This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed:
Sensor Frameworks for Engine Health Management

Submitted by: Jack Alexander Higgs

COPYRIGHT
Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop). This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

Declaration
This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:
Abstract

This project seeks to investigate whether a Multi-Agent System coupled with an array of sensors and wireless technology can be used to gather, store and analyse metrics to determine the health of a combustion engine in an efficient and effective way. A sensor architecture for the AgentScape OS has been extended to use new sensing technologies and a new version of AgentScape, with engine health management showing itself to be a discipline that can be more generic and open source in the future.
CONTENTS

4.8 Software Design ........................................ 54
  4.8.1 Data Service ..................................... 54
  4.8.2 CAN Sensor Service ............................... 56
  4.8.3 WiVib Sensor Service ............................. 57
  4.8.4 Sensor Agents .................................... 58
  4.8.5 User Interface Design .............................. 59
4.9 Methodologies and software for EHMS development ... 61
  4.9.1 Software .......................................... 61

5 Implementation ........................................ 63
  5.1 Code Descriptions .................................... 63
    5.1.1 WiVib Sensor Service ........................... 63
    5.1.2 CAN Sensor Service ............................... 67
    5.1.3 Data Service .................................... 69
    5.1.4 Sensor Agents .................................... 71
  5.2 Build and Deployment ................................ 73
    5.2.1 AgentScape repository ......................... 73
    5.2.2 Maven Projects .................................. 73

6 Testing .................................................. 75
  6.1 Sample Output ....................................... 75

7 Evaluation and Conclusions ............................. 77
  7.1 Development Issues .................................. 77
    7.1.1 Hardware Delays ................................ 77
    7.1.2 WiVib Development ............................... 78
    7.1.3 Developing on AgentScape 2 ..................... 80
  7.2 Future Work .......................................... 81
    7.2.1 EHMS Gui ....................................... 81
## List of Figures

2.1 Example design for an agent based system .......................... 24  
2.2 Division of Labor for CBM-related Equipment health monitoring 25  
2.3 Agent diagnostic process ............................................. 27  
2.4 COMMAS vs. centralized intelligent systems ...................... 28  
2.5 MAMDSS Architecture .................................................. 30  

4.1 An abstract view of the AgentScape environment .................. 39  
4.2 The AgentScape software architecture ............................... 40  
4.3 Sensor Network Platform architecture ............................... 41  
4.4 AgentScape 2 Architecture ............................................ 43  
4.5 Showing how CAN can centralise control ......................... 45  
4.6 Standard format data frame ............................................. 46  
4.7 The WiVib 4/4 Pro Device ............................................ 48  
4.8 Overall Hardware Architecture ....................................... 51  
4.9 Sensor Network Platform Architecture for AgentScape 2 ....... 54  
4.10 Schematic of process flow of an application using PCAN-Light 56  
4.11 User Interface Mock-up / Prototype ............................... 60  

5.1 WiVib Sensor Service UML class diagram .......................... 64  
5.2 JAXB XML to Object binding ......................................... 65  
5.3 CAN Sensor Service UML class diagram ........................... 67
LIST OF FIGURES

5.4 Data Service UML class diagram .................. 70
5.5 Sensor Agent UML class diagram .................. 71
A.1 Server Interface Diagram ............................ 86
A.2 Architecture of Peak CAN USB System .......... 87
A.3 WiVib UML Object Model ........................... 88
Listings

<table>
<thead>
<tr>
<th>Listing</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Binding to a service.</td>
<td>72</td>
</tr>
<tr>
<td>7.1</td>
<td>Creating a CAN Sensor Agent at runtime.</td>
<td>81</td>
</tr>
<tr>
<td>code/wivibservice/SensorService.java</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>code/wivibservice/WiVibSensorService.java</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>code/wivibservice/WiVibSensorServiceImpl.java</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>code/canservice/CANLight.java</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>code/canservice/CanSensorServiceImpl.java</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>code/canservice/CanReadThread.java</td>
<td>106</td>
<td></td>
</tr>
<tr>
<td>code/canservice/CanProcessThread.java</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>code/dataservice/CsvDataServiceImpl.java</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>code/dataservice/Channel.java</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>code/agents/SensorNodeAgent.java</td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>code/agents/WiVibSensorNodeAgent.java</td>
<td>115</td>
<td></td>
</tr>
<tr>
<td>code/agents/CanSensorNodeAgent.java</td>
<td>118</td>
<td></td>
</tr>
<tr>
<td>code/pom.xml</td>
<td>121</td>
<td></td>
</tr>
<tr>
<td>code/wivibservice/sample.xml</td>
<td>122</td>
<td></td>
</tr>
</tbody>
</table>
Acknowledgments

I would like to take this opportunity to thank the individuals who devoted their time and expertise to this project. Without these people this project would not have been possible.

Cubewano

To everyone at Cubewano who assisted me in this project. Special mention to Lakhi Dhatt for ordering the items I needed and Andrew Burtenshaw for his unwavering IT infrastructure support - often in the face of adversity.

Craig Fletcher
For asking me to take on this project and providing the cash, direction and enthusiasm.

Shaun Addy
For advising on accelerometers that satisfy engine requirements.

University of Bath

I would like to thank Dr Julian Padget for agreeing to supervise this project and in particular for the feedback received at various stages which has had a great impact on the outcome of the project.

Finally I would like to thank my family for their input and support and also my house mates Scott, Tim and Ben for continually raising the bar.
Chapter 1

Introduction

1.1 Problem Description

According to Volponi, in the broadest context an Engine Health Management System (EHMS) deals with the monitoring, detection, isolation, predictive trending and accommodation of engine degradation, faults, and failures (Volponi and Wood, 2007). The key drivers for implementing an EHMS include minimising the cost of development, increasing asset availability and readiness and potentially increasing safety. The relative importance of these factors will differ depending on the operating and business environment of the end user.

The application of ‘health’ or ‘condition’ monitoring is intended to increase overall system reliability through the use of intelligent monitoring technologies. A consistent health management philosophy integrates the results from the EHMS for the purposes of optimising operations and maintenance through:

1. Prediction, with confidence bounds, of remaining useful life of critical components.
2. Isolating the root cause of failures after the failure has been observed.

Fault isolation is a critical component to maximizing asset availability.

The majority of research and implementations for engine health systems are centred around gas turbine engines and other such safety critical systems.
The options currently available for small to medium sized engine manufacturers tend to be complete bespoke solution based (from sensor hardware down to the individual software components), and consequently they tend to be very expensive and limiting.

Recent advancements in open source monitoring architectures, in particular Multi-Agent Systems (MAS), may mean a generic framework for EHMS may become feasible. Harman (Harman, Padget and Warnier, 2009) states:

“The ability to reason about their environment and use sensor data in a autonomous self-aware manner makes the agents especially suitable for sensor based applications”

Despite the disparity and closed nature of current EHMS implementations, they all tend to draw from the same mathematical, statistical artificial intelligence methods that are universal and have potential cross-cutting applications (Volponi and Wood, 2007).

1.2 Cubewano

Cubewano, formed in 2007, design and manufacture small, high-quality, high power-to-weight ratio internal combustion engines (mostly of the single rotor Wankel type) incorporating custom Horse Power ranges with multi-fuel capability, including kerosene (also known as JP8, JET-A1 and Heavy Fuel). Most of the engines Cubewano produce are still in the research and development phase and implementing an Engine Health Management System is a priority for both them and their customers. Applications for their engines are almost exclusively for the small professional (primarily military) UAV market.

1.2.1 Resources

The bulk of this project is focused on engines produced by Cubewano. As such they have provided many of the resources needed to complete this project. These include, but are not limited to:

- A premises to conduct all prototyping and engine testing.
• A runnable engine.
  – Engine parts at different stages of degradation.
  – Support from Test Engineers to rebuild, run and instrument engines.
• Hardware to instrument engines.
  – Data Acquisition Unit.
  – Appropriate array of sensors.
• Expertise on the more ‘engineering’ aspects of the project and general guidance/supervision.

1.3 Aims

The aim of this project is to research, design and implement a system for measuring the health of a single-rotor Wankel engine using a Multi-Agent System architecture alongside a specialised array of sensors.

• Research current methods of realising an Engine Health Management System, principally for ground-based testing of internal combustion engines.
• Establish appropriate sensor and data acquisition technology to use with the test engines and MAS architecture.
• Development of automated logging and analysis with a view to making intelligent decisions based on engine health data.

1.4 Outcomes

There are a number of desired outcomes for this project:

• Analysis of the relationship and ways of working between computer scientists and engineers and how we can build reusable software for engineers.
  – Engineers often build throw-away software to complete a specific one-off task etc.
• Improvements to and observations of the multi-layered semantics-ready sensor architecture for a Multi-Agent System implemented in a practical engineering context.

• A prototype sensor framework for an Engine Health Management System for a small internal combustion engine.

• Perhaps working towards a generic framework for implementing an Engine Health Management System for different engines in different environments.

• Observations on data acquisition using Agent technology.
Chapter 2

Literature Survey

2.1 Background

The last two decades have seen a dramatic increase in the applications of Artificial Intelligence (AI) techniques. With the fast evolution of hardware and computing power that began in the 1980s and the diffusion of such hardware, research in AI finally became mainstream, and ideas originating from the 1960s found many applications which were not possible at the time. There are four main types of techniques that have been developed:

1. Formal AI methodologies, which include the relatively new field of Intelligent Agents (IAs)
2. Fuzzy Logic
3. Artificial Neural Networks (ANNs)
4. Generic Algorithms (GAs)

Together, these techniques can be grouped into the single definition of ‘soft computing’. In the last few years, they have been used in a broad range of applications. Intelligent Agents, for example, are widely used in the World Wide Web. Soft computing techniques have become increasingly popular for control systems, where capabilities can bring great advantages in terms of optimisation of functionality.

Due to their conservative nature, engine manufacturers and the aerospace industry have been relatively resilient to the introduction of soft computing methods. A large amount of research is currently being undertaken and
specific applications developed, especially in the military, but wide-scale application in the civilian field (and even R&D within academia) is lagging behind.

An area where soft computing could potentially be exploited with good results is Engine Health Management (EHM). Gas Turbine Engines are driving significant amounts of research focused on increasing reliability and improving functionality and costs. But there are many types of engines and propulsion systems which could benefit from EHM (for the purposes of this project we are looking at rotary engines produced by Cubewano).

This literature survey will examine what Engine Health Management is, the motivations for implementing such a system and some of the functionality of current EHM systems. The author will then focus on soft computing methods, specifically Intelligent Agents, and begin to look at how they can be and are used in an engineering and EHM context.

2.2 Engine Health Management

2.2.1 What is Engine Health Management?

Engine Health Management (EHM) is an umbrella term bringing together many different ideas and concepts. As Larkin, Moawad and Pieluszczak (2004) state:

> The ‘M’ stands for management, which is much more encompassing than monitoring as it has an active, real-time authority during engine operation, while monitoring is typically a passive concept.

A standard automobile, for example, will monitor an engine with a variety of sensors and tell the driver when something is wrong via some kind of interface (usually a dashboard). It is then up to the driver to diagnose what exactly is wrong and take appropriate action. In contrast a modern Formula One car will sense when a driver is about to stall and the ECU (engine control unit) will then decide take control of the engine in order to keep it running. It is this feedback loop and automated decision making that Larkin emphasises.

Volponi and Wood (2007) agrees by defining an Engine Health Management System (EHMS) as something which deals with the monitoring, detection, isolation, predictive trending and accommodation of engine degradation, faults and failures. He goes on to say that over three decades ago the
acronym EHM would have been recognised as meaning Engine Health Monitoring, instead of Management, the former referring to passive observations and the latter to an active pursuit, with dependencies on the former.

Hess (2005) extracts three key components of an EHMS:

- **Enhanced Diagnostics** - the process of determining the state of a component to perform its function(s), high degree of fault detection and fault isolation capability with very low false alarm rate.

- **Prognostics** - actual material condition assessment which includes predicting and determining the useful life and performance life remaining of components by modeling fault progression.

- **Health Management** - is the capability to make intelligent, informed, appropriate decisions about maintenance and logistics actions based on diagnostics/prognostics information, available resources and operational demand.

Whilst certainly important components of any EHMS Volponi and Wood (2007) also draws other characteristics that are typically found in Engine Health applications. There must be a process which collects data, and then turns that raw data into usable (actionable) information. In addition there has to be a way of detecting and isolating faults / degradation to facilitate diagnostics and prognostics. However, in Volponi’s view, experience has shown that EHM needs very from engine-to-engine and even customer-to-customer.

There is no specific one-size-fits-all approach for an EHMS, but rather an array of capabilities that can be assembled to address the specific needs of an end-user.

On the other hand, in his survey of EHMS Tumer and Bajwa (1999) comment that engine monitoring systems have become increasingly standard in the last two decades, in step with advances in computer technology. In fact, Tumer references the first Aircraft Gas Turbine Engine Monitoring System guide which was published by the SAE in 1981. Whilst this may be the case it really depends on that is meant by ‘standard’. Volponi asserts that although EHMS can draw on mathematical, statistical and artificial intelligence methods that
are more generic in nature the majority of solutions are very much engine specific. Larkin et al. (2004) agrees with this assessment, stating:

Although the fundamental role of an EHMS does not change, its various applications will heavily influence its architecture based on key differences in the application.

Ultimately Engine Health Management can be best visualised as a portfolio of capabilities from which building blocks can be drawn to create customised architectures that best meet individual user needs.

2.2.2 Key Drivers

Economics

Engine Health Management is an increasingly important practice for both military and commercial organisations, especially in a challenging economic environment. A presentation by Avtec (2009) highlights the compelling economic reasons for implementing such systems. Commercial and military aviation are under tremendous pressure to reduce maintenance costs.

- World’s airlines spend nearly $40 billion per year (roughly 20% of total budget) on maintenance
- For the military, engine maintenance is around 60% of maintenance budget
- This could mean the inability to meet critical mission objectives at the national security level.

Both Hess (2005) and Larkin et al. (2004) agree that EHM reduce costs and turn around time.

The purpose of an EHMS is to provide a ‘net’ increase in overall mission safety, reliability and performance, while providing at the same time a ‘net’ decrease in overall propulsion system life cycle cost and turn around time.
Volponi and Wood (2007) believe that economics is becoming the prime driver for implementation of EHM systems in commercial, military and power system sectors. Airlines seek to reduce cost of ownership and increase flight safety by reducing in-flight shutdowns, unplanned engine removals and delays and cancellations. The engine manufacturer can obtain benefit from an EHMS by either being able to produce and sell its product at a lower initial cost, or by making their product more attractive by selling the ability to derive additional value through the use of EHM features of their engine product. These factors are important to the military as well. Hess (2005) lists some of the goals of EHM for the Joint Strike Fighter programme. These include, but are not limited to:

- Reduce Maintenance Manpower, spares & repair costs;
- Maximize lead time for maintenance & parts procurement;
- Reduce life cycle costs;
- Eliminate scheduled inspections which reduces aircraft down time.

All sources agree that implementing an EHMS can have major economic benefits, even if the initial cost of introducing EHM is high. Logan (2005) states:

Machinery performance monitoring and health assessment are areas where the exploitation of software agent technology can yield substantial near-term economic benefits to the Navy.

For the US Navy, Logan emphasises that personnel costs comprise over 50% of total operations and support budget and he identifies automating maintenance as something that can produce significant cost reductions.

Safety

Engine safety is now reaching unprecedented levels. Engineers are producing very safe engines of all varieties, but according to Volponi and Wood (2007) there is always a trade in increased weight, cost or reduced performance. He argues that one way to mitigate the penalties of increasing safety is to introduce EHM capabilities. The trades between introducing a new sensor and creating a robust engine structure are often very favorable to the former
solution. Hess (2005) details some of the safety goals of introducing EHM to the Joint Strike Fighter. They include catching potentially catastrophic failures before they occur, enhancing mission reliability and aircraft safety and health reporting to both the pilot and ground crew in real time.

However, Jaw (2005) identifies two practical problems facing Engine Health Management that can directly impact safety. These are too many false alarms can lead to dangerous human intervention, and insufficient sampling and data storage which can lead to gross inaccuracy and negate the purpose of EHM. Tumer and Bajwa (1999) agree, stating:

Routine use of Engine Monitoring Systems poses challenges, mainly due to the abundance and ambiguity of the data to interpret, and due to the high number of false alarms that cause the users’ reluctance to rely on the results.

It is apparent that if implemented badly, Engine Health Management can add little value, and could even be detrimental to safety. Volponi argues that EHMS in of themselves cannot make an engine safer, but their judicious application enables those operating and maintaining the engine to do so with higher margins of safety. Any engine generally provides indication of its health; the question is whether or not anyone is ‘listening’. He goes on to make the point that if an end customer has an idea of ‘perceived safety’ where an EHM system is required to ensure safety, the advantage of having one could become a detriment.

Different Perspectives

There are many different ways an EHMS can benefit individuals in an organisation. Volponi and Wood (2007) list some of these:

- Fleet Manager Perspective
  - Avoid engine damaging events
  - Faster troubleshooting
  - Smaller engine inventory
  - Improve shop planning
- Logistics Manager Perspective
  - Maintenance cost
– Sustaining support
– Support personnel

• Line Maintainer Perspective
  – Event troubleshooting
  – Vibration survey / trim balance
  – Wiring diagnostics

Maintenance

Traditionally, according to Angulo, Gonzalez, Raya and Catala (2007), there have been two maintenance philosophies employed: preventive maintenance that uses statistics to estimate machine behavior, leading to very conservative estimates of the probability of failure; and corrective maintenance, with the machine running until it fails and then being restored to good health; hence maintenance costs are reduced, but unexpected failures may result in longer than expected system down times.

For Angulo, the key driver for Condition-Based Maintenance (CBM - this can fall under the umbrella of EHM) is to accurately detect the current state of engineered systems and their operating environments and use that information for maintenance and prognosis activities. This, it is implied, will create a vast improvement on traditional maintenance philosophies.

One example of this, as cited by Volponi and Wood (2007), is in specific fuel consumption that can be achieved by improved blade tip sealing in the High Compressor and Turbine. Due to various wear mechanisms, the clearance between blade tips and seals erodes over the life of the engine, with a corresponding loss of efficiency. Through engine monitoring techniques, active clearance control systems can recover performance lost due to deterioration. What we see here is adaption to changing performance profiles. This could equally apply, for example, to the effect different fuel types on an engine.

2.2.3 EHM Capabilities and Standard Practices

It is generally accepted that most EHMS consist of two subsystems. These are a ground based diagnostic systems which were the first to evolve an on-board or embedded Engine-Hosted Health Management.
Vibration

Vibration based methods are a very common diagnostic tool for all engine types. Volponi and Wood (2007) describe vibration monitoring systems as the first to evolve:

Starting in the test cells, these systems were very specialised and limited in their scope. These systems were largely comprised of independent programs for vibration monitoring and analysis of high bandwidth (2KHz) accelerometer data.

Using data collected from accelerometers, algorithms have been developed to detect when bearing and gear damage has occurred. Damage in gears and bearings produce changes in the vibration signatures of engines. Over the past 30 years, numerous vibration based algorithms for mechanical component damage detection in transmissions have been developed. Typically, these algorithms are based on some statistical measurement of vibration energy as discussed by Zakrajsek (1989), or by spectral analysis and comparisons of RMS as described by Angelo (1987).

Sensors are mounted as close to the vibration source as possible, as this provides a more distinct signal, but often this is not possible due to space or temperature constraints and they are instead mounted on the engine case. Signal processing techniques such as digital filtering, FFT's are implemented to detect signal spiking, noise and drop-outs.

Jaw (2005) describes a relatively simple method of implementing vibration monitoring:

Vibration data are sampled periodically and keyed to engine spool speeds. Vibration amplitude is then derived from the sampled data through the Discrete Fourier Transform (DFT), and broad-band energy amplitude is extracted from the data through a root-mean-square (RMS) algorithm. The data can be further processed to generate waterfall plots and diagnostic indicators.

Data Acquisition

Whilst collecting and storing data from sensors in a test cell environment is relatively simple, monitoring engine in the field certainly is not. Tumer and
Bajwa (1999) reveals that commercial aircraft conduct continuous on-board monitoring of performance parameters, but only transmit to ground when an exceedance is detected. That is to say that the measured values pass a predefined threshold. Data is not stored for further analysis.

Engine manufacturers are working on determining whether on-board diagnostics for maintenance purposes would fulfill business requirements in a more efficient way, since the problem is due to the inability and cost of transmitting and storing large amounts of data. Certainly for applications where weight is an issue, such as UAVs, data storage and analysis presents a big problem.

### 2.2.4 Future work

Tumer and Bajwa (1999) believe that most engine manufacturers of all types are currently conducting research on the implementation of EHM systems, particularly in the condition-based monitoring space. He argues that the future of EHM systems for commercial aircraft is strongly dependent upon weighing the cost of implementing such a system with their longterm benefits. Jaw (2005) is concerned with the lack of standardised processes and terminology with the EHM field. To remedy this Jaw recommends industry-wide forums. Volponi and Wood (2007) believe that new sensor technologies under development now will enable future EHM systems to provide a more comprehensive assessment of engine conditions.

### 2.3 Multi-Agent Systems

#### 2.3.1 Agents

Generally, according to jiong Gu, feng Dong, jun Wu and Yang (2008), an agent is an entity that has knowledge, intention and capacity, and is capable of doing some specific tasks without or with few instructions from an external source. Wooldridge (2002) offers this definition:

> An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives

However, both Wooldridge (2002) and Weiss (1999) agree that there is no universally accepted view on the definition of agency or indeed of intelligence.
Shoham and Leyton-Brown (2009) goes as far as refusing to give a formal definition, simply stating that Multi-Agent systems are those systems that include multiple autonomous entities with either diverging information or diverging interests, or both. An intelligent agent, according to Wooldridge and Jennings (1995), is one that is capable of flexible autonomous action in order to meet its design objectives, where flexibility means three things:

- Reactivity - An agent can perceive its environment (e.g., the physical world, a user via a graphical user interface, a collection of other agents, or the Internet) and respond in a timely fashion to changes that occur in it.
- Pro-activeness - An agent can not only respond to environmental modifications, but also take spontaneous measures to achieve objectives.
- Social ability - An agent can interact and communicate with other agents or humans via various agent communicational languages.

This view of agent intelligence is also supported by jiong Gu et al. (2008). Whilst most authors agree that there is no single definition of an Agent or Intelligent Agent, it is quite rare to find complete contradictions - most attempts to describe Agents from first principles tend to concur to a certain extent. The debate seems to be philosophical in nature, and not of practical consequence (in this context).

### 2.3.2 Why use Agents for Health Monitoring?

According to IAC and Honeywell (2009) agents add value to prognostics and health management, as suggested in figure 2.1 Agent based systems based upon the general architecture shown in the figure have been built via funded programmes involving high-value assets of the U.S. Department of Defense.
jiong Gu et al. (2008) states that compared with a centralised system, Multi-Agent Systems are more reliable and flexible with faster problem solving ability and less communication. Jeni E. Mangina, McArthur and McDonald (2002) concur, arguing that smaller software entities and software agents with special capabilities are used to interact in a flexible and dynamic way in order to solve problems more efficiently:

Instead of building large applications that model the condition-monitoring activity as a monolithic, isolated process using a restrictive number of data resources and reasoning mechanisms, multi agent systems (MAS) have been introduced from the field of distributed AI, which studies problem solving by multiple knowledge-based processes (Bond and Gassel, 1998).

Logan (2005) motivates the use of agents through suggesting that they are able to automate the bulk of the work necessary to continuously monitor machinery health. They can autonomously perform complex information procession tasks to identify future failures and accurately predict remaining component life. Software agents can be deployed to monitor and analyse hundreds of thousands of data points, while being integrated into existing automation system environments. Figure 2.2 illustrates the division of labor between software agents and humans for a typical health monitoring situation.
2.3.3 Implementing Multi-Agent Systems for EHM and CBM

Shipboard Agent Systems

Logan (2005) describes the technology behind real time diagnostic/prognostic agents aboard T-AO and T-AOE class ships with their main application being the monitoring of diesel and gas turbine engines with remote support via VPN.
Diagnostic Agents for Real-Time Equipment Health Monitoring

A diagnostic agent’s main function is to assess the health on the machinery covered by its diagnostic knowledgebase on a continuous basis. A user can configure diagnostic agents according to specific needs by developing machinery-specific knowledgebases. These knowledgebases could be developed by conducting Failure Mode and Effects Analysis (FMEA), and indeed most automotive part designs are required to be evaluated by the use of FMEA (Teng and Ho, 1995).

The work tasks performed by a diagnostic agent include the following:

- Monitoring all data points in its equipment diagnostic knowledgebase;
- Detect anomalous device behavior;
- Continuously analyse all alarm conditions;
- Alert user upon fault detection;
- Record all diagnostic events to historical logs.

Knowledge is embedded in agents which means staff turnover becomes less of an issue. A feature of diagnostic agents is their real-time assessment of behavioral anomalies of machinery and their probabilistic assessment.

Prognostic Agents for Predictive Maintenance

The purpose of prognostic agents is to predict machinery problems at their earliest stages. Problems can often be identified by analysing historical performance trends. Generation of time to failure metrics can help determine when maintenance should be carried out. Prognostic agents detect subtle equipment performance anomalies for all machinery components associated with their knowledgebases.

The work tasks performed by a prognostic agent include the following:

- Record historical data for all sensor signals related to devices in its knowledgebase
- Conduct machinery performance anomaly and alarm prediction
- Predict future equipment faults based on predicted alarm conditions
- Calculate remaining useful life (time to failure)
• Alert user upon fault prediction
• Record all prognostic events to historical logs

Detailed view of a Diagnostic Agent

![Figure 2.3: Agent diagnostic process](image)

What is interesting about Logan’s approach is the abstraction from the data acquisition and processing as well as the alerting system. As we can see from figure 2.3 all data sits in a real time database. The solution is particularly useful in this case as most navy class ships have deep seated method of data acquisition and the multi-agent system can be deployed on top of that structure.

COMMAS

The COMMAS system as presented by leni E. Mangina et al. (2002) is a condition monitoring multi-agent system for use on machinery in power plants. It has a hierarchical layered approach where the intelligent reasoning is distributed throughout each layer:

• Attribute reasoning agents process the data received, with each agent assigned to monitor each parameter and analyse the data and identify any changes.
• Cross-sensor corroboration agents extract more information from the data and identify any real faults through the correlation of the results.
Meta knowledge reasoning agents assess the health of the plant by collecting all the information from the agents of the previous layers and interpret the analysed output to establish what course of action should be taken and inform the user.

Agents are able to evaluate, interpret and combine information associated with data from the plant being monitored. Again it is assumed that online data is available from the sensors of the plant. Figure 2.4 shows the intelligent agents have been grouped into three categories. There seems to be more emphasis on overall system health as opposed to Logan’s approach where individual components (and engines) are measured, with little mentioned in terms of the bigger picture.

**Multi-agent Based Maintenance Decision Support System (MAMDSS)**

Jiong Gu et al. (2008) describes the MAMDSS system which is primarily designed for use on power plant equipment. The system consists of five main agents:

1. Interface agent - As a way of man-machine interface, the agent replaces
conventional man-machine interface, because it has the capacity of autonomy and learning.

2. Information agent - The information in a system could be managed by this agent because it has the capacity of information retrieval and process.

3. Coordination agent - This agent can solve the problem in cooperation or coordination among agents in system.

4. Task agent - There are many task agents in a system. Each task agent has a kind of function of one function module in the system, such as evaluation, prediction, decision making, etc.; it can carry out the task assigned.

5. Facilitator - Facilitator is a special agent which has the function to allow agents to communicate with each other.

These agents are grouped into three layers: a resource layer; a communications layer and a task layer. The architecture is shown in figure 2.5. The author comments that the strength of the architecture is in its scalability - it is relatively easy to add more agents for specific tasks and distribute agents across a network.
2.4 Conclusion

This survey has given a brief overview of the subjects that require investigation in order to determine the scope and true nature of this project. It is by no means an exhaustive account; the areas of Engine Health Management and Multi-Agent Systems are too massive to condense into a literature review. However, the sources cited provide an excellent way of grappling with the ideas presented in further detail.

It is clear that there are no generic methods of conducting engine health management with solutions tending to be bespoke and all encompassing. In addition, the Multi-Agent Systems we have examined here that do condition
monitoring / prognostic analysis also exhibit few similarities. The lack of similarity should not come as a surprise. Condition monitoring is both a complex and expensive exercise, with very different requirements between different applications (i.e. different engine types, power plants etc).

What is clear however, is that all of these systems have at least one thing in common. They are accessing, using and storing data from physical sensors. This common ground provides an opportunity to work towards a generic architecture to support networks of sensors, or at the very least establish best practices. This is one of the issues the rest of this project will seek to tackle.
Chapter 3

Requirements

3.1 Introduction

Requirements are an important part of any software project. The requirements will be split up into functional and non-functional categories. Throughout, consistent use is made of certain words indicating the necessity of the requirement, defined as:

- **Must** - the system must implement the described functionality in order for the project to be considered a success, i.e. they are mandatory.

- **Should** - such functionality is desirable and useful but is not absolutely required, the system will still be usable without it.

- **May** - such functionality would add value but is not considered important.

Sorting the requirements by these qualifiers leads to an approximate ordering that indicates the main increments of the project. That is, all **must** requirements will be addressed first, followed by **should** and so on.

The literature review, input from employees at Cubewano, supervisor advice, a survey of potential hardware and of course the author’s own judgement have all had significant impact on the formation of these requirements. It should be noted that many of these requirements are somewhat abstract in nature, due to the technical complexity of the problem domain and in the interests of not crossing over into design.
It should also be made clear that the requirements are ambitious for the time available. The product of this project will undoubtedly only fulfill some of them. However, listing ambitious requirements does inform design decisions to take into account long term objectives. Cubewano hope that the resultant prototype of this project will be taken forward in the future to achieve all the requirements.

3.2 Functional Requirements

3.2.1 Engine Health Management System

1. The EHMS **must** provide support for a wireless sensor network.
   
   (a) The EHMS **must** function across networks (wireless and wired Controller Area Network).
   
   (b) There **should** be capacity for multiple wireless nodes.
   
   (c) It **should** be easy to extend to new communication interfaces and sensors.
   
   (d) The EHMS **must** be capable of recovering from physical conditions such as power failure, connection loss etc. The majority of these cases would result in engine shutdown.
   
   (e) The physical components **must** be robust enough to be used within harsh test cell environments.
   
   (f) The EHMS **should** have the facility to change resolution (i.e. the frequency of data samples).

2. The EHMS **must** enhance engine safety by providing early detection of conditions that might progress to critical engine failure.

   (a) The system **must** support the use of an array of sensors to instrument an engine.
   
      i. The system **must** support the processing of data from three accelerometers and an RPM signal.
   
      ii. The system **may** support the processing of data from thermocouples.
   
      iii. The number of sensors being processed at a given time **should** be flexible.
iv. A wireless data acquisition device must interface between the physical sensors and the software components, and should therefore be durable and resistant to interference.

(b) The system must use profiling and automated algorithms (via Agents) to make sense of, store and display data in a meaningful way.

i. Data validation must occur for all sensor data where interference and other such factors can compromise accuracy.

ii. The system must be able to deduce the state of the engine from the sampled data.

3. The EHMS must indicate the state of the engine to the user.

(a) There must be a GUI with appropriate output.

(b) An alarm should sound if the engine is deemed to be in a critical state.

4. The EHMS must take appropriate action depending on the measured state of the engine.

(a) If the engine is determined to be in a critical state the system must cut the ignition.

(b) If the engine is determined to be in a damaged state the system must alert the user. 
*There may also be the option of dropping the RPM down to idle towards the upper bounds of a damaged state.*

(c) If the engine is determined to be in a normal state the system must take no further action.

5. The EHMS must be able to store/log data when tests are being run for later analysis.

(a) The user must be able to specify the frequency that data is collected and saved at.

3.2.2 Sensors

Vibration Sensors

1. Due to the nature of Wankel engines 1-D accelerometers should be used.
2. Accelerometers **must** be small enough to be mounted to a Sonic 30 Wankel engine.

   (a) The sensor, cable and if necessary charge amplifier **should** be very light in weight (as a package).

   (b) The sensor mounts **must** be either adhesive or magnetic.

3. The engine can reach very high temperatures and therefore the accelerometers **must** be able to withstand temperatures of up to 180°C.

4. Accelerometers **should** not drift excessively whilst running.

   (a) The ideal accelerometer **should** exhibit very good stability over a wide temperature range (i.e. in the region of 2 pC/g).

5. Sensors **must** be compatible with wireless data acquisition devices. *Meaning standard electrical connections, power requirements etc.*

**Controller Area Network**

1. The system **must** be able to interface with the Controller Area Network (CAN) protocol used by the Engine Control Unit (ECU).

2. The CAN **must** provide RPM and temperature data.

3. There **must** be the capacity for the CAN interface to control engine RPM and ignition.

   (a) The interface **must** not interfere with or compromise the CAN or engine.

   (b) When the system deems the engine needs to be turned off (for safety reasons), the EHMS CAN interface **must** take precedence over all other CAN interfaces.

**Data Acquisition**

1. A data acquisition (DAQ) device **must** be used to collect data from various physical sensors and transmit to the EHMS.

   (a) The DAQ device **must** be wireless.

      i. The DAQ **should** operate on a universal standard (e.g. WiFi 802.11b/g).
(b) There must be at least inputs/channels for accelerometers.
(c) There may be other inputs (such as normal DC channels), which could be used in the future.

2. The DAQ must provide accurate data in harsh environments - test cells can often exceed 50°C.

3. The DAQ should provide very good resolution; at least 16-bit. *Any less and values could drift quite wildly - especially data from thermocouples and other low voltage sensors.*

### 3.3 Non-Functional Requirements

1. The EHMS must be reliable.

   (a) Hardware failures should be detected by the system, reported to the user and the engine should be shut down unless specified not to by the user.

2. The EHMS should be user friendly.

   (a) Must assume only a basic knowledge of running software.
   
   (b) The system should not ‘panic’ the user into inappropriate action, or allow for misinterpretation of output.

3. The EHMS must be ‘real-time’ and perform to acceptable levels at all times.

   (a) Speed of responses to/from the data service must be within a reasonable time frame.
   
   (b) There may be an option to disable logging to enable enhanced performance if necessary.

4. The AgentScape middleware multi-agent operating system must be used.

   (a) The platform must build on and extend the Middleware platform for Heterogeneous sensor networks as developed by Harman et al.

5. The EHMS should be platform independent.

6. The EHMS must be scalable.
7. All sensors \textbf{should} transmit data at peak rates unless specified by the user.
Chapter 4

Design

4.1 Introduction

To formulate a coherent design for a prototype Engine Health Management System we must first understand the basic concepts of the various technologies that will be used, and also why we are using them in the first place. Following the discussion of concepts the overall architecture and design of the EHMS will be presented.

4.2 Language

The EHMS will extend the work by Harman et al on an AgentScape Sensor Network Platform and as such Java will be the language used to develop extensions to the platform that will culminate in a prototype system. Although choice of language/technology is limited by extending this platform there are some compelling reasons as to why it is a good idea to do so.

The AgentScape OS has been developed using both C++ and Java. The obvious benefit both bring is system independence. However, the option of developing a system to C/C++ is a major bonus due to Cubewano using an ECU system coded in C/C++. Therefore if the EHMS were to be used in the field (engine fitted to an Unmanned Air Vehicle) it would almost certainly be implemented as an extension to ECU software. Support for J2ME is in development which will enhance further the case for use in the field.
4.3 AgentScape

The AgentScape OS is a platform that provides the middleware infrastructure needed to support mobility, security, fault tolerance, distributed resource and service management, and services access, to agent applications (Warnier, Oey, Timmer and Brazier, 2007).

Agents and objects are basic entities in AgentScape. A location is a place in which agents and objects can reside (see figure 4.1). Agents in AgentScape can communicate with other agents by message passing and they can also access services. Services are external software systems accessed by agents hosted by AgentScape. Locations are supported by one or many hosts. Each host in an AgentScape location runs a Host Manager, and one or more Agent Servers.

AgentScape can be thought of in three layers (see figure 4.2): A kernel layer, called the AgentScape Operating System (AOS) providing basic support functionality for the middleware layer. The middleware layer implements the functionality required to support mobile agent applications and agent communication. Middleware layers on separate hosts communicate with each other through the kernel layer using an RPC-mechanism. The application layer is the layer within which the actual agent applications are run. This layer also provides additional higher level application support (Mobach, 2009).

AgentScape’s middleware services implement agent specific functionality. The current set of services, as described by Warnier include (Warnier et al., 2007):

- **Location Manager** - Every location has a Location Manager, which runs on one of the hosts within that location and manages the loca-
tions hosts. Locations typically represent a same single administrative domain.

- **Host Manager** - Each host (typically, one physical machine) runs its own Host Manager to manage the middleware processes for which it is responsible, regulating and guarding access to its resources.

![Figure 4.2: The AgentScape software architecture (Warnier et al., 2007).](image)

- **Agent Server** - An AgentServer provides a programming language specific runtime environment for agents. Each host can run one or more AgentServers to host agents supporting e.g., different programming languages.

- **Web Service Gateway** - The Web Service Gateway enables agents to communicate with web services using the SOAP/XML protocol.

- **Lookup Server** - The Lookup Server keeps track of the current location of agents. Strictly speaking, this service is not part of the AgentScape middleware as it can be run as a stand-alone application. Two versions exist, a centralized, unsecured version and a decentralized secured version.
Agents residing in AgentScape are unaware of the modularity inherent to the middleware. Agents only see the Agent Server, which presents an interface through which agents communicate with other agents, interact with services and migrate to other locations. Agents formulate messages to communicate with other agents and rely on the AgentScape middleware to route the message to the correct location and deliver the message.

### 4.3.1 Multi-Agent Sensor Network

AgentScape’s modular design has made it straightforward to add new services. Harman and others have developed a multi-layered semantics-ready sensor architecture based on the AgentScape middleware. Figure 4.3 shows the architectural layout of the platform.

The Engine Health Management System will extend this architecture to support the types of physical sensor and their appropriate methods of data acquisition needed to determine the health of an engine.

![Figure 4.3: Sensor Network Platform architecture (Harman et al., 2009).](image)
To support sensor networks two services have been added: a generic sensor service and a database service. They provide interfaces for agents.

**Sensor Service**

The sensor service provides access to multiple sensor infrastructures and therefore different types of physical sensors can be used with the sensor service. There is a minimal interface which can be extended if needed.

**Database Service**

The database service provides an interface to access different database implementations. It also enforces policies that agents can define for different sensors. Crucially for the EHMS the agent can chose to use the sensor data directly, for example by promoting it to a rich client, or it can chose to store the sensor data in a database.

**Limitations of implemented architecture**

It can be argued that the sensor platform as implemented by Harman is not a true representation of the architecture designed by Harman et al. The difference between sensor agents and sensor/data services is somewhat confused.

For example the physical sensor agents are inherently aware of the way data is stored using RDF. Triples are created within the sensor agent that are then passed to the RDF data service (which is itself an agent - this is contentious with the architectural design) to be stored. Sensor agents should only interpret and use raw sensor data. Interpreted data should be sent to the data service in a generic form (perhaps using value objects or delimited strings) and the data service should then store/retrieve data in an RDF format.

The reason for the some of the limitations in the implementation of the platform can be attributed to AgentScape. Creating a service based architecture is challenging and everything tends to become an Agent simply out of convenience rather than because it is appropriate. AgentScape 2 offers a way of creating proper user services which is described in the next section.
4.4 AgentScape 2

Since beginning to design and implement this project a new early milestone version of AgentScape has been released. The new version is a significant rewrite of the old AgentScape and crucially separates user Agents and Services into two distinct categories. The new system architecture, which reflects the new distinction can be seen in Figure 4.4.

![AgentScape 2 Architecture](image)

Figure 4.4: AgentScape 2 Architecture (TUDelft, 2010).

A summary of some of the main changes in AgentScape 2 are:

- Split core-AgentScape functionality from extensions
- New Service Architecture
- Changes to the API
- New build-environment: Maven based instead of ant
- New AgentScape website and documentation

The new build environment means it is now easier to build and deploy services and agents in a controlled manner. Maven also introduces dependency management which is crucial for any large scale application. A more thorough and detailed overview of AgentScape version 2 can be viewed on the AgentScape website (TUDelft, 2010).
4.4.1 Service Server

In the context of the Sensor Network Platform the most useful addition to AgentScape 2 is the new service architecture. The service server provides a plug in mechanism for running application services. User level services can be supplied by users of AgentScape, and have the ability to provide services for agent applications.

Agents often run with limited permissions (no direct database access etc). Services, however, can be configured to run with these special permissions. Whereas agents may not be allowed to open connections to a database, this functionality can be delegated to a specific service. Because the services are managed by AgentScape (as well as the interaction between agent and service) this is a mechanism that prevents agents from accessing critical system resources.

4.4.2 Using AgentScape 2

Officially the milestone 1 release is not recommended for regular use. However, it represents a clean break with the old AgentScape and it makes sense to convert the Sensor Network Architecture developed by Harman (Harman et al., 2009) to the new platform sooner rather than later. Doing this early means there is no painful transition to the new AgentScape for the EHMS as it has been developed to it from the start. It also presents an opportunity to test the milestone release and report any problems to the AgentScape developers.

4.5 Controller Area Network

This section describes elements of the Controller Area Network (CAN) protocol and characteristics of a system model that are needed to extract data and exhibit control over a device connected to such a network. CAN will have to be used by the EHMS for both sensing certain metrics and exhibiting control over the engine being monitored. For a complete description of the CAN protocol, see the CAN specification version 2.0 (Bosch, 1991).
4.5.1 Background

A CAN is an asynchronous serial bus network that connects devices, sensors and actuators in a system or sub-system for control applications. This multi-master communication protocol, first developed by Robert Bosch in 1986, was designed for automotive applications needing data rates of up to 1 Mbps and high levels of data integrity. The CAN protocol is now being used beyond automotive applications as a generic embedded communication system for microcontrollers, as well as a standardized communication network for industrial control systems. There is also a documented ISO standard (11898) for CAN.

Cubewano use CAN for all of their engines and engine control units (ECUs). An advantage to this is that ECUs can have a single interface rather than analog and digital inputs to every device in the system as shown in figure 4.5. This decreases overall cost and weight in vehicles such as UAVs.

![Without CAN](image1.png) ![With CAN](image2.png)

**Figure 4.5:** Showing how CAN can centralise control.

4.5.2 CAN protocol and terminology

CAN was designed as a simple and robust broadcast bus capable of operating at speeds of up to 1 Mbit/s. It uses Carrier Sense Multiple Access/Collision Resolution (CSMA/CR) to determine access. Message transfer over CAN is controlled by four different types of **frame**:

- **Data frames**: a frame containing node data for transmission.
- **Remote Transmit Request (RTR) frames**: a frame requesting the transmission of a specific identifier.
• **Overload frames**: a frame transmitted by any node detecting an error.

• **Error frames**: a frame to inject a delay between data and/or remote frame.

The layout of a standard format data frame is shown in figure 4.6.

![Figure 4.6: Standard format data frame.](image)

Each CAN data frame is required to have an 11-bit (standard format) identifier, or a 29-bit (extended format) identifier. The identifier serves two purposes: first it is used as a priority to determine which message will be transmitted next; second the identifier may be used by receivers to filter out messages they are not interested in, and so reduce the load on a receiver’s resources.

### 4.5.3 Priority based arbitration

CAN features an automatic ‘arbitration free’ transmission. A CAN message that is transmitted with highest priority will ‘win’ the arbitration, and the node transmitting the lower priority message will sense this and back off and wait.

The CAN physical layer supports two states termed *dominant* (0) and *recessive* (1). If two or more CAN controllers are transmitting at the same time and at least one of them transmits a ‘0’ then the value on the bus will be a ‘0’. This mechanism is used to control access to the bus and also to signal errors. The CAN protocol calls for nodes to wait until a *bus idle period* is detected before attempting to transmit. If two or more nodes start to transmit at the same time, then by monitoring each bit on the bus, each node can determine if it is transmitting the highest priority message and should continue or if it should stop transmitting and wait for the next bus idle period before trying again.
The arbitration mechanism employed by CAN means that messages are sent as if all the nodes on the network shared a single global priority based queue. This message-based protocol has some distinct advantages. Pazul (Pazul, 1999) states one example:

An automotive airbag sensor can be connected via CAN to a safety system router node only. This router node takes in other safety system information and routes it to all other nodes on the safety system network. Then all the other nodes on the safety system network can receive the latest airbag sensor information from the router at the same time, acknowledge if the message was received properly, and decide whether to utilize this information or discard it.

4.5.4 Structure of a CAN Node

The structure of the CAN protocol can be described in terms of the following layers:

- Application Layer
- Object Layer
  - Message Filtering
  - Message and Status Handling
- Transfer Layer
  The Transfer Layer represents the kernel of the CAN protocol. It presents messages received to the object layer and accepts messages to be transmitted from the object layer. The transfer layer is responsible for bit timing and synchronization, message framing, arbitration, acknowledgment, error detection and signaling, and fault confinement.
    - Fault Confinement
    - Error Detection
    - Message Validation
    - Acknowledgment
    - Arbitration
    - Message Framing
Transfer Rate and Timing
- Information Routing

- Physical Layer
  - Signal Level and Bit Representation
  - Transmission Medium

4.6 WiVib 4/4 Pro

The WiVib-4/4 Pro is a battery or externally powered device that measures vibration and process parameters from machinery and other mechanical systems. It is manufactured by Icon Research Ltd. It provides the necessary resolution, sample rates and sample sizes to enable good levels of monitoring on a Cubewano Wankel engine.

![The WiVib 4/4 Pro Device.](image)

The basic operation is that analog signals are digitised by the on-board analog-to-digital converter (ADC) and stored in internal memory. This data
is then passed over a standard WiFi 802.11 Ethernet network to a host computer for processing and display. The devices can be switched off under software control and woken up by the internal programmable real-time clock. Wakeup Mode, as it is called, helps to maximise battery life.

The WiVib-4/4 Pro (referred to from here as the WiVib) is an eight-channel device allowing four standard ICP type accelerometers to be connected to channels 1 to 4 and four further DC or 4-20mA signals to be connected to channels 5 to 8. Channels 1 to 4 utilise 24-bit high-speed simultaneous acquisition. This enables advanced functions such as orbits and simultaneous triaxial measurements to be performed in addition to standard time traces and spectra. Bearing demodulation (sometimes called enveloping) is available on these channels. Channels 5 to 8 provide slower 16-bit functionality for process measurements. Accelerometers that provide temperature output can be accommodated by connecting the acceleration output to one of the ICP channels (1 to 4) and the temperature output to one of the DC channels (5 to 8).

Anti-aliasing of the incoming signal is performed by continuous time analog pre-filters followed by digital filtering. Incoming data is therefore oversampled, but this process is entirely invisible to the user. The final data stream is buffered in internal memory and then transferred via the wireless Ethernet under control of the host PC.

### 4.6.1 Communication

The sensor is controlled by a set of ‘commands’ sent from a host computer or similar device via the wireless interface. All commands issued are asynchronous and force a reply from the sensor, either as an item of data or as a simple acknowledgment. In this way, the controlling PC should always know the activity and status of each device on its network. This manual describes the command set.

Communication with the WiVib is possible in infrastructure or ad hoc modes, though ad hoc is not recommended as connectivity reliability is not as good. The ex-factory default mode is infrastructure mode.

Communication to the WiVib is also possible via a USB port on the device. Icon provides a configuration utility called WiVibConfig which enables network settings to be adjusted. Status information relating to network communication is available from this port which can be displayed using WiVib-Config.
4.6.2 Wireless Network

Any number of WiVib devices can exist on a network, and each is distinguished by its unique IP address. IP addresses can be allocated statically or can be allocated automatically by DHCP. Each WiVib exists on a network in a similar fashion to any other wireless device.

Network settings are programmed into the WiVib using a configuration utility available from Icon Research. Thereafter the WiVib can be connected to a full network via an access point or similar.

Wireless security using WEP, WPA or WPA2 is available.

4.6.3 WiVib Server

Purpose

For the purposes of this project the WiVib Server (as supplied by Icon Research with the WiVib unit) will be used for communication. The WiVib Server acts as a delegated controlling engine for multiple WiVib devices. The server will accept the incoming connections from the WiVib devices and perform the following control operations:

- Interrogate for settings from new or changed devices
- Synchronise real-time clock and wakeup intervals
- Control the sampling of pre-defined configurations
- Provide a pass-through message routing facility

Protocol

The protocol is based around an XML document. The document may contain elements for one or more WiVib devices, thus allowing a single message to pass configuration details for an entire network. Within the WiVib element are several sections, some of which are optional depending on the use of the document. The names and contents of the sections are:
XML Configuration documents may be transmitted by a TCP stream. The Server will listen for connections made by applications on a specified port number. All connected applications will receive information on WiVib detection and the results of samples taken from these WiVibs. An architectural diagram of the WiVib Server can be seen in the appendix A.1. A sample configuration file can also be viewed in the appendix C.6.

4.7 Hardware Design

A significant amount of time was spent searching for appropriate hardware to enable collection of engine health data. The overall hardware architecture can be seen in figure 4.8.

![Figure 4.8: Overall Hardware Architecture](image)
4.7.1 Accelerometers

As detailed in the requirements section 3.2.2 there are some stringent demands for the accelerometers due to the size and temperature constraints of Cubewano’s range of engines. A number of options and suppliers were looked at. It became immediately clear a charge output ICP accelerometer.

All piezo-electric accelerometers work by measuring the charge generated by a crystal that is being compressed or shear loaded by a mass influenced by acceleration. In most applications this high impedance charge output is converted to a low impedance voltage output by the use of integral electronics. However in some applications integral electronics are not appropriate such as high temperature as is the case with Cubewano’s engines. Charge output accelerometers are self-generating and typically have amplifying electronics mounted several feet away from the local heat source. In this case the WiVib provides the amplification needed.

Shaun Addy from Cubewano advised on the selection of a charge output ICP accelerometer that would fit on the engine. The Dytran 3225E1 accelerometer fulfills all requirements in order to be used on a Cubewano engine. The schematic for this accelerometer can be found in the appendix.

4.7.2 WiVib

The WiVib 4/4 Pro can handle up to four ICP accelerometers and sample them simultaneously. One of the advantages the WiVib has over other wireless data acquisition boards is the support for charge amplifier accelerometers out of the box. This is in fact the main reason it has been chosen. All the signal conditioning and additional functionality it delivers is a bonus. Other solutions were looked at (MicroStrain base stations for example) but they did not provide for charge amplifier accelerometers and nor were they particularly cost effective. The WiVib adheres to the data acquisition requirements outlined in section 3.2.2.

The WiVib can be configured to connect to any wireless network. In this case Cubewano have their own wireless network and a machine has been chosen to host the WiVib Server. A 10v - 30v power supply is also needed to power the unit.

---

1ICP is the trademarked PCB name for IEPE accelerometers. It stands for 'Integrated circuit-piezo electric'.
4.7.3 CAN

Cubewano use a CAN implementation developed by a German company called Peak. The system connects to an Engine Control Unit (ECU) via a USB hardware interface. The ECU will transmit RPM data (and other metrics) over the CAN automatically when the engine is running, and will listen for messages sent by user applications over CAN for controlling both the throttle and the ignition.

4.7.4 Measuring Engine Health

The motivation behind the hardware architecture is to allow for three accelerometers to be placed on an engine and be read simultaneously along with an RPM signal provided by the CAN BUS interface.

As the health of an engine (example becomes unbalanced, bearing surfaces degrade) deteriorates so the amplitude of the vibration the engine generates increases. By monitoring the vibration levels mapped to RPM over a long period of time this gradual deterioration of the health of the engine can be assessed until the vibration levels get to a point where the engine needs to be taken out of service and overhauled.

Analysis of the frequency content of the engine vibration signal will indicate not only that the health of the engine has deteriorated but also root causes can be attributed to the problem (this is dependent of good accelerometer placement).
4.8 Software Design

Figure 4.9: Sensor Network Platform Architecture for AgentScape 2

Figure 4.9 shows the overall system design, originally conceived by Harman et al, updated for AgentScape 2. The EHMS will use this architecture to attain raw data from sensors via sensor services, interpret and use the data via sensor agents and then log the interpreted data via a data service for future analysis. The following sections describe the different parts of this design.

4.8.1 Data Service

The data service interface and implementation provide the mechanism to store interpreted sensor data. Sensor node agents will bind to a data service upon startup and therefore there must be a data service running at the same location/host as the sensor agents.
Storage Structure

The data service implemented for the EHMS will be a simple CSV logger. The reason for this is Cubewano (and engineers in general) use the CSV format to store all their engine test data (both CAN and general data acquisition). It allows for historical test data to be loaded in Microsoft Excel for easy analysis. Certain important clients also expect analysis data to be in Excel / CSV format. The CSV files generated by the data service will be named by the start time (e.g. 99/99/9999 00:00:00:000.csv).

The data being fed into the data service will need to be in a generic format and a generic value object (a simple POJO) will provide a way of encapsulating interpreted sensor data. The value object will need to provide at the very least the name, unit and value of the particular metric being sent to the data service. The data service can then cycle through the collection of received value objects and log the data.

Although beyond the scope of this project, an RDF storage mechanism could easily be implemented/transferred to the AgentScape 2 sensor architecture.

Data Dictionary

The EHMS data that will need to be logged:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Field Unit</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>HH:mm:ss:SSS</td>
<td>Date</td>
</tr>
<tr>
<td>Revs Per Minute</td>
<td>RPM</td>
<td>int</td>
</tr>
<tr>
<td>Accel 1 Average</td>
<td>g</td>
<td>float</td>
</tr>
<tr>
<td>Accel 1 Peak</td>
<td>g</td>
<td>float</td>
</tr>
<tr>
<td>Accel 2 Average</td>
<td>g</td>
<td>float</td>
</tr>
<tr>
<td>Accel 2 Peak</td>
<td>g</td>
<td>float</td>
</tr>
<tr>
<td>Accel 3 Average</td>
<td>g</td>
<td>float</td>
</tr>
<tr>
<td>Accel 3 Peak</td>
<td>g</td>
<td>float</td>
</tr>
</tbody>
</table>

Any number of fields could be saved to or read from CSV. The above table contains the mandatory fields. Metrics such as end plate temperature, fuel flow or ambient humidity could be added if needed. Raw vibration is in a trace format (array of floats) and each trace could be very large indeed. It is therefore sensible to log the average and peak values from each trace and these can be used for analysis (e.g. outside bounds, gradual increase in average vibration etc).
4.8.2 CAN Sensor Service

The CAN system architecture can be seen in the appendix [A.2]. The USB hardware is controlled by DLL drivers (labeled PCAN-Light). Fortunately, Peak supply a JNI interface that enables native calls to the methods within the DLL drivers from applications developed in Java.

<table>
<thead>
<tr>
<th>Start</th>
<th>Operate</th>
<th>Finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init function to establish connection between non-Plug-and-Play hardware and application software.</td>
<td>Basic functions and additional functions.</td>
<td>Call Close function.</td>
</tr>
<tr>
<td>Init function to establish connection between Plug-and-Play hardware and application software.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.10:** Schematic of process flow of an application using PCAN-Light

Schematic process flow

1. **Start**
   There are two different ways to connect a PCAN-Light client with a PCAN Hardware:
   
   (a) Start with Non-Plug-and-Play hardware:
       Call the Init function for Non-Plug-and-Play hardware
   
   (b) Start with Plug-and-Play hardware:
       Call the Init function for Plug-and-Play hardware.

2. **Operate**
   After a successful start CAN messages can be read and written in addition to calling other functions provided.

3. **Finish**
   Call the Close function. This process disconnect the application software from PCAN hardware.
Figure 4.10 shows that the CAN Service will call the init function on the CAN-Light API before beginning to continuously poll the CAN for messages. Once the service is no longer needed the close function needs to be called to disconnect the application from the PCAN hardware.

Instead of using a circular ring buffer or similar as is used in many data acquisition applications (Sommerville, 2004), the CAN Service will process messages and maintain a map of message IDs to message data. This means that the consumer of the service can take a snapshot of all the latest messages currently being received by the CAN Service. Message data will be available to consumers in hexadecimal format. No interpretation of CAN data happens in the CAN Service. This means it can potentially be used by many different systems.

The CAN Service will also provide functionality to act as an actuator for engine control. Target RPM and ignition values will be passed to the service and it will create and send a message to be picked up by the ECU. These messages should have high priority to fulfill CAN requirements 3.2.2.

4.8.3 WiVib Sensor Service

There are two ways to communicate with a WiVib device over a wireless network. One can do it directly over a TCP link, or through a TCP link to the WiVib Server which then communicates with the WiVib. To implement the service using direct WiVib communication an algorithm such as the following would be used:

Listen for device to connect
Interrogate device to ascertain its identity
Send Configuration statements
Loop
  Send Acquire Command
  Send Data Request Command
  Receive Data Blocks
End Loop

This essentially turns the WiVib Service into a fully fledged server. The implications of this are costly error handling and testing to ensure all bases are covered as described in the WiVib command protocol. On the fly configuration changes and storage of specific configurations also present further complexity.
The WiVib Server on the other hand is controlled by sending XML configuration files via a TCP socket. XML files (to the same schema) are continuously sent back with measurements and samples filled in. This means it is easy to quickly change the type of measurements we want to take. A set of Java objects will have to be generated that conform to the server’s XML schema. These objects can then be made available for polling by consumers of the WiVib service.

4.8.4 Sensor Agents

Following from the work of Harman (Harman et al., 2009), all sensor agents can only be assigned to one sensor on the network. This means that all sensor agents must only bind themselves to one sensor service. Harman asserts that this increases fault tolerance due faults in individual sensors / services not disrupting the functionality of other agents in the system. This assertion continues to hold.

Upon startup, all sensor agents should bind to a data service implementation. If no service can be found, or the user does not want to log sensor data then that should be accommodated (i.e. the agent should still perform all other functions). The user can specify the logging frequency, mainly for performance considerations (this is more critical for RDF and relational database services).

Data expiry times as implemented by Harman do not make sense in an EHMS context since data is either saved permanently or it is not saved at all. However, data expiry could easily be bolted on the the data service and/or sensor agents if RDF is implemented in AgentScape 2.

CAN Sensor Agent

The CAN Sensor Agent should bind to the CAN Sensor Service upon startup and start continuously polling the service. Once data is being received the agent should interpret the data. The algorithm for interpreting the CAN feed in an EHMS context is:

\[
\text{If Message ID equals 1536 Then} \\
\quad \text{RPM} = \text{ConvertHexToFloat(Byte1 + Byte2)} \\
\quad \text{Ignition} = \text{ConvertHexToFloat(Byte3 + Byte4)} \\
\text{End If}
\]
All other message IDs are thrown out since we are not interested in the data they contain. Once data is interpreted the agent should send the data to the bound data service for logging via a generic value object. In addition the agent should send the data to the user interface (also an agent) using standard agent messaging protocols.

The agent should also send RPM and ignition data to the CAN Service if commanded to do so in order to exhibit control the engine.

**WiVib Sensor Agent**

The WiVib Sensor Agent will bind to the WiVib Sensor Service upon startup. Once the agent confirms a WiVib has been connected the agent will continuously poll for data. It will extract/interpret the data from WiVib data structures and relay traces to the data service and user interface.

### 4.8.5 User Interface Design

Due to the dynamic nature of user interfaces, textual descriptions and diagrams are not good enough for expressing user interface requirements. Evolutionary or exploratory prototyping with end-user involvement is the only practical way to design and develop graphical user interfaces for software systems (Sommerville, 2004).

Figure 4.11 shows an initial mock-up of the user interface for the EHMS. The interface allows the user to set many types of configurations/measurements in a hierarchical collection of engines and measurement points. This means a user can quickly setup and save test configurations for particular engine types. Two further tabs should be added that show live trace vibration data and historical trending data whilst tests are running.

The user interface will be an agent to enable efficient communication with sensor agents.

---

1Note this is only for Cubewano’s CAN - other networks will have different message configurations
Figure 4.11: User Interface Mock-up / Prototype
4.9 Methodologies and software for EHMS development

An iterative design/development methodology will be employed to implement the EHMS. A prototype system will be developed from the ground up across, with initial phases devoted to attaining levels of very basic operation (e.g. simply injecting a working agent). Then further functionality and complexity can keep on being added.

Iterative processes are particularly important given that AgentScape 2 is a new platform and as new features are released it is important that the Sensor Network Platform can take advantage of them. One such feature in the pipeline is allowing multiple implementations of one service interface. This may subtly change the sensor network architecture.

4.9.1 Software

- **Eclipse IDE** - The Integrated Development Environment that will be used for developing the Java code and maintaining the build and deployment (Maven based). Eclipse can also be configured to run Unit tests and launch AgentScape / inject agents.

- **NetBeans IDE** - NetBeans has very good visual editors for quick graphical user interface prototyping.

- **AgentScape 2** - The Sensor architecture uses AgentScape 2. The distribution contains a number of jar files, which contain AgentScape core libraries, dependencies and agents. The default hierarchy looks like this:

  |- agentscape
     |- agents
     |- lib
        |- system
        |- shared [optional]
        |- services [optional]

- **WiVib Server** - Acts as a delegated controlling engine for multiple WiVib devices.
• **Subversion** - A subversion server has been setup on one of Cubewano’s application servers to provide source control. This provides a backup and a history of development.
Chapter 5

Implementation

This chapter provides details of the implementation and decisions that were made to produce a prototype system. It also includes observations on developing with AgentScape 2 and specific aspects of the new Maven build architecture which greatly decreases the complexity of deploying and running agents.

5.1 Code Descriptions

5.1.1 WiVib Sensor Service

The WiVib Sensor Service, like all other AgentScape services, performs invocations it receives from service clients (usually agents). The connections and method invocations between client and service are mediated by the AgentScape middleware, which can be used for regular access to services enabling interaction between client and service to be as transparent as possible. Transparency between client and service interactions help to make the implementation of the services as convenient as possible.
WiVibSensorService Interface

The first component of the WiVib Service is the service interface. The interface is implemented by the service and used by the client to enable service binding. It is simply a regular Java interface, except that it extends from the `org.iids.aos.service.Service` interface and a `SensorService` interface. The `Service` interface is empty and only marks an interface for use as an interaction medium between the client and service. The `SensorService` interface simply forces implementation of some common methods between sensor services.
All methods and variables defined in the service interface are made available to both the service implementation and the client. The methods made available in this case can be seen in appendix C.1.2. Because this interface is to be shared between the client and the service, this interface is separated from the agent and service in a shared library. The AgentScape integration plugin is used to automatically copy the JAR to the AgentScape shared library directory `lib/shared`.

**WiVib Object Model**

Also contained within the service interface is the WiVib object model. This model is based on an XML schema\(^1\) provided with the WiVib. JAXB was used to generate the model from the schema.

JAXB is an acronym derived from Java Architecture for XML Binding. It constitutes a convenient framework for processing XML documents, providing significant benefits as compared to previously available methods such as the one following the Document Object Model. In fact, other methods of encapsulating the WiVib XML Schema were considered and partially implemented including JDOM and JAXP. These proved too cumbersome for generating and parsing complex and deeply hierarchical XML as defined the WiVib Server schema.

\[\text{Figure 5.2: JAXB XML to Object binding (Ort and Mehta, 2003).}\]

JAXB simplifies access to an XML document from a Java program by presenting the XML document to the program in a Java format, as shown in figure 5.2. Appendix A.3 shows the structure of the generated Java objects. These Java objects can then be marshaled to XML which conforms to the WiVib schema and unmarshalled back to Java objects. The model allows the client to change the WiVib configuration whilst providing access to trace and trending data.

\(^1\)Schema: A schema is an XML specification that governs the allowable components of an XML document and the relationships between the components.
WiVibSensorServiceImpl

Now that the service interface has been specified, we can implement the actual service. The service is a separate maven module within the EHMS project and thus is contained within a separate JAR. It is a single Java class which completely implements the service functionality by implementing the interface WiVibSensorService. The service also extends from the AgentScape class AbstractDefaultService which handles the initialisation of the service and the interaction between client and service.

The service is begins with the initialise() method which sets a WiVib Server name and port number from parameters passed down from the client. JAXBContext and Unmarshaller objects are also instantiated at the initialisation stage. Testing has shown that if the Context and Unmarshaller objects are created within the method they are used it can add almost a second on to the processing of XML. Since the processing of XML happens continuously then that would be a serious detriment to performance.

The service then begins the ServerLinkWatchdog() thread. This thread continuously loops looking to open a socket connection to the WiVib Server. Once this has been achieved it creates a ServerLinkHandler() thread, sets a connected flag and continues looping until the connected flag becomes false and the process starts again.

The ServerLinkHandler() thread is the main part of the service. It creates an input stream from the socket and continuously loops, reading incoming data as it becomes available. When data has been read into a string, it is then processed by unmarshalling it to a WiVibNetwork object which contains a list of WiVib objects. This list is available for a client to poll. If the WiVib Server is configured to collect samples (via the XML) then the thread will be receiving data from the stream almost continuously.

The service also provides methods to send an XML configuration to the WiVib server. This allows for different types of samples to be taken. An XML file or a marshaled WiVibNetwork object is sent to the WiVib server using the socket output stream. All data has to be sent in an ASCII encoding.

The code for the WiVibSensorServiceImpl can be viewed in appendix C.1.3. There were many problems encountered whilst implementing the service and these are discussed in the evaluation section.
5.1.2 CAN Sensor Service

The CAN Sensor Service performs invocations it receives from service clients (usually agents). Figure 5.3 shows class relationships and hierarchy.

Figure 5.3: CAN Sensor Service UML class diagram.
CanSensorServiceInterface

The CAN Sensor Service Interface module consists of two packages. The sensorservice.can package contains the service interface. As in the WiVib Service Interface this implements SensorService and Service and provides the method specifications to be used by both the client and the service implementation.

The peak.canlight package contains the JNI interface to the CAN-Light device drivers and the associated data structures required. The main class for using the CAN-Light API is CANLight. There are several steps to go through in order to use the API properly:

1. Create a new CANLight object:
   ```java
   can = new CANLight();
   (or use an constructor which returns an initialized object)
   ```

2. Call the initializeAPI method passing the hardware parameter which you want use. Example to initialize a USB hardware (channel 1):
   ```java
   can.initializeAPI(CANLight.PCAN_USB_1CH);
   ```

3. If necessary call setFilter(), resetFilter() or resetClient(). Example:
   ```java
   can.resetClient();
   can.resetFilter();
   can.setFilter(0x000,0x123,CANLight.TYPE_ST);
   ```

4. Call the read or write method. Example:
   ```java
   CANMessage msg;
   msg = can.read();
   can.write(msg);
   (do not forget to check if msg is null after calling the read method)
   ```

5. At the end call the close method. Example:
   ```java
   can.close();
   ```
CanSensorServiceImpl

CanSensorServiceImpl is a Java class which completely implements the CAN Service functionality by implementing the interface CanSensorService. The service also extends from the AgentScape class AbstractDefaultService in the same way the WiVib Service does. The code can be found in appendix C.2.2.

The client should invoke the connect() method to initialise the CAN-Light API before other methods are called. To account for this all other methods check for initialisation before proceeding. The startCanRead() method is called which starts two threads: CanReadThread and CanProcessThread.

The CanReadThread loops to continuously poll the Can hardware device using the canLight.readEx native method. Once a message is read a map of message IDs to CanDataRow which is sent by the main service object is updated with the new values. CanDataRow is a data structure which includes the message ID, the CAN data and a CAN timestamp. The rate at which data is acquired by the this thread is very rapid indeed.

The CanProcessThread continuously processes the data received by the CanReadThread thread. It outputs a map of message ID to a list of strings containing the CAN data. The message data itself is converted into Hexadecimal format for easy processing by the client. This processing means that the client can poll and receive a map of standard types and does not have to import specific CAN data types.

The CanSensorServiceImpl provides methods for a client to read processed raw data and close the CAN-Light API. It also enables the client to send CAN messages by invoking the sendCanMessage method and including the desired message ID, length and byte data as parameters.

5.1.3 Data Service

As figure 5.4 shows, the Data Service is relatively simple. It performs invocations it receives from service clients (usually agents).
Figure 5.4: Data Service UML class diagram.

DataService Interface

As in the other service interface the DataService implements SensorService and Service and provides the method specifications to be used by both the client and the service implementation.

CsvDataServiceImpl

The CsvDataServiceImpl is a Java class which implements DataService and extends from AbstractDefaultService in the same way as the other service described. It provides just three methods for the client to use: initialise, log and close.

The initialise method takes as its parameters the directory in which to write the log file and a list of Channel objects. The Channel object provides a generic data structure for storing interpreted sensor data. The individual fields can be seen in figure 5.4. For trace data the object contains methods to extract the mean average and the peak value at runtime. The file name is taken to be the current date and time (in short format) and a FileWriter
object is opened. The writer then loops through the list of channels and writes the CSV header using the channel name and unit fields.

To create a line of CSV data the `log` method is called with a list of channels as the parameter. The list of channels must be in the same order and have the same number of elements as the list passed to the initialisation method. The time in milliseconds is written along with the trace average and peak values if the channel contains trace data, or just the channel value otherwise. The writer is then flushed ready to receive the next `log` invocation.

The `close` method simply flushes and closes the `FileWriter` object to ensure that the CSV is saved properly. The Data Service code can be seen in appendix ??.

### 5.1.4 Sensor Agents

![Sensor Agent UML class diagram](image)

*Figure 5.5: Sensor Agent UML class diagram.*
SensorNodeAgent

The SensorNodeAgent class is an abstract class that must be inherited by all sensor agents. It provides the methods required by all sensor agents to bind to and use a Data Service (in this case a CSV Data Service). It also registers the name of the agent using the AgentScape API. This reduces the amount of code that would otherwise have to be reproduced by sensor agents.

To use the Data Service (or any other user service), each agent has a service broker that allows it to connect to services. This broker finds a suitable service based on a query from the sensor agent, as shown in listing 5.1.

```java
DataService ds = getServiceBroker().bind(DataService.class);
boolean logged = ds.log(...);
getServiceBroker().release(ds);
```

Listing 5.1: Binding to a service.

The service broker is referenced by using getServiceBroker() from the Agent class. It is used to obtain a reference to a service of a given type, in this case DataService. The broker returns a client stub of the interface.

Once the service binding has been setup then the sensor agents can interact with the Data Service as if it were a local object. When the sensor agents are done with the service, it is manually released by SensorNodeAgent using the service broker release method.

WiVibSensorAgent

The WiVibSensorAgent extends the SensorNodeAgent and is responsible for gathering vibration data and passing it on to the Data Service. The service broker is used to connect to a WiVibSensorService which is initialised with a WiVib Server address and port number. A server configuration is passed to the WiVib Service that kicks off sampling. An internal thread is started which continuously polls the service for the list of WiVib objects being maintained by the service, along with a processing count. The processing count signals that a new sample has arrived. Therefore if the processor count has changed we log the data, otherwise poll again.

As each sample is picked up by the agent, it converts the sample into a Channel object which is then put into the channel list. Each sample channel has its own channel object. All data within the WiVib objects is of type String, so the time wave traces have to be converted to an array of floats in order to be useful.
Once data has been processed and the Channel list updated, the data is passed to the Data Service (after initialisation) which logs to CSV only if there has been a new measurement made.

**CanSensorAgent**

The *CanSensorAgent* extends from the *SensorNodeAgent*, using the service broker to connect to a *CanSensorService*. The service is initialised and an invocation is sent to start reading data. If this happens successfully then a thread is created which continuously polls the service for CAN data. If relevant data is found (the message ID 1536 in this case) then the RPM and and Ignition are converted from hexadecimal and copied into Channel objects. The channels are then sent to the data service for logging.

### 5.2 Build and Deployment

Maven is now being used as the primary build environment for AgentScape 2. It has a number of advantages and is a significant improvement on the cumbersome Ant build of the old AgentScape. An example *pom.xml* can be seen in appendix C.5.

#### 5.2.1 AgentScape repository

AgentScape 2 has its own repository which Maven can use to download the platform and any dependencies. The repository is located at: 

http://repo.agentscape.org/repo.

The most important dependency for developing with AgentScape is agentscape-core which provides the core AgentScape functionality.

#### 5.2.2 Maven Projects

Development in AgentScape now falls into three categories: agents, services and shared libraries. Each agent is essentially a JAR containing a main class which extends the *Agent* class. There must be a Main-Class (pointing to the main agent) entry in the manifest in order the the JAR to be injected into AgentScape. The customisation of the JAR manifest can be achieved
by using the \texttt{maven-jar-plugin}, which has to be specified as part of the \texttt{<build>} element of the project POM.

Services also need manifest customisations by using the \texttt{Service-Class} to specify which class is used to bootstrap the service. The interface that is used as the medium between service and client must be specified using the \texttt{Service-Interface} element.

To enable interaction between agents and services, the interface of the service has to be available as a dependency for both the agent and the service. Therefore the interface is deployed as a shared JAR, placed into the \texttt{lib/shared} directory.

\textit{agentscape-integrate-plugin}

The AgentScape Maven plugin automatically deploys projects JARs to the AgentScape installation folder. When inheriting from the AgentScape project, automatic deployment is achieved by setting a single property:

- \texttt{org.iids.aos.install.type}: type of project. Possible values are agent, service and shared
- \texttt{org.iids.aos.install.home}: AgentScape installation directory. Defaults to $\texttt{${user.home}}$/agentscape

The plugin automatically copies the JARs to the appropriate directories:

- Service JARs are copied to agentscape/lib/services
- Shared JARs are copied to agentscape/lib/shared
- Agent JARs are copied to agentscape/agents
Chapter 6

Testing

Throughout the development of the new AgentScape 2 sensor architecture and the individual components of the EHMS, throw away unit testing was used to ensure that each class/method worked as it was supposed to. The lack of a formal testing approach was due to lack of development time (see section 7.1.1).

6.1 Sample Output

The following output shows AgentScape starting with the CanSensorService,DataService and WiVibSensorService all starting as normal:

```plaintext
> Starting AgentScape
> Using lookupservice: http://lookup2.agentscape.org
> Starting Kernel
> [WARN] KernelStartup:295 - Did not recover kernel state:
5 No roles to recover!
> Starting system service
7 [org.iids.aos.systemservices.lookupservice.AsLookupServiceServer]
8 new LookupServiceInfo from LookupServiceServer, impl org.iids.aos.
  systemservices.lookupservice.AsLookupServiceServer@239525
9 > Starting LocationManager: MyLocation
10 > Starting system service [org.iids.aos.locationmanager.LocationManager]
11 > Starting system service [org.iids.aos.hostmanager.HostManager]
12 > Starting system service [org.iids.aos.service.server.ServiceServerImpl]
13 [WARN] RegistrationModule:98 - Could not connect to location manager
14 > Starting service [org.iids.aos.sensorsservice.can.CanSensorService]
15 > Starting service [org.iids.aos.sensorsservice.dataDataService]
16 > Starting service [org.iids.aos.sensorsservice.wivib.WiVibSensorService]
17 > Starting system service [org.iids.aos.agentserver.java.JavaAgentServer]
18 > Starting system service [org.iids.aos.agentserver.binary.BinaryAgentServer]
```
The following output shows a sample of a saved CSV log file for vibration data:

<table>
<thead>
<tr>
<th>DateTime</th>
<th>WiVib1:1 Average (g)</th>
<th>WiVib1:1 Peak (g)</th>
</tr>
</thead>
<tbody>
<tr>
<td>03:37:45:770</td>
<td>-0.017062157</td>
<td>8.735911</td>
</tr>
<tr>
<td>03:37:45:897</td>
<td>-0.017062157</td>
<td>8.735911</td>
</tr>
<tr>
<td>03:37:53:531</td>
<td>-8.0035534E-10</td>
<td>10.0.0251333E</td>
</tr>
<tr>
<td>03:37:54:304</td>
<td>-7.566996E-10</td>
<td>10.0.0380424</td>
</tr>
<tr>
<td>03:37:54:746</td>
<td>4.274625E-10</td>
<td>10.0.0321796</td>
</tr>
<tr>
<td>03:37:55:409</td>
<td>-1.294825E-9</td>
<td>9.0.03643861</td>
</tr>
<tr>
<td>03:37:56:427</td>
<td>-3.6561687E-10</td>
<td>10.0.03370636</td>
</tr>
<tr>
<td>03:37:57:218</td>
<td>-0.017450409</td>
<td>8.934698</td>
</tr>
<tr>
<td>03:37:58:002</td>
<td>-1.6643753E-9</td>
<td>9.0.02690824</td>
</tr>
<tr>
<td>03:37:58:685</td>
<td>1.1.823431E-10</td>
<td>10.0.0265096</td>
</tr>
<tr>
<td>03:38:06:942</td>
<td>-1.6516424E-9</td>
<td>9.0.03041841</td>
</tr>
<tr>
<td>03:38:07:506</td>
<td>1.3060344E-9</td>
<td>9.0.03495768</td>
</tr>
<tr>
<td>03:38:08:070</td>
<td>-2.7057467E-11</td>
<td>11.0.0297007</td>
</tr>
<tr>
<td>03:38:08:642</td>
<td>-2.8376235E-10</td>
<td>10.0.03800501</td>
</tr>
<tr>
<td>03:38:09:087</td>
<td>4.2984993E-10</td>
<td>10.0.03216241</td>
</tr>
<tr>
<td>03:38:09:658</td>
<td>4.947651E-10</td>
<td>10.0.02707132</td>
</tr>
<tr>
<td>03:38:10:222</td>
<td>-7.512426E-10</td>
<td>10.0.03662295</td>
</tr>
<tr>
<td>03:38:11:015</td>
<td>4.292475E-10</td>
<td>10.0.03011633</td>
</tr>
<tr>
<td>03:38:11:697</td>
<td>-6.0936145E-10</td>
<td>10.0.03033665</td>
</tr>
<tr>
<td>03:38:12:261</td>
<td>5.0931703E-10</td>
<td>10.0.04153914</td>
</tr>
<tr>
<td>03:38:12:819</td>
<td>3.0195224E-10</td>
<td>10.0.02679637</td>
</tr>
<tr>
<td>03:38:21:075</td>
<td>1.6825652E-10</td>
<td>10.0.03191932</td>
</tr>
<tr>
<td>03:38:21:632</td>
<td>-4.3655746E-11</td>
<td>11.0.02526772</td>
</tr>
</tbody>
</table>
Chapter 7

Evaluation and Conclusions

The prototype framework that has been created shows promise for becoming a fully fledged EHMS. The system fulfills many of the basic requirements and particularly pleasing is the frequency at which data is logged. Significant development time and resources will be needed for the framework to become a product used in an industrial setting. However, Multi-Agent Systems and in particular AgentScape have shown themselves to be a good basis for developing multi-sensor data acquisition applications.

The architecture allows sensors of any type to be added to the sensor network in a uniform way. All engine health and condition monitoring applications need to acquire and store data and the implemented framework delivers a universal way doing this. Therefore, with continuing support and a maturing AgentScape 2 it is not unreasonable to suggest that the framework could offer an open source solution to the problem of condition monitoring. Agents would become plug-ins that cater for different sensor and machine types.

7.1 Development Issues

7.1.1 Hardware Delays

The search for appropriate hardware to fulfill project requirements began in January 2010 after completion of the Literature Review and the finalising of requirements. The search took a quite a substantial amount of time from beginning to placing purchase orders; around four weeks. The main reason for this was due to the specialist nature of the equipment being sought.
Wireless vibration data acquisition units are not readily available for bespoke condition monitoring on small combustion engines. Often vibration sensing systems are sold as part of an overall machinery health solution. A good example of this is the Honeywell OneWireless Equipment Health Monitoring system\(^1\). Once the WiVib had been singled out, a great deal of correspondence with the supplier to ensure the device was appropriate before an order was placed.

Due to the small form factor of Cubewano’s range of engines finding suppliers of accelerometers that are small enough and could withstand high temperatures also proved difficult. Once possible options had been narrowed down they were put to Cubewano for feedback and approval. The accelerometer settled upon had a long lead time of five weeks which was compounded by further delays in purchasing and delivery.

The impact of the hardware search taking longer than expected and then facing long lead times and delays in delivery was that time to develop using the hardware was significantly cut compared to what was originally planned. This is the primary reason why aspects of the system such as the user interface and the virtual/analysis agents have not been implemented. The WiVib and CAN hardware interfaces are challenging to get to grips with and then develop on, and more access to them would have enabled implementation to go a lot further.

### 7.1.2 WiVib Development

On the advice of Icon Research and given the time available the WiVib Sensor Service communicates with the WiVib device via the WiVib Server. This presented two problems:

1. Representing the WiVib Server XML Schema in Java.
2. Communicating with a closed, black-box server written in .NET over TCP.

#### XML Schema

The problem of representing the XML schema was discussed in the design and implementation sections. Initially JDOM / SAX were used in the following

\(^1\)http://hpsweb.honeywell.com/Cultures/en-US/Products/Wireless/default.htm
object model:

- **ServerLink**: Interface from the application to the WiVibServer. Handles the TCP link and distributes XML document sections to the appropriate class objects for processing.

- **WiVib**: Container class representing a single WiVib device.

- **Channel**: Container class to manage a single Channel of a WiVib.

- **Measurement**: Container class for a single measurement configuration.

- **Sample**: Container class for a sample produced by the server.

Organised in a hierarchy of:

```
ServerLink
  WiVib
    Channel
    Measurement
    Sample
```

Each object would generate and parse its own XML, with the server link concatenating the various XML sections together. This turned out to be a cumbersome and error prone method. Fortunately support at Icon Research pointed to a XML Schema that could be used to generate an object model using JAXB.

One of the problems with Java is that it provides a lot of bad solutions for a given problem. Using JAXB significantly reduced the complexity of the WiVib Sensor Service.

**Communicating**

Whilst receiving samples from the WiVib Server was not a problem, sending back configurations proved to be very difficult. Many different types of output streams were tried to no avail. Eventually it was found that all outgoing streams should be encoded in ASCII. Not being aware of this is indicative of the lack of detailed documentation that the WiVib and the WiVib Service possess.
CHAPTER 7. EVALUATION AND CONCLUSIONS

7.1.3 Developing on AgentScape 2

Due to an EHMS being a long term project for Cubewano the decision was taken to convert to AgentScape 2 to a painful transition later on in the system development, the idea being that the EHMS can mature along with AgentScape 2. The old AgentScape model is redundant and it was viewed that there was little point continuing to develop a prototype proof-of-concept system on it.

Limitations of Milestone 1

One of the major limitations that can create architectural nuances is only allowing one service implementation supported for each shared service interface. There are many instances where the ability to have multiple implementations for a particular service interface would be desirable. Data Access Objects provide a good example. Currently a new database would have to be implemented as an entirely new service.

Having approached the AgentScape development team with this limitation they reported that service configuration and management (like offering multiple instances of a service interface) are on the short-term planning road map.

Another limitation in the context of an early release is the lack of a facility to report bugs and then for the AgentScape development community to release fixes quickly using Maven.

Bugs

A major bug was discovered in callback handling for service invocations. A null pointer exception occurs when a service is bound to an agent, and the agent calls a method on the service that takes no parameters. To fix this bug the sources for the agentscape-core JAR were downloaded and the appropriate null checks put in place. This then added complexity of ensuring the new agentscape-core was used instead of the default repository binary.

The AgentScape development team have been made aware of the bug. Apparently it was assumed that an invocation with no arguments reports Object[0] instead of null.
7.2 Future Work

Harman (2009) lists many potential future developments for the Sensor Network Platform, and they still apply on the new AgentScape 2 architecture. In fact, this project has considered the design and implementation of two of them, namely extending communication interfaces, sensor compatibility and alternative data services.

7.2.1 EHMS Gui

The logical next stage in the implementation the EHMS is a rich client to display sensor data and allow for configuration changes. A design and Java mock-up were considered in section [4.8.5] and it now remains for the mock-up to be hooked into the sensor network architecture.

The GUI should be implemented as an Agent, and it could then create the appropriate sensor node agents at runtime. The following code is an example of how this could be done:

```java
private byte[] createAgent() throws AgentCreationException {
    Class2Jar cj = new Class2Jar();
    Thread.currentThread().setContextClassLoader(getClass().getClassLoader());
    cj.addPackage(getClass().getPackage(), true);
    cj.setMainClass(CanSensorAgent.class);
    try {
        return cj.getJar();
    } catch (Exception e) {
        throw new AgentCreationException("Problem creating raw agent", e);
    }
}

private void injectAgent(byte[] sensorAgent) throws AgentStartupException {
    System.err.println("Injecting sensor agent of " + sensorAgent.length + " bytes");
    try {
        AgentArchive arch = AgentArchive.fromJar(sensorAgent);
        AgentScapeID loc = getCurrentLocation();
    }
```
Listing 7.1: Creating a CAN Sensor Agent at runtime.

Sensor data could be propagated to the GUI using standard AgentScape messaging. The GUI would then use a data service to save both CAN and WiVib data in one CSV.

### 7.2.2 Virtual and Analysis Agents

Currently the system can only log live data. Eventually the EHMS should have agents capable of analysing and trending using both live AND historical data from the data service. This will enable visibility of gradual degradation of the rotor bearings and provide an early warning system for when the engine exhibits vibration characteristics which are out of character. An initial implementation could show trending historical data on the EHMS Gui (perhaps on a scrolling chart) which is continuously supplemented by new live data.

### 7.2.3 Extending CanSensorAgent to act as an Actuator

The CanSensorAgent currently polls for data, interprets data and sends the data to be logged. To conform to the design and requirements the agent should, when required, invoke commands from the CAN Service to send CAN messages and exhibit control over the engine. Once the CanSensorAgent is hooked up to a user interface where it can receive commands, sending CAN data would be relatively straightforward to implement.
Bibliography


Avtec (2009), ‘Engine health management systems’, *Concept Demonstration*.


Mobach, D. (2009), Enforcing Integrity of Agent Migration Paths by Distribution of Trust, PhD thesis, Computer Science Department, Vrije Universiteit Amsterdam.


Pazul, K. (1999), Controller area network (can) basics, Technical report, Microchip Technology Inc.


Appendix A

Diagrams

Figure A.1: Server Interface Diagram
Figure A.2: Architecture of Peak CAN USB System.
Figure A.3: WiVib UML Object Model.
Appendix B

Accelerometer Documents
3. CASE AND CONNECTOR MATERIAL: TITANIUM.
2. WEIGHT LESS CABLE: 0.6 GRAMS.

ARROW INDICATES SENSE AND DIRECTION OF
INPUT ACCELERATION FOR NEGATIVE GOING
OUTPUT SIGNAL.

MOUNTING SURFACE
ADHESIVE MOUNT WITH CYANOACRYLATE
ADHESIVE OR DYTRAN MOUNTING WAX -
MODEL 6273

KNURLED CABLE NUT. TIGHTEN ONLY
BY HAND. DO NOT USE PLIERS TO
TIGHTEN. TO INSTALL, DO NOT ROTATE
ACCELEROMETER INTO CABLE END
BUT RATHER ROTATE CABLE NUT.

COAXIAL CONNECTOR, DYTRAN SPECIAL
3-56 THREAD. MATES ONLY WITH MODEL 60034XX CABLE

MODEL 6003A03 CABLE.
LENGTH, 3 FT. (INCLUDED)

10-32 COAXIAL CONNECTOR, JACK
(PART OF 6003A03 CABLE, INCLUDED)
## SPECIFICATIONS

### MODEL 3225E1 MINIATURE CHARGE MODE ACCELEROMETER

<table>
<thead>
<tr>
<th>SPECIFICATION</th>
<th>VALUE</th>
<th>UNITS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PHYSICAL</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Weight</td>
<td>0.6</td>
<td>grams</td>
</tr>
<tr>
<td>Size (OA x Length HEX x Height)</td>
<td>0.25 x 0.43 x 0.150</td>
<td>inches</td>
</tr>
<tr>
<td>Mounting Provision</td>
<td>Flat Mounting Surface for Adhesive Mount</td>
<td></td>
</tr>
<tr>
<td>Connector, Radially Mounted [1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Case Material</td>
<td>Titanium</td>
<td></td>
</tr>
<tr>
<td>Sensing Element Type</td>
<td>Piezoceramic Planar Shear</td>
<td></td>
</tr>
<tr>
<td><strong>PERFORMANCE</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sensitivity [2] [3]</td>
<td>&gt;5.5</td>
<td>g/Cg</td>
</tr>
<tr>
<td>Rms F.S.</td>
<td>±250</td>
<td>%</td>
</tr>
<tr>
<td>Frequency Response, ±10% [3]</td>
<td>10 to 150,000</td>
<td>Hz</td>
</tr>
<tr>
<td>Mounded Resonant Frequency, Nom.</td>
<td>40</td>
<td>kHz</td>
</tr>
<tr>
<td>Amplitude Nonlinearity (Zero Based Best Fit Line Method)</td>
<td>±2%</td>
<td>% FS Max.</td>
</tr>
<tr>
<td>Strain Sensitivity, Max.</td>
<td>.0005</td>
<td>g's per microstrain @ 255µm</td>
</tr>
<tr>
<td><strong>ENVIRONMENTAL</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum Vibration</td>
<td>40</td>
<td>g's RMS</td>
</tr>
<tr>
<td>Maximum Shock</td>
<td>500</td>
<td>g's Peak</td>
</tr>
<tr>
<td>Temperature Range</td>
<td>-55 to 300</td>
<td>°C</td>
</tr>
<tr>
<td>Thermal Coefficient of Sensitivity</td>
<td>See Graph on This Page</td>
<td></td>
</tr>
<tr>
<td>Environmental Seal</td>
<td>Hermetic</td>
<td></td>
</tr>
<tr>
<td><strong>ELECTRICAL</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output Signal Polarity for Acceleration Toward Top Case Grounding</td>
<td>Negative</td>
<td></td>
</tr>
<tr>
<td><strong>SUPPLIED ACCESSORIES:</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTES:**
1. Connector mates only with Dytran cable model 9003AXXX (XXX is length in feet).
2. Measured at 50 Hz, 1.0 RMS per ISA RP 3.2.
3. Actual sensitivity is given on a calibration certificate traceable to NIST, supplied with each instrument. Low frequency response is dependent upon the discharge time constant of the charge amplifier.

## TYPICAL THERMAL SENSITIVITY GRAPH

![Graph showing thermal sensitivity as a function of temperature in degrees Fahrenheit.](image-url)
Appendix C

Code Samples

Due to the large size, it was not suitable to include all source code from the EHMS Sensor Framework in this chapter. Instead key aspects are included that highlight the points made in the dissertation. The full source code is available on the supplied CD in the folder src. Alternatively the code can made available on request.
C.1 WiVib Service

C.1.1 SensorService.java

```java
package org.iids.aos.sensorservice;

/**
 * @author jack_higgs
 */
public interface SensorService {

  public static final String WII_REMOTE_LABEL = "wiiMote";
  public static final String GPS_SENSOR_LABEL = "gpsSensor";
  public static final String CAN_SENSOR_LABEL = "canSensor";
  public static final String WIVIB_SENSOR_LABEL = "wivibSensor";

  public boolean connect();
  public boolean close();
  public boolean isConnected();
}
```

C.1.2 WiVibSensorService.java

```java
package org.iids.aos.sensorservice.wivib;

import java.util.List;
import org.iids.aos.sensorservice.SensorService;
import org.iids.aos.service.Service;
import wivib.WiVibNetwork;

public interface WiVibSensorService extends SensorService, Service {

  public void initialise(String ServerName, int ServerPort);

  public void setServerLinkStatus(boolean value);

  public WiVibNetwork.WiVib getSelectedWiVib();

  public void setSelectedWiVib(WiVibNetwork.WiVib value);

  public boolean SendServerConfiguration(String filePath);

  public boolean Remove(WiVibNetwork.WiVib OldItem);

  public List<WiVibNetwork.WiVib> getListOfWiVibs();
}
### C.1.3 WiVibSensorServiceImpl.java

```java
package org.iids.aos.sensorservice.wivib.impl;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamTransformer;
import org.iids.aos.sensorservice.wivib.WiVibSensorService;
import org.iids.aos.service.AbstractDefaultService;

import wivib.WiVibNetwork;

/** @author jack_higgs *
 */

public class WiVibSensorServiceImpl extends AbstractDefaultService implements WiVibSensorService {

    private Socket m_StreamingConnection = null;
    private String m_ServerName = "";
    private int m_ServerPort = 0;
    private boolean ServerLinkStatus = false;
    private boolean m_CloseNetwork = false;
    private boolean m_DropLink = false;

    private WiVibNetwork.WiVib theSelectedWiVib = null;
    public List<WiVibNetwork.WiVib> WiVibList;
    private JAXBContext jaxContext;
    private Unmarshaller unmarshal;

    public void initialise(String ServerName, int ServerPort) {
        try {
            WiVibList = new ArrayList<
                WiVibNetwork.WiVib>();
            m_ServerName = ServerName;
            m_ServerPort = ServerPort;
        }
    }
```

jaxContext = JAXBContext.newInstance("wivib");
unmarshal = jaxContext.createUnmarshaller();
}
catch (Exception ex) {
    ex.printStackTrace();
}
}

public boolean connect() {
    new Thread(new Thread() {
        @Override
        public void run() {
            ServerLinkWatchdog();
        }
    }).start();
    return true;
}

public boolean isConnected() {
    return ServerLinkStatus;
}

public void setServerLinkStatus(boolean value) {
    if (ServerLinkStatus != value) {
        ServerLinkStatus = value;
    }
}

public WiVibNetwork.WiVib getSelectedWiVib() {
    return theSelectedWiVib;
}

public void setSelectedWiVib(WiVibNetwork.WiVib value) {
    if (theSelectedWiVib != value) {
        theSelectedWiVib = value;
    }
}

public boolean close() {
    try {
        m_DropLink = true;
        m_CloseNetwork = true;
        return true;
    } catch (Exception ex) {
        ex.printStackTrace();
        return false;
    }
}

private void ServerLinkWatchdog() {
    int RetryCounter = 0;
    m_CloseNetwork = false;
    ServerLinkStatus = false;
    while (!m_CloseNetwork) {
        try {
            if (!ServerLinkStatus) {
                if (RetryCounter == 0) {
                    try {
InetSocketAddress hostEndPoint = null;
for (InetAddress IPA : InetAddress.getAllByName(m_ServerName)) {
  hostEndPoint = new InetSocketAddress(IPA, m_ServerPort);
  m_StreamingConnection = new Socket();
  m_StreamingConnection.connect(hostEndPoint);
}
if (hostEndPoint != null) {
  m_DropLink = false;
  setServerLinkStatus(true);
  new Thread(new Thread() {
    @Override
    public void run() {
      ServerLinkHandler();
    }
  }).start();
} else {
  RetryCounter = 60;
  setServerLinkStatus(false);
  throw new Exception("Unable to find local IP4 socket");
}
} catch (Exception ex) {
  ex.printStackTrace();
  RetryCounter = 60;
  setServerLinkStatus(false);
}
```java
else
    RetryCounter--; 
}
Thread.sleep(1000);
} catch (Exception err) {
    err.printStackTrace();
}

private void ServerLinkHandler() {
    BufferedReader Stream;
    try {
        Stream = new BufferedReader(new InputStreamReader(
            m_StreamingConnection.getInputStream()));
        int CheckAliveCounter = 0;
        String Message = "";
        while (!m_DropLink) {
            if (Stream.ready()) {
                char[] ReadBuffer = new char [2048];
                int BytesRead = 0;
                do {
                    BytesRead = Stream.read(
                        ReadBuffer, 0,
                        ReadBuffer.length);
                    Message = Message.concat(
                        new String(
                            ReadBuffer, 0,
                            BytesRead));
                } while (Stream.ready());
                int StartPos = -1;
                int EndPos = -1;
                do {
                    StartPos = -1;
                    EndPos = -1;
                    if (Message.length() >=
                        29) {
                        StartPos = Message.indexOf("<\nWiVibNetwork>");
                        EndPos = Message.indexOf("</\nWiVibNetwork>",
                        StartPos);
                        if ((EndPos != -1)
                            && (StartPos !=
                            -1)) {
                            ProcessMessage(
                                Message,
                                substring(
                                    StartPos,
                                    EndPos -
                                    StartPos)
                            ,
```
Message = Message.substring(EndPos + 15);
CheckAliveCounter = 0;
while ((EndPos != -1) && (StartPos != -1));
else if (CheckAliveCounter++ == 100) {
  SendOverLink(" ");
  CheckAliveCounter = 0;
}
try {
  Thread.sleep(50);
} catch (InterruptedException e) {
  e.printStackTrace();
}

m_StreamingConnection.shutdownInput();
m_StreamingConnection.shutdownOutput();
m_StreamingConnection.close();
}

private void ProcessMessage(String Message) {
  try {
    WiVibNetwork foo = (WiVibNetwork)
    .unmarshal(new StreamSource(new StringReader(Message)));
    WiVibList = foo.getWiVib();
    System.out.println(WiVibList.get(0).getSN());
  } catch (JAXBException e) {
    System.err.println("Could not unmarshal Message");
  }
}

public boolean SendServerConfiguration(String filePath) {
  try {
    WiVibNetwork foo = (WiVibNetwork)
    .unmarshal
unmarshal(new FileInputStream(filePath));
Marshaller m = jaxContext;
createMarshaller();
StringWriter sw = new StringWriter();
m.marshal(foo, sw);
return SendOverLink(sw.toString());
}
}
public boolean SendOverLink(String Message) {
byte[] WriteBuffer;
PrintStream output;
try {
WriteBuffer = Message.getBytes("ASCII");
OutputStream os =
mStreamingConnection.getOutputStream();
output = new PrintStream(os, true, ", "ASCII");
output.write(WriteBuffer, 0, WriteBuffer.length);
return true;
} catch (IOException e) {
if (!mStreamingConnection.
isConnected())
m_DropLink = true;
System.out.println(e);
return false;
}
public void Remove(WiVibNetwork.WiVib
OldItem) {
synchronized (WiVibList) {
WiVibList.remove(OldItem);
}
public List<WiVibNetwork.WiVib> getListOfWiVibs() {
    return WiVibList;
}

C.2 CAN Service

C.2.1 CANLight.java

/**
 * Defines a CAN Message
 * @author jack_higgs
 */
public class CANLight {

    /**
     * Standard Constructor
     * If an object is created with this constructor use init() to initialize the object
     */

    public CANLight() {
    }

    /**
     * Initializes the PCANLight for a specific hardware
     * @param hwType the hardware to be used
     */
    public native boolean initializeAPI(int hwType);

    /**
     * Activates the PNP hardware, initializes it and makes some tests
     * Use this method only with PNP hardware!
     * @param baudRate the baudrate to be used
     * @param msgType the type of the message frame (standard or extended)
     * @return 0, if successful, otherwise the error code
     */
    public native int init(int baudRate, int msgType);

    /**
     * Deactivates the hardware and frees all used resources
     * @return 0, if successful, otherwise the error code
     */
APPENDIX C. CODE SAMPLES

```java
public native int close();

/**
 * Reads the next message.
 * With this method it is possible to reuse a previously created message object.
 * @param message An existing message object that should be "filled" with data
 * should not be null
 * @return 0, if successful, otherwise the error code
 * @see peak.canlight.CANMessage
 */
public native int read(CANMessage message);

/**
 * Reads the next message.
 * With this method it is possible to reuse a previously created message object.
 * @param message An existing message object that should be "filled" with data.
 * Should not be null
 * @param rcvTime The TPCANTimestamp structure for the message’s timestamp
 * @return 0, if successful, otherwise the error code.
 * @see peak.canlight.CANMessage
 * @see peak.canlight.CANTimestamp
 */
public native int readEx(CANMessage message, CANTimestamp rcvTime);

/**
 * Writes a message.
 * @param message The message that should be written to the can bus
 * @return 0, if successful, otherwise the error code
 * @see peak.canlight.CANMessage
 */
public native int write(CANMessage message);

static {
    System.loadLibrary("pcanlight.jni");
}
```

C.2.2 CanSensorServiceImpl.java

```java
package org.iids.aos.sensorservice.can.impl;

import java.util.HashMap;
import java.util.List;

import org.iids.aos.sensorservice.can.CanSensorService;
```

import org.iids.aos.service.AbstractDefaultService;
import peak.canlight.CANLight;
import peak.canlight.CANMessage;
import peak.canlight.RcvEventDispatcher;
import static peak.canlight.CANConstants.*;

/**
 * @author jack_higgs
 */
public class CanSensorServiceImpl extends AbstractDefaultService implements CanSensorService {
    // Static Token For Synchronisation
    static public final Object token = new Object();
    // CANLight object
    static private CANLight canLight = new CANLight();;
    // Store the state of Peak CAN connection
    private boolean connected;
    // Map to store received messages
    private HashMap<Integer, CanDataRow> receivedData = new HashMap<Integer, CanDataRow>();
    // Map to store processed messages
    private HashMap<String, List<String>> processedData = new HashMap<String, List<String>>();
    // Thread to read CAN messages
    CanReadThread canReadThread;
    // Thread to process CAN messages
    CanProcessThread canProcessThread;

    /** (non-Javadoc)
     * @see org.iids.aos.sensorservice.SensorService#connect()
     */
    public boolean connect() {
        int err;
        try {
            if (canLight.initializeAPI(PCAN_USB_1CH)) {
                err = canLight.init(BAUD_1M, MSGTYPE_STANDARD);
                if (err == ERR_OK) {
                    isConnected = true;
                    return true;
                }
            }
            catch (Exception ex) {
                System.err.println(ex.getMessage());
                isConnected = false;
                return false;
            }
        }
        catch (Exception ex) {
            System.err.println(ex.getMessage());
            isConnected = false;
            return false;
        }
        // Process Last Error code
        long lastErrorCode = canLight.getLastErrorCode();
        System.err.println(lastErrorCode);
        isConnected = false;
        return false;
    }
}
public void startCanRead(boolean useTimerMode) {
    if (isConnected()) {
        // Begin reading from CAN
        changeReadMode(useTimerMode);
        CanProcessThread = new CanProcessThread(receivedData, processedData);
        canProcessThread.start();
    }
}

public HashMap<String, List<String>> getData() {
    return processedData;
}

private void changeReadMode(boolean useTimerMode) {
    int res;
    // Stop timer thread
    if ((canReadThread != null) && (!canReadThread.isInterrupted())) {
        canReadThread.interrupt();
        try {
            canReadThread.join();
        } catch (InterruptedException ex) {
        }
    }
    // Create new thread
    canReadThread = new CanReadThread(canLight, receivedData);
    if (useTimerMode) {
        res = canLight.resetRcvEvent();
        // An error occurred. Inform user and exit.
        if (res != ERR_OK) {
            System.err.println("Could not reset CAN receive event");
            return;
        }
        // Start Timer Thread to read CAN Messages
        canReadThread.start();
    } else {
        // Set canReadThread as RcvEvent Listener
        RcvEventDispatcher.setListener(canReadThread);
        // Set Rcv Event
        canLight.setRcvEvent();
    }

    /* (non-Javadoc)
     * @see org.iids.aos.sensorservice.can.CanSensorService#sendCanMessage
     * (org.iids.aos.sensorservice.can.CanTxData)
     */
    public boolean sendCanMessage(String msgId, int length, byte[] msgData) {
        canReadThread = new CanReadThread(canLight, receivedData);
        if (useTimerMode) {
            res = canLight.resetRcvEvent();
            // An error occurred. Inform user and exit.
            if (res != ERR_OK) {
                System.err.println("Could not reset CAN receive event");
                return;
            }
            // Start Timer Thread to read CAN Messages
            canReadThread.start();
        } else {
            // Set canReadThread as RcvEvent Listener
            RcvEventDispatcher.setListener(canReadThread);
            // Set Rcv Event
            canLight.setRcvEvent();
        }
    }
CANMessage msgToSend;
int res;

// We create a TCLightMsg message
struct
msgToSend = new CANMessage();

// We configure the Message. The ID (max 0x1FF),
// Length of the Data, Message Type (Standard in
// this example) and the data
msgToSend.setId((int) hexTextToInt(msgId));
msgToSend.setLen(Byte.parseByte(Integer.valueOf(length).toString()));
msgToSend.setMsgType(MSGTYPE_STANDARD);
msgToSend.setData(msgData);

if (isConnected()) {
    // The message is sent to the configured hardware
    res = canLight.write(msgToSend);
    // The Hardware was successfully sent
    if (res == ERR_OK) {
        return true;
    }
}

return false;

/* (non-Javadoc)
 * @see org.iids.aos.sensorservice.SensorService#close()
 */

public boolean close() {
    int ret;
    ret = canLight.close();
    try {
        // We stop the CAN thread
        if ((canReadThread != null) && (!canReadThread.isInterrupted())) {
            canReadThread.interrupt();
            canReadThread.join();
        }
        // We stop the Process thread
        if ((canProcessThread != null) && (!canProcessThread.isInterrupted())) {
            canProcessThread.interrupt();
            canProcessThread.join();
        }
    } catch (InterruptedException ex) {
        return false;
    }
    // An error occurred. We show the error
    if (ret != ERR_OK) {
        System.err.println("Could not close/release CAN");
        return false;
    }
} else {
```java
setConnected(false);
return true;
}
/**
 * @return the connected
 */
public boolean isConnected()
{
    return connected;
}
/**
 * @param connected the connected to set
 */
private void setConnected(boolean connected)
{
    this.connected = connected;
}

@SuppressWarnings("null")
public String getDeviceID()
{
    int[] dvcNum = null;
    if(canLight.getUSBDeviceNr(dvcNum) == 0)
    {
        return dvcNum.toString();
    }
    else
    {
        return "";
    }
}

private static long hexTextToInt(String toConvert)
{
    long iToReturn = 0;
    int iExp = 0;
    char chByte;
    // The string to convert is empty
    if (toConvert.equals(""))
    {
        return 0;
    }
    // The string have more than 8 character (the equivalent value
    // exceeds the DWORD capacity
    if (toConvert.length() > 8)
    {
        return 0;
    }
    // We convert any character to its Upper case
    toConvert = toConvert.toUpperCase();
    try
    {
        // We calculate the number using the
        // Hex To Decimal formula
        for (int i = toConvert.length() - 1; i >= 0; i--)
        {
            chByte = (char) toConvert.getBytes()[i];
            switch ((int) chByte)
            {
                case 65:
                    iToReturn += (long) (10 * Math.pow(16.0f, iExp));
                    break;
                case 66:
                    iToReturn += (long) (16 * Math.pow(16.0f, iExp));
                    break;
                case 67:
```
C.2.3 CanReadThread.java
private CANLight canLight;
private HashMap<Integer, CanDataRow> dataCollection;

// Constructor
public CanReadThread(CANLight can, HashMap<Integer, CanDataRow> data) {
    this.canLight = can;
    this.dataCollection = data;
}

public void run() {
    while (true) {
        // Call ReadEx
        readData();
        // Sleep
        try {
            Thread.sleep(10);
        } catch (InterruptedException ex) {
            return;
        }
    }
}

private void readData() {
    // Variables
    int ret;
    int msgId = 0;
    CANMessage canMessage;
    CANTimestamp rcvTime;
    CanDataRow dataRow;

    try {
        // Create A New Message
        canMessage = new CANMessage();
        // Create A New Time Stamp
        rcvTime = new CANTimestamp();
        // Read message
        ret = canLight.readEx(canMessage, rcvTime);

        if (ret == ERR_OK) {
            // Get Message ID
            msgId = canMessage.getId();
            // Check If Message Was Already Processed
            if (dataCollection.containsKey(msgId)) {
                dataRow = (CanDataRow) dataCollection.get(msgId);
            } else {
                dataRow = new CanDataRow();
            }
            // Set Read Message
            dataRow.setMessage(canMessage);
            // Set Read Timestamp
            dataRow.setRcvTime(rcvTime);
            //Update Message COunter
```java
dataRow.setCounter(dataRow.getCounter() + 1);
}

// Critical Area
synchronized (CanSensorServiceImpl.token) {
    // Put Message In the dataRowCollection
    dataCollection.put(msgId, dataRow);
}

canMessage = null;
recvTime = null;
while ((ret & ERR_QRCVEMPTY) == 0);
}
}

/**
 * @param recData
 * @param proData
 */
public void processRcvEvent() {
    readData();
}
```
public CanProcessThread(HashMap<Integer, CanDataRow> recData, 
        HashMap<String, List<String>> proData) {
this.recData = recData;
this.proData = proData;
}

public void run() {
    // Variables
    CANMessage msg = null;
    String msgIDStr = "";
    String msgLength = "";
    String msgType = "";
    String msgData = "";
    String msgCount = "";
    String msgRcvTime = "";
    List<String> msgDataString = null;
    while (true) {
        for (CanDataRow dataRow : recData.values()) {
            // Reset Variables
            msg = null;
            msgIDStr = "";
            msgLength = "";
            msgType = "";
            msgData = "";
            msgCount = "";
            msgRcvTime = "";
            msgDataString = null;
            // Get CanMessage
            msg = dataRow.getMessage();
            // Get Type
            if (msg.getMsgType() == MSGTYPE_STANDARD) {
                msgType = "Standard";
            } else {
                msgType = "Extended";
            }
            // Message Length
            msgLength = String.valueOf(msg.getLen());
            // Message ID
            msgIDStr = Integer.toHexString(msg.getId()) + "h";
            // Message Data (convert to Hex to be used by client)
            byte[] d = msg.getData();
            for (int dataIndex = 0; dataIndex < msg.getLen(); 
                    dataIndex++) {
                blockData = Integer.toHexShort(d[dataIndex] & 0xff);
                if (blockData.length() == 1) {
                    blockData = "0" + blockData;
                }
                msgData = msgData + blockData + " ";
            }
APPENDIX C. CODE SAMPLES

```java
// Message Count
msgCount = String.valueOf(dataRow.getCounter());

// Add Rcv Time
msgRcvTime = String.valueOf(dataRow.getRcvTime().getMillis()) + "." + String.valueOf(dataRow.getRcvTime().getMicros());

// Check If Message Was Already Processed
if (proData.containsKey(msgIDStr)) {
    msgDataString = proData.get(msgIDStr);
} else {
    msgDataString = new ArrayList<String>();
}
msgDataString.add(msgType);
msgDataString.add(msgLength);
msgDataString.add(msgData);
msgDataString.add(msgCount);
msgDataString.add(msgRcvTime);
synchronized (CanSensorServiceImpl.token) {
    // Put Processed Message In the proData Map
    proData.put(msgIDStr, msgDataString);
}
```

```java
try {
    Thread.sleep(100);
} catch (InterruptedException ex) {
    System.err.println(ex.getMessage());
    return;
}
```

C.3 Data Service

C.3.1 CsvDataServiceImpl.java

```java
package org.iids.sensorservice.data.impl;

import java.io.FileWriter;
import java.io.IOException;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.List;
```

```java
    /∗∗
    *
    ∗/
    package org.iids.sensorservice.data.impl;
```
import org.iids.aos.sensoragents.channel.Channel;
import org.iids.sensorbservice.data.DataService;

/**
 * @author jack_higgs
 */
public class CsvDataServiceImpl implements DataService {
    private FileWriter output;
    SimpleDateFormat dateFormat;
    boolean initialised;

    /**
     * Create CSV header information
     * @param logDirectory
     * @param channels
     * @param return
     */
    public boolean initialise(String logDirectory, List<Channel> channels) {
        if (initialised) {
            return true;
        }
        Date now = new Date();
        String fileName = new SimpleDateFormat("HH:mm:ss:SSS").format(now) + ".csv";
        dateFormat = new SimpleDateFormat("HH:mm:ss:SSS");
        try {
            output = new FileWriter(logDirectory + "\" + fileName);
            output.append("DateTime , ");
            for (Channel channel : channels) {
                // If channel contains a trace then we save the average and peak values
                if (channel.isTrace()) {
                    output.append(channel.name + " Average (" + channel.units + "), ");
                    output.append(channel.name + " Peak (" + channel.units + "), ");
                } else {
                    output.append(channel.name + " (" + channel.units + ", ");
                }
            }
            output.append("\n")
            output.flush();
            initialised = true;
            return true;
        } catch (IOException e) {
        }
    }
}
C.3.2 Channel.java

```java
public boolean log(List<Channel> channels) {
    if (initialised) {
        Date now = new Date();
        try {
            output.append(dateFormat.format(now) + " , ");
            for (Channel channel : channels) {
                if (channel.isTrace()) {
                    output.append(Float.toString(channel.getTraceAverageValue()) + " , ");
                    output.append(Float.toString(channel.getTracePeakValue()) + " , ");
                } else {
                    output.append(Float.toString(channel.getValue()) + " , ");
                }
            }
            output.append("\n");
            output.flush();
            return true;
        } catch (IOException e1) {
            e1.printStackTrace();
            return false;
        }
    }
    return false;
}

public boolean close() {
    try {
        output.flush();
        output.close();
        initialised = false;
        return true;
    } catch (IOException e) {
        e.printStackTrace();
        return false;
    }
}
```

package org.iids.aos.sensoragents.channel;
```
```java
public class Channel {
  int id;
  public String name;
  public String units;
  boolean isTrace;
  float[] trace;

  public Channel(int id, String name, String units, float[] trace) {
    this.id = id;
    this.name = name;
    this.units = units;
    this.trace = trace;
  }

  public float getTraceAverageValue() {
    float Sum = 0;
    for (int Index = 0; Index < trace.length; Index++)
      Sum += trace[Index];
    return Sum / trace.length;
  }

  public float getTracePeakValue() {
    float Max = 0;
    for (int Index = 0; Index < trace.length; Index++)
      if (Math.abs(trace[Index]) > Max)
        Max = Math.abs(trace[Index]);
    return Max;
  }

  public boolean isTrace() {
    return isTrace;
  }
}
```
C.4 Agents

C.4.1 SensorNodeAgent.java

```java
package org.iids.aos.sensoragents;

import java.util.ArrayList;
import java.util.List;
import java.util.Timer;
import org.iids.aos.agent.Agent;
import org.iids.aos.exception.AgentEscapeException;
import org.iids.aos.sensoragents.channel.Channel;
import org.iids.sensorservice.data.DataService;

public abstract class SensorNodeAgent extends Agent {

    private static final long serialVersionUID = 1L;
    protected static String NAME_KEY = null;

    protected String myName = null;
    protected Timer delaySendTimer = null;
    protected int delaySendTimeSecs = 0;
    public int dataUploadCount = 0;
    public List<Channel> channels;

    protected DataService dataService;

    protected void setup(String nameKey, String myName, int sendDelaySecs) {
        NAME_KEY = nameKey;
        this.myName = myName;
        delaySendTimeSecs = sendDelaySecs;
        // initialise array of channels
        channels = new ArrayList<Channel>();
    }

    /*
     * (non-Javadoc)
     *
     * @see org.iids.aos.agent.Agent#run()
     */
```
C.4.2 WiVibSensorNodeAgent.java

```java
/*
 * @Override
 * public void run() {
 * // register agent service and name
 * startWithName(myName);
 * // create data service binding
 * try {
 * dataService = getServiceBroker().bind(DataService.class);
 * getLog().info("DataService is bound to sensor agent " + NAME_KEY);
 * } catch (Exception e) {
 * getLog().error("Could not bind Data Service to sensor agent " +
 * NAME_KEY);
 * dataService = null;
 * } try {
 * register(getPrimaryHandle(), NAME_KEY);
 * } catch (AgentScapeException e) {
 * getLog().info("Problem registering with AS" + ", Error is:" + e);
 * return;
 * } } */

/**
 * Registers the agent
 * @param myName
 */
protected void startWithName(String myName) {
    this.myName = myName;
}
```
import java.util.ArrayList;
import java.util.List;
import org.iids.aos.sensoragents.SensorNodeAgent;
import org.iids.aos.sensoragents.channel.Channel;
import org.iids.aos.sensoragents.wivib.WiVibSensorAgent;
import wivib.MeasurementLevel;
import wivib.WiVibNetwork;
import wivib.MeasurementLevel.Traces.Trace;

public class WiVibSensorNodeAgent extends SensorNodeAgent {
    private static final long serialVersionUID = 1L;
    private final String sensorName = "wivibsensoragent.name";
    private final String myName = "firstWivibSensorAgent";
    private static final int sendDelaySecs = 0;
    private static final String CONFIGFILE = "C:\Temp\sample.xml";
    private static final String LOGDIRECTORY = "C:\\Temp";

    private int logCount;

    WiVibSensorService wvSS;
    List<WiVibNetwork.WiVib> wiVibList;

    @Override
    public void run() {
        setup(sensorName, myName, sendDelaySecs);
        super.run();
        try {
            logCount = 0;
            // Get a WiVib service
            wvSS = getServiceBroker().bind(WiVibSensorService.class);
            // Begin the service
            wvSS.initialise("192.168.2.4", 8001);
            wvSS.connect();
            // Send configuration
            wvSS.SendServerConfiguration(CONFIGFILE);
            // Begin thread to poll for data traces
            new Thread(new Thread() {
                @Override
                public void run() {
                    getWiVibList();
                }
            });
        }
    }
}
```
private void getWiVibList() {
    wiVibList = new ArrayList<WiVibNetwork.WiVib>();
    while (true) {
        // Get list of updated WiVibs
        wiVibList = wvSS.getListOfWiVibs();
        int tmpCnt = wvSS.getProcessCount();
        if (!wiVibList.isEmpty()) {
            for (WiVibNetwork.WiVib wiVib : wiVibList) {
                updateChannels(wiVib);
                if (dataService.isInitialised()) {
                    if (tmpCnt != logCount)
                        { dataService.log(channels); logCount = tmpCnt; }
                } else {
                    dataService.initialise(LOG_DIRECTORY, channels);
                }
            }
        }
    }
}
```
try {
    fltTrace = aStringToaFloat(strTrace);
} catch (NumberFormatException ex) {
    getLog().error(ex.getMessage());
    return;
}
Channel cnl = new Channel(channelID, channelName,
    ml.getUnits(), fltTrace);

boolean chlExists = false;
for (int i = 0; i < channels.size(); i++) {
    if (channels.get(i).id.equals(cnl.id)) {
        channels.set(i, cnl);
        chlExists = true;
    }
}
if (!chlExists) {
    channels.add(cnl);
}

private float[] aStringToaFloat(String trace) throws NumberFormatException {
    String[] aStrTrace = trace.substring(1, trace.length() - 2).split(",");
    float[] aFltTrace = new float[aStrTrace.length];
    for (int i = 0; i < aStrTrace.length; i++) {
        aFltTrace[i] = Float.parseFloat(aStrTrace[i]);
    }
    return aFltTrace;
}

C.4.3 CanSensorNodeAgent.java

package org.iids.aos.sensoragents.can;
import java.util.HashMap;
import java.util.List;
import org.iids.aos.sensoragents.SensorNodeAgent;
import org.iids.aos.sensoragents.channel.Channel;
import org.iids.aos.sensorservice.can.CanSensorService;

/**
 * @author jack_higgs
 */
```java
public class CanSensorNodeAgent extends SensorNodeAgent {

    private static final long serialVersionUID = 1L;

    private final String sensorName = "cansensoragent.name";
    private final String myName = "firstCanSensorAgent";

    private static final int expirationDelay = 500000;

    private static final int sendDelaySecs = 0;

    private CanSensorService cnSS;

    @Override
    public void run() {
        setup(sensorName, myName, sendDelaySecs);
        super.run();
        try {
            // Get a CAN service
            cnSS = getServiceBroker().bind(
                    CanSensorService.class);
            // Connect
            if (cnSS.connect()) {
                cnSS.startCanRead(true);
                new Thread(new Thread() {
                    @Override
                    public void run() {
                        pollCan();
                    }
                }).start();
                getLog().info("Releasing CAN service");
                getServiceBroker().release(cnSS);
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
        }
        private void pollCan() {
            while (cnSS.isConnected()) {
                HashMap<String, List<String>> data = cnSS.getData();
                // Only interested in message 1536
                if (data.containsKey("1536")) {
                    try {
                        String[] hexData = data.get("1536").get(2).split(" ");
                        int RPM = Integer.parseInt(hexData[0] + hexData[1], 16);
                        Channel rpmChnl = new
                                Channel(sensorName + "RPM"
"", "RPM", 
"RPM", RPM);  
int Ignition = Integer.parseInt(hexData[2] + hexData[3], 16);  
Channel ignitChnl = new Channel(sensorName + "Ignition", "On/Off", Ignition);  
for (int i = 0; i < channels.size(); i++) {  
    if (channels.get(i).id.equals(rpmChnl.id)) {  
        channels.set(i, rpmChnl);  
    } else {  
        channels.add(rpmChnl);  
    }  
    if (channels.get(i).id.equals(ignitChnl.id)) {  
        channels.set(i, ignitChnl);  
    } else {  
        channels.add(ignitChnl);  
    } 
}  
if (dataService.isInitialised()) {  
    dataService.log(channels);  
} else {  
    dataService.initialise(LOGDIRECTORY, channels);  
    dataService.log(channels);  
}  
}  
catch (Exception ex) {  
    getLog().error("Problem with CAN data processing.");  
    break;  
}
C.5 Sample pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.iids.aos</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <groupId>org.iids.aos</groupId>
  <artifactId>WiVibSensorAgent</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>WiVibSensorAgent</name>
  <properties>
    <org.iids.aos.install.type>agent</org.iids.aos.install.type>
    <org.iids.aos.install.path>C:\local\apps\agentscape</org.iids.aos.install.path>
  </properties>
  <build>
    <plugins>
      <!-- configure agent jar to include classpath + main class -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <configuration>
          <archive>
            <manifest>
              <mainClass>org.iids.aos.sensoragents.wivib.WiVibSensorNodeAgent</mainClass>
            </manifest>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>org.iids.aos</groupId>
      <artifactId>WiVibSensorServiceInterface</artifactId>
      <version>1.0-SNAPSHOT</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</project>
```
C.6 Sample Config: WiVibConfig.xml

```xml
<?xml version="1.0"?>
<WiVibNetwork>
  <WiVib SN="00−C0−1B−0E−67−77">
    <Wakeup>
      <TO>0</TO>
      <WE>0</WE>
      <WI>00:30</WI>
      <WT>00:00</WT>
    </Wakeup>
    <Status>
      <BLOnSample>0</BLOnSample>
      <StartupAction>2</StartupAction>
    </Status>
    <Channels>
      <Channel CH="1">
        <Units>g</Units>
        <Coupling>AC</Coupling>
        <Sensitivity>97.8</Sensitivity>
        <Offset>0</Offset>
      </Channel>
      <Channel CH="2">
        <Units>g</Units>
        <Coupling>AC</Coupling>
        <Sensitivity>100</Sensitivity>
        <Offset>0</Offset>
      </Channel>
      <Channel CH="3">
        <Units>g</Units>
        <Coupling>AC</Coupling>
        <Sensitivity>100</Sensitivity>
        <Offset>0</Offset>
      </Channel>
      <Channel CH="4">
        <Units>g</Units>
        <Coupling>AC</Coupling>
        <Sensitivity>100</Sensitivity>
        <Offset>0</Offset>
      </Channel>
      <Channel CH="5">
        <Units>degC</Units>
        <Coupling>DC</Coupling>
        <Sensitivity>10</Sensitivity>
        <Offset>0</Offset>
      </Channel>
    </Channels>
  </WiVib>
</WiVibNetwork>
```
<Channel CH="6">
  <Units>degC</Units>
  <Coupling>DC</Coupling>
  <Sensitivity>10</Sensitivity>
  <Offset>0</Offset>
</Channel>

<Channel CH="7">
  <Units>degC</Units>
  <Coupling>DC</Coupling>
  <Sensitivity>10</Sensitivity>
  <Offset>0</Offset>
</Channel>

<Channel CH="8">
  <Units>degC</Units>
  <Coupling>DC</Coupling>
  <Sensitivity>10</Sensitivity>
  <Offset>0</Offset>
</Channel>
</Channels>

<Samples>
  <Measurement Name="12">
    <ChannelNum>1</ChannelNum>
    <Units>g</Units>
    <SampleRate>2560</SampleRate>
    <TraceLength>512</TraceLength>
    <AutoGain>0</AutoGain>
    <Gain>1</Gain>
    <Trigger>OFF</Trigger>
    <Tacho>OFF</Tacho>
    <Format>Timewave</Format>
  </Measurement>
</Samples>
</WiVib>
</WiVibNetwork>