# Distributed Operating Systems on Wireless Sensor Networks

A thesis

submitted in fulfilment

of the requirements for the Degree

of

Masters of Philosophy

at the

University of Waikato

by

**Paul Hunkin**


THE UNIVERSITY OF
WAIKATO
*Te Whare Wānanga o Waikato*

University of Waikato

2017

*'Sensor networks are notoriously difficult to program, given that they encompass the complexities of both distributed and embedded systems.'*
– David Chu [17]

*'The computer should be doing the hard work. That's what it's paid to do, after all.'*
–Larry Wall

*'Progress is made by lazy men looking for easier ways to do things.'*
– Robert A. Heinlein

# Abstract

This thesis proposes the use of traditional distributed operating system and distributed systems techniques that are adapted and applied to the wireless sensor network domain. These techniques are applied to the creation of a wireless sensor network operating system that allows complex applications to be created without special programmer knowledge of sensor network programming or architecture. The resulting system is capable of executing a high level user application written in conventional single-system-image form, without the user being aware of the mesh architecture or underlying sensor node hardware.

A wireless sensor network is a collection of battery-powered embedded systems that communicate over low-bandwidth radio. Because of their limited hardware, niche deployments and use of embedded processors, programming techniques for wireless sensor network nodes are generally relatively esoteric compared to most software programming tasks. This can be relatively complex for programmers not familiar with the wireless sensor network domain. A naive approach to writing a wireless sensor network application may well result in considerably reduced battery life due to inefficient use of the limited power resources, requiring an expensive and time-consuming replacement or patching process.

As a result of this complexity, traditional wireless sensor network applications are written as simply as possible. The majority of these applications simply move passive data readings back across a mesh to a more powerful server. While this is a sufficiently effective approach in some situations, for other sensor network deployments involving large amounts of complex data it is more efficient for the sensor network application to process at least some of the data inside the mesh[91], saving on unnecessary data transmissions. However in the real world, the complexity of writing such an application in many cases precludes this from being created. An operating system that provides power-efficient distributed processing while presenting a more standard unified single system image to the application developer would provide new possibilities for sensor network application developers in terms of creating dynamic and complex sensor network applications.

This thesis covers the design decisions, development process and evalua-

tion of the Hydra distributed wireless sensor network operating system, an operating system that provides these services. The system is evaluated in the form of a scenario for monitoring intruders over a large area using accelerometer monitoring[95] – during this scenario, power efficiency is gained due to the intelligent Hydra operating system services, as the resulting accelerometer data is not moved across potentially multi-hop network links. Application code complexity is also reduced due to the higher-level single system image programming environment.

# Acknowledgements

*'Not all those who wander are lost'*
– J R R Tolkien

Thanks to everyone who helped in the writing process.
If you're reading this, you're probably one of them – so, thanks!

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

> *'Sensor networks are notoriously difficult to program, given that they encompass the complexities of both distributed and embedded systems.'*
> – David Chu (EECS, Berkeley) [17]

## 1.1 Computing trends: smaller and more numerous

Since the heyday of mainframe computing in the 1980s and early 1990s[59], computing hardware and software has increasingly trended towards increasingly capable and numerous smaller devices rather than one large monolithic system. While each individual smaller device may be less powerful in computation terms than larger devices, their low individual cost and flexibility, along with increasing availability of network bandwidth, has ensured their popularity.

This trend has been seen in the consumer electronics space with the availability of affordable smartphones[25], laptops and the more recent arrival of niche networked applications – for instance, smart home automation tools such as Nest[62]. The advent of 'cloud computing'[64] is moving heavy computation

into a set of low cost, relatively interchangeable distributed servers, where a task is broken down automatically and a number of servers are load-balanced to solve the project cooperatively[5].

This trend is driven by increased efficiency in production of core computer hardware components such as CPUs, solid state memory modules and radios. Prices for these items have dropped dramatically in recent decades, allowing cheap mass production of small efficient devices.

This growth in low cost computing has made possible a whole new class of networked computer system – the 'Wireless Sensor Network'.

## 1.2 Sensor networks

In 1998, the 'Smart Dust'[72] project pioneered the concept that became the modern wireless sensor network[2]. Supported by DARPA, The Smart Dust project aimed to build a complete sensing and communication system in one cubic millimeter[39]. The project was conceived to serve diverse applications such as battlefield surveillance and monitoring, inventory management and environmental sensing.

A more complete exploration of the history of sensor networks is included in the 'Wireless Sensor Networks Background' chapter.

The legacy of the hardware that the Smart Dust project pioneered continues to this day. While today's typical wireless sensor nodes are generally larger than the Smart Dust goal of one cubic milliliter, nodes are still much lower powered than conventional smartphone or other mobile hardware platforms. While a consumer-range[40] low-end-tier 2015 era mobile phone contains a gigabyte or more of RAM and a radio capable of communicating at multi-megabit speeds, a typical wireless sensor network node[46] contains only a

Figure 1.1: A MSB430 Scatterweb sensor node

few kilobytes, a limited-capacity processor such as the TI MSP430[36] and a radio that transmits orders of magnitudes slower than typical consumer data connections. By limiting the hardware these nodes are able to be mass-produced at relatively low prices.

A wireless sensor network is made up of many individual nodes. Historical trends and future predictions show sensor network nodes staying at approximately the same amount of computing power per node, with decreasing per-node manufacturing costs as manufacturing processes become more efficient. As such, we can expect sensor network meshes to become both larger and more common.

Below is an illustration of a typical sensor node – specifically the MSP430-based Scatterweb[73] node. This node was chosen for the target hardware platform due to its range of sensors, relatively low cost and wide availability, and also because it was specifically made for teaching wireless sensor network concepts. As such it is quite a flexible platform.

A more complete analysis of the MSP430 node and its specific hardware features is contained in the 'Implementation' chapter, where the process of building the sensor network operating system for the Scatterweb platform is dis-

cussed. However it should be noted that the node contains 8kbytes of RAM and a micro-controller operating at 16 MHz, and is powered by a trio of AAA batteries. This approximate hardware configuration is the typical wireless sensor network node as addressed by this thesis.

## 1.3 Why are sensor networks important?

Sensor networks are one of the fastest growing areas in both research and commercial development. Sensor networks have been used around the world for many important tasks. Deployments range from just a few nodes to thousands.[3]. A few sample deployments are outlined below, which cover a range of node counts and tasks.

1. Meteorology and hydrology monitoring in Yosemite National Park, USA[56].
2. Tracking animals in the wild in Kenya[44].
3. Intrusion detection to certain geographical areas in Sweden[95].
4. Habitat monitoring of nesting birds on an island in Maine, USA[68]
5. Collecting temperature and movement information in modern buildings[74].

As can be seen from the uses above, sensor networks enable us to perform large scale operations with data over a distributed geographic area. This was much more difficult to accomplish before sensor networks became common, requiring more resource-intensive data gathering techniques.

## 1.4 Traditional sensor network applications

> *'Despite the significant effort made, successful deployments and real-world applications of sensor networks are still scarce, labor-intensive and often cumbersome to achieve'*[10]

Recent decades have seen order-of-magnitude improvements in hardware cost

and efficiency. However the software programming techniques and application tools used for building user applications for sensor networks have not improved at the same rate[47]. Traditionally, large-scale reliable distributed system programming is relatively complex. As such typical large distributed data processing systems usually present an abstraction layer to the user, hiding the complexity inherent in the distributed operations. A well-known example of a 'big data' distributed processing system that takes this approach is Apache Hadoop[34], which allows efficient distributed data set processing across a computer cluster. It does this by exposing a simple programming model that abstracts away the underlying hardware.

Distributed systems programming complexity is greatly magnified when combined with the challenges inherent to low-memory, low-power embedded systems. As such, the difficulty of sensor network programming is a constraining factor in the widespread adoption of sensor networks and in many cases has limited the complexity and capabilities of sensor network applications. While many research and commercial projects have successfully approached this problem by providing a user-friendly interface for reading data from a sensor network, if an application is intended to be more complex than invoking simple data collection APIs then the system must be programmed using various low-level programming techniques, where network transactions and classic distributed systems problems must be handled manually and in a power-efficient manner[47].

A popular low-level programming language for wireless sensor networks is Nes-C[31], an extension to the C language that allows component-based event-driven programming. Nes-C is the programming language used for application development in the TinyOS[51] operating system, the most widely deployed sensor network operating system. The combination of TinyOS and Nes-C is extremely flexible, compact and power-efficient and as such has yielded considerable success in both academia and commercial deployment scenarios in the

last decade[52]. However for best results it requires an application programmer who is familiar with the sensor network domain and the Nes-C language. Experience in other domains such as smartphone application development may not be sufficient. As such the bar for sensor network application deployment is relatively high, damaging the potential of sensor networks as a concept.

```
 module RealMainP @safe() {
  provides interface Boot;
  uses interface Scheduler;
  uses interface Init as PlatformInit;
  uses interface Init as SoftwareInit;
}
implementation {
  int main() @C() @spontaneous() {
    atomic
      {
    platform_bootstrap();

    call Scheduler.init();
    call PlatformInit.init();
    while (call Scheduler.runNextTask());
    call SoftwareInit.init();
    while (call Scheduler.runNextTask());
      }

    __nesc_enable_interrupt();

    signal Boot.booted();

    call Scheduler.taskLoop();
    return -1;
  }

  default command error_t PlatformInit.init() { return SUCCESS; }
  default command error_t SoftwareInit.init() { return SUCCESS; }
  default event void Boot.booted() { }
}
```

Figure 1.2: An example of Nes-C code

When programming sensor network applications, energy-efficiency is the most overriding concern. A typical sensor network node uses a low-capacity battery – in the case of the MSP430, three AAA cells. As such the power supplies of sensor network nodes can be exhausted very quickly if care is not taken to avoid this.

In addition to this, a number of other attributes are widely accepted as being part of the core requirements of an effective complex wireless sensor network programming model[1]. These requirements must therefore be part of any

effective sensor network operating system. These additional attributes are:

- Efficiency – the system must make efficient use of the wireless sensor network hardware resources, including the aforementioned battery reserve, but including RAM and flash memory (if available). Because of the highly limited nature of the hardware, performance can easily suffer under the impact of the communications.
- Scalability – the mesh as a whole must be able to scale to different demands and node populations.
- Localization – the system needs to be capable of utilizing the resources of the whole mesh, with the ability to spread application tasks across the devices.
- Synchronization – Sensor nodes need to cooperate to send back data, if only to avoid duplication, to keep various timers in sync and to perform multi-hop networking where needed.

It is useful to evaluate a distributed sensor network operating system against these requirements.

## 1.5 Distributed Systems

Distributed processing systems (such as the previously mentioned Hadoop) exist to efficiently process data over multiple nodes. However several key differences between this domain and wireless sensor networks exist:

- They are used for larger data loads – multiple terabytes of data is not uncommon.
- They are utilized on much more powerful hardware than a typical wireless sensor network node.
- Networking speeds are much higher, generally with physical fiber or Ethernet links rather than low-bandwidth radios.

- Energy is not a scarce resource.
- Individual nodes are not as important, as any node can (generally) process any data load identically, and does not contribute its own unique data. On sensor networks the choice of node can be more important, as a node may have different sensor data or power reserves or requirements based on its geographic location in the mesh.

However many of the techniques used for distributed processing can still be used. More specifically, the more specific distributed operating system domain (rather than generalized distributed processing) is useful in the design of a wireless sensor network distributed operating system, as for efficiency and simplicity it is desirable to do these operations on the operating system level rather than as part of a user application.

### 1.5.1 Distributed Operating Systems

The concept of the distributed operating system first appeared in 1954, in the description for a general purpose computing system called 'DYSEAC[50]. This document contained the description of a peer to peer multi-computer operating system, that would 'coordinate the diverse activities of all the external devices into an effective ensemble operation. Several other distributed operating systems were developed around this time. However, none achieved widespread success[93].

Much more widespread distributed operating system research was done in the 1980s, with systems such as Amoeba[84] providing a general-purpose free distributed operating system. Interest waned in the 1990s as more powerful hardware and increasingly complex software reduced the desire and ability to use the distributed computational model. In more recent years, the availability of inexpensive high-bandwidth connections has brought about the rise of 'cloud computing - a movement back towards distributed systems, making use

of hardware in remote data centers.

For more discussion of the history of the distributed operating system and modern distributed operating system design and implementation, see the 'Background' chapter.

A distributed operating system is defined as 'the logical aggregation of operating system software over a collection of independent, networked, communicating, and spatially disseminated computational nodes[83]. In other words, a distributed operating system provides an abstraction layer that (from the point of view of an application), a number of computers look like a single computer. This abstraction greatly reduces the complexity of the applications built on top of it, as they gain the distributed computation benefits without incurring the complexity costs.

This abstraction is the core distributed operating system concept, often termed 'macro-programming'[12]. In addition to this, a number of other attributes are widely accepted as being part of the core requirements of an effective distributed operating system[94][76]. These additional attributes are:

- Performance – the system must make efficient use of the distributed hardware resources. Because of the highly distributed nature of the system, performance can easily suffer under the impact of the communications.
- Reliable – the system must expect and correctly handle hardware errors. Distributed hardware means that the chances of failure of one component is much higher.
- Resource name resolution – a way of resolving hardware devices as needed.
- Resource management – the system needs to be capable of utilizing the resources of the whole mesh. This is also referred to as 'load sharing, as the application tasks are spread across the devices.
- Process management – a typical distributed operating system manages a set of processes.

- Synchronization – Concurrent processes inherently need to cooperate, and this must be done in a synchronized fashion to avoid errors and deadlock.
- Flexibility – the operating system must be adaptable to a range of conditions and deployments.

As an operating system must address all of these attributes in order to be considered a true distributed operating system, they are key to the design of a wireless sensor network distributed operating system. Many of these attributes also overlap with the desired attributes of a wireless sensor network operating system.

## 1.6 Towards a distributed wireless sensor network operating system

At this point it is useful to consider what a middle ground between highly-abstract data collection and low-level sensor device programming would consist of – a system that provided enough of an abstraction layer to efficiently hide the distributed computation tasks inherent in wireless sensor network programming, yet was flexible enough to allow general-purpose computing for complex applications, while simultaneously making efficient use of limited energy reserves.

This thesis seeks to discover if the distributed operating system paradigm could be useful in building such a system, helping remove some of the user application programming complexity on wireless sensor networks. By utilizing a distributed operating system, the bulk of the complexity is removed from the user applications and application developers can be presented a familiar programming model using commonly deployed programming languages and APIs, conceptually treating the entire sensor mesh as a single logical computer.

## 1.7 Problem statement and scope

An examination of the following statement is presented in this dissertation:

> 'By adapting distributed operating system techniques for the wireless sensor networks domain, wireless sensor network user application programming complexity can be decreased. These applications can later be executed on a sensor network mesh in an energy-efficient manner.'

To aid in this examination, a distributed operating system for wireless sensor networks called 'Hydra' was designed, developed and evaluated. The examination of Hydra is addressed by discussing the following topics:

- **Theoretical basis**: For distributed operating systems on wireless sensor networks to be useful, it is necessary that the techniques used be applicable and adaptable to the sensor network domain, typical deployment tasks and characteristics of the data collected. This is investigated by reviewing both current sensor network operating systems and deployments, and traditional distributed systems techniques in the Background chapter. The specific techniques that are most useful from both domains and the approach taken to create a distributed operating system based on them are then identified in the Design chapter.
- **Feasibility**: Using distributed operating system methods on a sensor network may be theoretically possible, but it needs to be demonstrated that doing so is practical. The Implementation chapter discusses the decisions made while creating the Hydra software, and any compromises to the design that were required as a result of moving from a theoretical design to a real-world development.
- **Suitability**: The Deployment chapter discusses the choice of a test scenario. The test scenario is the main method of evaluating the effectiveness of the wireless sensor network distributed operating system. As

such it should be emblematic of a typical wireless sensor network task, as outlined briefly in this chapter and in more depth in the Background chapter.

- **Evaluation**: The Evaluation chapter discusses the results of the test scenario, focusing on correctness, performance and scalability. The previously outlined criteria for attributes of efficient wireless sensor network operating systems as well as distributed operating systems will also be used to evaluate.

The conclusions to this thesis are presented in the Conclusion chapter. The possibility of future work and improvements to the system is also discussed.

## 1.8 Availability

The Hydra implementation includes the features discussed in the design chapters of this thesis, though at the time of writing it is not yet stable enough to be deployed in production. The development code is available in the public Subversion repository:

`http://bieh.net/svn/hydra`

# Chapter 2

# Distributed Operating Systems

> *‘ However, as every parent of a small child knows, converting a large object into small fragments is considerably easier than the reverse process... ’*
>
> – Andrew S. Tanenbaum, ‘Computer Networks’ 4th ed.

This chapter presents a discussion on existing distributed operating systems and their design choices, with the intent of establishing the primary characteristics of existing distributed operating systems. Once the characteristics of the existing systems have been established, this knowledge is used in the later Design chapter to evaluate how the two areas of sensor networks and distributed operating systems can be combined above and beyond the existing work to be applied to one or many of the common wireless sensor network use-cases.

Distributed operating systems and wireless sensor networks are two research areas that have historically remained relatively separate, with limited overlap between the two.

This thesis seeks to properly evaluate how effectively the distributed operating system paradigm can be applied to wireless sensor networks. Therefore this chapter first reviews the established research to:

- Define a distributed operating system, outlining typical hardware and

deployment configurations.

- Identify the core attributes and historical context of the classic distributed operating system.
- Identify more recent trends in modern software implementation and deployments that have applied these classic distributed operating systems techniques to other areas.

## 2.1 What is a distributed operating system?

This thesis presents the design and evaluation of a distributed operating system for a wireless sensor network. As such, the concept of a 'distributed operating system' must first be defined.

A distributed operating system is 'the logical aggregation of operating system software over a collection of independent, networked, communicating, and spatially disseminated computational nodes[83]. In other words, a distributed operating system provides an abstraction layer that allows a number of computers to look like a single computer.

This abstraction is the core distributed operating system concept. In addition to this, a number of other attributes are widely accepted as being part of the core feature set of a complete and effective distributed operating system[76]. These additional attributes are:

- Performance – whether the system makes efficient use of the distributed hardware resources. Because of the highly distributed nature of the system, performance can easily suffer under the impact of the communications.
- Reliable – whether the system expects and correctly handle hardware errors. Distributed hardware means that the chances of failure of one component is much higher.

- Resource name resolution – whether the system provides a way of resolving hardware devices as needed.
- Resource management – whether the system is capable of utilizing the resources of the whole mesh. This is also commonly referred to as 'load sharing, as the application tasks are spread across the devices.
- Process management – whether a set of processes can be handled, or if the system allows only one task.
- Synchronization – as concurrently executing processes inherently need to synchronize on some level (if only to share access to shared hardware resources), this must be done in a coordinated fashion to avoid errors and deadlock.
- Flexibility – whether the system is adaptable to a range of conditions and user application requirements (ie: it should be generalisable to a range of tasks).

An operating system implementation must address all of these attributes in order to be considered a useful distributed operating system. Therefore they are key to the design of a capable wireless sensor network distributed operating system.

## 2.2 Classic distributed operating systems

As stated in the introduction, the concept of the distributed operating system first appeared in 1954 as part of the description for a general purpose computing system called 'DYSEAC[50]. This document contained the description of a peer to peer multi-computer operating system, that would 'coordinate the diverse activities of all the external devices into an effective ensemble operation. Several other distributed operating systems were developed around this time. However, none achieved widespread success[93].

Much more widespread distributed operating system research was done in the 1980s, with systems such as Amoeba[84] that provided a general-purpose free distributed operating system. Interest waned in the 1990s as more powerful hardware and increasingly complex software reduced the desire and ability to use the distributed computational model. In more later years, the availability of inexpensive high-bandwidth connections brought about the rise of 'cloud computing – a movement back towards distributed systems, making use of hardware in remote data centers. This trend is discussed later in this section.

The original pieces of computing hardware on which these early distributed systems operated were extremely rudimentary in terms of processing capacity and memory limitations – much like that of a modern wireless sensor network node (as will be specified in more detail later in this section). As the creators of these original distributed operating systems were still able to accomplish complex distributed systems tasks despite these limitations, it suggests that accomplishing similar tasks on sensor network nodes may be in principle possible despite the limited CPU and memory capacity of sensor node hardware.

### 2.2.1 Classical distributed operating system computational models

There are a number of main computational models that are used by distributed operating systems. These computational models refer to the layout of nodes in the distributed system, and the control flows between them. When designing a sensor network distributed operating system, the computational model chosen will be central to the design. Therefore it is paramount that we select the correct model.

The primary computational models adopted by traditional distributed operating systems are the **centralized**, **decentralized**, and **distributed**[86][92] models. Each model is outlined in turn.

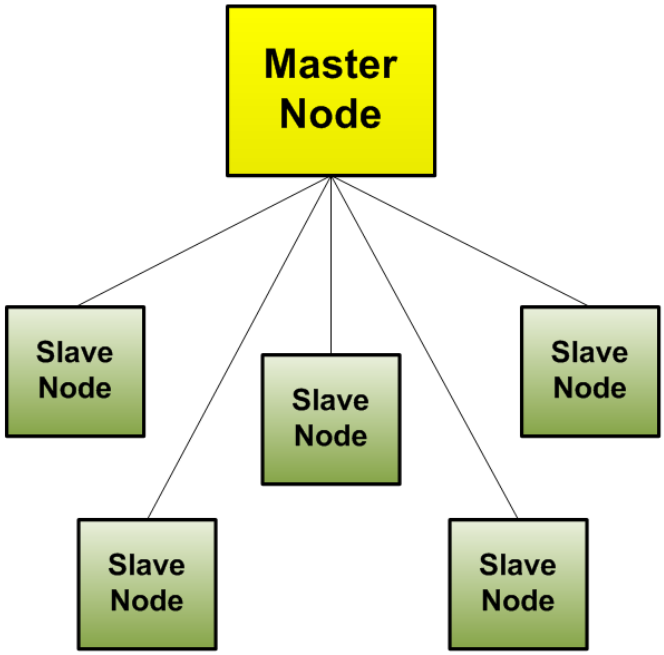


Figure 2.1: A centralized layout[92]

#### 2.2.1.1 The centralized model

The **centralized** model has one master node, with all other nodes controlled by the master. This is the simplest computational model. An example is shown in Figure 2.1

Because there is one known master node, there is no need for complex decision-making. This means that the primary advantage of this architecture is simplicity – both in terms of development time and run-time overhead. However the system can be adversely affected if the master node encounters a problem, as it is a single point of failure. In addition to this, there may be considerable network overhead caused by the need to move all network transmissions to and from one central point, which may be far removed from where the transmissions originate. This may mean that the system is less efficient in terms of power usage and memory overhead due to having to pass messages through intermediate nodes, and the master node may exhaust its computational or

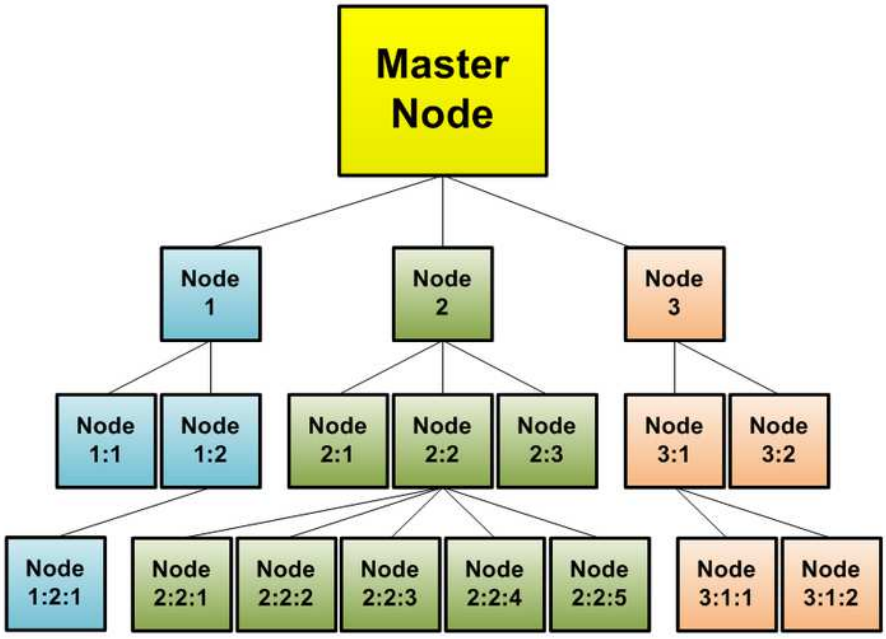


Figure 2.2: A decentralized layout[92]

power resources quickly due to the increased load placed upon it.

#### 2.2.1.2 The de-centralized model

The **decentralized** model is similar to the centralized one, but can exist in layers - a master node of one layer can be a slave of a higher-level master node. However, there is ultimately one node at the top of the hierarchy. An example is shown in Figure 2.2.

The decentralized model tries to address one of the main failings of the simpler master-node model – the requirement to move all messages to a single point. This is accomplished by attempting to handle as much processing as possible at lower levels in the hierarchy, which if successful will reduce overall network load and individual node load on the nodes higher in the hierarchy. However, this comes with a complexity cost, as knowledge of the node structure must exist. Additionally, while there is still one single master node that can fail, the

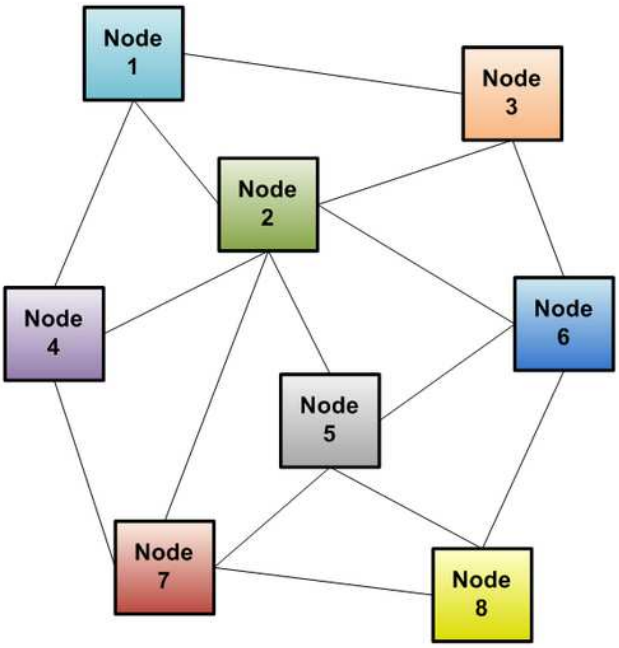


Figure 2.3: A distributed layout[92]

case of nodes lower in the hierarchy must be handled correctly as well, with their responsibilities shifted as needed.

#### 2.2.1.3 The distributed model

The **distributed** model has no hierarchy - nodes are connected in a peer-to-peer mesh. This provides the most flexible layout, but is complex to manage. An example is shown in Figure 2.3.

The peer to peer model is the most resilient in terms of protection against failure and can be the fairest when it comes to spreading load over the mesh. This makes it ideal for many sensor network applications, as these attributes are often key to a successful real-world sensor network deployment. However, constructing a distributed peer to peer mesh is complex. Nodes must often maintain at least partial knowledge of the state of the mesh, and managing out of date information is a complex problem. If care is not taken when designing

and implementing the system, the positive benefits of a distributed layout can be subsumed by the overhead caused by increased complexity.

### 2.2.2 Reliability

Ensuring the reliability of a distributed operating system is extremely important. A large amount of research has been put into developing a number of different reliability techniques for distributed systems and distributed operating systems in particular[89][87]. When designing a distributed operating system for wireless sensor networks reliability will need to be a key consideration, as real-world scenarios for sensor networks typically require a system that can complete the task reliably. The following discussion outlines the existing reliability techniques that can be applied to sensor networking.

In general, the process for ensuring reliability can be divided into two sections – error detection and error handling.

#### 2.2.2.1 Error detection

Error detection is the process of determining whether a particular system is in an ‘error state. There are a number of techniques that can be used for this.

- Replication checks - a system can perform the same task simultaneously in multiple locations, checking to make sure that all the results are consistent. If an inconsistency is detected, then the system is in error.
- Timing - a system can make sure that a result is delivered inside a certain time window. If a timeout occurs, the system is in error.
- Constraints - a system can make sure that the result is inside some predetermined constraint window. If its outside, the system is in error.
- Background diagnostics - a system can perform run-time checking of any processing that is occurring. The type and effectiveness of diagnostics

that can be performed depend on the application.

Once an error state has been detected, the task of the system then becomes error handling

#### 2.2.2.2 Error handling

The goal of error handling is to restore the system to a 'safe state, pre-error. Two main techniques have been developed to perform this task.

- Rollback – the operations that brought the system into the error state are applied in reverse order. This – assuming a deterministic system – will cause the system to return to a safe state.
- Checkpointing – the system periodically is required to snapshot the entire state of the system, allowing a previously valid state to be restored.

Once the system has been restored from the error state, normal functioning can resume. The choice of the most appropriate reliability technique is heavily dependent on the individual requirements of the distributed operating system being developed – the type of environment it runs in, the types of tasks being performed, and so on.

## 2.3 Distributed operating system concepts applied in modern cloud systems

In recent years, research and commercial interest in conventional full-featured distributed operating systems has waned. This can be attributed to a number of factors, primarily the rise of cheap, powerful computing hardware that made it possible to consolidate large workloads on to one physical machine.

However variations on various concepts first developed in the distributed op-

erating system domain are still in use today in more modern software systems. These typically exist as platform or application layer services, rather than working on the operating system level, and are primarily found in large-scale server deployments to handle datasets with processing requirements too large to efficiently be contained on one physical machine, or to move computation off devices with comparatively fewer resources. These use-cases are expected to continue to become more and more common over time[49] as mobile devices such as smartphones become more widespread and more intelligent power usage techniques are required.

### 2.3.1 Automatic scaling

Various Platform As A Service (PAAS) providers allow automatic scaling of application level code across multiple servers. This provides 'the illusion of infinite computing resources available on demand'[6]. While of course in practice the computing resources available are finite, this technique is useful as it allows a level of transparent scaling to be achieved horizontally across relatively cheap computational hardware.

PAAS providers such as Heroku[35] or Microsoft Azure[8] create multiple virtual machines and use distributed operating system techniques to provide an abstraction layer allowing user applications to be automatically shared, partitioned and duplicated across multiple virtual machine instances. Higher level systems such as Google App Engine[30] go even further, requiring that all applications be written to a specific framework to provide completely programmer-transparent scaling.

At a lower level of abstraction, load balancing systems under Amazon Web Services[7] can be configured to redistribute tasks to nodes most suited to each task, based on some heuristic defined by the application. In this case the application must be manually configured to receive tasks, which can lead to

more efficient layout than a fully transparent method if sufficient care is taken to define the heuristic and distribution strategies. However this means more of the distributed systems burden is placed upon the application programmer.

### 2.3.2 Fault tolerance

Techniques developed as part of distributed operating system fault tolerance systems are used in many commercial and open source data storage systems. A common example of this is the 'distributed database', a database that is under the control of a central database management system (DBMS) in which storage devices are not all attached to a common CPU.[69]. These distributed database systems use replication and error detection techniques common in distributed operating systems to achieve consistency.

### 2.3.3 Conclusion

This chapter briefly summarizes the existing literature and research in the distributed operating system domain. It outlines the existing design and implementation strategies used in both fields.

The next chapter describes the wireless sensor network domain.

# Chapter 3

# Wireless Sensor Networks

> *'Negotiating the design space for WSNs can be daunting even for the experienced designer.'*
>
> – Wireless Sensor Networks: From Theory to Applications[42]

This chapter presents a discussion on an overview of existing wireless sensor network programming operating systems and their respective programming models.

In order for this thesis to evaluate how effectively the previously-established distributed operating system paradigm can be applied to wireless sensor networks, it is necessary to first review the established research. This provides:

- A definition of a wireless sensor network, outlining typical hardware and deployment configurations.
- An outline of the basic differing approaches in wireless sensor network operating system design, examining the design choices made by the most widely-deployed sensor network operating systems.
- An overview of the broad categories of sensor network deployment, identifying which type of sensor network software is typically deployed on each deployment category.

Once the characteristics of the existing software systems and the fundamental

different types of sensor network deployments have been established, this thesis then provides the context to evaluate how the two areas of sensor networks and distributed operating systems can be combined above and beyond the existing work to be applied to one or many of the common wireless sensor network use-cases.

## 3.1 Wireless sensor networks

Before developing a sensor network distributed operating system, it is useful to survey the existing non-distributed operating systems that exist for sensor network platforms. By examining the design choices of existing wireless sensor network software platforms, this thesis seeks to identify the fundamental requirements of sensor network operating systems to evaluate the Hydra sensor network operating system in terms of these requirements. The decisions made by these existing systems help inform the decisions made when designing a *new* sensor network operating system, as many of the same requirements and restrictions will apply to a new system, regardless of if it adapts distributed operating systems techniques. This section primarily addresses traditional sensor network operating systems rather than distributed systems specifically built for sensor networks. Projects that provide some forms of distributed systems services on sensor networks are covered in the next section.

Because wireless sensor networks are a relatively new area and one that has significantly different requirements than most other areas of software development, no standard paradigm for programming wireless sensor networks has emerged. Many systems have been developed to run on wireless sensor nodes, which can be divided into three categories.

### 3.1.1 Single node operating systems

The first wireless sensor network applications were written to directly use the embedded hardware, with no operating system abstraction. While this approach makes efficient use of the limited hardware of the wireless sensor nodes, it was not particularly efficient in terms of programmer time and energy. Small operating systems have became popular, some designed specifically for wireless sensor networks and some developed for generic embedded systems and ported to sensor node hardware. While these operating systems are very simple compared to more common OS's such as Linux or Windows, they perform roughly the same role - to provide the programmer with a simpler abstraction than the underlying hardware, resulting in application software being easier to write and maintain.

Applications written to run on all of these operating systems operate on a single sensor node - the programmer must be aware of this, with interactions with other devices explicitly specified in the design and implementation phases of application development. This approach maps most closely to traditional single-machine programming models for non-sensor network platforms, and as such is often simpler for application software development.

#### 3.1.1.1 Component-based programming model

TinyOS[51], was developed in 2000 and is currently the most widely used operating system for wireless sensor networks[52]. TinyOS is a component based operating system, written in a language called NesC. NesC is based on the C programming language but with language and runtime extensions for the type of programming required for TinyOS.

TinyOS is based on the concept of connecting various types of 'components' together, which provide abstractions for various operations such as reading

from sensor devices or sending network packets. It is completely non-blocking and event based. Any operation that takes longer than 200 milliseconds must be implemented with callbacks. The TinyOS operating system does not provide any method of threading or the conventional process abstraction, which can lead to added complexity when implementing longer running CPU intensive tasks. However, several third-party extensions exist for adding lightweight thread support to the OS[21]. It has also been extended to support run-time reprogramming of sensor nodes, which allows for dynamically updating application code[70].

Support for conventional tasks/processes with blocking execution contexts is a long standing historical debate among wireless sensor networks operating system researchers. If the operating system supports these, it makes application programming significantly easier in many cases. However, this comes with a memory, processing and power overhead compared to a pure event based system, which is undesirable on hardware as constrained as wireless sensor nodes.

#### 3.1.1.2 Multiprocessing programming model

Despite this concern, several operating systems have been developed for wireless sensor networks that *do* provide various types of processes/tasks and blocking execution. The Contiki[22] operating system provides protothreads[23], a lightweight threading abstraction that saves memory by not storing the execution context of each thread. MANTIS[11] is another operating system for wireless sensor networks that provides more traditional threads, with pre-emptive multitasking and full context switching while still striving to maintain a small memory footprint of less than 500 bytes of RAM. LiteOS[14] is a more recent system built on similar principles.

### 3.1.2 Virtual machines

Application virtual machines such as the Java Virtual Machine and the .NET Common Language Runtime have long been popular on both conventional consumer desktop and higher end server systems. They provide platform independence and program isolation. In recent years this technique has been used in sensor node programming to assist with the problem of *reprogrammability* - the ability to dynamically inject new program code into a sensor node in order to deploy updates to existing code or entire new applications. Virtual machines are a useful technique to provide this functionality as their bytecode is more easily moved, modified and relocated at run-time than traditional native machine instructions. The bytecode is also typically smaller in size than native code, increasing the efficiency of moving application code over the network link.

Mate[53] is a stack-based virtual machine that interprets application-specific bytecode - the virtual machine is designed for running a specific class of applications, which lets each bytecode instruction represent a relatively large amount of functionality. This technique necessarily limits the flexibility of the system, but lets the program be represented in a very small amount of virtual machine code, which reduces power and storage requirements.

VMStar[48] is a framework for allowing easy creation of application specific virtual machines similar to Mate.

Another use of virtual machines on wireless sensor networks is similar to the use on more conventional platforms - hardware abstraction, providing a common platform to write to across multiple hardware devices. The 't-kernel'[32] is a wireless sensor network operating system kernel implementing a virtual machine that provides this feature. It also provides additional features that are not provided by the underlying hardware such as memory protection and virtual memory.

The Java Virtual Machine specification has been implemented on sensor network hardware as an alternative to more limited virtual machine systems. These implementations typically do not provide the entire Java stack, but still allow use of some of the Java framework. Popular implementations include Darjeeling[13] and SunSPOT[81][75].

These virtual machines are typically used to support execution using the single-node operating system model.

### 3.1.3 Sensor network deployments

Sensor networks have a number of applications, the characteristics of which inform the choice of application software and operating system design deployed on the sensor network[79]. Though sensor networks are generally focused on data collection, the type of data and the event patterns associated with the data can vary widely between deployments.

Sensor network applications can be broadly categorized into two categories:

- 'Static data, where nodes passively collect data periodically and return them to a central hub for processing. Examples of these include temperature monitoring, humidity monitoring, and other environmental attribute monitoring systems.
- 'Dynamic data, where nodes are waiting for a specific event to happen to a node or group of nodes before processing it. Examples include road traffic monitoring, intruder detection or landslide detection.

As a result of these different characteristics, a sensor network operating system may perform better on one category of application over another. A useful distributed sensor network operating system could adapt to either category – existing work in the area has concentrated on dynamic reconfigurability for existing sensor network systems on the application level[82].

## 3.2 Existing distributed systems services on sensor networks

Before beginning the Hydra project, a review of the existing literature was conducted, searching for embedded operating systems that:

- provided the full distributed operating feature set as described previously in this chapter
- operated on typical wireless sensor network hardware as described in section 2.2

No wireless sensor network operating system that fulfilled these requirements was found.

However, there are examples of operating systems that provide at least *some* distributed system services for 'wireless sensor platforms'. These generally are designed to operate on hardware more similar to conventional desktop or laptop systems than the considerably more restrained hardware environment of most wireless sensor nodes. However they provide some valuable insights into what is currently possible on at least superficially similar systems, and so they are discussed here.

A useful technique when writing distributed applications is to have the ability to perform operations on a number of nodes at once. This makes it simpler to perform generic operations across the mesh. A number of systems provide this abstraction.

This approach can make programming sensor network applications simpler as less knowledge is needed about the underlying network and hardware. The cost is less flexibility in writing complex applications as the details of the lower-level system are not available.

### 3.2.1 Group-level abstraction

The concept of a group-level abstraction in wireless sensor network programming is to provide a method for dividing up the mesh of sensor nodes into logical groups, then performing operations on those groups rather than on the individual nodes. This technique hides the details of communication between the nodes. It is useful for implementing various types of 'localized algorithms', a term specifying operations where a sensor node is limited to interacting with only the nodes in its group.

Various criteria can be used to allocate node groups. The most common method is to use the physical location of the node - this works well to minimize network hops in sensor networks given their wireless mesh topology. Another method is to define the groups based on some other property of the node, such as the type of sensors or the energy levels remaining in the node. Groups can be static or dynamic, depending on configuration.

A well-known example of a system that allows this segmentation is FACTS[88], which allowed researchers to perform in-network distributed event monitoring and analysis using a hierarchical layer of nodes[95].

### 3.2.2 Network-level abstraction

The network-level abstraction concept treats the entire sensor network as one abstract machine. There are two main approaches in this category - the database abstraction and macro-programming.

Treating the sensor node as a database is a useful metaphor as sensor nodes are often used to collect data. TinyDB[58] and Cougar[29] allow programmers to issue SQL-like queries over the sensor mesh, with intelligent caching, query dissemination and data acquisition techniques to minimize the power requirements of each query. The database abstraction is an intuitive interface

to sensor nodes that is easy to use for simple data collection - a use case that covers a large subset of sensor node applications. However, these queries tend to be insufficient for implementing more complex systems.

Macroprogramming attempts to address this limitation by providing a more flexible method of creating sensor node applications that work at the network level. Regiment[63] is a functional programming language similar to Haskell that is specifically designed for macroprogramming wireless sensor networks. The functional language choice makes it easier for the compiler to create the node-specific code that eventually runs on the individual sensor devices. Kairos[33] is another similar system that provides an extension to existing programming languages with 'var@node' syntax to allow shared memory abstraction across nodes. This lets programs that run on individual nodes to have limited access to hardware on other nodes. TrainSense[77] and Net-Tree[91] are middleware systems which provide intelligent placement of computation across the network based on heuristics such as power usage.

A challenge in the network-level abstraction is the problem of referring to mesh resources. Spatial Programming[80] is a system that allows resources to be referred to by their physical location - for example, 'Hill1:microphone[0]' for referring to the microphone sensor node on 'Hill1'. This is a similar naming problem to that faced by distributed operating systems.

The various forms of network-level abstraction, in a similar fashion to the group-level abstraction, simplify the task of the application programmer. This again comes at a price of flexibility - it is often difficult to implement complex applications that run on the mesh using the network level abstraction, due to limiting or cumbersome programming methods.

### 3.2.3 Conclusion

This chapter briefly summarized the existing literature and research in the wireless sensor network and distributed operating system domain. It outlined the existing design and implementation strategies used in both fields.

The next chapter describes how the existing techniques and work in the wireless sensor network and conventional distributed operating system fields can be used to inform the design of a distributed operating system for sensor networks. This operating system will provide the distributed operating system feature set as outlined in section 2.1.2. This design is suitable for implementation on sensor network hardware, as is outlined in chapter 4.

# Chapter 4

# Design

> *'A distributed operating system is one that looks to its users like an ordinary centralized operating system but runs on multiple, independent central processing units. This is easier said than done'*
>
> – Andrew Tanenbaum

This chapter presents an overview of the high level design decisions made when architecting the Hydra distributed operating system. These design decisions are informed by the requirements of providing distributed operating system services while simultaneously providing useful functionality on a wireless sensor network.

The implementation decisions made when implementing this system design are discussed in the following chapter.

The effectiveness of the Hydra operating system as designed here is evaluated against the deployment scenarios in the Evaluation chapter of this thesis.

### 4.0.1 Distributed Operating System

As discussed in the Distributed Operating Systems Background chapter, the canonical definition of a 'distributed operating system' is an operating system

that provides a certain set of services to user applications. The core goal of this thesis is to investigate the practicality of distributed operating systems on wireless sensor networks. As such, the distributed operating system needed to be designed in such a way that it met the canonical distributed operating system requirements.

A summary of these canonical services are as follows:

- Single system image – transparently merge spatially distributed hardware so that it appears as one logical machine.
- Performance – the system must make efficient use of the hardware resources.
- Reliable – the system must handle hardware errors in some fashion.
- Resource name resolution – a way of resolving hardware devices.
- Resource management – the system needs to be capable of utilizing the resources of the whole mesh.
- Process management – a typical distributed operating system manages a set of processes.
- Flexibility – must be adaptable to a range of conditions.

The decisions behind the design of the Hydra operating system were influenced by the various strengths and weaknesses of similar design decisions made by related projects. The primary projects used to compare these decisions were the widely-deployed single-system wireless sensor network operating system Contiki[22] and the research-focused distributed operating system Amoeba[84]. General background material on these systems can be found in the appropriate Background chapters of this thesis.

In order to be an effective wireless sensor network operating system, the design of Hydra was also informed by the requirements of typical real-world wireless sensor network scenarios. Two scenarios were selected for this, based on previously published material – *event detection*, specifically a motion-detecting fence

security system[95] implemented at Freie Universitt[15] in Berlin and *livestock monitoring*, envisaged here is primarily position and behavioural tracking, similar to the requirements of ZebraNet[44], a tracking system for herds of Zebra at the Mpala Research Centre[16] in Kenya. This deployment is broadly similar to many other livestock tracking deployments[60][4][54].

These two scenarios of event detection and livestock monitoring differ in several important ways, which made them good test cases for evaluation:

- Mobility – static node placement versus a mix of static and dynamic moving nodes.
- Data – unpredictable 'alarm' events versus periodic data reporting.
- Node failure – low expected node failure rate of static nodes on a security barrier versus higher potential failure rates when attached to livestock.

The design of Hydra as presented in this chapter provides a workable wireless sensor network solution for these scenarios. Additionally the design incorporates the previous mentioned list of distributed operating system requirements. This chapter presents the design of the Hydra system in terms of the list of distributed operating system requirements. Each requirement is evaluated to ascertain how it can be designed to fit the wireless sensor network domain.

The Hydra operating system is evaluated against the deployment scenarios in the Evaluation chapter of this thesis.

## 4.1 Single system image

Typical wireless sensor network operating systems like Contiki have largely been based on the single-node model, where code executes on one node at a time, and network operations must be explicitly defined.

However, as established in the Distributed Operating System Background

chapter, distributed operating systems such as Hydra must present a collection of independent nodes as a single logical system. In a distributed operating system network communications are performed to link the individual nodes on a low level, under an abstraction layer that hides the communications.

A useful way of implementing a single system image is by pooling a set of independent computing devices together and executing native user code on the next available processor. In this architecture processes run on exactly one workstation, with the operating system managing load balancing to keep the distributed execution invisible to the user. Process migration can optionally be supported. The Amoeba distributed operating system is an example of a distributed operating system that provides a single system image service in this way. This design was evaluated for use in Hydra, but unfortunately was not suitable for the wireless sensor network domain when larger user applications were required – owing to the limited resources of a typical wireless sensor network node, it is likely that a complex application will not fit on a single sensor network node.

While it *is* possible to implement a wireless sensor network distributed operating system using this technique – where a program exists entirely on one node and simply makes remote requests to any other hardware resources it needs – it was desirable when designing Hydra to support finer-grained partitioning of user applications into smaller objects. This design meant that application code and run-time memory was not constrained to exist entirely on one node, allowing more efficient use of the resources available across the sensor mesh. The finer granularity made much more efficient power optimization possible and, at the same time, reduced the amount of program data that will be sent as the individual code objects will be smaller than the entire application. The Hydra design as presented in this chapter provides these services as part of the operating system layer.

To implement the single system image abstraction requirement, Hydra was

designed to execute user code on a distributed virtual machine, as opposed to more conventional native machine code instructions. The use of a high-level virtual machine allowed user applications to be more easily split up and distributed at run-time – bytecode instructions could be transparently redirected to perform distributed networked operations as necessary. These program fragments could also be moved between nodes dynamically at runtime. This allowed the virtual machine to treat an entire mesh of sensor nodes (or other compatible devices) as one logical machine. In this way, the single system image goal was achieved as required by the distributed operating system requirements.

Implementing these techniques on the operating system level allowed a Hydra sensor network application developer to design complex applications easily, without being aware of the details of the underlying network communications. Communication with multiple remote nodes is typically a requirement of any distributed application, especially with low-level sensor network programming techniques – under a distributed operating system design like the above, this burden is removed from the application programmer.

When implementing an application in practice, the complexity of the application code is decreased as a result of these operating system services. Pseudo-code for a fence monitoring scenario as previously discussed would be as simple as:

```
for a in accelerometers:
    if get_activity_amount(a) > THRESHOLD:
        sound_alarm()
```

Figure 4.1: Psudeo-code for a fence monitoring application

And for monitoring positions of a herd of livestock:

```
    for g in gps_devices:
        if is_outside_geofence(g):
            sound_alarm()
```

Figure 4.2: Pseudo-code for livestock geofencing

In both situations the dynamic nature of the code relocation as designed worked to help optimize for power usage, moving code fragments to the places they are most efficient.

### 4.1.1 Virtual Machines

Operating systems such as Contiki and Amoeba both concentrate on running native machine code, usually a dialect of ahead-of-time compiled C or assembly. This is the most efficient method of executing user code, but the low level of abstraction from the hardware inherent in machine code means there are limitations in how the code can be manipulated at runtime, especially in regards to dynamically partitioning and moving code across different physical hardware devices and platforms as required by the Hydra design.

As such, the Hydra system runs user applications under a virtual machine. Virtual machine bytecode is typically hardware-agnostic – the same bytecode can be portably executed under many different environments. This allows virtual machine bytecode to be moved easily between computation nodes, which is a key goal of our wireless sensor network distributed operating system.

Virtual machines can also optimize and further compile the bytecode into the machine code for the specific execution platform – a task commonly performed by Just In Time compilers. JIT-compiled code usually offers significantly better performance than interpreted code, as well as potentially allowing better performance than traditional static compilation, as many optimizations are

only feasible at run-time.

Because wireless sensor networks are a relatively new area and one that has significantly different requirements than most other areas of software development, no standard paradigm for programming wireless sensor network virtual machines has emerged. While there are a large number of virtual machine implementations, most are tightly bound to their particular esoteric bytecode format or research focus. Therefore it was necessary to either adapt a comparatively niche and unknown virtual machine standard from an existing wireless sensor network project, construct a new virtual machine technology (and associated toolchain) from scratch, or adapt an existing system design from outside the sensor network domain, implementing parts of the system on sensor networks as required.

Because the goal of this thesis is to develop a new distributed operating system – not to develop a new virtual machine technology – the latter option was selected in order to reuse as much existing technology as possible. User applications in the Hydra distributed operating system were programmed using **Java**.

#### 4.1.1.1 Adapting Java to WSNs

Java and the JVM has a reputation as being slow and memory-hungry, which does not naturally translate to being a good fit for the wireless sensor network domain. However, much of this is due to the standard library that the Java stack typically provides. The full Java standard library is not feasible to support on a sensor network – the GNU Classpath[18] implementation of this standard library includes nearly 4000 base classes, many of which would be superfluous and unnecessary on a sensor mesh. A subset of the standard library was selected for support, the design of which allowed for extension with further library support as necessary.

The use of the core Java language means that many of the programming constructs, APIs and development tools that are commonly used in desktop or server development can now be utilized on sensor networks. Java is also familiar to a large number of developers outside of the sensor network domain. In addition, because Hydra and the Hydra toolchain targeted the Java bytecode specification, rather than the Java language itself, any language that compiled to Java bytecode can be used. At the time of writing, over 60 languages[65] exist that target Java Virtual Machine bytecode as a compilation target, including Python[45], Ruby[26], PHP[66] and Javascript[67]. In real-world scenarios, implementing user code algorithms in a higher-level language is preferable, given that these systems may be implemented by users who are not primarily wireless sensor network developers.

Java has been used in the past to run application code on wireless sensor networks for both research[55] and industry[81] projects – while the focus of this thesis is not on implementing a low-power JVM specifically, many of the techniques from these other projects are applicable here too.

While bytecode interpretation will consume more energy from the wireless sensor network node on a per-instruction basis compared to native assembly or compiled C code, the flexibility provided by running higher level bytecode allows for the more advanced distributed operating system features to be performed. This can lead to a net decrease in *overall* power usage across the mesh.

### 4.1.2 Single System Image with Java

A standard Java program can be thought of as a collection of objects, where each object contains a number of bytecode methods and/or references to other objects. Hydra was designed to execute a micro Java Virtual Machine on each node as the only native code user service. Each virtual machine can contain

a number of Java Objects. When the Java bytecode for an object method requires an operation on another object, the operation is invoked by Hydra either locally or remotely, conceptually similar to a standard Java remote method invocation or Ameoeba Remote Procedure Call (RPC). This remote object invocation was designed to be transparent to the programmer.

## 4.2 Performance / Efficiency

It is generally a goal of operating systems to make efficient use of the limited hardware resources available to them. This is especially true when designing an operating system for sensor networks, as the hardware resources are more limited than is typical. Because a typical wireless sensor node runs on low-capacity batteries and replacing those batteries in the field is often inconvenient or impossible, energy efficiency is a primary concern of sensor network developers. Therefore the Hydra design as described here incorporates a number of features in order to achieve power efficiency when implemented.

Typical distributed operating systems such as Amoeba does not have power usage minimization as an explicit goal, as they are targeted at workstation or server hardware that is connected to a reliable power source. However the distributed processing / load balancing model it employs helps to make efficient use of the hardware resources available to it, indirectly reducing overall power usage.

Contiki provides a single-node threading model called 'protothreads'[24] to allow for low-power execution.

> Protothreads is a programming model ... that combines the advantages of event-driven (sometimes also called state machine) programming and threaded programming. The main advantage of the event-driven model is efficiency, both speed and memory usage. The main advantage of the threaded model is algorithm clar-

> ity. Protothreads gives you both. A protothread is an extremely lightweight thread. As with event-driven programming, there is a single stack; but like threaded programming, a function can (at least conceptually) block.[37]

This hybrid approach is adapted for use in the Hydra execution model. Hydra is designed to provide support for both event callbacks and longer-running code. When executing either pattern in a distributed fashion while maintaining a single system image model, the largest challenge is to minimize network communication. The primary energy consumer on a sensor network node is the radio – the vast majority of energy expended by a sensor node is used on network communications. Therefore reducing network communication leads directly to a reduction in overall power usage.

### 4.2.1 Minimizing network communication in Hydra

Network communication power usage is potentially one of the most serious obstacles for a wireless sensor network distributed operating system, as network communications are by necessity common in the coordination of spatially distributed components and radios are by far the most power-intensive component in a typical sensor node. To address this obstacle, the Hydra system was designed to make use of the object partitioning service provided by the Hydra virtual machine – in particular the capacity for dynamic movement of objects. As part of object migration, the total data consumed by each object to a remote node is recorded. The amount of data used by each object is periodically tallied and the object is dynamically migrated to another location in the mesh when appropriate. As a result, objects that communicated frequently tended to migrate to become 'closer' to each other. The goal of object migration then becomes to reduce the amount of network traffic, as a result decreasing the power requirements of the application as a whole.

An important requirement to ensure efficiency is to determine the optimal layout of program objects. The Hydra design provided a number of algorithms for determining the placement decisions of objects. Most of these algorithms operate on a *per-node* basis – they do not attempt to maintain knowledge about the entire mesh. This means that the memory and processing requirements for these node-local algorithms are reduced. This can be contrasted with the Amoeba approach, where complete knowledge of the load on each node is known by a single master dispatcher.

The Hydra virtual machine design defined a timeslice in which it tracks the data transmitted by each object to any remote nodes that it interacts with. This can result from many types of interactions – for example remote reads, remote writes, and remote method invocations. At the end of each timeslice, the Java Virtual Machine provided this data to the placement algorithm.

The placement algorithm then calculates the savings if the object was in a different location in the mesh. If the potential savings were over a threshold calculated from the estimated cost of moving the object, the system moves the object to appropriate node. This provides an estimate of what would have happened if the object had been moved in the last timeslice, and is used to predict its power usage in the future.

It was important that the layout algorithms be able to perform efficiently in varied scenarios. In the livestock deployment scenario, nodes are attached to animals and can move unpredictably. As such the inputs to the layout algorithm will change over time. In the intruder detection scenario, node placement is static, and thus inputs will remain more stable.

The basic operation of a generic layout algorithm is shown in Figure 4.3.

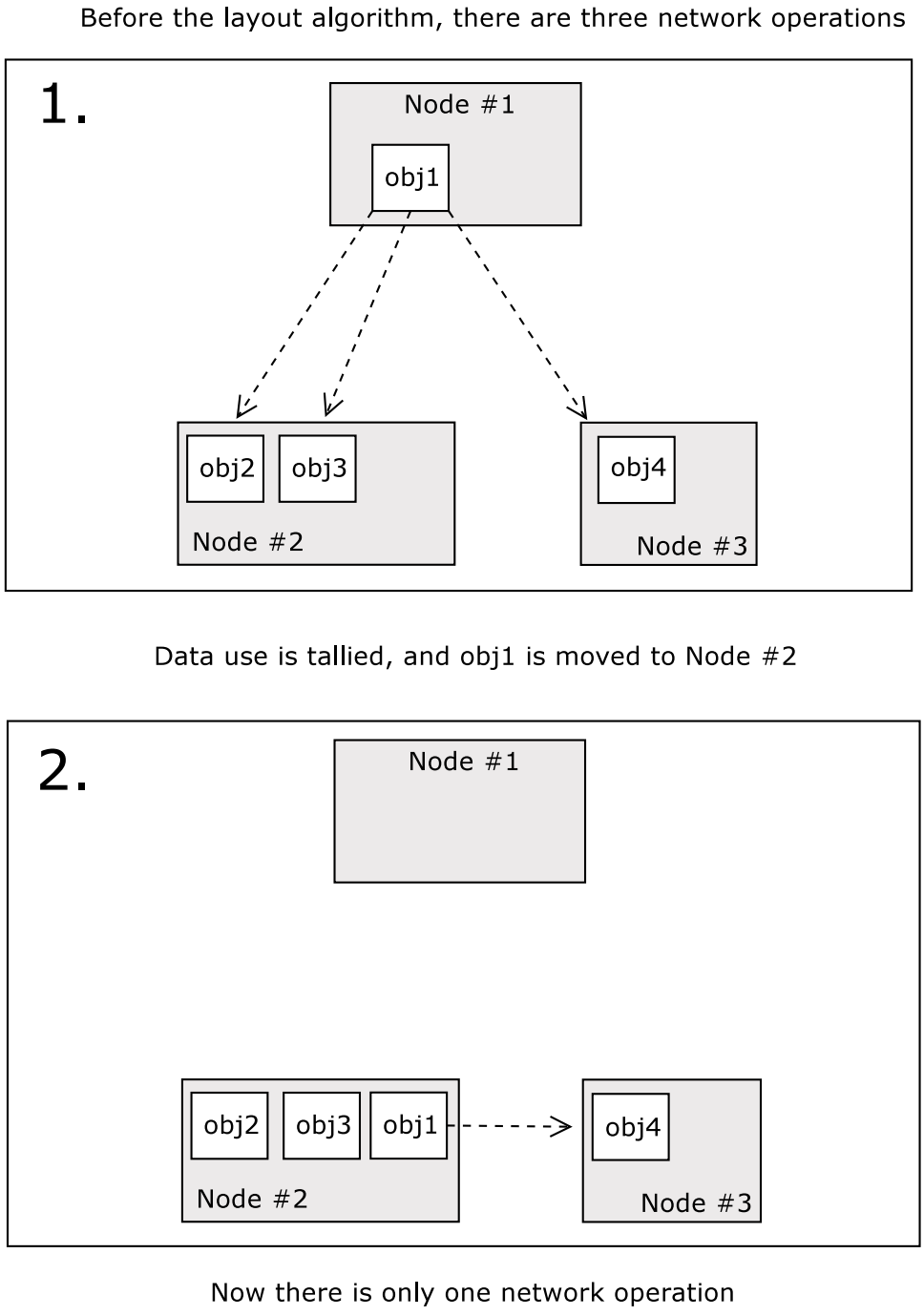


Figure 4.3: A layout algorithm in operation

Hydra was designed with an internal interface for plugging in layout algorithms, although only one is used at a time. No single algorithm is suitable for every situation, so the choice is user-configurable. A number of layout algorithms have been developed, with the most useful described below:

#### 4.2.1.1 Layout algorithm #1: 'None'

This was a null implementation that simply leaves all program objects in their original positions, never moving. This was useful for when the programmer wishes to place objects manually, overriding the automatic placement.

#### 4.2.1.2 Layout algorithm #2: 'Simple'

This monitored the communications an object has with other nodes. Each timestep, the object was moved to the node with the most bandwidth usage.

#### 4.2.1.3 Layout algorithm #3: 'Better'

This was a heuristic algorithm that attempts to find an intelligent placement, based on the individual nodes knowledge of the mesh. It attempted to approximate knowledge of the mesh by monitoring interactions with other nodes, and the status metadata attached to some network messages.

The inputs to the heuristic were:

- The number of objects on a node, either explicit or estimated through object movement updates
- Node RAM remaining
- Node power

Not all of these inputs were necessary at all times, and could be disabled or weighted differently with configuration settings as the scenario dictates.

#### 4.2.1.4 Layout algorithm #4: 'Perfect'

This algorithm required full knowledge of the mesh layout. This is available under the simulation environment, or by doing periodic full-mesh updates of status whenever a change occurs. This causes considerable traffic. However, it produces the most optimal placement (based on the previous timestep), and is useful for benchmarking and evaluation purposes, as well as where power usage is less of a concern than traffic efficiency.

## 4.3 Reliability

All operating systems have reliability as one of their key goals. However the approaches taken towards ensuring reliability differ between operating systems.

Single node operating systems like Contiki concentrate on restoring and managing software errors, as well as explicit notifications for gradual hardware failure like power loss. The loss of one node will typically result in the loss of all computation being performed on that node, unless the user application was specifically built to avoid such failures.

Distributed operating systems like Ameoba or Hydra are more susceptible to general reliability problems than conventional single-system operating systems, as the more complex mesh of connected hardware is statistically more prone to failure in individual components. Errors in a hardware device that is connected to the distributed operating system must be handled in a way that limits the impact on the system as a whole. Ideally the user applications would not be aware of the failure. While Ameoba itself is not fault-tolerant[85], some extensions to it have provided fault-tolerance services[20].

When designing Hydra, it must be recognized that the potential for hardware failure is exacerbated in wireless sensor networks as the hardware is relatively

failure-prone compared to conventional server or desktop systems, thanks to less robust hardware and limited energy reserves.

The design of a fault-tolerant system is informed by the types of hardware failure and their consequences. There are two main classes of hardware failure in sensor networks.

- Immediate – the hardware suffers an unexpected and sudden failure, such as being crushed by livestock or an intruder. In this case there is no way to anticipate the failure.
- Gradual – the hardware deteriorates over time, usually because of depleting energy reserves or cumulative hardware wear. At a certain point it can be determined that the node should transition into a failure state and disable itself gracefully.

In many sensor network deployments (where nodes simply pass raw sensor data back to a central collection point), the loss of a number of individual nodes may not be of a large concern. As long as sufficient sensor nodes continue to operate, the sensor mesh as a whole will continue to operate, although possibly with decreased efficiency and possibly network re-routing overhead as the mesh adapts to the damage.

However the distributed operating system techniques employed by Hydra increased the importance of individual nodes. Because objects were stored on individual nodes, the loss of a node meant the loss of those objects. In almost all cases, the loss of an object means the application failed to operate as expected. If the object that is lost was a currently executing object, then without robust reliability mechanisms, the application will be effectively terminated.

As such it was necessary to design the sensor network operating system such that applications can be protected from hardware failure. The reliability system must – without requiring explicit support on the part of the user application developer – provide these services:

- Provide protection from gradual failure
- Provide protection from immediate failure
- Be transparent – where possible the user application should not be aware of the failure of the underlying hardware.
- Be robust – in the event of hardware failure, the system as a whole must continue executing the user application without changes
- Be efficient – require as small amount of additional resources as possible

One possible solution for this issue would be to use hardware *redundancy* – that is, to run multiple sensors at once to perform the task. Supporting this would be a useful addition to Hydra. However the shared/single-system execution model makes this complicated to support both transparently and efficiently without explicit user-level multi-processing support.

Hydra implements a *checkpointing* system to accomplish the desired reliability goals.

### 4.3.1 Checkpointing

Checkpointing is a widely deployed reliability technique that involves taking periodic snapshots (or 'checkpoints') of the current system state. When a fault is detected, the objects from the anomalous node are restored to the state that they were in when the last checkpoint was taken. Checkpoints are stored in such a way that they are not damaged by the loss of a node.

This approach has several advantages:

- Conceptual simplicity – because Hydra controlled the entire virtual machine, it is relatively straight forward to extract the entire state. Checkpoints are done per-node, not per-object, allowing a compact representation of an entire nodes state in a single checkpoint.
- Efficiency – when compared to other reliability techniques such as 'hot

standby', checkpointing can be more efficient in terms of overall CPU and power usage. Only one node executes a thread at a time, other nodes simply manage checkpoint states.

However, there are several inherent challenges to checkpointing, especially in a wireless sensor network environment:

- Storage requirements – because wireless sensor network nodes have such limited storage, the additional operational overhead of maintaining checkpoint data is problematic
- Checkpoint location – most of the failures of wireless sensor network nodes are due to hardware failure on the individual node. This means that a node cannot store its own checkpoints, as the checkpoints would likely be lost or otherwise become inaccessible due to the hardware failure.
- Lost operations – if an object performs operations and then fails before those operations have been checkpointed, its actions since the last checkpoint will be lost.

A viable wireless sensor network distributed operating system needs to implement a checkpointing system that correctly handles these challenges.

**Storage requirements** as well as the transmission overhead are reduced by only checkpointing the *differences* between the current state and the state at the last checkpoint. This is accomplished with minimal overhead – objects have an internal flag that indicates if they have changed. On checkpoint, only the 'changed' objects are transmitted and the flag is reset for all objects. When the checkpoint update is received, the diff is applied to the saved state.

**Checkpoint location** is difficult, as the optimal checkpoint placement depends on the nature of the application and the likely failure states. In some applications, failure of a node is often accompanied by the failure of nodes that are physically nearby. For example, an application that monitors road

traffic using a number of tightly clustered sensor nodes could lose an entire geographic cluster in the event of a traffic accident. In this case the correct strategy is to place checkpoints as far from the original node as possible. However, in applications where this is not an issue, it is advantageous to locate checkpoints as *close* to the original node as possible to minimize the number of hops required to transfer the data and to ease restoration. The strategy used by Hydra is configurable on a per-application basis, allowing the user to pick whichever they feel most appropriate.

**Lost operations** are encountered when an object performs operations – executing code, remote function calls, etc – but then fails before the results of these operations can be checkpointed. The system needs to be capable of handling this correctly, without loss of data.

In order to prevent this, operations are cached until they are successfully checkpointed. In this way, any node can be lost, and the system can recover. This occurs without impacting the user application.

In real-world usage scenarios, hardware failure will likely still require replacement hardware to be deployed – for example, an animal with a defective tracker node will fail to be tracked, and a fence section with a defective monitoring node will fail to raise the alarm. However the rest of the system should go on unchanged in either scenario.

## 4.4 Resource name resolution

Naming is a common problem in distributed operating system design. Resources are spread across multiple different nodes and a naming scheme is used to allow user applications to refer to the resources that they need. Hydra uses a very simple naming system to remove the overhead of potentially slow name lookups. This naming scheme was implemented specifically for Hy-

dra based on the constraints of the project and the range of likely hardware availability for testing.

- Node IDs – sensor nodes are assigned a unique numeric identifier, conceptually similar to the MAC address on conventional network interface cards. Node IDs are assigned statically and sequentially. This numbering system allows for simple identification and discovery of nodes.
- Sensor IDs – individual sensor objects on nodes are assigned numeric sensor IDs, that correspond to the type of sensor that is being accessed. At runtime, constant strings can be used.
- Object IDs – individual virtual machine objects have a unique numerical identifier, although this is not visible to user applications. It is used internally in the virtual machine.

The combination of these two systems means that any individual sensor device can be uniquely identified and discovered with two numbers – the node ID and sensor type. It is to be noted that this approach limits the system from fully utilizing sensor nodes with two of the same type of sensor attached. It would be a relatively minor change to extend the naming system to allow this.

It should be noted that while this approach is not sufficiently secure to be deployed in a potentially malicious environment, it is sufficient for the current research purposes.

## 4.5 Resource Management

Distributed operating systems need to make effective use of the limited hardware resources. This is especially true in the sensor network domain, where resources are so limited. An important aspect of this is to *load share* resources, making the best use of the cumulative resources available across the mesh.

Ameoba accomplishes this by use of the 'process descriptor', which defines the hardware environment that a process requires.

> The key element of Amoeba's process management mechanisms is the process descriptor. It is a portable structure that [in part] ... describes the properties of the host on which the process may run (e.g., CPU architecture, memory availability, etc.). A process can only run on a host that has the properties matching those in its process descriptor. [20]

Using these requirements as well as existing knowledge about the load on each node, the Amoeba kernel will identify nodes that fit the execution criteria for a particular process and execute the system on that hardware node.

While this technique is simple and works efficiently to distribute whole-process machine code across potentially different platforms, Hydra has a different set of challenges. Because of the virtual machine, we can safely assume that all nodes can at least execute the code. However they may not be able to execute the entire program at once – there may not be sufficient memory or energy resources on the node to accomplish this. Therefore it is useful to have relatively sophisticated techniques to manage resources, which can allocate resources on the Object level, rather than the entire process.

Hydra uses two primary techniques to manage resources.

### 4.5.1 Intelligent object location

When a new object is created, the selection of the node to create it on is important. It can be assumed that the new object will be utilized in some way by the creator, so to avoid network overhead, ideally the object will be placed on the node the creation request originates from. However, this may not always be possible – for example, the object may not have sufficient RAM. If this is the case, a number of attributes are considered to find a placement

for the new object. Nodes are ranked on these attributes, which include (in order of importance)

- Power remaining (excluding nodes with a low-power warning)
- Number of network hops from the originating node
- Free memory
- CPU capacity

Once a node is selected, a remote creation request is sent and the object ID is returned.

### 4.5.2 Paging

Paging is a technique that is widely used on conventional desktop and server operating systems. It refers to the ability of these systems to dynamically and transparently use hard disk storage instead of RAM when more working memory is required than the amount of physical RAM in the machine. Memory that has not been accessed recently can be 'paged' to disk when free physical memory becomes low. This technique is accomplished by using the Memory Management Unit (MMU) present in the CPU of these machines to intercept reads and write operations. These operations are then performed on hard disk storage instead of RAM.

While hard disk operations are much slower than RAM operations, this allows operating systems to continue to operate even when physical memory is exhausted.

Hydra has a similar capability. While there is no MMU on a conventional sensor node, because Hydra operates as a virtual machine, the same type of operation is possible. Hydra can page objects in and out of mass storage as necessary, allowing many more objects to be created at runtime than conventional RAM limitations allow. This allows Hydra applications to operate

without concern of overloading memory limitations on individual nodes – allowing the creation of more complex systems.

## 4.6 Process Management

Hydra was designed to support a single-process model – only one user application can be executing simultaneously.

Traditional operating systems provide support for multiple user processes running simulataneously. However, a single user process is usually sufficient for sensor networks, which tend to concentrate on one task. It is typically unlikely on sensor networks that multiple indpendent applications are required – if multiple tasks are needed, they can be modules of the same application.

However it should be noted that the Contiki sensor network operating system allows multiple processes, so it may be useful to extend Hydra to provide similar functionality in future. The fundemental design of the Hydra system does not preclude this from taking place – it would just require some more complicated user-side library support.

## 4.7 Flexibility

An important goal of any operating system is flexibility. The system must be able to perform a number of varied tasks on a variety of hardware configurations. Hydra accomplishes programming flexibility through its use of a general purpose bytecode language (compiled from the general purpose Java language). Another aspect of flexibility is that of hardware configuration flexibility. In the sensor network domain, it is often necessary to add and remove sensor hardware at runtime. Hydra supports dynamic addition and removal of sensor nodes – additional nodes can be added simply by powering them on and

letting them broadcast an initial message to the mesh. Nodes can be removed simply by switching them off and letting the reliability system perform any necessary cleanup.

This flexibility of platform independence allows for hybrid approaches to sensor network hardware when deploying Hydra–for example, a sensor network that required mass storage as part of its application could associate with a Linux-based Java Virtual Machine that allowed use of its hard drive. The combination of the object migration system and the virtual machine abstraction layer allows a hybrid approach to sensor node hardware. Hydra virtual machines running on platforms with more resources accept more objects. In a deployment, this can be used to allow a set of simple embedded sensors to be supplemented with a small number of more powerful Linux-based base stations. In another scenario, a variety of hardware models of sensor nodes can belong to the same mesh. As long as they have some method of communication, the hardware abstraction given by the Java Virtual Machine means that they can share data and code seamlessly.

## 4.8 Design finalization

This chapter outlines the design of the Hydra wireless sensor network distributed operating system. The following chapter discusses the next stage in the process – implementing the design.

# Chapter 5

# Implementation

> *'The goal of Computer Science is to build something that will last at least until we've finished building it.'*
>
> – Anonymous

The previous chapter outlined the design decisions and requirements of the wireless sensor network distributed operating system. This chapter describes the implementation of that design.

The core goal of this thesis is to investigate the practicality of distributed operating systems on wireless sensor networks. As such, it is useful to discuss the challenges and techniques used to implement the design outlined in the previous chapter, as the distributed operating system needed to be implemented in such a way that it met the design specifications and yet was also compact enough to fit in the space available on a typical wireless sensor node.

The primary hardware platform targeted was the Scatterweb wireless sensor network node[73].

Figure 5.1: A MSB430 Scatterweb sensor node

## 5.1 Scatterweb

The Scatterweb sensor network platform was developed at the University of Berlin, and consists of a modular wireless sensor network board and a software middleware package designed for the hardware. Hydra makes use of the Scatterweb hardware platform, with some of the underlying driver software adapted from the Scatterweb middleware. Specifically we use the 'Modular Sensor Board 430' node, which was developed in 2005[73].

The Scatterweb platform was targeted for several reasons:

- It has been used as the basis for a number of research projects, both hardware and software-based[61][28][9][71]
- It is extensible, consisting of a number of boards that can be customized as necessary, so it is flexible for multiple deployment scenarios
- Out of the box, it consists of a number of embedded useful sensor devices (see below for details)
- The software and compiler build-chain is well documented and supported.
- It is otherwise a 'typical' sensor node platform, with constrained CPU and RAM and a conventional radio with power supply.

The specifications of the MSB430 Scatterweb sensor nodes are:

- A `MSP430F1612` CPU, operating at 8mhz
- 5 kilobytes of RAM
- 55 kilobytes of flashable ROM
- A `CC1020` radio, operating on 402-470 MHz / 804-940 MHz, at a maximum data rate of 153.6 kBaud
- A MicroSD card reader/writer.
- Various sensors – humidity, acceleration, light.
- Powered by 3 x AAA batteries.

As of this writing, the most popular sensor node hardware platform is the `TELOSB/TMote Sky` node, developed at Berkeley. The TMote platform uses a `MSP430F1611` CPU, with 10 kilobytes of RAM, 48 kilobytes of ROM and provides 1024 kilobytes of external flash that can be used for mass storage. Adding support for the TMote platform or other similar hardware platforms should be straight forward, given the similarity in CPU architecture.

Hydra will make use of the increased RAM and external storage if available, but does not require it. As such, the Hydra design is applicable to most wireless sensor network nodes, irrespective of their capacity.

## 5.2 Java

Hydra applications are developed using the Java language[78]. One of the benefits of Java is that there are many implementations of the virtual machine runtime that have been developed and tested thoroughly over many years. Popular implementations include:

- Sun Java[43] – the official implementation of the Java specification. It includes a toolchain, compiler, and runtime libraries for several operating

system platforms. The bytecode language the compiler produces is an open standard.

- Dalvik[57] – A Java variant developed by Google for the Android mobile phone operating system project, Java language applications are compiled to a Dalvik-specific bytecode language.
- MikaVM / TinyVM[38] – Small Java Virtual Machines intended to run on embedded devices.
- JCVM[19] – Compiles Java code to C, so code can be run without the overhead of an interpreter.

While these implementations are deployed and work well in many situations, unfortunately it became obvious that they were not suited for Hydra – the very specific requirements for distributed operating services as outlined in the Design chapter would require considerable modifications to the Java Virtual Machine, with deep integration required with the rest of the system. As such, Hydra implements a custom micro Java virtual machine called `HydraVM` that both interprets Java bytecode and provides the distributed systems services required for the distributed operating system to function.

## 5.3 Toolchain

While it was not possible to use the existing virtual machine technology directly, the Java software stack is much larger than the core virtual machine – and it was desirable to utilize as much of this existing software stack as possible to reduce the amount of software that needed to be built. The standard (and most widely deployed) Java compiler suite is the Sun Java environment, and Hydra was implemented to leverage this compiler toolchain as part of the build process.

The Sun `javac` compiler is used directly to take `.java` source files and output

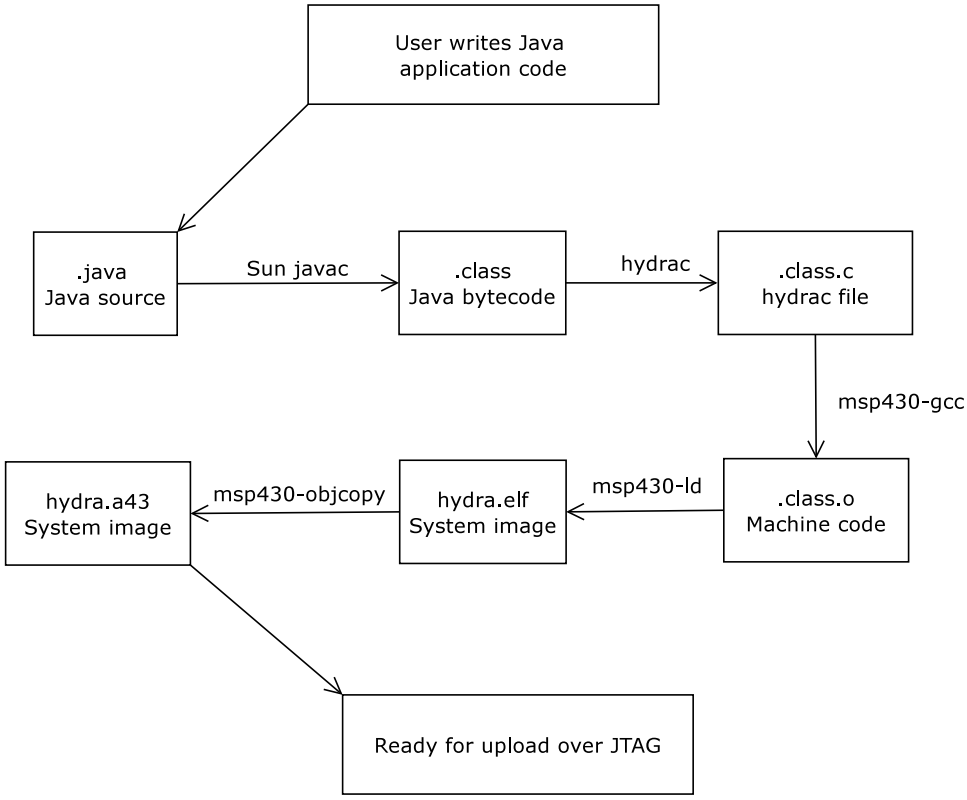


Figure 5.2: An overview of the Hydra application compilation process

compiled `.class` bytecode. However in order to further optimize the output for the Hydra platform, `hydrac` then performs several further compilation steps using a custom toolchain. An overview of the steps involved in the toolchain are outlined in Figure 5.6 – the toolchain is complex, but it means that as much as possible is done at compile time, rather than runtime.

While the purpose of this thesis is not directly to create the *most optimal* virtual machine – or virtual machine toolchain – it is nevertheless desirable to minimize resource usage where necessary to ensure the efficient and useful operation of the distributed operating system.

Explanation of each step in the toolchain follows.

### 5.3.1 Creation of `.java`

The user writes Java code in their text editor of choice, against a subset of the standard Java APIs and some Hydra-specific APIs as necessary.

### 5.3.2 `.java` to `.class`

The standard Sun Java compiler `javac` is used at this stage. It takes a `.java` source file as input, and produces a standard Java `.class` file.

### 5.3.3 `.class` to `.class.c`

The class file produced in the previous step is not optimized for size, however – a simple 'Hello world' program compiles to 426 bytes of bytecode. While this output file size is still small, the number of instructions per byte is relatively small compared to other systems.

The primary uses of space in the Java bytecode size are:

- The symbol table / constant pool. Java programs make heavy use of the 'constant pool', a table which contains literal values that may be used in the program.
- String symbols. These are strings — stored in the constant pool – such as function names that are used at runtime for lookups. For example, when a method is invoked, the method signature used is a string that is retrieved from the constant pool.
- Bytecode is stored in a format primarily intended for fast execution rather than compact storage.

It is also important to note that the `.class` file requires parsing to build up the constant pool, function table, fields, and so on. This parsing requires a

non-trivial amount of CPU and RAM – and thus power – on a limited sensor node platform.

In order to address these issues, the Hydra-specific build tools are used. `Hydrac` is the first step in this process. It is a code generator that takes a `.class` file as input, parses it, and produces a `.c` file as output. The `.c` file can be compiled with standard GCC tools and linked into the system image at compile time.

```
const byte test_method_1_bytecode [] = { //main BYTECODE (7 bytes)
    3, 184, 0, 2, 167, 255, 252
};

const vm_method_t test_method_1 = { //main
    1, "main", 1, &test_method_1_bytecode, 1, 1, 1
};

const vm_method_t *test_vm_methods [] = { //METHODS
    &test_method_0, &test_method_1
};

const vm_class_t class_test = { //FINAL CLASS OBJECT
    0, "Test",
    (vm_cpentry_t **)test_vm_constantpool,
    (vm_method_t **)test_vm_methods,
    NULL, 21, 0, 2
};
```

Figure 5.3: A portion of the output from `hydrac`

This `.c` file contains a pre-processed set of initialized C structs that provide the function definitions, as well as the bytecode for each method. An example is shown in figure 5.3

There are two main benefits to this strategy. Firstly, the pre-processing and parsing of the `.class` file is moved to occur at compilation time, not runtime. This means that the – rather complex – parsing logic is not required to be loaded on to the sensor nodes, decreasing the code footprint. Secondly, the runtime memory usage is reduced - the `MSP430` hardware platform provides much more addressable ROM than RAM, and the `const` declaration in the generated code causes the code to be located in ROM, leading to savings in RAM.

### 5.3.4 `.class.o` to machine code

The next step is to compile the `.c` file. In this case `msp430-gcc` is used to produce the object file. This is a fork of GCC for the `msp430` CPU, which is used by the Scatterweb hardware platform.

#### 5.3.4.1 What about multiple architectures?

Note that currently this implementation restricts migration to only those platforms that can execute MSP430-style machine code. However as MSP430 emulators exist for a number of platforms[27], a VM implementation for x86 or similar could use an emulator as the base code for executing the foreign machine instructions on faster hardware.

Alternately a system image for alternate platforms (as per the following subsection) can be compiled ahead of time that includes all necessary machine code for multiple platforms – ie: a msp430-platform system image can be compiled for msp430 sensor nodes, an x86 system image can be compiled for x86 nodes. The compiled code can then exist on each node until it is activated by the higher-level instructions.

### 5.3.5 Produce system image

The object file is then linked in with the Hydra `libvm` and basic operating system layer to produce a `.elf` system image. The `.elf` is then processed with `msp430-objcopy` to create the final system image file. JTAG is used to flash the image on to the device.

### 5.3.6 Runtime

At runtime the virtual machine uses the structures created in the `hydrac` output file directly, locates the compiled bytecode, and starts execution. More information about the virtual machine operation is in the next section.

## 5.4 HydraVM

The Hydra virtual machine is the core of the distributed operating system runtime. It is an interpreter of the Java bytecode language. Although interpreters are slower than more advanced Java Virtual Machine techniques such as Just-In-Time compilation, the limited resources of the sensor node hardware precludes these.

The Hydra virtual machine can be run on wireless sensor network hardware as well as on desktop or server Linux systems, communicating with the sensor node virtual machines over a network link. We have tested Hydra on 16-bit `MSP430`-based Scatterweb nodes, and 32 or 64-bit Debian Linux.

Java standard library methods are implemented in native C code. Because most Java applications will make heavy use of the standard library, the majority of the processing time will be spent in the relatively fast C. Typically a sensor network application will also spend much of its time waiting on I/O events from sensors.

The Hydra virtual machine runtime provides a subset of the standard Java libraries.

### 5.4.1 Hydra virtual machine distributed execution

A collection of Hydra virtual machines cooperate to run a single application. A thread can execute on only one virtual machine at a time – in other words, there is only one program counter per thread, and it can move from node to node.

The virtual machine can transition between one of several states:

- Active – executing code, or waiting for I/O that will then return to

executing code

- Inactive – not executing code, no objects.
- Waiting – waiting for remote operations.

Objects are 'owned' by one (and only one) node. Initially they are owned by the creator, but ownership changes when objects are moved. Nodes maintain a local knowledge of where objects are, based on their view of the network. If this knowledge becomes out of date and a node addresses a message to an node that no longer owns an object, that node will know where the object has gone and will transmit a routing update to the sender accordingly.

A number of remote operations are supported:

- **Object move**: an object is being relocated from one node to another. The state of the object is transmitted, along with some additional execution state data if the current object contained the program counter.
- **Remote invoke**: a method on an object can be invoked remotely. The method is executed and a remote object reference is returned to the caller.
- **Remote read**: read the state of a primitive (such as an integer)
- **Remote set**: set the state of a primitive
- **Status update**: request or return the state of the node in terms of metrics such as (estimated) power usage, CPU use, memory use
- **Checkpoint update** transmit the checkpoint state or checkpoint delta of an object. The checkpoint implementation is discussed in the next section.

### 5.4.2 Checkpointing

An important goal when implementing the checkpointing system was that it perform its operations efficiently in terms of memory and CPU usage. This

applied not only to the serialization and de-serialization of checkpoints, but also to the storage of checkpoints at runtime. The more compact the storage of checkpoints at runtime is, the more checkpoints can be saved. Likewise, efficient serialization and de-serialization allows for efficient and quick recovery from errors.

However in some ways, these two goals are mutually exclusive. Increasing the complexity of the compression may decrease the storage requirements for individual checkpoints, but it decreases the ease of saving and restoring, increasing CPU and time requirements on CPU-poor devices.

A trade-off therefore had to be made between the two factors. It was decided that it was most desirable to store the checkpoints in the most compressed form possible - smaller checkpoints means smaller data transmissions over the air.

#### 5.4.2.1 The checkpoint object

Checkpoint objects contain a complete snapshot of the state of the node at the time of the checkpoint. In this way, a single checkpoint object can be used to restore a node. The structure of a checkpoint object is stored as a packed C memdump, the structure of which is displayed in figure 5.4,

Each individual node is in charge of deciding when it will create a checkpoint and where it will send it to. Checkpoint objects are created on the sending node, and transmitted in the compressed form. The receiving node receives the checkpoint object and stores it directly in the compressed state after stripping out the ‘deleted objects’ section (if any).

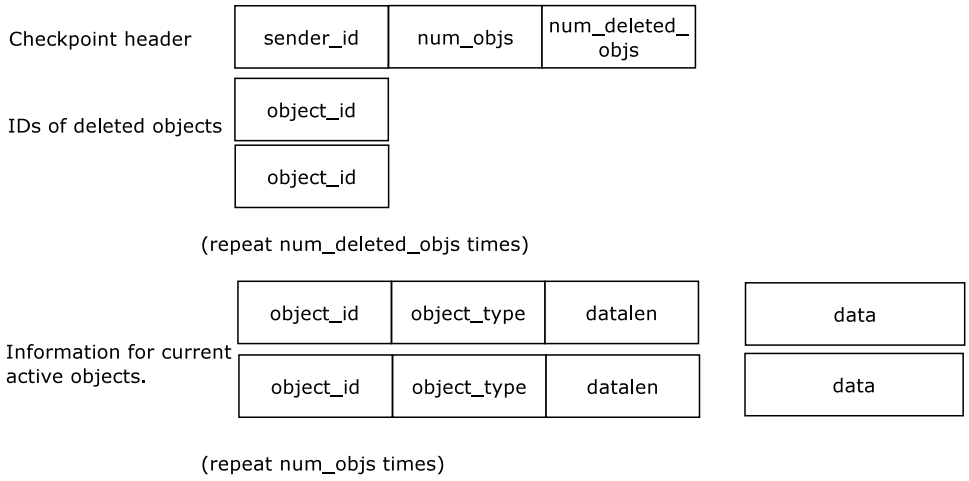


Figure 5.4: The structure of a checkpoint

#### 5.4.2.2 Checkpoint diffs

Because the entire state of an object can be relatively large, the checkpointing system supports the concept of checkpoint 'diffs'. Objects have a single bit internal flag that marks them as 'dirty' to the checkpointing system. When objects are initially created they are 'dirty'. In addition, whenever their internal data state or code pointer changes, they are also marked as 'dirty'. The checkpointing system traverses the local object graph and packs only the 'dirty' objects into the checkpoint update. After the checkpoint is complete, the 'dirty' state is reset on all objects.

In this way, no separate code path is required for diffs vs full checkpoints - the dirty/clean system ensures that the correct objects are transmitted each time.

When a checkpoint is received by a node that has already received a checkpoint from the same sender, the receiver compares the old and new checkpoint objects and merges the update into the old, updating data where necessary. While this takes a certain amount of CPU time, the benefits of smaller checkpoint updates transmitted over the network outweigh the cost.

### 5.4.3 Operation caching

As outlined in the previous chapter, it is necessary to cache messages that are transmitted between checkpoint updates in order to restore a state that is synchronized with the rest of the mesh.

In order to reduce code complexity, these messages are intercepted at the network layer and cached until they are obsoleted by the arrival of a new checkpoint. Because the caching is done at a low layer, it is not necessary for the upper virtual machine layer to cache (potentially expensive) per-message metadata.

The messages are cached as part of the checkpoint and applied sequentially when a checkpoint is restored in order to bring the checkpoint up to date.

### 5.4.4 Paging

The paging system is implemented by monitoring the usage patterns of objects in a 'least recently used' list – objects that have been unused longest are paged out, removed from RAM and stored in the paging area. When an object is accessed by the virtual machine, if it is paged out it is restored from the paging area and moved to the top of the LRU list.

The paging area backend storage implementation is either a separate memory block on Linux-based virtual machines, or to a predefined (empty) area of the flash memory on Scatterweb devices.

## 5.5 HydraSim

HydraSim is a simulation framework that was developed to test layout algorithms for Hydra.

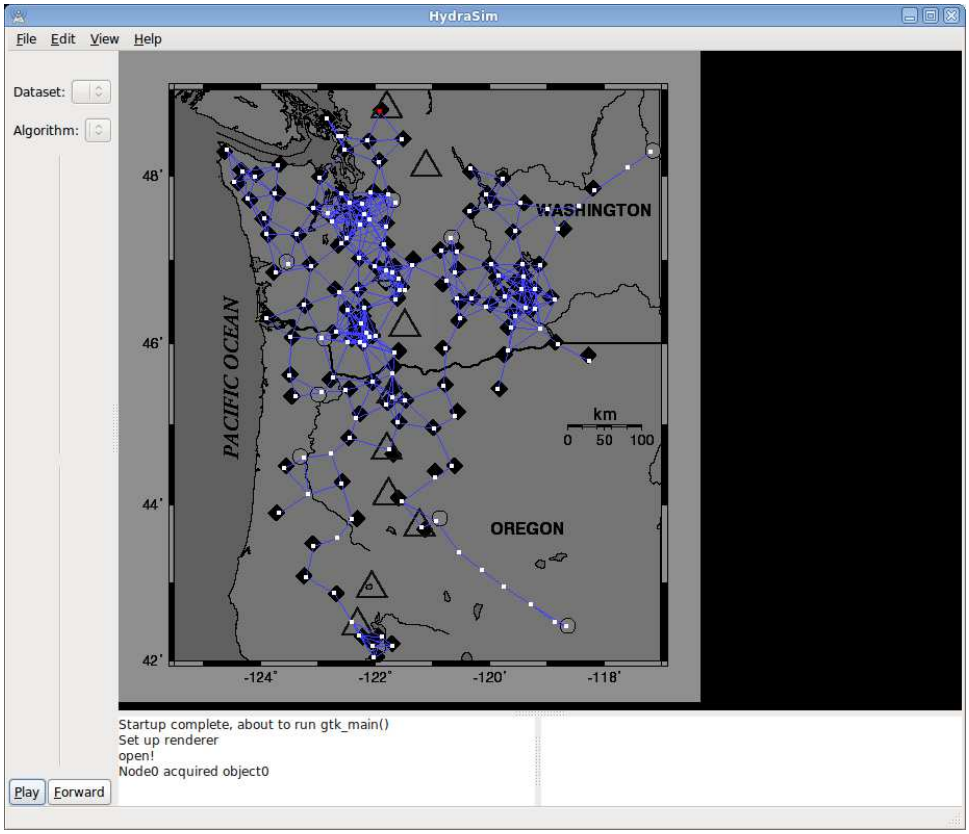


Figure 5.5: The HydraSim tool, running a simulation on an earthquake dataset with a large number of nodes

It is a GTK2 application that implements the high-level migration logic for migration algorithms, and allows automated repeatable testing against different event test datasets. It does not execute a virtual machine process for simulated nodes, it only models the input and output events. Migration algorithms are written in C, using an API that matches that of the virtual machine.

This allows for much faster development and debugging of migration algorithms compared to testing on physical hardware or on full virtual machines. Once the testing and development in HydraSim is complete, the migration algorithm code can then be directly used in the full Hydra virtual machine.

## 5.6 Hydra Test Framework

The Hydra Test Framework is a separate system from HydraSim. Unlike HydraSim, the test framework provides simple mechanisms for large-scale automated testing of the Hydra virtual machine.

The test framework is implemented as a Python application that spawns a large number of Hydra `nativevm` processes under Linux. These processes are put into a specialized 'test mode' that causes them to output changes in state to a network socket. The test framework receives these messages and processes them to determine the health of the mesh and whether an error has occurred. In this way the test framework and virtual machine processes do not have to exist on the same physical machine, allowing for larger number of test nodes.

This framework can perform automated tests on a number of the Hydra subsystems:

- Remote access – remote `get`/`set`/`invoke` operations are monitored for correctness and to make sure that all involved nodes are successful.
- Migration – it can monitor the location of objects and ensure that they

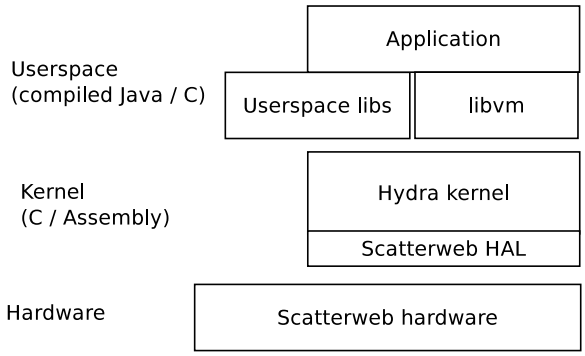


Figure 5.6: The structure of the Hydra OS

are migrated correctly from object to object.

- Checkpointing – the checkpoint and object state can be compared on both sides of the operation to ensure there has been no error in transmission or serialization/de-serialization.

It also allows testing of loss in the mesh. The framework can optionally terminate virtual machine processes with or without warning. This tests the predictable and unpredictable shutdown logic, along with the reliability system. A key test of the reliability and checkpointing system is that it can correctly deal with any particular node(s) in the mesh being terminated unexpectedly.

## 5.7 Hydra structure

The Hydra virtual machine is implemented as a static library - `libvm`. This library provides an interface for external C programs to invoke the virtual machine. It is linked into either the Hydra operating system layer for running on sensor node hardware.

### 5.7.1 Hydra operating system layer

The Hydra operating system layer is based on an earlier wireless sensor network operating system project[41], and provides a number of common operating

system features that abstract away the underlying hardware. These include:

- IP, ICMP and UDP networking layers, SLIP layer2 protocol.
- Task management, threading via a (partial) implementation of the PThreads API
- Heap memory management (`malloc()` and `free()`)

A major component of the Hydra operating system layer is the hardware-specific driver layer that implements a common driver interface for various classes of devices. These typically include:

- Radio interface
- Serial port (for raw text or SLIP)
- Mass storage (such as the MicroSD cards on Scatterweb)
- Sensors such as the accelerometer
- Misc devices such as the Piezo and LEDs

To port to a new hardware platform it is only necessary to implement this section of the operating system layer. Currently the only implementation of the driver layer is for Scatterweb hardware.

The final output produces a binary image that can be uploaded over JTAG.

### 5.7.2 Platform independence

Hydra is platform and architecture independent (except for the hardware abstraction layer component of the kernel). Because the virtual machine code is implemented as a library – `libvm` – it becomes possible to create `nativevm`, a native Linux application that runs the virtual machine library against the Linux APIs for networking and process management. `nativevm` is created as part of the standard build process.

This means that the core Hydra technology can run on 16, 32 and 64 bit plat-

forms. Platform independence allows for hybrid approaches to sensor network hardware – for example, a sensor network that requires mass storage as part of its application can contain a `nativevm` process that allows use of the local hard drive. This allows a set of simple embedded sensors to be seamlessly and dynamically supplemented with a small number of more powerful Linux-based base stations.

A variety of different hardware models of sensor nodes can belong to the same mesh. As long as they have some method of communication, the hardware abstraction given by the Java Virtual Machine and the platform-independent network transmission means that they can share data and code seamlessly.

## 5.8 Implementation Summary

This chapter outlined the implementation of the Hydra design. The implementation takes the form of a multistage compilation pipeline '`hydrac`', a hardware-agnostic distributed virtual machine '`hydravm`' (in the form of a `libvm` static library) and a low-level runtime hardware driver abstraction layer. These components are combined to implement the entire Hydra operating system stack.

In the following chapter the deployment to evaluate the Hydra implementation is outlined.

# Chapter 6

# Deployment

> *'Software and cathedrals are much the same – first we build them, then we pray.'*
>
> – Sam Redwine

A useful method of demonstrating the real-world effectiveness of the prototype wireless sensor network distributed operating system is to use it to perform a typical wireless sensor network task.

The goal of this thesis is to discover whether the distributed operating system techniques can be applied in a sensor network context, and if so, whether they provide valuable improvements. As such, the following statements need to be demonstrated to prove this applicability:

- That the distributed operating system can perform a basic, typical sensor network task.
- That the distributed operating system can be practically extended to perform more complex tasks, such that the improvements and advantages over more conventional systems can be demonstrated.

This chapter discusses the process of deploying Hydra to prove these statements.

The first step is to select a sensor network use-case.

## 6.1 Selecting a use-case

Sensor networks are widely deployed against a highly varied range of applications. As such, there are a large number of possible use-cases we could implement as a prototype. The criteria for use-case selection were as follows:

- The use-case is well-known and, if possible, outlined in published research. This means that the use-case is more likely to be widely accepted as an effective test of a sensor network system.
- Sufficient information on the existing implementation(s) is available that a Hydra implementation is possible. Ideally this would take the form of source code, though a sufficiently detailed description of the systems involved would suffice. This is necessary to allow for a proper comparison of the two systems.
- The task is suitable to run on the Scatterweb hardware – it had been implemented in the past on the same class of hardware device. This means that the two systems will be compared on the merits of their software only.
- The task is relatively simple, but extensible – the use-case should allow a simple implementation that can be extended as necessary. This allows the testing to prove the additional 'valuable improvements' requirement.
- The task appears to be one that would generate events dynamically, rather than simple periodic data collection. This means that the application would benefit from the distributed paradigm and allow us to highlight the advanced in-mesh data processing capabilities the distributed operating gives sensor network applications.

A literature search was undertaken with these criteria in mind, with some

examples named in the Background chapter.

## 6.2 Fence Monitoring

The use-case selected for detailed evaluation was an application first outlined in *'Fence Monitoring – Experimental Evaluation of a Use Case for Wireless Sensor Networks'*[95], published at the European Conference on Wireless Sensor Networks (EWSN) in 2007.

This use-case is described as follows:

> *"The Fence Monitoring project is a use case for Wireless Sensor Networks (WSNs) focused on collaborative, in-network data processing. The goal is to develop a distributed event detection algorithm that can reliably report security relevant incidents (e.g. a person climbing over a fence) to a base station. The vision is that through cooperation of many sensor nodes the accuracy of event detection can be greatly improved, while at the same time saving energy by reducing multi-hop communication with the base station."*[95]

This use-case meets the necessary criteria outlined at the beginning of this chapter:

- The project was published in major conferences – eWSN and SenSys.
- The algorithms and system structure used is outlined in the paper. Source code for the system outlined in the paper was later published, and so can be compared.
- The implementation used the same Scatterweb hardware Hydra was developed on.
- Fence monitoring is a specialized case of signal processing – in this case, accelerometer data over time. This processing has a large amount of scope for extension.

- The project helped to establish the validity of in-network data processing, a technique that our sensor network distributed operating system uses extensively.

## 6.3 The implementation

The implementation of the fence monitoring use-case is therefore a valid method of demonstrating the effectiveness of the Hydra distributed operating system. The first implementation task is to examine the fence monitoring system as described in the paper – henceforth referred to as the 'reference application' – as a Hydra user application.

### 6.3.1 The Reference implementation

The algorithm as described in the paper is as follows:

- Identify low-level events: events must be between 100 and 500ms. The total intensity must be above a pre-determined threshold.
- At least three events must take place inside 1750ms
- Old events are purged after this time.
- If at least three other nodes also have detected events, then trigger a climb event and inform the base station

The reference application was implemented in FACTS[88], a middleware package that provides a high-level query language for sensor networks. FACTS applications consist of 'rules' for processing sensor data, which are applied to filter sensor data and produce a result. FACTS runs on top of the Scatterweb software platform, which, like Hydra, was developed specifically for the Scatterweb hardware platform. Figure 6.1 shows an example of a FACTS rule. The full FACTS code for the fence-monitoring system is in Appendix A.

The reference application as described[95] contained 15 of these rules that, combined, process accelerometer signal data to classify events. The Hydra implementation of this application is written as a Java-based Hydra user application, implementing the core functionality described in the paper – identifying

```
rule aggregateBasicEvents 100
eval (( count { basicEvent }) >= 3)
eval (( sum { basicEvent duration }) >= 0.49)
eval (( sum { basicEvent duration }) <= 1.71)
define eventCandidate [ intensity = ( max { basicEvent intensity }) ]
retract { basicEvent }
```

Figure 6.1: An example of a single FACTS rule

and reporting specific movement events.

The reference application used a complex communication hierarchy, with several layers of nodes. Nodes were explicitly delegated into groups, and event data was passed up the various layers to eventually be distilled into a single event.

This communication architecture was specifically designed to optimize network traffic. From the paper:

> *'The advantages of this design for the raw data aggregation layer are twofold: Memory usage is kept at a minimum by aggregating sensor readings as they are being sampled, and excessive energy consumption is avoided during in- tervals in which no events occur. The drawback is that the raw data itself is not available for event detection. However, we regard this as unproblematic given the right selection of data items to aggregate.'*

The architecture diagram reproduced in Figure 6.2 displays this architecture.

While the FACTS middleware and the underlying Scatterweb system layer provide an abstraction layer to much of the low level operations (e.g 'transmit a buffer of data to a node over the radio'), the node hierarchy and interactions were explicitly specified in the FACTS ruleset.

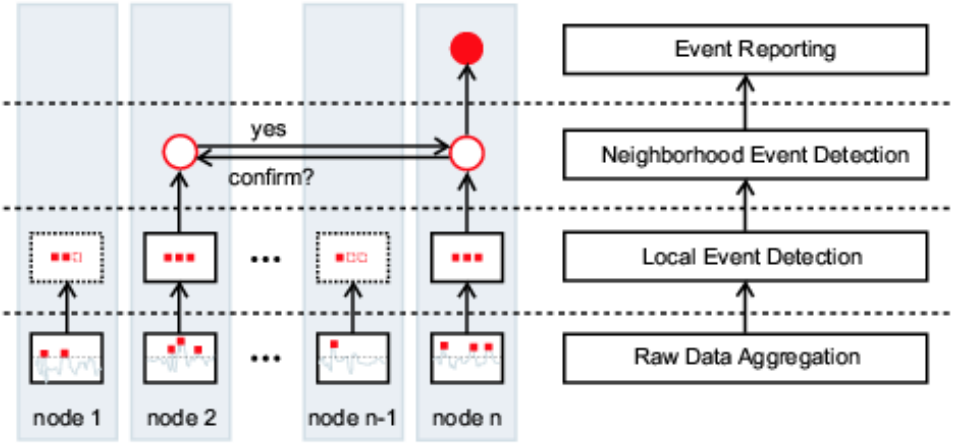


Figure 6.2: The network architecture of the reference application, reproduced from [95]

## 6.4 The 'Basic' Hydra implementation

One of the goals of the Hydra implementation was to leverage the distributed operating system paradigm in such a way that this explicit specification of node goals and hierarchy was avoided.

### 6.4.1 Implementation details

The first Hydra implementation of the system consists of a single Java application. A truncated code listing showing the application structure is reproduced in Figure 6.3.

This code is intentionally as simple and readable as possible – making a more compact implementation would be possible, but that was not the goal. It reads from sensors in turn, watches for an initial event above a threshold, then performs further processing using the algorithms described in the reference application paper. This processing determines if the event class is an 'intruder' or not.

When an event is detected, a piezo beeper on the currently executing node

```
class FenceDetect{

    static const int NUM_NODES = 8;
    static const int CACHE_SIZE = 10;
    static int THRESHOLD = 10;

    public boolean isIntruder(int [] data){
        //Data processing took place here as per the paper
    }

    public static void main(String[] args)
    {
        int cache[NUM_NODES][CACHE_SIZE];
        int pos = 0;

        for(int i=0;i<NUM_NODES;i++){
            Sensor s = Hydra.getSensor(i);

            int reading = s.read();

            //Data processing took place here as per the paper

            if(reading > THRESHOLD){
                Processor p = new Processor();

                if(p.isIntruder(cache[i]) == true){
                    System.out.println("Intruder alert!");
                    Hydra.setLED(0, true); //piezo beep
                }
            }
        }
        pos++;
    }
}
```

Figure 6.3: The initial implementation

is enabled and a notification is printed to the standard output stream – note that the `stdout` stream is automatically transported over the network to a log collector on a remote machine. A more complex event reporting system could easily be swapped in as desired.

### 6.4.2 Evaluation

This system operated in a similar fashion to the reference application, while needing no explicit separation or programming of node hierarchies to achieve the desired distributed event processing.

However, there was one major difference – this Hydra application was inherently single-threaded on a mesh level. Only one execution context could be

operating at once, and while this context could move over the mesh as necessary, overall efficiency was degraded as a result of the lack of parallel execution while code waited for other nodes to process data and return. It was possible for events to be lost while this took place.

## 6.5 The 'Parallel' Hydra implementation

A useful improvement to the basic implementation would be to support parallel execution. While this is not strictly necessary to create a distributed fence monitoring platform, it would mean that the CPU resources in the mesh could be more efficiently utilized.

The Hydra userspace API supports the concept of 'task' objects, which leverage the object migration infrastructure. Task object support is built into the standard userspace library, and the basic single-threaded application was henceforth modified to take advantage of this feature.

### 6.5.1 Implementation details

The new taskpool-based system defines a `Processor` object, a design pattern that means that a resource-intensive task is managed by a single object. This `Processor` object extends the `TaskPoolSensorProcessor` interface, which is part of the Hydra standard libraries and allows the object to be scheduled inside a task pool. A truncated code listing showing the the task-pool application can be seen in Figure 6.4.

Hydra also supports a subset of the Java threading APIs. Utilizing the `Runnable` interface is also an option to achieve distributed execution - in this way sensor nodes can be thought of as CPU cores that can be scheduled. A truncated code listing of the thread-based application can be seen in Figure 6.5.

```
class FenceDetect{

    static const int NUM_NODES = 8;

    class Processor extends TaskPoolSensorProcessor{
        public int run(Sensor s){
            while(true){
                //Will block until some data is available
                int reading = s.readBlocking();

                //Data processing took place here as per the paper

                if( /* intruder detected */ ){
                    return 1;
                }
            }
        }
    }

    public static void main(String[] args)
    {
        TaskPool tp = Hydra.makeTaskPool(NUM_NODES);

        for(int i=0;i<NUM_NODES;i++){
            Sensor s = Hydra.getSensor(i);
            tp.addTask(new Processor(s));
        }

        while(true){
            int [] results = tp.runAll(); //will block until something ↩
                returns

            for(int i=0;i<NUM_NODES;i++){
                if(results[i] == 1){
                    System.out.println("Intruder alert!");
                    Hydra.setLED(0, true); //piezo beep
                }
            }
        }
    }
}
```

Figure 6.4: A Parallel implementation (truncated)

It is generally more efficient to use the task pool system – the Hydra scheduler can use the knowledge of the Sensor object associated with the task to move the object to the correct node immediately instead of waiting for the location heuristics to select it. In addition, task pool tasks do not have to be scheduled to run simultaneously, so if there are insufficient resources remaining in the mesh to support simultaneous execution, some objects can be delayed as necessary.

However, the `Runnable`/Java threading system is more familiar to many programmers and so is retained as an optional use or for program architectures

```
class FenceDetect{

    static const int NUM_NODES = 8;

    class Processor implements Runnable{
        private Sensor s;

        Processor(Sensor sensor){
            s = sensor;
        }

        public void run(){
            while(true){
                int reading = s.readBlocking();

                //Data processing took place here as per the paper

                if( /* intruder detected */ ){
                    System.out.println("Intruder alert!");
                    Hydra.setLED(0, true); //piezo beep
                }
            }
        }
    }

    public static void main(String[] args)
    {
        for(int i=0;i<NUM_NODES;i++){
            t = new Thread( new Processor( Hydra.getSensor(i) ) );
            t.start();
        }
    }
}
```

Figure 6.5: A alternate threading/Runnable based Parallel implementation

that do not fit the task pool design pattern.

## 6.6 Extended system

In order to demonstrate the more advanced processing tasks possible with a distributed operating system, a more advanced test case is necessary. The advanced test case is an extension of the basic fence monitoring concept, improved in some way.

The results section of the reference fence monitoring paper states:

> *59.7% of the all other [non-fence-climbing] events are also classified as event candidates...this level of accuracy observed is less than the one we had expected...'*[95]

> *'...the level of accuracy we achieved in our experiments is by far not sufficient for a production- level deployment'* [95]

From this it can be seen that accuracy was an issue. The rationale for this as stated in the paper was a combination of a simplistic algorithm – causing large numbers of false positives – and contention for node resources caused by nodes combining data gathering and processing. A useful improvement to the reference system would then be an application that:

- Reduced or removed the contention caused by limited node resources
- Supported one or more complex signal processing algorithms, which could reduce the number of false positive results

The distributed operating system paradigm is very useful here in addressing both of these issues. The first point has already been addressed in the task pool / threaded implementations – Hydra will automatically place `Processing` objects in appropriate places in the mesh, where resources are not contended. Therefore the most important improvement would be to support more complex signal processing algorithms.

### 6.6.1 Fast Fourier Transforms

A Fast Fourier Transform (or FFT) is a very common signal processing task. It is an efficient method of computing the Discrete Fourier Transform (or DFT), a method of decomposing a signal into it's component frequencies.

It is important to note that while the details and exact uses of the FFT are outside the scope of this thesis, the process of calculating the DFT of a set of accelerometer data is a useful example of an intensive processing task of the kind that would be difficult to calculate in-network on a conventional sensor network system. This type of intensive in-network processing over large datasets is something that the distributed operating system services can make

relatively simple – the FFT/DFT algorithm is simply an example implementation of one such processing task.

### 6.6.2 Implementation details

The FFT system was heavily based on the task pool implementation. A `FFTProcessor` object was added that contains a standard Java implementation of the FFT algorithm. When the original algorithm detects a possible event candidate, the `FFTProcessor` object is initialized with the dataset and executed. It will be instantiated somewhere in the sensor mesh, the result will be calculated and then returned to the caller.

For simplicities sake, this implementation does not use a separate task pool or thread for the FFT calculations.

A truncated code listing of the thread-based system can be seen in Figure 6.6.

## 6.7 Accuracy and evaluation

The results of our implementations compared to the reference system are described and discussed in the following chapter, 'Evaluation'

```
class FenceDetect{

    static const int NUM_NODES = 8;

    class FFTProcessor{
        public int run(int [] data){
            //Perform the FFT calculation on data[] here
            return result;
        }
    }

    class Processor extends TaskPoolSensorProcessor{
        public int run(Sensor s){
            while(true){
                int reading = s.readBlocking();

                //Data processing took place here as per the paper

                if( /* intruder detected */ ){

                    System.out.println("Possible intruder");

                    //Now run the FFT and make sure
                    FFTProcessor fft = new FFTProcessor();
                    if(fft.run(s.getHistoricalData()) > THRESHOLD){
                        System.out.println("Intruder Alert!");
                        Hydra.setLED(0, true); //piezo beep
                    }
                }
            }
        }
    }

    public static void main(String[] args)
    {
        //as before, build up the task pool
    }
}
```

Figure 6.6: A parallel FFT implementation

# Chapter 7

# Evaluation

> *'A 'passing' test doesn't mean 'no problem.' It means no problem* observed. *This time. With these inputs. So far. On my machine.'*
>
> – Michael Bolton[90]

This chapter presents a discussion on the results obtained while testing the Hydra system, contrasting them with the results gained from the reference implementation described in *'Fence Monitoring – Experimental Evaluation of a Use Case for Wireless Sensor Networks'*[95], published at the European Conference on Wireless Sensor Networks (EWSN) in 2007.

We evaluate the following systems:

- The reference implementation (the 'full event detection' system as described)
- The 'Basic' Hydra implementation
- The 'Parallel' Hydra implementation

## 7.1 Evaluation metrics

We chose to compare several metrics when evaluating each system. These are:

- Sensitivity – how sensitive was the system in detecting an event. 100% sensitivity equates to ‘no events were missed‘.
- False-positives – how *accurate* was the system when categorizing an event into ‘intruder’ and ‘non-intruder’? 100% false-positive rate equates to ‘all events were mis-categorized‘.
- Complexity – how complex was the user application code? We use ‘application lines of code’ as an approximate proxy for program complexity here.
- Efficiency – how efficient was the system in terms of packets sent? This directly corrosponds to the amount of power required to run the system in the long term.

Each Hydra system was run for 1200 seconds under similar inputs to those that the reference implementation was given, as described in the paper. These were run in real life, not under simulation – while the *exact* data inputs the reference implementation was given are not available, the means of testing was described and a similar test applied to the Hydra deployment. Hydra results were averaged across 10 test runs.

Results for the reference implementation were taken directly from the paper.

## 7.2 Evaluation results

| **Criteria** | **Reference** | **HydraBasic** | **HydraParallel** | **HydraParallelFFT** |
|---|---|---|---|---|
| Sensitivity | 100% | 40% | 100% | 80% |
| False-positives | 59.7% | 60% | 70% | 30% |
| Efficiency | 1000 (approx.) | 10353 | 1623 | 1862 |

Due to the library support provided by Hydra, the lines of code required for a parallel task pool are relatively low. Note that the FFT complexity stated

above does not include the FFT calculation code, only the code used to invoke it. This code would typically be part of a standard library.

## 7.3 Discussion

- The Basic implementation was less sensitive than the Parallel implementation. This was due to the single-threaded polling architecutre of the basic application causing the system to miss events. The Parallel implementation successfully detected all the events, as it is not built around a single active polling thread.
- The false-positive rate is much lower on the FFT implementation, due to the FFT providing deeper analysis of the data at runtime.
- The efficiency of the basic implementation is relatively poor. The rapid polling caused a large number of packets to be sent and recieved throughout the mesh. Parallel implementation efficiency is comparable to the reference implementation, though still higher. This is due to the parallel processing/blocking nature of these implementations causing events not to be processed unless some activity has taken place.

## 7.4 Application Complexity Comparison

It is difficult to give a direct comparison of the overall application development complexity of Hydra vs traditional wireless sensor networks, as this is dependent on a number of factors. However there is one obvious comparison we can make comparing the sample codebases:

| **Criteria** | **Reference** | **HydraBasic** | **HydraParallel** | **HydraParallelFFT** |
|---|---|---|---|---|
| Lines of Code | 233 | 63 | 68 | 75 |

The complexity of the basic and parallel systems are low compared to that of the reference implementation. The code architecture is conceptually much simpler, thanks to the single-system image provided by Hydra. It should also be noted that the Java syntax that the Hydra applications are written in will most likely be easier to build, maintain and extend than the custom bytecode used in the reference implementation.

Further investigation and comparison using larger and more varied applications from other sensor network systems would be an interesting research problem for future work. It is likely that the single-system-image concept will lead to much simpler application code on larger user tasks.

## 7.5 Summary

We have demonstrated that it is feasible to use the Hydra sensor network distributed operating system to perform more complex sensor network tasks. Hydra applications can perform complex processing-intensive tasks and the distributed operating system layer will automatically place the workload in the appropriate place in the mesh.

Our FFT-based implementation has higher accuracy to the reference implementation described by Wittenburg et al while being conceptually much simpler from a programming perspective (discounting the in-app FFT implementation). This is possible because of the benefits of the distributed operating system.

Efficiency could be increased by further optimisation of the network protocol underlying the distributed operating system services.

# Chapter 8

# Conclusion

*'The function of good software is to make the complex appear to be simple.'*

– Grady Booch

## 8.1 Thesis summary

In this thesis, we sought to demonstrate that distributed operating systems techniques can be applicable to the wireless sensor network domain, potentially decreasing the complexity of user application development for wireless sensor networks. By utilizing a distributed operating system, the bulk of the complexity is removed from the user applications and application developers can be presented a familiar programming model using commonly deployed programming languages and APIs, conceptually treating the entire sensor mesh as a single logical computer.

In other words:

> 'By adapting distributed operating system techniques for the wireless sensor networks domain, wireless sensor network user application programming complexity can be decreased. These applica-

> tions can later be executed on a sensor network mesh in an energy-efficient manner.'

In order to demonstrate this, a distributed operating system for wireless sensor networks called 'Hydra' was designed, developed and evaluated in this thesis. This operating system provides the aforementioned distributed processing and operating system-level services on typical commodity wireless sensor network hardware. The examination of Hydra was addressed by discussing the following topics.

- **Theoretical basis**: For distributed operating systems on wireless sensor networks to be useful, it was necessary that the techniques used be applicable and adaptable to the sensor network domain, typical deployment tasks and characteristics of the data collected. This was investigated by reviewing both current sensor network operating systems and deployments, and traditional distributed systems techniques in the Background chapter. The specific techniques that were most useful from both domains and the approach taken to create a distributed operating system based on them were identified in the Design chapter.
- **Feasibility**: Using distributed operating system methods on a sensor network is theoretically possible, but it was necessary to demonstrate that doing so was practical. The Implementation chapter discussed the decisions made while creating the Hydra software, and any compromises to the design that were required as a result of moving from a theoretical design to a real-world development.
- **Suitability**: The Deployment chapter discussed the choice of a test scenario. The test scenario as described in this chapter was the chosen method of evaluating the effectiveness of the wireless sensor network distributed operating system. As such it was emblematic of a typical wireless sensor network task, as outlined briefly in this chapter and in more depth in the Background chapter.

- **Evaluation**: The Evaluation chapter discussed the results of the test scenario, focusing on correctness, performance and scalability. The previously outlined criteria for attributes of efficient wireless sensor network operating systems as well as distributed operating systems was be used to evaluate.

After these discussions were completed, a number of associated conclusions can be drawn from the Hydra system.

## 8.2 Conclusions

The Hydra operating system and associated software ecosystem as outlined in this thesis provide several benefits to simplify wireless sensor network application software development:

- 'A single system image' programming model, where the operating system provides an abstraction over the disparate and physically distributed underlying sensor node hardware, allowing the application developer to conceptually treat the entire sensor mesh as a single logical computer.
- A familiar programming environment, through the use of the Java toolchain and HydraVM. This means that many of the programming constructs, APIs and development tools that are commonly used in desktop or server development can now be utilized on sensor networks. Java is also familiar to a large number of developers outside of the sensor network domain.
- The hydrac buildchain provides tooling to compile, then compact and compress program code, easing the deployment process.

In summary, the single system programming model and familiar programming tools free developers to concentrate on their application rather than the sensor network environment.

The Hydra operating system also performs conventional distributed operating system tasks that on standard wireless sensor network operating systems are required to be handled manually by the sensor network application developer.

- Performance – by using the automatic process object/code layout systems discussed in the Design and Implementation chapters, the system will automatically make efficient use of the distributed hardware resources.
- Reliable – the system will expect and correctly handle hardware errors with the various failure modes described in the Implementation chapter.
- Resource name resolution – by presenting a single system image with well-known resource names, the system provides a useful abstraction layer to traditional resource name resolution issues.
- Resource management – By using the technique of automatic application code movement, Hydra will automatically balance to utilize the resources of the whole mesh, as the application tasks are spread across the devices.
- Synchronization – Concurrent processes inherently need to cooperate, and this must be done in a synchronized fashion to avoid errors and deadlock.
- Flexibility – the operating system must be adaptable to a range of conditions and deployments.

Furthermore in the Evaluation chapter we have demonstrated that these techniques allow the addition of significant and valuable additions of complex processing to a typical sensor network task, improving the results.

Notably these improvements are not limited to the example application – the ability to transparently utilize the processing power of the entire sensor mesh is an improvement that is useful to many classes of application.

## 8.3 Future work

The concept of applying distributed operating system services to sensor networks still needs additional investigation in several areas. There are a number of topics for further research, extending the Hydra system.

- More layout algorithms – find better ways of laying out the program code objects, as well as more efficient methods of optimizing their runtime movement.
- Mixing hardware classes – make a hybrid mesh that consists of Linux or specific high-powered nodes, which would transparently add processing power to the mesh.
- Smarter layout – at compile time, statically analyze the object code to identify hardware resources access, and place them accordingly. Add additional runtime metrics to place objects more efficiently.
- Add support for languages other than Java. Java was chosen because of its ease of splitting application code by Object boundary. C would be more efficient to run, but harder to partition.

# References


[1] airccse.org/journal/cnc/5413cnc01.pdf. A new programming model to simulate wireless sensor networks.

[2] Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer networks*, 38(4):393–422, 2002.

[3] H.M. Ammari. *The Art of Wireless Sensor Networks: Volume 1: Fundamentals*. Signals and Communication Technology. Springer, 2013.

[4] Ivan Andonovic, Craig Michie, Michael Gilroy, HockGuan Goh, KaeHsiang Kwong, Konstantinos Sasloglou, and Tsungta Wu. Wireless sensor networks for cattle health monitoring. In Danco Davcev and Jorge-Marx Gmez, editors, *ICT Innovations 2009*, pages 21–31. Springer Berlin Heidelberg, 2010.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[6] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[7] AWS. http://aws.amazon.com.

[8] Azure. http://www.windowsazure.com.

[9] Tobias Baumgartner, Ioannis Chatzigiannakis, Sándor Fekete, Christos Koninis, Alexander Kröller, and Apostolos Pyrgelis. Wiselib: a generic algorithm library for heterogeneous sensor networks. In *Proceedings of the 7th European conference on Wireless Sensor Networks*, EWSN'10, pages 162–177, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] Jan Beutel, Kay Römer, Matthias Ringwald, and Matthias Woehrle. Deployment techniques for sensor networks. In *Sensor Networks*, pages 219–248. Springer, 2009.

[11] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, 2005.

[12] Urs Bischoff and Gerd Kortuem. A state-based programming model and system for wireless sensor networks. In *Pervasive Computing and Communications Workshops, 2007. PerCom Workshops' 07. Fifth Annual IEEE International Conference on*, pages 261–266. IEEE, 2007.

[13] Niels Brouwers, Koen Langendoen, and Peter Corke. Darjeeling, a feature-rich vm for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 169–182, New York, NY, USA, 2009. ACM.

[14] Qing Cao, Tarek Abdelzaher, John Stankovic, and Tian He. The liteos operating system: Towards unix-like abstractions for wireless sensor networks. In *Proceedings of the 7th International Conference on Information Processing in Sensor Networks*, IPSN '08, pages 233–244, Washington, DC, USA, 2008. IEEE Computer Society.

[15] Mpala Research Center. fu-berlin.de/en/.

[16] Mpala Research Center. mpala.org.

[17] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph M. Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The design and implementation of a declarative sensor network system. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 175–188, New York, NY, USA, 2007. ACM.

[18] GNU ClassPath. gnu.org/s/classpath.

[19] Alexandre Courbot, Mariela Pavlova, Gilles Grimaud, and Jean-Jacques Vandewalle. A low-footprint java-to-native compilation scheme using formal methods. In *Proceedings of the 7th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications*, CARDIS'06, pages 329–344, Berlin, Heidelberg, 2006. Springer-Verlag.

[20] rica de Lima Gallindo, Francisco Vilar Brasileiro, and Vladimir Soares Cato. Reliable processing on the seljuk-amoeba operating environment. In *SCCC*, pages 105–114, 1997.

[21] Cormac Duffy, Utz Roedig, John Herbert, and Cormac J. Sreenan. Adding preemption to tinyos. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*, pages 88–92, New York, NY, USA, 2007. ACM.

[22] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[23] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM.

[24] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42. Acm, 2006.

[25] economist.com: smartphones reach masses. http://www.economist.com.

[26] Justin Edelson and Henry Liu. *JRuby Cookbook*. O'Reilly Media, Inc., 2008.

[27] MSP430 emulators. http://stackoverflow.com/questions/22285541/ is-there-an-emulator-of-msp430- chip-that-works- without-the-actual-chip-and-integ.

[28] Anna Förster, Amy L. Murphy, Jochen Schiller, and Kirsten Terfloth. An efficient implementation of reinforcement learning based routing on real wsn hardware. In *Proceedings of the 2008 IEEE International Conference on Wireless & Mobile Computing, Networking & Communication*, WIMOB '08, pages 247–252, Washington, DC, USA, 2008. IEEE Computer Society.

[29] Wai Fu Fung, David Sun, and Johannes Gehrke. Cougar: the network is the database. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 621–621, New York, NY, USA, 2002. ACM.

[30] GAE. http://appengine.google.com.

[31] David Gay, Philip Levis, Robert Von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Acm Sigplan Notices*, volume 38, pages 1–11. ACM, 2003.

[32] Lin Gu and John A. Stankovic. t-kernel: providing reliable os support to wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 1–14, New York, NY, USA, 2006. ACM.

[33] Ramakrishna Gummadi, Nupur Kothari, Ramesh Govindan, and Todd Millstein. Kairos: a macro-programming system for wireless sensor networks. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–2, New York, NY, USA, 2005. ACM.

[34] hadoop website. hadoop.apache.org.

[35] Heroku. http://heroku.com.

[36] MSP430 Homepage. http://www.ti.com/ww/en/launchpad/launchpads-msp430.html.

[37] Protothread homepage. code.google.com/p/protothread.

[38] Kirak Hong, Jiin Park, Taekhoon Kim, Sungho Kim, Hwangho Kim, Yousun Ko, Jongtae Park, Bernd Burgstaller, and Bernhard Scholz. Tinyvm, an efficient virtual machine infrastructure for sensor networks. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 399–400, New York, NY, USA, 2009. ACM.

[39] http://robotics.eecs.berkeley.edu/ pister/SmartDust/. smartdust.

[40] http://www.tomsguide.com. toms hardware best smartphones 2014.

[41] Paul Hunkin. *Operating System for Low-memory Devices.* URL: wand.net.nz/ pwh4/520/report.pdf.

[42] S. Ramakrishnan Ibrahiem M. M. El Emary. *Wireless Sensor Networks: From Theory to Applications.*

[43] Sun Java. http://sun.java.com.

[44] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet. In *ACM Sigplan Notices*, volume 37, pages 96–107. ACM, 2002.

[45] Josh Juneau, Jim Baker, Frank Wierzbicki, Leo Soto, and Victor Ng. *The Definitive Guide to Jython: Python for the Java Platform*. Apress, Berkely, CA, USA, 1st edition, 2010.

[46] Manish Karani, Ajinkya Kale, and Animesh Kopekar. Wireless sensor network hardware platforms and multi-channel communication protocols: A survey. In *Proceedings on 2nd National Conference on Information and Communication Technology, New York, NY, USA*, pages 20–23, 2011.

[47] Oliver Kasten. *A state-based programming model for wireless sensor networks*. PhD thesis, Citeseer, 2007.

[48] Joel Koshy and Raju Pandey. Vmstar: synthesizing scalable runtime environments for sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 243–254, New York, NY, USA, 2005. ACM.

[49] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu, and Bharat Bhargava. A survey of computation offloading for mobile systems. *Mob. Netw. Appl.*, 18(1):129–140, February 2013.

[50] Alan L. Leiner. System specifications for the dyseac. *J. ACM*, 1(2):57–81, April 1954.

[51] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. pages 115–148. 2005.

[52] Philip Levis. Experiences from a decade of tinyos development. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (Berkeley, CA, USA, 2012), OSDI*, volume 12, pages 207–220, 2012.

[53] Philip Levis and David Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM.

[54] Hua LI and Wen-xiu LI. Wireless sensory networks for the livestock, poultry and aquaculture industry pollution monitoring applications [j]. *Fishery Modernization*, 2:006, 2008.

[55] Xing Liu, Kun Mean Hou, Hongling Shi, Chengcheng Guo, and Haiying Zhou. Efficient and portable reprogramming method for high resource-constraint wireless sensor nodes. In *2011 IEEE 7th International Con-*

*ference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 455–460, Oct 2011.

[56] Jessica D Lundquist, Daniel R Cayan, and Michael D Dettinger. Meteorology and hydrology in yosemite national park: A sensor network application. In *Information Processing in Sensor Networks*, pages 518–528. Springer, 2003.

[57] Dalvik Virtual Machine. http://dalvikvm.com.

[58] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[59] mainframes: where are they now? Ia mainframeswhere

[60] Kevin Mayer, Keith Ellis, and Ken Taylor. Cattle health monitoring using wireless sensor networks. In *Proceedings of the Communication and Computer Networks Conference (CCN 2004)*, pages 8–10, 2004.

[61] Tomasz Naumowicz, Robin Freeman, Andreas Heil, Martin Calsyn, Eric Hellmich, Alexander Brändle, Tim Guilford, and Jochen Schiller. Autonomous monitoring of vulnerable habitats using a wireless sensor network. In *Proceedings of the workshop on Real-world wireless sensor networks*, REALWSN '08, pages 51–55, New York, NY, USA, 2008. ACM.

[62] nest. nest.com.

[63] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 489–498, New York, NY, USA, 2007. ACM.

[64] Future of cloud computing survey. http://talkincloud.com.

[65] Wikipedia: List of JVM languages. http://en.wikipedia.org/wiki/list of jvm languages.

[66] P8 PHP on the JVM. http://www.infoq.com/presentations/nicholson-php-jvm.

[67] Rhino Javascript on the JVM. http://www.mozilla.org/rhino/.

[68] Joseph Polastre, Robert Szewczyk, Alan Mainwaring, David Culler, and John Anderson. Analysis of wireless sensor networks for habitat monitoring. In *Wireless sensor networks*, pages 399–423. Springer, 2004.

[69] Princeton. http://www.princeton.edu/ achaney/ distributed database.html.

[70] KaiGuo Qian and ZhiQiang Xu. Reprogramming in wireless sensor networks. In *Proceedings of the 9th International Symposium on Linear Drives for Industry Applications, Volume 4*, pages 657–662. Springer, 2014.

[71] Mohammad Al Saad, Leszek Mysliwiec, and Jochen Schiller. Scatterplug: A plug-in oriented framework for prototyping, programming and teaching wireless sensor networks. In *Proceedings of the Second International Conference on Systems and Networks Communications*, ICSNC '07, pages 37–, Washington, DC, USA, 2007. IEEE Computer Society.

[72] Michael J. Sailor and Jamie R. Link. Smart dust: nanostructured devices in a grain of sand. page 1375, 2005. URL: http://duca.acm.jhu.edu/papers/Caching-1.pdf.

[73] Jochen Schiller, Achim Liers, and Hartmut Ritter. Scatterweb: A wireless sensornet platform for research and teaching. *Comput. Commun.*, 28(13):1545–1551, August 2005.

[74] Lars Schor, Philipp Sommer, and Roger Wattenhofer. Towards a zero-configuration wireless sensor network architecture for smart buildings. In *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 31–36. ACM, 2009.

[75] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java™ on the bare metal of wireless sensor devices: the squawk java virtual machine. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 78–88, New York, NY, USA, 2006. ACM.

[76] Pradeep K. Sinha. *Distributed Operating Systems: Concepts and Design.* Wiley-IEEE Press, 1st edition, 1996.

[77] Hugues Smeets, Chia-Yen Shih, Marco Zuniga, Tobias Hagemeier, and Pedro Josè Marrón. Trainsense: a novel infrastructure to support mobility in wireless sensor networks. In *Wireless Sensor Networks*, pages 18–33. Springer, 2013.

[78] Java specification. docs.oracle.com/javase/specs.

[79] Girts Strazdins, Atis Elsts, Krisjanis Nesenbergs, and Leo Selavo. Wireless sensor network operating system design rules based on real-world deployment survey. *Journal of Sensor and Actuator Networks*, 2(3):509–556, 2013.

[80] Ryo Sugihara and Rajesh K. Gupta. Programming models for sensor networks: A survey. *ACM Trans. Sen. Netw.*, 4(2):1–29, 2008.

[81] SunSpot. http://www.sunspotworld.com/.

[82] Marcin Szczodrak, Omprakash Gnawali, and Luca P Carloni. Dynamic reconfiguration of wireless sensor networks to support heterogeneous applications. In *Proc. of IEEE DCOSS Conf*, pages 52–61, 2013.

[83] Andrew S. Tanenbaum. Distributed operating systems anno 1992. what have we learned so far? *Distributed Systems Engineering*, 1(1):3–10, 1993.

[84] Andrew S. Tanenbaum and Sape J. Mullender. An overview of the amoeba distributed operating system. *SIGOPS Oper. Syst. Rev.*, 15(3):51–64, July 1981.

[85] Andrew S Tanenbaum and Sape J Mullender. An overview of the amoeba distributed operating system. *ACM SIGOPS Operating Systems Review*, 15(3):51–64, 1981.

[86] Andrew S. Tanenbaum and Robbert Van Renesse. Distributed operating systems. *ACM Comput. Surv.*, 17(4):419–470, December 1985.

[87] Andrew S. Tanenbaum and Robbert Van Renesse. Reliability issues in distributed operating systems. *Sixth Symp. Reliability in Distr. Softw. and Database Syst.*, 1987.

[88] K. Terfloth, G. Wittenburg, and J. Schiller. Facts a rule-based middleware architecture for wireless sensor networks. In *Communication System Software and Middleware, 2006. Comsware 2006. First International Conference on*, pages 1 –8, 0-0 2006.

[89] IBM Fault tolerance techniques for distributed systems. http://www.ibm.com/developerworks/rational/library/114.html.

[90] twitter: michaelbolton. https://twitter.com/michaelbolton/ status/485990054037159938.

[91] Xinyan Wang, Ruixin Zhang, Quanguang Ba, and Mingliang Wang. Nettree: A power-aware topology for wireless sensor networks. In *Proceedings*

*of the 9th International Symposium on Linear Drives for Industry Applications, Volume 4*, pages 693–700. Springer, 2014.

[92] Wikipedia. Article: Distributed operating system, distributed computing models section.

[93] Wikipedia. Article: Distributed operating system, history section.

[94] Wikipedia. Distributed operating systems : Design considerations.

[95] G. Wittenburg, K. Terfloth, F. López Villafuerte, T. Naumowicz, H. Ritter, and J. Schiller. Fence Monitoring – Experimental Evaluation of a Use Case for Wireless Sensor Networks. In *Proceedings of the 4th European Conference on Wireless Sensor Networks (EWSN)*, pages 163–178, Delft, The Netherlands, 2007.

# Appendix A

# FACTs code

```
/*
 * Copyright 2006, Freie Universitaet Berlin. All rights reserved.
 * Please refer to the COPYING file for detailed licensing information.
 *
 * Contributors: Georg Wittenburg
 */

/*
COPYING: Copyright 2005-2006, Freie Universitaet Berlin. All rights reserved↩
    .

These sources were developed at the Freie Universit t Berlin, Computer
Systems and Telematics group.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation
and/or other materials provided with the distribution.

- Neither the name of Freie Universitaet Berlin (FUB) nor the names of its
contributors may be used to endorse or promote products derived from this
software without specific prior written permission.
```

```
This software is provided by FUB and the contributors on an "as is" basis,
without any representations or warranties of any kind, express or implied
including, but not limited to, representations or warranties of
non-infringement, merchantability or fitness for a particular purpose. In no
event shall FUB or contributors be liable for any direct, indirect,
incidental, special, exemplary, or consequential damages (including, but not
limited to, procurement of substitute goods or services; loss of use, data,
or profits; or business interruption) however caused and on any theory of
liability, whether in contract, strict liability, or tort (including
negligence or otherwise) arising in any way out of the use of this software,
even if advised of the possibility of such damage.

This implementation was developed by the CST group at the FUB.

For documentation and questions please refer to the web site at
http://www.inf.fu-berlin.de/inst/ag-tech/projects/FACTS/

Berlin, 2006
*/

/**
 *
 * This ruleset implements fence monitoring, i.e. it aggregates low-level
 * events (both locally and within an n-hop neighborhood) and routes high-↩
     level
 * events to a base station.
 *
 **/

ruleset FenceMonitoring



/**
 * Generic system setup.
 **/

name system = "system"
fact system [broadcast = 255, tx-range = 10]
slot systemID = {system owner}
slot systemBroadcast = {system broadcast}
slot systemTxRange = {system tx-range}
```

```
/*
 * Define standard names and slots for this ruleset.
 */

name init = "init"
name shake = "shake"
name climb = "climb"
name alert = "alert"



/*
 * Step 0: Build Routing Tree / Route Alerts to Sink
 *
 * Build a spanning tree for routing and send any alert facts back to the ↩
     sink.
 */

name createRoute = "createRoute"
name route = "route"
fact route [nextHop = 0]
slot routeNextHop = {route nextHop}

rule buildRoutingTreeOnInit 250
<- exists {init}
-> set routeNextHop = systemID
-> define createRoute [source = systemID]
-> send systemBroadcast systemTxRange {createRoute}
-> retract {createRoute}
-> retract {init}

rule addRoute 240
<- exists {createRoute}
<- eval (routeNextHop == 0)
-> set routeNextHop = {createRoute source}
-> set {createRoute source} = systemID
-> send systemBroadcast systemTxRange {createRoute}
-> retract {createRoute}

rule retractCreateRoute 235
<- exists {createRoute}
-> retract {createRoute}

rule processAlertsAtSink 230
<- exists {alert}
<- eval (routeNextHop == systemID)
```

```
-> call print ("SE", {alert owner})
-> retract {alert}

rule routeAlertsToSink 225
<- exists {alert}
//-> send routeNextHop systemTxRange {alert}
-> retract {alert}



/*
 * Step 1: Node-Local Event Processing
 *
 * Aggregate low-level shake events into a high-level climb event.
 */

name localAggregation = "localAggregation"
fact localAggregation [
    discardEventsAfter = 10,    // Use 30000 for Haskell run.
    minShakeIntensity = 200,
    minShakeDuration = 100,
    minCombinedShakeDuration = 500,
    maxCombinedShakeDuration = 1750,
    minShakeEventsTrigger = 3
]
slot localAggregationDiscardEventsAfter = {localAggregation ↩
    discardEventsAfter}
slot localAggregationMinShakeIntensity = {localAggregation minShakeIntensity↩
    }
slot localAggregationMinShakeDuration= {localAggregation minShakeDuration}
slot localAggregationMinCombinedShakeDuration= {localAggregation ↩
    minCombinedShakeDuration}
slot localAggregationMaxCombinedShakeDuration= {localAggregation ↩
    maxCombinedShakeDuration}
slot localAggregationMinShakeEventsTrigger= {localAggregation ↩
    minShakeEventsTrigger}

slot newShakeTime = {shake time
    <- eval ({this modified} == true)
}

rule purgeOldAndWeakShakeEvents 200
<- exists {shake}
-> retract {shake
    <- eval ({this time} < (newShakeTime - ↩
        localAggregationDiscardEventsAfter))
```

```
}
-> retract {shake
    <- eval ({this intensity} <= localAggregationMinShakeIntensity)
}
-> retract {shake
    <- eval ({this duration} <= localAggregationMinShakeDuration)
}

rule aggregateShakeEvents 190
<- exists {shake}       // FIXME: Workaround for the rule engine.
<- eval ((count {shake}) >= localAggregationMinShakeEventsTrigger)
<- eval ((sum {shake duration}) >= localAggregationMinCombinedShakeDuration)
<- eval ((sum {shake duration}) <= localAggregationMaxCombinedShakeDuration)
-> define climb [confidence = ((max {shake intensity}) * (max {shake ↩
    duration}))]
-> retract {shake}



/*
 * Step 2: One-Hop Neighborhood Event Aggregation
 *
 * Broadcast new climb events to one-hop neighbors, reply with ACK or NACK
 * depending on local events. After a timer runs out, decide whether to send
 * the event to the base station.
 */

name ack = "ack"
name nack = "nack"
name neighborhoodSendDelayTimerExpired = "neighborhoodSendDelayTimerExpired"
name neighborhoodAggregationTimerExpired = "↩
    neighborhoodAggregationTimerExpired"

name neighborhoodAggregation = "neighborhoodAggregation"
fact neighborhoodAggregation [
    minShakeEventsTrigger = 3,
    minAckTrigger = 1
]
slot neighborhoodAggregationMinShakeEventsTrigger= {neighborhoodAggregation ↩
    minShakeEventsTrigger}
slot neighborhoodAggregationMinAckTrigger= {neighborhoodAggregation ↩
    minAckTrigger}

slot newReceivedClimbEvent = {climb
    <- eval ({this modified} == true)
    <- eval ({this owner} != systemID)
```

```
}
/*
slot newReceivedClimbEventID = {climb id
    <- eval ({this modified} == true)
    <- eval ({this owner} != systemID)
}
*/
slot newReceivedClimbEventOwner = {climb owner
    <- eval ({this modified} == true)
    <- eval ({this owner} != systemID)
}

rule delayOnNewLocalClimbEvents 150
<- exists {climb
    <- eval ({this owner} == systemID)
}
-> call defineLater ({neighborhoodSendDelayTimerExpired}, 1)

rule broadcastNewLocalClimbEvents 145
<- exists {neighborhoodSendDelayTimerExpired}
-> send systemBroadcast systemTxRange {climb}
-> retract {neighborhoodSendDelayTimerExpired}
-> call defineLater ({neighborhoodAggregationTimerExpired}, 3)

rule ackNewReceivedClimbEvents1 140
<- exists newReceivedClimbEvent
<- exists {climb
    <- eval ({this owner} == systemID)
}
-> define ack // [eventID = newReceivedClimbEventID]
-> send newReceivedClimbEventOwner systemTxRange {ack
    <- eval ({this owner} == systemID)
}
-> retract {ack
    <- eval ({this owner} == systemID)
}
-> retract newReceivedClimbEvent

rule ackNewReceivedClimbEvents2 130
<- exists newReceivedClimbEvent
<- eval ((count {shake}) >= neighborhoodAggregationMinShakeEventsTrigger)
-> define ack // [eventID = newReceivedClimbEventID]
-> send newReceivedClimbEventOwner systemTxRange {ack
    <- eval ({this owner} == systemID)
}
-> retract {ack
```

```
    <- eval ({this owner} == systemID)
}
-> retract newReceivedClimbEvent

rule nackNewReceivedClimbEvents 120
<- exists newReceivedClimbEvent
-> define nack // [eventID = newReceivedClimbEventID]
-> send newReceivedClimbEventOwner systemTxRange {nack
    <- eval ({this owner} == systemID)
}
-> retract {nack
    <- eval ({this owner} == systemID)
}
-> retract newReceivedClimbEvent

rule evalAcksOnTimeout 110
<- exists {neighborhoodAggregationTimerExpired}
<- eval ((count {ack}) >= neighborhoodAggregationMinAckTrigger)
-> define alert [confidence = {climb confidence}]

rule retractAcksAfterTimeout 100
<- exists {neighborhoodAggregationTimerExpired}
-> retract {ack}
-> retract {neighborhoodAggregationTimerExpired}
-> retract {climb}

rule retractNacks 90
<- exists {nack}
-> retract {nack}
```