Imperial College London
Faculty of Engineering
Department of Computing

Self-healing in Wireless Sensor Networks

Themistoklis Bourdenas

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the Imperial College London,
September 2011

Declaration of Originality

I herewith certify that all material in this dissertation which is not my own work has been properly acknowledged.

Abstract

Wireless sensor networks (WSNs) and pervasive systems are increasingly used for applications such as building monitoring and control, health-care and environmental monitoring. The users are frequently non-technical and devices may not be easily accessible, thus their management and complexity should be transparent to the users. To this extent, the systems need to be self-healing, able to respond to failures. We extend previous work on self-managed cell (SMC), which introduced an infrastructure for autonomous pervasive systems, with fault detection and recovery services.

We present a middleware for constrained platforms, which supports dynamic adaptation of network components imposing small overheads. It provides an event-driven paradigm for expressing system behaviour based on policies. We identify and define sensor readings' fault models extracted from long-running, real-world sensor deployments. We describe a fault detection mechanism for sensor readings based on heuristic and Bayesian probabilistic approaches that accurately identifies error occurrences in readings and minimises false positives. We implemented a recovery mechanism that responds to sensor and communication link degradation to dynamically reorganise the original role and task allocation among sensor nodes without disrupting service operations. Finally, we present a case study on a production-quality, multi-hop routing middleware, ITA Sensor Fabric, where we prototyped an adaptive routing mechanism, which set-ups virtual circuits for sensor data subscriptions avoiding recurring communication link and traffic congestion patterns that appear in the network.

Evaluation of the framework shows that the embedded policy management system is lightweight for power constrained nodes. The self-healing service accurately identifies erroneous sensors and is capable to effectively reconfigure network assets to improve quality of information while maintaining long life expectancy of the system.

Acknowledgements

I need to thank my thesis supervisor, Professor Morris Sloman, for his strong support and guidance thought my Ph.D. studies. His guidance was invaluable in order to shape this work to its current state. He helped me improve my research skills with his close attention to my work and critical thinking. I am very grateful to my second supervisor, Dr. Emil Lupu, for his advice and for asking the difficult questions required to improve the quality of my work.

I, also, have to thank my supervisors during my internships in IBM Research, David Wood, Petros Zerfos, Flavio Bergamaschi and in the National Institute of Informatics (NII), Kenji Tei and Shinichi Honiden for giving me the opportunity to carry out my research in different environments and bring different perspectives that provided a fresh view in my work as a whole.

Finally, I want to thank all my friends and colleagues in Imperial College, who made the office environment more pleasant and provided a distraction from work when one was needed.

‘Any sufficiently advanced technology is indistinguishable from magic.’

Arthur C. Clarke

Publications

Book chapter:

- Bourdenas T., Sloman M. and Lupu, E.C., "*Self-healing for Pervasive Computing Systems*", Architecting Dependable Systems VII, Springer LNCS 6420, Casimiro A., Lemos R. de, Gacek C. (Eds.), p. 1-25, 2010.

Conferences / Workshops:

- Bourdenas T., Wood D., Zerfos P., Bergamaschi F. and Sloman M., "*Self-adaptive Routing in Multi-hop Sensor Networks*", In Proc. of the 7th International Conference on Network and Service Management (CNSM), Paris, October 2011.
- Bourdenas T., Tei K., Honiden S. and Sloman M., "*Autonomic Role and Mission Allocation Framework for Wireless Sensor Networks*", In Proc. of the 5th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), Ann Arbor, MI, October 2011.
- Bourdenas T., Bergamaschi F., Wood D., Zerfos P. and Sloman M., "*Forecasting Routes and Self-adaptation in Multi-hop Wireless Sensor Networks*", In Proc. of SPIE conference on Defense Security and Sensing (DSS), Orlando, FL, April 2011.
- Bourdenas T. and Sloman M., "*Starfish: Policy Driven Self-Management in Wireless Sensor Networks*", ACM/IEEE ICSE 5th International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), Cape Town, May 2010.
- Bourdenas T. and Sloman M., "*Self-healing in Wireless Sensor Networks*", In the 8th International Conference on Pervasive Computing, Doctoral Colloquium, Helsinki, May 2010.

- Bourdenas T. and Sloman M., "*Towards Self-healing in Wireless Sensor Networks*", In Proc. of the 6th International Workshop on Body Sensor Networks (BSN), Berkeley, CA, June 2009.

Contents

List of Tables

List of Figures

List of Abbreviations

ACK	Acknowledgement
ACR	Adaptive Current Routes
AFR	Adaptive Forecasting Routes
AFSM	Adaptation Finite State Machine
AOP	Aspect Oriented Programming
API	Application Programming Interface
BSN	Body Sensor Network
CF	Calibration Function
CM	Calibration Matrix
CPU	Central Processing Unit
CTP	Collection Tree Protocol
DR	Delivery Rate
EBNF	Extended Backus Naur Form
ECA	Event Condition Action
ECG	Electrocardiogram
FP	False Positives
FSM	Finite State Machine

GAP	Generalised Assignment Problem
GNU	GNU’s Not Unix
GPS	Global Positioning System
GUI	Graphical User Interface
HMM	Hidden Markov Model
IBM	International Business Machines
ID	Identifier
ILP	Integer Linear Programming
ITA	International Technology Alliance
KB	Kilobytes
KPI	Key Performance Indicator
LP	Linear Programming
LQI	Link Quality Indicator
LU	Lower Upper
MAC	Media Access Control
NACK	Negative Acknowledgement
NFC	Near Field Communication
NN	Nearest Neighbours
NP	Non Polynomial
NSR	Network Status Report
OS	Operating System
PC	Personal Computer
PDR	Packet Drop Rate

PHT	Prediction History Tree
PRR	Packet Reception Rate
PSFQ	Pump Slowly Fetch Quickly
QAP	Quadratic Assignment Problem
RAM	Random Access Memory
RMSE	Root Mean Squared Error
ROC	Receiver Operating Characteristic
ROM	Read Only Memory
RSSI	Received Signal Strength Indication
SMC	Self Managed Cell
SML	Starfish Module Library
SNR	Signal to Noise Ratio
SPINE	Signal Processing In Node Environment
SR	Static Routes
SRV	Sensor Reading Validity
TC	Temporal Correlation
TP	True Positives
VM	Virtual Machine
WSN	Wireless Sensor Network

Chapter 1

Introduction

The development of small wireless sensors and smart-phones that include various sound, video, motion and location sensors has facilitated pervasive systems, i.e. applications that integrate in the environment and augment human-computer interactions. These include healthcare, which requires monitoring body readings as well as activity; traffic condition estimation using GPS and accelerometers; monitoring and controlling temperature, humidity and lighting levels in buildings; environmental monitoring and flood warning and even tracking wildlife movement. These pervasive systems are expected to perform in an extensive number of environments, ranging from urban to rural, with different requirements and resources. They are often mobile and thus subject to dynamic changes in environment and application requirements that demands flexible adaptation. Since users of such applications do not necessarily have technical skills, the system needs to be self-managing. Moreover, some applications such as healthcare and flood warning, may be life-critical. Consequently, the self-healing ability of the system with regard to faults and errors, becomes imperative.

Wireless sensor networks (WSNs) are the interface of the pervasive systems with the physical environment. They provide input necessary for integration and evaluation of the system's operational context. *Context* is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves [Dey01]. Due to the unique need of pervasive computing to blend in the physical environment of end-users, context information is vital. Consequently, the systems need to be context-aware. The above definition is very generic implying that the set of information, which is defined as context, depends on the application's needs. A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task [Dey01]. The actual deployment of sensing devices varies substantially from a small set of nodes to hundreds, forming complex network structures.

Typical sensor node devices run on batteries and thus have limited processing and power resources. Nodes communicate by creating asymmetric, ad-hoc, short-range and potentially mobile wireless channels. Rather than the traditional client-server model, the interactions between nodes resembles the peer-to-peer (p2p) model, where nodes interchangeably take the roles of both server and client. As a result, many of the research issues in p2p networking become relevant in the context of WSNs, such as efficient multi-hop routing, given that direct communication between nodes is not always feasible due to radio limitations. WSNs are usually organised in mesh or hierarchical structures for efficient communication. Information flows inside the network towards collection centres and eventually to one or more network information sinks. Sinks are gateway nodes that provide access to the WSN network cloud, collect data for analysis and provide an interface for pervasive applications to access the network's resources and interactive environments for administrators to manage them.

WSN management frameworks have been proposed for supporting node configuration and maintenance. Yet, many management frameworks found in literature [AMC07, PH07, RNL03, RCK+05] assume the existence of a centralised manager located behind a network sink that controls, synchronises network nodes and monitors faults and misbehaving network components. The manager can periodically query parts of the system to collect information and analyse metrics to discover malfunctions. A global view of the network is maintained centrally, including routing tables, node connectivity and energy maps. As the manager resides outside the network, an unlimited amount of resources and processing power are assumed, contrary to nodes inside the network. Consequently, the sink becomes an attractive point for executing a wide range of maintenance operations and performance evaluation. However, central management introduces overheads and limitations in a system as we discuss later in chapter 2. Moreover, proposed solutions look into the problem of fault-tolerance as separate components: *dynamic network infrastructure*, *sensor data faults*, *network failures*, *readings correction mechanisms*, *network recovery mechanisms*, etc., or as a combination of some of them. In this thesis, we look into a holistic, autonomous architecture that places these issues under a common framework that allows network administrators to express adaptation in a natural way.

1.1 Motivation

Fault management in distributed systems has been studied extensively in the past. Tanenbaum and Van Steen [TvS02] give a detailed summary of fundamental issues in fault-tolerance. In the literature three terms have been used to define defective behaviour in systems – *faults*, *errors* and *failures.* A *fault* is typically a consequence of a malfunction or misbehaviour of a system's component. An *error* is an unexpected state that the system should not have reached under normal execution, being the immediate of *fault* occurrences. Finally, a *failure* is the manifestation of one or more errors that have appeared in the system, as observed by the user. In distributed systems, *faults* are to be expected. The role of fault-tolerance is to prevent observation of *failures* by effectively handling *errors.*

Faults in pervasive systems, however, are more frequent than in traditional distributed systems. Due to their exposure in their operational environment, they are subjected to physical damage, circuitry fouling or user abuse. Therefore, provisions for handling and recovering from faults are critical. Systems should be able to cope with both transient and permanent errors by provisioning for potential outages or malfunctioning of their resources. However, the task of planning ahead to cover all potential erroneous states quickly becomes infeasible for any but the most trivial systems. Because of their complexity, prediction of all possible system states is a challenging problem. As stated before, the goal of fault-tolerance is to mask faults in order to prevent observation of failures. Fault masking is typically achieved by introducing redundant components and thus increasing the probability that sufficient working components remain in service, even though some may degrade or fail. A single malfunctioning component should not halt the operations of the entire system. Instead, the objective of a fault tolerant system is to achieve graceful quality of service degradation as components fail, containing the faults' areas of effect.

Fault management can be separated in to three tasks – *fault detection*, *fault identification* and *system recovery*. *Fault detection* deals with the exposure of an anomaly or malfunction in one or more system components. Typically, systems collect several operational metrics that quantify the system's performance. Comparison of the metrics to a model of expected behaviour indicates deviations that may hint at faults or component degradation that threatens the reliability of the system. *Fault identification* attempts to diagnose the cause of a system's malfunction using detected faults. This process provides a diagnosis inferred using domain-specific knowledge from observed symptoms. The inference process may be guided by heuristic rules or statistical and probabilistic models. Finally, *fault recovery* involves masking of fault and proactive actions to prevent failures. Masking can be achieved by exploiting redundancy in systems, for example, providing backup components that allow service operation without perceivable disruptions to users.

Pervasive systems and WSNs introduce unique characteristics to fault management that are not relevant in other distributed systems. Concerns such as power consumption – a limited resource in sensor nodes, high network drop-rates – due to the ad-hoc nature of communication, and incorporation of mechanical and physical components – such as thermometers, accelerometers, microphones, pH and ECG sensors that are exposed to every day use, constitute a unique challenge to fault-management in large scale networks. Exceptional cases are very likely to occur, hence, failures in WSNs are more frequent and are anticipated, being a part of the system's life-cycle. Human administrators of these networks have to cater for heterogeneous components, with regard to processing power, hardware resources, sensor devices, storage and communication capabilities [AMC07], which may fail in several ways. For example, sensor accuracy deteriorates over time due to physical phenomena such as overheating or chemical fouling. Additionally, external factors such as physical damage, variable wireless link quality (particularly in mobile systems) and devices that may fail completely disrupting network operations.

Frequent replacement of devices and manual recalibration are impractical even in small scale networks and hinder adoption by non-expert users with limited technical skills. A pervasive system is required to be self-diagnosing, self-healing and maintain operational state to mask component failures from users by graceful degradation of its service. Given the scale of intended deployment and pervasiveness, such systems are required to function as autonomous services that manage themselves and respond effectively to dynamic conditions with as little human intervention as possible. To this extent, in this thesis we explore the integration of autonomic computing principles in pervasive systems and propose an adaptive, self-healing framework.

1.2 Autonomic Sensor Networks

Autonomic computing [KC03] brings together different disciplines of computer science to promote *self-managing* systems that adapt to their operating environment and user needs. To this end, different systems exhibit different degrees of adaptation to dynamic environments without manual administration [HM08]. Autonomic computing is defined by its properties – *self-configuration*, *self-healing*, *self-protection* and *self-optimisation*, also referred as the *self-** attributes. *Self-** attributes are not orthogonal. For instance, in order for a network to heal a defective component, it should be able to reconfigure itself and possibly optimise the use of its resources to allow for alternative solutions.

Common among autonomic systems is a closed feedback control-loop pattern that consists of four distinct steps – *monitoring*, *analysis*, *planning* and *execution*, as illustrated in figure 1.1. The *monitoring* step collects performance metrics to measure the condition of different system components. The *analysis* phase processes information collected in order to infer the system's status. Analysis of monitored attributes may identify node or sensor failures, poor or failed communication links, low battery levels or poor quality of service due to overloaded processors or communication links. The *planning* phase constructs an alternative system configuration, based on observed symptoms and identifies actions necessary to counter defects and to transition to a better state. Finally, the *execution* phase applies the reconfiguration plan that

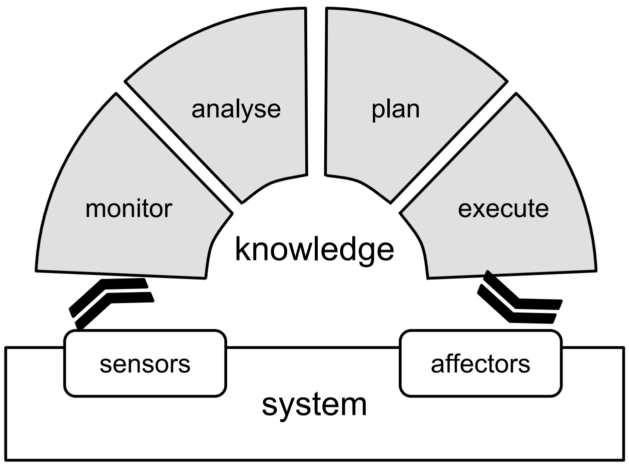


Figure 1.1: Closed feedback control-loop in Autonomic systems

has been produced. This process implies a dynamic system infrastructure that enables in-situ reorganisation without disrupting operations.

Sensor networks are structured in three layers, shown in figure 1.2, which presents the functional architecture of a wireless sensor network (WSN) application. The bottom layer, *sensing*, samples the environmental attributes and extracts features such as the mean or variance of sensor readings, e.g. sensing room temperature for air-conditioning control in buildings. The middle layer, *analysis*, processes events, e.g. change of room temperature, to infer system context. For instance, a node may infer room condition by fusing information from extracted features. Information fusion is not, however, restricted to features but extends to decisions as well. A decision fusion process combines localised or low-level decisions with domain-specific knowledge to infer higher level concepts, e.g. based on local decisions of several rooms it may be inferred that the air-conditioning in the conference room under-performs. The top layer, *network*, involves node management and orchestration. The functional role of the node manager is to organise the flow of information, e.g. the dissemination of decision from fusion centres to other ends of the network, as well as the nodes' structure in the network, e.g. ad-hoc, hierarchical, clustered communications, etc. The network manager finally assigns tasks among nodes and orchestrates their interactions.

Faults in this architecture propagate upwards affecting decision quality in higher layers. Consequently, faults need to be identified and contained as close to their source as possible. We look

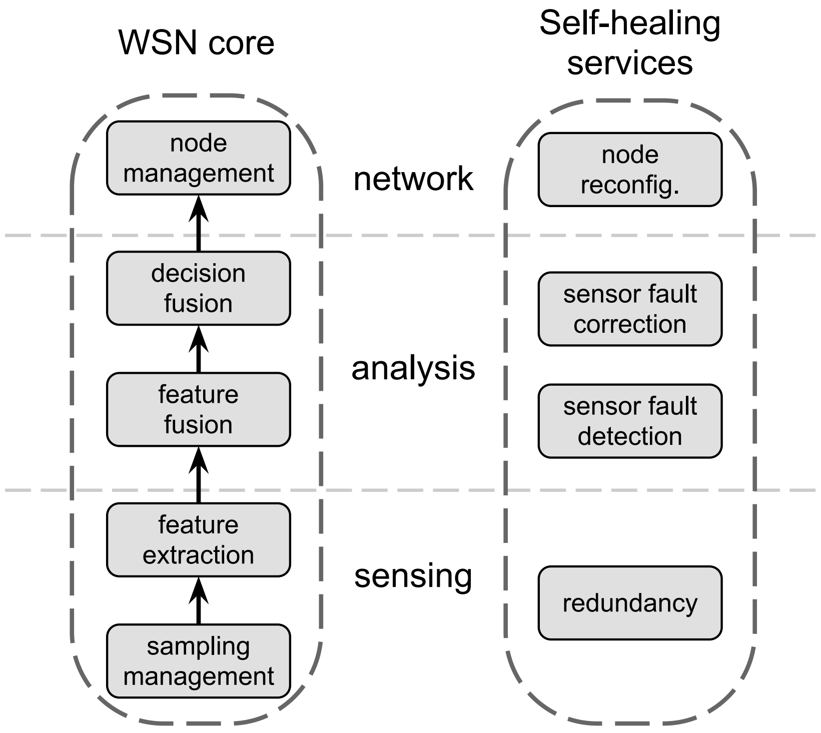


Figure 1.2: Functional architecture of a self-healing wireless sensor network

at extending services in different layers of the architecture that allow the network to tolerate faults and heal. In the *sensing* layer *redundancy* of resources, i.e. sensing devices deployed, provisions for fault tolerance. *Fault detection* and *correction* mechanisms can be deployed in the *analysis* layer based on features extracted from sensors and domain-specific knowledge of the application environment. Finally, at the *network* layer a reconfiguration mechanism is required to handle identified failures, which will be able to restore broken communication links, reallocate assigned tasks of failed components or redistribute available resources.

Approaches in autonomic computing can be split in two categories, top-down and bottom-up [CGL08]. We follow the former approach where typically an abstract model is defined in the system and the middleware attempts to refine it in order to be distributed among participants and implement the required service. In the latter approach instead of a model, requirements and goals are defined in the system. The global behaviour is expected to emerge out of interactions among components in the environment that form ad-hoc relationships, recursively composing larger systems to satisfy the specified requirements. Examples of such systems are swarm robots systems and their emerging behaviour from their interactions [KKL04, WFGDF05].

While the ad-hoc nature of bottom-up approach has certain properties, such as versatility and robustness, that benefit autonomic computing [McF86], we favour the properties of top-down approaches such as formal specification of the behaviour that is exhibited by the system and its control of its operations by human administrators.

1.3 Objectives

The objective of this thesis is to present an autonomic, self-healing framework for pervasive systems. We present an architecture for pervasive systems that includes the services necessary to identify, recover and potentially modify its operational plan dynamically in order to adapt to its operational challenges. We focus on failures caused by deterioration of a component's quality, rather than on systematic attacks of malicious participants. Moreover, we attempt to minimise manual administration of the network and delegate much of the maintenance tasks from the human operators to the the system itself. The services presented throughout this thesis are controlled by *policies*, i.e. Event-Condition-Action (ECA) rules, which are a lightweight means for expressing adaptation strategies. Policies, further, separate the adaptation aspects of the system them from its core operational logic and components.

We, initially, present the overall architecture of the system. In addition, we look into more detail individual parts of the self-healing service such as sensor readings fault-detection and recovery, autonomic task allocation among sensor nodes and pro-active adaptation for recurring communication link failures and congestion.

1.4 Contribution

The contributions of this thesis are briefly stated below.

- A middleware platform for WSNs that brings dynamic adaptation of network components in constrained platforms with small overheads and provides an event-driven paradigm for expressing system behaviour using the *policies* abstraction.

- Definition of self-healing services in the Self-Managed Cell (SMC) architectural pattern and prototyping an SMC instantiation for the tinyOS sensor platform.
- Identification and formal definition of sensor reading faults studied in long-running, real-world WSN deployments.
- A novel, adaptable, probabilistic fault detection mechanism for sensor readings that improves accuracy of sensor faults characterisation and minimises false positives.
- A dynamic task-allocation mechanism that responds to failures to mitigate service degradation due to sub-components' faults by reorganising network assets.
- A case study on a self-adaptive, multi-hop routing middleware for real-world systems that is designed to avoid repetitive communication link failures.

1.5 Thesis Structure

Chapter 2 presents a literature review in the fields of WSN management, network adaptation and fault handling.

Chapter 3 defines the operation of a sensor fault handling framework. It further provides background on the Self-Managed Cell (SMC) architectural pattern for autonomous computing systems. Finally, it presents an overview of self-healing services added in the SMC architecture.

Chapter 4 discusses *starfish*, a platform for embedded wireless sensor networks that follows the SMC architecture that introduces self-healing services. An overview of its components is presented, such as *finger2*, an embedded policy system and *starfish editor*, a mission specification and authoring environment based on obligation policies.

Chapter 5 formalises the models of different fault types in sensor readings and expands on a fault detection mechanisms that can be applied inside WSNs. Insight and evaluation results are provided from case studies in WSN applications.

Chapter 6 discusses autonomic adaptation of assigned WSN missions in response to continuously changing environmental conditions, component degradation and failures in the network.

Chapter 7 presents a case study on routing adaptation in response to failures in a multi-hop WSN network using time-series forecasting models to avoid repetitive communication link faults and congestion in high traffic networks.

Chapter 8 summarises the thesis contributions and presents discussion and future directions for research sections.

Chapter 2

Related Work

In this chapter we present an overview of the related work in the area. The study is partitioned in three parts. First, we look into management architectures for WSNs. We, also, examine programming paradigms and abstractions that support development in pervasive environments as well as network programming mechanisms for on-the-fly node adaptation. Finally, we present fault identification and handling approaches. They incorporate sensor node collaboration for defective node classification.

2.1 WSN Management

As the sensor network size increases manual administration and management of nodes becomes virtually impossible by human operators. Management frameworks have been proposed to support configuration, monitoring and remote administration of the nodes. Most existing frameworks focus on centralised managers that assumes the role of the leader that synchronises nodes in the network and collect information on performance for human operators to study [PH07, AMC07]. Such systems can adopt either an *active* monitoring model, where the leader periodically injects queries to different parts of the network from a sink node, or a *passive* monitoring model, where metrics are collected locally on nodes and are piggybacked on normal network traffic that eventually reaches the manager that resides outside the network. In the following paragraphs, we describe some representative WSN management frameworks.

2.1.1 Manna

Manna presented in [RNL03] is an an architecture for network management in WSNs. The network monitors node or network failures. The basic building blocks for the Manna architecture are the *services*, *functions* and *models* orchestrated together to construct a management scheme for the application. A *service* in the system is a complex operation that is a composition of *functions* provided by the framework. *Functions* are processes that operate on supported *models*. A *model* represents the state of a network's aspect, e.g. node energy levels. Execution of *services* is dictated by management policies that respond to model modification.

Models are built and reside in a base station outside the network and represent a global view of the system. Examples of *models* may include the *network topology*, which maps nodes to the physical area in their deployment, the *neighbouring set* of nodes that can be reached directly, i.e. in one-hop, the *routing table*, for relaying of messages in a multi-hop network and the *residual energy map*, which provides an estimate of nodes life expectancy. *Models* are built by *functions* that run on a network manager and collect network information by actively querying nodes.

Regarding the functional roles in the Manna architecture, three different type of participants are identified – the *manager* of the WSN, the *agents* and *standard nodes*. A single manager, which typically resides behind the sink, outside the network, controls the sensor cloud. *Agents* operate as local leaders in the network and are placed inside the network close to nodes they manage. They are essentially cluster-heads that aggregate information from standard nodes and propagate it to the manager in order to control communication costs inside the network.

The authors also describe a fault management system [RSO+04] that operates on top of the Manna architecture, as a proof of concept. It operates in a network of heterogeneous, clustered, event-based monitoring system for temperature observation. Being an event-based system, the cluster-head only sends notifications to the sink when temperature exceeds a predefined threshold. Fault-tolerance is provided by two services, the *coverage area maintenance* service and the *failure detection* service. The two services detect temporary or permanent node failures, i.e. power depletion or physical damage, and discover areas that are not covered by sensor monitoring. There are two phases in the fault-management operations, first, *installation* where services are initiated at nodes and second, *operation* that is monitoring during application's normal execution. During *installation*, nodes send information to *agents* that build localized topology and energy maps, which are finally aggregated at the *manager* constructing global maps. During the *operation* phase, nodes run management activities like energy level checking along with their normal readings collection tasks. When a node state is modified it reports it back to the *manager*. The *manager* can also query nodes by sending 'get' commands to *agents* over the network, which in turn will collect data from the nodes under their cluster and report them back to the manager.

The Manna architecture is based on a centralised management authority that collects information to build global models of the network and uses policies for adaptation. The adaptation decision point resides outside the network and thus, outcomes need to be propagated back to the nodes. The approach significantly contributes to shorter life of cluster-heads and nodes around the sink that support network traffic to keep network models up-to-date. In addition it introduces extended delays on message and decision delivery. Finally, the Manna network focuses only on network faults, not sensor faults, even though it appears that it could be extended with corresponding *functions* and *services*.

2.1.2 Sympathy

Sympathy [RCK+05] is an another approach for node management in WSN environments. It is meant as a debugging and detection tool for WSN applications during their development and deployment phases. It employs a centralized approach for detecting faults on data-gathering sensor applications and assumes networks with regular exchange of messages among nodes and the sink, rather than event-driven networks. As a detection tool, it provides failure reports on the network, localizing the suspected source and speculating causes. Causes fall in three categories; *node failure*, *path failure* between a node and the sink and *sink failure* when a node cannot be reached even though it appears operational to the rest of the network.

Even though information flows towards the sink for analysis, it does not impose any hierarchical structure in the network, e.g. cluster-heads or tree-like propagation. It employs both *active* and *passive* monitoring techniques for collecting network performance metrics. A failure is considered to happen if a monitored node generate less traffic than originally expected. The metrics for the decision are collected in three ways – nodes actively running Sympathy code and periodically transmitting collected data, sink passively monitoring application traffic and, last, extract metrics from the application that runs at the sink.

Sympathy collects three types of metrics related to faults in WSNs that focus on node communication. *Connectivity metrics*, such as routing tables and neighbour lists, *flow metrics* measuring node packet traffic load and *node metrics* that include node uptime as well as good and bad packets counters.

The localization technique for the failures includes four stages. The sink collects metrics from nodes and they are analysed to detect insufficient data received from nodes. If insufficient data are detected, Sympathy tries to pinpoint the root-causes of the failure from those metrics and possibly by actively doing some queries in the networks if necessary. Finally, in the last stage a localized source is attached to each failure. An empirical decision tree is the basis for network failures characterization.

2.1.3 SPINE

SPINE (Signal-Processing in Node Environment) [KGG+09] is a node programming framework developed for tinyOS 2.x [LMP+05] nodes that assists development of WSN applications focusing particularly on body sensor networks (BSN). Nodes are running the SPINE middleware while a base station coordinator endpoint aggregates information from nodes at the sink.

The framework's goal is to provide feature extraction components that assist in the distributed classification problem that BSN applications solve for obtaining activity recognition, contextawareness, patient's vitals condition, etc. The node side of the platform supplies development facilities such as data storage buffers, mathematical function libraries and common feature extractors for signal processing. Furthermore, it includes a features transfer protocol between the node and the BSN coordinator. At server side, the framework provides an API that can manage the sensors and make service requests on the network, scalable to adapt in different platforms like a PC or a mobile phone.

The framework supports only star network topologies with the BSN coordinator at the centre of the star. This is very reminiscent of the centralized applications, although, in the case of BSNs nodes are usually one hop away from the base station. Sensors unlike other management platforms do not transmit raw collected data to the sink, but do some initial local preprocessing, i.e. features extraction, before updating the sink, saving bandwidth and, consequently, power over the wireless medium.

2.1.4 Redflag

Redflag [UBH09] is a middleware platform for sensor nodes running tinyOS 2.x. It focuses as a support layer between the operating system and the application that helps to identify sensor readings as well as network faults. It consists of two independent services – *sensor reading validity* (SRV) and *network status report* (NSR), which provide fault notifications to the application through API interfaces. Similarly, the application can configure the parameters of the algorithms used.

The *sensor reading validity* service employs signal processing techniques to reduce the effect of persistent errors, such as noise or drift, as well as a rule-based system to characterise potentially suspicious readings and provide notifications to the application layer. Even though Redflag provides a mechanism to apply a linear calibration function $output = \alpha \cdot input + \beta$, there is no mechanism for detecting drift faults or extract parameters α and β. Instead, they are provided as parameters from the application. Moreover, the rule-based system is based on predefined heuristics that are domain-specific and rely solely on information gathered from a single sensor, without contradicting readings with surrounding sensing devices. Examples of rules are hard thresholds on readings' variance, upper/lower limits and difference between consequent samplings.

Similarly, the *network status report* service monitors fault that appear at the network level. The authors propose a collaborative scheme that is similar to other approaches in the literature [HL06, CKS06]. However, Redflag improves the scheme by using duty-cycling and bounding message broadcasts in epochs as well as allow more flexible thresholds for deciding that a node has disappeared. The main assumption for the technique to work is that nodes have synchronised clocks [SY04]. Nodes broadcast a 'hello' message that neighbours listen at predefined time periods. If a node misses a a consequent number of neighbour's 'hello' messages, it considers that neighbour suspicious and initiates a neighbourhood collaboration process to determine it has disappeared. It broadcasts an alarm for the suspected node and waits for a broadcast that rejects the alarm to which it responds with an acknowledgement message. If no rejection message is received after a number of epochs the node is considered failed. Nodes maintain a neighbour status table that contains a counter of missed messages, residual energy as well as link quality between nodes, which is also a metric that determines the selection of a node's neighbours. However, there is no discussion on how this metric is collected from the network.

The authors evaluate their techniques by replaying traces from existing sensor deployments, injecting persistent and random faults using Bernoulli processes. Node failures are similarly simulated as Bernoulli processes and for node link quality the SNR-based packet error model [LCL07] of the Tossim emulator [LLWC03] for tinyOS 2.x is used.

2.1.5 Summary of WSN management

WSN management systems tend to be centralised, dealing with fault identification and planning centrally. While a base station is a good processing point in the network with regard to computational power and storage it creates traffic bottlenecks, uneven power consumption on nodes and introduces operational delays due to multi-hop routing. We are looking into a framework that is designed to operate in a distributed fashion without requiring a central authority that collects a composes information. The need for communication is still required for reaching a consensus but we investigate ways that push decision centres inside the network rather than behind a gateway node.

Furthermore, most management frameworks are focusing on network faults without adequately catering for data faults that appear on their sensors. We are trying to amend for data faults that appear in the network to improve the quality of information extracted. Frameworks like RedFlag attempt to cover both aspects of faults in the network, however, the models introduced are simple heuristics that do not cover an adequate breadth of faults. Instead they are simple rules that rely on *a-priori* knowledge and hard thresholds on attribute values. Our approach supports such heuristics for cases where fault discrimination is simple, but also incorporates machine learning techniques in order to identify more complex fault types such as drift.

Finally, adaptation on these frameworks is limited and is not supported at the node level. We present a framework that allows behavioural modification to a significant extend on the node level. The framework consists not only of an adaptable centralised manager with monolithic small clients spread across an area. Instead it is able to disseminate behavioural changes and plan modifications over the entire network, which is not present in the management systems discussed in this section.

2.2 Network Adaptation

Fault management in WSNs implies a dynamic infrastructure where modifications can take place during runtime in order to address observed faults. Dynamic adaptation to new conditions at run-time is an essential attribute for self-healing pervasive systems. Adaptive systems have the benefit of undisrupted operation catering for new requirements that had not been predicted during the initial deployment of an application. The number of components in pervasive systems and their distribution renders manual reprogramming of nodes impractical. Exposure and interaction with the physical world further highlights the necessity for an adaptation mechanism as it is a dynamic and unpredictable environment. Adaptation is relevant in our work in terms of response to failures in the system by modifying network operations to utilise available resources.

In the following section, we discuss in more detail approaches in the literature for software and network adaptation.

2.2.1 Code Updates

Code updates have been proposed as a failure recovery process, where faults are caused by software bugs or there is a need to include additional features [HC04, KW05, PBM11]. Manual reprogramming of each sensor is impractical, even in networks with a small number of nodes. On-line software updates is currently supported by some embedded operating systems, notably Contiki [DGV04, DFEV06], and there have been extensions for tinyOS 2.x [MALW10], but the distribution of the updates over a multi-hop, unreliable network with high communication cost remains a research topic. A naïve approach for updating software on the sensors is flooding the network with the new version. Though it is a simple solution, it is inefficient as flooding of redundant large messages drain node resources quickly. Moreover, in heterogeneous networks nodes also receive updates that are not relevant to them.

A more elaborate method for disseminating software updates is presented in [MLM+05]. The configuration system of the network organises nodes with the same role so that they can be fully connected and updates are propagated through role specific network paths, reducing flooding of messages. Efficiency in the propagation process depends, of course, on the success of the configuration management process to distribute roles in the network so that nodes of the same role remain connected, given the remaining constraints of role assignments imposed by the application. In the case of fragmented areas of a functional role, neighbours of a different role must participate in the update process. The authors assume that role assignment in the network is static.

Another issue that the authors attempt to tackle is reduction on updates size transmitted over the air. The embedded operating system used is tinyOS [LMP+05], the de-facto operating system for WSNs, which builds all of the application's code into a single binary object. As a result the complete binary image needs to be transmitted to a node. Contrary to the clean separation of components existing in the source code level of nesC [GLB+03], the native tinyOS language, compiled code is a uniform binary image. FlexCUP is introduced which actually breaks the single C source files into multiple compilation units, called packages, based on the nesC components used. The outcome is the production of several smaller object units that can be distributed separately and linked on the motes themselves, instead of the PC. In this way, the authors succeed in distribution of smaller updates, by propagating into the network only packages that have been modified by the update, saving retransmission of standard components that tinyOS provides.

Impala [LM03] is middleware for driving adaptation in WSN. The concept in Impala is to provide a native adaptation mechanism that does not rely on embedded virtual machines for reprogramming nodes. Instead, it hot-swaps running applications. Applications that run on sensor nodes are comprised of independent modules that are compiled into native, binary code. Modules are the smallest units of updates that are transmitted to nodes. Even though module transmission limits the updates size significantly it still appears that the granularity is still coarse resulting in significant overheads for small changes, such as parameter modifications. The adaptation mechanism in Impala is hot-swapping of applications that is driven by two sources – applications and devices. Application driven adaptation is expressed an adaptation FSM (AFSM), where states are different applications that are installed on a node and transitions occur when certain application specific conditions are met. Device driven adaptation is triggered by a failure in a device. Applications provide dependency associations with hardware devices that nodes are equipped with, e.g. GPS device, radio etc. When the middleware detects a device failure, dependent applications are turned off and replaced by back-up solutions specified by the developer. The design of Impala is tied to the facilities provided by the linux operating system that runs on hand-held personal devices, used for prototyping. However, the majority of sensor nodes operate on more constrained environments [LMP$^+$05, DGV04] that do not share similar properties. Furthermore, the impact and effectiveness of the update distribution mechanism has been evaluated over the IEEE 802.11.2 communication channel that is very reliable and power consuming compared to the more widely used IEEE 802.14.5 protocol that is used in low-power sensor communications.

Transmission of node images, even if they involve only parts of the program, impose big overheads for program modification, especially when changes are frequent and small modifications and the number of affected nodes is large. Even dissemination of only a few KB of code over the air may prove to be more than the limited power constrained nodes are able to handle. Maté [LC02] is an attempt for lightweight software updates in sensor nodes. Maté is a virtual machine (VM) developed for sensor nodes running tinyOS. It allows a customisable instruction set, where instruction op-codes can be associated to native functions. Nodes are reprogrammed by transmitting updated scripts that run in the node virtual machine. Script transmission in Maté allows significant reduction in code size transfers, but it is not possible to modify the mapping of op-code to functions at run-time or add new native functions on the node.

2.2.2 Policies

Policies are another means of lightweight adaptation in response to events. They have been used in traditional distributed systems for network management [Slo94]. Policies define responses to events in the system and their expressiveness is constrained, as it is not a general purpose programming language. Instead, it permits invocation of lower-level components, developed in native code and installed on sensor nodes. Policies simplify parametrisation of algorithms and allow delegation of network management to a less technical administrator instead of the system developer. In addition, policies can be transformed in formal logic for analysis and conflict resolution that is not possible with imperative languages. Management frameworks for sensors that handle fault detection have also been proposed in the past [RNL03, LDCO06]. Some of these frameworks use policies for reacting to detected faults inside the networks. However, policy evaluation points are located centrally typically in the sink and decisions are taken by collecting information from nodes and later pushed back to the nodes.

Facts [TWS06] is a middleware architecture for WSNs that uses policies for decision-making and adaptation in the network. The authors use the term *rules* instead of policy, but they are essentially the same event-condition-action (ECA) mechanism. Information in the system is represented as *facts* that are stored in a local *fact repositories* and processed by *rules*. *Rules* incorporate application logic in an event-based, high-level language. *Rules* can call functions that implement resource-critical operations in the native code of the platform. A *fact* can be a simple value, constrained to boolean, integer, float or string, without any support for aggregate types like arrays or lists. *Facts* and *rules* are stored and operate on sensor nodes inside the network, allowing scripting node behaviour using rules in a distributed scheme. Some support constructs are introduced such as *rule-sets* that are collections of *rules* and *facts* that form closed components and *slots* that are a filtering mechanism for *fact* selection from the repository. However, there is no attempt for adaptation using mechanisms for reprogramming node behaviour with new sets of *rules*, even though transfer of *facts* between nodes is supported. Finally, modelling information as *facts*, i.e. values, and *rules* that control operation on them does not support encapsulation. In contrast modelling rules to respond to events from modules would provide more object-oriented and reusable entities that hide implementation details from the policy author providing only interfaces. The *rule-sets*, the system's equivalent of a component, are a remedy for associating data with operations, but they still expose the internals of the component.

Auxo [DWSC10] is an architecture-centric approach to adaptation in pervasive systems. It uses a model very similar to that introduced in the nesC [GLB$^+$03] programming model, where *components* are defined by the *interfaces* that they *provide* and *use* making them closed, reusable building block for service specification. Application are defined by wiring components together, i.e. connecting *use* and *provide* slots among components. While wiring in nesC is static, defined at compilation time, Auxo allows run-time adaptation by replacing hardcoded wiring of components with policies, i.e. ECA rules, that dictate service composition. The runtime reconfiguration process is supported by an *adaptation service* in the system. The system is designed for *x86* and *ARM/linux* platforms. Instead, we scale down for embedded, resource-constrained platforms.

Misra and Jain [MJ10], also, use policies as a distributed decision-making mechanism in the network, but constrained their use for self-configuration and adaptation of sensor node topologies, i.e. building neighbourhood groups, based on three metrics; physical distance between nodes, residual energy and neighbour count per node.

Pham et al. [PPS+09] propose an adaptation platform for pervasive systems that is based on component graphs. It separates a pervasive applications in two distinct layers – the *constructor logic* and the *component implementation* layer. An application is defined as a composition of primitive components that provide input/output interfaces. The components can be grouped to define new aggregate components that are considered as a single unit defining their own connections. The approach for component interfacing and composition closely resembles the nesC [GLB+03] paradigm for module definition and wiring. The main distinction, however, is that components in this case are *network objects* that can remotely interact over network protocols. This introduces the necessity for *discovery* and *event queue* mechanisms for building a directory of available *network objects* and manage asynchronous communication through events and notifications. Finally, adaptation is supported by the hot-swapping ability of components in the system. The main requirement for hot-swapping is for a component to define interfaces that allows to serialise its state in order to be replaced by a different component that provides the same interface options to replace the original functionality. Adaptation logic is expressed as a transformation in the component graph by modifying connections and replacing components in response to events. In spite of targeting pervasive systems, the platform applies only to unconstrained devices as constructor and adaptation logic is stored and processed centrally and assumes high-capacity, reliable communication channels between network objects. It has been prototyped for desktop servers and hand-held devices like smart-phones.

We use policies as the driving mechanism for adaptation in our framework as well. However, policies are much more integrated in our platform as they are able to control virtually all operations of the system. In the approaches discussed above, policies are used either centrally as a decision mechanism or they are used to control only an aspect of the system. We use policies as the means to define behaviour, modify parameters and compose available components and resources in the network in order to build comprehensive services. Policy evaluation and decision points are moved inside the network operating on the sensor nodes themselves. We build on top of policies higher level construct and tools that dictate the operations of the network and define how components are composed and interact with each other. Adaptation occurs in terms of policies by applying a different set of rules. As policies are lightweight and can be easily transmitted without disrupting node operations, modifications of services and operational plans happen by transmission and activation of policies to remote nodes. We discuss these aspects in more detail in chapters 4 and 6.

2.3 Sensor Fault Handling

In this section, we present fault handling approaches from the literature that present fault detection and adaptation mechanisms as well as modelling of sensor networks to exhibit tolerance of faults inside the network utilising neighbourhood collaboration and sensor redundancy. Most distributed fault handling approaches, in an attempt to reduce traffic, initially locally determine fault occurrences before propagating information to a central base station. This way, they reduce control message overheads, thus, energy costs.

2.3.1 Fault Detection

In distributed approaches, a considerable part of fault detection is transferred to the nodes themselves. The goal is self-monitoring and self-detection of individual parts of the network. For example, an individual node could detect a poor link quality by the number of packet drops it had with an adjacent node but it is not able to report a failure in its communication, or it could report a very low power level but not the depletion of its battery. The latter events could only be reported by third-party observers. Nevertheless, moving some monitoring tasks to the sensors reduces communication costs; hence lengthens network overall life. Based on the fact that physical events are spatially and time correlated, while faults are exceptional cases, neighbourhoods can utilise sensor observation correlation to identify the cause of irregularities in readings.

A localised algorithm for faulty sensor detection is presented in [DCXC05], for robust event boundaries detection in dense WSNs. Neighbouring nodes cooperate to detect abnormal behaviour of nodes that present a large deviation in a constrained area. A neighbourhood $N(S_i)$

is defined as a set of nodes around node S_i and a set $\{x_1^{(i)}, x_2^{(i)}, ..., x_k^{(i)}\}$ of readings from its nodes. The deviation of nodes from the median value of the neighbourhood is computed to characterise faulty nodes that exceed a threshold. Median is commonly used instead of the mean as it is more robust, filtering out extreme outliers in a set.

The authors in [CKS06] extend the idea for scenarios in which nodes are not aware of their physical location. The authors present an alternative algorithm, also based on local communication that focuses on sensor errors, such as drifting, random noise and completely irregular values. Nodes initially exchange readings within their clique and calculate differences. In the second step, each node separates its neighbours into two sets – those that deviate below a threshold and the rest. If at least half of the neighbours belongs to the former set, the node is characterised as 'likely good', otherwise 'likely faulty'. In the third step, 'likely good' nodes collect the characterisation of their neighbours. If at least half of them are 'likely good' then the node is promoted to a 'good' node. Finally, the remaining 'likely good' and 'likely faulty' nodes are given a definite label based on the 'good' nodes classified in the last step.

Node collaboration is also employed for node faults [HL06], where a group of nodes collects high confidence in a decision, e.g. a node's disappearance, before sending an alarm to a central base station to handle the fault. A two-phase timer $C = \{C_1, C_2\}$ approach is proposed, where nodes maintain a timer pair for each neighbour, which is reset when a message from the corresponding node is received. If timer C_1 runs out, then the node suspects that its neighbour may have failed and broadcasts a liveness query to its area, while starting timer C_2. Nodes that receive the broadcast either confirm that the suspected node is not reachable or contradict it with a response stating last time they have interacted with it. The scheme enables nodes to decide locally whether a node has failed or the timer expiration was due to a partial fragmentation of the network.

Multi-sensor collaboration is also demonstrated in [KPSV03] for identification of faulty sensors. The impact of each particular sensor is evaluated during information fusion. If removal of a sensor from the fusion process yields consistently improved accuracy compared with an existing model then the sensor is regarded as faulty. Combination of different types of sensors allows for cross validation of evidence for sensed phenomena, strengthening inference belief of the system.

Self-diagnosis on sensor nodes [HRR05] introduces a self-monitoring layer on nodes consisting of three accelerometers that monitor potential node impacts, by inferring collisions from irregular and intensive movement in the three-dimensional space, which may partially damage the sensor and affect its operation. Nodes analyse input from the accelerometer sensors trying to match it with known impact patterns. This approach differs from traditional fault identification approaches that use external observers to identify consequences of faults, while this approach tries to identify a cause before a malfunction becomes apparent. It, further, requires additional, application-specific, hardware components, i.e. the accelerometers, for fault detection. As long as the communication module of the node remains intact, a node can provide hints that it might have been affected by physical damage.

FIND [GZH09] is a centralised approach for fault sensor data detection. It does not rely on an *a-priori* model of the observed events or neighbourhood collaboration. Instead, it bases fault detection decisions on ranking of nodes and their distance. It targets networks that use sensors whose signal attenuates in space, e.g. acoustic volume or radio signal strength. The position of all sensor nodes is assumed to be known to the base station, which partitions the space in multiple *faces*. Faces are a fragmentation of the physical space, such that for each face a unique ascending distance sequence to the deployed sensors exists. The algorithm proposed operates on mismatches of observed event intensities sequences with estimated face sequences in order to identify sensors that exhibit either biased or random reading faults. The approach, however, is restricted for attenuating signals and could not work for a substantial set of sensors such as thermometers or accelerometers. Moreover, it is a centralised approach that requires collection of all information in a base station. The nodes are assumed to be stationary, as movement would require recalculation of faces fragmentation, which is computationally expensive to update on-line. Finally, sensor node deployment density dictates the granularity of the physical space partitioning in faces. This makes detection accuracy highly dependent to node density, where high accuracy levels may require a high density/redundancy of nodes in the environment.

2.3.2 Tolerating Faults

There are approaches in the literature that avoid the fault detection phase acknowledging that all sensor observations are subject to faults. Instead, they try to minimise the amount of transient errors by using models of monitored attributes based on a-priori knowledge. An issue of this approach, apart from how to build a model, is the mechanism that can decide between *observed* and *predicted* values produced from models.

The prediction history tree (PHT) [MPD04a] attempts to tackle this issue by building a binary tree that represents the time-line of sensor readings, where each tree level represents the potential values of the monitored attribute at the a time instance. The leaf nodes represent time instance t, the current time instance, while the root of the tree is time instance $t-h$, where h is the tree depth. The children of a node are the *observed* values from the sensor and the *predicted* values from the model. An error value is associated with every node, which is the difference of its value from the *observed* value. When a new reading arrives the tree selects a new root from current root's immediate children, which is going to represent the node's output for the time instance $t-h$. Consequently, the approach assumes that the applications are tolerant to some configurable delay of h samples from nodes. The authors examine different approaches on selection of a new root based on the error costs of their branches. They study the effectiveness of minimum root mean square error as well as a min-max approach [MPD04b]. Apart from the implicit requirement of delay tolerance, the approach also requires large amounts of memory in order to maintain large binary trees. The accuracy of the approach is a trade-off against delay and storage requirements. Finally, the approach is applicable on binary decisions, i.e. high-level states, rather that readings from sensors.

An alternative to model employment for monitored attributes is fusion of observations from multiple sources, which can be used to compensate for fault occurrences. An example of multi-modal sensor fusion for compensating for errors is described in [KPSV02]. The authors demonstrate a fault-tolerant system for moving nodes location detection that incorporates four types of sensors; Received Signal Strength Indication (RSSI) distance discovery, speedometer, accelerometer and a compass. Different metrics are associated by building a mathematical system that uses the functions of Euclidean distance in two-dimensional space, Newton mechanics and trigonometry laws. The resulting model contains fifteen equations describing twelve variables. Consequently, a model of the physical world is provided, rich enough that the equation system remains solvable even when a subset of its equations is missing. Moreover, malfunctioning sensors can be identified by looking at variables that do not satisfy the system. The scenario described is just a motivation example proposed by the authors. While the approach can potentially be very effective, it is not always possible to accurately define associations of monitored attributes in a way that would provide strong indications for faults.

Multi-sensor fusion becomes more straightforward when observations from directly comparable sensors are fused, i.e. sensors that measure the same attribute. The benefits from sensor fusion are increase of Signal to Noise Ratio (SNR), enhanced robustness and reliability in sensor failures, identification of malfunctioning sensors and improved resolution and precision of measurements decreasing the uncertainty of the decisions.

Feature fusion in homogeneous sensors has been used for reliable location extraction of sensors [KHB07]. Measurements from sensors include signal strength and angle of receiving over an IEEE 802.11a network. By combining readings from unreliable sources of relative locations, they can average out systematic error, to better pinpoint the location of a sensor in the environment.

Data fusion is also proposed in Sasha [BBJ05] that outlines a self-healing framework for sensor networks. A centralised architecture is proposed for production of monitoring scripts that define quality on readings collected from a group of densely deployed sensors monitoring the same attribute. A neural-network approach is used to determine validity of sensor readings and later disseminate quality weights for sensors to a tree hierarchy topology of nodes. The approach is not very flexible as it requires centralised collection of data for decision making and imposes a hierarchical tree structure in a network that requires more processing and communication to maintain.

A distributed approach for decision-level fusion from sensors is presented in [WHVC05]. Error correcting codes are introduced for distributed, fault-tolerant classification fusion. Classification of an M-class problem is solved in the fusion centre by the combination of binary decisions from deployed sensors. The error-correcting code matrix for an M-class problem with N observing sensors, is defined as a $M \times N$ matrix, where rows are codewords that correspond to classes and columns represents the binary decision rule for a sensor. The fusion centre collects binary decisions from corresponding nodes and deduce a final decision from the code matrix. It may be possible that there is no exact match of the aggregated binary decisions with a codeword, thus the closest codeword is preferred. The Hamming distance is used as the metric between codewords. The Hamming distance of two binary vectors corresponds to the number of distinct positions that contain different symbols. The fault-tolerance of the system is established by the minimum Hamming distance of codewords, being tolerant to erroneous local decisions from a subset of sensors. Appropriate encoding of the code matrix has a significant effect to the effectiveness in tolerating faults by the fusion centre. Ideally, the codewords should exhibit maximum possible distance from each other to maximise sensor faults endurance. The design of a code matrix is not a trivial issue, exhaustive search for an optimal code matrix is a computationally intensive process, which is, nevertheless, carried out off-line.

Ye et al. [YMC^{+}08] reason about information fusion in order to indirectly cope with errors in terms of higher-level context. Context is defined as triplet of *subject*, *relation*, *object*. Mapping functions transform sensor input of *subject* and *object* into a different domain that allows values to be associated in terms of granularity, i.e. associate a 2D position vector with the coarser information of 'conference room'. Consequently, the authors build a framework that is based of Bayes theorem to calculate conditional confidence by integrating (i.e. fusing) context information, collected from unreliable sensors, that is overlapping or containing.

Sensors tend to gain a systematic bias and drift due to ageing or low power. As a result, readings are gradually moving away from the ground truth, degrading the quality of observations and the accuracy of the overall system. Manual replacement and recalibration of drifting sensors can be both impractical and cost-ineffective even in networks of moderate size. An alternative to replacement is on-line recalibration of the sensing devices by applying a translation function on their readings, returning to an acceptable state of operation.

A collaborative in-situ recalibration [BMEP03a] attempts to detect and correct systematic bias, but instead of being based on supplying known stimuli for the sensors to calibrate against, it uses observations from adjacent nodes. The approach is different from traditional calibration since sensors are calibrated against outputs from their peers instead to external, manual measuring tools. The approach assumes synchronisation among participating nodes. A calibration function (CF) $F_{i,j}(x)$ is defined that maps a reading x of sensor s_i to the reading of sensor s_j. A confidence level value $w_{i,j}$ is attached to a calibration function. A node's calibration matrix (CM) is defined as a matrix of size $S \times S$, S being the number of collaborating nodes. Element (i, j) of the matrix corresponds to the function $F_{i,j}$.

The calibration process is separated in two phases; initially pair-wise relationships between nodes are calculated, i.e. the $F_{i,j}$ functions. The second phase is formulated as an optimisation problem on functions $F_{i,j}$. Pair-wise functions are computed by collected time-series of neighbouring readings, weighted to represent the correlation of the sensor pair. Lowly correlated time-series are filtered out and linear functions of the pair-wise difference of sensors are produced. Functions $F_{i,j}$ are expected to be inconsistent, due to sensing and fitting errors. The second phase of the algorithm minimises the inconsistencies using a heuristic optimisation algorithm on the calibration matrix.

2.4 Summary

We gave an overview of representative WSN management systems in the literature with a focus on the fault aspects of the network and what fault detection mechanisms they employ. We also provided a summary on the state-of-the-art on dynamic, in-situ node reprogramming and adaptation. We argue that complete node reprogramming should be avoided unless necessary for introducing new functionality that was not provisioned in the initial deployment. The arguments against node reprogramming are the high cost of binary transmission and the disruption of operations due to the need to reboot sensor nodes.

Conversely, we look into behavioural adaptation through the mechanism of policies, a lightweight behaviour definition language. We review systems from the literature that make use of policies to support adaptation. The described approach only exposes to the policy mechanism some aspects of the system. On the contrary, we propose a holistic solution, where the system is controlled completely by policies that become the service specification mechanism in the environment. This advantage of this approach is a fully customisable and adaptable environment that can self-configure its components.

Finally, we reviewed mechanisms that detect network faults in WSN or sustain faulty sensor readings in collaborative environments, such as information fusion techniques. Collaborative approaches reviewed use heuristics that are based on expert knowledge on the environments they operate. While heuristics provide lightweight approaches for processing inside the network, we attempt to introduce machine learning mechanisms that operate inside the network that can more effectively discriminate between faults and unexpected events in the monitored environment.

In the next chapter, we present an overview of how such mechanisms are tied together to support self-healing pervasive applications that rely on sensor nodes for receiving input from their operational environment. We define our self-healing framework and what are the challenges that we tackle throughout this thesis.

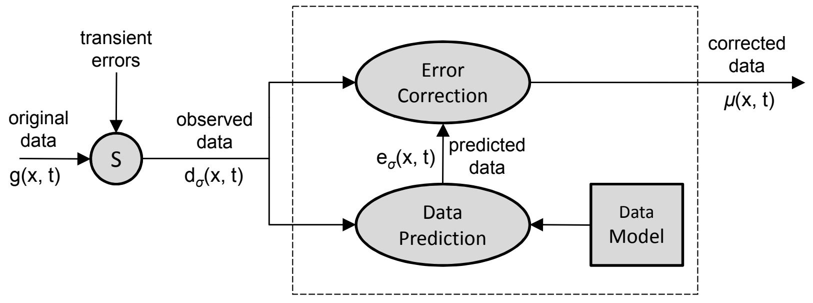
Chapter 3

Self-Healing Framework

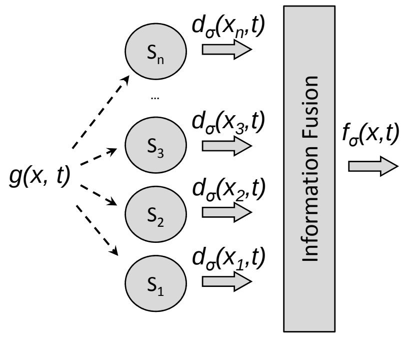
In this chapter, we provide an overview of a self-healing framework for sensor readings. We, further, discuss the communication implications of a distributed fault detection and handling approach. Finally, we present previous work and background on the Self-Managed Cell (SMC) architecture. We describe its fundamental components, their functional role in a system and their interactions. We, also, describe the extension services on the original architecture that are required in a self-healing framework for faults that appear in pervasive computing systems.

3.1 Sensor Readings Fault Model

We formalise the concept of a self-healing framework for sensor reading faults and its requirements in order to understand the faults that need to be detected and how the system needs to adapt to the faults. Values received from sensing devices are subject to transient or permanent faults due to environmental factors such as interference that distorts the observed attribute, electronic fouling of a sensor's circuitry, physical damage or deterioration of quality due to low energy levels of nodes. Consequently, the sensor's observation, $d_\sigma(x,t)$, at the time instance t of an attribute modelled as a random variable X, will be the *ground truth*, $g(x,t)$, plus a random *error factor*, $\sigma(x,t)$, as illustrated in equation 3.1.



(a) Model based error correction



(b) Information fusion process

Figure 3.1: Fault handling models

$$d_\sigma(x,t) = g(x,t) + \sigma(x,t) \tag{3.1}$$

The goal of a self-healing framework is to minimise the *error factor*, $\sigma(x,t)$, in order to approximate the ground truth as accurately as possible. Hence, the reading from a sensor, where the ground truth value is $g(x,t)$ at time instance t, becomes the approximation function $\mu(x,t)$, dependent on variables x and t.

The error factor is affected by both *transient* and *permanent* faults on sensors. A *transient* error is a random deviation from reality that does not deteriorate the state of the sensor. Instead, it only affects the input temporarily. A *permanent* error is an error that the sensor cannot recover from and has an effect on all subsequent readings, unless corrective measures are taken.

A typical approach in the literature to reduce transient errors, is *model-based* correction [MPD04a], where the expected behaviour of observed attributes in the environment is formally represented as a mathematical, probabilistic or heuristic, rule-based models. This approach provides an estimate of the input values based on *a-priori* knowledge of the observed subject by correlating readings in time, which constitutes a prediction model.

Figure 3.1(a) illustrates this approach, where transient errors in a sensor distort the ground truth. The observed value, $d_\sigma(x,t)$, is subject to errors. The estimated value of the system, $\mu(x,t))$ is a composition of the observed and expected, $e_\sigma(x,t)$, values. Assuming a linear combination of the two values the *error correction* unit would produce and update parameters

α and β in equation 3.2. The expected value is generated by the *data prediction* unit that uses the attribute model with recently observed values to produce its estimation.

$$\mu(x,t) = \alpha d_\sigma(x,t) + \beta e_\sigma(x,t) \tag{3.2}$$

In cases where multiple sensors are available, cooperation in a neighbourhood can be employed to reduce errors. The process, known as *information fusion* [HM04, Yan06], is illustrated in figure 3.1(b). Observations from a group of sensors are aggregated in the fusion point, where they are combined to produce the estimated value. The fusion function depends on the type of sensors involved – *homogeneous* sensors that monitor the same attribute, or *heterogenous* sensors that monitor different but correlated attributes.

In homogeneous groups the fusion function, $f_\sigma([x_i/i \in S], t)$, is a combination formula among participating nodes in set S. For instance, majority voting for binary random variables or an averaging formula for continuous random variables. Such schemes can be enhanced with weighted alternatives, where weights represent belief in the sensor's quality or degree of relevance for the considered attribute [OAVRH06]. Homogeneous fusion is a case of *explicit* redundancy, where readings from defective nodes can be adequately replaced with readings from remaining nodes. In heterogeneous fusion there is instead an *implicit* redundancy of sensing devices, where different types of correlated sensors monitor different attributes of the same phenomenon. Loss of a sensor cannot be entirely compensated. Instead, remaining sensors produce a rough estimation of missing values and provide a potentially degraded but operational service.

Finally, permanent, non terminal (*fail-stop*) errors may manifest over time in sensors and affect their accuracy. Such faults are commonly referred to as *drift*, i.e. deviation from ground truth, and are cumulative – the error increases over time amplifying the effect of previous errors. Collaborative on-line recalibration algorithms [BMEP03b] have been studied that use co-located nodes to construct a correction function, $\Delta_\sigma(x,t)$.

By combining different fault correction components – model-based, collaborative and drift correction – the estimation function $\mu(x,t)$ in equation 3.2 can be extended into equation 3.3.

$$\mu(x,t) = \alpha d_\sigma(x,t) + \beta e_\sigma(x,t) + \gamma f_\sigma([x_i/i \in S], t) + \delta \Delta_\sigma(x,t) \tag{3.3}$$

Parameters α, β, γ and δ are the weights for each component that represent the relative impact each observation estimation function has on the final result. A fault handling framework should allow deployment of such functions on the network nodes when necessary to maintain the quality of the monitoring service within acceptable levels. In set-ups where energy conservation is important, deployment of these functions is dictated by indications of fault manifestation. Hence, fault detection mechanisms for accurate identification of sensors' state are the first step for a self-healing pervasive systems. It should be noted that in our study we make the assumption of linear drift functions in order to simplify the modelling and our analysis. This assumption may not always be correct as drift may be exponential rather than linear in some cases, however this does not have a major impact on the framework. The correction function $\Delta_\sigma(x,t)$ would have to be modified to cater for the specific type of drift, but the models we present in the thesis are still relevant.

3.2 Distributed Fault Detection

Centralised solutions for fault monitoring and handling, where performance metrics and signals are collected outside the network for analysis in a resource unrestricted environment, do not scale as the network size and complexity increase. This is mainly caused by increased communication cost due to relaying messages from remote locations of the network. Distributed solutions become more attractive in large-scale networks. However, even in a distributed environment there are alternatives for building the network structure and managing communications. Implications in communication and effectiveness of metrics collection for fault detection in sensor readings are discussed in this section.

Local Detectors

Local fault detectors can be applied for checking sensor features to validate readings with respect to a model of the attribute. Models are tied to the application and deployment usually by setting hard thresholds that define erroneous behaviour, e.g. the room temperature cannot exceed 42 °C. Such heuristic thresholds represent an expert's knowledge about the domain. Moreover, without external knowledge it is not feasible to assess whether a threshold has been exceeded due to a sensor malfunction or an unexpected event that renders the irregular value legitimate, e.g. a fire started in the room. Furthermore, fixed thresholds do not cope well with variable-state attributes, i.e. attributes that modify their behaviour over time. Additionally, large number of false positives, that are typically yielded in local detectors, are potentially more critical and expensive than the number of false negatives.

Local fault detectors inherently depend on the accuracy of monitored attributes models, assuming that *a-priori* knowledge exists. If this is not the case, it may instead confuse the system, decreasing overall quality. Moreover, local monitors are unable to handle unexpected behaviour that has not been foreseen when building the model. For instance, a model that restricts acceleration values of a sensor to those that can be achieved by a human, falls short when the user uses a vehicle that will increase sensor values. Nevertheless, local detectors are an initial, inexpensive detection step. They can operate even when a node has additional information from its neighbourhood and are a first indication of fault appearances.

Collaborative Detectors

Collaborative detectors use input from a multitude of signals in the system and associate them to reach conclusions on component state. They overcome limitations of local monitors that cannot use external information for reasoning. Collaborative detectors can be either homogeneous, i.e. sensors monitoring the same attribute, where direct comparison of features may occur, or heterogeneous, i.e. sensors that monitor different attributes that are correlated, where implicit redundancy of information can be used to assess operational correctness.

Nodes can reason about their state, realising which sensors are more likely to be defective in their neighbourhood, e.g. using voting schemes. For instance, sensors can compare their input's variance to identify whether high or low values are the result of normal behaviour or errors in the sensing device caused by noise in the signal. The key assumption in using collaborative detectors is that a phenomenon or event has an area of effect. Consequently, all sensing devices within that area observe its effects. We assume that faults manifesting in nodes are stochastically uncorrelated, thus, defective sensors can be identified as they deviate from the majority. This assumption will be valid for many cases may not be correct in some deployments where sensors from the same manufacturer have identical failure characteristics. The probability of concurrent faults decreases as the number of participating sensors increases.

In heterogeneous collaboration, algorithms try to utilise implicit redundancy of sensors. Implicit redundancy exists in the network when deploying sensors that monitor different attributes of a phenomenon that combined give insight on occurring events in the environment. Such attributes, though not directly comparable as in the case of homogeneous detectors, exhibit a degree of correlation allowing study of their interdependence. Consider an ambient deployment of thermometer and humidity sensors – although these attributes are not straightforwardly comparable, we can determine their dependence by studying their correlation. Similarly, sensors of a 3D-accelerometer, though of the same type, are also not directly comparable, as they monitor three different attributes – acceleration in orthogonal axes. Nevertheless, inputs in all three axes are correlated when the device is worn by a patient.

Collaboration of sensors deployed in different nodes involves network communication and thus raise the issue of information distribution in the network. A completely distributed communication approach requires all nodes in a neighbourhood to exchange information with each other. Every node applies, respectively, the fault detection algorithms locally. The benefit of this approach is that it is decentralised. Lack of central arbitration increases resilience to failures of individual nodes. Moreover, decisions are made at the enforcement points avoiding extra routing of reconfiguration messages from evaluation points to the enforcement points.

The downside of a fully distributed approach is the increased energy consumption, even though it is distributed fairly among participating nodes. Consumption increases as every node must use its radio for listening for incoming messages from neighbours. Active listening for incoming messages consumes significant amounts of energy preventing nodes from entering a low-power sleep mode. Constant radio listening can be prevented by using duty-cycle solutions that wake nodes at predetermined time periods for communication. Nevertheless, this requires synchronisation among nodes. Power consumption deteriorates further in scenarios where the sensing radius of a node is greater than its communication range, requiring multi-hop communication protocols to disseminate information in its sensing neighbourhood. In such cases, the distributed approach does not scale well as the communication cost for a group of N nodes is $O(N^2)$ requiring every member of the group to communicate with all other members.

A hierarchical approach that using local leaders helps to alleviate, to a degree, most of these problems. Nodes are assigned to potentially overlapping clusters that elect a representative leader (cluster-head), which collects feature information from the group members. Clustering allows leaf nodes to enter a sleep mode when not sampling or transmitting data, saving significant amounts of energy. However, it introduces uneven energy consumption in the network at the cluster-heads. Leadership roles can be assigned to more powerful nodes with greater storage and processing capacity. Furthermore, leader outages can be addressed with leader re-election schemes [DHS02]. Structure inside the networks decreases communication expenditure, when the sensing radius is greater than the communication radius. The worst case scenario remains $O(N^2)$, when every node has only one single-hop neighbour and the group forms a minimal connected graph. Nonetheless, the complexity of communication reduces to $O(Nlog_b(N))$ for an even distribution of nodes in the physical space, where the logarithm base, b, is the average connectivity degree of nodes in the group.

3.3 Self-Managed Cell

The Self-Managed Cell (SMC) is a structural, architectural paradigm for autonomic management of pervasive systems [LDS+08]. It is defined as a set of functional components that forms an autonomous management domain that is not divisible, in other words, it is the minimum set of functional roles that can accommodate an autonomous system. SMCs are the building blocks for the construction of large-scale systems by forming federations between them. The platform facilitates addition and removal of components by providing a well-defined interface for interactions with other SMCs. A typical SMC can be considered as a set of sensor and actuator nodes in addition to a smart-phone controller that form a body sensor network monitoring the health of a patient, the devices in an intelligent building, a group of autonomous vehicles collaborating on a search and rescue mission, or the routers and firewalls managed by an Internet Service Provider.

As an architectural pattern, the SMC can be customised and tailored on instantiations applied at different levels of scale. The core SMC design, as seen in figure 3.2, consists of a set of services that includes a *discovery service* for location and authentication of new components and neighbouring SMCs, a *policy service* for specifying adaptive behaviour, an *event bus* supporting publish-subscribe interactions between services and components. These are the three core services of an SMC. In addition a cell maintains a set of *managed objects*. *Managed objects* are, essentially, the components that carry out the tasks of the system. They encapsulate heterogeneous resources such as software and hardware modules or even remote SMCs that the host SMC interacts with as a *managed object* for composition to form complex structures. In order to abstract this complexity from the developer, *managed objects* are connected to the SMC *event bus* via *resource adapters*. The adapters are necessary in the system to hide the heterogeneity of *managed objects* and provide uniform interfaces for interactions that abstract communication details.

The *event bus* component uses the publish/subscribe programming abstraction [EFGK03a] to decouple communication among components. Removing hard dependencies among them simplifies software implementation and maintenance by easily matching notifications with components that have registered interest for them.

The *discovery service* locates nearby SMCs that can interact with the original and collects their profiles. SMC profiles include their credentials and the services, i.e. *managed objects*,

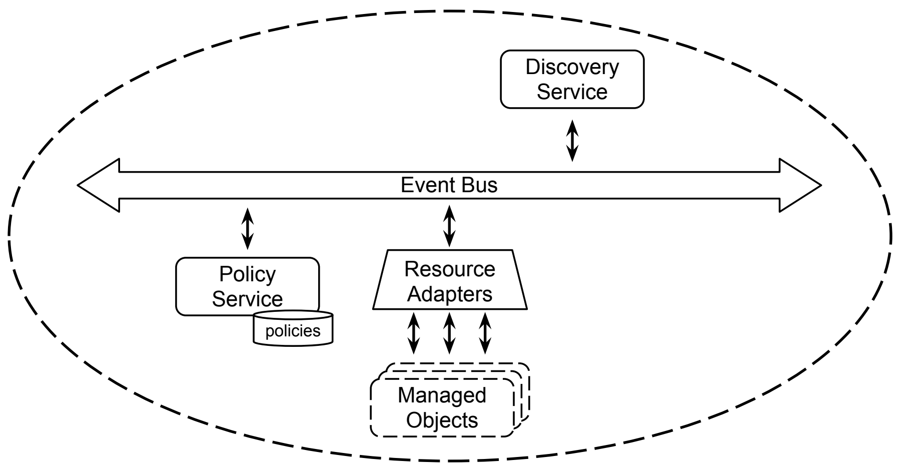


Figure 3.2: The core Self-Managed Cell architecture

they provide. It further carries out the admission control process for negotiating access of remote components by SMCs.

The *policy service* is the control mechanism of the SMC that supports specification of adaptive behaviour in network management. It uses interpreted policies that allow for changes in execution at run-time without disruption of system's operation. There are two types of policies – *obligation* and *authorisation* policies, based on the Ponder2[1] policy management system [DDLS01]. *Obligation policies* are Event-Condition-Action (ECA) rules that express system behaviour in an event-driven model. *Authorisation policies* are the access control mechanism of SMC that define what resources or services can be accessed by remote SMCs. The policy notation is presented in figure 3.3.

def ⟨*authpolicy*⟩ [+/−]	**def** ⟨*obligpolicy*⟩
subject ⟨*role*⟩	**on** ⟨*event*⟩
target ⟨*role*⟩	**if** ⟨*condition*⟩
if ⟨*condition*⟩	**do** ⟨*action*⟩
action ⟨*name*⟩	
(a) authorisation policy	(b) obligation policy

Figure 3.3: Syntax of policies in the SMC environment

Events and actions in the system are provided by *managed objects* through the local *resource adapters* that conceal location and communication details. *Managed objects* may in fact be local or remote with regard to the SMC. Policy conditions are predicate functions of *managed objects*

[1]http://www.ponder2.net

that return a boolean value and control policy execution. Roles in *authorisation* policies refer to semantic labels that have been assigned to SMCs. Role assignment to SMCs is a many-to-many relation, where an SMC may be assigned many roles and one role can be assigned to more than one SMC. SMCs advertise their roles when discovered by a neighbouring *discovery service.* Authentication mechanisms can verify the role assignment in a distributed SMC environment [ZKS+08a]. *Authorisation* policies can *allow* or *reject* access to a remote resource, i.e. a managed object or a subset of its functionality. Interaction *allow* or *reject* is denoted by the [+/−] symbols, respectively, in the policy specification. Each *managed object* is placed under a specific domain path. The domains are also the identifiers that policies can use to access *managed objects*' functionality by referencing their location in the hierarchy. SMC federations are achieved by using architectural patterns, such as collaboration or composition, to form more complex structures to build larger-scale distributed systems [FL07].

3.4 Self-healing Services in SMC

The core SMC architecture provides the fundamental infrastructure for an autonomous environment. However, it does not define specific mechanisms that support the *self-healing* attributes of an autonomous system. In this thesis we extend the SMC architecture with the specification of services that support the *self-healing* characteristic in pervasive WSN environments. We identify the services that are essential in this context and discuss their interactions and integration in the SMC core architecture.

The goal of a *self-healing* system is to adapt in response to faults and errors before users observe deterioration in the provided service or a failure. Essentially, a combination of an adaptive framework with a set of detection and correction algorithms is required. The SMC model provides the underlying adaptive framework of *managed objects* along with a *policy service* that supports flexible behavioural modification. Algorithms and services deployed are controlled in term of policies. We take a top-down view on these services by first identifying their functional role in the architecture by extending the SMC model and later describe them

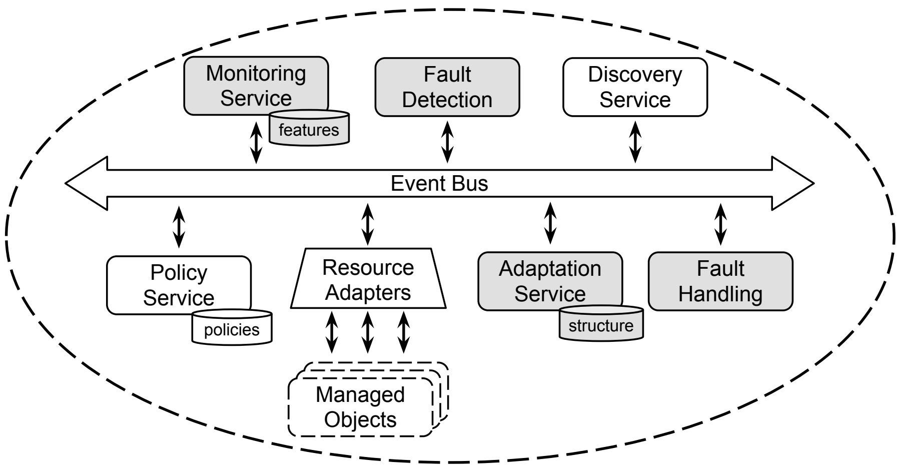


Figure 3.4: Self-Managed Cell architecture with self-healing services

in further details, with case studies and examples of mechanisms that provide a *self-healing* platform.

Figure 3.4 extends the original SMC architecture in figure 3.2 with the services that are required to build a self-healing system. These include a *monitoring service* that collects metrics from the system to detect its status, a *fault detection service* that analyses metrics and compares them with a model of acceptable system operation, a *fault handling service* that produces a plan to transition the system to an improved state and, finally, an *adaptation service* that controls the structure of the network and modifies it to satisfy new operational plans. These services are installed in the SMC as *managed objects*, therefore, they are fully integrated in its publish/subscribe system and expose their interfaces to the policy mechanisms through the SMC domain system.

The *monitoring service* is the initial step of a feedback mechanism necessary for the system to be aware of malfunctioning components. Metrics collected are defined by the system attributes that need to be adapted. In the case of WSNs, they include sensor readings accuracy, link channel drop-rates or computation and storage load. For sensor accuracy, metrics involves features collected from sensors readings. An aggregation mechanism collects features from the SMC's *managed objects* being either local or remote sensing devices. The feature extraction part of the service provides an API for policies to select the set of features and their sampling rate in order to customise monitoring in accordance with the application's needs.

The *fault detection service* operates on features collected by the *monitoring service* to validate input with provided models and identify misbehaving components. The *fault handling service* makes a decision for an appropriate strategy that will most effectively reduce the impact of the identified fault. Such strategies may involve isolating defective components, substituting them with predictions of missing data or attempting to repair them whenever possible.

The *adaptation service* manages the structure of the SMC, its components and interactions with neighbouring SMCs. It assesses reorganisation of assigned operational roles and allocation of resources based on inferences derived by the *fault handling service* on the state of other components. Adaptation actions include activation or deactivation of policies, reassignment of roles to nodes or reconfiguration of the network's communication structure.

The services described compose the closed feedback control-loop in the autonomic computing paradigm. Figure 3.5 illustrates the association of the services with the phases defined in autonomous systems. The *monitoring service* corresponds to the *monitoring* phase of the control-loop collecting information on the system's condition. The *fault detection* service corresponds to the *analysis* step, diagnosing the state of the system and identifying potential counter-actions. The *planning* step in the architecture is performed by the *fault handling* and *adaptation services* that creates a plan derived from the *analysis* phase decisions. The two services look at different aspects; *fault handling service* tries to apply corrective actions on available misbehaving resources to restore their operation, while the *adaptation service* modifies the structure of the network to cope with degraded or lost resources. Finally, the *execution* step is handled by the *policy framework* of the SMC that can apply changes in different components of the network.

Knowledge, which is the epicentre of the control-loop, is encapsulated in the SMC in different forms. One form is policies that define behaviour and drive the adaptation mechanisms that operate on the network. This is definitely a human-centric approach of expressing knowledge about the system where the network administrator can specify system response to exceptional conditions. *Knowledge* can also be encapsulated in the system through models and machine learning mechanisms that define the correct operational attributes of the system. *Knowledge*

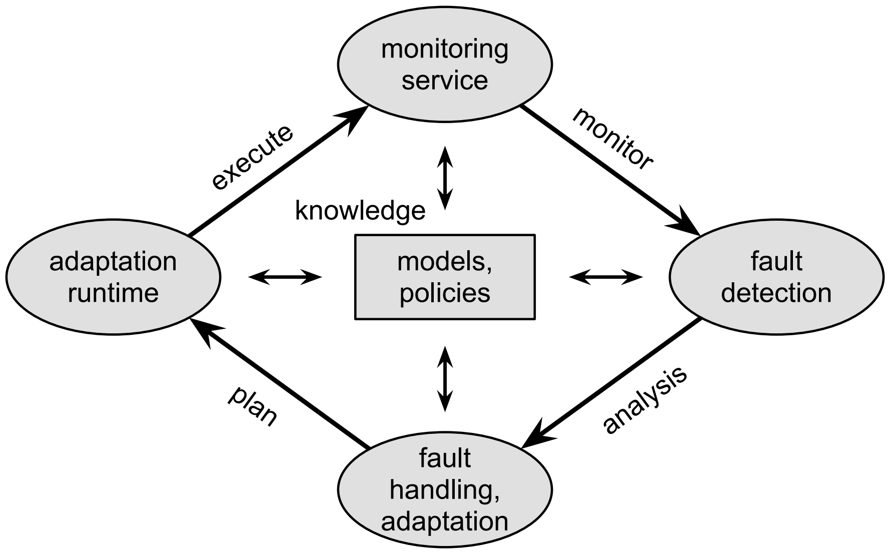


Figure 3.5: Self-healing SMC feedback closed-loop

controls all steps in the feedback control-loop orchestrating components and their operation. Throughout this thesis, we present examples of how *knowledge* is described for fault handling and reconfiguration in pervasive networks.

3.5 Policy Specification

Obligation policies control system behaviour and interactions in the SMC. The *adaptation service* modifies current active policies when adaptation is necessary. Policies are interpreted at run-time, being essentially a means of scripting behaviour that can inexpensively be swapped at run-time. Figure 3.6 demonstrates the expressiveness of *obligation policies* and provides insight to the specification of SMC functionality.

Policy ‘AdaptOnTemperatureDrift’ responds to an event from the *fault detection service* and disables temperature collection on a faulty node. Furthermore, the policy instructs the set-up of a recalibration process on the temperature sensor. Identifiers in italics (*fault*, *type*, *node*) are context variables of the triggered event while ‘drift’ and ‘temperature’ are literals that correspond to an appropriate constant code. Finally, ‘TemperatureCollect’ and ‘RecalibTemp’ literals are identifier in the SMC’s policy repository. Policies only provide the glueing of available functionality on a node while *managed objects*, such as ‘faultService’ and ‘policy’, encapsulate

```
def AdaptOnTemperatureDrift
    on faultService.Detected(fault, type, node)
    if fault is drift and type is temperature
    do node.policy.Disable(TemperatureCollect),
        node.policy.Enable(RecalibTemp)
```

On detection of thermometer's drift disable readings collection and start sensor recalibration instead.

```
def TemperatureCollect
    on timer.Off(id)
    if id is sample_temperature
    do buffers.Append(temperature, sensor.Sample(temperature))
```

In response to a pre-set timer, sample temperature from the sensor and store the reading in a buffer.

Figure 3.6: Policy re-configuration and adaptation example

the implementation in native code. More details on the domain structure of *managed objects* are provided in chapter 4.

Similarly, the second policy 'TemperatureCollect' responds to the notification of a periodic timer and uses the on-board sensor to sample readings from the thermometer and store them in a buffer under the 'temperature' id. Both 'buffers' and 'sensor' are *managed objects* in the SMC. These examples illustrate how policies assemble primitives to construct interactions among components in the system. In later chapters implications on remote interactions are also discussed.

3.6 Summary

In this chapter, we have presented an overview of the work in this thesis. We introduced a fault model for handling sensor readings of both transient and permanent nature. We discussed implications of communication between sensor nodes that cooperate for detecting faults in a local area. Finally, we presented previous work on the Self-Managed Cell architecture for autonomic pervasive system and discussed the self-healing services that are our contributions.

In the following chapter, we present an implementation of the SMC architecture for WSNs and discuss how network adaptation can be expressed in terms of policies. In the remainder of the thesis we look into more detail of the self-healing services in the framework such as monitoring, analysis, recovery and planning.

Chapter 4

The Starfish Framework

The *starfish* platform is an adaptation framework for WSNs that focuses on the self-healing aspects of the network. The framework's fundamental components are *finger2*, an embedded policy management system for sensor nodes; the *Starfish Module Library* (SML), a module library that simplifies programming of motes providing basic functions and tools required in sensing applications; and *starfish editor*, a client side graphical user interface for authoring and control of *policies*, *missions* and *roles* on motes.

Starfish is, essentially, framework to facilitate building *Self-Managed Cells* (SMCs), introduced in section 3.3, in the context of pervasive systems and sensor nodes. It provides the infrastructure for management and adaptation within the network as well as deployment of self-healing strategies. A sensor node in *starfish* resembles an autonomous cell, i.e. the smallest unit for self-management. Groups of nodes collaborate together to form federations that provide composite services. The core SMC components in figure 3.2 have been prototyped for tinyOS 2.x[1], an operating system for embedded sensor nodes. The event-driven programming model of tinyOS fits the SMC *event bus* service, which carries messages from local and remote components. A *discovery service* for detection and incorporation of neighbouring cells has also been prototyped. *Finger2* provides the *policy service* of the SMC for sensor nodes. Finally, the *managed objects* in the node SMC are the *starfish modules* that are self-contained software

[1]http://www.tinyos.net/

components written in native code, closely resembling the nesC language module system. They encapsulate hardware components, i.e. sensors, radio, etc., and implement functions providing interfaces for the policies to orchestrate their operation. An SMC can interact with *modules* that may reside either locally or on remote nodes. Communication with remote modules is handled transparently by the middleware, through *object adapters*.

In this chapter, we focus on autonomic pervasive systems. We discuss the evolution of the original *finger* policy middleware in order to incorporate concepts that have been introduced in previous work on the SMC architecture, such as *missions* and *roles*, in order to define self-healing strategies in terms of the policy-based adaptation that can detect sensor faults or component failures and handle them by applying necessary system reconfiguration.

4.1 Policy Management on Sensor Nodes

Policies provide a high-level abstraction for scripting sensor node behaviour in a network. However, they are not a full fledged programming model. Policies are interpreted at run-time on sensor nodes and are designed to be lightweight and compact to transmit over-the-air. While they allow adaptation to new conditions without requiring nodes to reboot, they are not a platform that supports complete software updates and node reprogramming. Instead, it is a means for controlling behaviour by orchestrating existing software components on nodes.

Finger2 is an embedded policy system for sensor nodes and part of the *starfish* framework. It is based on Ponder 2[2] [TDLS09] policy system. While Ponder 2 scales down to mobile devices, such as smart phones or the gumstix platform[3], it requires a Java runtime environment. However, due to the dependency on the Java platform it cannot scale down to low-end nodes that run the TinyOS 2.x environment.

Consider a scenario of health-care body deployment, illustrated in figure 4.1, where a *nurse* and a *doctor* are treating a *patient* in a hospital. The *patient* is equipped with a wearable sensor

[2]http://www.ponder2.net/
[3]http://www.gumstix.com/

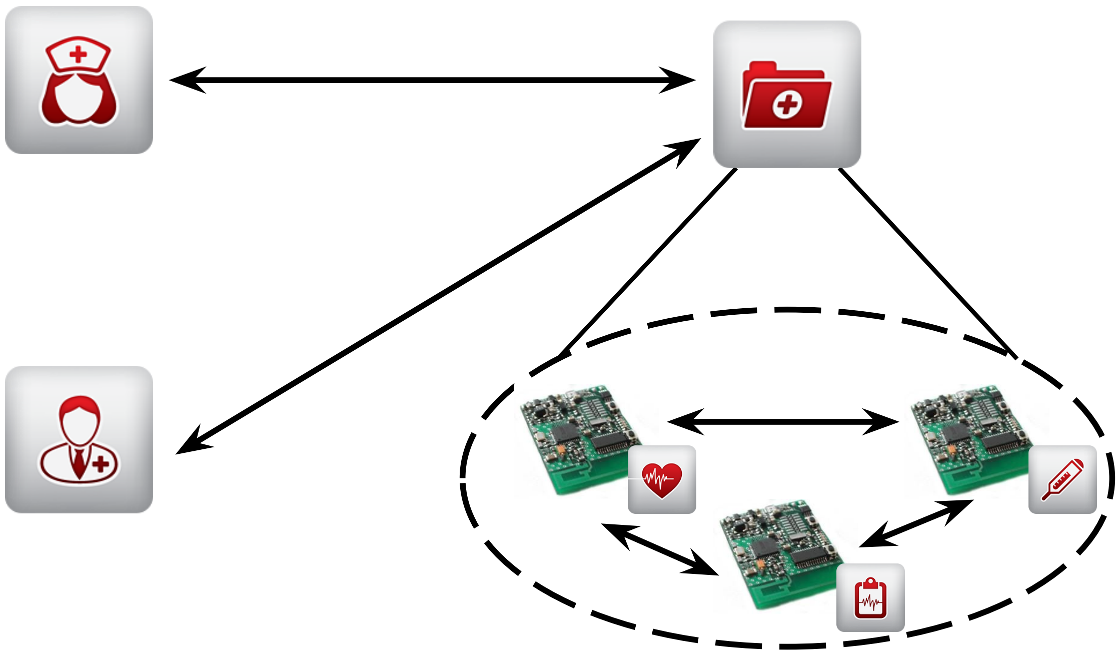


Figure 4.1: Health-care scenario and roles

network consisting of several nodes that sample his vitals and activity for close monitoring of his condition. Hospital personnel can query a *patient*'s history of observations using an application on their personal devices that communicate with the *patient*'s wearable sensor network. The deployed network on the *patient* consists of several components – a *thermometer* for measuring body temperature, an *ECG* monitor and an *activity* recognition system that uses accelerometers. This is, evidently, a heterogeneous pervasive application that involves a range of devices, i.e. constrained body sensor nodes and more powerful smart-phone devices. Each participating role (appearing in italics in the text) is assigned to a device that executes required tasks and may also provide an interface for user interaction. Each device in this example is considered an autonomous component in the system, i.e. an SMC.

Figure 4.1[4] illustrates the functional architecture of the system in terms of participating roles and their interactions in this scenario. Two types of SMC interactions can be observed from this example. A *composition* pattern of three sensor nodes with different roles; *thermometer*, *ECG* and *activity*, which are composed together in a single SMC with the *patient* role. Other SMCs can interact with the *patient* SMC instead of addressing its individual subcomponents.

[4]Icons by http://dryicons.com

Furthermore, there are the bidirectional, *peer-to-peer* collaboration interactions between the *nurse/patient* and *doctor/patient* roles, e.g. a *nurse* querying a *patient*'s condition or an alert from the *patient* to the *nurse*.

Even though the policy model does not define how the low-level processes work, e.g. sensor sampling or radio communication, it does specify application behavioural logic by manipulating native components. Figure 4.2 demonstrates interactions between components and nodes expressed in policies. The first policy *ECG_request* runs on the *nurse* SMC handling the update request on *patient*'s ECG that is received from the device's GUI. The request results in the installation of a new policy, *ECG_update*, at the *patient*'s SMC.

```
def ECG_request
    on gui.RequestUpdate(patient, type)
    if type is ECG and network.IsAvail(patient)
    do patient.policy.Install(ECG_update)

On nurse's request for an ECG update, install
'ECG_update' on patient's endpoint.

def allow_nurse_policy_install+
    subject nurse
    target patient
    if nurse.type is staff_nurse and power.Level() > 20
    action policy.Install

Authorise access of 'policy.Install()' to a staff nurse
given that the local power level is above 20%.

def ECG_update
    on ecg.event.onEvt(type, value)
    if type is onECGReading
    do nurse.gui.Update(type, value, timer.Now())

On receiving of an ECG reading, update nurse's
GUI module with the reading and a local timestamp.
```

Figure 4.2: Health-care scenario in policies

The policy language abstracts communication details of involved network components enabling the policy author to expresses communication as interactions between roles. The *discovery service* and the *finger2* runtime handles the low-level details and implementation of communication among physical nodes using the operating system's networking primitives. In order for the *nurse* SMC to perform a remote action on the *patient* SMC, in this case to install a new policy, an authorisation is required from the *patient*'s part. Access control of remote interactions is handled by *authorisation policies* in the system. The corresponding *authorisation policy* for installation of new policies is *allow_nurse_policy_install* in figure 4.2. It permits invocation of the 'Install' action of module 'policy' on the *patient* SMC by the *nurse* SMC, if the 'type' attribute of the nurse is 'staff_nurse' and the local power level of the node is greater than 20%. The *authorisation policy* resides in and is checked by the *patient* SMC. All remote interactions in *starfish* are subject to *authorisation policy* checks. An authentication mechanism for association of SMCs to roles is beyond the scope of this thesis, but it has been studied in [ZSLK09].

According to the scenario in figure 4.1, the *patient* SMC is a composition of three SMCs; *ecg*, *temperature* and *activity*. Each role in this case corresponds to a physical sensor node. However, multiple roles can be assigned to a single SMC/node. Essentially, one of the three devices is also assigned the role of the representative of the composite SMC, for interactions with others, i.e. the *nurse* and the *doctor*. Specifics on the assignment mechanism of roles to physical devices are described in more detail in chapter 6.

The *ECG_update* policy on *patient*'s SMC demonstrates a second type of role interaction between *patient* and *ecg roles*. Instead of a remote action invocation, this is a remote event notification. The *patient* role responds to event notifications originating from SMCs with the *ecg* role. The two roles may not reside on the same physical device, but communication details are again abstracted by the *starfish* runtime environment. The *event* module is a special module that handles exactly these remote event interactions, and is described in more detail in section 4.2.2. Finally, the *nurse* SMC is updated with the ECG reading along with an attached local time-stamp.

Through the description of the health-care scenario some aspects of the *starfish* platform have been introduced without much detail. In the remainder of this section, we look into these concepts and abstractions of the framework in more detail.

4.1.1 Modules

Modules are the low-level processes of sensor nodes that are implemented in native code and encapsulate hardware and software functionality on the platform. Modules follow a component oriented design in that they hide implementation details and state and provide clean interfaces for interaction with other components. Modules support three classes of interfaces – *EventI*, *PredicateI* and *ActionI*. The correspondence of these interfaces is straightforward with the three parts of *obligation policies* – event, condition, action. The event part of an *obligation policy* contains a module function that provides the *EventI* interfaces. An event also carries context, which is represented as the context variables in the *EventI* interface. The condition part is a composition of simple predicate functions provided by modules. The *PredicateI* interface defines a function that returns a boolean value. Finally, the action part of the policy is a chain of calls to *starfish* module functions that export the *ActionI* interface. Both actions and predicates calls take arguments that can be constants, event context variables or return values of predicate/action functions calls.

Events, predicates and actions are referenced using a domain hierarchy. Domains are also used in Ponder 2 for referencing *managed objects*, i.e. modules in the *starfish* context. In Ponder 2 domains are filesystem-like directory tree structures. Given that *starfish* targets more constrained devices, we simplify the domain structure that references *managed objects*. A domain is a dot '.' separated string of identifiers with a length of two or three. Figure 4.3 gives the formal grammar of *starfish* domain in EBNF form.

$$\begin{aligned}
\textit{managed\_object} &::= \textit{role module function} \\
\textit{role} &::= \textit{variable}\ \text{DOT} \mid \text{ID DOT} \mid \epsilon \\
\textit{module} &::= \text{ID DOT} \\
\textit{function} &::= \text{ID} \\
\textit{variable} &::= \text{ID}
\end{aligned}$$

Figure 4.3: EBNF grammar of module domains

The grammar is straightforward, functions are referenced by the module to which they belong followed by the function's identifier, while the associated role is an optional prefix. If the role prefix is omitted a local module call is assumed, otherwise the call is propagated to the nodes in the SMC's proximity that are assigned the specified role. The role part in the domain can either be referenced directly by its unique identifier or by a variable, as in the case of *ECG_request* policy in figure 4.2, where the role identifier is contained in the '*patient*' context variable.

Remote interactions between SMCs are subject to increased latency due to unreliability of the wireless medium. Consequently, synchronous remote procedure calls can be prolonged. In order to avoid stalling of *obligation policy* execution, and thus all node operations, remote interactions are prohibited in cases where a call is required to return a value that is part of the policy execution, i.e. inside nested function calls. Essentially, remote interactions are not allowed in the predicate part of *obligation policies* or function calls that return values used as actual arguments to another call in the action part of *obligation policies.* Even though this design decision appears restrictive, it greatly simplifies the design and performance issues of the embedded policy system without severely crippling the expressiveness of policies. Synchronous calls on remote SMCs can straightforwardly be transformed to asynchronous calls, where the return value can be implicitly returned by a remote event as a context variable, which can be used on subsequent calls in the handling policy.

4.1.2 Missions

Missions are sets of associated policies that combined accomplish a specific task. For instance, the transmission of the average temperature from a node to a cluster-head can be described as a combination of *obligation policies*, such as periodic polling of the on-board thermometer, calculation of the average temperature value over a sampling window and finally, transmission of the value to the cluster-head using the radio.

The scope of a single policy is usually very fine-grained, specifying only a small part of task. Consequently, as applications become more sophisticated the complexity of managing single policies escalates making the administrator's task increasingly onerous. Missions are a tool that assist organisation and grouping of policies [LDS$^+$08].

4.1.3 Roles

Roles, such as *nurse*, *patient* and *ecg*, have been extensively used during the health-care scenario example. A role is an abstraction of the *starfish* framework that simplifies interaction definition among participating SMCs. Communication between devices over an unreliable wireless medium is challenging to implement. As a result, we attempt to separate concerns of reliable communication and service/application logic. Low-level module operations can handle delivery of messages while the service developer can define communication in terms of functional components in the network, i.e. roles, during development. The operations of a role are defined by missions that are associated with it. Missions and roles have a many-to-many relation; a mission may be associated to many roles and a role may contain several missions. SMCs are assigned roles at deployment time either manually, by the network administrator, or autonomously after policy and requirement analysis. The autonomic process is described in detail in chapter 6.

4.1.4 Configurations

Finally, a configuration is the initial set of roles, missions and modules loaded on a node. For instance a node may be assigned the roles of *data collector* and *cluster-head*. Assignment of these roles implies that necessary missions and modules must be included in its configuration as well. In addition, a configuration may include modules that are not required by currently assigned roles. Inclusion of additional modules is allowed to accommodate for future roles that an SMC may be required to play, in order to adapt behaviour. Roles, missions and policies can be loaded dynamically on nodes once they are deployed without disrupting their operation. However, modules, which are essentially nesC[5] code, cannot be modified after the initial configuration, due to the limitations of the operating system (tinyOS) that does not support binary images reprogramming on-the-fly. It would be feasible to extend *starfish* to support such operation, should it be deployed in other platforms that support code-distribution and in-situ reprogramming [DGV04].

[5]http://nescc.sourceforge.net/

4.2 Starfish Architecture & Implementation

Having introduced the concepts used in the *starfish* framework, we consider the system's architecture, its abstractions, components, tools and prototype's implementation.

4.2.1 Finger2 Architecture

Figure 4.4 illustrates the high level architecture of *finger2*. Central component in the architecture is the *obligation manager* that processes obligation policies on nodes and is the driving execution and adaptation mechanism of the system. It consists of two components, the *event manager* and an embedded *virtual machine*. The *event manager*, as the name implies, processes incoming events of the system. Event notifications may either be internal from the node, originating from modules such as sensors or timers, or they can be external to the node, from neighbouring SMCs over the network. The *event manager* looks up policies that are associated with the event from the local node *policy repository*. Collected policies are dispatched to the embedded *virtual machine* for execution. Policies are executed sequentially by the VM without any execution order guarantee. Initially the condition part of the node is executed and if its predicates are satisfied, execution of the action part takes place.

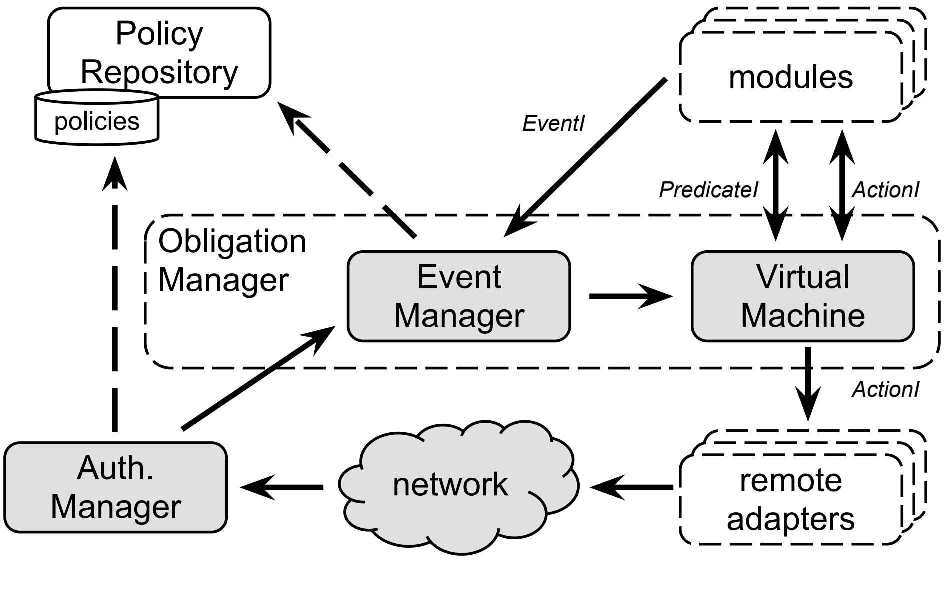


Figure 4.4: finger2 architecture

The *virtual machine* calls module functions from node's local directory. The VM is implemented using the function dispatcher mechanism [GLC07] that is provided in the nesC language making it very efficient at runtime. All VM calls to modules are synchronous. In other words the VM blocks until a module function finishes its execution. Thus, modules should typically have short execution times. Long running and computation intensive operation can be scheduled with the 'task' mechanism of nesC, which addresses this exact issue. Local *module adapters* are used by the VM for remote module invocations to collaborating SMCs, which propagate invocation to the corresponding device. A mapping of roles to node devices is maintained and updated by the node's *discovery service*. Remote invocations are asynchronous calls that do not block the VM execution.

External event notifications or remote module invocations that arrive at the node from the network are first received by the *authorisation manager* to enforce access control policies. The *authorisation manager* looks up the *policy repository* for a corresponding *authorisation policy* that would allow the local node to respond to a remote event or execute a request from the remote party. If authorisation succeeds the event or request is propagated to the *obligation manager* for execution. Authorisation logic is based on the assigned roles of the source SMC. Again, the *discovery service* catalogue of remote SMCs' roles is consulted by the *authorisation manager*. Means for authentication of claimed roles and SMC identity in an embedded environment are beyond the scope of this thesis, but have been studied by Zhu et al. [ZSLK09].

Solid black arrows in figure 4.4 represent the execution flow that has two starting and two ending points; local modules or remote SMCs over the network in both cases. Bidirectional arrows between the *virtual machine* and local modules illustrate that both predicates and actions can return values to the *virtual machine* to be used in further execution. On the other hand, only action execution is allowed on remote SMCs, through adapters, while the unidirectional arrow refers to the asynchronous nature of the communication, i.e. the call does not directly return a value. Finally, dashed arrows indicates data access, i.e. retrieval of policies from the *policy repository* by the *authentication* and *obligation managers*.

Evolution from Finger

Finger2 has evolved from the original Finger policy system [ZKS+08b] developed for tinyOS 1.1.x for the Imperial College BSN node[6]. Apart from porting to the newer version of the tinyOS platform, *finger2* is a major revamp to the original system that significantly extends policy expressiveness and introduces new functionality. Below we briefly list the improvements over the original version.

- Introduction of missions and roles abstractions that support management of policies and remote interactions between SMCs.
- Introduction of an extensible framework of composable modules that encapsulate functionality and promote re-usability of common processes in sensor networks.
- A discovery services that maintains a presence list of neighbouring SMCs and their associated roles.
- An embedded virtual machine (VM) that allows execution of more complex and expressive policies.
- Obligation policy events support more than one context variable.
- Policy conditions support more complex expressions instead of limited primitive boolean predicates on the context variable.
- Actions allow for multiple module function calls as well as nested function calls.
- Enumerations and constants can be defined and used in the policies in order to improve their readability, hence, their maintenance and updating.
- Policy are now stored in binary format for more efficient transmission and processing instead of the original string representation.

[6]http://vip.doc.ic.ac.uk/bsn/index.php?article=167

4.2.2 Starfish Module Library

The Starfish Module Library (SML) for *finger2* is a collection of *starfish* modules that support the most commonly used functions of WSN applications. These include sensor sampling, feature extraction, buffering, timers for scheduling of events and network primitives for exchange of messages among nodes. In this section, we briefly discuss some modules of interest, the extension infrastructure of the module system and how modules are implemented in tinyOS native code.

4.2.3 Fundamental Modules

The *sensor* module encapsulates node hardware sensors abstracting details for the policy author. It provides interfaces for both periodic sampling, *sensor.Sense()*, and immediate sampling, *sensor.Get()*. Modules usually provide asynchronous interfaces for tasks that require an arbitrary amount of time for completion. For example, periodic sampling of sensors are returned through a *sensor.Reading()* event emitted by the module when a new reading is available. The context variables of the event are the *type* of the sampled sensor (e.g. temperature, humidity, acceleration, etc.) and the observation *value*. Context variables can be checked and passed as actual arguments in the condition and action parts of the *obligation policy*.

Buffer is an auxiliary module that provides storage facilities on sensor nodes. Policy *Initialize* in figure 4.5 allocates a new array of 50 elements for storage of the temperature readings using *buffer.Alloc()* action. The *buffer* module emits an event when one of its allocated datastores is full, which triggers policy *SendMeanTemp*. Other modules in SML can also operate on buffers, for instance, the *features* module. Function *features.Avg()* takes a buffer identifier as an argument and calculates the mean value of its elements. Feature extraction functions include operations such as average, median, variance etc. The composition of *buffer* and *features* modules provides a feature extraction platform inside the network for preprocessing sensor readings before transmission. Moreover, it allows to dynamically modify the set of extracted features by modifying the policies acting on nodes.

```
  def Initialize
    on boot.Done()
    if sensor.IsEquipped(tempt)
    do buffer.Alloc(tempt, 50), sensor.Sense(tempt, 250)
```

On boot-up allocate a buffer of size 50 for readings and set the temperature sensor to sample at 4Hz frequency (250ms period).

```
  def StoreReadings
    on sensor.Reading(type, value)
    if buffer.Exists(type)
    do buffer.PushBack(type, value)
```

On receiving a sensor reading store it in the appropriately allocated buffer.

```
  def SendMeanTempt
    on buffer.Full(type)
    if type is tempt and network.IsAvail(temptGtor)
    do network.Send(temptGtor, temptUpdate, features.Avg(buffer.Get(type)))
```

When the buffer is full, send the average temperature over the network to node 'temptGtor'.

Figure 4.5: Sample policies that demonstrate starfish module library

Arithmetic (add, sub, mul, div), association (is, equal, not equal, less, greater, less or equal, greater or equal) and logical (and, or, not) operators included in policies similar to that in figure 4.5 are implemented as module predicates. Modules *arith*, *assoc* and *logic*, respectively, provide these fundamental operations for usage in conditional clauses of policies. Even though syntactic sugar is used for such operators, in order to improve policy readability, these operators are translated to module predicates and actions that are no different from the other system modules. In essence an expression such as '$x < 5$' or 'x **and** y' will be transformed into $assoc.less(x, 5)$ and $logic.and(x, y)$ respectively and executed by the VM.

Timers on embedded systems are used for scheduling activities, e.g. collection of data, periodic heart-beat messages with neighbours, duty-cycling, etc. Module *timer* provides an interface for scheduling event emissions for policies to perform future or periodic tasks. *timer.Periodic()* and *timer.OneShot()* are the two actions that can schedule an event in the future. The *timer* module allows allocation of several timers that are distinguished by an id, similar to the *buffer* module. When a timer is fired, its id is included as a context variable of the event *timer.Off()*.

The *network* module provides communication primitives for policies and other modules for message exchange among nodes. Action *network.Send()* transmits a message to another node. The first two arguments, as seen in policy *SendAvgTemp*, are the destination node id and a message type id, followed by a variable number of arguments that are inserted in the message as payload. Module *network* only supports direct link communication. Multi-hop routing could be implemented as an extension module using one of the approaches found in the literature. The *network* module also includes action *network.BCast()* that broadcasts a message and a predicate *network.IsAvail()* that checks neighbouring node presence. Incoming messages trigger the *network.Recv()* event with a variable length signature, analogous to *network.Send()*.

Additionally, module *serial* provides *Send()* and *Recv()* functions, similar to *network*, for communication with the serial port of the node. It is a tool for communication between nodes and a terminal client. Serial communication is used for application logging and debugging purposes or controlling the sensor network from a desktop terminal.

The *event* module is a special component in *starfish* that provides the means for propagation of events to remote SMCs. It provides an action *event.Emit()* with a variable number of arguments, where the first one is the event id, which identifies the event, and the rest are event context variables. The result of the action is emission of an event notification to the local *event bus* in addition to a radio broadcast to nearby SMCs. Neighbouring SMCs that receive the broadcast replay the event in their local *event buses* if the event passes the authorisation stage. The authorisation is granted depending on roles that are associated with the originating SMCs. A receiver SMC has a directory of the transmitter node's roles in its local *discovery service*. A corresponding event is provided *event.onEvt()* that is to be used as a trigger event in the receiving SMC for corresponding *obligation policies*.

Policy Management

Network adaptation is supported by a set of *starfish* library modules – *policy*, *mission* and *role*, which manipulate the local *policy repository* and *discovery service* to enable/disable or dynamically load and remove elements. Actions *Enable()/Disable()* operate on element IDs and are the most basic means of adaptation. *Install()* uploads and enables a new element on a remote node. In order to install an element to a remote SMC, e.g. a policy, it must reside in the local repository of the initiating node in order to be transferred to the target node. *Remove()* deletes an element from the local repository, as it may be desirable to free up space from unused elements due to memory constraints. Consequently, adaptation is controlled either by a node internally, enabling a different policy profile in response to the occurrence of an event, or by new behaviour installation from remote SMCs that have the authorisation access to perform such a task.

Module Extensions

The module library can be extended by the addition of new modules required in specific domains or for implementation of specialised algorithms, e.g. multi-hop routing. The design facilitates easy integration of new modules through three simple interfaces, *EventI*, *PredicateI* and *ActionI*, that were discussed earlier. Module functions that provide such interfaces can be integrated with *obligation policies* to configure node behaviour and *authorisation policies* to define access control for remote SMCs.

The *EventI* interface defines a single nesC event, i.e. a function, named *evt()*. A low-level nesC module, which implements the operations of a *starfish* module, must provide an *EventI* interface for each event it offers. Events in nesC are an implementation of the signals and slots mechanism, a specialisation of the observer design pattern [GHJV95]. They map language events to function callbacks that are invoked at a later stage with corresponding context, i.e. context variables. The context variables are mapped to the event variables in the *obligation policy*.

Similarly, for predicates and actions a nesC module provides the *PredicateI* and *ActionI* interfaces that define two nesC commands; *evaluate()* and *perform()* respectively. The function's arguments are those provided by the policy author in an obligation policy. A *starfish* module is modelled very closely to a nesC component structure being, in essence, a composite nesC module, referred in the nesC terminology as a 'configuration'. Figure 4.6(a) presents 'policyP', the base implementation of a *policy* module in *starfish*. The figure shows that events, predicates and actions are exported, *provides* indicates a nesC interface of the corresponding type. ‘PolicyMgmtI’ interface used by the ‘policyP’ module is the interface provided by the *policy repository* component to allow its manipulation by others in the system.

While provided *EventI*, *PredicateI* and *ActionI* interfaces can be recognised by the middleware and hooked in *finger2* runtime, there are dangling interfaces, e.g. ‘PolicyMgmtI’, that are not wired to a source. Therefore, the construction of a *starfish* module requires one more step, the wiring of a module’s internal interfaces. A composite nesC module ‘policyC’, aka a ‘configuration’, is shown in figure 4.6(b). The purpose of a ‘configuration’ is to define the internal, implementation-specific interface connections of native modules in order to be complete and executable by the *finger2* VM. It is also necessary, as shown, to forward all *starfish*

```
// policyP.nc file
module policyP {
  provides {
    interface EventI as Installed;
    interface EventI as Removed;
    interface ActionI as Install;
    interface ActionI as Remove;
    interface PredicateI as IsInstalled;
  }

  uses {
    interface PolicyMgmtI;
  }
}

implementation {
  [...]
}
```

(a) policyP nesC module

```
// policyC.nc file
configuration policyC {
  provides {
    interface EventI as Installed;
    interface EventI as Removed;
    interface ActionI as Install;
    interface ActionI as Remove;
    interface PredicateI as IsInstalled;
  }
}

implementation {
  components PolicyRepoP, policyP;

  // internal wiring of the ‘policy’ module
  policyP.PolicyMgmtI –> PolicyRepoP;

  // forward policyP interfaces
  Installed = policyP.Installed;
  Removed = policyP.Removed;
  Install = policyP.Install;
  Remove = policyP.Remove;
  IsInstalled = policyP.IsInstalled;
}
```

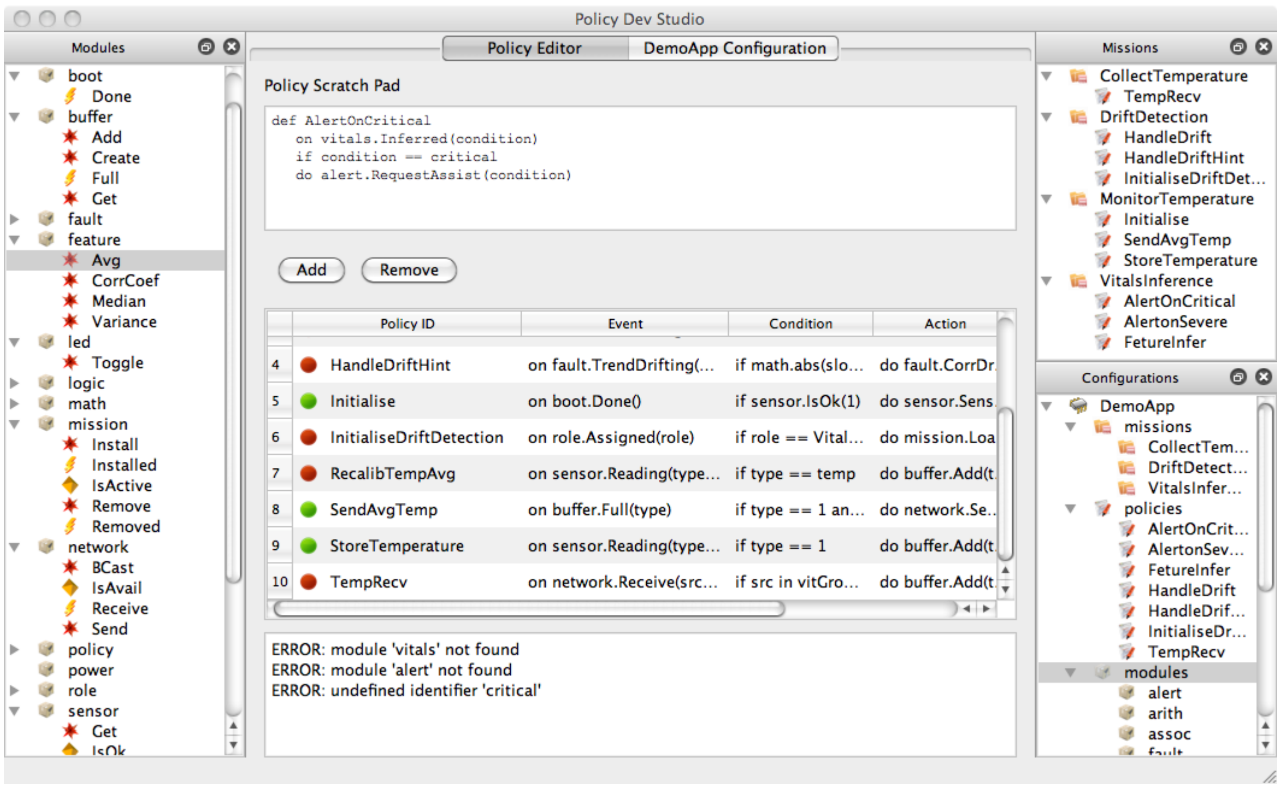
(b) policyC nesC configuration

Figure 4.6: Example definition of module ‘policy’

interfaces that are provided by the core module (‘policyP’) for the VM. While this involves repetitive, boilerplate code, we provide tools that simplify this process leaving only internal logic implementation to the developer.

4.2.4 Starfish Policy Editor

A complementary component to the *starfish* framework is an integrated environment that supports authoring, management and deployment of policies, missions and roles for sensor nodes. Moreover, the environment supports authoring of *starfish* modules and node configurations. The *starfish editor* includes a policy compiler that checks policy syntax and semantics and produces binary code for the embedded *finger2 virtual machine* running on the tinyOS platform. Figure 4.7 shows an snapshot of the Starfish application that showcases the policy editor of the application in the centre and the module, missions and configurations explorers at the sides of the window.



```
def AlertOnCritical
   on vitals.Inferred(condition)
   if condition == critical
   do alert.RequestAssist(condition)
```

		Policy ID	Event	Condition	Action
4	🔴	HandleDriftHint	on fault.TrendDrifting(...	if math.abs(slo...	do fault.CorrDr
5	🟢	Initialise	on boot.Done()	if sensor.IsOk(1)	do sensor.Sens
6	🔴	InitialiseDriftDetection	on role.Assigned(role)	if role == Vital...	do mission.Loa
7	🔴	RecalibTempAvg	on sensor.Reading(type...	if type == temp	do buffer.Add(t
8	🟢	SendAvgTemp	on buffer.Full(type)	if type == 1 an...	do network.Se..
9	🟢	StoreTemperature	on sensor.Reading(type...	if type == 1	do buffer.Add(t
10	🔴	TempRecv	on network.Receive(src...	if src in vitGro...	do buffer.Add(t

```
ERROR: module 'vitals' not found
ERROR: module 'alert' not found
ERROR: undefined identifier 'critical'
```

Figure 4.7: The Starfish policy editor

The *starfish editor* has been developed in the python[7] programming language and is portable to all major desktop platforms – Linux, Mac OS X and Windows. In the rest of this section we examine in more detail the facilities it provides for WSN development.

Policy Editor

The policy editor is a significant component of the application, as policies are the main *starfish* abstraction to support node behavioural specification. The policy editor allows the network administrator to compose *authorisation* and *obligation policies* that are parsed and semantically checked on-the-fly. The policy editor can be seen in the centre of figure 4.7. The authoring area for policies lies on the top of the window. Below it lies a list of already created policies. The policy that is selected from the list appears in the authoring area and the two views are synchronised in real-time when changes are made in either of them. The validity of a policy is indicated by a green or red sign in front of it in the list view, which allows for a quick consistency overview of the existing policies. Policies are subject to both syntax and semantic checks, i.e. whether the modules they refer to exist and are available in the system. Errors of policy compilation are presented to the user in an error log at the bottom of the window.

The policy editor compiles policies in byte-code that is uploaded on sensor nodes for execution in the VM, instead of their textual representation. Figure 4.8 provides a formal representation of the policy language presenting its grammar in EBNF. Non-terminal symbols are presented in lower-case italics while terminal symbols are in upper-case. As mentioned before the policy language operators are syntactic sugar for actions and predicates that are implemented by modules and they are transformed into function calls when compiled to byte-code.

Mission/Role Editor

In addition, to policy authoring support the *starfish editor* provides explorer panes with available policies, missions and roles. GUI facilities for creating new missions and roles are provided. Mission manipulation, i.e. adding/removing policies. Changes in policies or renaming

[7]http://www.python.org

$$
\begin{aligned}
\textit{policy} &::= \text{DEF ID } \textit{obligation}\\
&\mid \text{DEF ID } \textit{auth authorization}\\
\textit{authorization} &::= \textit{subject target condition actionIds}\\
\textit{subject} &::= \text{SUBJECT ID}\\
\textit{target} &::= \text{TARGET ID}\\
\textit{actionIds} &::= \text{ACTION } \textit{idlist}\\
\textit{auth} &::= \text{ALLOW} \mid \text{DISALLOW}\\
\textit{obligation} &::= \textit{event condition action}\\
\textit{event} &::= \text{ON } \textit{notification}\\
\textit{condition} &::= \text{IF } \textit{expr}\\
&\mid \epsilon\\
\textit{action} &::= \text{DO } \textit{fcalllist}\\
\textit{fcalllist} &::= \textit{fcall} \text{ COMMA } \textit{fcalllist}\\
&\mid \textit{fcall}\\
\textit{fcall} &::= \textit{resource} \text{ LEFT\_PARENTHESIS } \textit{exprlist} \text{ RIGHT\_PARENTHESIS}\\
\textit{notification} &::= \textit{resource} \text{ LEFT\_PARENTHESIS } \textit{idlist} \text{ RIGHT\_PARENTHESIS}\\
\textit{resource} &::= \textit{role module function}\\
\textit{role} &::= \textit{variable} \text{ DOT}\\
&\mid \text{ID DOT}\\
&\mid \epsilon\\
\textit{module} &::= \text{ID DOT}\\
\textit{function} &::= \text{ID}\\
\textit{variable} &::= \text{ID}\\
\textit{exprlist} &::= \textit{expr} \text{ COMMA } \textit{exprlist}\\
&\mid \textit{expr}\\
&\mid \epsilon\\
\textit{expr} &::= \textit{expr arith\_op expr}\\
&\mid \textit{expr assoc\_op expr}\\
&\mid \textit{expr logic\_op expr}\\
&\mid \text{NOT } \textit{expr}\\
&\mid \text{MINUS } \textit{expr}\\
&\mid \textit{fcall}\\
&\mid \text{NUMBER}\\
&\mid \text{ID}\\
\textit{idlist} &::= \text{ID COMMA } \textit{arglist}\\
&\mid \text{ID}\\
&\mid \epsilon\\
\textit{arith\_op} &::= \text{PLUS} \mid \text{MINUS} \mid \text{MUL} \mid \text{PLUS}\\
\textit{assoc\_op} &::= \text{IS} \mid \text{EQ} \mid \text{NEQ} \mid \text{LESS} \mid \text{GREATER} \mid \text{LEQ} \mid \text{GEQ}\\
\textit{logic\_op} &::= \text{AND} \mid \text{OR}
\end{aligned}
$$

Figure 4.8: EBNF grammar for finger2 policies

is automatically tracked in order to update dependent missions. Similarly, role creation and association with missions is supported.

Module Editor

Similar to authoring and listing of available policies, the environment supports authoring of modules, which are the components that policies operate on. The editor provides a full list of available modules in the system, which also provides introspection by including provided events, predicates and actions. It compiles a comprehensive list of library utilities that can be used by policies. This list can be seen on the left side of the editor in figure 4.7.

In section 4.2.2, we looked into some fundamental modules in *finger2*. Nevertheless, these modules do not cover all functions required in a sensor network application. WSN developers can define their own *starfish* modules to be used through policies. Section 4.2.2 presented examples on how a *starfish* module can be authored in nesC. However, the process involved a lot of boilerplate code that is cumbersome and error-prone. The *starfish* editor streamlines this process by generating boilerplate code allowing creation of new modules through the GUI of the application. The module authors are left to implement the core logic of the new component in nesC as the wiring of the new component with the *finger2* framework is automated.

Configuration Editor

Finally, in order to assist deployment of modules in nodes, the one component of *starfish* that cannot be transferred over-the-air as an update, the *starfish editor* provides a configuration editor the creates node profiles. These are essentially a set of ROM images to be flashed on node devices. The environment provides a list of available roles and missions that a node can be loaded with, and which of those should be enabled initially. A dependency checker automatically handles inclusion of policies and modules that are associated with those roles by parsing missions and policies and include them in the binary image. Moreover, the interface allows the administrator to load additional modules not required by selected roles in order to provision for dynamic deployment of new missions that may be required in the future.

4.3 Behaviour Adaptation with Policies

The architecture and expressiveness of policies have been discussed so far in detail, however we have not yet demonstrated behavioural adaptation through policies. There are three classes of adaptation supported in *starfish* – local node adaptation of its *objectives*, network group adaptation on *operations* and adaptation driven by new *functional requirements*.

In the first case, *objectives* adaptation, a node may decide that it needs to modify its behaviour, i.e. objectives, in response to an event that prevents operation in the current state, for instance, when a sensing device of the node fails. These kind of faults or events can be considered ahead of time and a node can be scripted with an appropriate alternative behaviour. It can be considered that policies are not necessary for such adaptation as this can be easily hard-coded on the node instead. Nevertheless, we argue that policies allow for an easier mechanism to express this kind of adaptation, which can also be scripted by the network administrator in a high-level language, instead of being hard-coded by the developer.

In the case of *operational* adaptation, the behavioural modification is a consequence of a failure that results in either the elimination or severe degradation of a service component, which impedes the system operation. Alternatively, the configuration of the network may have been degraded to the extent that an alternative configuration becomes more efficient for the operation of the network. In such scenarios, reassignment of functional roles may assist the replacement of a missing component or optimisation of the service, as long as redundant available components in the network exist. A role re-assignment processes can operate autonomously and we look at this mechanism in more depth in chapter 6.

Finally, adaptation on updated *functional requirements* involves modification of nodes' functions to accommodate new processes that were not provisioned during the deployment process. Adaptation on *functional requirements* involves a human administrator, who authors new missions that describe orchestration of existing components. While *starfish* provides the platform (*finger2*) and the primitives (policy, mission and role modules) to allow this form of adaptation, it does not provide a protocol for dissemination of updates over multi-hop sensor networks.

However, there are several approaches in the literature that span from flooding to epidemic protocols [DHM06] to more targeted solutions [WZC06].

In this section, we demonstrate *objectives* adaptation expressed with policies. Finite State Machines (FSMs) are commonly used to express behavioural modification in response to events [UBH09, MSD10]. Their simple model is very convenient to express behaviour with a formal mechanism that allows model analysis. An FSM consists of a finite set of states S, an initial state $s_0 \in S$ and a set of final states $F \subseteq S$. It consumes as input a possibly infinite number of symbols that belong to a finite set Σ called the alphabet. A set of transitions Δ is defined, where a transition $\delta \in \Delta$ is a function $\delta : S \times \Sigma \rightarrow S$, which given a symbol from the input transitions the system from the current state to a new one.

An example of an FSM that describes behavioural adaptation on nodes is shown in figure 4.9. It demonstrates a simplified, yet functional, FSM that handles sensor fault detection and isolation in a network that measures temperature in a room. The FSM provides a high-level view of node behaviour expressed by missions. Each mission is encoded as a state of the FSM. The alphabet

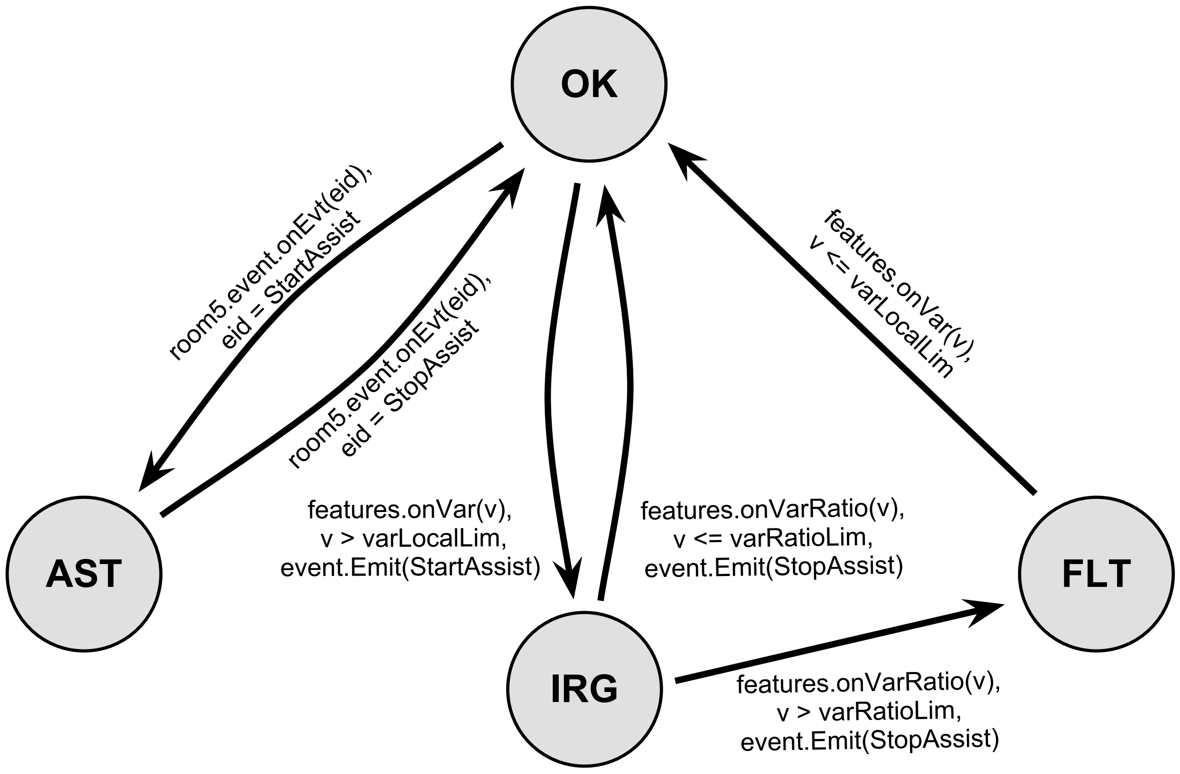


Figure 4.9: FSM mission adaptation on node

for the state transitions are *starfish* events with optional context variable predicates and side effects that are executed during the transition. Missions involved in the FSM are mutually exclusive, although it does not imply that they do not share any common policies/tasks. There are four missions in this example – *healthy* (OK), *irregular* (IRG), *faulty* (FLT) and *assist* (AST), which are abbreviated in the FSM for illustration purposes.

In the *healthy* state nodes execute policies that sample sensor readings periodically and propagate them to a base station, while they also monitor the quality of their readings by extracting the local variance feature. If the local variance monitor exceeds a heuristic threshold, the node transitions to the *irregular* state/mission according to the FSM, which initiates a cooperative monitor on variance that compares local variance with the neighbourhood's average. Initiation is triggered by emission of event 'StartAssist' that is broadcast by the node. If the variance ratio of the sensor to that of its neighbourhood is acceptable, i.e. lower that a predefined threshold, it returns back to the previous *healty* state. Group evidence attributed the increased variance to an unexpected event instead of a isolated fault. On the other hand, if the ratio surpassed the threshold, the node transitions to the *faulty* state/mission. Increased variance over the group is translated as noise, hence, data dissemination to the base station is halted. However, local monitoring of variance still operates and the node returns to a *healthy* state if variance drops low enough, as noise is usually a transient effect.

A node in *healthy* state that receives an a 'StartAssist' event from its group transitions to the *assist* state/mission for participating in collaborative group variance monitoring. It starts broadcasting its local variance feature for neighbouring nodes to calculate variance ratio. Arrival of a termination event 'StopAssist' transitions the node back to its original *healthy* state halting broadcasting of its variance. In the example, the transition to *assist* state is triggered by an event that originates from an SMC with the 'room5' role. All nodes of the group are assigned a common role that allows them to identify their group, in this case an identifier of their placement, 'room5'. If a node overheard the collaboration request from a different group it would not have triggered the transition as its sensor readings would probably be irrelevant for a node group in a different room. Node, i.e. SMCs, may contain several roles that describe different aspects of their operations.

The example FSM of figure 4.9 is in essence a *meta*-mission that controls mission execution on a single node. The FSM can be easily transformed to a set of policies that overlook mission changes on a node. A simple utility has been created that parses FSMs written in a simple custom language and transform them into missions that can be loaded in *finger2*. The mission's policies are shown in figure 4.10 as an example.

```
def healthy2irregular
    on features.onVar($v$)
    if $v$ > varLocalLim and mission.isActive(healthy)
    do mission.Swap(irregular, healthy),
        event.Emit(StartAssist)

def irregular2healthy
    on features.onVarRatio($v$)
    if $v$ <= varRatioLim and mission.isActive(irregular)
    do mission.Swap(healthy, irregular),
        event.Emit(StartAssist)

def irregular2faulty
    on features.onVarRatio($v$)
    if $v$ > varRatioLim and mission.isActive(irregular)
    do mission.Swap(faulty, irregular),
        event.Emit(StartAssist)

def faulty2healthy
    on features.onVar($v$)
    if $v$ <= varLocalLim and mission.isActive(faulty)
    do mission.Swap(healthy, faulty)

def healthy2assist
    on room5.event.onEvt($eid$)
    if $eid$ is StartAssist and mission.isActive(healthy)
    do mission.Swap(assist, healthy)

def assist2healthy
    on room5.event.onEvt($eid$)
    if $eid$ is StopAssist and mission.isActive(assist)
    do mission.Swap(healthy, assist)
```

Figure 4.10: FSM mission adaptation in policies

It evident that an FSM description can be automatically translated into policies that enforce the execution of the FSM logic on the node. The advantages of encoding the FSM as policies instead of being hard-coded in the node include the flexibility to modify the FSM model at runtime without manual reprogramming of nodes. Furthermore, FSM parameters, for instance the constant variance thresholds used in this example, can be modified during run-time if necessary. Finally, it separates node adaptation logic, which is a cross-cutting concern, from actual task execution following the principles of *aspect oriented programming* (AOP) [KLM+97, Lie96].

4.4 Finger2 Overheads

We have prototyped the *finger2* middleware for nodes running tinyOS 2.x in order to provide estimates on performance and memory requirements of the SMC architecture on resource constrained devices. The prototype includes the *policy repository*, *obligation manager*, embedded *virtual machine* and an *authorisation manager* that does not implement any authentication processes. A simplified version of a *discovery service* has also been prototyped that periodically broadcasts node id and roles and collects similar broadcasts from neighbours. Performance and resource requirements give an impression of the impact the middleware has on nodes and applications.

The numbers we present are obtained by compiling the prototype for the Tmote Sky / TelosB platform[8], which are also the nodes used in the motelab test-bet [WASW05]. Binary image and stack size might differ slightly on other platforms, e.g. micaz. The total memory requirements of the *finger2* middleware is 24.38 KB in ROM and 0.72 KB of RAM. These numbers include all mentioned middleware services, but do not include *starfish* modules that may be installed alongside. The RAM requirement does not include storage requirements for policies, as they are application specific. The minimum memory size for a policy stored in binary form is 24 bytes and it increases depending on the number of event context arguments involved and function invocations. From the prototyping experience the average size of policies tends to be around 40 bytes. In order to put these numbers in perspective, typical contemporary sensor nodes range between 48 – 128 KB of ROM and 4 – 16 KB of RAM and the trends are increasing. This indicates that adequate memory space should be left for developers to build applications on top of *finger2*.

[8]http://www.sentilla.com/

For comparison, lets consider the most basic application that comes bundled with the tinyOS 2.x, the *Blink* application. *Blink* when loaded on a node blinks the three LEDs that a node like TelosB is equipped with. It is the equivalent of the famous 'hello, world!' application for TinyOS. Blink occupies 2.59 KB of ROM and 55 bytes of RAM on a TelosB. We deployed an application with the same functionality that is developed on top of *starfish* to provide better insight of the overheads of the platform. Consequently, we prototyped a module that switches the nodes LEDs and installed a policy that listens to a periodic timer event to trigger this module. The size of the *Blink* application in *starfish* is 26.32 KB in ROM and 0.8 KB in RAM. The size of the installed policy that triggers the *Blink* module is 24 bytes.

The overheads described above are significant for a simple application like *Blink*. Nevertheless, *starfish* is not intended for such trivial applications. Another example of a stock application is one that periodically polls the onboard sensor of the node and transmits its readings over the network. When compiled, for the TelosB platform, the application occupies 14.81 KB of ROM and 1.73 KB of RAM. This application, while still simple, better demostrates the benefits of an embedded middleware, such as *starfish*. The application can makes use of already provided *starfish* modules, like the *timer* for periodic events scheduling, the sensor and the *network* to poll readings and transmit them respectively. The application only requires a single policy, though more complex, with a size of 40 bytes. The policy responds to a *timer*'s event and uses *network*'s *Send()* action to transmit a reading from the *sensor* module. The total size of the application written using *starfish* is 36.45 KB in ROM and 1.92 KB in RAM.

With respect to computation requirements and performance, the impact of *finger2* VM appears to be minimal as the average processing time introduced by *obligation policies* is $74\mu s$ while for *authorisation policies* it drops to $53\mu s$. These numbers account for the time required to match an incoming event with an active policy stored in the local repository and invoke associated predicates and actions or check for access control in the case of authorisation. Execution time for the actual evaluation of predicates and execution actions is not included in these numbers, as they are classified as 'user' execution time and depend on the application.

4.5 Summary

We presented the components of *starfish* framework, an platform for autonomic, self-healing pervasive systems that is comprised of three main parts: the *finger2* embedded policy system, which control node behaviour and adaptation in the system; the *starfish* runtime support that is a collection of modules that provide basic sensor network operation as well as facilities for supporting adaptation on the SMC; and finally the *starfish* editor, a client application that supports the authoring and deployment of *starfish* policies, missions, roles and modules on sensor nodes. Policies are the central means of adaptation in our framework and allow separation of concerns in the system, i.e. operational from adaptation logic. We demonstrated how an adaptation FSM can be expressed in terms of *obligation policies* and presented tools that make the translation automatically.

Chapter 5

Fault Classification

Sensor reading faults manifest as a result of circuit ageing and exposure to harsh environments and user mistreatment, that may result in node physical damage or electronics fouling. Moreover, low energy levels on nodes contribute to sub-optimal circuit operation leading to accuracy deterioration. A necessary initial step in an autonomous pervasive system is its ability to detect and identify erroneous readings from sensors to provide input for recovery mechanisms to isolate or repair (when possible) the faulty network component.

In this chapter we formalise fault models for sensor readings that closely resemble faults occurrences in real-world deployments and devise a flexible fault-detection model. The model is flexible with regard to configuration of resources to balance the trade-off between Quality of Information (QoI) and power consumption. Finally, we present a case study on long-running traces that were collected from a real-world WSN deployment for experimental evaluation of the framework's detection accuracy.

5.1 Fault Modelling

We first introduce the faults that appear on sensor readings and study their different classes by modelling them. We also conduct a case study on faults that we have encountered in real-world deployments and another case study that quantifies the impact of sensor readings degradation in classification applications such as activity recognition.

5.1.1 Taxonomy of Sensor Faults

We provide a taxonomy of faults found in sensors that aid their study and development of detection and recovery mechanisms. We identified four classes of faults – *short*, *constant*, *noise* and *drift* faults. Table 5.1 summarises the characteristics and impact on the input for each class.

Table 5.1: Classes of Sensor Reading Faults

Class	*Definition*	*Impact*
SHORT	momentary irregularity in the reading values of a sensor	very small impact as long as spikes remain sparse, easily canceled out by simple rules
CONSTANT	invariant repetition of an arbitrary value that may be relevant to the observed phenomenon	impact could be significant if value is relevant, otherwise it is similar to losing the input of the sensor
NOISE	prolonged increased variance on the reading values of sensors	impact depends on the signal-to-noise ratio, it can be smoothed using mean
DRIFT	smooth persisting deviation (e.g. linear or exponential) of the observed value from the ground truth	initially minor impact that is accumulated over time, eventually distorting readings

In general, *short* faults are perceived as irregular spikes on the input signal, *constant* faults are indicated by a flat signal, whereas *noise* appears as an unstably fluctuating signal. Finally, *drift* error is a permanent, additive deviation of readings from the ground truth. We define formal models for each fault class to assist elaboration on their analysis and identification of features that allow discrimination of these classes from accurate readings.

Short faults

modelled as $S_s = \alpha g(x,t)$, where $g(x,t)$ is the ground truth and $\alpha \in [-k,k], k \in \Re$ a momentary value fluctuation. Short faults manifest on sensors with a uniform probability p.

Constant faults

modelled as $S_c = v$, a persistent value v over a period of time t. Missing readings can also be modelled as a constant faults with $v = 0$, however the cause of the fault is a network failure instead of sensing error.

Noise faults

modelled as $S_n(x,t) = g(x,t) + N(0,\sigma^2)$, where $N(0,\sigma^2)$ is a Gaussian distribution with mean value zero and standard deviation σ. Similar noise models can be found in [UBH09]. Noise faults, are persistent with a duration of t readings.

Drift faults

modelled as $S_d(x,t) = g(x,t) + f(t)$, where $f(t) = \alpha t + \beta$ is a linear monotonic function adding an accumulative error. The α parameter determines the deviation rate of the sensor readings, while the β parameter models jumps in the readings that appear as step changes in the time series. Drift differs compared to the other classes in that is a permanent error that affects subsequent readings.

Modelling of faults allows study of their properties and helps the design of detection and prevention mechanisms. Modelling is also necessary to evaluate the accuracy of detection mechanisms. Unless we artificially inject faults on a sensor input, we are unable to assert that the regions detected are actually faulty as we cannot verify the ground truth. Injecting artificial errors enables precise identification of faulty regions and hence allows evaluation of detection mechanisms.

We selected a linear drift function $f(t)$ for two reasons. First, it simplifies our analysis and, second, evidence from the case study presented in section 5.3 supported this assumption. Exponential drift may be common for other sensors e.g. those which suffer from chemical fouling. The specific type of drift could easily be accommodated by modification of the function $f(t)$

above, thus the assumption of linear drift does not impact the generalisation of the the proposed detection model. A different set of features might be more appropriate from what was used for linear drift, but the core fault detection model should not be dependent on the nature of the drift.

5.1.2 Faults in Real World Deployments

The fault models presented closely follow errors observed in long-running, real-world deployments. We describe such a deployment in this section and contrast identified faults with our modelling. Victoria & Albert Museum, in central London, has deployed a WSN for monitoring temperature and humidity in its exhibition rooms as part of project Ocean[1]. Sensors are deployed in the ambiance as well as inside exhibit casings. Examined data includes two years trace from 78 nodes equipped with a pair of thermometer and humidity sensors. All nodes are able to reach the base station in a single hop. They transmit reading samples every few minutes and have their clocks desynchronised on purpose in order to minimise message collisions over the air. The base station only logs one reading per node every 15 minutes and selects the one closest to the deadline, discarding the rest. This case study was chosen as it used a substantial number of sensors in a real deployment over a long lifetime, which allowed us to gain insight on behaviour of nodes through a long period of time, from freshly calibrated and fully charged to the time that their power levels have nearly or completely depleted.

Nodes are periodically replaced, in rounds, for recalibration and power replenishment, usually annually. Node *IDs* remain immutable when replaced, even though the actual device serial number changes. We have manually examined the node traces to identify instances where sensor behaviour appeared to be abnormal.

We isolated cases where faults are apparent and verify that they follow one of the models introduced. An example of sensor drift manifestation is shown in figure 5.1, where two neighbouring nodes' humidity readings, averaged daily, are plotted for a period of 130 days. Node B appears to drift over a period of roughly 110 days from node A. Even though node readings are very

[1]http://www.vam.ac.uk/content/journals/conservation-journal/issue-46/the-ocean-project-at-the-v-and-a/

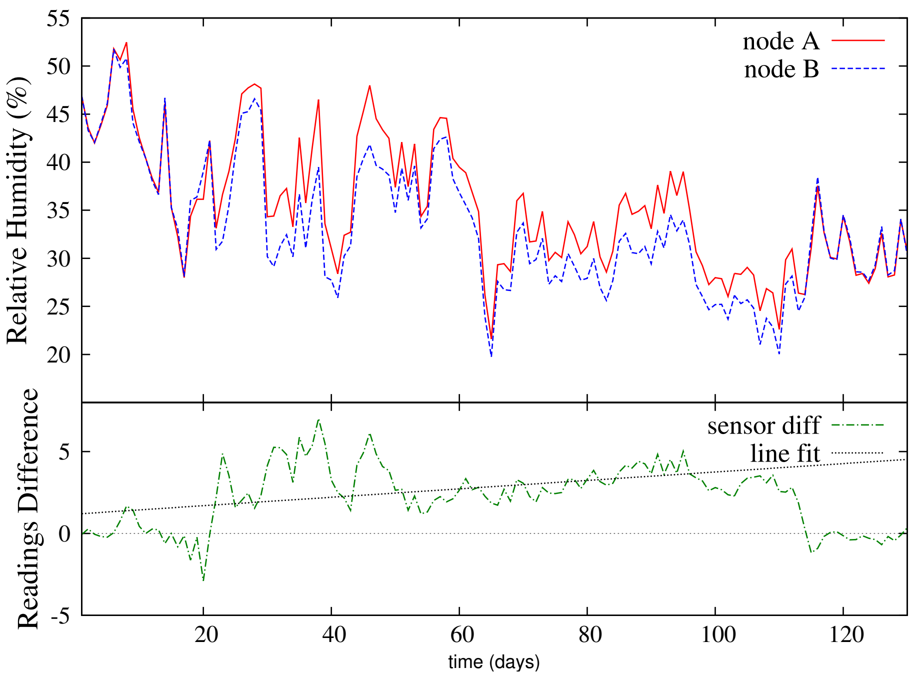


Figure 5.1: Readings faults in real-world deployments

close initially, an event appears to increase variation in readings for roughly 20 days. However, after the end of the event node B does not appear to realign with its neighbour, but instead starts to steadily deviate from node A. The difference of the two sensor readings is plotted in the figure below along with the trend-line that indicates a smooth linear increase. Replacement of node B with a fresh one on day 117 makes more evident that the node was drifting rather than showcasing an actual difference, given that the newly calibrated sensor realigns its readings with those of node A.

5.1.3 Sensor Fault Impact

We use two case studies, where we examine the effects of sensor faults in the classification accuracy of the networks. The first network employs three accelerometer sensors worn on a subject's ear [AEP$^+$07] to determine user activity, while the second one includes 19 accelerometers and one fibre optic sensor for evaluating a surgeon's skill through gesture recognition [KLDY05]. The traces of both the case studies were provided by the Institute of Biomedical Engineering, Imperial College London.

In the first case, the activity recognition trace is collected using the e-AR sensor, an Imperial College London BSN node[2], communicating collected data to a nearby data-logger. It consists of three accelerometers that monitor three orthogonal axes. The experiment involves trace from twelve different subjects performing predefined activities in the following order: sitting, reading, eating, standing, tilting her head, walking, sitting, slouching and lying on a sofa. Each activity is performed for a few tens of seconds up to a minute. As such, the original trace did not exhibit significant amount of faults, caused by ageing of sensors. In [PSM$^+$07], where the trace was originally used, the authors proposed a Bayesian network mixture model approach for their classification method. Further analysis of the trace is presented in [MPT$^+$08] where the authors also use ambient sensors that collect image blobs for profiling of behaviour.

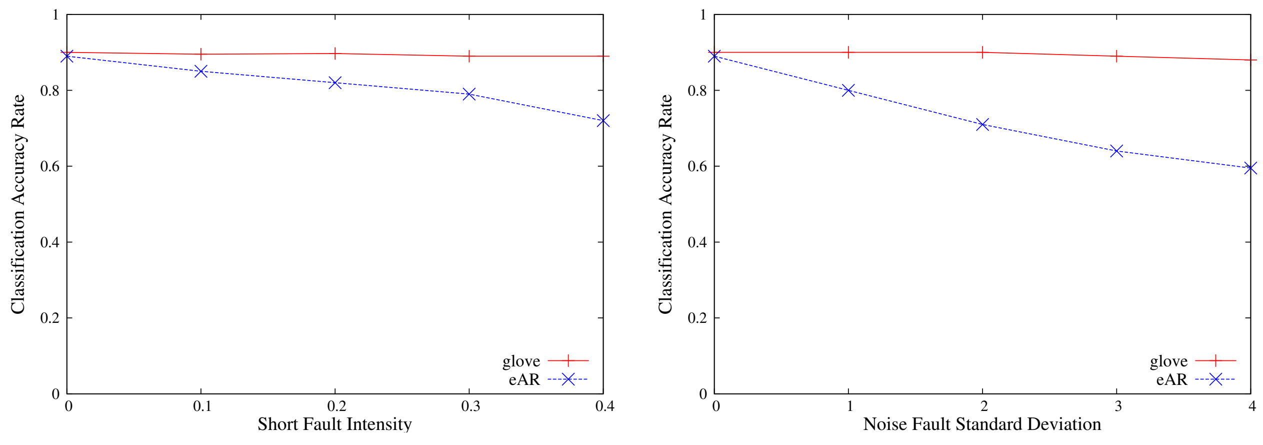
The second trace is collected from a glove that has 19 attached accelerometer sensors on the fingers and the back of the hand and an optical sensor across the palm that measures bending of the hand. A surgeon wearing the glove performs five activities operating a tool tip; left/right traverse, open/close, up/down traverse, rotating the roticulator and rotating the tool tip anti-clockwise. The 20 sensors are attached to four different nodes that propagate readings to a sink node. Five different subjects perform the activities in the trace, for the duration of a few seconds.

For the classification of the activities, we have used a simple k Nearest Neighbours (k-NN) algorithm. In k-NN a data-point's distance in space is compared to a set of previously classified samples and the class that has the most appearances in the k closest samples to it is assigned to the instance. We use a vector of extracted features from a rolling mean of window size 50 for each signal. The k parameter was set to 5 and we use the Euclidean distance in our vector space.

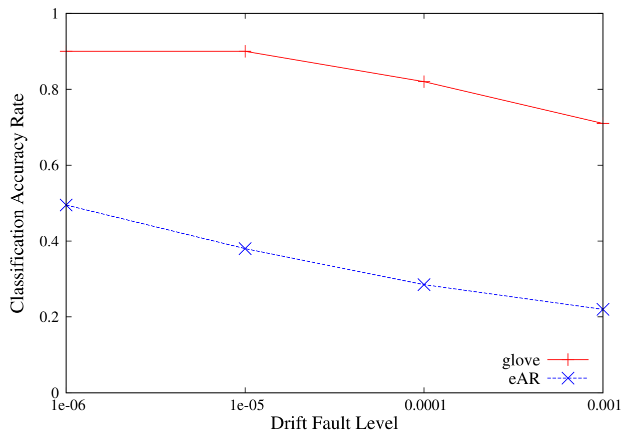
We inject faults from classes modelled in section 5.1 to study the impact of each class to activity classification accuracy. For every BSN node we inject faults into 1/3 of the signals. Accuracy decline in overall classification is illustrated in figure 5.2. In general, the glove trace is much more resilient to fault occurrences that the e-AR trace due to significantly more input signals

[2]http://vip.doc.ic.ac.uk/bsn/

involved in the classification process.



(a) Classification impact of *short* faults (b) Classification impact of *noise* faults



(c) Classification impact of *drift* faults

Figure 5.2: Sensor fault impact on classification accuracy

Short faults, shown in figure 5.2(a), have the least impact on classifications. We have set the occurrence of *short* faults to 5% in the trace and we examine results for different values of the parameter α in the $\{0.1, 0.2, 0.3, 0.4\}$ set. Classification in e-AR trace is affected even with small intensity *short* faults due to the small number of input signals, however classification for the glove experiment is virtually unaffected.

Similarly, the glove trace is resilient to *noise* faults as well, as shown in figure 5.2(b). However, *noise* has even more impact on the e-AR accuracy, which drops from 90% to 60% for *noise* errors with standard deviation set to 4. The x-axis of the figure is the values of the σ parameter in the additive Gaussian distribution model, $N(0, \sigma^2)$, of *noise* faults.

Finally, *drift* has the greatest impact of all faults classes, shown in figure 5.2(c). We apply linear *drift* to a third of the input signals, where parameter α ranges from 10^{-6} to 10^{-3}. Contrary to the other fault classes, *drift* confuses even the glove trace classification in higher values of the α parameter, while the e-AR trace accuracy suffers significantly even in low *drift* cases.

5.2 Fault Detection Model

Figure 5.3 presents the proposed fault detection model per sensor. It consists of three main components – *Feature Extraction*, *Fault Classification* and *Temporal Correlation.*

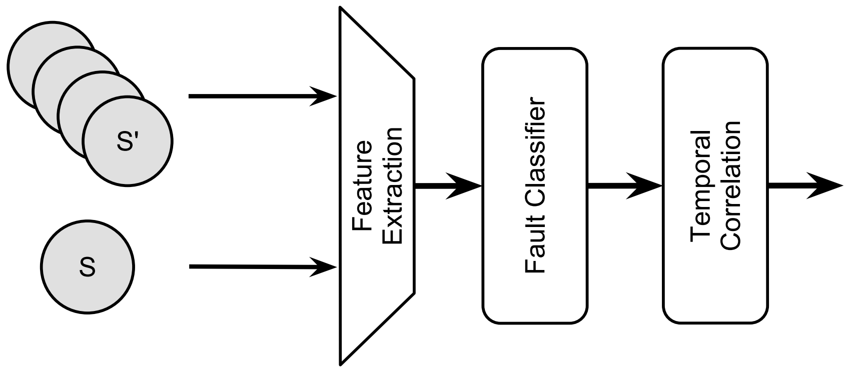


Figure 5.3: Fault Detection Model

Feature extraction collects features produced from sensor readings that help discriminate between correct and erroneous sensor behaviour. A feature selection mechanism that is configurable through policies filters features that are relevant to the application's requirements.

Fault classifier is the component that makes a decision on sensor readings based on receiving signals from extracted features. Such a unit incorporates some of our knowledge and expectations on the monitored attributes in order to make decision on sensor accuracy. Machine learning techniques are usually robust methods for pointing to a good decision based on observed data, even though they lack representation of the formal underlying structure of the system.

Finally, *temporal correlation* is an optional component that operates on the output of the *fault classifier* adding a sense of continuity and consistency on the decisions of a typically stateless classifiers. There are classifiers that include this attribute however many machine learning techniques are endowed with the Markov property, that the next system state depends only on the current one, which propagates incorrect decisions.

In the following sections, we look in more depth the three main components of this model, studying alternative instantiation of such services.

5.2.1 Feature Extraction

Feature extraction highlights specific aspects of readings from sensors. Such aspects are utilised for identifying anomalies or inconsistencies in collected values. We discuss what we consider are the features that assist in detection of erroneous sensor behaviour.

The selection of features is based on attributes that each fault class, described in section 5.1.1, is expected to affect. Features like *mean* and *median* value over a moving window of readings remove the inherent noise of the sensors and allows discrimination of outliers. Furthermore, *variance* provides a metric on input fluctuation. These features can indicate the presence of *short constant* and *noise* faults in readings. *Linear regression* on input samples provides the trend-line that the sensor is following over a long period. It can significantly contribute for identification of *drift* error.

While *unary* features like these provide some indication on the sensor's condition, errors in many cases are indistinguishable from events. Synergy of multiple sources enables observation correlation to discriminate between the two cases. The main assumption is that a manifested event has an area of effect contrary to faults that are expected to be stochastically uncorrelated among nodes. We argue that this assumption applies to a large class of networks that monitor temperature, humidity or illumination in buildings, human vitals on body deployments, acoustic or seismic activity in open areas, etc. The assumption of stochastically uncorrelated sensor faults might not always be the case, for instance, sensors from the same manufacturer tend to have similar failure patterns or a set of co-located sensors may be damaged at the same time. Nevertheless, we argue that for a significant number of cases faults follow this assumption and, thus, are good candidates for detection in our system.

Collaborative features such as *difference*, *ratio* and *correlation coefficient* can contrast observation and utilise their dependencies to reason on fault conditions. These basic features can be composed to more elaborate ones such as *difference of mean values* or *variance ratio* among sensors. Starfish provides an adaptive, configurable mechanism for operators to allow balancing between resource consumption and detection accuracy.

Our goal is to provide a flexible feature extraction mechanism that will allow composition of features concealing details in their implementation. A similar approach on streaming feature extraction in WSNs has been presented in DexterNet [KGG+09] based on SPINE framework[3] for tinyOS that allows applications to easily extract features from sensor deployments focused on body sensor networks. However, DexterNet does not provide for dynamic modification of the streaming feature extraction at run-time.

$$abs(reading(s) - avg(s))$$

$$ratio(var(s), var(clique(cid, s)))$$

Figure 5.4: Composition of feature extractors

We use a simple language based on expressions similar to function calls to allow users to define required features. Figure 5.4 presents sample feature composition that can be expressed, where s is a sensor in the system. Primitive features such as *mean*, *variance*, *ratio* etc. are provided as system functions that operate on their arguments that can either be sensor readings or output values from other feature extractors.

Such composite features can be extracted using the *features* and *buffer* modules discussed in section 4.2.2. Direct transformation of expressions in figure 5.4 into obligation policies that can execute directly on *starfish* nodes is supported by a translation tool and examples are showcased in figure 5.5.

The figure presents the transformation of the two feature expressions of figure 5.4 into obligation policies. The first expression is contained in a single policy that is triggered by a reading of the humidity sensor and calculates the absolute difference of the sampled value and the past average

[3]http://spine.tilab.com/

```
$abs(reading(\text{HUMIDITY}) - avg(\text{HUMIDITY}))$
  **def** policy0
    **on** sensor.Reading(*type*, *value*)
    **if** *type* **is** HUMIDITY
    **do** features.Abs(*value* - features.GetAvg(*type*))


$ratio(var(\text{HUMIDITY}), var(clique(\text{room5}, \text{HUMIDITY})))$
  **def** policy1
    **on** sensor.Reading(*type*, *value*)
    **if** *type* **is** HUMIDITY
    **do** event.Emit(Clique, *type*, nodeId, *value*)

  **def** policy2
    **on** room5.event.OnEvt(*evt*, *sensor*, *node*, *value*)
    **if** *evt* **is** Clique **and** *type* **is** HUMIDITY
    **do** features.UpdateClique(room5, *sensor*, *node*, *value*)

  **def** policy3
    **on** features.OnClique(*cliqueId*, *sensor*, *value*)
    **if** *cliqueId* **is** room5 **and** *type* **is** HUMIDITY
    **do** buffers.PushBack(*cliqueId*, *value*)

  **def** policy4
    **on** sensor.Reading(*type*, *value*)
    **if** *type* **is** HUMIDITY
    **do** features.ratio( features.Var(*type*), features.Var(room5) )
```

Figure 5.5: Feature extractors translated in obligation policies

value readings. The second expression is more complicated and requires network communication among a group of nodes. The *clique* feature is the set of latest readings from a group of nodes. In this case the clique's ID is 'room5' and is a role assigned to a set of nodes that designates their co-location property. Role assignment as discussed in section 4.1 creates dynamic groups in the framework and assists communication.

The *clique* feature is supported by two functions of the 'features' module. The first is 'features.UpdateClique()' action, which updates the module with a sensor's latest reading. Event 'features.OnClique()' is fired when latest readings from all sensors are received and carries the mean value of the clique's readings. Policies 1-4 in figure 5.5 demonstrate a composite feature extractor that uses the *clique* feature and its translation into obligation policies. The first policy propagates the sensor reading to the network emitting an event with the 'Clique' ID. The second policy collects all the 'Clique' events from SMCs that are assigned the 'room5' role and updates the local 'features' module. When all sensor readings are collected the average is calculated and the third policy stores the received value in an allocated buffer. Finally, a local reading triggers the fourth policy that compares local variance of the sensor with the clique aggregate variance.

Defining feature extraction in terms of obligation policies helps updating them dynamically when necessary to meet dynamic requirements of the application. Consequently it allows on demand allocation of resources for fault detection.

5.2.2 Fault Classifiers

The *fault classifier* unit uses collected features for reasoning about a sensor's condition to infer its state. We compare rule-based heuristics and Bayes probabilistic classifiers in the trace from the V&A deployment and investigate their performance.

Heuristic Rules

Rule based systems can express expert knowledge on normal behaviour of monitored properties and characterisation of faulty readings. Although it is a lightweight and efficient mechanism for error discrimination, it can be ineffective when a priori knowledge of the underlying attributes and their relations is inadequate.

We applied heuristic fault detectors on the two body area network deployments discussed in section 5.1.3 – the e-AR and glove traces. We inject faults at one third of the sensors in each trace. Table 5.2 summarises the overall accuracy of detection rules per fault class. The table presents the average results for experiments that were run with different fault parameters. Evaluation of the *constant* fault detection is omitted as, in the case of accelerometer, detection is trivial due to the unstable nature of the accelerometer's signal.

Table 5.2: Fault Detection Accuracy

detector	hit-rate	fall-out
SHORT (stdvar)	99.54%	0.77%
NOISE (variance)	98.33%	7.43%
NOISE (corrcoef)	100%	6.32%
DRIFT (regress.)	95.22%	19.47%
DRIFT (corrcoef)	78.22%	3.89%

The metrics used for the evaluation are the *hit-rate* of detected faults, where a hit indicates at least one alert is triggered during fault manifestation, and the *fall-out*, which is defined as the ratio of false positives to the sum of false positives and true negatives. The experiments run for roughly 10 minutes. Nodes take 32 samples per second from their sensors, while the duration of injected faults like *noise* or *drift* is 1200 samples.

The local detector for *short* faults proved to be very accurate, yielding very low false positives. A simple heuristic that checks whether the reading falls inside the $\hat{\mu} \pm 3 \cdot \sigma$ range, where $\hat{\mu}$ is the mean value over a recent sampling window and σ is the standard deviation.

For *noise* and *drift* faults we use two different heuristics, one using local features of readings variance and linear regression analysis respectively for each class and a second that compares to neighbouring sensors using correlation coefficient. Both *noise* detection techniques have high hit-rates, however, the correlation coefficient approach slightly decreases the number of false positives. Local variance of sensor readings is, however, less expensive to extract in terms of power consumption. We use a window of 200 samples in the calculation of correlation coefficient. The window size is a trade-off between detection delay and the number of false positives.

The linear regression technique for *drift* faults has a higher hit-rate compared to correlation coefficient, but at a cost of high number of false positives. As expected, further analysis of the results indicated that the *drift* cases that escaped detection from the heterogeneous method are those that have a very smooth deviation from ground truth, thus with lower impact. Regression analysis for detecting the input's trend is computationally intensive to perform on the sensor nodes. Instead we take a very rough estimation by calculating the slope of the line passing over two data-points, one at the beginning and one at the end of the regression window. In order to tolerate outliers that could significantly skew, instead of the first and last reading in the window, we use the median of the first five and last five readings. The approach yielded slightly degraded results compared to actual linear regression, but is comparable, considering the computation gains.

In the V&A deployment, *short* and *const* faults can be easily expressed by heuristic rules with very low false positives. *Noise* faults can also be described with relative success, but the *drift* faults are more subtle and elusive to heuristics. Figure 5.6 demonstrates heuristic rules for different fault classes. Rules are expressed in terms of composite feature extractors that were discussed in section 5.2.1 and can be transformed into obligation policies.

```
if $abs(reading(s) - avg(s)) > 3 * stdev(s)$ and
    $abs(reading(s) - clique(s)) > T_{Sclique}$ then short

if $var(s) < T_{Clocal}$ and
    $ratio(var(s), var(clique(s))) < T_{Cclique}$ then const

if $var(s) > T_{Nlocal}$ and
    $corrcoef(reading(s), clique(s)) < T_{Nclique}$ then noise

if $regr(avg(s)) > T_{Dlocal}$ and
    $ratio(regr(avg(s)), regr(clique(s))) > T_{Dclique}$ then drift
```

Figure 5.6: Heuristic rules examples for sensor readings fault detection

These rules express our expectations for the monitored attributes. For instance, we do not expect the temperature and humidity readings to make sudden jumps. As a result, any reading that exceeds the mean value of readings more that three times their standard deviation becomes suspicious for *short* fault. Further evidence of the fault is provided, if the sensor reading's difference from the mean value of its neighbouring clique is greater than a threshold.

Similar rules are devised for other fault classes. *Const* faults are characterised by a low variance on the sensor's reading and is further validated when the ratio of a node's variance to its clique's is very low. Increased variance in the input is used to characterise *noise* faults that are described by two rules. The first is based on a local decision when the variance of a sliding window that is calculated on every new sample is above the T_{Nlocal} threshold. The second rule uses the observation of the neighbourhood and compares the correlation coefficient of local readings with that the mean value of the neighbour.

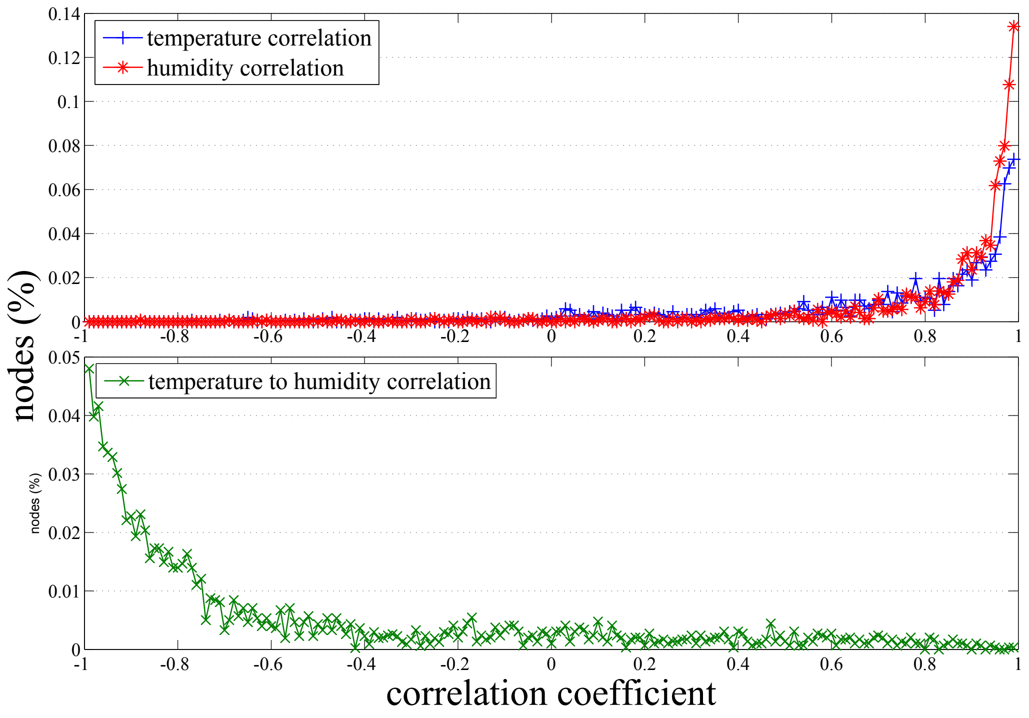


Figure 5.7: Distribution of correlation coefficient metric of temperature and humidity sensors

Figure 5.7 shows that the majority of neighbouring nodes exhibit a very high correlation coefficient, when they are healthy. The top plot shows the probability density function (PDF) of the correlation coefficient metric between pairs of co-located temperature and humidity sensors. It demonstrates that these sensors exhibit a very high correlation in their readings. The lower plot demonstrates that a similar negative correlation exists when comparing readings of temperature to humidity sensors, even though the monitored attributes are not homogeneous. Nevertheless, the correlation evidence in this case is weaker.

The second rule, in figure 5.6, is necessary to override cases where an increased variance is caused due to an event in the environment observed in an area instead of a random error. Sensor input is typically somewhat noisy but this comparison allows discrimination of input that is noisy beyond an acceptable level. For *drift* detection, we rely on the long term analysis of the input by calculating the trends and deviation between neighbouring nodes. Thus, the rule calculates the trend of the median difference of readings of two sensors and alerts for drifting in cases where this trend appears to be deviating from a low threshold.

Composition of basic features allow for a variety of attributes that can be examined. The rules work by short-circuiting the condition parts resulting in avoiding calculation of the second part of the condition if the first one is false. Consequently, power consuming functions that require node communication are invoked only after an initial hint from local functions.

Bayes Classifier

Heuristic rules are very tied to deployment relying on hard thresholds and fail to accurately capture complex relations of attributes resulting in a high rate of inaccurate classification and false positives. A naïve Bayes classifier is a method of supervised learning. Although it assumes independence of random variables, in practice it outperforms other methods even when the model does not follow the variable independence assumption. Among its strengths is the fact that it is a very efficient method for probabilistic inference.

The formal model of the classifier is shown in equation 5.1, where the probability of the random variable C, in this case sensor's state (*healthy*, *noise*, *drift*), is dependent on the values of features $F_1, ..., F_2$. In other words the *posterior* probability of the value of a random variable C is the product of the *prior* probability and the *likelihood* given a vector of features.

$$p(C|F_1, ..., F_n) = \frac{p(C)p(F_1, ..., F_n|C)}{p(F_1, ..., F_n)} \tag{5.1}$$

As the main assumption in a Bayesian classifier is that features $F_1, ..., F_2$ are independent their conditional probability of the vector is the product of their separate conditional probabilities shown in equation 5.2.

$$p(F_1, ..., F_n|C) = p(F_1|C)p(F_2|C)...p(F_n|C) \tag{5.2}$$

Consequently, a classifier based on this model selects the value for the random variable C that is most probable, as expressed in equation 5.3, where the *argmax* function returns the argument that maximises the expression that follows. The denominator from equation 5.1 can be removed as it is a constant.

$$classify(f_1, ..., f_n) = argmax_c|\ p(c) \prod_{i=1}^{n} p(f_i|c) \tag{5.3}$$

The naïve Bayes classifier is an efficient supervised learning mechanism that is feasible to implement inside the network running on individual nodes, or cluster heads. Its memory requirements are relatively low, requiring storage of link matrices of joint-probabilities between random variables F_i and C. Communication is limited to cases where a classifier uses features that require information from other sensors inside a clique.

The Bayes classifier is also adaptable with regard to detection accuracy as a trade-off against features that are used for the classifier's input. In general, features that include information from other nodes provide more accurate detection of faults, but increase the operational cost, as communication increases the power consumption. The features used are similar to those in a rule-based classifier and their characteristics (i.e. joint-probabilities of random variables) can be learned by statistical sampling from existing observations.

5.2.3 Temporal Correlation

The classifiers discussed in the last section do not consider the history of decisions made during the operation. For chaotic faults in readings such as *short* faults, which appear instantaneously, this has no important consequences. However, for *noise* and *drift* classes that appear for a prolonged period, the classifiers alone return a significant number of false negatives.

Consistency of fault detection allows for disambiguations between different instances of the same fault class. It is also important to have an accurate estimation of a fault's duration for developing automated mechanisms that can study its effect and determine its scale. For instance, identifying the time interval for which *drift* appears, allows for a better estimation of a linear function that can be applied to reverse the effect.

Furthermore, even though the classifiers have been successful in detecting fault cases they yield a significant amount of instantaneous or short-lived alerts of false-positives. Fault detection alerts trigger processes of self-recovery from faults, either by isolating a sensor or applying correction functions, increasing control traffic between nodes. It becomes crucial to confine such resource usage only in cases that are necessary for avoiding false alerts.

This is the role of the *Temporal Correlation* (TC) unit in the fault detection model in figure 5.3 that is used to compensate for the lack of temporal continuity in classifier decisions. TC unit's role is to incorporate history into the final decision.

Hidden Markov Models (HMMs) is a probabilistic inference tool that we use to achieve consistency by post-processing decisions from classifiers. HMMs provide a mechanism to determine the state of system, which is unknown, based on a series of observations, which are perceivable by the system. The model is presented in figure 5.8. There are two planes in the model – the *Hidden States* and the *Observations*. The former plane consists of all the potential states a system may be in, in our modelling this includes *healthy*, *noise* and *drift*. We focus on *noise* and *drift* faults as they are the most challenging to classify. The *Observation* plane includes signals that are provided as input to the model. Classifier's output decisions are used as input observation for the HMM.

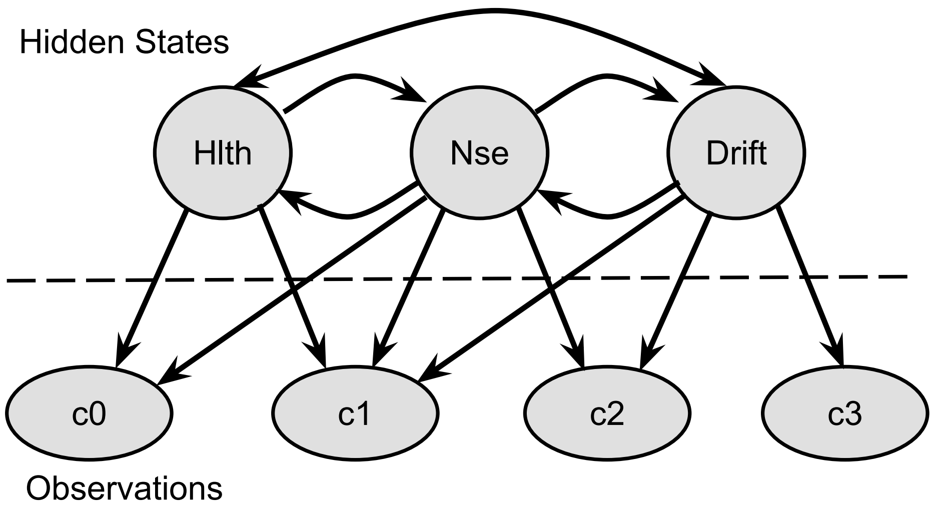


Figure 5.8: Hidden Markov Model for decision post-processing

The system may transition from one state a to state b with a transition probability $p_{a,b}$. A symbol λ, i.e. observation, may appear while the system resides in a state s with an emission probability e_s^λ. Given this model there is a number of inference problems. *Filtering* is the process of estimating the distribution of hidden states at the end of a sequence of observations, given the model's parameters. The filtering problem is solved with a forward algorithm that calculates the belief on a state based on evidence, i.e. observations. The forward algorithm is efficient. For an HMM with S states running on a observation sequence of length N, it has a complexity of $O(N \times S^2)$. Given the constant, very small number of states in our model, the algorithm is transformed into linear to the length of the observations' history, $O(N)$.

This model has the limitation of being able to detect only one type of fault, i.e. either *noise* or *drift*, in the system even though they may manifest concurrently. We argue that this is not a major flaw as it allows the model to remain simple and robust, while the dominating fault type is detected in such cases.

5.3 Case Study: Victoria & Albert Museum

We study the effectiveness of the fault detection model by measuring its accuracy in identifying faults. We use the trace collected in the Victoria & Albert museum deployment that was described in section 5.1.2. We evaluate our detection mechanisms by manually selecting traces that appear to have no major faults and inject faults following the models introduced in section 5.1.

5.3.1 Fault Detection

Fault injection is necessary to evaluate a fault detection mechanism. It enables testing and measuring accuracy of detection and false alarms, which otherwise would not be possible to assess without knowledge of the ground truth. Fault injection allows control of the exact period of faults and testing a range of intensities of the effect. Experimental analysis, showcased in section 5.1.2, provides evidence that our models closely follow real-world error conditions.

We randomly inject faults of all four classes with different scale, parameters and duration. *Short* and *const* class faults were easier to identify as expected with simple heuristic rules presented in figure 5.6. In the evaluation section we focus on the *noise* and *drift* classes that are more challenging to accurately detect with heuristic rules.

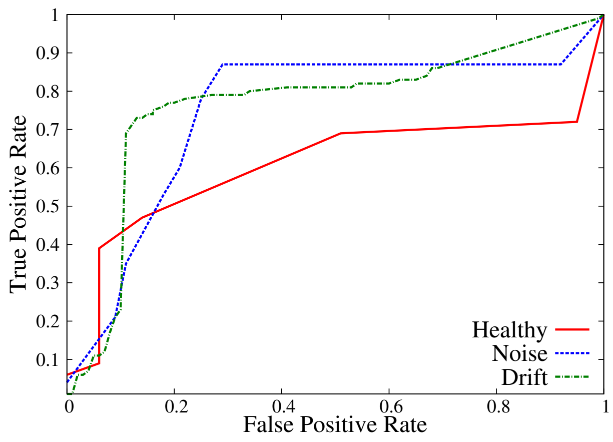
Intensive *noise* faults are relatively easy to identify, especially when information from co-located sensors is used, as a sensor's variance in readings is significantly skewed. However, for noise faults with lower intensity (i.e. lower values of σ in their Gaussian distribution) heuristic rules are not as effective. Increased variance can also be translated as a behavioural change in the environment. Hence, disambiguation between fault or normal behaviour becomes hard without the use of information external to the node.

Drift errors present lower detection rates, usually, being a very smooth deviation from the ground truth that is accumulated over time. Their initial phase has minor effects, thus, becomes hard to discriminate. Again, without reference from neighbouring nodes *drift* may be confused with a normal decrease/increase of temperature due to external factors, e.g. moving from summer to autumn.

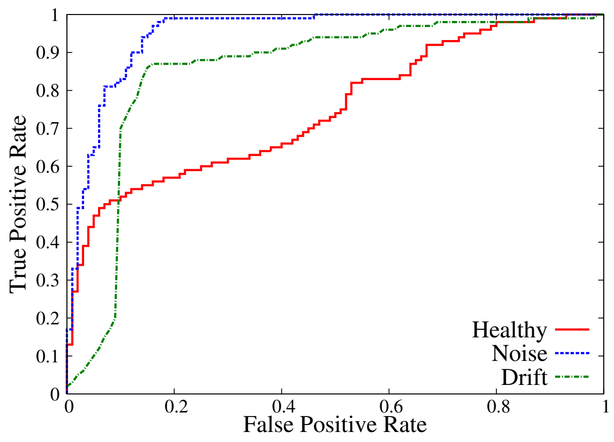
Plots in figure 5.9 summarise the accuracy detection results of different fault classification approaches. Performance of each classifier is presented in the form of the Receiver Operating Characteristic (ROC) curve for each sensor condition – *healthy*, *noise*, *drift*. The ROC curve compares the relation between *false positives* (FP) ratio and *true positives* (TP) ratio of a detection mechanism for binary variables by modifying the discrimination threshold used. A perfect classifier has a ROC curve that is as close to the top left corner of the plot as possible, which translates to a perfect detection without any false alerts.

The results have been extracted by averaging classification algorithms accuracies for faults on both temperature and humidity sensors over 9 groups of nodes. The node groups were manually hand-picked based on their proximity and room location from the museum's deployment floor-plan.

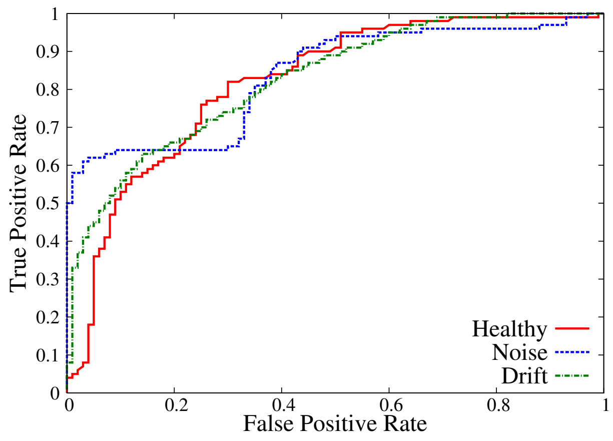
Figure 5.9(a) presents the accuracy results of a rule-based classifier, presented in figure 5.6, using



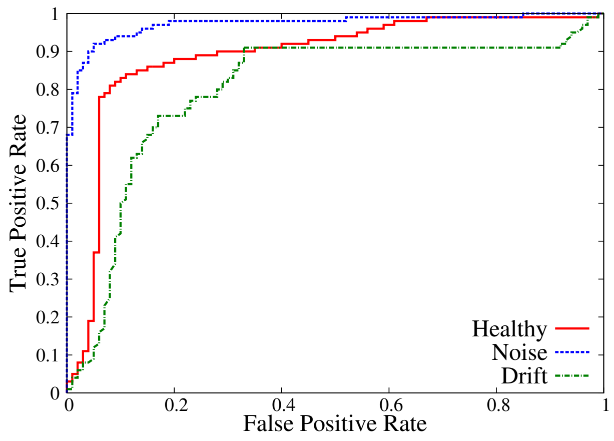
(a) Local features rule-based classifier



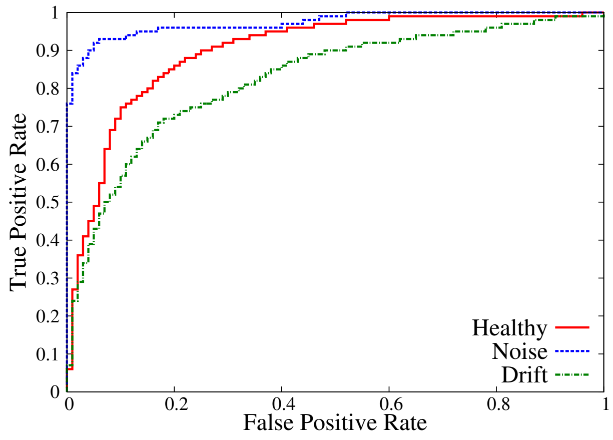
(b) Local features rule-based classifier with HMM



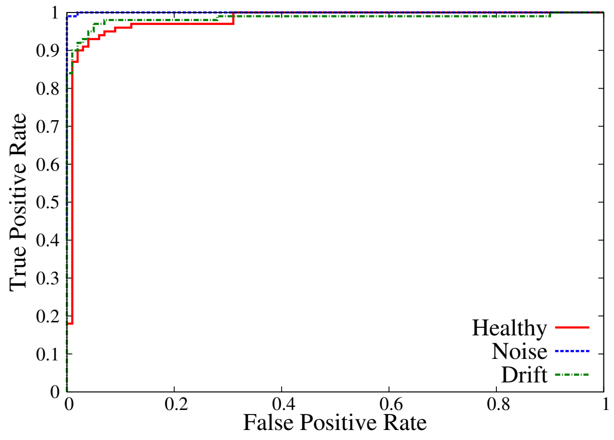
(c) Rule-based classifier



(d) Rule-based classifier with HMM



(e) Naïve Bayes classifier



(f) Naïve Bayes classifier with HMM

Figure 5.9: ROC analysis of fault classification approaches

only local features, i.e. only the first part of the rules. It is evident that fault classification is not accurate and particularly the classification of healthy state is confused. As the tolerance to false positive increase so does the successful detection of faults, however the lack of external information causes healthy states to be confused. This is also the ROC curve that demonstrate the detection accuracy of Redflag [UBH09] that uses the same heuristic rule for discriminating noisy signal based on a variance hard threshold. Even though Redflag has a mechanism to correct *drift* error it does not support a detection mechanism, thus cannot be compared.

Figure 5.9(b) presents the classification of the same local rule-based classifier post-processed by the HMM model introduced. The HMM post-processing increases the accuracy of fault detection especially for noise faults that used to have disparate alerts. The HMM is able to correctly infer those alerts as a single event of long duration. However, the classification of the healthy state does not substantially improve. Both classification methods are limited by the lack of context obtained from neighbouring nodes.

In figure 5.9(c) and figure 5.9(d), we plot the ROC curves for a rule-based classifier that uses information from neighbouring nodes in addition to local features, as defined in figure 5.6. We present result both with and without HMM post-processing to highlight the improvement they provide on the initial classification. Results are significantly improved compared to the corresponding local approaches. It is interesting to note that HMM provide great consistency on fault classification as, in general, local classification with HMMs provide higher true positives on fault detection rather the simple rule-based classifier with neighbourhood information. The HMM adjustment does not improve on the number of events that have been detected but rather constrains their manifestation periods, thus increases the overall true positive sampling instances in the ROC curves.

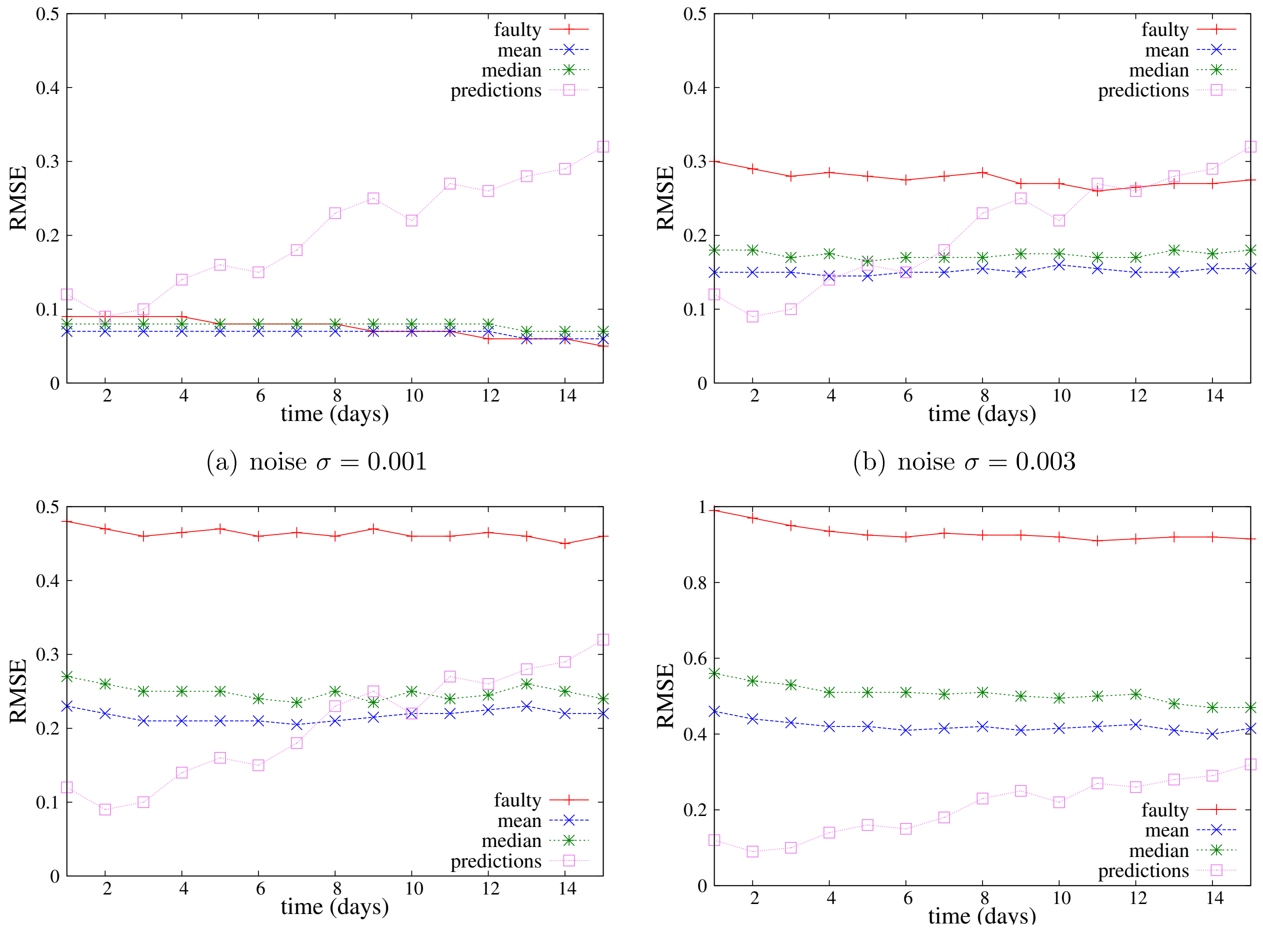
In general, the HMMs and rule-based classifier approach that uses network information yields limited false positive rates and increases correct classification, especially for *noise* faults. *Drift* faults still present moderate success in detection, but this is mostly caused by misclassification of early stage instances of *drift*, when the impact on sensor readings is minimal due to slow manifestation of the error.

The most significant improvement on fault detection accuracy is presented in figure 5.9(e) and figure 5.9(f), which are the ROC curves of the naïve bayes classifier without and with HMM support respectively. Features used are the same as in the case of the rule-based classifier. The naïve bayes classifier accuracy is very close to the results of a rule-based system with HMM support, that provides moderately high detection. However, HMM support dramatically improves classification, moving all the curves to the upper left corner of the diagram. Most notably, *noise* fault detection achieves almost perfect classification rates, while misclassified healthy states are decreased to a minimum, reducing the vast amount of false alerts.

5.3.2 Fault Correction

When a reading fault is identified the system may select to ignore the reading, if it is transient; isolate the sensor, by removing it from the data collection mission; or attempt to replace the faulty signal either by applying a filter function or by replacing it with a prediction model. In this section, we present experimental results that compare different approaches for the case of faulty temperature readings in the Victoria & Albert museum trace. We present an analysis on the qualitative improvement that is gained with a self-healing network that detects and automatically provides alternatives to faulty sensor readings. We focus our study on *noise* and *drift* faults and compare *filtering* and *replacement* approaches with regard to root mean squared error (RMSE).

In the case of *noise* faults, we apply correction of sensor readings by applying an averaging and a median filter that consider sensor readings of the past hour. Alternatively, we use forecasting based on the Holt-Winter's additive model for time-series for replacement of the faulty sensor. The Holt-Winter's approach builds a model by using historic data from sensor readings and is able to create prediction for future values. It decomposes the signal in three components – *level*, *trend* and *seasonality*. The output is a linear combination of the three components. The prediction model is described in more detail in section 7.1.2. We prototyped the recovery mechanisms in python and run them against sensor traces with injected *noise* of different intensities – $\sigma \in \{0.001, 0.003, 0.005, 0.01\}$.



(a) noise $\sigma = 0.001$ (b) noise $\sigma = 0.003$

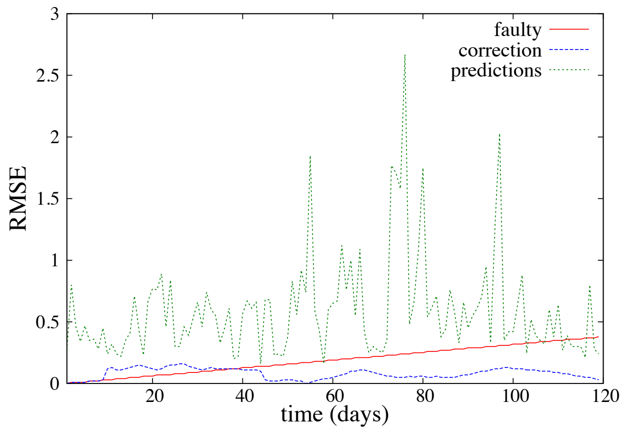
(c) noise $\sigma = 0.005$ (d) noise $\sigma = 0.01$

Figure 5.10: Root Mean Squared Error (RMSE) of readings estimators on noisy sensors

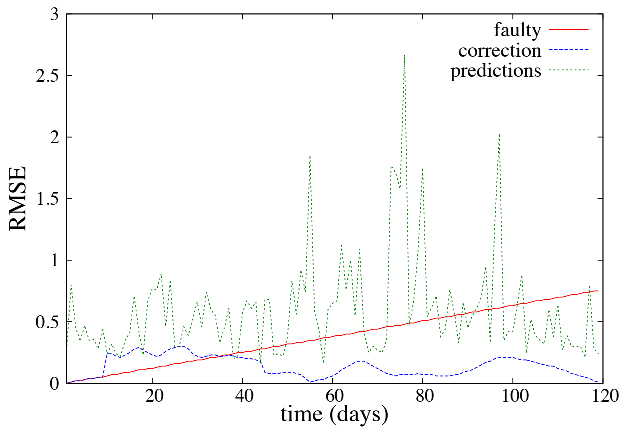
Figure 5.10 presents the root mean squared error (RMSE) values for each readings estimator and compares it with the RMSE values from a faulty sensor without applying any healing mechanisms over a period of 15 days. In the first, figure 5.10(a), the noise injected in the signal is very low. Consequently, the RMSE from the faulty sensor is minor and the averaging and median filters doe not provide any improvement. The Holt-Winter based prediction estimator actually returns higher error rates. On the contrary, in figures 5.10(b)-5.10(c), which present results with *noise* σ values of 0.003 and 0.005 respectively, the RMSE of the faulty sensor is increased. The forecasting estimator performs better in the initial days after the failure. However, the predictions start to become inaccurate as the model projects values further away in the future. Averaging and median filters present a stable RMSE that after the first week of erroneous readings becomes more resilient than the forecasting estimator. However, as the intensity of the *noise* faults increases (e.g. in figure 5.10(d)) averaging and median filters cannot adjust well to the increased variation of the signal and return higher RMSE compared to the forecasting estimator. Between the two filtering methods, averaging filter consistently outperforms the median. A cause appears to be the low sampling rate that includes only 4 readings in an hour. An ideal self-healing mechanism should consider the value of the standard deviation of sensor input as well as the time distance from the appearance of the faulty behaviour in order to select an optimal mechanism. Policies can express such behaviour by dynamically swapping correction techniques on the fly based on these criteria.

Figure 5.11 shows, respectively, the root mean squared error values for *drift* errors of different intensities – deviation of 0.5, 1, 2, 3 degrees Celsius over a period of 6 months. Figures that demonstrate *drift* RMSE expand to longer time periods, 120 days, as the effects of *drift* become more intense over time compared to its initial phase. Contrary to *noise* faults, RMSE of *drift* readings start very low but they linearly increase over time as expected, since we consider linear *drift* of sensor nodes. The forecasting estimator is again unaffected by the fault's intensity as it only uses past healthy observations for its model. In this case, a long period of forecasting predictions is presented that spans to over 4 months. The prediction quality fluctuates a lot over time presenting spikes for days that do not follow the expected pattern. Nevertheless, it still performs better than the faulty readings for deviations higher than 2 degrees over a period of 5 months.

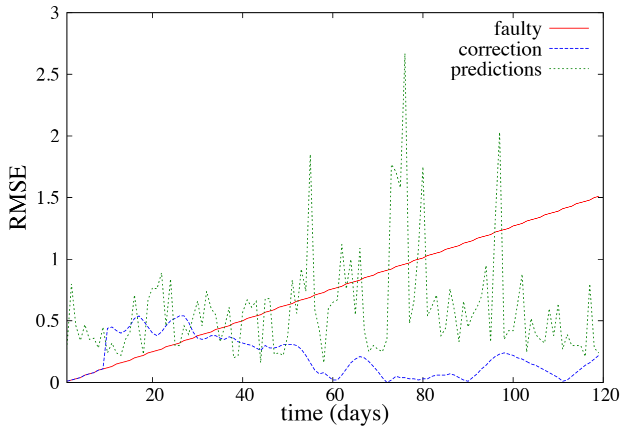
Finally, we apply a correction function on sensor readings that attempt to inverse the effect of *drift*. As the *drift* is linear (i.e. $\alpha x + \beta$), we estimate the α and β parameters by applying linear regression on the distance of the daily average value of readings from the faulty sensor to the daily average value of its neighbouring clique. Nodes in the same neighbourhood tend to be highly correlated as described earlier in this chapter, so we expect their difference to remain stable under normal conditions. The curve that presents the correction functions performance in figures 5.11a-d initially performs similar to the fault sensors as there is not enough data for linear regression estimation and soon after it has a sudden jump in its error rate. Due to the small number of samples the initial estimation is not very accurate, however, this changes quickly. After a period the faulty sensor's RMSE increases while the correction estimator's RMSE decreases having more accurately identified the amount of drift. After that point in



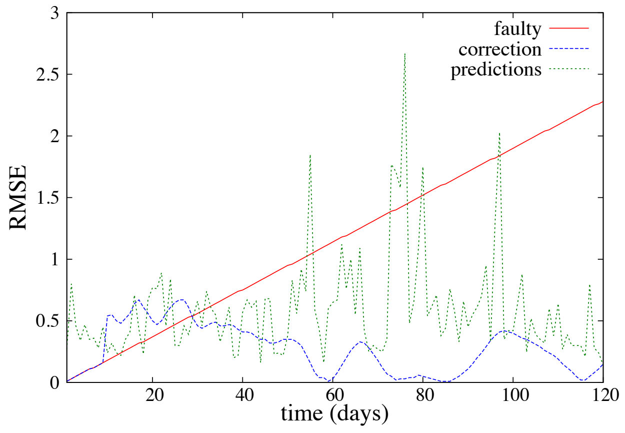
(a) deviation of $0.5^o C$ over 5 months



(b) deviation of $1^o C$ over 5 months



(c) deviation of $2^o C$ over 5 months



(d) deviation of $3^o C$ over 5 months

Figure 5.11: Root Mean Squared Error (RMSE) of readings estimators on drifting sensors

time, it significantly outperforms the other approaches. In the example presented in figures 5.11a-d some humps in the RMSE curve can be spotted. These are due to days that the sensors exhibits a lower than usual correlation to its neighbours. Such occurrences can be explained by external factors, e.g. a window that is opened and affects temperature in one part of the room. However, such conditions do not significantly degrade the quality of the estimator.

The average improvement of root mean squared error for *noise* fault on a single sensor per day is shown in figure 5.12, derived from average RMSE presented in figures 5.10a-d. In the figure the improvements of the estimators are compared. First, the averaging filtering approach and second an adaptive estimator that start using forecasting values from the Holt-Winter's model for an initial period that depends on the *noise* intensity and later switches to the averaging filter for the remaining time.

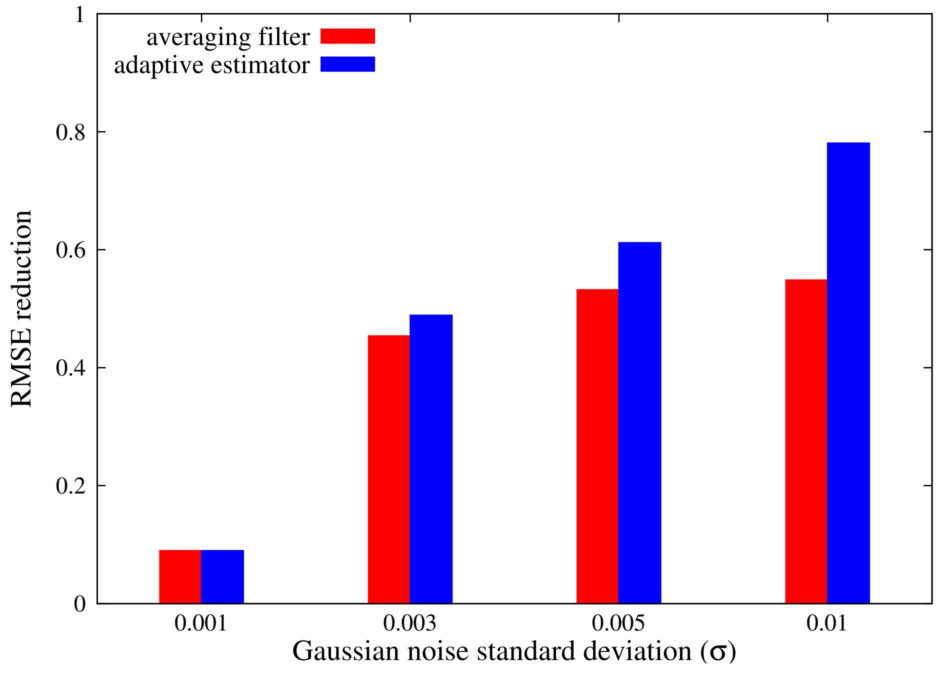


Figure 5.12: RMSE reduction of averaging filter and optimal estimators over noisy sensor

Similarly, figure 5.13 presents overall RMSE improvement for the case of a faulty node that exhibits *drift*. In this case, we compare the estimator that uses a reverse correction function and estimator that uses Holt-Winter's forecasting against the RMSE of the drifting sensor. The forecasting mechanism does not provide any improvement for small amounts of drift. On the

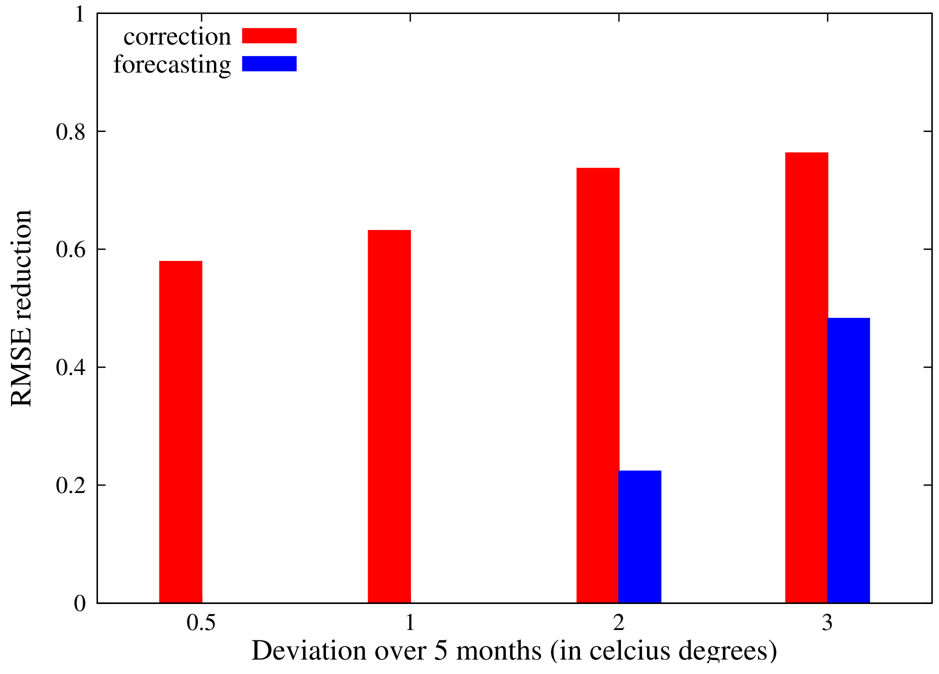


Figure 5.13: RMSE reduction of averaging filter and optimal estimators over drifting sensor

contrary, it provides worse RMSE rates, and thus, is omitted from the figure. This is a result of projecting predictions too far in the future that become more inaccurate as the prediction model does not have enough evidence for those trends. On the other hand, the correction function that is devised by linear regression on the distance between sensor against neighbourhood readings gives a very good estimation of the ground truth as long as the involved sensors are correlated.

Sensor nodes in the V&A deployment are substituted annually for battery replacement and recalibration. By manually analysing the trace, we identify instances where faults manifest in the trace, similar to the case in figure 5.1. The histogram in figure 5.14 presents the observations of *noise* and *drift* faults in sensor input from our analysis. Time in the horizontal axis is measured in weeks past a node's deployment and normalised over a year, which is their replacement period, i.e. approximately 52 weeks. Instead of a batch replacement of all nodes at the same time, node substitutions are spread over the year. Consequently, there is constantly a mix of recently calibrated and older nodes in the trace.

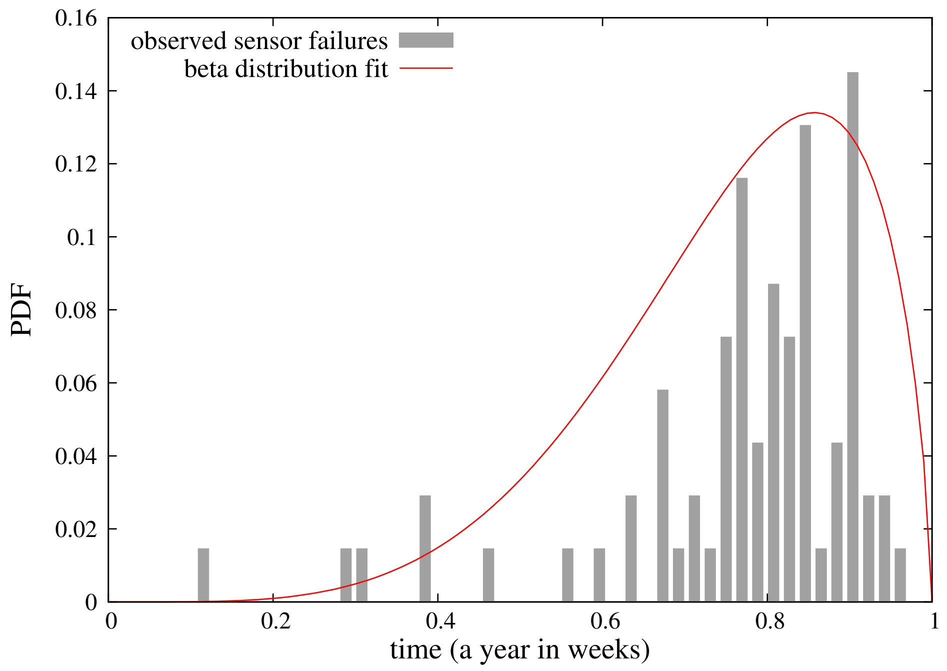


Figure 5.14: Probability Density Function of sensor failures during their lifetime

Our observations indicate that the probability of a node reporting faulty readings increases with its age. Most fault appearances occur close to the end of node lifetimes, in the last third of their deployment time. The observations fit a *beta distribution*, where beta function's parameters extracted from the fitting process are $\alpha = 5.2$ and $\beta = 1.7$. The curve in figure 5.14 presents the probability density distribution for the time instance that a node starts reporting faulty readings since initial deployment. The probability density function of the beta distributions is defined in equation 5.4.

$$f(x;\alpha,\beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{\int_0^1 u^{\alpha-1}(1-u)^{\beta-1}du} \tag{5.4}$$

The mean value of the beta distribution, given equation 5.5, results in sensors failing around their 39th week of deployment (roughly after the 75% of their lifetime). Consequently, a sensor operates with errors for 13 weeks (91 days) on average.

$$\hat{\mu}_B = \frac{\alpha}{\alpha+\beta} \tag{5.5}$$

The standard deviation of the beta distribution is given in equation 5.6, which results to $\sigma = 0.153$.

$$\sigma_B = \sqrt{\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}} \tag{5.6}$$

Results from recovery techniques are reported in this section in an attempt to quantify and provide some insight on the benefits of self-healing mechanisms in a pervasive system. Other more elaborate healing mechanisms may be employed as well that can further decrease RMSE. For instance, Kalman filters or a domain-specific prediction model that incorporates knowledge on temperature behaviour might prove to be even more accurate in similar applications. However, our goal is to give an estimate of the benefits provided from a self-healing approach and the benefits of fault detection rather than present the most effective filtering mechanism.

5.4 Summary

In this chapter, we presented a classification framework for data faults in sensor networks. The framework describes a two-phase solution. In the first phase a classifier infers sensor condition based on collected local and nearby node features. In the second phase, we use a Hidden Markov Model to refine the decisions of the classifiers by modelling the persistent nature of the fault classes. The framework is flexible in the sense that refining of fault detection is not dependent on the classifier used by the network. We further demonstrate the accuracy of our approach and the benefits of a sensor readings recovery mechanism by a case study on real-world trace collected in a museum environment in central London.

Chapter 6

Role & Mission Adaptation

In section 4.3, we discussed and provided examples for different types of adaptation supported by the *starfish* framework – local node adaptation on its *objectives*, network group adaptation on *operations* and adaptation driven by new *functional requirements*. This chapter focuses on adaptation of network node operations in the face of failing components. We focus on the planning phase of the autonomic feedback closed-control loop and more specifically on dynamic task reallocation in pervasive systems. We propose a mechanism that automatically allocates *starfish* roles and missions at deployment-time. Furthermore, it autonomously updates the initial plan based on metrics it collects during the system's execution.

Task allocation has been studied extensively in traditional distributed systems, however, the setting differs significantly from pervasive systems. In traditional distributed systems allocated tasks typically involve batch processing and splitting jobs in parallel chunks to increase the overall system throughput. Power consumption or limited resources are not concerns in such cases. In pervasive applications, allocation of the tasks is motivated by spatial distribution of the monitoring process to cover the observation range of the system and provide monitoring redundancy rather than load balancing and parallelism. Tasks are, typically, not computationally heavy, long running batch processes. Instead the model is *event-driven* where tasks respond to events from their environment with short running processing. Minimising communication is more crucial given that it is the most costly operation on power-constrained nodes.

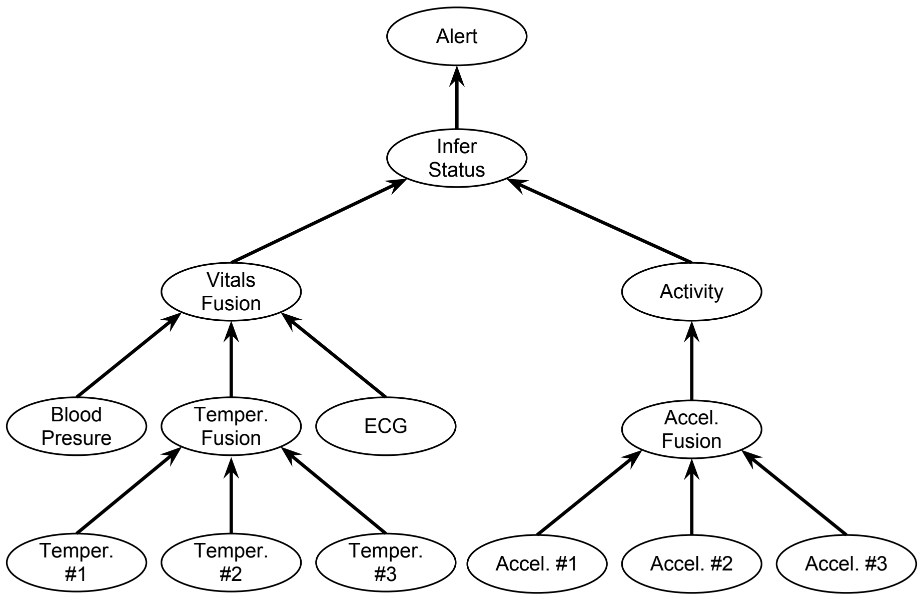


Figure 6.1: Health-care scenario task graph

6.1 Task Adaptation Scenario

We use, as motivation, a scenario from health-care similar to that introduced in figure 4.1 in chapter 3, where patients' condition is monitored by wearable sensors. A patient's condition is monitored by a network of wearable sensors that monitor his attributes, such as temperature, blood pressure and ECG, as well as his activity, using a 3D accelerometer. Reports are sent to a terminal (e.g. personnel mobile device or stationary node) that produces alerts for a clinic's medical staff.

Figure 6.1 illustrates the application task graph, where the vertices are *starfish roles* and edges represent message exchanges. The application uses three thermometers and accelerometers tasks for enhancing sensing accuracy. Temperature and acceleration fusion tasks collect readings from individual sensing tasks to extract a single reading for each attribute. Accordingly, a *Vitals Fusion* task collects extracted features to infer patient state based on his vitals. Another, independent, branch of the application performs *Activity* recognition from accelerometer readings. Decisions from the two branches are collected by the *Infer Status* role that infers patient status given his activity context. If necessary, the *Alert* task is executed.

Arrows between tasks indicate the flow of messages in the task graph. The application exhibits a hierarchical structure with multiple information fusion centres. Typically, leaf nodes are sampling tasks that encapsulate sensing devices of the network, while intermediate nodes are fusion and processing centres. Even though, for simplicity, the example application is a tree graph, our approach can also be applied to graphs that are not necessarily acyclic or may involve several sinks.

Tasks may have different hardware requirements, which constrains their placement in the network. For example, the *ECG* task must be placed on a node with an ECG sensor. Moreover, if there are multiple sensors available, the most reliable and accurate should be preferred. Another consideration for placement is minimisation of communication cost among nodes. This is achieved either by tasks residing on the same node, thus eliminating the cost, or by placing them in nearby nodes with good communication channels. Finally, we must also consider that certain tasks are mutually exclusive, thus cannot be placed together. For instance, the three *Temperature* tasks should be allocated on different sensor nodes for redundancy. Alternatively, some tasks need to be placed only in specific locations, e.g. one *Accelerometer* task needs to be on a node attached the patient's arm.

Even for simple applications, this task assignment to nodes can be overwhelming for a human administrator. The process does not scale well with the size of the network and becomes impossible to handle manually in a dynamic environment, where components degrade or fail and response time is critical. Processes and tools are necessary that will allow the definition of application requirements for autonomic orchestration and deployment that reacts to component degradation and failure during the system's lifetime.

We assume the existence of a monitoring and fault-detection mechanism, as in chapter 5, that grades component quality. We assume that multiple tasks can be executed on a single node concurrently. Typically, tasks in such systems perform short chunks of processing, e.g. sensor sampling, data buffering and feature extraction. The programming paradigm is event-driven, i.e. tasks are triggered by a network message, timers, sensor readings, etc.

6.2 Extracting Task Graph from Roles

Figure 6.2 demonstrates two policies that are associated with the 'Temperature Fusion' role in the scenario from figure 6.1. The first policy, 'TemperatureAggregation', related to collection of thermometers node readings and temporary storage of these values in a local buffer. The second policy, 'TemperatureFusion', fuses sensor readings by calculating their mean value and passes it to the 'fusion' module of the 'VitalsFusion' role.

```
def TemperatureAggregation
  on Temper.event.onEvt(type, value)
  if type is temperatureReading
  do buffer.PushBack(type, value)
```

Collect temperature readings from nodes with 'Temper' role and store them.

```
def TemperatureFusion
  on buffer.Full(type)
  if type is temperatureReading
  do VitalsFusion.fusion.Update(type, features.Avg(buffer.Get(type))),
       buffer.Clear(type)
```

Fuse temperatures by averaging and propagate value to the 'Vitals-Fusion' SMC.

Figure 6.2: 'Temper' and 'Temper Fusion' roles' policies

The policies in figure 6.2 are a service specification that composes lower-level primitives, i.e. *starfish* modules. There are two kinds of information that can be extracted from this specification: a mission's dependencies on other roles; and a mission's resource requirements, i.e. modules used in the mission. Role dependencies are extracted by remote SMC interactions defined in the policies (see section 4.1).

There are two classes of remote SMC interactions – event notifications and action invocations. In the example of figure 6.2, the first remote interaction is 'Temper.event.onEvt(*type*, *value*)', a remote event notification from the SMC with the role 'Temper' with the local role, which is 'Temper Fusion'. Consequently, a directed edge is created between the two roles as seen in figure 6.1. The edge direction is from the remote SMC to the local as it refers to an event notification.

Similarly, the second remote SMC interaction is an action invocation to the 'VitalsFusion' SMC, which creates an edge between the two roles with a direction from the local to the remote SMC.

Two types of task properties are not included in the mission specification: the number of instances required per task and placement constraints, such as tasks that should not be placed on the same node and tasks that should be placed on a restricted network area. The network administrator can specify these properties in the *starfish* editor.

6.3 Integer Linear Programming Formulation

The task of role allocation in nodes is a combinatorial optimisation problem, reduced to a generalisation of the Quadratic Assignment Problem (QAP), an NP-hard problem, where n facilities need to be allocated to n locations minimising the cost of allocation given a flow between facilities and distances, i.e. weights, between locations. However, there are no co-location constraints in QAP, only a single separation constraint.

We formally define the task allocation problem in Integer Linear Programming (ILP). This entails mapping the task graph $G_t(T, E)$ and its constraints, extracted from policies, to the network graph $G_n(N, L)$. The problem is similar for both the initial and dynamic allocation that occur during the lifespan of an application.

For the graph node sets T and N, of G_t and G_n, respectively, a set of binary variables $x_{t,k} \in X$ is defined, where $t \in T$ and $k \in N$. If variable $x_{t,k}$ is set then task t is allocated to node k. Consequently, the objective function to maximise in the ILP problem is given in equation 6.1.

$$\epsilon = \sum_{\substack{t \in T \\ k \in N}} f(t,k) x_{t,k} + \sum_{\substack{i,j \in N \\ e=(a,b) \in E}} g(e,i,j)(x_{a,i} \cdot x_{b,j}) \tag{6.1}$$

There are two main components in the equation. The first sum is the utility function $f(t, k)$ for allocating a task to a node. The second sum is the utility function $g(e, i, j)$ that represents the cost of allocating two tasks a, b that are dependent in the task graph $G_m(T, E)$, in two nodes i, j of the network. It is possible that $i = j$.

In addition to the objective function, ILP allows the definition of constraints on the allocation problem. We identified three constraints in section 6.1. The first is the unique allocation of a task as expressed in equation 6.2. Every task instance can only be placed on a single node. Note that in the example of figure 6.1 the leaf temperature and acceleration sampling tasks are different instances of the same task.

$$\sum_{k \in N} x_{t,k} = 1, \ \ t \in T \tag{6.2}$$

The second constraint discussed was the mutual exclusion of some tasks that prevents placement on the same node. For a set T_s of mutually exclusive tasks, the constraint in equation 6.3 enforces that they will be spread out appropriately. For instance, in the healthcare scenario such a set of mutually exclusive tasks would be the three temperature sampling tasks, which need to be spread out to use different thermometers for redundancy purposes.

$$\sum_{t \in T_s} x_{t,k} \leq 1, \ \ \forall k \in N \tag{6.3}$$

Finally, there are cases that some of the tasks need to be constrained only to a specific area of the network. For instance, we need to measure acceleration from the sensors attached on the patient's torso, not his arm. In such cases, a set N_c of eligible nodes can be defined by the policy author. The property can be enforced with the constraint of equation 6.4.

$$\sum_{k \in N_c} x_{t,k} = 1, \ \ t \in T \tag{6.4}$$

Next, we define the node and communication utility functions. Utility functions express the benefit of a proposed allocation. The accuracy and availability of the required resources, i.e. sensors, is considered. Additionally, to achieve high quality and reliable measurements the packet drop rates of the communication channels between nodes that perform interdependent tasks are also considered.

The node utility function, defined in equation 6.5, concerns the quality of resources on a node. As discussed before, resources specific to a node are identified by corresponding *starfish* modules. For some resources, such as a sensor, an associated score is attached that is indicative of the quality condition of the node. The scoring scheme derives from a fault detection mechanism similar to the one discussed in chapter 5, which updates resource scores on-line. Balance between readings quality and economical energy consumption remains a consideration, thus, the second component of the utility function favours nodes with higher residual energy levels.

$$f(t,k) = e_k \prod_{r \in R(t)} score(r,k) \tag{6.5}$$

The communication utility function, in equation 6.6, considers $DR_{i,j}$ the message delivery rate between nodes i and j that is measured by the monitoring service of the network. Nodes are not necessarily directly within range and may have to relay messages via intermediaries.

$$g(d,i,j) = min(e_n) \cdot D_d \cdot DR_{ij},\ \forall e_n \text{ where } n \in L_{i,j} \tag{6.6}$$

$L_{i,j}$ is the set of nodes that relay a message from node i to node j. For direct link communication the set is reduced to just two nodes, i and j. Again residual energy is part of the equation. Energy-wise the best node pair selection is the one that involves nodes with high power levels for avoiding task reallocation in the close future. Therefore, the utility function tries to select a pair whose $L_{i,j}$ set has the most residual energy in its most depleted node, i.e. if n is the node with the lowest residual energy ($min(e_n)$ in the path between nodes i and j we favour nodes selection of nodes i, j with the maximum $min(e_n)$.

The original objective function 6.1 provided is not a linear equation due to the $x_{a,i} \cdot x_{b,j}$ product. However, there is a standard way for transforming the problem into a linear one by introducing a set of binary, utility variables Y that replace the product in the equation.

$$y_{a,i,b,j} = x_{a,i} \cdot x_{b,j}, \;\; y_{a,i,b,j} \in Y \tag{6.7}$$

In addition, some new constraints are required that maintain the properties of the replaced binary product. This constraints are presented in equations 6.8-6.10.

$$y_{a,i,b,j} - x_{b,j} \leq 0 \tag{6.8}$$

$$y_{a,i,b,j} - x_{a,i} \leq 0 \tag{6.9}$$

$$x_{a,i} + x_{b,j} - y_{a,i,b,j} \leq 1 \tag{6.10}$$

The simplex algorithm, used in LP-solvers to find the optimal solution, tries to minimise the objective function, instead of maximising it as in equation 6.1. There is, however, again a straightforward transformation of the function to achieve this.

6.4 Heuristic Approximation

The simplex algorithm in linear programming problems gives an optimal solution in polynomial time. However, when the variables are constrained to be integers, as in the formulation above with the binary $x_{t,k}$ variables, the problem becomes NP-hard. Consequently, the solution is not viable as the network size or the complexity of the mission increases. The problem is essentially a combinatorial optimisation selecting T binary variables from a set of $T \times N$ variables. The solution is a combination of those variables under the constraints discussed before. An exhaustive exploration of all combinations requires exponential complexity of $O(2^{T \cdot N})$.

Heuristics can be used to approximate the problem solution in polynomial time. They, typically, involve either random sampling of solutions or stepwise improvements on a selected solution. The former is inefficient as it does not provide any gradual improvement over-time. The latter includes hill-climbing approaches, where the algorithm tries to make small changes to a randomly selected solution to transition to a better one. This method is usually trapped in local optimum solutions when the problem space is not convex.

Simulated annealing [Cer85], shown in figure 6.3, is a general *meta*-heuristic algorithm that combines the two approaches. It tries to emulate atom behaviour during the cooling of metal materials that try to reach a balance state by making jumps and releasing energy. Jumps are more frequent at high temperatures. As the material cools down, the jumps get less probable and the system, as a whole, reaches an equilibrium state.

```
$K \leftarrow 1$
$S \leftarrow$ random solution
**repeat**
  **for** $i \leftarrow 1$ **to** $M$ **do**
    $S_i \leftarrow$ transition$(S)$
    **if** $C(S) \geq C(S_i)$ **OR** $rand(0,1) < e^{\frac{C(S)-C(S_i)}{K}}$
      $S \leftarrow S_i$
  $K \leftarrow c \cdot K$
**until** $C(S)$ does not change
```

Figure 6.3: Simulated annealing algorithm

Simulated annealing uses a temperature parameter, K that decreases over time with a rate $c \in (0,1)$. Like hill-climbing, the algorithm tries to transition to a better solution, one with lower cost $C(S)$, by making small adjustments to the existing one. While the temperature is high, there is a relatively high probability that the algorithm will progress to a solution with a higher cost than the current one. Consequently, this will eventually allow it to escape from a local minimum to another area. As the temperature drops, it becomes less likely to make bad transitions and eventually the termination condition is reached. More frequent transitions at the initial rounds allow the algorithm to search a larger problem space until it, finally, reaches an area that it will explore exhaustively. An optimal solution is not guaranteed, but in practice a very good estimation is obtained, having explored a broad spectrum. The simulated annealing process in this case looks for a permutation of the vector in equation 6.11.

$$V = \{x_{t,k} \mid t \in T, k \in N\} \tag{6.11}$$

The cost function of a solution is derived from equation 6.1 in the original ILP formulation by transforming the utility to cost functions. The algorithm makes the transition by randomly switching a variable of vector V in every step, i.e. a role assignment to a node. A transition is first asserted against constraints 6.2-6.4 before being considered as an eligible solution. It should be noted that the objective function does not need to be transformed to a linear equation any more.

The complexity of the heuristic is not directly dependent on the input and is capped by the parameters of steps and rounds of the algorithm. While there is no time guarantee, in practice it only takes a few rounds for the algorithm to stabilise and provide a solution.

6.5 Evaluation

In this section we experimentally evaluate our approach to autonomic dynamic reconfiguration of the network. Initially, we consider the ability of the simulated annealing heuristic to approximate the optimal solution. Later, we attempt to quantify the benefit of the reconfiguration in terms of sensor data quality and packet delivery. Finally, we compare the life-time performance of our approach to more power-saving focused methods that appear in the literature. For the evaluation, we use a realistic sensor network application specification, as was described in section 6.2. In addition, we use larger examples of randomly generated task graphs, in order to investigate how the approach scales on larger networks and missions.

6.5.1 Quality of the Solution

Initially, we examine how closely the heuristic solutions compare to the optimal. For the ILP problem solving, we used the GNU Linear Programming Kit[1], while for the simulated annealing we implemented a prototype in Python. Even though the simulated annealing code runs in an interpreted language, the execution time is in the order of a few seconds per round, while the

[1]http://www.gnu.org/software/glpk/

execution time for the native implementation of the ILP solver requires several hours (or days) for large mission sets.

Initially, we map the health-care scenario to a wearable network of eight nodes. Each node is equipped with a different set of sensors so that there are four thermometers, four 3D accelerometers, two ECG and two blood pressure monitors in total. We use randomly generated task and network graphs. Sensor are assigned a random score and nodes are connected with asymmetric links that are also assigned random packet drop rates.

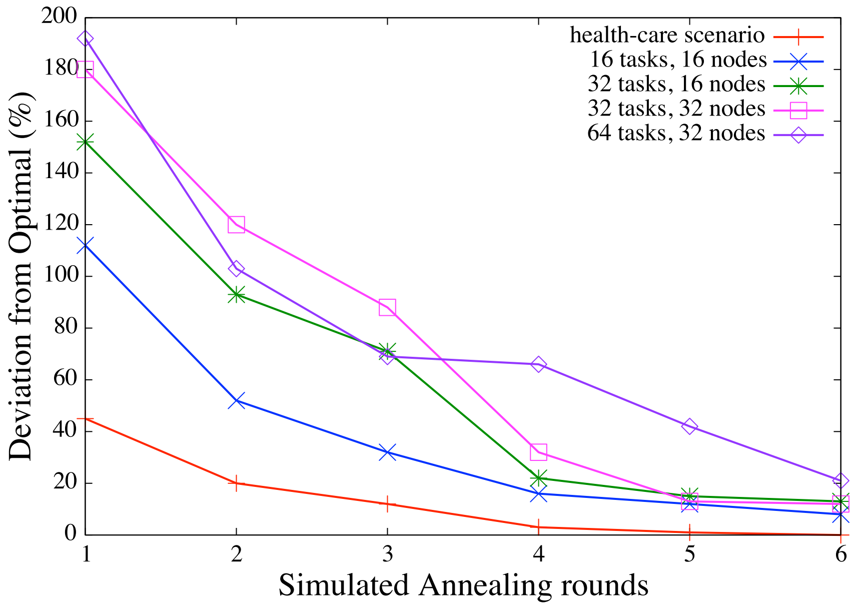


Figure 6.4: Simulated annealing approximation quality

Figure 6.4 compares the solutions provided by simulated annealing in relation to the optimal solution. In the health-care scenario, consisting of 14 tasks on 8 nodes, simulated annealing usually approaches the optimal solution only in a few number of rounds. Similarly, we try the same test for randomly generated graphs of different sizes and we observe that the simulated annealing can approach very close to the optimal solution, typically in 5 to 6 rounds.

We limit the experiments to problems with 64 tasks and 32 nodes as computation time for the ILP becomes impractical with increased size. However, the trend demonstrates that simulated annealing is capable of providing similarly good solutions as the problem size increases.

6.5.2 Quantify Benefits

We now quantify the benefits of dynamic adaptation and task reallocation at runtime, where sensor and node link quality metrics vary during the simulation. Runtime reallocation implies transmission of tasks over the network. In the case of Starfish, this is a relatively cheap process, as tasks, which are encoded in compiled obligation policies are typically between 20-100 bytes. However, in other frameworks, task migration might involve transmission of much larger binary images for nodes. Consequently, constant relocation of tasks can be inefficient or even impractical.

We consider a more conservative reallocation scheme to reduce this overhead; a partial reallocation of tasks, whose quality metrics have dropped beyond a threshold. We study how these two approaches affect overall service quality in the system. More specifically, the dynamic, tabula-rasa, approach considers reallocation of all tasks in every round given updated metrics from the network. The partial, conservative, approach reconsiders only allocation of tasks whose operation is hindered by significant degradation of resources.

In order to study how drop rates are affected by neighbouring nodes and cross-traffic, we use the Castalia simulator [PPB07]. Castalia provides an accurate radio and wireless channel model, modelling the error-prone behaviour of low-power wireless links, which is common for wireless sensor networks. Node deployment in the network is random, but it retains a connected graph. We use a simple sensor quality classification model with three possible states: *accurate*, *degraded* and *faulty* with numerical score 3, 2 and 1 respectively. Each sensor may deteriorate at any round with a small probability. Finally, node energy consumption is modelled to increase with the number of tasks allocated in order to capture computational and communication costs.

We ran a test of 250 rounds comparing the two adaptive approaches and a static deployment that does not migrate tasks after the initial allocation. In the conservative (i.e. partial reallocation) approach, a task is migrated from a node if one of its required resources becomes *faulty*, the communication link's delivery rate with a dependent task drops below 50% or its node has disappeared from the network due to power depletion.

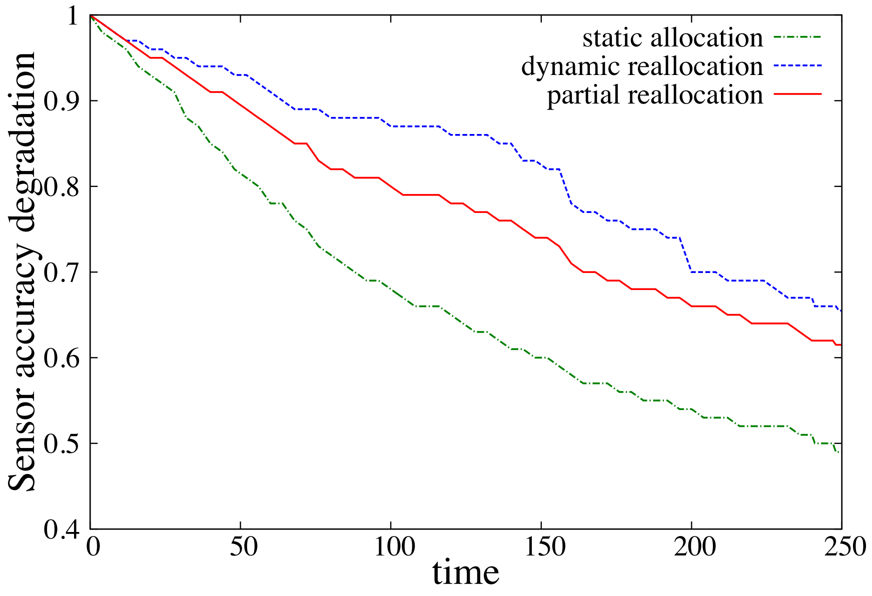


Figure 6.5: Sensor readings quality degradation in the network

Figure 6.5 presents the average sensor data quality degradation compared to the initial allocation over multiple simulation runs. The static, non-adaptive approach degrades quickly, unable to take advantage of the remaining high accuracy sensors. The dynamic approach provides the slowest degradation, immediately adapting to sensor degradation. Partial reallocation follows a similar, graceful degradation, but adapts slower to changes being frugal on task movements.

Similarly, figure 6.6 shows the average delivery rate of messages to nodes in relation to the initial

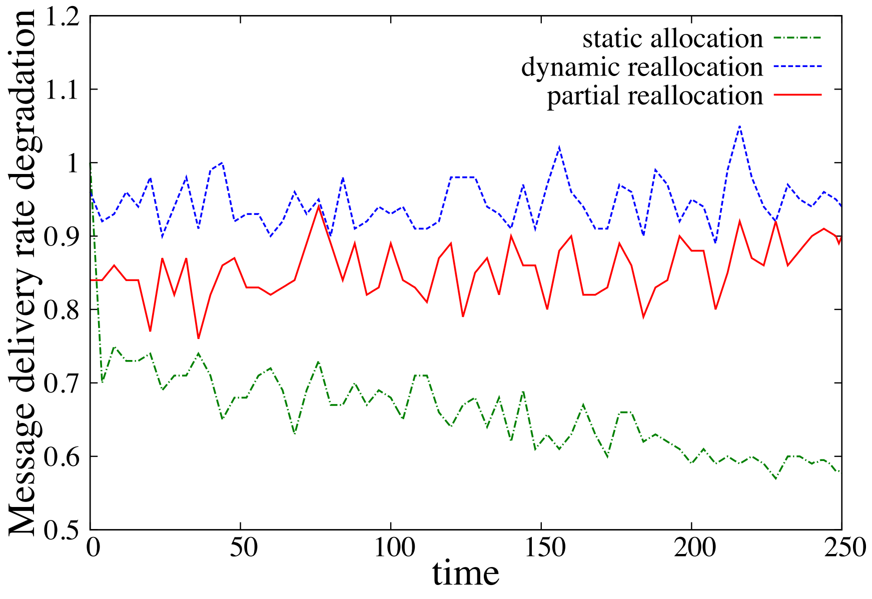


Figure 6.6: Node link quality degradation in the network

allocation. In all scenarios, the link quality drops from the initial allocation as a result of the traffic that the application introduces in the network. Message transmission affects drop rates in a node's neighbourhood. As can be seen, the static approach exhibits a sharp drop initially followed by further decrease at a slower pace fro the ramining of the simulation. Both adaptive approaches, while still affected by collisions in the wireless medium, manage to maintain a stable trend on their delivery rates.

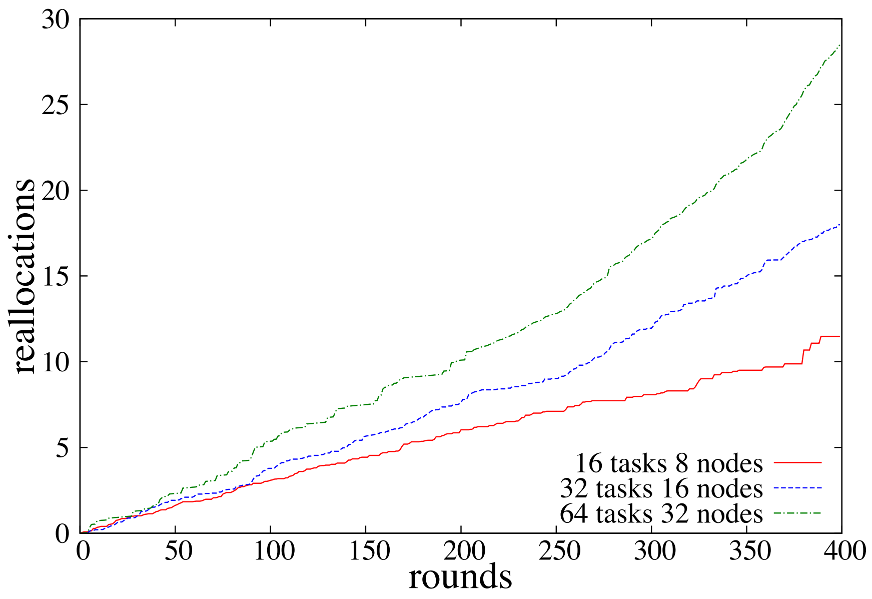


Figure 6.7: Cumulative task migrations in partial reallocation scheme

Finally, figure 6.7 shows the cumulative number of average role migrations in the case of partial reallocation for problems of different sizes. For reference, the tabula-rasa reallocation on every round results in migration of almost every task, as it is very sensitive to subtle changes resulting in task thrashing between the network nodes. Partial reallocation significantly limits the amount of task movement inside the network making it a more viable alternative, while at the same time provides smooth service degradation on component failures. Even as the network size increases the number of migrations does not increase drastically. An increase in the rate as tasks are reallocated, however, can be noticed as the application grows older and nodes start running out of power.

6.5.3 Comparison with Energy Focused Deployment

Our model tries to find a trade-off between quality metrics and energy consumption. Consequently, we examine how the network's lifetime is affected compared to a model that only considers task allocation based on the nodes' residual energy. We compare our approach to the objective function used in [PP10]. We modify our original cost function pair in the objective equation 6.1 to only reflect the power consumed by computation and communication load imposed on a node by each task. Furthermore, we examine a third model where energy consumption is completely ignored in the utility functions and task assignment is solely based on the quality metrics.

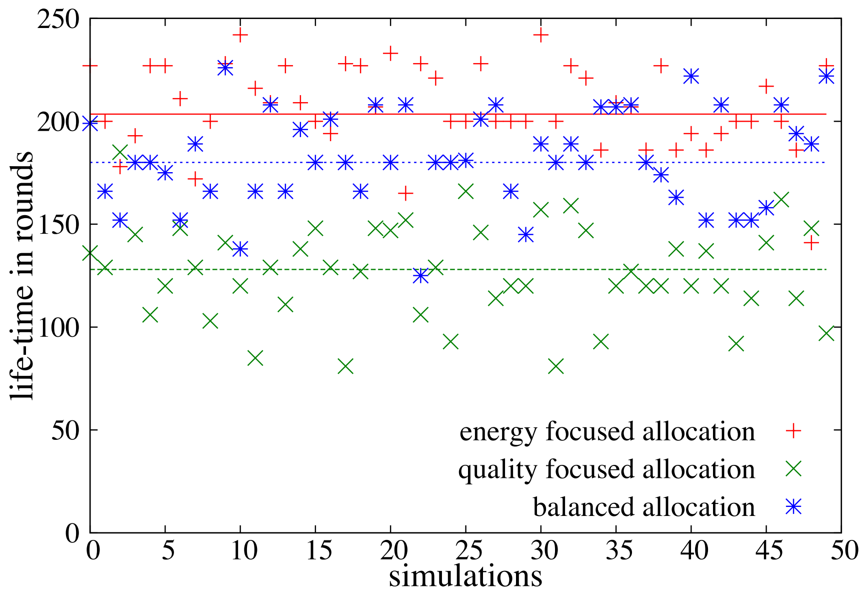


Figure 6.8: Application's lifetime based on task allocation scheme

For every model, we run 50 simulations with random task and network graphs of random sizes. For all simulations, we use the partial migration scheme. As expected, the balanced approach, followed for all previous simulations, falls in-between the two extremes as shown in figure 6.8. The figure depicts the running time in rounds of different simulation instances. The horizontal level-lines denote the median network lifetime for each approach. Lifetime fluctuates in each model due to the random nature of graphs and manifesting faults. However, it is clearly observable that the energy focused allocation supports longer running applications than the other two models. More specifically, the average running time for the energy focused allocation is 206 rounds, 127 rounds for energy unaware allocation and 182 rounds for the balanced approach.

Consequently, the energy focused model provides 62% lifetime increase compared to an allocation scheme that ignores energy constraints. On the other hand, the balanced cost-function pair increases the average lifetime of the application, while it degrades more gracefully as components start to fail in the network. It sacrifices roughly 12% life-time on average, compared to the pure energy aware model, while it improves quality metrics at the same time.

6.6 Related Work

In [AGGB10] the authors formulate the problem of task allocation as an ILP problem, similar to what we attempt later. However, the objective function in that case only accounts for minimising power consumption in nodes. The authors account for computation costs, selecting voltage power for CPU, and exchange of messages. They use LU factorisation and Fast Fourier Transform as applications to test the system. A similar approach is also presented in [TEO05], however, the authors make the assumption that only one task is assigned to a node at a time, which resembles batch processing rather than pervasive applications. Similarly, Johnson et al. [JRP^{+}10] consider single task allocation per node using utility and requirement functions for selection of nodes, but also examining dynamic scenarios where new tasks are introduced during the network lifetime. Integer linear programming (ILP) has been used [PS03, PP10, AKF^{+}10] to optimise the energy cost of the network, while other approaches consider failures that are caused by battery depletion [CGC09, ZLG07] by dynamically adapting their configuration when a node reaches low battery levels to extend network lifetime.

Other approaches include heuristics or agents-based auctions [XLTD09, SYH09]. In [EXT^{+}09] a market model for nodes is built, where clients bid for a task based on local knowledge of their resource availability. Bids from nodes are evaluated centrally by the node that offers the task on the market. GAP-E, a knapsack approximation algorithm, is also used to model sensor agents that bid for tasks [LNV09]. Lie et al. [LBBL10] collect task satisfaction from competed tasks and employ an admission control mechanism based on available capacity of the network. Salazar et al. [SRAA10] introduce a distributed algorithm, where nodes are collectively searching for a solution to the combinatorial problem of node configuration. Nodes diffuse good solutions to neighbours until the network stabilises. The approach, however, assumes each node has knowledge about other nodes' possible configurations. The expensive solution-search operation takes place on power constrained nodes.

Task allocation approaches in the literature aim solely for a network set-up with minimal energy consumption or prolonged lifetime. Our contribution is integration of component failure and degradation in the task allocation decision mechanism for self-healing of the network. We consider optimisation on reliability and quality of collected information. This entails both high delivery rates among nodes and high quality of sensed data. An economical task allocation indirectly implies also a reliable system, where nodes operate for longer periods and sensors function optimally. Consequently, we attempt to achieve an allocation that balances both information quality and energy consumption.

Furthermore, most approaches focus on long running tasks in WSNs similar to traditional distributed systems. We argue that this model does not fit modern pervasive applications that are predominantly data-driven; typically, monitoring application and data aggregation processes. Nodes are responding to events that involve execution of brief, concurrent tasks and transmission of several messages. Our second contribution is a task-allocation model more appropriate for pervasive systems that is directly extracted from the service specification. Our mechanism reasons about task assignment in terms of roles and missions of the *starfish* framework. A task is a *starfish role* and the two terms are used interchangeably throughout this chapter. As a result, tasks, dependencies and their requirements are automatically extracted from the *roles* and the policy code in their *missions*. Task dependencies are dynamically updated when the administrator modifies them without requiring manual update of separate models.

6.7 Summary

In this chapter, we presented a mechanism for autonomic role allocation to SMCs and on-line adaptation in pervasive systems that consist of several networked components. We discussed how a task graph of the deployed service is extracted directly from its policy specification in the *starfish* framework and consequently we transformed the task graph into a linear programming problem so we could formally analyse it. The linear programming problem has integral constraints on the variable and ILP problems are NP-hard. Consequently, an approximation method is proposed using simulated annealing, a general *meta*-heuristic method, which searches a broad spectrum of the problem space. We evaluated the benefits of on-line adaptation in terms of service life-time and impact of degradation in sensor readings and communication links.

Chapter 7

Network Adaptation

Chapter 6 has dealt with adaptation of service deployment as a consequence of node failures (e.g. energy depletion), sensor failures (e.g. noise or drift) or link degradation. In large-scale area networks nodes are usually unable to establish direct connections with all other nodes and rely on multi-hop communication where intermediate nodes forward messages from a producer node to a consumer. These networks may be susceptible to various communication disruptions such as connectivity loss due to unreliable links, packet drops due to noise on the wireless medium or high-volume of traffic overloading links and network buffers.

While many of the faults can be attributed to random events, some of them exhibit specific repeating patterns caused by periodic events in the environment, such as day-night cycle of nearby electrical equipment, movement of inhabitants or vehicles in the environment generating noise or affecting signal paths. Periodic events detected by multiple nodes in the sensor network may result in increased traffic within a region of the network leading to congestion and possible message loss. Finally, in hostile environments, causes may include adversaries that try to compromise communication.

We look into recurring network connection faults and propose an effective way to forecast repetitive patterns using time-series analysis to avoid affected areas. We propose a novel application-level, autonomic routing service to adapt sensor readings routes to avoid areas that are expected to have low link-quality and prevent overloading good quality routing paths. The routing service maintains forecasting models for each link performance metric and decides route allocation to active network paths that match the delivery requirements in the short future. Our work is motivated from errors observed in a large scale deployment in a desert environment. The approach is focused and integrated as a prototype in the ITA-developed Sensor Fabric [WGB+09], a sensor network middleware that provides sensor identification, discovery, access control interoperability, data dissemination and management of sensor nodes predominately focused on military applications. We describe the use of the extension mechanisms of the Fabric to collect real-time network information on node availability, link packet drop rates and traffic loads in order to select the routes that maximise the likelihood of message delivery across the network over an unreliable multi-hop network. Finally, we present performance evaluation results obtained from simulated environments to assess the effectiveness of proposed network failure forecasting feature.

7.1 Adaptive Routing with Forecasts

In this section we describe our approach for predicting recurring link failures and packet congestions in the network. We discuss performance metrics collected in our approach, the forecasting mechanism used and, finally, route selection.

7.1.1 Performance Metrics

We measure the performance and reliability of the network by collecting a set of application-layer metrics from nodes, which allows the approach to be independent from the underlying network. We account for node availability, drop rate of network links and traffic characteristics of feed subscriptions. Based on these attributes, we build forecasting models to update multi-hop message routes.

Node availability is not a symmetric relation between a pair of nodes. Wireless links are usually asymmetric, having as a consequence that if node A is directly reachable from node B the opposite is not necessarily implied in a wireless sensor network. Periodic beacon broadcasts can be used to verify a node's presence to its neighbours. In order to conserve battery power, nodes do not constantly listen for incoming messages. Instead, duty-cycling is a very common process, where nodes turn-on their radio only for certain periods to allow for message exchange. Due to the inevitable bursts of traffic and synchronisation issues of the process we allow for a threshold of consecutive messages that can be missed before a neighbour is considered unavailable.

Apart from node availability, we also consider the quality of the wireless links, based on measured packet drop rate (PDR). We measure PDR by piggybacking sequence numbers on messages at each hop. The approach has the advantage of being inexpensive requiring only to append a few extra bytes on existing traffic, hence, minimise energy overheads. However, there are drawbacks to the approach. First, there is a non-bounded delay for metric updates. In case no messages are received by a node, either lack of traffic or a large number of dropped messages can be inferred. On the other hand though, the beacon messages between nodes allow to disambiguate between link failures and lack of traffic and also sets an upper bound on the update delay. In case of low underlying traffic, sampling of the underutilised links is weak for statistical inference. To compensate for statistical bias, additional low frequency control messages can be introduced over low-traffic links to sample their status, if necessary. Furthermore, we introduce a confidence level on the link quality metric. The confidence level is a real number value in the range $(0, 1]$ that quantifies the statistical confidence on the observations for link PDR, based on the number of packets that have been relayed over the link. The confidence is the fraction of a minimum acceptable number of messages, c, that need to be relayed over the link in order to have a reliable metric on the link quality. We cap the confidence level to 1, even when a link accommodates more than c messages. We choose to follow a more lax and less taxing approach of counting sequence numbers instead of ACK or NACK messages due to the severe overheads they introduce. However, the use of confidence factor on link quality compensates to a certain extent for their weaknesses.

With regard to network traffic, nodes monitor the volume of traffic that they relay and the volume of messages they produce. Messages originating from other nodes, passing through intermediaries count as a relayed traffic, while messages generated from a local platform attached to the node are considered originating traffic. Originating traffic is unavoidable, whereas relayed traffic could be rerouted to bypass the node in case of overload. They are both used to train a prediction model on future message volumes.

7.1.2 Forecasting Mechanism

Three forecasting models for different metrics of the network are used to produced two network prediction models – link quality and the traffic load graphs. The first metric caters for recurring isolation of nodes, the second considers the packet drop rates between node links and, finally, a forecast model is used for predicting the traffic volumes that sensor feeds produce.

We consider these metrics as a time-series and use a fitting model that describes their behaviour. We have selected the Holt-Winter Additive Seasonal model that captures trends in addition to periodic effects in time-series. Holt-Winter applies exponentially decreasing weights on historic data to update the model. It decomposes the time-series in three components; the level S_t, local trend b_t and the periodic factor I_t. Each of these components are updated incrementally (on-line) using exponential smoothing. Forecasts in the model are calculated as a linear combination of the aforementioned components, shown in equation 7.1, where t is the current time instance, m is the units in the future for the prediction and L is the period of the time-series.

$$F_{t+m} = S_t + b_t * m + I_{t+m-L} \tag{7.1}$$

We use the IBM Watson Forecasting library (WatFore) to construct and manage the forecasting models. The WatFore library provides a fully automated, extensible and scalable streaming predictive analytics framework, suitable for monitoring any type of performance KPIs (Key Performance Indicators). It implements a number of streaming algorithms (including the Holt-Winters Additive Seasonal) in a streaming fashion that does not require permanent storage of historical performance measurements, thus bounding memory requirements for maintaining and using forecasting models. This is particularly important in our application domain, as sensor platforms cannot be assumed to have large storage capabilities solely for performance monitoring purposes. Furthermore, the incremental updates to the forecasting models with newly obtained measurements from continuous monitoring minimizes the processing requirements for keeping the models up to speed, imposing only marginal overhead to the sensor platform. The library also provides methods for calculating the periodicity of the performance metric using Fourier analysis, and automatic training of the forecasting models once enough data measurements have been collected.

7.1.3 Subscription Routes Selection

Selecting subscription routes from constructed models is not trivial. We construct a link graph $G_R = (V, E_R)$ of the network, where the vertices V are the network nodes, and edges E_R are the direct links between them. Edge weights represent the expected failure rate between node pairs. Weights are calculated as a linear combination of node availability and the product of link PDR and the confidence level of the metric. The graph G_R essentially represents a map of link health in the network. Applying a shortest path algorithm on G_R between the producers and the consumers of feeds, gives a prediction for the most reliable route, i.e. the one that is less likely to drop messages in the near future.

In event-driven, multi-hop sensor networks manifestation of a monitored event results in bursts of traffic from sensors in the network, due to the increased frequency that monitored attributes in the affected area are changing. Bursty traffic is bound to increase packet loss due to transmission collisions and congestion in nodes' network buffers. In such cases, it is preferable to diverge high traffic flows to use different nodes for relaying messages. Consequently, we use the information on the traffic volumes that a sensor generates to separate high volume flows over different paths. To prevent congestion, we enhance the routing map G_R generated based on link qualities, using the expected traffic of the channels in order to prevent overloading healthy channels with too many subscriptions. To achieve this we increase link costs on graph G_R by a proportion of the overall traffic they expect to carry which penalises high traffic links. In order to determine the load of a link, we normalise the number of packets that are expected to traverse the link based on allocated feed subscriptions. All loads are expressed as a proportion of the link with maximum load. However the actual link utilisation is not known, so this could result in penalising links with low utilisation which carry a relatively high percentage of traffic even though the total traffic is quite low. In order to resolve this issue, the administrator can specify a threshold above which the congestion prevention algorithm would start.

Finally, the intention is to avoid routing traffic through congested links while avoiding throttling links with low to medium utilisation that can carry more traffic. Thus, instead of a linear scale on link cost penalties, we use an exponential scale so that penalisation will mostly affect the costs of highest-traffic links of the network, which are also the most likely to exhibit congestion.

7.2 Case Study: ITA Sensor Fabric

We perform a case study by applying our approach to ITA Sensor Fabric [WGB$^+$09], a sensor network middelware designed and implemented by IBM, that supports multi-hop communication used in real-world deployments.

The Fabric middleware is a network management layer that connects assets in a sensor network to clients/actors providing a publish/subscribe communication abstraction [EFGK03b]. Sensors act as publishers providing data feeds of raw or processed sensor readings. Client nodes are the consumers of this information and can subscribe to sensor feeds to receive readings as they become available. There can be multiple subscribers to published messages and publishers are not aware of the identity or address of the subscribers, i.e. there is a decoupling between publishers and subscribers. Clients refer to a directory service to locate potential messages types of interest that they subscribe to. The Fabric infrastructure matches publications of these messages to subscriptions and sets up routes over the multi-hop network for relaying messages to the subscriber.

Fabric supports multi-hop communication among nodes in the network while abstracting details of their location from the application developer, who perceives the existence of a fully connected network. Fabric provides the abstraction of a communication bus, where nodes can publish information, i.e. sensor feed readings, that eventually reach consumers that are subscribed to these feeds. Fabric builds an expandable platform of assets, where producers of information, e.g. physical or even virtual sensors, generate data that consumers, e.g. fusion centres or applications, subscribe to without imposing a single endpoint/sink in the network.

Fabric middleware uses virtual circuit routing instead of connectionless datagrams as it targets military environments, where all nodes are not equally trusted. Routing selection is also affected by administrator policies that are enforced by a policy management system that dictate whether a data subscription can be relayed by certain nodes. This would be more complex to do with connectionless datagram routing requiring per hop decisions instead of a decision at the set-up time. Subscription routes are locally cached on nodes. When routes are updated in Fabric Registry, nodes do not immediately update their current routes. Instead, nodes update data subscription routes only when they break due to a link failure or bad reception rate that degrades below a predefined threshold. Then the producer node sets-up a new subscription path using the updated route from the Registry.

Sensor data feeds are identified using globally unique names that consumers, i.e. subscribers, can refer to and receive produced data. Information for available resources and assets, as well as real-time metrics on network status, are stored in a distributed database, the Fabric Registry.

7.2.1 Fabric Components Architecture

Figure 7.1 presents the architectural components of Fabric. We provide a brief description of these component introduce the terminology used in the remaining of this chapter.

Registry is a distributed Gaian database[1] operating on Fabric nodes. It contains all information on network state including node IDs, physical location, neighbouring sets, assets, registered data subscriptions and virtual circuit paths between nodes. The database is distributed among a subset of Fabric nodes, each maintaining local data. Information retrieval happens as a query that collects data from nodes that eventually get propagated to the request point. Registry communication can take place over a secondary low-traffic link that is not subject to our

[1]http://www.alphaworks.ibm.com/tech/gaiandb

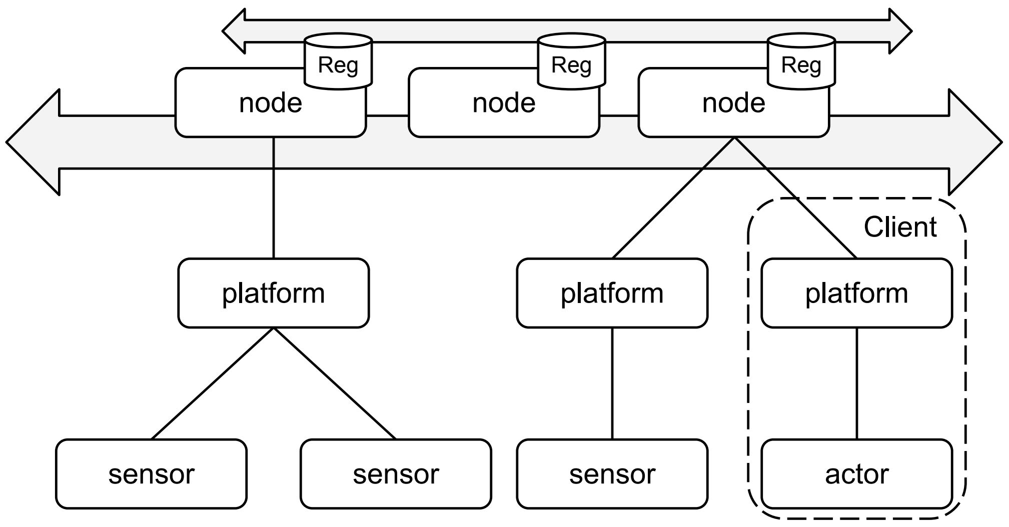


Figure 7.1: Fabric component architecture

mechanism as it is considered more reliable, due to the sparsity of data on it.

Node is the network endpoint of Fabric, which runs a Fabric Manager service. The Fabric Manager provides multi-hop communication and the publish-subscribe service. Fabric nodes are different from typical sensor nodes that are considered to be resource constrained devices. Instead they run a Java runtime with the processing capabilities similar to a netbook. Fabric deployments employ heterogeneous devices; ranging from unattended sensor nodes to aerial unmanned vehicles. In addition, Fabric nodes maintain a part of the Fabric Registry that stores local runtime information and are the extension points in Fabric as discussed later.

Platform is an adaptor that connects sensors and actors to a Fabric node. Platform is the equivalent of what is usually considered a sensor node in the literature, a small, constrained device with low-power radio running on batteries. In spite of being logically a separate component to the Fabric node, a platform could also reside on the same physical device.

Sensors are attached to platforms and are the producers/publishers of information in the network. They provide feeds of data that actors can subscribe to in order to receive readings updates. A sensor may encapsulate a hardware sensing device or it can be a virtual device that produces information by consuming feeds from other network endpoints, i.e. a fusion centre.

Data Feeds are series of values produced by sensors. One sensor may provide multiple feeds, for example two separate resolution feeds from a camera or a feed with raw thermometer readings as well as their averages.

Actors are either human users or software services. Similar to sensors, they have a unique identifier that allows the middeware to route information towards them.

Client is a virtual entity, consisting of an actor and a platform through which it can interact with Fabric.

7.2.2 Extension Infrastructure

The Fabric core provides a minimum set of services required to implement a distributed communication bus service, while maintaining a small footprint and overhead in the system. Additional capabilities are introduced as plug-ins, grouped into extension families. An extension family is a user-defined collection of plug-ins that share data and management operations. Fabric allows for three types of plug-ins; *Message Plug-ins*, *Fablets* and *Services*.

Message Plug-Ins

Nodes process messages as they are relayed by Fabric on each hop. *Message Plug-ins* are modules that can be attached to a node's Fabric Manager to process messages directly. There are three sub-types of Message Plug-ins: node, task and actor – allowing filtering of messages that are related to any of these. Their life-cycle is managed by the Fabric Manager and they are, typically, short-lived operations, such as policy enforcement, filtering, transformation, logging, caching and encryption, without the ability to have side-effects outside their controlled environment. Plug-ins can be registered to operate either on incoming or outgoing messages of a node allowing messages to be decrypted, processed and encrypted again using different plug-ins.

Within the Fabric Manager, the *Registry* contains information about each data-feed that flows over the bus. This includes what tasks it is part of, where it was generated, who is subscribed to it and the characteristics of its destination actors. This information is available to message plug-ins as they are applied to each individual data-feed message.

Fablets

Fablets are extensions that run on nodes independently of the message flow. They run in separate threads, managed by the Fabric Manager, and are more flexible than Message Plug-ins allowing a broader range of operations. They can directly access Fabric resources such as the Registry and the publish-subscribe bus, but also other non-Fabric resources such as storage devices or application databases. Typical uses of Fablets include accessing non-Fabric resources and platforms, or implementation of data fusion algorithms.

Fabric Services

Fabric Services are the mechanism used to implement most high-level Fabric features, a modular approach that builds on Fabric's core message passing functions. Services are complementary to other plug-ins. They are separate processes that work on the side and can be attached to Fabric though the Actors mechanism to interact with the node's local bus. For instance, Fabric's sensor subscription service is implemented to provide sensor data feeds as a service on top of Fabric's core features: communication bus, the Registry and event handling.

7.2.3 Integration into ITA Sensor Fabric

We prototyped a forecasting routing service and integrated it into the ITA Sensor Fabric as an extension family. We discuss synergy among developed plug-ins and implementation details for the subscription virtual circuit decision mechanism. Figure 7.2 gives an overview of the adaptive *Forecast Routing* service architecture for Fabric.

The node availability metric is already provided in Registry by the Fabric Manager's *Discovery Service*. However, we have implemented message plug-ins to measure link quality as well as

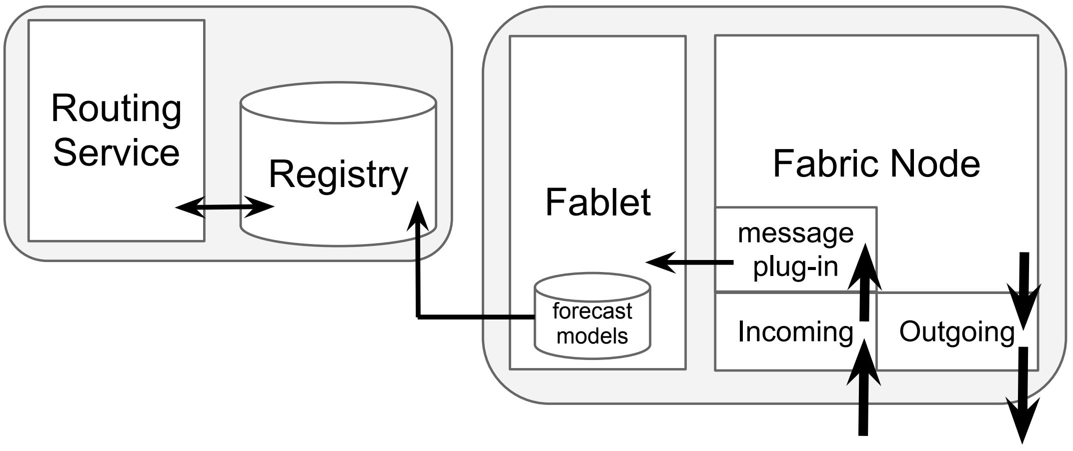


Figure 7.2: Fabric plug-in architecture

generated and relayed traffic. Three message plug-ins have been prototyped for measuring link packet drop rates and message traffic load. An *outgoing* message plug-in at the transmitting node inserted messages sequence numbers related to a node pair, while an *incoming* message plug-in at the receiving node checks the sequence number to verify whether any messages have been lost from that link. The message plug-in system in Fabric permits piggybacking information on messages as an extension, without needing to modify the underlying feed subscription service. A third message plug-in monitors a node's local publish/subscribe bus for feed messages and counts them per time unit to quantify traffic of the node.

Message plug-ins are expected to be short-lived and avoid the use of external resources such as hard-disk writes or network communication as this would have a performance impact on the number of messages a node can process. Thus, message plug-ins write information extracted from messages to a Fablet that is running alongside the Fabric Manager on the node. Fablets, being separate threads, have their own execution flow control and memory storage. They collect information posted by local message plug-ins and use it to update the forecasting models they maintain. The link quality and traffic load forecasting models in the Fablet periodically update the distributed Fabric Registry with new values for monitored attributes. Although the forecasting model incorporates information from all samples collected throughout a system's lifetime, it is relatively small in size – in the order of a few kilobytes. As a result, it can be serialised and stored in the Registry in a binary format. Updating forecasting models locally, rather than close to the Registry, significantly reduces the communication overhead introduced, compared to propagating observations to a sink to perform forecasting model update outside the network.

The *Routing Service* pulls forecasting models from Fabric Registry to update its routing paths. After the forecasting phase of the algorithm, it updates the subscription routes table in the Registry used by nodes when they need to deploy new subscriptions.

7.3 Evaluation

For the evaluation of the forecasting effectiveness on route selection we emulated network scenarios that we consider fit well with expected periodic failure error classes in sensor networks. We initially evaluated the effectiveness of the algorithm for coping with node and link failures and then considered congestion effects in high-traffic networks. In all scenarios, we use a grid layout, where nodes can directly communicate only with its immediate neighbours. Hence, most nodes can send messages directly to 8 neighbours while nodes at the corners are limited to 3-5 neighbours, depending on their position.

Feed subscriptions, as in the ITA Sensor Fabric framework, may set-up from any point of the network. Hence, there is no single sink in the network, but there are multiple subscribers that consume data from producers. Subscribers may be terminal recipients of information or may in turn produce new data to be consumed by other nodes, after processing their input feeds. This creates an environment where information does not have a single flow among nodes. We randomly generate feed subscription in the simulated network set-ups that are examined in this section.

We compared three routing approaches in the simulations. The first one is the static paths that are currently implemented in the ITA Sensor Fabric framework. This is a naïve approach that provides a lower bound of network performance – an indication of the impact of failures in the network, as it is unable to respond to them. The second approach is dynamic adaptation of routes based on the metrics discussed. However, instead of forecasting future values, route adaptations are based on recent observations. Essentially, this approach performs adaptation based on current network status. Finally, we make use of future predictions of metric values, by projecting from historic data using the Holt-Winter additive model provided by the IBM WatFore library, to dynamically adapt routes in Fabric Registry.

7.3.1 Periodic Node Communication Failures

We first studied the accuracy of forecasting fail-stop communication link failures inside the network. We emulated a 5×5 network grid where 26 subscription are placed among nodes randomly. Sensor feeds produce data regularly in random intervals between 1 to $10sec$. In every run, 8 nodes, roughly 1/3 of the population, experience periodic failures that cause them to disappear from their neighbourhood for random time intervals. Failure times and duration are selected from the range 10 to $50sec$, with an average close to $20sec$. For this particular scenario, we assume that links between nodes are ideal and do not drop packets due to noise, in order to study only the effects of node disappearance. We emulated the scenario running Fabric on desktop clients where different nodes run on separate virtual machines. We emulated communication failures by editing linux iptables[2] to add rules that drop packets from nodes to isolate them.

Figure 7.3 shows the overall packet delivery rate achieved in the network, as an average of several experiments, with three different approaches mentioned earlier; static routes (SR), adaptive current routes (ACR) and adaptive forecasting routes (AFR). The static routing achieves a 74% packet delivery rate, which we consider as the lower bound because there is no effort to adapt to node failures. The dynamic selection of routes based on recent observations improves the rate close to 85% while forecasting outperforms both, reaching a 95% packet delivery rate. Even though ACR is able to adapt to nodes that have a longer uptime phase based on recent observation, its decisions quickly become outdated. However, AFR by projecting these values in the future achieves better adaptation of the routing schemes as it is able to predict which nodes are going to be available in the next rounds.

As shown in figure 7.3, the AFR method exhibits similar performance with SR. The dive in

[2]http://www.netfilter.org/

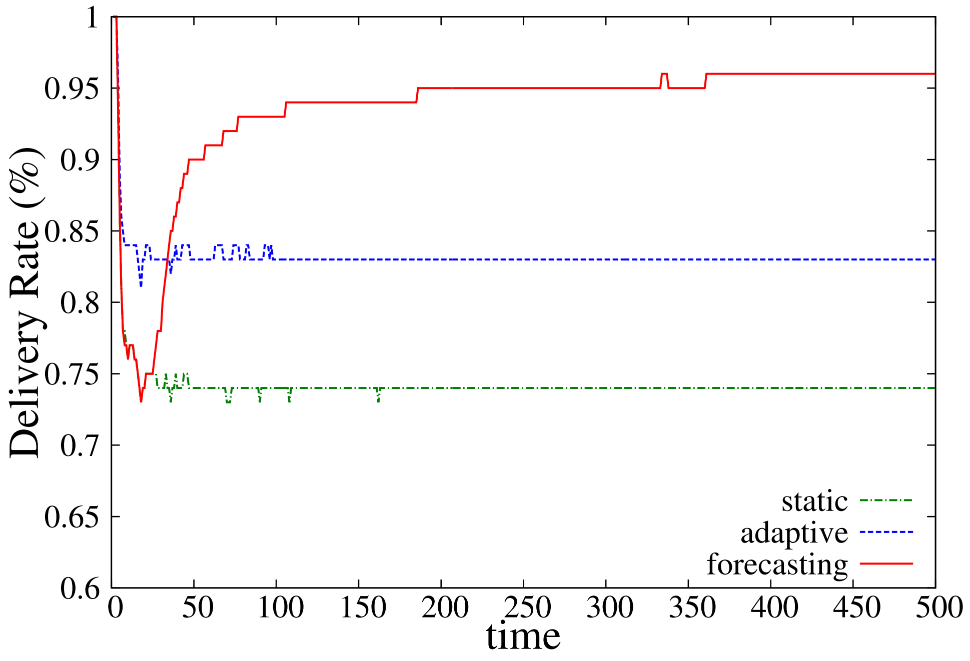


Figure 7.3: Average message delivery rate on periodic node disappearance

the graph is due to the training phase that is required by the Holt-Winter model in order to start producing predictions. As soon as the model is trained and routes are adapting, a sharp increase at the delivery rate is presented. Furthermore, as the model continues to collect feedback from the network, it further improves its forecasting ability until it converges at 95% packet delivery. It should be noted that a portion of the failed messages are due to destination nodes, instead of intermediates, that have failed. In that case, there is no alternative delivery path, but the messages are still counted as undelivered.

Figure 7.4 presents results from a similar set-up, however nodes do not have stable down/up-time periods. Instead their periods follow a Gaussian distribution with a random average value in the range of 10 to 50sec, as before, and σ value 2. Static routing is mostly unaffected from this change as it was expected. Average node downtime is not changing in the experiment, only the fixed periodicity that nodes disappear from the network. Similar situations are not uncommon in mobile networks, where patrolling nodes may occasionally come into contact with stationary nodes. Delivery rate of forecasting routing is affected by the introduced irregularity in node disappearance, though still remains high around 90%. The irregularity appears to also affect the routing based on recent observations, but not by a significant proportion (2%) to

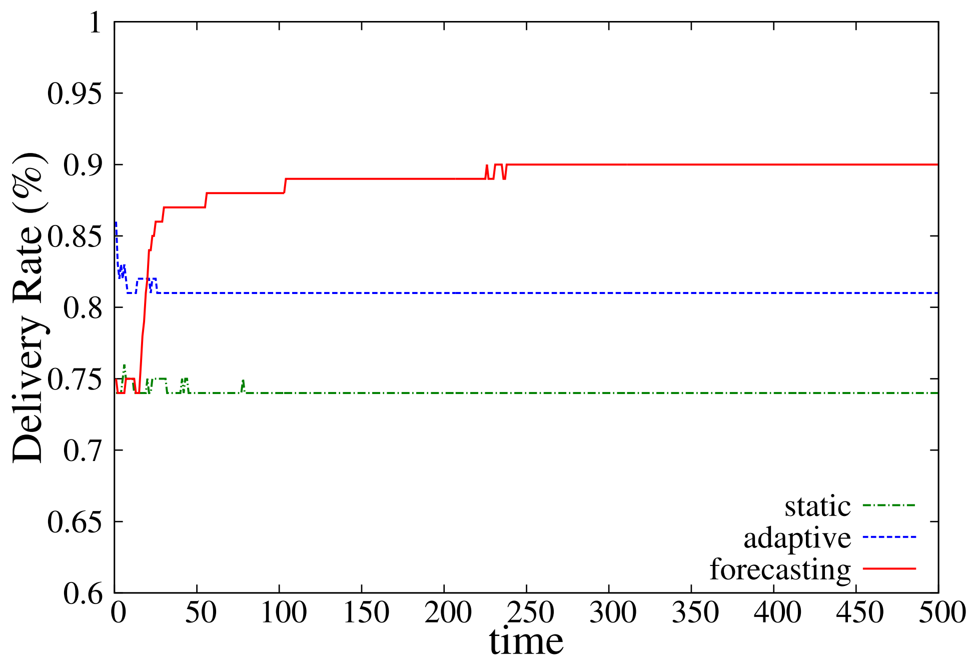


Figure 7.4: Average message delivery rate on irregularly periodic node disappearance

affect any change to the performance.

7.3.2 Node Link Reliability

The second network aspect we studied is link quality between nodes. During the lifetime of a deployed network we have noticed that links may exhibit recurring, periodic issues with delivery rates. This was typically due to moving obstacles that interfered with the signal, such as environmental inhabitants that have a certain routine or connectivity can be affected by nodes themselves being carried by entities, which move, even though they remain in theoretical communication range.

In order to address such repetitive adjustments on link quality, we apply the forecasting model on message drop rates of links in the network and study its effectiveness in this section. We use Castalia [PPB07] as a simulation environment. Castalia is built on top of Omnet++[3] and provides realistic link quality behaviour in a sensor network based on traffic, signal interference, node distance and noise in the wireless medium. To introduce the periodic fluctuation on the link quality, we modify the underlying connectivity map during the simulation. We study how

[3]http://www.omnetpp.org/

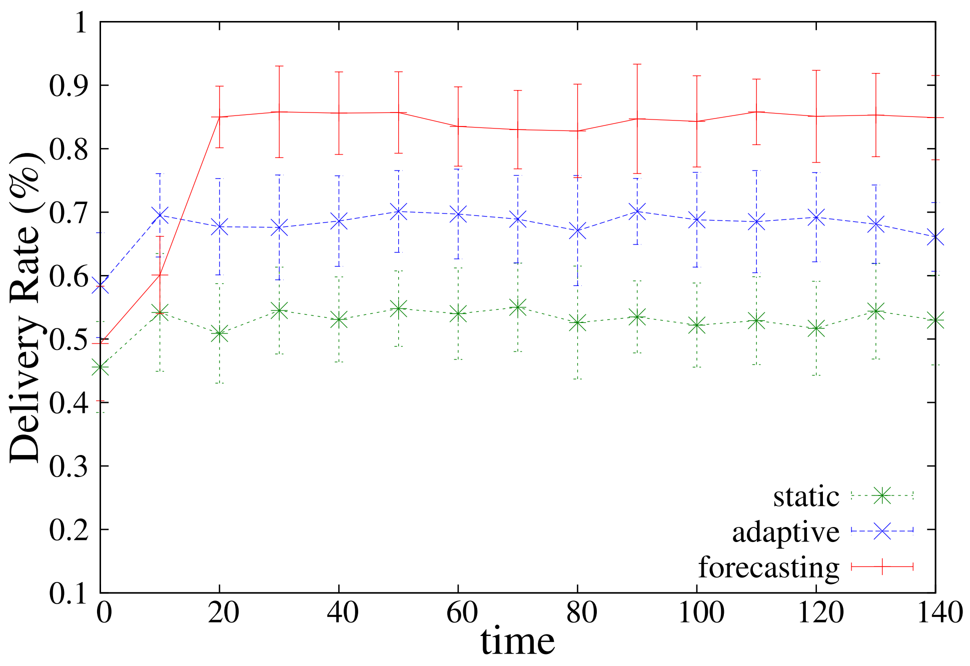


Figure 7.5: Packet delivery rate over periodically unreliable links

feed subscription delivery rates are affected and how effective is dynamic forecasting in such situations.

Figure 7.5 illustrates the performance of each approach when link quality varies periodically over time. The graph presents averages for every ten rounds and the variance is illustrated as the y-axis error bars. *SR* is, again, the reference line of network degradation reaching an average message delivery rate slightly above 50%. *ACR* does improve the naïve, static approach but on average it does not reach 70% message delivery rates. *AFR* performs best in this case as well. After an initial training phase, of roughly 20 rounds, it increases the delivery rate slightly below 90%.

7.3.3 Traffic Load and Congestion

Exclusive use of best quality paths in the network may result in over-utilisation of nodes causing packet congestion in their network buffers. Congestion can be caused either in incoming buffers, when a node is not able to process receiving packets fast enough, or in the outgoing buffers, when the medium is very busy for transmission and packets get queued. Castalia emulates

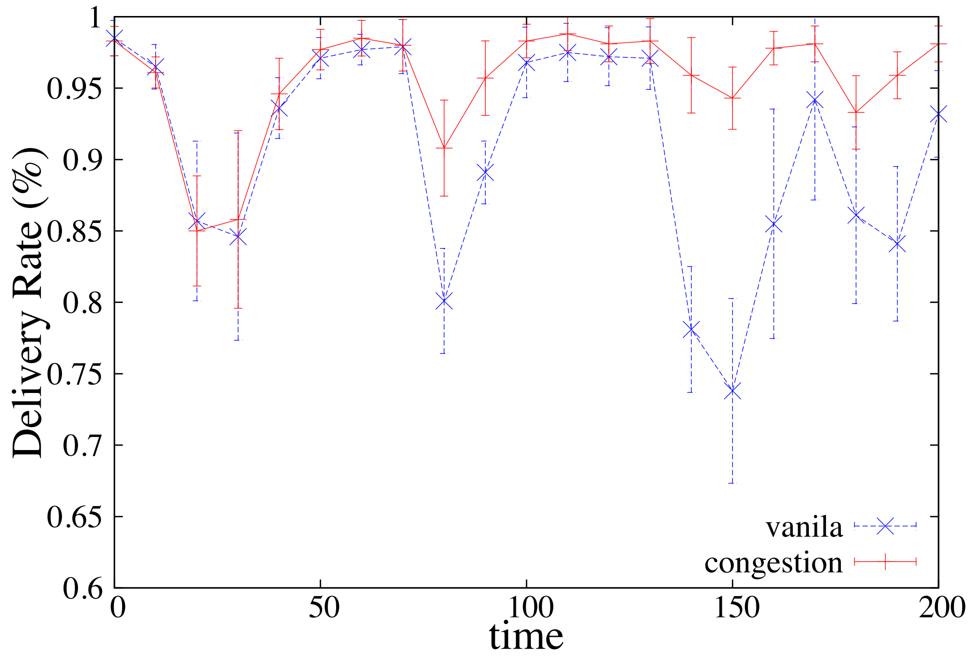


Figure 7.6: Average delivery rate with congestion forecasting and ideal network links

MAC and physical layer buffers and we study the behaviour of our forecasting approach under heavy traffic. We compare the approach from previous paragraphs, which ignores traffic load on nodes, with the the traffic-aware penalisation scheme that was introduced in section 7.1.3.

We ran an experiment, where nodes generate random medium-level traffic and there are four events during the simulation that cause group of nodes in the network to generate increased traffic. Each event produces different volumes of traffic. Figure 7.6 shows the routing behaviour in an ideal network, where no packets are dropped, apart from those in congested buffers. We ran the experiment in an ideal network to observe solely the impact of congestion and quantify the benefit of preventing node overload with excessive traffic. The four events that cause increased traffic can be easily observed in the graph as delivery rates fall sharply for the traffic-unaware scheme. On the other hand, the heavy traffic load penalisation scheme learns over time to spread traffic through different routes in the network grid avoiding packets due to congestion. It should be noted that even in an ideal network there are packet drops without heavy traffic. Such drops can be attributed to the half-duplex radio used in the simulation, where packets are lost on a node when the radio is in the transmission state.

Figure 7.7 presents the results of the same experiment that runs on a realistic network, where

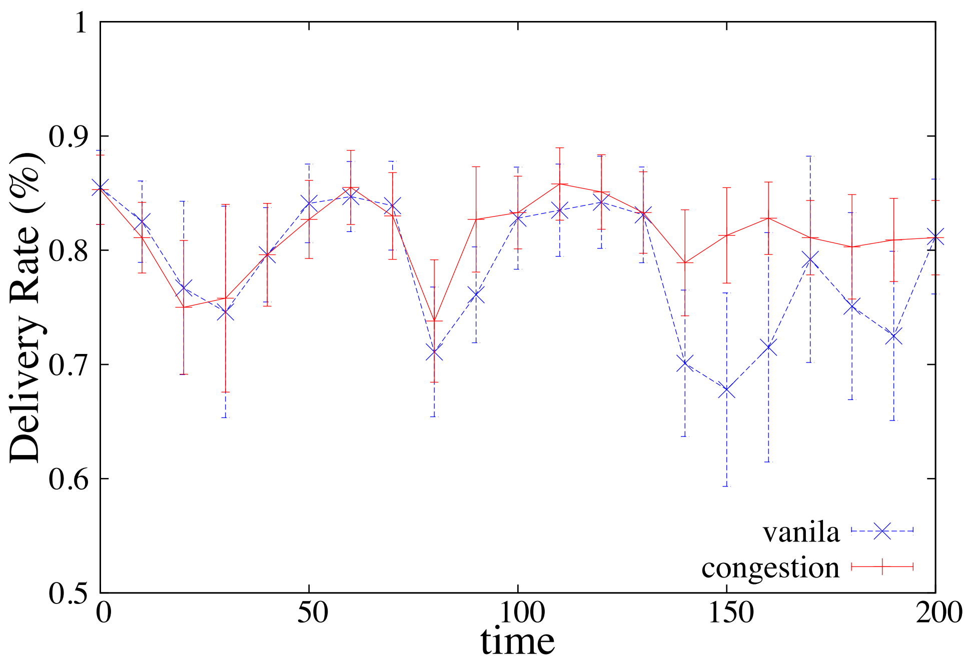


Figure 7.7: Average delivery rate with congestion forecasting and realistic network links

links drop packets due to noise similar to the set-up used in section 7.3.2. Overall, packet delivery rates are lower and their variance is increased for both cases. However, the trends remain similar, where the congestion-preventing scheme performs better during high-traffic events. In this case, the gap between the two approaches is closing however. This can be explained as a side-effect of the noisy links that reduce the amount of received packets, hence the effects of buffer congestion are decreased.

7.4 Related Work

In the literature the problem of reliable multi-hop communication among sensor nodes has been studied extensively. Approaches [PH07, RSO$^+$04, UBH09] include collections of metrics such neighbour availability, measured as beacon messages that detect neighbour presence or absence from the network; and link quality, based on the reception rate of messages. Neighbourhood collaboration is used in [HL06] for detection of missing neighbours. First, nodes monitor their neighbours by exchanging 'hello' messages. Then, the neighbourhood exchanges local observations of missing nodes to reach a local consensus before it triggers a failure alert for a missing node to the sink. RedFlag [UBH09] improves on the original algorithm using clock synchronisation among nodes. If a node misses a configurable number of handshakes, a neighbourhood consensus protocol starts. Nodes monitor information on link quality and neighbours' residual energy to infer whether failures are caused by broken links or power depletion. Similarly, Memento [RB07] monitors fail-stop node failures using heart-beats to tag a failed node after it misses a number of consecutive heart-beats. However, it further introduces a variance-bound mechanism that sets an upper bound on false positives.

The collection tree protocol (CTP) [GFJ+09] is an efficient data collection protocol for multi-hop sensor networks. It is based on two main ideas to improve message delivery rates and reduce imposed overheads. A datapath validation mechanism avoids looping of messages among nodes caused by dynamic link quality changes and adaptive node beaconing that reduced beacon messages of nodes with healthy links to conserve energy. However, the rate is increased when links start loosing packets. A backpressure collection protocol [MSKG10] improves the delivery rates of CTP for dynamic environments with moving sinks. However, both protocols target datagram packet routing and do not account for recurring patterns on failures and traffic.

Silberstein et al. [SPG+07] discuss how they cope with failures in a system that suppresses updates of new readings, unless new readings differ above a threshold. They compare several schemes including application level ACK messages, sequence numbers and hints of previous, possibly lost values. They use a Bayesian approach at the sink to infer missing values using models learned from the data instead of interpolating.

Detection of dropped and missing packets is a concern in many collection protocols. NACK messages have been used in PSFQ [WCK05] and Garuda [PSAV08] for detecting missing packets. However, NACK messages require an indefinite amount of packets stored in intermediate nodes. For streaming applications delay or lack of traffic is considered as a symptom of fault in the network [RCK+05, SBD02]. We follow a less taxing approach counting sequence numbers.

Passively monitoring the link quality using snooping has been used in [WTC03] tracking link layer sequence numbers. Congestion levels is monitored using buffer occupancy levels in [SAA03] and channel loading in [WEC03]. However, both methods require the radio to operate constantly in listening mode, thus consuming high levels of energy. Snif [RRV06], which operates as a secondary system with its own dedicated wireless channel deployed on the side of to the normal sensor network for monitoring purposes, is also a snooping approach.

Forecasting approaches in literature focus on link availability prediction based on node movement [JHR01] in mobile networks and life-time expectancy of nodes [MDP03]. Furthermore, a predictive model for minimising transmission time in networks based on cross-traffic estimations has been introduced in [YXZL06]. Finally, in [LFS10], a time-series model is proposed for predicting link quality in the network based on RSSI and LQI metrics. In our approach, we selected to utilise only application layer metrics and avoid common approaches such as RSSI and PRR measurements. Our choice was dictated by the constraints of the custom radio interfaces of our deployments that did not provide such information to higher levels.

7.5 Summary

In this chapter, we studied a case of network adaptation in multi-hop WSNs. We presented a dynamic routing service based on forecasting recurring network attributes and integrated in the ITA Sensor Fabric middleware. While the area of reliable multi-hop routing has been extensively studied, recurring effects of link failures and congestion, which appear in open space environments with mobile nodes, have not been a consideration in most of the literature. Forecasting trends of the network allows pro-active adaptation of routing paths for long running subscriptions, avoiding periodic network degradation. We demonstrated the effectiveness of forecasting for periodic failures using the Holt-Winters Additive model. Network attributes can be predicted and are able to distinguish different periods in the input, which enables effective estimation on future node connectivity.

Chapter 8

Conclusion

In this thesis, we have pointed out the significance of autonomic pervasive systems and the self-healing ability in response to component failures. We have looked into detection and healing mechanisms in different levels of such systems – faults in sensor readings, communication link failures and modification of application's deployment plan to adapt in dynamic environments.

We have studied these approaches with regard to a common architecture, the Self-Managed Cell, and provided a framework that promotes decentralised adaptation based on policies. We showcased tools that were developed to support this framework as well as experimental results that support our approaches for its self-healing services.

8.1 Summary of Thesis Achievements

1. *A middleware platform for WSNs that brings dynamic adaptation of network components in constrained platforms.*

   We presented *finger2* middleware, a policy management system that controls adaptation on embedded nodes with small overheads. It provides an event-driven paradigm for expressing system behaviour in terms of policies and allows separation of operational and adaptation logic.

2. *Definition of self-healing services in the Self-Managed Cell (SMC) architectural pattern and a prototype for the tinyOS sensor platform.*

   Starfish platform provides the Self-Managed Cell architecture for pervasive and WSN applications. It provides an infrastructure that allows autonomous components to compose into more complex services through well defined abstractions such as *missions* and *roles.*

3. *Identification and formal definition of sensor reading faults studied in long-running, real-world WSN deployments.*

   We identified four sensor readings fault types – *short*, *constant*, *noise* and *drift.* We described their attributes and formally modelled them in order to study their effects and impact on pervasive applications. Modelling of faults also contributes to our ability to devise detection and correction mechanisms.

4. *An adaptable, probabilistic fault detection mechanism for sensor readings that improves accuracy of sensor faults characterisation and minimises false positives.*

   We described fault detection model that can be configured in several degrees in order to balance fault detection accuracy and resource consumption on power constrained devices such as sensor nodes. The framework proposed involved Bayesian probabilistic classifiers for sensor readings and filtering of their decisions for incorporating the time dimension in the decision, which significantly removes false positives alerts. We further demonstrated the benefits of a detection and recovery mechanism based on a case study of a long-running, existing sensor network deployment.

5. *A dynamic task-allocation mechanism that responds to failures to decelerate service degradation due to sub-component faults by reorganising network assets.*

   We described how services, which are specified in terms of *missions* and *roles* in the *starfish* framework, can be statically analysed to automatically extract a task graph with component interdependencies. We proposed an autonomic task allocation mechanism that distributes tasks to nodes and dynamically modify the allocation plan to cope with component degradation.

6. *A case study on a self-adaptive, multi-hop routing middleware for real-world systems that is designed to avoid repetitive communication link failures.*

   Finally, we studied recurring patterns in communication link failures and congestion in multi-hop sensor networks. In order to minimise packet losses in such cases, we proposed a forecasting mechanism that can adapt the overlay communication network. We performed a case study on an industrial strength middleware platform that uses virtual circuits to connect consumers and producers of data in a sensor network.

8.2 Limitations of *starfish*

Adaptation in *starfish* is provided in term of policies. An immediate consequence of this is the restriction that only adaptation on the network's behaviour can be expressed. Native code updates that fix programming bugs and errors or add new functionality to support facilities that were not originally planned are hindered by the limitations of the underlying operating system to dynamically reprogram sensor nodes in-situ. Restrictions of native code updates are not a design limitation of *starfish* itself, but are rather inherited from the TinyOS platform that is build on. Should a different platform be used instead (e.g. contiki or SunSpots), code updates support could be added in the framework.

The core operations of sensor nodes are implemented in native code and policies are orchestrating native components by defining their interactions and controlling the flow of information among them. Policies should not, thus, be considered a programming mechanism for pervasive systems. Nevertheless, policies still provide a large degree of adaptation and autonomic configuration of the system as demonstrated throughout the thesis. They define new behaviour in the system and may be introduced to configure a different service.

Furthermore, policies require a human administrator that will author the service specification and adaptation logic and maintain/update this specification as necessary. The human factor is not entirely removed from the control-loop even though the system gains more autonomic properties. The system is not able to learn new adaptation approaches from its past observations. Certain mechanisms, such as the sensor fault detection service, may be replaced by alternatives that attempt to achieve such a goal [CRR+09, CSIR11], but the core logic of the system, which decides application of such mechanisms, remains scripted by a human operator.

Limitations on the fault-models presented are based on our assumptions for the sensors' behaviour. More specifically, we assume that events are observed by multiple sensors, deployed in an area, which have a very similar view of the environment. In other words, their inputs are highly correlated allowing us to discriminate deviating sensors as misbehaving. This attributes hold true for a number of sensors, such as thermometers, humidity sensors and accelerometers used in our case studies. However, the exact same models may not work similarly well in other cases. For instance the signal of acoustic or seismic sensors attenuates in space, while in other deployments there may not be adequate correlation among deployed sensors. In the former case, models presented can be modified in order to cope with the attenuation effect of an event. In the latter case, either injection of redundancy in the system should be considered or reliable behavioural modelling of the monitored attributes, which requires extended *a-priori* knowledge.

8.3 Scaling *starfish*

Most examples that have been discussed in this thesis involved small groups of nodes that were able to communicate directly and the information fusion centre we assumed to be within node communication reach. Nevertheless, scaling the SMC architecture to larger sensor networks that use multi-hop communication schemes introduces further challenges in collection of data and dissemination of information and policies. We have studied network failures in the multi-hop case study of ITA Sensor Fabric in chapter 7, however, multi-hop communication has not been integrated in *starfish* in the context of this thesis. An aspect that has not been highlighted in this thesis is dissemination of policies over such deployments. *Starfish* is designed so that implementation of multi-hop functionality can be added as an extension service that replaces the current 'network' module used to handle all communications between node SMCs.

In chapter 5, we looked into sensor data fault-models for small groups of closely located nodes, which create collaboration clusters to infer faulty behaviour among them. Scaling such deployments to larger areas can also introduce models that correlate information among different groups of remote sensors. Communication among groups can introduce new challenges on reliable multi-hop communication inside the network. Solutions found in the literature rely on hierarchical structures inside the network in order to reduce communication costs. We have not addressed such issues directly in our work but they are a direction that should be explored in the future.

The challenges of multi-hop routing in an unreliable, wireless medium have been studied extensively in the literature. Paradis and Han [PH07] provide an analytical survey on the matter. Challenges include reliable point-to-point communication between nodes, broken links and network fragmentation in addition to increased power consumption from relaying messages. Loss of sensor data messages may be tolerated in a system by use of models that compensate for gaps in sensor readings. However, control message loss prevention becomes more crucial. In the case of a node plan modification it can be catastrophic for the system if only part of the network adapts to the new plan while other nodes operate in an outdated fashion, leading to an inconsistent state.

Finally, a scaling factor in the SMC architecture is the complexity of services defined in policies. Policies are meant to be a mechanism to express adaptive behaviour in a system rather than being a means for programming sensor networks. Implementation details should remain in native code and provide interfaces, as defined in section 4.2.2, to be integrated in *starfish* and used in adaptation policies. As the complexity of services described in policies increases, management and maintenance of policies may require new constructs to tackle their complexity, similar to roles and missions. Over time, we created structures and paradigms on how source code is maintained in large code-bases. Complex policy-based services will require similar facilities. The *starfish* editor and tools such as the FSM conversion to policies (introduced in chapters 4.2.4 and 4.3 respectively) are a step towards that direction. Nevertheless, large-scale deployments are bound to expose the need for more elaborate tools that will incorporate common patterns in management of adaptive pervasive services.

8.4 Future Work

Future work in the direction of this thesis includes issues related to scaling of the framework on larger sensor networks as discussed in previous sections. Deployment of the middleware on a real network (i.e. outside a test-bed) with self-healing services that run at real-time (i.e. during the network's operation) will undoubtedly reveal new research challenges that relate to the scaling, management and propagation of the policies and SMC roles.

In terms of the models behind fault detection that encapsulate the knowledge on the system's condition, different deployments and sensor types exhibit different properties. More accurate modelling of monitored attributes can help increase the accuracy of the detection mechanisms. Also, on-line learning methods may assist adapting the monitoring mechanisms without a human operator updating a model when environmental behaviour changes.

In the planning phase of the algorithm, we may consider scenarios where specific sensor nodes are more valuable and difficult to replace than others and, thus, should be protected more than those that have redundancy. Adjustment to our allocation approach can be done to cope with requirements like this that provides additional information about a system's operational environment. Moreover, the dynamic allocation mechanism could be enhanced to deal with uncertainty in sensor accuracy in the local node ranking.

8.5 Closing Remarks

The scale of deployment of pervasive systems is highly dependent to the human management involvement and supervision that they require. Self-management is a key attribute for widespread adoption that will allow these systems to ubiquitously blend into the environment without being apparent to their users or hinder their tasks. To this extent, pervasive systems need to be able to collect knowledge about their operations and use it to adapt in a constantly changing environment. We believe that the techniques and tools we have implemented during the course of this thesis are a step towards this direction.

Bibliography

[AEP^{+}07] L. Atallah, M. Elhelw, J. Pansiot, D. Stoyanov, L. Wang, B. Lo, and G. Z. Yang. Behaviour profiling with ambient and wearable sensing. In *Proc. of 4th International Workshop on Wearable and Implantable Body Sensor Networks*, April 2007.

[AGGB10] S Abdelhak, C.S Gurram, S Ghosh, and M Bayoumi. Energy-balancing task allocation on wireless sensor networks for extending the lifetime. *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on DOI - 10.1109/MWSCAS.2010.5548700*, pages 781–784, 2010.

[AKF^{+}10] P Aghera, D Krishnaswamy, D Fang, A Coskun, and T Rosing. Dynaheal: Dynamic energy efficient task assignment for wireless healthcare systems. *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010 DOI -*, pages 1661–1664, 2010.

[AMC07] IF Akyildiz, T Melodia, and KR Chowdhury. A survey on wireless multimedia sensor networks. *Computer Networks*, 51(4):921–960, 2007.

[BBJ05] T Bokareva, N Bulusu, and S Jha. Sasha: toward a self-healing hybrid sensor network architecture. *Proceedings of the 2nd IEEE workshop on Embedded Networked Sensors (EmNets)*, pages 71–78, 2005.

[BMEP03a] V Bychkovskiy, S Megerian, Deborah Estrin, and Miodrag Potkonjak. A collaborative approach to in-place sensor calibration. *Information Processing in Sensor Networks*, pages 556–556, 2003.

[BMEP03b] Vladimir Bychkovskiy, Seapahn Megerian, Deborah Estrin, and Miodrag Potkonjak. A collaborative approach to in-place sensor calibration. In *IPSN*, Palo Alto, CA, 2003.

[Cer85] V Cerny. A thermodynamical approach to the travelling salesman problem: an efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985.

[CGC09] Ying Chen, Wenzhong Guo, and Guolong Chen. A multi-agent-based adaptive task allocation algorithm in wireless sensor networks. *Information Engineering and Computer Science, 2009. ICIECS 2009. International Conference on DOI - 10.1109/ICIECS.2009.5365341*, pages 1–4, 2009.

[CGL08] Valentino Crespi, Aram Galstyan, and Kristina Lerman. Top-down vs bottom-up methodologies in multi-agent system design. *Autonomous Robots*, 24:303–313, April 2008.

[CKS06] J Chen, S Kher, and A Somani. Distributed fault detection of wireless sensor networks. *Proceedings of the 2006 workshop on Dependability issues in wireless ad hoc networks and sensor networks*, page 72, 2006.

[CRR$^+$09] D Corapi, O Ray, A Russo, A Bandara, and E Lupu. Learning rules from user behaviour. *Artificial Intelligence Applications and Innovations III*, pages 459–468, 2009.

[CSIR11] D. Corapi, D. Sykes, K. Inoue, and A. Russo. Probabilistic rule learning in nonmonotonic domains. In *12th International Workshop on Computational Logic in Multi-Agent Systems, (CLIMA XII)*, Barcelona, Spain, July 2011.

[DCXC05] M Ding, D Chen, K Xing, and X Cheng. Localized fault-tolerant event boundary detection in sensor networks. *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, 2:902–913, 2005.

[DDLS01] N Damianou, N Dulay, E Lupu, and M Sloman. The ponder policy specification language. *Policies for Distributed Systems and Networks*, pages 18–38, 2001.

[Dey01] A.K Dey. Understanding and using context. *Personal and ubiquitous computing*, 5(1):4–7, 2001.

[DFEV06] A Dunkels, N Finne, J Eriksson, and T Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. *Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys)*, pages 15–28, 2006.

[DGV04] A Dunkels, B Gronvall, and T Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455 – 462, 2004.

[DHM06] J Deng, R Han, and S Mishra. Secure code distribution in dynamically programmable wireless sensor networks. *Proceedings of the 5th international conference on Information processing in sensor networks*, pages 292–300, 2006.

[DHS02] Stefan Dulman, Paul Havinga, and Faculty Of Computer Sciences. Wave leader election protocol for wireless sensor networks. *International Symposium on Mobile Multimedia Systems & Applications*, Feb 2002.

[DWSC10] B Ding, H Wang, D Shi, and J Cao. Taming software adaptability with architecture-centric framework. *Pervasive Computing and Communications (PerCom), 2010 IEEE International Conference on*, pages 145–151, 2010.

[EFGK03a] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131, June 2003.

[EFGK03b] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35:114–131, June 2003.

[EXT+09] N Edalat, Wendong Xiao, Chen-Khong Tham, E Keikha, and Lee-Ling Ong. A price-based adaptive task allocation for wireless sensor network. *Mobile Adhoc and Sensor Systems, 2009. MASS '09. IEEE 6th International Conference on DOI - 10.1109/MOBHOC.2009.5337039*, pages 888–893, 2009.

[FL07] Alberto E. Schaeffer Filho and Emil Lupu. Abstractions to support interactions between self-managed cells. In *Inter-Domain Management, First International Conference on Autonomous Infrastructure, Management and Security, AIMS 2007, Oslo, Norway, June 21-22, 2007, Proceedings*, pages 160–163, 2007.

[GFJ+09] O Gnawali, R Fonseca, K Jamieson, D Moss, and P Levis. Collection tree protocol. *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 1–14, 2009.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[GLB+03] D Gay, P Levis, R Von Behren, M Welsh, E Brewer, and D Culler. The nesc language: A holistic approach to networked embedded systems. *ACM Sigplan Notices*, 38(5):1–11, 2003.

[GLC07] David Gay, Philip Levis, and David Culler. Software design patterns for tinyos. *ACM Transactions on Embedded Computing Systems*, 6(4), 2007.

[GZH09] Shuo Guo, Ziguo Zhong, and Tian He. Find: faulty node detection for wireless sensor networks. *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, Nov 2009.

[HC04] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *ACM SenSys*, 2004.

[HL06] C Hsin and M Liu. Self-monitoring of wireless sensor networks. *IEEE Computer Communications*, 29(4):462–476, 2006.

[HM04] D.L. Hall and S.A. McMullen. *Mathematical techniques in multisensor data fusion.* Artech House, Boston, MA, 2004.

[HM08] MC Huebscher and JA McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3):7, 2008.

[HRR05] S Harte, A Rahman, and K M Razeeb. Fault tolerance in sensor networks using self-diagnosing sensor nodes. In *IEEE Intelligent Environments 2005*, Colchester, UK, 2005.

[JHR01] Shengming Jiang, Dajiang He, and Jianqiang Rao;. A prediction-based link availability estimation for mobile ad hoc networks. *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 3:1745 – 1752 vol.3, 2001.

[JRP$^+$10] MP Johnson, H Rowaihy, D Pizzocaro, A Bar-Noy, S Chalmers, T La Porta, and A Preece. Sensor-mission assignment in constrained environments. *IEEE Transactions on Parallel and Distributed Systems*, 2010.

[KC03] J Kephart and D Chess. The vision of autonomic computing. *IEEE Computer*, pages 41–50, Dec 2003.

[KGG$^+$09] P Kuryloski, A Giani, R Giannantonio, K Gilani, R Gravina, V.P Seppa, E Seto, V Shia, C Wang, and P Yan. Dexternet: An open platform for heterogeneous body sensor networks and its applications. *Wearable and Implantable Body Sensor Networks, 2009. BSN 2009. Sixth International Workshop on*, pages 92–97, 2009.

[KHB07] A Krohn, M Hazas, and M Beigl. Removing systematic error in node localisation using scalable data fusion. *In Wireless Sensor Networks, 4th European Conference, EWSN 2007, Delft, The Netherlands*, pages 341–356, 2007.

[KKL04] S. Kornienko, O. Kornienko, and P. Levi. Generation of desired emergent behavior in swarm of micro-robots. *European Conference on Artificial Intelligence (ECAI)*, 2004.

[KLDY05] Rachel King, Benny P.L. Lo, Ara Darzi, and Guang-Zhong Yan. Hand gesture recognition with body sensor networks. In *Proc. of 2nd International Workshop on Wearable and Implantable Body Sensor Networks*, April 2005.

[KLM+97] G Kiczales, J Lamping, A Mendhekar, C Maeda, C Lopes, J.M Loingtier, and J Irwin. Aspect-oriented programming. *ECOOP- Object-Oriented Programming, 11th European Conference*, pages 220–242, 1997.

[KPSV02] Farinaz Koushanfar, Miodrag Potkonjak, and Alberto Sangiovanni-Vincentelli. Fault tolerance techniques for wireless ad hoc sensor networks. *sensors*, pages 1491–1496, 2002.

[KPSV03] F Koushanfar, M Potkonjak, and A Sangiovanni-Vincentelli. On-line fault detection of sensor measurements. *Sensors, 2003. Proceedings of IEEE*, 2:974–979 Vol. 2, 2003.

[KW05] S. S. Kulkarni and K. Wang. Mnp: Multihop network reprogramming service for sensor networks. In *IEEE ICDCS*, 2005.

[LBBL10] CH Liu, C Bisdikian, JW Branch, and KK Leung. Qoi-aware wireless sensor network management for dynamic multi-task operations. *Sensor Mesh and Ad Hoc Communications and Networks (SECON), 2010 7th Annual IEEE Communications Society Conference on DOI - 10.1109/SECON.2010.5508203*, pages 1–9, 2010.

[LC02] Philip Levis and David Culler. Maté: a tiny virtual machine for sensor networks. *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, Dec 2002.

[LCL07] H.J Lee, A Cerpa, and P Levis. Improving wireless simulation through noise modeling. *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 21–30, 2007.

[LDCO06] Winnie Louis Lee, Amitava Datta, and Rachel Cardell-Oliver. Winms: Wireless sensor network-management system, an adaptive policy-based management for wireless sensor networks. *Technical Report University of Western Australia*, 2006.

[LDS$^+$08] Emil Lupu, Naranker Dulay, Morris Sloman, J Sventek, S Heeps, S Strowes, K Twidle, Sye Loong Keoh, and A Schaeffer-Filho. Amuse: autonomic management of ubiquitous e-health systems. *Concurrency and Computation*, 20(3):277, 2008.

[LFS10] L Liu, Y Fan, and J Shu.... A link quality prediction mechanism for wsns based on time series model. *2010 Symposia and Workshops on ...*, Jan 2010.

[Lie96] Karl Liebherr. *Adaptive Object Oriented Programming: The Demeter Approach with Propagation Patterns*. PWS Publishing Co., 1996.

[LLWC03] P Levis, N Lee, M Welsh, and D Culler. Tossim: accurate and scalable simulation of entire tinyos applications. *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, 2003.

[LM03] T Liu and M Martonosi. Impala: a middleware system for managing autonomic, parallel sensor systems. *ACM Sigplan Notices*, 38(10):107–118, 2003.

[LMP$^+$05] P Levis, S Madden, J Polastre, R Szewczyk, K Whitehouse, A Woo, D Gay, J Hill, M Welsh, and E Brewer. Tinyos: An operating system for sensor networks. *Ambient Intelligence*, 35, 2005.

[LNV09] T Le, TJ Norman, and W Vasconcelos. Agent-based sensor-mission assignment for tasks sharing assets. *Proc. Third Int. Workshop on Agent Technology for Sensor Networks (ATSN'09)*, pages 33–40, 2009.

[MALW10] W Munawar, M Alizai, O Landsiedel, and K Wehrle. Dynamic tinyos: Modular and transparent incremental code-updates for sensor networks. *Communications (ICC), 2010 IEEE International Conference on*, pages 1 – 6, 2010.

[McF86] G. McFarland. The benefits of bottom-up design. *SIGSOFT Software Engineering Notes*, 11:43–51, 1986.

[MDP03] M Maleki, K Dantu, and M Pedram. Lifetime prediction routing in mobile ad hoc networks. *Wireless Communications and Networking, 2003. WCNC 2003. 2003 IEEE*, 2:1185–1190, 2003.

[MJ10] S Misra and A Jain. Policy controlled self-configuration for unattended wireless sensor networks. *Journal of Network and Computer Applications*, 2010.

[MLM$^+$05] PJ Marrón, A Lachenmann, D Minder, M Gauger, O Saukh, and K Rothermel. Management and configuration issues for sensor networks. *International Journal of Network Management*, 15(4):253, 2005.

[MPD04a] S Mukhopadhyay, D Panigrahi, and S Dey. Model based error correction for wireless sensor networks. *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, pages 575–584, 2004.

[MPD04b] Shoubhik Mukhopadhyay, Debashis Panigrahi, and Sujit Dey. Data aware, low cost error correction for wireless sensor networks. In *IEEE WCNC*, Atlanta, GA, 2004.

[MPT$^+$08] D.G. McIlwraith, J. Pansiot, S. Thiemjarus, B.P.L. Lo, and G.Z. Yang. Probabilistic decision level fusion for real-time correlation of ambient and wearable sensors. In *Proc. of 5th International Workshop on Wearable and Implantable Body Sensor Networks*, June 2008.

[MSD10] Leonardo Mostarda, Daniel Sykes, and Naranker Dulay. A State Machine-Based Approach For Reliable Adaptive Distributed Systems. In *7th IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, March 2010.

[MSKG10] S Moeller, A Sridharan, B Krishnamachari, and O Gnawali. Routing without routes: The backpressure collection protocol. *Proceedings of the 9th ACM/IEEE*

International Conference on Information Processing in Sensor Networks, pages 279–290, 2010.

[OAVRH06] E. Ould-Ahmed-Vall, G.F. Riley, and B. Heck. Distributed fault tolerance for event detection using heterogeneous wireless sensor networks. *Georgia Tech/CERCS, Tech. Rep. GIT-CERCS-06-09*, 2006.

[PBM11] R.K Panta, S Bagchi, and S.P Midkiff. Efficient incremental code update for sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 7(4):30, 2011.

[PH07] L Paradis and Q Han. A survey of fault management in wireless sensor networks. *Journal of Network and Systems Management*, 15(2):171–190, 2007.

[PP10] A Pathak and V.K Prasanna. Energy-efficient task mapping for data-driven sensor network macroprogramming. *Computers, IEEE Transactions on DOI - 10.1109/TC.2009.168*, 59(7):955–968, 2010.

[PPB07] HN Pham, D Pediaditakis, and A Boulis. From simulation to real deployments in wsn and back. *World of Wireless, Mobile and Multimedia Networks, 2007. WoWMoM 2007. IEEE International Symposium on a*, pages 1–6, 2007.

[PPS$^+$09] H Pham, J.M Paluska, U Saif, C Stawarz, C Terman, and S Ward. A dynamic platform for runtime adaptation. *Pervasive Computing and Communications, 2009. PerCom 2009. IEEE International Conference on DOI - 10.1109/PERCOM.2009.4912746*, pages 1–10, 2009.

[PS03] H Park and MB Srivastava. Energy-efficient task assignment framework for wireless sensor networks. *Center for Embedded Networking Sensing (CENS), Technical Reports*, 2003.

[PSAV08] Seung-Jong Park, R Sivakumar, I Akyildiz, and R Vedantham. Garuda: Achieving effective reliability for downstream communication in wireless sensor networks. *IEEE Transactions on Mobile Computing*, 7(2):214 – 230, 2008.

[PSM$^+$07] Julien Pansiot, Danail Stoyanov, Douglas McIlwraith, Benny P.L. Lo, and G. Z. Yang. Ambient and wearable sensor fusion for activity recognition in healthcare monitoring systems. In *Proc. of 4th International Workshop on Wearable and Implantable Body Sensor Networks*, April 2007.

[RB07] S Rost and H Balakrishnan. Memento: A health monitoring system for wireless sensor networks. *Sensor and Ad Hoc Communications and Networks, 2006. SECON'06. 2006 3rd Annual IEEE Communications Society on*, 2:575–584, 2007.

[RCK$^+$05] Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. Sympathy for the sensor network debugger. *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, Nov 2005.

[RNL03] Linnyer Beatrys Ruiz, Jose Marcos Nogueira, and Antonio A. F. Loureiro. Manna: A management architecture for wireless sensor networks. *IEEE Communications Magazine*, 41(2):116–125, 2003.

[RRV06] M Ringwald, K Römer, and A Vialetti. Snif: Sensor network inspection framework. *Department of Computer Science, ETH Zurich, Technical Report*, 535, 2006.

[RSO$^+$04] LB Ruiz, IG Siqueira, LB Oliveira, HC Wong, JMS Nogueira, and AAF Loureiro. Fault management in event-driven wireless sensor networks. *Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, page 156, 2004.

[SAA03] Y Sankarasubramaniam, Ö.B Akan, and I.F Akyildiz. Esrt: event-to-sink reliable transport in wireless sensor networks. *Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing*, pages 177–188, 2003.

[SBD02] J Staddon, D Balfanz, and G Durfee. Efficient tracing of failed nodes in sensor networks. *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 122–130, 2002.

[Slo94] Morris Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2:333–360, 1994. 10.1007/BF02283186.

[SPG$^+$07] A Silberstein, G Puggioni, A Gelfand, K Munagala, and J Yang. Suppression and failures in sensor networks: A bayesian approach. *Proceedings of the 33rd international conference on Very large data bases*, pages 842–853, 2007.

[SRAA10] N Salazar, J Rodriguez-Aguilar, and J Arcos. Self-configuring sensors for uncharted environments. *Self-Adaptive and Self-Organizing Systems (SASO), 2010 4th IEEE International Conference on*, pages 134 – 143, 2010.

[SY04] F Sivrikaya and B Yener. Time synchronization in sensor networks: a survey. *Network, IEEE*, 18(4):45–50, 2004.

[SYH09] S Suenaga, N Yoshioka, and S Honiden. Group migration by mobile agents in wireless sensor networks. *The Computer Journal*, 2009.

[TDLS09] Kevin Twidle, Naranker Dulay, Emil Lupu, and Morris Sloman. Ponder2: A policy system for autonomous pervasive environments. In *The Fifth International Conference on Autonomic and Autonomous Systems (ICAS)*, 2009.

[TEO05] Yuan Tian, E Ekici, and F Ozguner. Energy-constrained task mapping and scheduling in wireless sensor networks. *Mobile Adhoc and Sensor Systems Conference, 2005. IEEE International Conference on DOI - 10.1109/MAHSS.2005.1542802*, pages 8 pp.–218, 2005.

[TvS02] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall, 2002.

[TWS06] K Terfloth, G Wittenburg, and J Schiller. Facts - a rule-based middleware architecture for wireless sensor networks. *Communication System Software and Middleware, 2006. Comsware 2006. First International Conference on*, pages 1 – 8, 2006.

[UBH09] I Urteaga, K Barnhart, and Q Han. Redflag: A run-time, distributed, flexible, lightweight, and generic fault detection service for data-driven wireless sensor applications. *Pervasive and Mobile Computing*, 5(5):432–446, 2009.

[WASW05] G Werner-Allen, P Swieskowski, and M Welsh. Motelab: a wireless sensor network testbed. *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on DOI - 10.1109/IPSN.2005.1440979*, pages 483– 488, 2005.

[WCK05] C.Y Wan, A.T Campbell, and L Krishnamurthy. Pump-slowly, fetch-quickly (psfq): a reliable transport protocol for sensor networks. *IEEE Journal on Selected Areas in Communications*, 23(4):862–872, 2005.

[WEC03] C.Y Wan, S.B Eisenman, and A.T Campbell. Coda: congestion detection and avoidance in sensor networks. *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 266–279, 2003.

[WFGDF05] A.F.T. Wineld, J. Sa M.C. Fernandez-Gago, C. Dixon, and M. Fisher. On formal specification of emergent behaviors in swarm robotic systems. *Advanced Robotic Systems*, 2005.

[WGB$^+$09] J. Wright, C Gibson, F. Bergamaschi, K. Marcus, R. Pressley, G. Verma, and G Whipps. A dynamic infrastructure for interconnecting disparate isr/istar assets (the ita sensor fabric). In *IEEE/ISIF Fusion Conference*, July 2009.

[WHVC05] TY Wang, YS Han, PK Varshney, and PN Chen. Distributed fault-tolerant classification in wireless sensor networks. *IEEE Journal on Selected Areas in Communications*, 23(4):724–734, 2005.

[WTC03] A Woo, T Tong, and D Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 14–27, 2003.

[WZC06] Q Wang, Y Zhu, and L Cheng. Reprogramming wireless sensor networks: challenges and approaches. *Network, IEEE*, 20(3):48–55, 2006.

[XLTD09] W Xiao, SM Low, CK Tham, and S Das. Prediction based energy-efficient task allocation for delay-constrained wireless sensor networks. *Sensor, Mesh and Ad Hoc Communications and Networks Workshops, 2009. SECON Workshops' 09. 6th Annual IEEE Communications Society Conference on*, pages 1–3, 2009.

[Yan06] G.-Z. Yang. *Body Sensor Networks*. Springer-Verlag, London, 2006.

[YMC+08] J Ye, S McKeever, L Coyle, S Neely, and S Dobson. Resolving uncertainty in context integration and abstraction: context integration and abstraction. *Proceedings of the 5th international conference on Pervasive services*, pages 131–140, 2008.

[YXZL06] Shouyi Yin, Yongqiang Xiong, Qian Zhang, and Xiaokang Lin. Prediction-based routing for real time communications in wireless multi-hop networks. *QShine '06: Proceedings of the 3rd international conference on Quality of service in heterogeneous wired/wireless networks*, Aug 2006.

[ZKS+08a] Y Zhu, SL Keoh, M Sloman, E Lupu, N Dulay, and N Pryce. A policy system to support adaptability and security on body sensors. *Medical Devices and Biosensors, 2008. ISSS-MDBS 2008. 5th International Summer School and Symposium on*, pages 37–40, 2008.

[ZKS+08b] Yanmin Zhu, Sye Loong Keoh, Morris Sloman, Emil Lupu, Naranker Dulay, and Nathaniel Pryce. An efficient policy system for body sensor networks. In *ICPADS*, Melbourne, Australia, 2008.

[ZLG07] Jinghua Zhu, Jianzhong Li, and Hong Gao. Tasks allocation for real-time applications in heterogeneous sensor networks for energy minimization. *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on DOI - 10.1109/SNPD.2007.255*, 2:20–25, 2007.

[ZSLK09] Y. Zhu, M. Sloman, E. Lupu, and S. L. Keoh. Vesta: A secure and autonomic system for pervasive healthcare. *3rd International Conference on Pervasive Computing Technologies for Healthcare*, May 2009.