this places SIXTH at a B- and well within the identified acceptability range. With regard to the internal consistency of the survey data Cronbach’s Alpha coefficient was utilized to measure this quantity; consequently a value of $\alpha = 0.804027653$ was found. As per (Darren et al., 1999; Kline, 2013) this is a good result, where $\alpha > 0.9$ is deemed excellent. It follows that the survey results can be deemed reliable.

Table 8.4 conveys the self-identified expertise of those surveyed. Given the known breakdown of the participants previously discussed it not surprising that a high level of confidence in Object-Oriented programming (4.5), Programming (4.7), Java (4.6), and Eclipse (4.3) was displayed. Nor would these appear inflated.

8.5.5 Usefulness of Abstractions

This section provides an analysis of the results of Question 16 in which the participants were asked to rank several key abstractions of the middleware on a 5-point Likert scale. As shown in Table 8.6 the results are favourable such that in all cases the mean $\geq 4.0$. The assessment of receivers was the most stable having the lowest co-efficient of variance (CoV), which is denoted as $\text{Var}$ in Table 8.6.

Sensor Nodes and the Sensor Data Format had the the highest CoV. The results with reference to the Sensor Data Format are of the most surprise to the author. This had been considered an area which was much considered was given in this area and indeed a set of three tasks were formulated and written up.
Table 8.3: System Usability Scale (SUS) Results

<table>
<thead>
<tr>
<th>Participant</th>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Q9</th>
<th>Q10</th>
<th>SUS Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>85.0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>77.5</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>85.0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>65.0</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>72.5</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>77.5</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>97.5</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>85.0</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>90.0</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>55.0</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>65.0</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>87.5</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>77.5</td>
</tr>
<tr>
<td>14</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>82.5</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>65.0</td>
</tr>
<tr>
<td>Mean</td>
<td>4.3</td>
<td>2.0</td>
<td>4.2</td>
<td>2.1</td>
<td>4.5</td>
<td>1.7</td>
<td>3.7</td>
<td>1.7</td>
<td>4.1</td>
<td>2.2</td>
<td>77.8</td>
</tr>
<tr>
<td>Median</td>
<td>4.0</td>
<td>2.0</td>
<td>4.0</td>
<td>2.0</td>
<td>5.0</td>
<td>2.0</td>
<td>4.0</td>
<td>2.0</td>
<td>4.0</td>
<td>2.0</td>
<td>77.5</td>
</tr>
<tr>
<td>StdDev</td>
<td>0.7</td>
<td>0.8</td>
<td>0.9</td>
<td>0.8</td>
<td>0.5</td>
<td>0.7</td>
<td>1.2</td>
<td>0.5</td>
<td>0.7</td>
<td>0.6</td>
<td>11.5</td>
</tr>
<tr>
<td>Mode</td>
<td>5.0</td>
<td>2.0</td>
<td>4.0</td>
<td>2.0</td>
<td>5.0</td>
<td>2.0</td>
<td>4.0</td>
<td>2.0</td>
<td>4.0</td>
<td>2.0</td>
<td>85.0</td>
</tr>
</tbody>
</table>

Figure 8.2: SUS Scores for the 15 Practitioner’s Surveyed

stable, usable, and flexible. However, these results are indicative of some satisfaction with the format. Three participants expressed that they perceived the abstraction as unhelpful.

Dissatisfaction expressed with regard to the Tasking Service may be due to a lack of use by the developers surveyed. Their application scenarios may not have required reconfiguration or such was performed directly through the adaptor layer. The Likert scale employed to gather such these observations was not, in this case, augmented with any free-text fields in which the participants could express the reasoning for their judgement.

8.5.6 Free-text Feedback

This section discusses the responses given in free-text to Questions 13, 14, and 17.

*I would be inclined to use SIXTH in appropriate research scenarios.*
Table 8.4: Self-identified Participant Expertise

<table>
<thead>
<tr>
<th>Participant</th>
<th>Java</th>
<th>Eclipse</th>
<th>OSGI</th>
<th>SNs</th>
<th>Generics</th>
<th>Middleware</th>
<th>OOP</th>
<th>Programming</th>
<th>SIXTH</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4.4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3.9</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>4.2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>4.6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3.7</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4.4</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4.3</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3.4</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3.8</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>3.8</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>2.8</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>3.1</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4.6</td>
</tr>
<tr>
<td>14</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4.2</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3.6</td>
</tr>
<tr>
<td>Mean</td>
<td>4.6</td>
<td>4.3</td>
<td>3.2</td>
<td>3.8</td>
<td>3.5</td>
<td>3.3</td>
<td>4.5</td>
<td>4.7</td>
<td>3.5</td>
<td>3.9</td>
</tr>
<tr>
<td>Median</td>
<td>5.0</td>
<td>4.0</td>
<td>3.0</td>
<td>4.0</td>
<td>3.0</td>
<td>3.0</td>
<td>5.0</td>
<td>5.0</td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>StdDev</td>
<td>0.5</td>
<td>0.8</td>
<td>0.9</td>
<td>1.4</td>
<td>1.0</td>
<td>1.0</td>
<td>0.6</td>
<td>0.5</td>
<td>1.1</td>
<td>0.9</td>
</tr>
<tr>
<td>Mode</td>
<td>5.0</td>
<td>4.0</td>
<td>3.0</td>
<td>4.0</td>
<td>3.0</td>
<td>3.0</td>
<td>5.0</td>
<td>5.0</td>
<td>4.0</td>
<td>4.2</td>
</tr>
<tr>
<td>Min</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2.3</td>
</tr>
<tr>
<td>Max</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.0</td>
</tr>
</tbody>
</table>

The likert-scale response to this question was overwhelmingly favourable. As illustrated in Table 8.5 the mean value was 4.4 (stdDev 0.7). Participant 1 cited the extensibility of SIXTH as “ideal for the development and evaluation of a wide range of techniques related to sensing and Cyber-Physical-Social computing”. Participants 3-6 felt that SIXTH was useful in a research context as it provided a springboard for rapid development through abstraction and the removal of some degree of low-level complexity. Several participants did not complete the free-text component of this question, but this does not seem related to any dissatisfaction as evidenced by their answers in Table 8.5

I consider a bespoke application to be superior to utilizing SIXTH.

This question is poorly worded. Consequently this has lead to inconsistent answers such that the level of agreement is stark disagreement with the sentiments offered in free-text. For illustration Participant 6 professed strong agreement with the superiority of bespoke solutions but wrote that “the abstractions and isolation elements [of SIXTH] free up resources that can be devoted to application development”.

This appears to be contradictory and is theorised to be due to the question phraseology.

In what ways could the SIXTH middleware be improved?

Several participants identified a desire for more documentation, code examples, and general improvement of the existing documentation. A need was expressed for several levels of documentation depending on how deeply the user was interacting with the middleware. While, disappointment can be felt that the documentation has failed to meet expectations this identification highlights genuine engagement from the user-base. A more fine-grained survey is proposed to identify what is missing, confusing, unclear etc. in the current documentation.

One participant (4) identified a desire for graphical monitoring tools to improve overall ease of use. In unconscious agreement, and as per Section 5.7.2 such tools have been developed, and should be pushed with the next release. This is described as future work in Section 9.3.6.

Participant 8 expressed general satisfaction however they felt it would be advantageous to bundle SIXTH with an autonomous database bundle. Therein databases tables would be dynamically created or updated in a generic manner. There is no intrinsic fault to the suggestions although it is felt best to separate this from the core, however such a bundle would be a good addition to release as an optional extra. The obvious problem would be the storage of heterogeneous data formats which could be combated by storing
as a blob or through custom tables based on defined modality (which expresses observation components e.g. x,y,z).

One participant (3) identified a desire to have the source code released alongside the current jar file releases, so as to empower tinkering and allow a greater understanding of the inner workings.

Several participants expressed a generic desire for alignment with standards. It is presumed this is a reference to OGC sensor standards such as SensorML and SensorThings. This is envisaged as set of components for intake and exportation of the aforementioned, while this is not in the scope of the thesis contributions it is felt that the extensibility model of SIXTH allows for easy incorporation of such services. For instance the REST framework described in Section 5.7.1 is easily modified to produce SensorML representations via refined translator abstraction. As with previous questions in many instances this free-text option was left blank.

### 8.6 Discussion

This chapter has provided the concluding section of the evaluation component of this thesis which was initiated in Chapter 7. Therein, SIXTH was assessed through usage in several case studies. Herein, SIXTH was evaluated in two distinct dimensions. Section 8.4 describes evaluation through the use of objectives software quality metrics. The middleware compared favourably against its nearest neighbours WSNWare and GSN. SIXTH was shown to have lower average cyclomatic complexity, reduced fields per type, lesser inheritance overheads, lower coupling, and a greater degree of abstractness. All of which are strongly indicative of a maintainable and well-designed code-base.

Section 8.5 describes evaluation through the lens of a practitioners survey underpinned by the industry standard System Usability Scale (SUS). The results were illuminating and generally very favourable, in particular the usability was assessed to be 77.8%, 9.8% above the average determined by (Sauro, 2011) showing a high degree of favourable responses from the user base.
Table 8.6: Usefulness of Key SIXTH Abstractions (Q16 of Bespoke Questionnaire)

<table>
<thead>
<tr>
<th>Participant</th>
<th>Adaptor</th>
<th>Receivers</th>
<th>Nodes</th>
<th>SDF</th>
<th>TS</th>
<th>DB</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4.7</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4.2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4.2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4.5</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>3.8</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4.8</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4.7</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4.3</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>3.3</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3.5</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4.3</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4.2</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>3.8</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.0</td>
</tr>
<tr>
<td>Mean</td>
<td>4.5</td>
<td>4.5</td>
<td>4.0</td>
<td>4.3</td>
<td>4.3</td>
<td>4.3</td>
<td>4.3</td>
</tr>
<tr>
<td>Median</td>
<td>5.0</td>
<td>5.0</td>
<td>4.0</td>
<td>5.0</td>
<td>4.8</td>
<td>4.8</td>
<td>4.8</td>
</tr>
<tr>
<td>StdDev</td>
<td>0.8</td>
<td>0.6</td>
<td>0.9</td>
<td>1.0</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>Var</td>
<td>18.4</td>
<td>14.3</td>
<td>23.1</td>
<td>22.5</td>
<td>19.5</td>
<td>19.5</td>
<td>19.5</td>
</tr>
<tr>
<td>Mode</td>
<td>5.0</td>
<td>5.0</td>
<td>4.0</td>
<td>5.0</td>
<td>4.8</td>
<td>4.8</td>
<td>4.8</td>
</tr>
<tr>
<td>Min</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2.3</td>
</tr>
<tr>
<td>Max</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5.0</td>
</tr>
</tbody>
</table>

These results provide strong evidence for the achievement of the primary thesis objectives as the software has been shown to be of high-quality and the user engagement positive.

This represents the satisfaction of the assessment objectives outlined in Section 1.3.2. The assessment described herein, and in Chapter 7, propels this thesis forward into Chapter 9 which provides a critique of the thesis contributions and identifies future work.
Part V

Coda
9 | Critique & Future Work

“*The covers of this book are too far apart.*”

– Attributed to *Ambrose Bierce*

9.1 Preface

This chapter begins the coda of this thesis by providing a critique of the work which has been thus far presented and an examination of some potential future work of relevance to SIXTH and the broader Sensor Web vision. Section 9.2 provides a critique of several important aspects of SIXTH, the core contribution of this thesis, and Section 9.3 discusses potential improvements to the research presented in the preceding chapters. This future work would present an improvement to the work undertaken to achieve the objectives given in Section 1.3.

9.2 Critique

This section describes a number of identified flaws which exist within SIXTH, and the assessment mechanisms utilized. Chapter 3 identified a desiderata for the development of SIXTH which serves as the functional requirements of the middleware. Before examining the achievement of each desideratum it should be established that the author is of the view that everything could, and should, be improved. However, this is not a particular criticism, rather it is the nature of software development.

9.2.1 Heterogeneity of Source

SIXTH has been demonstrated to support heterogeneous data sources, however, implicit in this desideratum is the question of how well this is achieved and what could be improved. AbstractSensorNetworkAdaptor plays a key role in facilitating the rapid prototyping of new adaptors, deficiencies therein are discussed in Section 9.3.1.

It is felt that the adaptor concept is over emphasised in the SIXTH system architecture at present. For instance an ITaskingMessage has a field to specify the adaptor of concern to the request. Upon reflection, the adaptor should be invisible to the user and such a field ought to refer to the SensorNetwork. Some re-factoring of the internals and minor modifications to the interface layer would be sufficient to make the adaptors more transparent and removed from the conscious viewpoint of the developer.
9.2.2 Abstraction & Uniformity

As with Heterogeneity of Source SIXTH has been demonstrated to provide domain abstraction and a suitable degree of uniformity. It must be assessed to what degree more could be done in this respect. The observations made in Section 9.2.1 are also applicable herein as de-emphasizing the adaptor would present a better abstraction, to wit, more closely mirroring the problem domain.

Uniformity is not feasible as an absolute, though it ought be strived for. For instance, the internals of ITaskingMessage\(<T>\) instances, i.e. instantiations of class T, are heterogeneous. SIXTH espouses the use of either Map or a Frequency object for tasking, however, the reason the mechanism is generic is because it is unforeseeable what manner of reconfiguration will be performed. This is the lesser evil, as rigidity in reconfiguration data type would constrain significantly Ease of Use and the overall goal of Flexibility (see Section 3.9).

9.2.3 Extensibility & Scalability

Chapter 7 demonstrated that SIXTH is extensible. As is thematic of this chapter, it is relevant to remark of the ease with which extension is performed. The user survey described in Section 8.5 was administered to long-term SIXTH users, many of whom have contributed extensions. Consequently, some insights can be gleaned as to where difficulties were encountered. Section 8.5.5 discussed how the participants rated the usefulness of SIXTH abstractions. Therein, the sensor data format was, to the authors surprise, rated negatively in many instances. In part this may be addressed by common data wrappers which are discussed in Section 9.3.4. The tasking service was also the subject of criticism, one factor which hampers its utilization is the need to specify the adaptor for message routing. This would be eliminated by the suggestions made in Section 9.2.1.

The peer-to-peer communication model delivers scalability support in SIXTH. The forthcoming Section 9.3.7 describes how the P2P functionality could be extended to facilitate load balancing and resource sharing.

9.2.4 Ease of Use

Ease of use is the epitome of a cross-cutting issue. The degree to which support for each of the other desiderata are augmented increases ease of use. As a caveat, too much abstraction or absolute uniformity are detrimental to ease of use. This is because, as discussed in Chapter 2, too much abstraction means the developer cannot work as required.

9.2.5 Application Support

This section presents a short exploration of the degree to which SIXTH adequately supports applications development. Arguably the points made in Section 9.2.3 and Section 9.2.1 are of acute relevance herein. From the perspective of the data consuming elements of an application the Data Broker and the object-oriented query model are most relevant to application support. Section 9.3.5 describes the application of functional programming which, it is felt, would make the formation of queries easier and, for lack of a better term, more natural.

9.2.6 Embedded Intelligence Support

Section 7.4.2 described a prototype bridge which linked the agent platform ASTRA to the sensor streams captured by SIXTH. This prototype is incomplete and under-tested. The prototype was further extended
in WAIST (see Section 7.5.1) for the specific needs of that application. To fully assess the agent bridge, it would be necessary to fully integrate with two or more distinct agent platforms. This would be a suitable test of the abstraction provided.

9.2.7 Runtime Reconfiguration

Section 9.3.6 discusses potential revisions to the SIXTH architecture with particular reference to how the advances of the implementation language, Java, can reshape the system in an idiomatic fashion; this is also touched upon in Section 9.3.5. In Section 9.3.6 specific commentary on how the aforementioned may augment reconfiguration is discussed. Section 9.2.2 has previously provided a discussion of some difficult realities of uniform tasking provision.

9.3 Future Work

This section describes a set of future work which ought to be undertaken to enhance the form and function of the SIXTH middleware.

9.3.1 AbstractSensorNetworkAdaptor Over-responsibility

The AbstractSensorNetworkAdaptor could be considered a God Object. Where such is defined as an object which does too much, and has too many assorted responsibilities. Were SIXTH to be re-developed with the benefit of hindsight many of the is-a relationships would be re-organized into has-a relationships. In doing so, modularity would be enhanced and class size simultaneously reduced. A potential down side to such a re-design would be an increase in the size of method chains when calling some object which is now a has-a. Longer method chains can be confusing.

9.3.2 Cognitive Load Assessment

It is prudent to consider a study of the cognitive load (Sweller, 1988) imposed by SIXTH. Given the subdivision of cognitive load (Sweller, 1988) the focus would be on identifying and mitigating the extraneous load. Extraneous load refers to difficulties in problem solving and learning stemming from the manner in which information is conveyed or from the tools given to the learner. The genesis of this consideration stems from the practitioners evaluation, described in Section 8.5. Crucial in mitigating this concern would be an extension of the user guide (see Appendix A) which was requested by many survey participants.

9.3.3 Output of Standards

The incorporation of service-level modules in SIXTH to output sensor descriptors and observations in research domain standards would be beneficial to the adoption of the platform. These would include OGC standards such as SensorML, PUCK, SensorThings and GeoPackage. In particular a SensorML description should be produced which describes a SIXTH deployment including all platforms, sensors, and capabilities. A thorough review of existing standards would be necessary to determine what is overly verbose, what is appropriate with regard to bringing value to the user-base and what is more widely utilized in practice.
9.3.4 Common Package

SIXTH presents very generic representations of sensor resources such as Sensor Data and Sensor Nodes. This is appropriate and a necessity for any middleware targeting a broad domain. However, it has been identified that the addition of a common extension would be useful. Proposed functionality includes common sensor data wrappers e.g. AccelerationSensorData or TemperatureSensorData which would extend the ISensorData interface with modality specific methods. This would be delivered as an OSGi bundle, dependant on the two core bundles, which would contain all the extended domain representations such as AccelerationSensorData. This would be included as a dependency for any bundle utilizing the extended common abstractions to reduce boiler-plate coding within their application.

9.3.5 Functional Programming

In the most recent iteration of the Java programming language, Java 8, functional programming features have been incorporated. In its purest form this paradigm completely avoids program state and mutable data, this is typified by languages such as Haskell (Thompson, [1999]). Aspects of this paradigm, particularly lambda expressions, have long been used in Object-Oriented languages such as Ruby (Flanagan et al., [2008]) providing succinctness and readability. Functional programming has enjoyed a resurgence in interest owing to its applicability in concurrent and event-driven contexts. Many see a blending of object-orientation and functional programming as desirable (Bobrow et al., [1986]; Gabriel et al., [1991]). This section considers the applicability of this paradigm to SIXTH. Listing 9.1 depicts program code for checking the eligibility of a person, a single boolean method is defined. It can be argued that the explicit creation of a class, or alternatively the anonymous class usage, is verbose. Listing 9.2 depicts the same functionality using a lambda expression. This is objectively shorter and increases readability, succinctness, and brevity.

The obvious application of lambda expressions within SIXTH is for the creation of simple queries. This would require some refinement to the general architecture; in this scenario queries would be modified to implement the functional predicate interface from java.util.function.Predicate<T>. In retrofitting lambda expressions into Java a functional interface was defined as specifying exactly one method. A predicate could be used for the construction of IGenericQuery<T> objects, which specify additional methods of convenience. Another possibility would be to drop those methods which specify composition, and split that functionality into a set of factory classes for building complex queries.

Listing 9.1: Eligibility Method sans Functional Programming

```java
class CheckPersonEligibleForSelectiveService implements CheckPerson {
    public boolean test(Person p) {
        return p.gender == Person.Sex.MALE &&
        p.getAge() >= 18 &&
        p.getAge() <= 25;
    }
}
printPersons(roster, new CheckPersonEligibleForSelectiveService());
```

Listing 9.2: Eligibility Method Using Functional Programming

```java
printPersons(
    roster, new
    (Person p) -> p.getGender() == Person.Sex.MALE
    && p.getAge() >= 18
    && p.getAge() <= 25
);```
### 9.3.6 Architecture Revisions

The entire code-base should be updated to Java 8, in some instances updates should be performed to utilize Java 7 abstractions for file I/O. The introduction of `Optional`, intended to replace `null` return values, provides an attractive concept for integration in scenarios where return values are not guaranteed.

Listing 9.3 depicts Java code utilizing `Optional<SoundCard>`. Therein on line 3 if `soundcard` is not null then it will be printed, if it is null nothing will happen. This reduces the common boilerplate conditional code written to deal with these issues. In SIXTH this would be applicable in many areas. For example, a `SensorNode` holds a map of `NodeProperty` instances, if a key was specified to which no property exists then, at present, null is returned. Another candidate is `ITaskingMessage` wherein data members are, by design, sometimes null. The return values of the getters could be swapped with `Optional` wrappers.

#### UI Release

In Section 8.5.6 survey participants expressed the desire for SIXTH UI components. Section 5.7.2 had previously discussed the implementation of such a feature set. However, this has not been released alongside the SIXTH core. Before doing so, the UI will require further testing, integration, and the creation of additional UI elements such as three-dimensional representations of typical WSN modalities e.g. tri-axial acceleration, temperature.

### 9.3.7 Distributed Load Balancing

Currently SIXTH has functionality to connect remote instances together. This feature should be extended to address issues of scalability and distribution more appropriately. Facilities will be available to share computational resources in an effort to tackle load balancing. Resource sharing serves to enable scalability particularly in the arena of cyber sensing wherein several IP addresses can be leveraged to access restricted resources collaboratively. This system will need to be highly fault tolerant to deal with network interruption and similar issues. The use of multiple IP addresses is advantageous in scenarios where IP addresses are blocked for high usage, such is the case with the Twitter API. Applications also exist to leverage SIXTH instance sin other jurisdictions to access content which has been suppressed.

### 9.4 Conclusion

This chapter has reflected on the design and implementation of the SIXTH middleware. Herein deficiencies of the current work have been identified and possible solutions discussed as future work. In the upcoming Chapter 10 this thesis is concluded via a reflection upon the objectives of the work.

---

10 | Conclusions

“Everything has to come to an end, sometime”


10.1 Preface

Having researched, desiderated, designed, developed, evaluated, and critiqued this final chapter, finally, draws this thesis to a conclusion through reflection upon the objectives identified at the beginning of the thesis in Section 1.3. In so doing an assessment of the degree of fulfilment with regard to each objective is provided. Particular attention is paid to the notable work and scholarly contributions of the thesis.

10.2 Feature Comparison

In Chapter 2 an extensive survey of existing middleware implementations was given as guidance for the identification of the desiderata in Chapter 3 and all subsequent work. Therein, Section 2.5.10 examined gateway-side middleware solutions and Table 2.10 provided a concise comparison of such systems across the identified comparison criteria. Table 10.1 expands that table to incorporate SIXTH and facilitate the forthcoming discussion explicitly framed around it.

Table 10.1: Comparison of SIXTH & Gateway-side Middleware

<table>
<thead>
<tr>
<th>Name</th>
<th>Heterogeneity</th>
<th>Reconfiguration</th>
<th>Extensibility</th>
<th>Energy Awareness</th>
<th>Scalability</th>
<th>Intelligence</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSNWare</td>
<td>✔✔✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Xively</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>GSN</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>LSM</td>
<td>✔✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>ThingSpeak</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>IrisNet</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>SenseWeb</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>TinySOA</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Smart-M3</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>MOSDEN</td>
<td>✔✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Hourglass</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Hi-Fi</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>SStreameWare</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Construct</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>SIXTH</td>
<td>✔✔✔✔✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
</tbody>
</table>

SIXTH has a high degree of support for heterogeneous information sources in line with near neighbours GSN and WSNWare. When compared with these in Section 8.4 SIXTH achieved metrics results indicative of a more maintainable and cohesive code-base. Chapter 7 has demonstrated the practical inclusion of diverse information streams in SIXTH fuelling the conclusion that has increased ease of integration for
sensor domain resources. The SIXTH reconfiguration methodology is a much more refined abstraction and encapsulation than those found in MOSDEN, GSN etc. and has proven useful in cyber and physical scenarios which are disparate in their reconfiguration parameters and methodology of resource communication. Extensibility of software is difficult to quantify, certainly Chapter 7 has shown SIXTH to be extensible as the design of Chapter 4 bears out. SIXTH is scored at the same level as MOSDEN in this regard, indeed extensions are provided in a similar fashion in the common Android context. The concept-level extensibility in SIXTH raises it above other systems. SIXTH could not be described as energy aware, as it is a lower-level integration layer SIXTH does not prescribe any network lifetime extension behaviour but rather supports it through an ease of integration with intelligence frameworks.

10.2.1 Primary Objective

Section 1.3.1 identified the primary objective of the thesis. This objective was the delivery of a new generic middleware framework which was strongly underpinned by explicit domain modelling while achieving sufficient balance suitable generality with sufficient specificity. Discussion in Chapter 2, Chapter 4 and Chapter 5 have detailed the domain abstraction methodology of SIXTH, and the absence or insufficiency of such functionality in other comparable middleware offerings. Many approaches, such as WSNWare or OASIS are overly specific to WSN domain and consequently are not permissive of a relaxed view of sensor resources, as advocated for in Section 2.3.1. Other approaches such as GSN, LSM, Xively, or MoSen are unconcerned with providing a domain representational model which adds conceptual difficulty to proper OOP extension, modification, or separation of concerns. SIXTH is an advancement over these pre-existing systems as its design as based upon the best of breed features and judicious combination of metaphors from diverse approaches.

10.3 Review of Secondary Objectives

In this section the objectives previously identified in Section 1.3 are re-visited with a view to assessing the degree to which the work enumerated in the proceeding chapters has delivered upon these crucial objectives. The following sections move through the objectives in-turn.

10.3.1 Objective 1: Review

To provide a comprehensive review and assessment of the state of the art in sensor network middleware and related research areas and in so doing determine the shortcomings of the various classifications of approach

Chapter 1 outlined the vision of the Internet of Things and the Sensor Web and the role to be played by pervasive Wireless Sensor Networks and the inherent need for strong middleware frameworks permissive of all data sources. Subsequently, a through review of existing middleware platforms and the wider research area was the subject of Chapter 2. Therein WSN, Sensor Web, and IoT middleware solutions were analysed and grouped in a taxonomy by the nature of their approach. The taxonomy was compiled through the analysis of many previously constructed within the literature. Taking these as the basis, the survey advances understanding through the addition of new, previously uncatalogued, middleware implementations and, secondly, through explicit discussion of feature-bleed which manifests in the identification of common sets of functionality which are distinct from approaches. The identification of common components arises from dissatisfaction with minimal feature sets categorised as entire approaches within the literature. Additionally this review explicitly focuses upon the abstractions which have
manifested both in-side and outside the network with a view to identifying the most natural abstractions. This middleware analysis yielded conclusions regarding the merits and demerits of all approaches and the unifying abstractions and cross-cutting issues which pervade the sensor middleware domain.

10.3.2 Objective 2: Assessment & Desiderata

To identify the form and associated functionality of a middleware which will address many of the shortcomings identified in 1.

An identification of the shortcomings of pre-existing work is woven into the fabric of Chapter 2. Consequently, Chapter 3 utilized these observations as its basis in identifying a generic middleware desiderata for gateway-side middleware, which had been identified in the conclusion of Chapter 2 as the focus of the core contributions. The desiderata identified are summarized below. For each desideratum, the specific feature(s) of SIXTH which satisfy the requirement are presented also.

- **Heterogeneity of Source:** Make it simple to incorporate data from any source. The Network adaptor component (see Section 4.4.7) is the primary means through which SIXTH delivers on this desideratum.

- **Abstraction & Uniformity:** Abstract complexity and heterogeneity. Provide uniform representations to the user. Make it so the user doesn’t have to care about the things they don’t care about. The overarching focus of Chapter 4 is to carve out a design which facilitates domain abstraction in the a minimalistic, and sufficiently simplistic manner. The evaluation detailed in Chapter 7 and Chapter 8 strongly supports the premise that this was successful.

- **Extensibility & Scalability:** It should be easy to add, or replace, components. The system should scale. The results of the usage analysis from Chapter 7 shows that SIXTH has been extended for a diverse set of use cases.

- **Ease of Use:** Remove all friction. All obstacles and complexity should be intrinsic. The results of the usability survey given in Section 8.5 showed that SIXTH received favourable responses placing the middleware 9.8% above average for usability assessment.

- **Application Support:** Provide strong support to the developer, through callback mechanisms, good domain abstraction etc. Be complete, do not be bloated. The mechanism described in Chapter 4 provide a high-degree of support to developers be they consumption focused (brokers, receiver etc.) or extensive-driven (domain representive objects, override-able defaults).

- **Embedded Intelligence Support:** The same supports need to be given to autonomic management suites; more and more networks will be so managed in future. Section 7.4.2 examined the integration of SIXTH with an Agent Intelligence framework thus illustrating that the same supports can be given to autonomous control systems as are utilized by human controllers.

- **Runtime Reconfiguration:** Where it makes sense - everything should be configurable. The design of SIXTH includes taskable entities as a core component of the architecture (see Section 4.3.5). This concept is applied to all suitable core abstraction such sensors, nodes, networks, and adaptors.

Many such desiderata, explicitly and implicitly, form the basis for the description of middleware implementations. However, this desiderata is unique in its composition to the best of the authors knowledge. **Embedded Intelligence Support** is the desideratum omitted with most frequency; its inclusion derives from the literature review and the opinions formed that in the future, currently being given form by research in the area, networks will be so large, and prevalent, that intelligent autonomous control is a necessity.

10.3.3 Objective 3: Build
To design and implement a new sensor middleware framework, entitled SIXTH, which will accommodate diverse information streams;

With this third objective the discussion moves into the primary contribution of this thesis: SIXTH. SIXTH is, primarily, and originally a gateway-side middleware solution for the rapid-development of sensor-driven applications. Chapter 4 described the development philosophy which is the foundation for the development of SIXTH. Subsequently, Chapter 4 provided a detailed description of the middleware design. The design was made concrete in Chapter 5 which describes the implementation of SIXTH in the Java language.

The remainder of this section discusses each subcomponent of objective 3 as originally given in Section 1.3.

sensor-driven applications to be abstracted from sensing resources;

The SIXTH design features a clear abstraction of the problem domain, to wit, sensor networks. This strongly OO domain representation allows the developer to interact with virtual representations of sensors, sensor nodes, observations, and networks. Explicit domain representation is absent from any prevailing near-neighbour domain middleware such as GSN, TinySOA, LSM, MOSEN, MOSDEN, and Midgard. The author is of the view that this is to their detriment in comprehensibility, and that SIXTH represents an advancement in the application of judicious OO cornerstones, building upon minority systems such as WSNWare which espouse more explicit domain representation.

data fusion from heterogeneous sources;

SIXTH allows data from any source, connected via an adaptor, to pass through the system. This is supported by a permissive and flexible sensor data format. The real-world projects discussed in Chapter 7 have demonstrated the use of SIXTH in fusing multiple informations streams for operations such as HAN regulation and waste management. The modularity of SIXTH allows for dynamic addition of adaptor mechanism during runtime; an advancement over many other implementations. The sensor observation format has been demonstrated to have abstractions advantages over those utilized in recent work.

ease of integration and extensibility;

SIXTH is designed to be highly extensible. This is manifest in the capability to dynamically augment the system by injecting adaptors and services during runtime. Further to this goal Chapter 5 described the strong separation of the interface and the implementation such that default implementations can be substituted. None of the systems reviewed have demonstrated such a strong separation of the system interface from the implementation; this is an important application of best practice to the sensor middleware domain.

support for user applications and intelligent agent management;

Section 4.6 and Section 4.4.6 describe the functionality which allows applications to interface with SIXTH. The efficacy of these concepts has been evaluated through the success of the case studies, enumerated in Chapter 7 and the general satisfaction expressed in response to the practitioners survey detailed in Section 8.5.

(Lillis et al., 2013), summarized in Section 7.4.2 describes the creation of an environment bridge which links SIXTH and its resources with an agent management framework. By utilizing this environment the agents can control sensor resources, in response to the observations conveyed to them.
Section 5.7.2 describes a set of dynamically generated, and updating, GUI elements. These components allow viewing of the sensor network, viewing sensor data in various ways, and controlling individual sensor nodes through their known descriptions. Similar mechanisms were also employed to dynamically create cyber sensors; this is discussed further in Section 7.3.4.

an extensible model for sensor network re-configuration;

Section 4.4.6 and Section 4.5.3 described the reconfiguration components of SIXTH which provide an object-oriented extensible model for reconfiguration of the network. This has been demonstrated as an advancement in the start of the art in comparison with the reconfiguration techniques displayed in several near-neighbour middleware implementations with regard to encapsulation, permissiveness, and ease of use.

class-level extensibility of domain abstractions;

The use of generic programming allows SIXTH abstractions such as queries, brokers, providers, and aggregates to be extended and realised for any other domain object. This is particularly important when one acknowledges as yet unforeseen extensions. Such functionality is not in evidence in other reviewed middleware implementations and constitutes an advancement in the state of the art, which has successfully applied the principles of generics to a sensor domain middleware.

a robust model decomposing device connection and message translation;

Section 4.4.7 described the decomposition of message translation and wrapping through a split responsibility principle. Such responsibilities are not decomposed as architectural features in related research which gives SIXTH an advantage in extensibility, and maintainability.

10.3.4 Objective 4: Study

To evaluate the efficacy of the middleware approach through a series of real-world case studies; each of which assesses the suitability of differing aspects of SIXTH.

Chapter 7 demonstrated the efficacy of SIXTH through its usage in eight diverse scenarios. SIXTH provided a network management framework for waste management (WAIST), and energy consumption regulation applications (ABLE & AUTHENTIC). The framework was extended to provide further support for cyber-sensors and social computing. The aforementioned extension was demonstrated in its usage in a mapping application and in a cyber-journalism scenario tracking wildfires in Tasmania. SIXTH, a gateway middleware, was successfully ported for use in-network on Android device. The systems mentioned above have all been successfully published. This diverse usage is strong evidence of flexibility, ease of use, re-usability, and extensibility.

10.3.5 Objective 5: Evaluate

To undertake an evaluation of system usability and quality through a practitioner’s survey and the application of quantitative code metrics.
Chapter 8 discussed the evaluation of SIXTH in terms of usability. This was conducted through a survey of fifteen SIXTH practitioners who have been utilizing the system for long periods of time in diverse application scenarios. The results were very favourable, the System Usability Scale score achieved was 77.8%, 9.8% above the average determined by (Sauro, 2011). Participants felt strongly that SIXTH provided appropriate domain abstractions (4.4/5). The formulation of these abstractions is unique among related research, providing a complete domain representation compared to other platforms. Participants also felt SIXTH reduced the burden of sensor application development (4.6/5), and indicated strongly that they would continue to utilize the framework (4.6/5). The results of the metrics analysis were also positive, both when considering SIXTH in isolation and in comparison with GSN, and WSNWare. The usage of metrics evaluation is sorely underused in this domain, and this work advances the use of such in the assessment of sensor middleware; laying the ground work for future developers. The figures obtained highlighted the importance of the modularity, and consequent low coupling, provided through the utilization of OSGi.

10.3.6 Objective 6: Frame with Patterns

To design a middleware such that design patterns will be adapted together with best design practice facilitating simplicity, clarity, extensibility, and flexibility.

Section 4.4.7 details the measured application of design patterns to the design of SIXTH. This is important in conveying how the system works to the programming literate. From the extensive literature review presented herein the explicit description of system architecture through design patterns is very rare in the sensor middleware research vein. The application of this lingua franca reflects a scholarly contribution of this work. This application has been successful; as detailed in Chapter 4 and Chapter 5 design patterns and best practice are explicitly used to frame the development and interaction of many of the core functionalities of SIXTH.

10.4 Closing Remarks

This thesis has provided an extensive review of the state of the art in sensor middleware (see Chapter 2), giving rise to the identification of a desiderata for further middleware development (see Chapter 3). This was the impetus for the development of the SIXTH middleware which presents several advancements to the state of the art with regard to sensor-domain middleware including the application of generic programming, judicious framing and utilization of design patterns methodology, crisp, complete domain representation, and the delivery of a highly modular architecture (see Chapter 4-6). The contributions of the thesis were thoroughly evaluated through three distinct strategies: Chapter 7 describes evaluation through usage in seven case studies, subsequently Chapter 8 examined SIXTH via a user survey and objective programming metrics.
Part VI

Appendix
A  |  SIXTH User Guide

A.1 Preface

SIXTH is a Java middleware for sensor applications which promotes modularity, extensibility, scalability, re-usability, and heterogeneity. Through SIXTH, sensor-driven applications are abstracted from sensing networks and information sources. SIXTH enables the dynamic tasking of sensor nodes to suit application demands in near real time. This guide to SIXTH begins with some conceptual grounding in the key middleware elements.

A.2 SIXTH Concepts

![The SIXTH Middleware Architecture](image)

Figure A.1: The SIXTH Middleware Architecture

Within SIXTH a number of core concepts are necessary to represent key abstractions of important Wireless Sensor Network (WSN) components.
Cyber Sensor: A software sensor that enables monitoring of non-physical (typically web-based) environments. In this document, a generic reference to a “sensor” can relate either to a virtual physical or cyber sensor.

Virtual Sensor Nodes: A virtual sensor node (VSN) can be a representation of a physical sensor node in program code or an implementation of a cyber sensing apparatus. For a physical sensor node, this refers to a piece of hardware, which may host numerous individual sensors (e.g. a Sun SPOT contains an accelerometer, temperature, and light sensors). In SIXTH a VSN is a taskable component meaning it can be reconfigured. Examples of typical behaviour modification include changing the sampling frequency of a modality and the duty cycling of a node. Multiple VSN instances may be grouped together to form an aggregate. A aggregate is seen as a tasking group such that sensors can be tasked simultaneously. The virtual sensor node maintains and provides access to the sensors which are linked to it; sensors are represented by virtual sensors.

Virtual Sensors: A virtual sensor (VS) can be either the representation of an individual hardware sensing apparatus such as the accelerometer of a Sun SPOT, or some component of the overall cyber sensor. Each of the sensory capabilities of a sensor node is represented by a VS e.g. one VS for temperature and one for humidity. A VS provides an access point for observations produced by its physical counterpart. In a physical context the VS monitors for potential problems with its physical counterpart such as time-out or failure. If a problem is identified this information is disseminated to appropriate receivers which may take action. This time-out detection is optional and the time-out threshold is configurable, this threshold is coupled with a known sampling frequency to form a buffer for message reception. As with virtual sensor nodes a VS is a taskable component.

Sensor Data: A high degree of heterogeneity is present in the data formats of sensor hardware (as well as supposedly standardised web formats), further compounded by multiple programmers with varying goals extracting different data. Within SIXTH we encapsulate sensed data into a common interface in order for applications to reason about data from different sources more easily. This is a flexible abstraction that ensures access to the data timestamp, the sample type, identification metadata, and the raw data. An individual piece of sensor data can hold either a single value or a collection of linked data. The design is necessary to reduce complexity when data is not useful without related values as with x, y, and z axis acceleration. It also aids in the use of cyber sensors, where multiple pieces of data relate to one entity. For example, retrieving information from a Twitter sensor may result in related data such as the username of the author or geolocation data being acquired also. It is important that these pieces of data are readily linked together in a convenient manner.

Sensor Network Adaptors: A means to collect sensor data from the producer or intermediaries (e.g. from a WSN or a cyber-based source). It acts as a software wrapper around the device-specific code required to gather the data, so that this data can be made available to the middleware. Sensor network adaptors present a uniform interface to heterogeneous sensing platforms, allowing the graceful integration of sensor networks into the middleware. The adaptor layer can be seen in [Figure A.1] The heterogeneous data formats of these platforms are transformed by the adaptors into the SIXTH sensor data standard, enabling the goal of data independence from sensor platform implementation. To limit the workload necessary to create a functional sensor network adaptor default behaviour has been implemented. Therein discovery service registration and maintenance, the creation and management of virtual sensor objects, data access and sensor data forwarding is managed. Due to this provision, implementing a functional adaptor requires only the conversion of received sensed data to the ISensorData standard and a forwarding of that data to the receivers. A fully functional adaptor will implement a tasking method which allows for reconfiguration of the sensor network e.g. set the sampling frequency for temperature at sensor node 01 to once every 10 seconds.

Wireless Sensor Network Adaptors: WSN adaptors connect to a WSN via connected gateway node. Messages received by the gateway node are intercepted and translated to the SIXTH sensor data standard.
For each sensing platform custom code must be written to allow for tasking and message passing in line with SIXTH standards. To date adaptors have been developed for WSN networks running on Sun SPOT, TelosB, Mica2, WaspMote and Tyndall sensor nodes.

**Cyber Adaptors:** A cyber sensing adaptor is developed for each web-based environment being monitored. Cyber sensors are produced according to demand (as opposed to discovered according to availability as physical sensors are). The adaptor is responsible for producing customised cyber sensors based on application needs, articulated through tasking messages. Cyber sensing adaptors provide the same tasking and sensor access mechanisms as their physical sensing counterparts. Using these mechanisms, cyber sensors can support dynamic tasking allowing sensor reconfiguration at runtime according to changing application/user requirements. For example, a sensor designed to monitor Twitter data (tweets) relating to a specific event (e.g., “earthquake”) can be reconfigured to monitor tweets about that event posted from specified locations.

**Message Wrappers:** Message wrapper are used in a sensing adaptor to perform the translation of requests or messages (perhaps originating from a user application) into the format understood by the network connected to the adaptor.

**Data Translators:** A data translator performs a similar function to the message wrapper. A translator is responsible for transforming a received piece of sensor data from the underlying network into one or more sensor data objects, moving the data into the SIXTH standard.

**Tasking Facilities:** The tasking of sensors is critical in any sensor network based middleware. All supported sensor networks can be tasked uniformly but with the flexibility such that certain networks may reject unsupported tasking types. Figure A.2 illustrates the tasking model with a physical WSN. The tasking of sensor nodes is performed via the sending of tasking messages, which are represented in a standard, flexible way. When an attempt is made to task via an adaptor or sensor node the middleware can indicate whether that type of message is supported. For instance where a proposed sampling rate is unsupported by the sensor the adaptor would indicate that this operation is impossible.

![Figure A.2: SIXTH Tasking Model](image)

**Peer to Peer (P2P) Support:** To address scalability concerns, SIXTH has been designed to support a P2P architecture. This system enables the connection of SIXTH instances running on separate machines. Remote resources such as sensor nodes are presented as local to remote peers and are tasked seamlessly across the network. For example, if a SIXTH peer has 3 sensor network adaptors connected to it, the other peers will also see those adaptors, receive their notifications, sensor data, and hold virtual sensor node objects representing those remote nodes. The connection mechanism that allows the peer to determine the location of other peers is a pluggable component. Connection enables can be as simple as text files or constitute a web registry. One option is to use JmDNS, a Java-based a multi-cast domain name service (DNS), for automatic service discovery. JmDNS facilitates the registration of services, service discovery, and facilitates the starting point of communication.

**Receivers:** Receivers are the simplest form of sensor data consumers. They are used to build the notification system described that allows SIXTH components to gain access to sensor data. Receivers fill the subscriber role of the publish/subscribe model wherein the sensors are the publishers, except for tasking receiver for which the adaptor publishes data.
• **Data Receivers** Receivers for the sensor data streams of the sensor network adaptors. These receivers form the foundation of the sensor data notifiers discussed in [Paragraph A.2](#). Data receivers are frequently incorporated into an application directly.

• **Node Receivers** Recipients of notifications regarding nodes and sensors that are disseminated by the sensor network adaptors. Notifications include sensor node registration or sensor time-out. As part of the security policy, the interface features flexibility to allow delivery of sensor, sensor node or adaptor description classes to limit access to re-taskable resources.

• **Adaptor Receivers** Receive notifications regarding the activation and removal of adaptors and their changing status.

• **Event Receivers** Receive updates regarding middleware events and operations.

• **Tasking Receivers** Tasking receivers receive *tasking messages* from the adaptors when a tasking command is issued. The message received allows the receiver to determine the effected sensors and the type of tasking performed. This facility allows applications to make decisions or alter behaviour based on a pending changes in the behaviour of the associated sensor network or individual sensor.

**Discovery Service** The discovery service is a key component of the SIXTH software architecture. It provides access to the virtual sensor nodes and the sensor network adaptors. Additionally, the discovery service is responsible for providing middleware component status notifications to event subscribers.

Access to the discovery service is restricted. Every component that gains access to its services receives a different proxy of the discovery service, in standard proxy pattern fashion. The proxy is aware of the access credentials of the caller and as such may restrict access to certain resources or limit the results of a given action. The role of the discovery service within the middleware is depicted in *Figure A.1* which shows the storage of references to virtual sensor nodes and adaptors, the security layer and the data retention policy.

**Data Retention Policy** The data retention policy enables autonomous control over the retention of the data stored within SIXTH. Primarily this concerns the sensor data stored in the virtual sensors. The default policy removes any sensed data older than one minute from the system, but the decision criteria of a custom retention policy may be much more complex based upon whichever properties are valuable to the user application. The only requirement is that a filtered set of sensor data objects be returned by the policy when invoked. The retention policy is managed by the discovery service.

**Event Notification** The *discovery service* listens for the registration of event subscribers and manages the distribution of updates to all subscribers. Which resources to distribute to the subscribers is decided by querying the access credentials of a subscriber and the security requirements of the resource. For example, when providing notification regarding a new sensing device, an authenticated subscriber receives the reconfigurable representation of that device; this is used to actuate the real sensing apparatus.

**Query Facilities** SIXTH supports the querying of sensor data, sensors, and nodes. Queries play an important role in SIXTH and are closely tied with the notification system. The querying facilities allow for two mechanisms by which data can be queried:

1. Queries can be constructed programmatically using a wealth of already developed Java classes that are provided with the middleware. Custom queries can also be created using the same framework.

2. A SQL-like query language has also been developed, which allows for queries to be created in a more intuitive manner, without the requirement that queries be written by Java programmers.
Query Language  SIXTH provides a SQL-like query language providing a subset of SQL functionality. The design of this query language was motivated by the familiarity of end users with SQL and its syntactic simplicity. Queries are parsed into Java objects and can be utilised for selecting a subset of sensors or, more commonly, for filtering an incoming stream of sensor data. More complex queries can be generated to set sensor sampling frequencies and the lifetime of the query.

SIXTH Query Language Syntax  The query language uses two familiar constructs of SQL; SELECT and WHERE, as well as another clause FOR. SELECT allows the selection of sensing modalities of interest. WHERE allows the specification of propositional logic to filter either to a set of sensors or a stream of data. FOR specifies the life-time of the query. This can be defined either as the number of values or a time period measured in seconds, minutes, or hours.

SELECT Temperature, Humidity
WHERE Temperature > 12 OR Humidity < 25
FOR 100 VALUES

Figure A.3:  Example query using all clauses

The only mandatory clause is SELECT. Modalities specified by SELECT can be appended to specify their individual sampling rates. The sampling rate can be specified in two ways, by specifying if the frequency sampling should occur in Hertz or by providing the amount of time between successive samples. The query shown in figure A.4 sets two separate frequencies on Temperature and Humidity. The first samples every 10 minutes and the second 5 times every second.

SELECT Temperature @ 10 m, Humidity @ 5 Hz

Figure A.4:  Frequency setting using SELECT

The optional WHERE clause allows the filtering of data streams based on arbitrarily complex propositional logic formulae. Figure A.5 illustrates an example of filtering a number of different modalities over several ranges. In this specific case, temperature data is filtered between 12 and 25 degrees and humidity between 25 and 50 percent.

SELECT Temperature, Humidity
WHERE Temperature > 12 AND Temperature < 25 OR Humidity < 50 AND Humidity > 25

Figure A.5:  Example of complex WHERE clause

Finally, the FOR clause allows the user to set the lifetime of the query within the WSN. If this clause is omitted the query persists as long as the application is running. As the clause allows the lifetime to be defined in terms of both time and the number of values received the user may require some knowledge of the WSN. This is because the number of values received is based on the entire network and the user would need to be aware of approximately how many nodes will forward data in response to the query as well as the frequency at which they will do so in order to effectively set the lifetime of the query in terms of values.

A.3  Technologies

OSGi  SIXTH is built on the Open Services Gateway initiative framework (OSGi). OSGi is a modularisation and service platform for Java; OSGi allows for the dynamic introduction and removal of modular functional components. These bundles can be installed, removed, stopped, or updated without requiring a reboot of the system. The OSGi service registry enables the detection of changes in services. OSGi is the basis for the popular Eclipse IDE.
The instructions and screenshots contained in this guide are based on Eclipse Juno (version 4.2.1). Because SIXTH uses OSGi, it is important to install a version of Eclipse that supports OSGi development. From the Eclipse downloads page\(^1\) you should download “Eclipse for RCP and RAP developers”. Installation and running instructions may vary slightly for other versions.

### A.4 Using this document

The section of this document are intended to introduce the reader to the capabilities of SIXTH in a logical order. Where interfaces and classes from the SIXTH core are mentioned in the text, these are hyper-links to the relevant API documentation. The core API documentation, as Javadoc, is located at [http://sixth.ucd.ie/javadoc](http://sixth.ucd.ie/javadoc).

### A.5 Running SIXTH

This chapter illustrates how an installation of SIXTH should be run, and how we can see sensor data being received through the middleware. To do this, we first see how SIXTH bundles can be installed into an Eclipse installation (Section A.5.1). Following this, we discuss how Eclipse Run Configurations are used to define which bundles are loaded and run (Section A.5.2).

Next, we illustrate how a demo involving a dummy sensor adaptor generating random data can be used in conjunction with a dummy receiver that simply prints data updates to the screen (Section A.5.3). Finally, a concrete example of using SIXTH with a real wireless sensor network (namely SunSPOTs) is given in Section A.5.4.

#### A.5.1 Installing SIXTH Bundles

The bundles that make up the SIXTH platform are available for download and can be used without charge for non-commercial use. These are installable through an Eclipse update site located at [http://sixth.ucd.ie/eclipse](http://sixth.ucd.ie/eclipse). In Eclipse, this can be accessed by selecting Help → Install New Software. Enter the URL of the update site in the “Work with” field and hit return. This will display a list of available SIXTH bundles, as illustrated in Figure A.6 (the actual bundle list will vary over time).

Having selected the bundles, click “Next”, and follow the remainder of the installation instructions. For the purposes of the demo in Section A.5.3 you should install each of the following bundles:

- SIXTH Core Implementation (in the “SIXTH Core” section)
- SIXTH Core Interfaces (in the “SIXTH Core” section)
- Dummy Receiver for SIXTH (in the “Receivers” section)
- Dummy Adaptor for SIXTH (in the “Adaptors” section)

Of these, you will require the “SIXTH Core Implementation”, and “SIXTH Core Interfaces” bundles, for any SIXTH deployment\(^2\).

---

2. The separation of the core into interfaces and implementation allows for alternative implementation of the SIXTH core APIs to be used instead
A.5.2 Run Configurations

To run any SIXTH-based application from within Eclipse, you will have to create a Run Configuration. This configures which bundles need to be run and in which order they are loaded into the OSGi environment.

To create a run configuration, select Run → Run Configurations from the menu. This will display a window similar to Figure A.7.

In the left-hand column, double-clicking on “OSGi Framework” will create a new OSGi run configuration (if you do not have an OSGi option, you are probably running the wrong edition of Eclipse: see Paragraph A.3).

Initially, every installed bundle will be selected, so you can click “Deselect All” on the right-hand side to clear this selection. Next, select those bundles that you want to run as part of your application. After this, it is generally advisable to click “Add Required Bundles” on the right-hand side to load any dependencies that your selected bundles may have.

Though not included as a specific dependency, it will be necessary to select an OSGi console whenever you run a SIXTH deployment. When running within Eclipse, it is therefore always advisable to also select the following bundles:
You can give your run configuration a name in the “Name” field, then clicking “Apply” will save the configuration.

A.5.3 Running the Dummy Demo

The dummy demo is intended as a simple way to demonstrate a bare-bones deployment of SIXTH. This consists of a “Dummy Adaptor” that simulates a sensor adaptor and generates randomised data and a “Dummy Receiver” that accesses all sensor data in the system and outputs it to the console.

To run the dummy adaptor and receiver demo you should:

1. Install the SIXTH Core Interfaces, SIXTH Core Implementation, Dummy Adaptor and Dummy Receiver bundles from the update site.

2. Create a run configuration that includes these bundles loaded above (in addition to the bundles required for the OSGi console mentioned in the previous section), namely:
   - ie.ucd.sixth.core.interface
   - ie.ucd.sixth.core.impl
   - ie.ucd.sixth.adaptor.dummy
   - ie.ucd.sixth.receiver.dummy

Running the demo will cause the dummy adaptor to generate data to simulate the creation of sensors and sending of sensor data to the receiver. The dummy receiver will print this data to the console, which is illustrated in Figure A.8. This includes “New” events (indicating a new sensor), “Operational” events (indicating that a newly created sensor has become operational) and data received from the dummy adaptor’s sensors.

The dummy receiver bundle is useful for testing any adaptors that you may wish to use or create. As it outputs all sensor data received by the middleware, it can be used to check whether adaptor data is being added successfully.
A.5.4 SunSPOT Example

This section demonstrates how a SunSPOT sensor network can be connected to a SIXTH deployment to make use of its observations. It is assumed that you have prior experience with deploying packages to SunSPOT sensors via the SunSPOT SDK (downloadable from http://www.sunspotworld.com/SPOTManager/).

The first step required is to download the device-specific package for SunSPOT from http://sixth.ucd.ie/downloads. Note that, unlike most SIXTH downloads, this is not an OSGi bundle and cannot therefore be installed via the SIXTH Eclipse update site. This package should be deployed to each of the individual SunSPOT sensor motes that you plan to use. Instructions on deploying packages to SunSPOT motes can be found in the SunSPOT Developer’s Guide (http://www.sunspotworld.com/docs/Blue/sunspot-developers-guide.pdf).

Next, it is necessary to download and install the requisite bundles: “SunSPOT Adaptor for SIXTH”, “Port Scanner for SIXTH”, and “RXTX Library” (ie.ucd.sixth.adaptor.sunspot, ie.ucd.sixth.scanner.port, and libraries.rxtx respectively). This process is described in Section A.5.1. This will allow the sensor data from the sensors to be available via SIXTH.

To see the data coming from the sensor network, we will use the dummy receiver bundle from the previous section to print all the data being received. To achieve this, firstly connect the SunSPOT access point to your computer using a USB cable. Then create a run configuration consisting of the following bundles (plus the standard bundles that should always be loaded, as discussed in Section A.5.2).

- ie.ucd.sixth.core.impl
- ie.ucd.sixth.core.interfaces
- ie.ucd.sixth.receiver.dummy
- ie.ucd.sixth.scanner.port
- ie.ucd.sixth.adaptor.sunspot
- libraries.rxtx

Running this run configuration, you will see the dummy receiver printing the acceleration and temperature data received from each of the sensors on which you installed the device package.

A.6 Writing an Adaptor

In SIXTH, an adaptor is the mechanism through which sensor data is passed to the middleware. When integrating physical sensors, this is typically done by an adaptor that wraps the hardware-specific code that extracts the raw data from the sensors. For cyber sensors, an adaptor is a wrapper that interacts with programmatically accessible environments (e.g. web APIs, web pages) and produces cyber sensors according to application needs.

The architecture of a SIXTH adaptor is shown in Figure A.9. In this Figure, YourAdaptor is a Java object that implements the adaptor itself. Typically, this is created by extending the AbstractSensorNetworkAdaptor class provided by the SIXTH core. The “data source connection” is the code used to connect to the specific sensor that is being wrapped by the adaptor (e.g. the code required to access data from a SunSPOT sensor mote or data from the Twitter API). The data received from the sensor must be wrapped in such a way as to allow the SIXTH middleware to communicate it to interested applications. This is done by creating SensorData objects to encapsulate this data.

During the lifetime of an adaptor, it may be (re)tasked, which configures its behaviour. This may include an alteration to the frequency at which sensor readings are taken, a change to the type of data being
gathers (for adaptors with access to multiple individual sensors) or other behavioural changes. Retasking takes place in response to a message received from applications using the middleware.

This chapter contains a short tutorial in writing a simple SIXTH adaptor. It omits the hardware-specific aspects and instead concentrates on writing an OSGi bundle that contains an adaptor, registering the adaptor with SIXTH, and making sensor data available. The sample adaptor in this example simply periodically polls the mouse position and makes its co-ordinates available to the middleware as sensor data.

To test this adaptor, you can re-run the dummy adaptor/receiver demo from Section A.5, replacing the dummy adaptor with your new one. The dummy receiver will still output the details of the sensor data it receives.

### A.6.1 Creating the Activator

You should begin by creating a project for a SIXTH bundle; remember to create a dependency on the SIXTH core. Once this is done, it is necessary to write an activator class. You have the option of writing this from scratch, or modifying the default activator class if Eclipse produced one when you created your project. This is the class that the OSGi framework will use when starting and stopping your bundle. For an activator, the key methods are start and stop, which will be called whenever the OSGi framework requests that your bundle start or stop sending sensor data respectively.

A typical activator class for a SIXTH adaptor (named MyAdaptor here) is as follows. Please note that prior to creating the MyAdaptor class, this activator will not yet compile successfully:

There are a number of points to be made about this code:

- On line 10, we maintain a reference to the adaptor being created, so we can stop it gracefully again later. As we will see in the following section, a SIXTH adaptor should implement the IAdaptor interface.

- On line 13, the adaptor’s constructor is invoked, passing the name of the adaptor. In this example, the MyAdaptor class is written in such a way so that a separate thread will be started to gather the sensor data as soon as the adaptor is created.
• The `stop()` method beginning on line 16 will be invoked when the bundle is stopped or unloaded from OSGi. This should cause a graceful shutdown of anything contained in the bundle. In this example, the bundle contains a single adaptor, which is typically stopped using the `unregister()` method (defined in the `IAdaptor` interface).

A.6.2 Creating the Adaptor

An adaptor must implement the `IAdaptor` interface. However, it is not necessary (or indeed recommended) for every adaptor to implement all the interface’s methods directly. Instead, an `AbstractSensorNetworkAdaptor` class is available. This provides the code to register with SIXTH and the discovery service, distribute sensor data, and other common tasks. A concrete adaptor implementation should typically extend this class and contain only the code required for the specific task at hand.

Normally, an adaptor will run continuously within a separate thread, thus a concrete adaptor implementation should also implement the `Runnable` interface. Creating a class that extends `AbstractSensorNetworkAdaptor` and implements the `Runnable` interface results in the code shown in Listing A.2 after inserting stubs for the methods that require implementation.

Listing A.1: SIXTH adaptor Activator

```java
package com.example;
import ie.ucd.sixth.core.adaptor.IAdaptor;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
public class Activator implements BundleActivator {
    private IAdaptor adaptor;
    public void start(BundleContext bundleContext) throws Exception {
        this.adaptor = new MyAdaptor("testAdaptor");
    }
    public void stop(BundleContext bundleContext) throws Exception {
        this.adaptor.unregister();
    }
}
```

Listing A.2: Skeleton for Adaptor Implementation

```
package com.example;
import ie.ucd.sixth.core.adaptor.AbstractSensorNetworkAdaptor;
import ie.ucd.sixth.core.misc.TaskingMessage;
public class AutoGeneratedAdaptor extends AbstractSensorNetworkAdaptor implements Runnable {
    public MyAdaptor(String name) {
        super(name);
    }
    @Override
    public void run() {
        // TODO Auto-generated method stub
    }
    @Override
    public boolean task(TaskingMessage taskingMessage) {
        // TODO Auto-generated method stub
    }
```
In this code, the `run()` method allows the adaptor to be run in a separate thread (as it is contained in the `Runnable` interface). This is where the sensing itself should occur. The `task()` method allows the adaptor to receive and react to tasking messages that request a change of behaviour (e.g. how frequently sensing should occur).

This code can now be extended to allow the adaptor to be run as a thread. Sample code for this is shown in Listing A.3. Typically, this consists of a `while` loop that runs within the `run()` method. The guard for this loop is typically a boolean variable (named `running` in this example) that can be used to stop the adaptor when necessary. This is shown in line 7 of Listing A.3. The `run()` method (beginning on line 27) will continue to iterate until the `running` variable becomes false. This is done via a call to the `unregister()` method (line 16), which as we have seen, will be called from the `stop()` method in the `Activator` class whenever the adaptor is deactivated by the OSGi framework. Setting the `running` variable to false will cause the `while` loop in the `run()` method to exit. Before the method finishes, however, it calls the `unregister()` method of the superclass on line 35. This causes the adaptor to be unregistered from the SIXTH core just prior to the thread stopping (a Java thread stops when the invoked `run()` method exits).

This strategy results in a graceful shutdown whenever this is required. The thread itself is initiated in the adaptor’s constructor on line 11.

Listing A.3: Threaded Adaptor Implementation

```java
package com.example;

import ie.ucd.sixth.core.adaptor.AbstractSensorNetworkAdaptor;

public class MyAdaptor extends AbstractSensorNetworkAdaptor implements Runnable {

    private boolean running = false;
    private int interval = 5000; // default to 5-second updates

    public MyAdaptor( String name ) {
        super( name );
        new Thread( this ).start();
    }

    @Override
    public void unregister() {
        this.running = false;
    }

    @Override
    public boolean task(TaskingMessage message) {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public void run() {
        this.running = true;
        int id = 0;
        while( this.running ) {
            // perform sensing
            super.unregister();
        }
    }
}
```
Listing A.4: Code for Fetching Mouse Position

```java
Point mousePosition = MouseInfo.getPointerInfo().getLocation();
double x = mousePosition.getX();
double y = mousePosition.getY();
```

The remainder of this section will concentrate on the run() method code is inserted to create sensor data and make this available to the middleware. The task() method will be discussed in Section A.6.3.

At this point, you should update the run() method to obtain the sensor data that you wish to send to the middleware. For this example, we include some simple code to obtain the x and y co-ordinates of the mouse pointer, as follows (note that Point in this case refers to the java.awt.Point class, which you will need to import):

Sensor data must be sent to the middleware as an instance of some class that implements the ISensorData interface. As with the adaptor itself, an implementation of this interface has already been provided, named SensorData. The SensorData class has a number of slightly different constructors, depending on whether the data includes one or multiple values, and whether a time is required to be attached to the data also. For this example, since we are sending both an x and a y co-ordinate, we will use the constructor that allows for multiple data values to be set. This constructor requires the following:

- **String nodeType**: This is the type of sensor network this data comes from. The typical value is the type of adaptor that is generating the data (retrievable from the adaptor’s getType() method).
- **int id**: The unique identifier of the sensor node that created the data. Physical sensors provide their own unique identifier such as a MAC address.
- **String modality**: The modality of the data is a description of what the data relates to. A number of default modalities are defined in the SIXTH class, including temperature, humidity, acceleration, GPS data etc. In this example, we will use our own modality string to describe that the data relates to co-ordinates.
- **Map<String,String> values**: A set of name/value pairs containing the sensor data itself.
- **String type**: The type of data being sent. For this, you should generally use one of the constants defined in the ISensorData interface, which include:
  - TYPE_REPLY: Data that is in direct reply to a query that the adaptor received.
  - TYPE_PERIODIC: Data that is generated periodically.
  - TYPE_FREQUENCY: Data of this type is a response to a query as to how frequently sensor data is being gathered. This does not represent actual sensor data but data about the adaptor.

Once the SensorData object has been created, this can be sent to the middleware using the addData() method, which is imported from the AbstractSensorAdaptor that our adaptor extends.

Finally, for the purposes of this example, we cause the thread to sleep for 5 seconds after each time it adds sensor data. The final code for the run() method is now shown in Listing A.5.

**Listing A.5: The run() Method for the Example Adaptor**

```java
@Override
public void run() {
    this.running = true;
    int id = 0;
    while (this.running) {
```
```
Point mousePosition = MouseInfo.getPointerInfo().getLocation();
double x = mousePosition.getX();
double y = mousePosition.getY();

Map<Modality, String> values = new HashMap<Modality, String>();
values.put(new Modality("x"), String.valueOf(x));
values.put(new Modality("y"), String.valueOf(y));

SensorData data = new SensorData(getType(), id, new Modality("coordinates"), values, ISensorData.TYPE_PERIODIC);
addData(data);
```

Testing the Adaptor

You can now see how this operates by reproducing the steps from Section A.5 to run the dummy adaptor/receiver demo, except that you should replace the dummy adaptor with the new adaptor you have created. The dummy receiver will still receive all the data that is sent and print it as its output. An example of the output from a sample run of this configuration is shown in Figure A.10.

Figure A.10: Sample Output from Mouse Position Sensor Adaptor

A.6.3 Tasking your adaptor

The process of (re)tasking an adaptor involves the receipt of tasking messages sent through the middleware. These are messages that request the adaptor to configure its behaviour in some way, or provide additional information. This section shows a simple retasking example that changes the frequency at which data is gathered. A discussion of other forms of tasking follows.

The tasking of sensor nodes is performed via the `TaskingMessage` class. This class allows for an optional argument of type `Object` which allows any Java object to be used as part of the tasking process, adding
an ease of extensibility to the tasking system for complex tasking procedures. When an attempt is made to task via an adaptor or sensor node a boolean is returned to the caller to indicate whether that type of TaskingMessage is supported on that platform. For instance in the case where a sampling rate unsupported by the sensing hardware proposed the adaptor would return false to denote that this operation is impossible.

In the previous section, the adaptor code from Listing A.3 contained a method named task(TaskingMsg msg). This method is invoked by the middleware core whenever a tasking message addressed to this adaptor is sent.

The TaskingMsg class is designed generically so as to place minimal constraints on the nature of tasking messages that can be sent to adaptors. Thus, the message contents can be represented as any type of Java object. For this particular example, we will use a Frequency object (contained in the ie.ucd.sixth.core.sensor.data package in the SIXTH Core Interfaces bundle). As changing the frequency of sensors is a common task, this class is provided to facilitate this. A Frequency object allows for the representation of a frequency either as a value in hertz (i.e. number of iterations per second) or as a millisecond value that represents the delay between iterations.

In the adaptor we created in the previous section, we defined a variable called interval, which is intended to store the number of milliseconds for which the adaptor will sleep between polling for the mouse pointer position. The following task() method will change this value according to the contents of a tasking message. The code for this is shown in Listing A.6.

Listing A.6: The task() Method to Change the Polling Frequency

```
@Override
public boolean task( TaskingMsg message ) {
    switch( message.getCommandType() ) {
        case SETFREQ:
            Frequency freq = (Frequency) message.getTaskingObject();
            this.interval = (int) freq.getFreqMS();
            break;
        default:
            System.err.println( "Unhandled retasking message" );
            return false;
    }
    return true;
}
```

In this code, we assume that the command type from the retasking message will be SETFREQ (used to set the frequency of an adaptor). Other command types are also available (REQUESTVALUE, REQUESTFREQ, RETASK, QUERY).

### A.7 Writing Receivers

A receiver is a component that can receive some form of data from SIXTH. Interfaces are available for the implementation all receiver types, depending on the information required. The principal types of receiver discussed in this chapter are as follows:

**Data Receiver** These are intended to allow applications to receiver data from the sensors themselves (Section A.7.1).

**Adaptor Receiver** These will allow a component to receive data about changes in the states of adaptors, in particular new adaptors coming online and existing adaptors de-registering from the system (Section A.7.3).
Retasking Receiver  During the lifetime of the system, adaptors’ behaviour may be changed by way of retasking messages. Whenever an adaptor has been retasked, it sends a notification to the middleware, which can be seen by retasking receivers. This type of receiver should be used by any component that wants to be aware of retasking that occurs in the system (Section A.7.4).

Query Data Receiver  This is a receiver for specific sensor data that matches a given query. Querying is the principal mechanism by which query data is filtered so as to avoid receiving all sensor data. This is left to Section A.8 for a full discussion of querying.

A.7.1 Data Receivers

A data receiver is a fundamental SIXTH component and one which is most useful to application development. All public sensor data is passed to established data receivers via the sensing adaptors, such data receivers must be registered with the OSGi service bus.

As illustrated in Figure A.11 data that is sent through the SIXTH Core is forwarded to all data receivers that have been registered. Applications that are required to make use of sensor data should incorporate a data receiver.

Figure A.11: Architecture of SIXTH Data Receiver

The creation of a functional Data receiver is accomplished by creating a class which extends the IDataReceiver interface and provides the desired method implementation. Listing A.7 shows code for a simple data receiver.

Listing A.7: Simple Data Receiver

```java
1 import ie.ucd.sixth.core.Credentials;
2 import ie.ucd.sixth.core.receiver.IDataReceiver;
3 import ie.ucd.sixth.core.sensor.data.ISensorData;
4 import java.util.UUID;
5 import org.osgi.framework.BundleContext;
6 import org.osgi.framework.ServiceRegistration;
7 public class MyDataReceiver implements IDataReceiver {
8     private ServiceRegistration<IDataReceiver> registration;
9     private UUID id;
10     public MyDataReceiver(BundleContext context) {
11         id = UUID.randomUUID();
12         registration = context.registerService(IDataReceiver.class, this, null);
13     }
14 }
```

180
overrides the only method from the IDataReceiver interface itself is the receive() method shown on line 21. This will be called every time the middleware core sends sensor data to the data receiver. The data itself is in the form of some implementation of the ISensorData interface, which has previously been discussed in Section A.6.

The remainder of this implementation comes about because the IDataReceiver interface is an extension of the ICredentialedReceiver interface, which is likewise extended by the other receiver types in this chapter also. Thus the discussion of these methods are relevant to the other receiver type discussed in this chapter also.

A.7.2 The ICredentialedReceiver Interface

A receiver of data is required to require some security credentials to the middleware core when requested. This is used in deciding which data receivers will be sent which data. The data receiver shown in Listing A.7 is a typical implementation of a credentialed receiver. Its constructor (beginning on line 15) firstly generates a unique identifier using Java’s UUID class, which is used later to provide the receiver’s credentials. Next it must register with the OSGi service bus (shown on line 17). The arguments to the registerService() method begin with the class of the type of service being registered. In this case, the MyDataReceiver class will only act as a data receiver, as shown. Alternatively, a class that implements numerous receiver interfaces may pass an array of classes here, each representing a receiver interface that has been implemented. The second parameter is the object itself that provides the services being registered, and the third parameter is for configuration properties, which are null in this case.

Calling the registerService() method results in a ServiceRegistration object being returned. We store this in the registration field, so that it can be re-used for deregistration in the unregister() method shown from line 26.

The final method that should be discussed is the getCredentials() method. This should return a Credentials object describing this receiver, so that the middleware can decide which data it is appropriate to sent to it. Line 32 of Listing A.7 shows a typical example of credential generation. The first parameter gives the name of the receiver, the second is its type (generally indicated by the name of the receiver interface it implements), and the third is the receiver’s unique identifier generated in the constructor.

NB: Each of the other receiver types discussed in this chapter must also implement these methods from the ICredentialedReceiver interface.

A.7.3 Adaptor Receivers

An adaptor receiver is intended to monitor the middleware for changes in the state of adaptors. Typically, this refers to new adaptors being added to the middleware and existing adaptors being removed. However,
the interface is designed in such a way as to allow adaptors to raise other types of event also.

Adaptor receivers are created by implementing the `IAdaptorReceiver` interface, which as with the data receiver interface, requires the implementation of `ICredentialedReceiver` also (as discussed in Section A.7.2). An example of a simple adaptor receiver can be seen in Listing A.8.

### Listing A.8: Simple Adaptor Receiver

```java
import ie.ucd.sixth.core.Credentials;
import ie.ucd.sixth.core.adaptor.AdaptorInformation;
import ie.ucd.sixth.core.adaptor.IAdaptor;
import ie.ucd.sixth.core.receiver.IAdaptorReceiver;
import java.util.UUID;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;

public class MyAdaptorReceiver implements IAdaptorReceiver {
    private ServiceRegistration<IAdaptorReceiver> registration;
    private UUID id;

    public MyAdaptorReceiver( BundleContext context ) {
        this.registration = context.registerService( IAdaptorReceiver.class, this, null);
        this.id = UUID.randomUUID();
    }

    @Override
    public Credentials getCredentials() {
        return new Credentials( this.getClass().getName(), IAdaptorReceiver.class.getName(), id);
    }

    @Override
    public void unregister() {
        registration.unregister();
    }

    @Override
    public void receive( AdaptorDescription desc, String event ) {
        if ( event.equals( IAdaptor.NEW_ADAPTOR ) ) {
            System.out.println( "A new adaptor named " + desc.getName() + " appeared" );
        }
        else if ( event.equals( IAdaptor.ADAPTOR_UNREGISTERING ) ) {
            System.out.println( "The adaptor named " + desc.getName() + " unregistered" );
        }
    }
}
```

The principal difference in implementation between the adaptor receiver and the data receiver shown in Section A.7.1 is that the `receive()` method differs as a result of the different type of data being received. In this case, the data sent consists of a `AdaptorDescription` object that describes the adaptor from which the received event originated. This provides methods through which information such the name, type and address of the adaptor can be found. In this example, we simply print the name of the adaptor from which the event was received (shown on lines 34 and 37).

The other argument to the `receive()` method is the status of the adaptor. Although any String is permitted to be sent as a status (to allow for expansion in the future). Two common strings are defined for convenience in the `IAdaptor` interface. `IAdaptor.NEW_ADAPTOR` and `IAdaptor.ADAPTOR_UNREGISTERING` respectively refer to a situation where a new adaptor has become available through the middleware, and where an existing adaptor has unregistered (likely due to its bundle being unloaded from OSGi). Sample code to handle these statuses is shown on lines 33 and 36.
One further feature to note is an alternative method of creating the receiver’s credentials, as shown on line 23. This differs from the example shown in Listing A.7 by finding the class names programmatically rather than specifying them as literal strings.

### A.7.4 Tasking Receivers

An implementation of a retasking receiver implements the `IRetaskingReceiver` interface, and is designed to monitor changes in the behaviour of adaptors. Whenever an adaptor retasks, in response to a tasking message, a notification is received by all retasking receivers. The `IRetaskingReceiver` interface also extends the `ICredentialedReceiver` interface, which is discussed above in Section A.7.2. Examples of implementing the methods in his interface are included in the above sections. Here we will only examine the `receive()` method that relates directly to a retasking receiver.

As illustrated in Listing A.9, the object passed to the `receive()` method is a `TaskingMessage`. This will be a copy of the message that was actually sent to the adaptor that caused it to retask. This type of object has been previously been discussed in Section A.6.3.

#### Listing A.9: Simple Tasking Receiver

```java
@override
public void receive(TaskingMessage msg) {
    System.out.println("Adaptor named " + msg.getAdaptor().getName() + " retasked");
    if (msg.getCommandType().equals(TaskingMessage.COMMANDTYPE.SETFREQ)) {
        System.out.println("Changed frequency to iterate every " + ((Frequency) msg.getTaskingObject()).getFreqMS() + " milliseconds");
    }
}
```

In addition to accessing the message type (seen in Section A.6.3), it is also possible to identify the adaptor that has caused the retasking notification to be sent. This can be done using the `getAdaptor()` method shown on line 3 of Listing A.9. Line 4 begins an example to print details of a particular type of tasking message, namely whenever the data collection frequency of an adaptor is changed.

### A.8 Querying

As discussed in Section A.7, a data receiver by default receives all of the sensor data that is made available to the SIXTH core. For some applications, this is the desired behaviour. However, for complex applications, or for SIXTH deployments that are intended to support multiple applications, it may be desirable to gain access to only a sensor data subset. The querying subsystem is designed for this purpose. Queries are created that match against particular sensor data. When data matching a query is sent to the core by a sensor, the data broker sends it to the application that created the query. In this way, the sensor data streams are filtered to only include suitable data. Queries may use the data itself, its modality or its source in deciding whether or not the data matches.

The following sections discuss how queries can be created and how an application can receive the data it requires.

#### A.8.1 Queries

In SIXTH, a query is simply some object that implements the `IQuery` interface. For common query types, a collection of `IQuery` implementations is available. These are contained in the `ie.ucd.sixth.core.query` package. In some cases, these are simple query implementations that check
if some aspect of the data is as required. For example, Listing A.10 shows a ModalityQuery being used to create a query that matches only temperature data.

Listing A.10: Sample Modality Query for Temperature

```java
IQuery query = new ModalityQuery(Modality.TEMPERATURE);
```

In other situations, queries are combined to create more complex queries. Listing A.11 shows an example of such a complex query being created[3]. This query matches any data that is not temperature or humidity data. This is done by firstly creating two separate ModalityQuery instances for each temperature and humidity. Next, a ConjugateQuery is used to combine these. This type of query combines multiple subqueries and matches data that can be matched against any of these subqueries. This will have the effect of matching any data that is not desired in our example. A NegationQuery then negates this, to create a query that only matches against data that is neither temperature nor humidity data.

Listing A.11: Complex Modality Query to Match Data not Temperature or Humidity

```java
IQuery temperatureQuery = new ModalityQuery(Modality.TEMPERATURE);
IQuery humidityQuery = new ModalityQuery(Modality.HUMIDITY);
IQuery tempOrHumidityQuery = new ConjugateQuery(temperatureQuery, humidityQuery);
IQuery notTempOrHumidityQuery = new NegationQuery(tempOrHumidityQuery);
```

### A.8.2 The Data Broker

To use a particular query, we must register it with the data broker. This is tasked with monitoring the incoming sensor data and only passing on data that matches queries registered by interested parties.

To use the data broker, two objects are required. One is the query itself (as discussed above) and the other is an instance of IQueryDataReceiver to which the sensor data matching the query will be sent.

Gaining access to the data broker requires us to provide credentials. Thus, like the other receivers outlined in Section A.7, the IQueryDataReceiver interface also implements the ICredentialedReceiver interface. This means that for this example it is necessary to implement a `getCredentials()` method as described previously in Section A.7.2.

An example of an implementation of an IQueryDataReceiver is shown in Listing A.12.

Listing A.12: Sample Query Data Receiver

```java
package com.example;

import ie.ucd.sixth.core.databroker.IDataBroker;
import ie.ucd.sixth.core.databroker.IQueryDataReceiver;
import ie.ucd.sixth.core.misc.Credentials;
import ie.ucd.sixth.core.misc.SIXTHMonitor;
import ie.ucd.sixth.core.query.ConjugateQuery;
import ie.ucd.sixth.core.query.IQuery;
import ie.ucd.sixth.core.query.ModalityQuery;
import ie.ucd.sixth.core.query.NegationQuery;
import ie.ucd.sixth.core.sensor.data.ISensorData;
import ie.ucd.sixth.core.sensor.data.Modality;
import java.util.UUID;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;

public class MyQueryDataReceiver implements IQueryDataReceiver {
```

[3] The same query can, of course, be constructed more succinctly.
In this code, we first see the `registration` and `id` attributes on lines 21 and 22. These serve the same function as in the previous chapter (seen in Listing A.7 in the data receiver implementation). These are initialised in the constructor on lines 26 and 27.

Next we see the query, which is constructed in the same way as in the previous section. Following this, it is necessary to register this query with the data broker so that notifications of matching data will be received.

A reference to the data broker is obtained by using the `discovery service`. This service allows any SIXTH component to find other components, whether these are system services like the data broker or other components such as adaptors, receivers, etc. This is shown on line 35. Gaining access to the discovery service requires the provision of credentials, which are returned by the `getCredentials()` method defined on line 55. Once the discovery service has been accessed, it can provide the data broker.

Having gained access to the data broker, the `registerInterest()` method is called on line 36. This indicates that whenever sensor data appears in the system that matches the given query, this implementation of a `IQueryDataReceiver` should be notified.

The `receive()` method on line 40 will be invoked by the data broker whenever relevant sensor data becomes available. It works in a similar way to the `receive()` method that was discussed for a data
receiver in Section A.7 with one exception. Whereas a data receiver will receive all sensor data, a query data receiver only receives data that matches a registered. As there is no limit to the number of queries a query data receiver can register, the data broker will also pass a reference to the specific query that the forwarded data satisfied as a parameter to receive(). Thus the query data receiver can keep track of which data was in response to which query. The second receive() method (beginning on line 31) facilitates the sending of notifications to the query data receiver. Once again, the relevant query is provided as a parameter. The getIdentity() method should allow the identity of the query data receiver to be ascertained at any time. By convention, this should be the classname of the implementing class, as shown on line 50. We note that instead of duplicating this in the getCredentials() method, on line 56, the getIdentity() method is invoked in the generation of the object’s credentials. Finally, the process of deregistering this receiver from the system is implemented in the unregister() method, which follows the example of the data receiver from Section A.7.

A.9 Frequently Asked Questions

A.9.1 How to access the Discovery Service

The discovery service enables access to other SIXTH services and to local and remote sensing resources. Listing A.13 shows how to gain access to the discovery service. To gain access to the discovery service, it is necessary to provide credentials (represented by the Credentials class). Classes will be required by their interface to include a getCredentials() method that will return these credentials. Examples include the the receivers discussed in Section A.7. Specific discussion of the creation of credentials is contained in Section A.7.2. In the following listing, it is assumed that a getCredentials() methods exists in the class containing this code.

Listing A.13: Accessing the Discovery Service

```java
IDiscovery discovery = SIXTHMonitor.getDiscovery(getCredentials());
```

A.9.2 How to utilise the tasking Service

The purpose of the tasking service is to deliver tasking commands to the appropriate adaptor. In Listing A.14 the a TaskingMessage is created using a formatted XML string appropriate for the targeted Twitter adaptor. On line 2 the retasking service reference is retrieved from the discovery service. Line 3 is crucial as the TaskingMessage must be tagged with an AdaptorDescription component, in this case we are conveying that the adaptor is a sensing adaptor, and has the network name "twitter" and implicitly we state it belongs to the local SIXTH node.

Listing A.14: Accessing the Tasking Service

```java
TaskingMessage msg = TaskingMessage.retask(xmlString);
IRetaskingService retaskingService = discovery.getRetaskingService();
msg.setAdaptorDescription(new AdaptorDescription("twitter", AdaptorDescription.SENSOR_ADAPTOR));
retaskingService.task(msg);
```
A.9.3 Using the SIXTH Deployment class

SIXTH can be utilised as a distributed system in which remote sensing resources can be utilised as local resources, for this reason there is a formalised software component: SIXTH deployment. Each SIXTH deployment class represents the sensing and service resources of a SIXTH deployment.

A.9.4 Accessing the local SIXTH deployment

When SIXTH is run in a distributed fashion, each running instance of SIXTH within an OSGi container is described as a “SIXTH deployment”, represented by some implementation of the ISixthDeployment interface.

Listing A.15 shows an example of retrieving a Map of all current ISixthDeployment references and an example of how to acquire a reference to the local deployment (this is the node on which you are working).

Listing A.15: Accessing the SIXTH Deployment Class

```java
IDiscovery discovery = .....
Map<String, ISixthDeployment> sixthDeps = discovery.getSixthNodes();
ISixthDeployment local = discovery.getLocalSixthNode();
```

Listing A.16 details the usage of the SIXTH node class to achieve the same result as in Listing A.14. On line 3 we retrieve the ISensorNetworkAdaptor from the SIXTH node and on line 4 the TaskingMessage is passed to that adaptor.

Listing A.16: Tasking Using the Deployment Class

```java
ISixthDeployment local = discovery.getLocalSixthNode();
TaskingMessage msg = TaskingMessage.retask(xmlString);
ISensorNetworkAdaptor adaptor = local.getSensorAdaptor("twitter");
adaptor.task(msg);
```

Listing A.17 shows the access to the sensing resources using the SIXTH deployment class. These methods all feature companions to retrieve singular entities.

Listing A.17: Accessing Sensor Resources

```java
List<IAdaptor> adaptors = local.getAdaptors();
List<ISensorNetworkAdaptor> sensorAdaptors = local.getSensorAdaptors();
List<IPipeAdaptor> pipeAdaptors = local.getPipeAdaptors();
List<IServiceFactory> serviceFactories = local.getServiceFactories();
```
Demographic Data

Age:
Gender:

Rate your proficiency:

<table>
<thead>
<tr>
<th>Technology</th>
<th>Novice □ □ □ □ □ Expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Novice □ □ □ □ □ Expert</td>
</tr>
<tr>
<td>Eclipse</td>
<td>Novice □ □ □ □ □ Expert</td>
</tr>
<tr>
<td>OSGi</td>
<td>Novice □ □ □ □ □ Expert</td>
</tr>
<tr>
<td>Sensor Networks</td>
<td>Novice □ □ □ □ □ Expert</td>
</tr>
<tr>
<td>Generics</td>
<td>Novice □ □ □ □ □ Expert</td>
</tr>
<tr>
<td>Middleware</td>
<td>Novice □ □ □ □ □ Expert</td>
</tr>
<tr>
<td>Object-Orientation</td>
<td>Novice □ □ □ □ □ Expert</td>
</tr>
<tr>
<td>Programming</td>
<td>Novice □ □ □ □ □ Expert</td>
</tr>
</tbody>
</table>

SIXTH Novice □ □ □ □ □ Expert

System Usability Scale (SUS)

1. I think that I would like to use this system frequently.
   Strongly Disagree □ □ □ □ □ Strongly Agree

2. I found the system unnecessarily complex.
   Strongly Disagree □ □ □ □ □ Strongly Agree

3. I thought the system was easy to use.
   Strongly Disagree □ □ □ □ □ Strongly Agree

4. I think that I would need the support of a technical person to be able to use this system.
   Strongly Disagree □ □ □ □ □ Strongly Agree

5. I found the various functions in this system were well integrated.
   Strongly Disagree □ □ □ □ □ Strongly Agree
6. I thought there was too much inconsistency in this system.
   Strongly Disagree □ □ □ □ □ Strongly Agree

7. I would imagine that most people would learn to use this system very quickly.
   Strongly Disagree □ □ □ □ □ Strongly Agree

8. I found the system very cumbersome to use.
   Strongly Disagree □ □ □ □ □ Strongly Agree

9. I felt very confident using the system.
   Strongly Disagree □ □ □ □ □ Strongly Agree

10. I needed to learn a lot of things before I could get going with this system
    Strongly Disagree □ □ □ □ □ Strongly Agree

Bespoke Questions

11. SIXTH provides appropriate abstractions.
    Strongly Disagree □ □ □ □ □ Strongly Agree

12. SIXTH makes sensor application development easier.
    Strongly Disagree □ □ □ □ □ Strongly Agree

13. I would be inclined to use SIXTH in appropriate research scenarios.
    Strongly Disagree □ □ □ □ □ Strongly Agree

Please Explain

__________________________________________________________________________

__________________________________________________________________________

14. I consider a bespoke application to be superior to utilizing SIXTH.
    Strongly Disagree □ □ □ □ □ Strongly Agree

Please Explain

__________________________________________________________________________

__________________________________________________________________________

15. The SIXTH architecture is easy to understand.
    Strongly Disagree □ □ □ □ □ Strongly Agree

16. Rank these features of the middleware:
Adaptor
Not Useful ☐ ☐ ☐ ☐ ☐ Very Useful

Receivers
Not Useful ☐ ☐ ☐ ☐ ☐ Very Useful

Sensor Nodes
Not Useful ☐ ☐ ☐ ☐ ☐ Very Useful

Sensor Data Format
Not Useful ☐ ☐ ☐ ☐ ☐ Very Useful

Tasking Service
Not Useful ☐ ☐ ☐ ☐ ☐ Very Useful

Data Broker
Not Useful ☐ ☐ ☐ ☐ ☐ Very Useful

17. In what ways could the SIXTH middleware be improved?


191


Brooks Jr., F. P. (1956). “No Silver Bullet: Essence and Accidents of Software Engineering”. In:


Buonadonna, P., J. Hill, and D. Culler (2001). Active message communication for tiny networked sensors.


Halstead, M. H. (1977). “Elements of software science”. In:


Harvey, P., A. Dearle, J. Lewis, and J. Sventek (2012). “Channel and active component abstractions for wsn programming-a language model with operating system support”. In:


Hoff, A. van (2009). “Java implementation of Multicast DNS”. In: URL {http://jmdns. sourceforge. net/}.


202


208


Team, T. (June 2013). Internet Of Things - ThingSpeak. URL: http://www.thingspeak.com


Wolf, P., A. Schmidt, and M. Klein (2008). “SOPRANO - An extensible, open AAL platform for elderly people based on semantical contracts”. In: 3rd Workshop on Artificial Intelligence Techniques for Ambient Intelligence (AITAmI’08), 18th European Conference on Artificial Intelligence (ECAI 08). Patras, Greece.


ZigBee-Alliance (2006). ZigBee specification.

“I have made this longer than usual because I have not had the time to make it shorter.”

– Blaise Pascal, (Pascal, 1920)