Improving Fault Tolerance Support in Wireless Sensor Network Macroprogramming

Guilherme de Maio Nogueira

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
Para
obtenção do título
de
Mestre em Ciência da Computação

Orientador: Prof. PhD. Marco Aurélio Gerosa

Co-Orientador: PhD. Animesh Pathak

During this work the author received support through the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement number 257178 (project CHOReOS - Large Scale Choreographies for the Future Internet), and through French National Research Agency ANR, contract nr. ANR-BLAN-SIMI10-LS-100618-6-01 (Project Murphy)

São Paulo, Março de 2015

Improving Fault Tolerance Support in Wireless Sensor Network macroprogramming

Esta versão da dissertação/tese contém as correções e alterações sugeridas pela Comissão Julgadora durante a defesa da versão original do trabalho, realizada em 01/12/2014. Uma cópia da versão original está disponível no Instituto de Matemática e Estatística da Universidade de São Paulo.

Comissão Julgadora:

- Prof. Dr. Marco Aurélio Gerosa (orientador) - IME-USP
- Prof. Dr. Alfredo Goldman Vel Lejbman - IME-USP
- Prof. Dr. Jó Ueyama - ICMC-USP

Agradecimentos

Primeiramente, ao meu orientador, Marco Aurélio Gerosa, que me apoiou, incentivou e me ensinou muito durante esses anos de mestrado. Seu profissionalismo em todos os aspectos, assim como a vontade de ensinar sempre e manter-se atualizado é incrível. Sem sombra de dúvidas é um professor e pesquisador excelente.

Also, I would like do thank Animesh for the great months spent in Paris under his supervision at Inria. I learned a lot there, and if I felt welcomed and productive, he was the one responsible. I miss that environment and our tennis matches in the afternoon.

Agradeço aos meus colegas de orientação, em especial Leo e Gustavo. Aprendi muito com vocês! Aos colegas de projeto CHOReOS, de LabXP, muitos que hoje são amigos de verdade. O IME me apresentou a pessoas completamente fora de série! Foi muito bom contar com todos nesta jornada! Espero ainda beber muitas cervejas com vocês!

À Marina, que aguentou a crônica falta de atenção dispensada durante o mestrado! Aos meus pais, meus amigos e familiares. À todos que dispensei por que precisava estudar!

Obrigado, and thanks for all the fish!

Resumo

NOGUEIRA, G. M. **Evoluindo o Suporte à Tolerância a Falhas na Macroprogramação de Redes de Sensores sem Fio**. 2014. 120 f. Dissertação (Mestrado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2014.

Redes de Sensores Sem Fio (RSSF) são sistemas distribuídos em rede para sensoreamento, compostos de pequenos dispositivos conectados entre si. Esses sistemas são utilizados para construir aplicações que medem e atuam no meio físico. Cada dispositivo da rede, chamado de nó, é equipado com sensores, e algumas vezes, atuadores. Os nós também comumente possuem limitações em termos de suprimento de energia e capacidade de armazenamento e processamento. Em adição à essas limitações, redes de sensores sem fio também estão sujeitas à diversos tipos de falhas, especialmente quando são implantadas em ambientes de condições naturais extremas, como florestas e plantações.

Por essas razões, desenvolvedores de aplicações para redes de sensores sem fio necessitam utilizar mecanismos de tolerância a falhas. Alguns dos mecanismos de tolerância a falhas são implementados em hardware, porém são mais comumente deixados para implementação em software. Além disso, a maior parte do desenvolvimento de aplicações para RSSF é feita em baixo nível de abstração, perto do sistema operacional. Desse modo, além de terem que concentrar-se na lógica da aplicação em baixo nível, os desenvolvedores ainda têm que implementar os mecanismos de tolerância a falhas junto à aplicação, pela falta de bibliotecas ou componentes genéricos para esse fim. Técnicas de programação em alto nível para RSSF já foram propostas na forma de linguagens e arcabouços de macroprogramação. No entanto, uma minoria lida com aspectos de tolerância a falhas.

O objetivo desse trabalho é incorporar funcionalidades para tolerância a falhas ao Srijan, um arcabouço de macroprogramação para redes de sensores sem fio. Srijan possui código aberto e é baseado em uma linguagem mista declarativa-imperativa chamada Abstract Task Graph (ATaG). Evoluímos o arcabouço para dar suporte à geração automática de código lidando com quedas de nós da rede e falhas que resultam em dados incorretos de sensores. Nesta dissertação, apresentamos a nossa implementação de tais funcionalidades, juntamente com a avaliação conduzida sobre a ferramenta. Mostramos que é possível prover um ar-

cabouço de macroprogramação com suporte apropriado ao desenvolvimento de aplicações para RSSF que necessitam tolerância a falhas.

Palavras-chave: redes de sensores sem fio, tolerância a falhas, macroprogramação, Srijan.

Abstract

NOGUEIRA, G. M. **Improving Fault Tolerance Support in Wireless Sensor Network macroprogramming**. 2014. 120 f. Dissertation (Master) - Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2014.

Wireless Sensor Networks (WSN) are distributed sensing network systems composed of tiny networked devices. These systems are employed to develop applications for sensing and acting on the environment. Each network device, or "node," is equipped with sensors and sometimes actuators as well. WSNs typically have limited power, processing, and storage capability, and are also subject to faults, especially when deployed in harsh environments.

Given WSNs limitations, application developers often design fault-tolerance mechanisms. Although developers implement some fault-tolerance mechanisms in hardware, most are implemented in software. Indeed, WSN application development mostly occurs at a low level, close to the operating system, which forces developers to focus away from application logic and dive into WSN's technical background. Some have proposed high-level programming solutions, such as macroprogramming languages and frameworks; however, few deal with fault-tolerance.

This dissertation aims to incorporate fault-tolerance features into Srijan, an open-source WSN macroprogramming framework based on a mixed declarative-imperative language called Abstract Task Graph (ATaG). We augment Srijan's framework to support code generation for dealing with devices that crash or report meaningless values. We present our feature implementation here, along with an evaluation of the tool, demonstrating that it is possible to provide a macroprogramming framework with appropriate support for developing fault-tolerant WSN applications.

Keywords: wireless sensor networks, fault tolerance, macroprogramming, Srijan.

Contents

Abbreviations

ANR	Agence Nationale de la Recherche (The French National Research Agency)
API	Application Programming Interface
ATaG	Abstract Task Graph
DSL	Domain Specific Language
FIFO	First In, First Out
FT	Fault Tolerance
GUI	Graphical User Interface
IEEE	Institute of Electrical and Electronics Engineers
LAAS	Laboratoire d'Analyse et d'Architecture des Systèmes (Laboratory for Analyses and Architecture of Systems)
LFR	Leader/Follower Replication
MAC	Media Access Control
OS	Operating System
PBR	Primary/Backup Replication
SCA	Service Component Architecture
SDK	Software Development Kit
WSN	Wireless Sensor Networks

List of Figures

List of Tables

List of listings

Chapter 1

Introduction

Wireless Sensor Networks (WSNs) are distributed systems composed of embedded devices, which typically have limited processing, storage, and power supply capability, but are equipped with sensors, actuators or both [MP11, MP12, ASSC02]. These systems are increasingly employed to sense and act on the environment, permeating a variety of application domains such as avionics, wide-area surveillance [MLZ+06], structural sensing, telemedicine, river flood monitoring [HUM+11], smart grid monitoring [HMLU10], and space exploration [PH07]. Langendoen et al. [LBV06], for example, reports the experience of deploying over 100 wireless sensor nodes over the course of three months in a precision agriculture setting (agriculture led by very close analysis of environmental conditions), monitoring a potato field's temperature and humidity.

WSN fault tolerance is critical because, in addition to their limited processing power, storage, and power supply, WSNs are also subject to failures, especially when deployed in harsh environments. Langendoen et al.'s [LBV06] report on the multiple failures they encountered in their experiment emphasizes the need for WSN worst-case scenario application design.

Failures in wireless sensor networks arise from a variety of sources and at three levels: Node, Network, or Sink [dSVB07]. Node-level failures may occur due to nodes' fragility: external events can damage them and they can begin behaving inappropriately. Network-level failures relate to network routing, which is essential for sensor data collection and WSN coordination. Communication links between nodes are highly volatile, meaning interference can occur, and nodes also sometimes have a degree of mobility. This volatility leads to network partitions, dynamic changes in topology, blocked links, message collision, and packet loss/corruption [PH07]. Finally, sink-level failures occur at a higher network level, wherein a device that collects network-generated data, a sink, is subject to its components' faults. Unless a fault tolerant mechanism is available, this device failure may cause system-wide failure, since the sensing data cannot be aggregated and accessed.

To deal with these kinds of faults, hardware and software solutions may be implemented. However, although hardware advances are important to WSNs, the technology can only be exploited if applications developers have access to proper software platforms. Nevertheless, there are a few experiences reported in the literature wherein WSN applications have been deployed relying on high-level programming support [MP11]. In most applications, the development process occurs at a low level, close to the operating system, which forces developers to deal with distributed applications' design, including network issues, synchronization, and data aggregation, data structure implementation, and fault management. Since developers must therefore direct their focus away from application logic and toward WSN's technical

aspects, it is challenging for domain experts to create WSN applications.

High-level WSN application programming approaches are implemented using a variety of tools and languages that provide different features for systems with varied characteristics [GGG05, NMW07, AJG07, Kar10]. However, only a few of them deal with fault-tolerance, and thus provide limited guarantees. For instance, transient faults, such as sensors temporarily providing erroneous readings, are often overlooked. According to Mottola and Picco [MP11], high-level programming frameworks in which faults are a first-class notion are necessary for developing WSN applications better adapted for harsh environments.

1.1 Goal

The general goal of this work is to investigate how to enhance macroprogramming languages to support fault tolerance mechanisms. To do so, we have chosen the Srijan macroprogramming framework [Pat08] (described in Chapter 4) to implement and evaluate fault tolerance mechanisms. We thus intend to foment discussion about better ways to specify and implement fault tolerance properties in WSN application development.

Based on this framework, this work's objective is to provide WSN application development support, specifically for building fault tolerant properties designed for network sinks. We do this by expanding Srijan to include: (i) fault-tolerance specification for Abstract Tasks; and (ii) code generation for common fault-tolerance processes, such as replication in master/slave fashion for sinks [SFR12] and data validation of sensor-produced values.

We use two approaches to evaluate the work. First, we use quantitative evaluation to analyze Srijan's framework code generation efficiency with our proposed fault tolerance support. Second, we conduct a qualitative study on how the fault tolerance features work, and compare it in detail to related work.

1.2 Context of the Work

This work's thematics were born from the author's participation in the CHOReOS and Baile projects, both of which studied the web service choreographies applicability at large scales. The CHOReOS project was financed by the European Commission and included the participation of several universities and private companies, USP being the only non-European one.

During his participation in these projects, he developed prototypes, conducted experiments, presented talks, helped with written deliverables, co-wrote papers and reports, and interacted with project partners during meetings and technical discussions.

Because of this, the author was invited intern at Inria Paris-Rocquencourt laboratory ARLES, with the goal of studying wireless sensor networks' fault tolerance.

This dissertation contains work that was conducted during and after his internship.

1.3 Structure of the Document

The text is divided in 8 chapters. We start by giving an overview of the Wireless Sensor Network field in Chapter 2. Following, we present current practices in fault tolerance for sensor networks in Chapter 3. In Chapter 4, we provide an overview of the Srijan, the wireless sensor network macroprogramming framework used as a base for this work. In Chapter 5, we present the proposed modifications to Srijan for supporting fault-tolerance aspects. Related

work regarding efforts to better support WSN application development and fault-tolerance is shown in Chapter 6. In Chapter 7, we present qualitative and quantitative evaluation of our work. Finally, in Chapter 8 we list our final remarks and future work.

Chapter 2

Wireless Sensor Networks

Wireless Sensor Networks began with sensors and sensing systems. They were single systems deployed in one location to measure and process environmental properties. Later, those systems evolved to Sensor Networks: inter-connected wired platforms, each with its own sensors. Sensor Networks could sense and process data from different areas and reason on that data, such as aggregating weather measurements from reporting stations near airports.

By the end of the 1990s, new technology enabled the production of low-power processing units equipped with sensors and wireless antennas. Thus came Wireless Sensor Networks [SMZ07], a system integrating *automated sensing*, *embedded computing*, and *communication*. In Wireless Sensor Networks, each networked unit senses and processes a small area, and, working together via wireless communication in a distributed and collaborative fashion, they gather and process information over large areas.

The most basic sensor network characteristic is physical world interaction. By means of sensors, they read environmental properties, such as temperature, humidity, and motion tracking. These measurements can then be processed to generate high-level information sent to decision systems. By means of actuators, the network then acts on the environment, such as by toggling a switch or turning on a motor.

These networks captured researchers' attention, who saw wireless sensor networks as an "exciting emerging domain of deeply networked systems of low-power wireless motes with a tiny amount of CPU and memory, and large federated networks for high-resolution sensing of the environment" [SMZ07]. The research area quickly expanded as technological advances produced cheaper and more efficient motes, and as more and more people recognized the potential applications of these networks.

In this chapter, we present applications, characteristics, and classifications of Wireless Sensor Networks.

2.1 Key Applications

Traditionally, sensor networks have been used in the context of high-end applications such as radiation and nuclear-threat detection systems, "over-the-horizon" weapon sensors for ships, biomedical applications, habitat sensing, and seismic monitoring. Initially, applications focused on networked biological and chemical sensors for security. Nowadays, however, the existing and potential applications of sensor networks have been envisioned in the following areas [SMZ07]:

Military applications: Monitoring enemy and friendly forces and equipment; mili-

tary theater or battlefield surveillance; targeting; battle damage assessment; and nuclear, biological, and chemical attack detection

Environmental applications: Microclimates; forest fires and river flood detection; and precision agriculture

Health applications: Remote monitoring of physiological data; tracking and monitoring doctors and patients inside a hospital; drug administration; and elderly assistance

Home applications: Home automation; and instrumented environments

Commercial/Business applications: Environmental control in industrial and office buildings; inventory control; vehicle tracking; and traffic flow surveillance

2.2 Properties of Wireless Sensor Networks

Wireless Sensor Networks can be employed in many different contexts, with different kinds of devices and configurations. In this section, we explain some properties of Wireless Sensor Networks and their importance.

2.2.1 Node configuration

The choice of nodes can heavily influence the overall network. Nodes come in many types, and when choosing one type there is always a trade-off. Node prices range from low to high, depending on battery life, processing power, sensing accuracy, and so on. They vary in size from large boxes to very small particles; these size variations in turn influence the available computing, storage, and transmission capacity resources.

All these properties influence other aspects of the network: how long the network can be left unattended, the sensing interval, how often the nodes communicate, how many nodes can be used to cover an area, and so on.

2.2.2 Deployment type

There are two types of WSN deployment: ad-hoc or planned. Ad-hoc deployment is random, for example, sensors might be dropped by airplanes to cover remote areas. A planned deployment usually involves sensors placed at fixed locations, such as in a factory or office building.

Deployment can also be classified by its periodicity; either as a one-time event or as a continuous process, wherein new nodes are added to replace failed ones or to increase accuracy in certain areas of interest.

Deployment type affects important properties such as expected node density, location, regular location patterns, and the expected degree of network dynamics.

2.2.3 Node mobility

Nodes may change their location after the initial deployment. A certain degree of mobility can result from the environment, such as from wind or water or from animal interference. Nodes may also be attached in a controlled manner to mobile platforms or robots, or to animals.

In other words, mobility can be an accidental or desired network property. It can be intended, such as robots moving towards an area of interest, or unintended, such as wild animals moving network sensors.

The degree of mobility relates to the quantity of moving sensors in the network, how often they move, and how long the distance they dislocate is.

Mobility influences networking protocols and distributed algorithms design. Routing protocols, transmission tables, and sensor accuracy are all affected by mobility.

2.2.4 Network Topology

One important sensor network property is diameter, that is, the maximum number of hops between any two nodes. In its simplest form, a sensor network forms a single-hop network, with every sensor node able to directly communicate with every other node.

An infrastructure-based network with a single base station forms a star network with a diameter of two. A multi-hop network may form an arbitrary graph, but often one constructs an overlay network with a simpler structure, such as a tree or a set of connected stars.

Topology affects many network characteristics, such as latency, robustness, and capacity, as well as the complexity of data routing and processing.

2.2.5 Processing architecture

WSN processing architecture, which refers to where data is processed in the network, is somewhat coupled with topology and available nodes. Processing architecture can be: centralized in a node with high processing power and with better power supply; distributed among the nodes; or, hybrid, in which pre-processing occurs in the nodes, and the heavy duty processing is centralized.

2.2.6 Area Coverage

Network coverage describes how much of the area of interest the sensor nodes cover, taking into account each individual sensor's range. Coverage can be sparse or dense. With sparse coverage, the area of interest is partially covered; the nodes may not detect some events of interest, or may take longer to detect them. With dense coverage, the area is completely sensored, and therefore sensors are more likely to detect significant events.

Degree of coverage influences data processing. Robust system design requires high coverage, yet consequently also produces a larger data-set to analyze.

2.2.7 Node Connectivity

Connectivity relates to both each node's connection range, and their connection's periodicity. Each node's range defines how the node is connected to others, creating a graph. This graph can be connected or partitioned, with moving nodes transferring information from one another. Nodes can also disable their antennas to save battery, and thereby split the graph.

A network can be: fully connected; intermittent when nodes are sleeping; and sporadic when nodes lose communication for long periods.

Connectivity influences the networking protocols and data-acquiring methods.

2.2.8 Network size

Network connectivity and coverage requirements, as well as the area of interest's size, determine the number of nodes participating in a sensor network. Network size in turn determines protocol and algorithm scalability requirements.

2.3 Programming Tools

When talking about sensor network applications, for which the hardware and applications and very domain-specific, it is important to emphasize available tools for building such applications. Here we cite some of the tools used to create such applications in both low and high levels of abstraction.

2.3.1 Low abstraction level tools

The primary low abstraction level platforms for application building are very close to the operating system. Among them we cite:

TinyOS: This most widely used WSN operating system [Lev12] supports several models of motes (such as Mica, Telos, iMote, and others), and offers the nesC language for its applications. It is a C extension with some constructs specific for the TinyOS execution model, which is event-based with a FIFO scheduler. The OS and its language's main characteristic is that they are designed to minimumally impact the device's resource usage, specially the power supply. For this reason, it is not recommended for CPU-bound applications. The platform provides a simulator called TOSSIM.

Contiki: This open-source operating system is more recent than TinyOS. It has an event-based execution model, but is multi-threaded. Although it does not support as many models as TinyOS, it has considerable usage [BHG+13], especially within the Internet of Things paradigm, including an active developer community. The programming language is C. Contiki developers also provide a virtual machine with a pre-installed system and the SDK containing its simulator, called Cooja, which has ready-to-run example applications.

nano-RK: As opposed to TinyOS and Contiki, nano-RK is specifically designed for supporting real-time applications. It has a monolithic architecture and a thread-based development model. The programming language is C, the supported model is MicaZ, and the platform does not have a simulator.

2.3.2 High abstraction level tools

High abstraction level sensor network programming is commonly referred to as *macroprogramming*. Macroprogramming simplifies development of distributed applications, comprising wireless networked nodes, by abstracting away many of the complexities involved in orchestrating communication among them [GGG05, Kar10]. Since this is the paradigm we base our work on, we describe its great variety of frameworks and macroprogramming languages with different paradigms and applicable scenarios in further chapters.

We specifically focus on the Srijan framework, our work's basis, with a detailed description on Chapter 4. Other frameworks are described in Section 6.2 of Related Works.

2.4 Discussion

A wireless sensor network is a distributed system designed to interact with the real world. It has its own properties and constraints to overcome, and it is applicable to different scenarios. Its operating systems, frameworks, and languages ecosystem is vast and diverse.

In this chapter, we presented an overview of the possible applications for sensor networks, their inherent characteristics, and the main available platforms.

Chapter 3

Fault Tolerance in Wireless Sensor Networks

As discussed before, WSN applications are inherently subject to failures. Thus, building practical and reliable sensor network systems is a significant challenge [GKMG07]. WSNs combine many of the difficulties of traditional embedded systems with those of traditional distributed systems, including the need for proper synchronization and fault tolerance. WSN volatility poses a challenge to ambitious application goals, such as long-term, multiple-year deployments, large-scale networks of thousands of nodes, and highly reliable data delivery. As the WSN field matures, strategies for detecting (and possibly correcting) the anomalies that are inherent to their physically coupled low-end system design will only grow in importance [JWOV11].

Tolerating faults, or maintaining fault resilience, involves prevention, detection, and recovery. Prevention involves taking actions before the faults' occurrence to reduce the probability of the network breaking. Detection describes actively monitoring the wireless network system at network, device, or data-layers. Finally, recovery amounts to overcoming a system/component failure, which can be done by replicating a system component, discarding corrupted data, or even informing an administrator of the failure's occurrence [dSVB07, PH07].

3.1 Prevention

Fault prevention in wireless sensor networks aims to prevent failure by ensuring the network has enough sensors to provide redundancy, while still covering the desirable area. In other words, it aims for full coverage and connectivity. Prevention usually happens during the design and deployment phases. Most commonly, coverage and connectivity are expressed using the following metrics: (i) k-coverage, which means that any location in the network is monitored by at least k nodes, and (ii) k-connectivity, which means that the network can remain connected even if k - 1 nodes fail.

One maintains k-connectivity by employing k-connected routing algorithms in the network stack, which are usually implemented in the operating system. However, for these algorithms to provide good connectivity, the deployment must have a low node sparsity. With high sparsity, the area coverage might be high, but k-connectivity will be low. Huang et al. [HT05] try to solve this coverage problem by provident polynomial-time algorithms to analyze coverage given the area and number of sensors available.

Different routing algorithms exist to suit different network topologies, such as flat, hier-

archical and location-based protocols [AKK04, AY05]. These algorithms all aim to extend the sensor network's lifetime by reducing battery usage in transmission, while not compromising data delivery. The routing algorithm is usually chosen by the underlying platform or framework, and the developer does not have to actively choose one if not strictly necessary.

3.2 Detection

To detect faults, we must first understand how they occur. The work of Souza et al. [dSVB07], Paradis and Han [PH07], and Jurdak et al. [JWOV11] provide an overview of faults types that occur in WSN, their common sources, and their detection mechanisms.

Faults can be grouped into categories according to which network layer experiences failure when they occur. Commonly, they are grouped into network, node/device, sink, and data-related faults.

Network-related failures occur when there is a communication problem among network nodes. Several factors come into play when ensuring WSN node connectivity. Nodes can have some degree of mobility, even if small, such as deploying sensors on plants, that can alter their radio link range. Radio interference can also occur, especially in urban areas. Furthermore, when a network is too dense, message collision can occur if several spatially close nodes transmit bursts at the same instant [SPMC04]. Szewczyk et al. [SMP+04], for example, reported a delivery rate of only 58% of their deployment's messages.

Node-related failures can range from erratically behaving software, to failing hardware, to external factors destroying the node. When deploying in the wild, nodes can be destroyed or removed by animals and bad weather conditions. When left unattended, batteries may run out. Software bugs can also lead to node failures, such as memory leaks leading to application crashes, infinite loops, and high CPU usage, leading to quickly depletion of battery.

Sink-related failures, as defined by Sá et al. [dSVB07], occur on a higher network level. Sinks are devices that collect all the data in a certain region. Failures in these devices are especially significant, since they bridge the sensing nodes and the backend system, and are deployed in much smaller numbers [GY03]. Sinks are usually high-capacity devices, with a permanent power supply or a more efficient supply, such as batteries combined with solar cells [MOH04, LBV06]. However, sink-failures are detected using the same techniques as node-related failures.

Data-related failures occur when there are discrepancies in data collected by the sensing nodes or received by the sink. These failures can occur due to node hardware failure, bad sensor calibration, or, in case of reception, by transmission interference. Data failures can be perceived by establishing the relation between nodes, and comparing in time or space their data range [JWOV11].

Detecting each kind of failure depends on how they present, i.e. what is the measurable property affected by its occurrence. Detection also depends on how quickly one needs to find the fault, or at each network layer it needs to be detected – i.e,. a network layer fault can lead to data layer faults and be detected only in the higher level. When detecting faults at their layer of occurrence, the following metrics are commonly used [JWOV11]:

At the network-level, one can detect connectivity loss and intermittent connection by monitoring the packet delivery rate. If it is zero, this means the connection is lost; if there is high variability, it indicates intermittent connection. It is also possible to detect routing loops if a packet is received with the origin address equal to the forwarder, which might occur due to a failing node's damaged routing tables.

At the node and sink-level, it is possible to detect anomalies by the lack of interaction among nodes, node and sink, and sink and backend systems. Resets can be detected when a packet counter is zeroed. Also, for nodes running on batteries, it is possible to monitor the battery voltage decrease rate. If the battery decrease rate is too high in comparison to other nodes, it can indicate hardware malfunctions or software bug. This is useful, as monitoring CPU and memory usage would lead to too much power usage.

Finally, at the data-level, one can detect: temporal anomalies by analyzing a single node reading's value time series; spatial anomalies by comparing the variation among nodes in the same neighborhood; and spatiotemporal anomalies by correlating the metrics.

There are different ways to design a detection mechanism based on these metrics. One main difference is architecture type: centralized, in the sink or backend, and distributed or hybrid. As examples of fault detection mechanisms employed for each category, we can cite:

Detecting network faults in a centralized matter, Sympathy [RCK+05] acts as a network-related metrics collector and reasoning tool. All the metrics collected are transmitted within the application packets, and the network sinks run code to reason on the metrics. Octopus [JRBB11] is another centralized tool similar to Sympathy, but acts as a backend system visualization tool. Memento [RB06] is a distributed tool for detecting failures based on variance of time between packet arrivals, which can be used to detect network faults and node crashes. These tools are implemented on top of TinyOS.

Among tools for detecting node crashes, there are examples such as the aforementioned Memento. Chen et at.[CKS06] and Yuan and Zhang [YZ08] also provide distributed detection of device crashes. However, the latter is focused on sink nodes, using state-checkpointing of the sink to neighboring nodes, which can take the role of sink if it fails.

Data anomaly detection has the largest variety of techniques employed. Wang et al. [WLO+08], Krishnamachari and Iyengar [KI04], and Luo et al. [LDH06] use Bayesian classifiers to detect unreliable data. The first in a centralized way, and the latter two in a decentralized one. Both Obst [Obs09] and Chang et al. [CTB09] use recurrent neural networks. Obst uses a decentralized approach and Change et al. use a centralized one. Rajasegarar et al. [RLPB06] use a data clustering algorithm implemented in a hybrid manner: nodes build data clusters in a distributed fashion, which later can be aggregated and evaluated. Finally, Rajagopal et al. [RNEV08] provide a distributed online fault detection mechanism with an algorithm using statistical correlation of groups of measures taken by closely placed nodes.

3.3 Recovery

Although there are many sources for WSN system faults, recovery is mainly based on replication. Sá et al. [dSVB07] divide recovery techniques into two categories: active replication and passive replication. In active replication, the replicated nodes or tasks are active before failure, and in passive replication the replicated node or task is activated only when an instance fails.

Active replication is applied in scenarios in which all or many nodes provide the same functionality. For example, in sensing applications, all nodes have a sensing task that reads data and sends it to a sink. In this case, all nodes capable of sensing are deployed and active; if one fails, the sink can still collect the results from the other nodes. Some approaches to recovery using active replication are cited below, which can be used alone or combined.

> Multipath routing [GGSE01, AKK04] works at the network level to avoid network partitioning in response to node failure. In this sense, the goal is to provide a k-connected network, meaning the network remains connected when k-1 nodes fail.
>
> Sensor value aggregation, or sensor value fusion [WHVC05], seeks to derive high-level information from low-level sensor readings. The replication of sensor nodes can provide fault tolerance through tradeoffs between reading interval precision and the number of failures.
>
> Disregarding data from faulty nodes simply and effectively avoids propagating wrong values to tasks that aggregate or process them [dSVB07]. The challenge in this case comes from identifying the faulty data and deciding how much data can be discarded before reaching a state where no conclusive reasoning is possible.

One applies passive replication when maintaining task or node replicas is costly or otherwise undesirable. When it is applied, the primary replica alone actively processes input data. Only when the primary replica fails does a backup becomes active. When the application needs to maintain a state among replicas, there are two options: the active replica's internal state can be synchronized; or, input data can be forwarded to inactive replicas, which will not generate output. Given the constraints of WSN systems, applications usually have little or no state, minimizing the overhead of synchronization.

The replication itself is divided into two steps: node selection and service distribution. Node selection selects which node will take over the primary replica. It can be pre-selected, or, in cases where all nodes have equal capabilities, chosen in a distributed manner by self-election or group election. Service distribution describes the chosen node's service or task activation. In some cases, the code for all tasks is present in every node, and they only require a simple configuration change; in others, one must inject code into the node.

In both active and passive replication, the system can reach a state where there is no possible automatic recovery. When that happens, the reasonable solution is to inform a network administrator of the failures so they can be manually fixed.

3.4 Discussion

In this chapter, we presented the concepts of fault tolerance in Wireless Sensor Networks, as well as commonly used techniques for achieving fault tolerance in WSN applications.

These concepts and techniques are important to understanding the types of faults that occur in WSN, but they also underscore the challenges of dealing with faults in this context.

Chapter 4

Srijan, a WSN macroprogramming framework

Srijan is a graphical macroprogramming toolkit for simplifying WSN application development. Like other current WSN macroprogramming tools, it results from an academic realm; it was initially developed by Animesh Pathak during his doctoral course at the University of Southern California [Pat08]. Its first version was published in 2008, and since then it has been open source software [1] published under the Eclipse Public License. Although there is no known use outside academia, Srijan is often acknowledged as a data-driven macroprogramming tool [Oce08, dBSdR+07, BK07, Bis08, ZS09], a data-abstraction programming tool for sensor networks [BDU+12, CB11, LRST07], and as a network-level programming tool [LKK10, Ame11].

Srijan's framework is built upon the concepts defined by the Abstract Task Graph (ATaG) language [BPRL05], a wireless sensor network data-driven macroprogramming language where applications are defined using a mixed declarative-imperative code. Srijan further improves ATaG by providing graphical tools for specifying WSN programs and networks, and a compilation module able to tune ATaG compiler's parameters.

Srijan is composed of the following components:

Task Graph Specification GUI: implemented as an Eclipse plugin in which the user designs the abstract task graph representing the application.

Code Template Generator: a customizable code-generator module that generates templates for imperative code that the developer fills in with the application logic.

Target System Description GUI: in which the developer edits the target network's description.

Compilation and Deployment module: which takes the imperative template code and the system description to generate node-specific code, build node-level binaries, and deploy them to the target system nodes.

Figure 4.1 shows an application's development workflow in Srijan. Each component is followed by its output, which is connected as an input to another component of the framework. In the next sections, we provide an explanation of each component

[1] Available at http://code.google.com/p/srijan-toolkit/

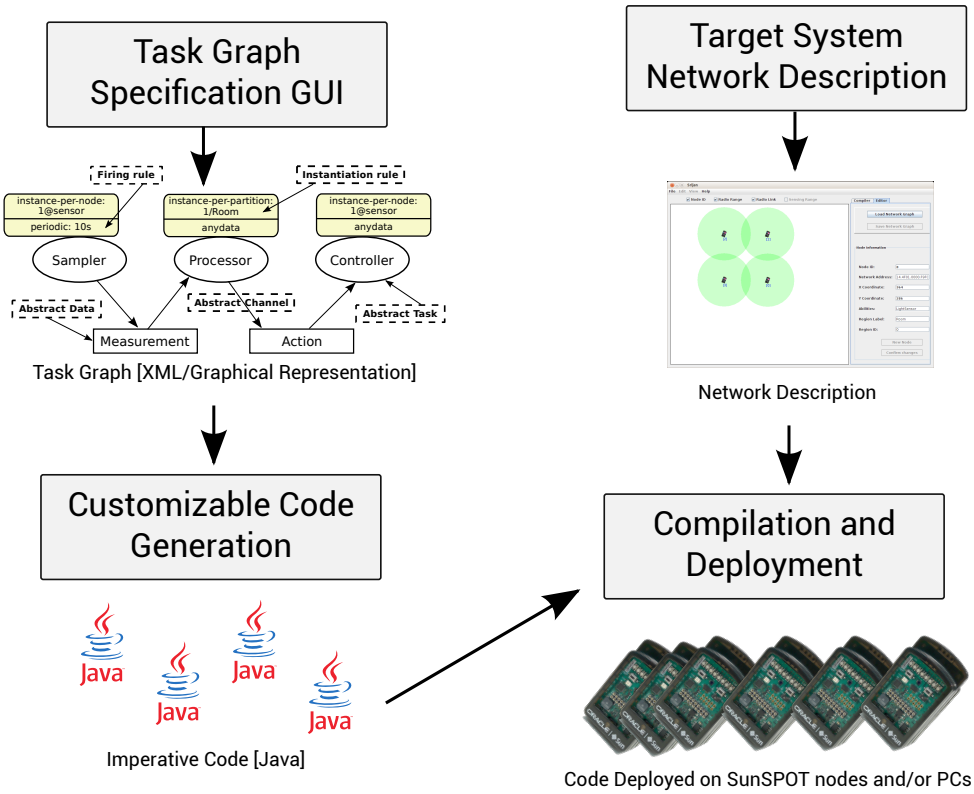

Figure 4.1: *Srijan components and flow*

4.1 Task Graph Specification and the ATaG language

Abstract Task Graph (ATaG) is a wireless sensor network macroprogramming language based on two concepts: a data-driven macroprogramming paradigm and a mixed imperative-declarative specification [BPRL05].

In data-driven programming, developers break up their application's functionality into processing units called tasks. Tasks interact with each other solely by means of data items that they produce and/or consume, and do not share state in any other way. A task can be scheduled for execution upon the availability of its operands by an underlying runtime system managing the data received from other tasks. This paradigm is attractive for several reasons:

- tasks can use data items at any needed abstraction level;
- tasks do not have to specify how the needed data is produced;
- there is no direct task-to-task coupling, making programs more extensible and reusable;
- can be easily supported by an event-driven runtime system.

Mixed imperative-declarative specification facilitates separation of functionality from other non-functional aspects, such as task placement and coordination. In WSN, this separation is desirable, as it enables a program, without modification, to be synthesized onto various deployments. These deployments may have different non-functional requirements that can be specified by modifying only the ATaG program's declarative portion, which is also useful for building GUI editors, such as Srijan.

WSN applications can have different interaction paradigms, which lead to some overspecialization of languages and tools. ATaG, however, has been shown to be useful for at least three common interaction paradigms, as described by Pathak et al. [PMB+07]:

Hierarchical Data Collection is used by the largest fraction of sensor networks applications. In this paradigm, usually several identical nodes are employed to sense the environment, and their data is compressed in the network and then sent to a base station.

Localized Interactions encompass applications in which nodes have to interact with other nodes in their vicinity before making decisions. These localized interactions are useful in scenarios such as tracking a moving object or finding the contour of an oil spillage. Localized interactions also differentiate WSN from traditional distributed applications, where the processors' geographical location is irrelevant.

Actuation Driven by Sensing, or "sense-compute-actuate," represents applications where data collected from sensors in a region is used to make decisions about actions to be taken in a possible different region. Examples of such applications include traffic control and fire-fighting.

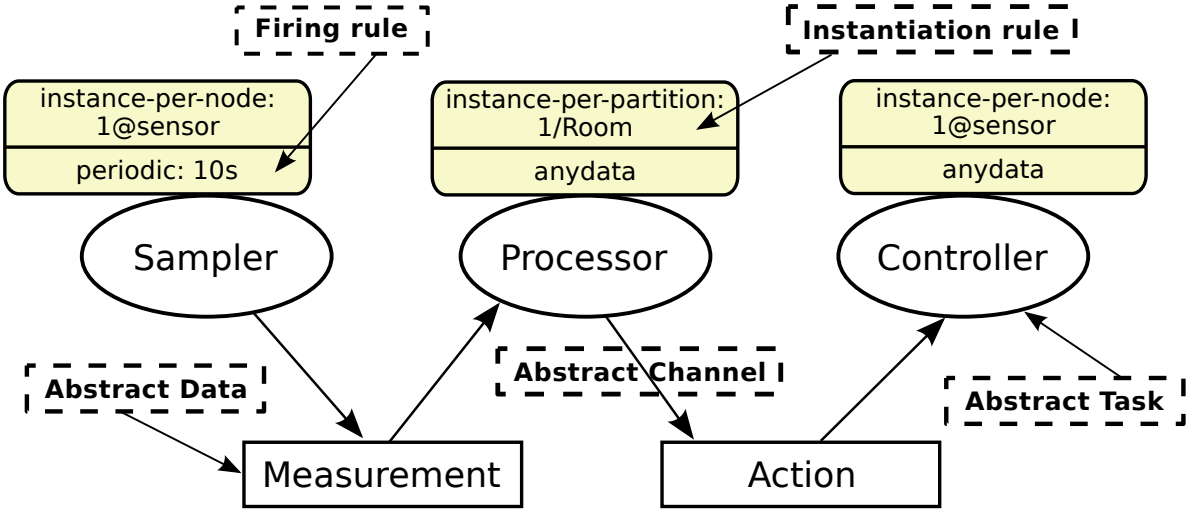

Figure 4.2: *Sample sense-compute-actuate application in graphical ATaG*

What makes ATaG so versatile is its declarative language portion, the Task Graph (Figure 4.2), which is a graphical description of the program containing the following components:

Abstract Tasks, representing the network's processing units. They are annotated with instantiation rules, specifying where they can be located, as well as firing rules, specifying whether a task is triggered periodically or upon receiving certain data item(s). Each task is labeled with a unique name by the programmer and has an associated imperative code in the target system programming language.

Abstract Data, represents a type of application-specific data object that can be produced and/or consumed by abstract tasks.

Abstract Channels, connects task declarations to data declarations. They represent not only data consumed or produced by tasks, but also are annotated with logical scopes expressing the interest of a task in instances of a data item.

In the majority of sense-compute-actuate applications, for example, sensing tasks are periodically fired, and only produce data items. They usually have an instantiation rule wherein one task per node contains the desirable sensor. Similarly, actuating tasks are instantiated in each node containing the desirable actuator, but will be fired upon receiving a certain data item. The computing tasks, however, aggregate data items produced by sensors, process them, and generate output designed either for another computing task or for actuating ones.

Example

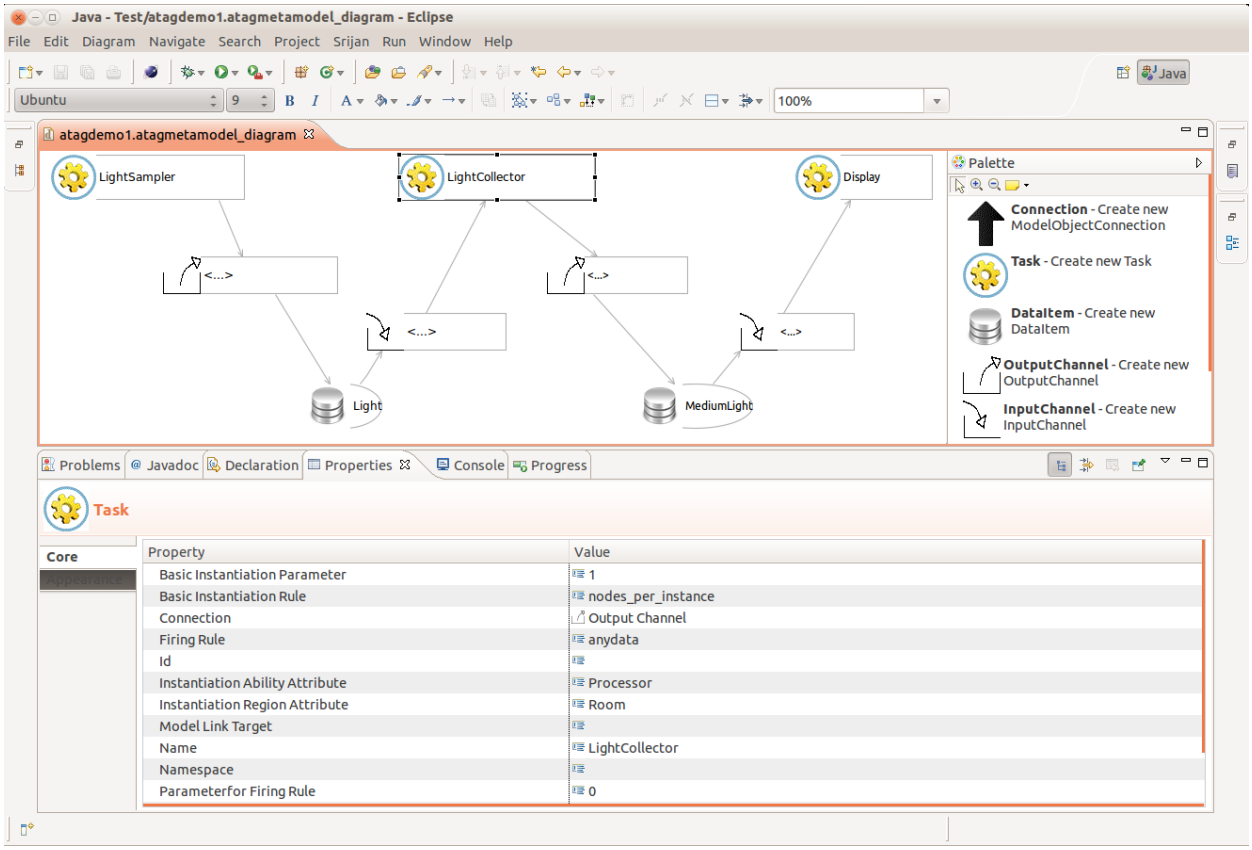

Figure 4.3: *Srijan's Task Graph editing GUI as an Eclipse Plugin*

Srijan's task graph GUI is implemented as an Eclipse plugin, which helps designing task graph specifications of abstract tasks, data, and channels together with their annotations. In Figure 4.3, we show a sample application for sensing light and actuating on a display. It contains three tasks:

LightSampler, a task responsible for sensing light, which is instantiated in every node with a light sensor.

LightCollector, a task responsible for gathering all light data from the samplers, which is instantiated in exactly one node per Room, which must have the "Processor" ability. It computes the current medium light value for the room and outputs a value for the controller.

LightController, is a task to be instantiated in each Room in nodes with actuators. Based on the collector's output, it may increase or decrease the light in the room.

4.2 Customizable Code Generator

The customizable code generator is responsible for generating code templates for each task and data item defined in the ATaG declarative part. These code templates need to be fulfilled by developers with the actual task logic or extensions to the data item. After they are completed, the framework can take the system description, the abstract graph, and the templates and generate the full code to be deployed into each network node.

The current implementation deals only with Java as the imperative language, targeting SunSPOTs and JavaSE devices. As such, there are four types of Java classes that are generated by this module:

Constants Each task and data item has an assigned integer id, which is matched to a final static variable in a wrapper class. It is generated only as a helper for naming tasks across the application.

Data items For each data item, a matching Java class provides methods for its serialization and deserialization. If the item was annotated with a schema in the declarative portion of ATaG, the class fields are automatically generated. If not, the developer has to implement the fields. They are generated as Java Beans with *getters* and *setters* for all fields. DataItems also have serialization and deserialization methods that read and write from the communication channel.

Tasks For each task, two classes are generated: an abstract class, which encompasses the communication with the framework and controlling states, and the concrete task class, which extends the abstract one and must be completed by the developer. Abstract Tasks are responsible for each task's boilerplate code and provide the methods for receiving data items meant for the task, and for sending data items produced by the task to the data pool. They extend ATaGAppTask, which implements Runnable and provides methods for retrieving the region the task is operating on.

Manager The manager contains the definition of logical scopes corresponding to each data item, so the runtime system can deliver it to the appropriate tasks as they are produced. It does not require any editing by the developer.

Example

We provide an example of generated code in Appendix A.

4.3 System Description

One of ATaG's characteristics is that its declarative part is deployment agnostic. That means application developers can use the same program specification to deploy into different wireless network configurations, or systems, according to Srijan's naming convention.

The system description GUI (Figure 4.4) is used to create a target system description, which the ATaG compiler takes as input during task allocation. This description consists of global parameters of the network, such as area covered, number of nodes, and radio range, and of the following parameters for each node:

- ID
- Physical Address (MAC)

- Physical Position (coordinates)
- Abilities (Sensors and/or Actuators attached)
- Partition/Region
- Node type (BaseStation, Node, etc)

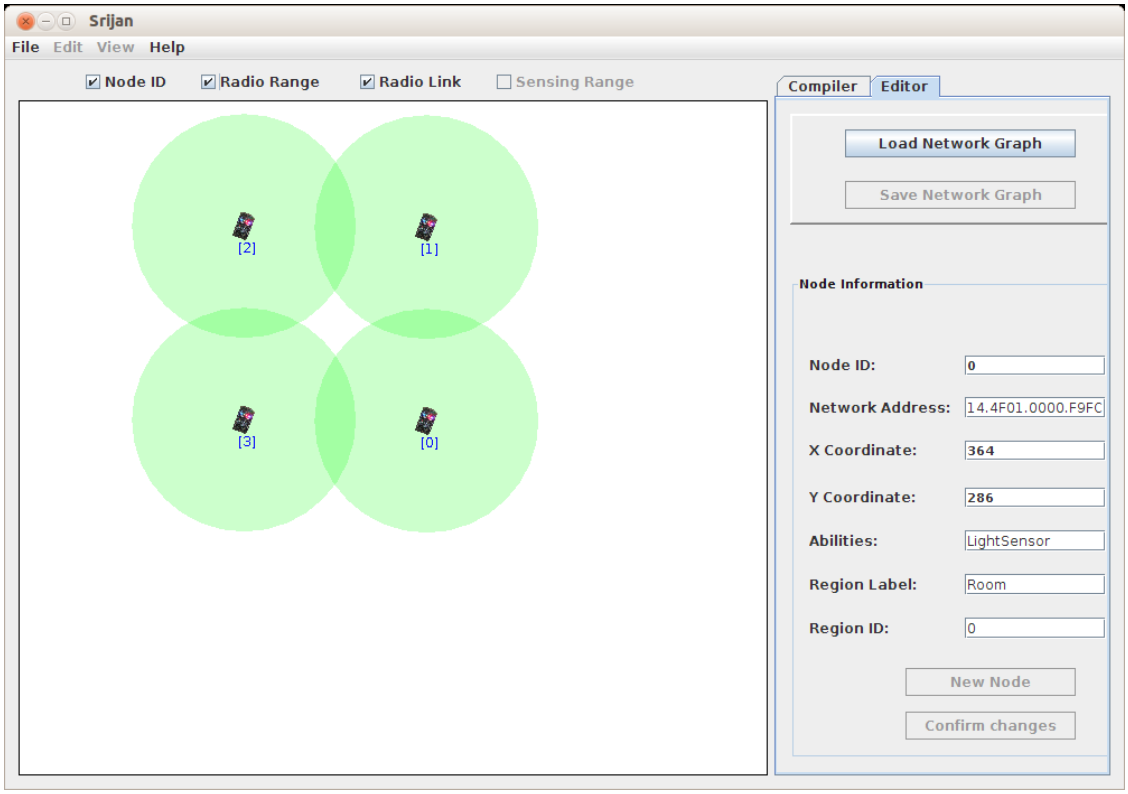

Figure 4.4: *Srijan system description GUI*

Example

```
10 1000 1000 100 100

00 0014.4F01.0000.79B5  20 20 Room:0 Processor               hostspot
01 0014.4F01.0000.F9FC  20 20 Room:0 LightSensor             freespot
02 0014.4F01.0000.78E3  20 30 Room:0 LightSensor,LEDActuator freespot

03 0014.4F01.0000.72B3  20 20 Room:1 Processor               hostspot
04 0014.4F01.0000.7A61  20 20 Room:1 LightSensor             freespot
05 0014.4F01.0000.79B1  20 30 Room:1 LEDActuator             freespot

06 192.168.1.10         20 20 Room:2 Processor               j2se
07 0014.4F01.0000.7A61  20 20 Room:2 LightSensor             freespot
08 0014.4F01.0000.74E1  20 30 Room:2 LightSensor,LEDActuator freespot
09 0014.4F01.0000.7CA5  20 30 Room:2 Processor hostspot
```

Listing 1: *Sample network description file*

4.4 Compilation and Deployment

The compilation and deployment module provides the application developer the ability to tune the compilation parameters, generate node-level code, and deploy the generated code to the nodes in the target system. Currently, the ATaG compiler supports only random optimization, and the module enables the randomization seed setting. However, the toolkit can be extended to support more optimization techniques as they are developed [PP10].

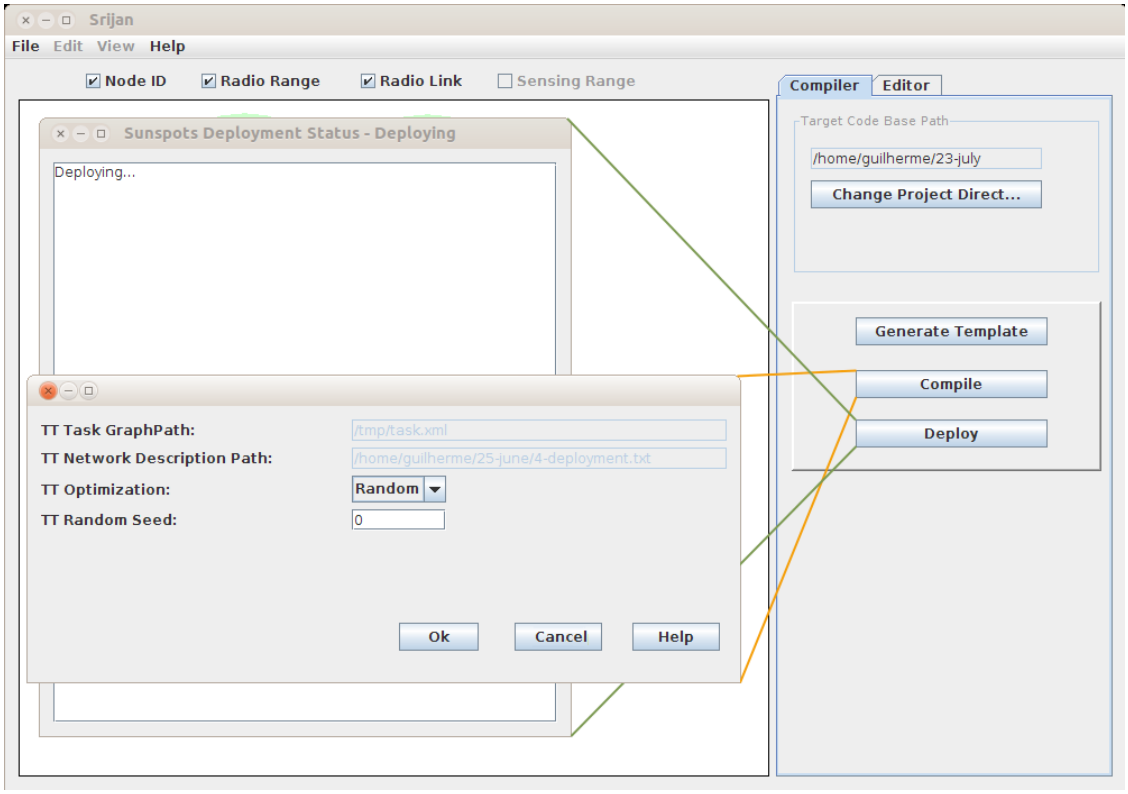

Figure 4.5: *Srijan compiler and deployment module*

4.5 Discussion

In this chapter we presented the Srijan framework, a macroprogramming platform for wireless sensor networks. This framework is used as a basis for our fault tolerance extensions. As such, we presented the data-oriented paradigm this framework follows, as well as the tools it is composed of, and code examples.

Chapter 5

Supporting Fault-Tolerance in Srijan

This work aims to implement fault-tolerance features into Srijan, a macroprogramming framework for Wireless Sensor Networks. In this chapter, we present the features we added and discuss how we implemented them.

We named this implementation **Srijan-FT**.

5.1 Fault tolerance features

To deal with WSN application faults, actions can be taken for prevention, detection, and recovery. Prevention relates to active replication, dealing with the problem of maintaining communication and having as many active sensor nodes as possible. Communication guarantees are commonly given by maintaining the network infrastructure as connected and redundant. Communication is also most commonly treated in hardware or operating systems [PH07, JWOV11]. Detection and recovery can be performed using different techniques, as discussed in Chapter 3, and are dealt with at the operating system or application level. In this work, we decided to deal with detection and recovery of network node unavailability, or *crashes*, and data anomalies from sensing devices. In the following subsections, we explain which methods we employed for each.

5.1.1 Crash faults

Crash faults are rooted in a single node's hardware or software problems. These faults' observable symptom is that the node stops transmitting data, and thus the other nodes stop receiving data. Because crashes are linked to a single node, detection, and sometimes recovery, can be highly effective [JWOV11].

There are several sources of crash faults, such as battery depletion, poor or defective hardware components, external agents destroying the device, overuse of resources, etc. One can locate the fault's source by actively monitoring devices and their resource usage, however it is costly and not always useful. We decided to support fault recovery without information about what caused them.

Recovering from crash faults involves replicating the available tasks on the crashed device. By the sense-compute-actuate convention, sensing tasks are already fault tolerant against crash, since they are replicated *per-se* by being instantiated in every single node capable of receiving them. Computing tasks, on the other hand, are not supposed to run on every node, and to tolerate crashes they should have some sort of replication associated with them.

Computing tasks are the ones receiving and processing the sensor data, e.g. the LightCollector example from Section 4.1. They are most often run on sinks, devices connected to a large or constant power supply that has a higher processing power. These sinks are usually deployed on a one-per-area basis. Therefore, to support fault tolerance against crash, the application developer must provide at least one other sink as a replicating device per region.

We implemented methods for automatically replicating processing tasks when redundant sinks are available.

5.1.2 Data anomalies

Data anomalies can occur due to hardware malfunctions, software bugs, or external factors altering the sensing device state. To detect such anomalies, it is necessary to analyze the sensing-data time series and the correlation among neighboring nodes. Once detection is done, we can leverage active sensor nodes' replication and ignore faulty data in the overall computation.

Detection can take place in either a centralized or distributed fashion. Since Srijan targets sense-compute-actuate applications, and computing tasks are usually run on sinks, we decided to implement centralized data anomaly detection. Although we imposed on the computing node a greater processing requirement, in a decentralized approach this processing power would be necessary on sensing nodes, which have less processing and power capability.

Based on common techniques for time-series data anomaly detection [GGAH14], we decided to use a simple algorithm using the moving average to identify and discard outliers. This is a simple algorithm, but sufficient enough to show that it can be integrated into the framework.

5.2 Runtime System

To implement these fault-tolerant features, we first had to understand how each node's runtime system was laid. We had to define the hook points of the framework where the fault tolerant modules would be plugged. The ATaG runtime system is mainly organized in three layers: network communication, framework control logic, and user-defined tasks.

The control layer, called ATaGManager, is composed of runtime classes managing the control of the application and node information. The most important class is also named AtaGManager, and is responsible for setting up the node once it is started. In Srijan, all nodes have the code for all tasks. What distinguishes them is the configuration performed by the ATaGManager. To support modifications of the configuration in runtime or to start or stop a task based on events, it is necessary to extend the manager to control tasks based on input from a fault tolerance module. Since not all tasks require fault tolerance, and the FT protocol is strongly related to tasks, we decided to insert hooks in: (i) the Abstract Task classes, which will talk to the FT protocol when the task needs fault tolerance, and (ii) the ATaGManager, which will load the protocol for the tasks and receive calls from the protocol to start or stop tasks (Figure 5.1).

5.3 The Fault Tolerance Module

The initial implementation of this work was done as part of the MURPHY project[1], in partnership with LAAS' dependability analysis laboratory. As such, the fault tolerant

[1] http://cedric.cnam.fr/~sailhanf/murphy

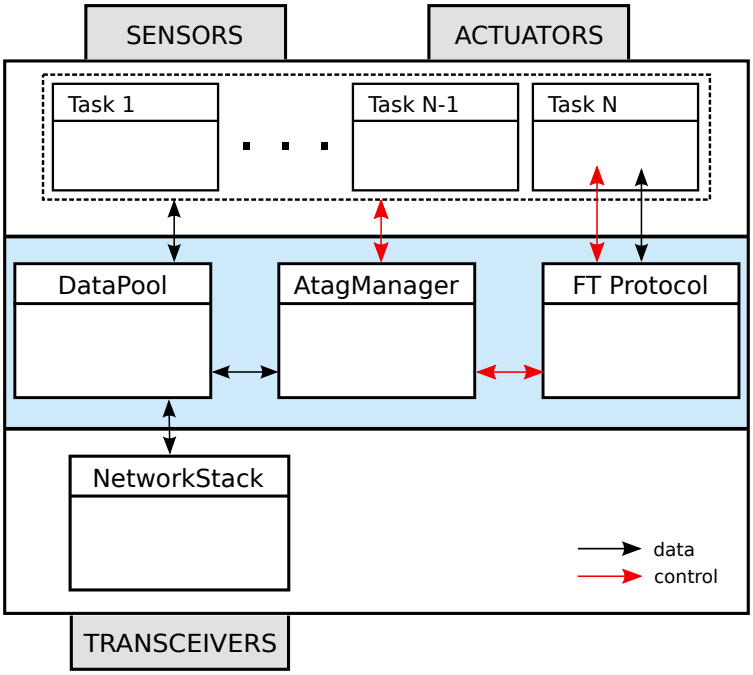

Figure 5.1: *Runtime system with fault tolerance protocol*

module is heavily based on the fault tolerance protocol defined by LAAS. It leverages Service Component Architecture (SCA) to define a generic component system (Listing 2). The fault tolerant methods implement a common interface and basic operations, which are wired to the protocol at runtime. These basic operations are:

`syncBefore`: a method that is called before the computation is performed

`proceed`: a method that is called when the computation is allowed to continue

`replyLog`: used to save information after proceed is executed

`syncAfter`: a method that is called after the computation is performed

```
1  package fr.inria.arles.srijan.ftm.api;
2
3  public interface FTMProtocolAPI {
4
5      void executeBefore(byte[] s);
6
7      void replyLog(int id, String s);
8
9      void executeAfter(byte[] s);
10
11     Object proceed(FTTask task, Object o);
12
13 }
```

Listing 2: *Generic fault tolerance protocol interface*

The rationale behind this basic protocol is that such generic operations can be used to implement a great number of replication and verification methods.

5.3.1 Crash Faults

The mechanism to treat crash faults works by replicating sink/processor tasks, which are deployed on nodes with larger processing capabilities and power supplies.

The two methods chosen are Primary-Backup replication and Leader-Follower replication [SFR12]. Leader-Follower replicates systems by sending the input data to both master and slave nodes, with only the master producing the output, whereas Primary-Backup replicates systems by sending snapshots of the master device's internal state to the slave (shown in Figure 5.2). The former is only suitable for deterministic systems, as it assumes the same input will always produce the same output. The latter is suitable for any system.

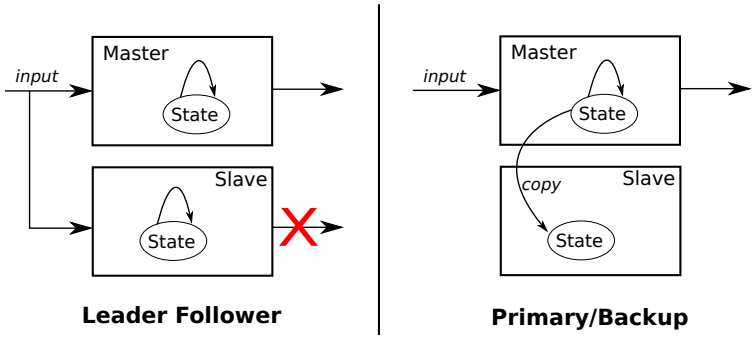

Figure 5.2: *Primary-Backup vs Leader-Follower replication*

We implemented a prototype application using both methods integrated into the ATaG runtime system, demonstrating how they can work together. We also began improving the code generation tool to support such integration.

With crash faults, the FT protocol is used as follows: LFR implements only the SyncBefore operation, which is used to forward inputs, and PBR implements only SyncAfter, which is used to sync states among replicas after the computation is performed.

In Srijan, the protocol is instantiated in ATagManager, as the code sample 23 shows. We use the FraSCAti SCA framework [SMR+12] to load the composite files with the proper wirings for each method. Depending on which node type loads the protocol, different actions are taken: if it is a master node, one that is active, the protocol is returned. If it is a slave node, one that is inactive in the beginning of the execution, we configure ATagManager as a task starter service in the fault tolerance protocol. If the master fails, the protocol calls a method asking the manager to start the task in the slave replica. A reference to the task is also set in the controller, if the node is a slave replica.

FTTaskController is an interface the SlaveServer implements. The server receives inputs from the corresponding methods in the master protocol, and dispatches them to the fault tolerant task to process. This combines with the abstract task class, which implements the FTTask interface and provides methods to handle these generic calls: SyncAfter is used to deserialize internal states into the concrete class' annotated fields, whereas SyncBefore is used to forward inputs to the task without generating output.

In the master replica, the FT protocol is called with the result of state serialization. Serialization takes place by looping through all fields of the concrete class and checking for the *@FTState* annotation. The state is then sent to the slave via the syncAfter call.

The aforementioned code parts are available in Appendix B.

5.3.2 Data Faults

Data fault tolerance mechanism implementation is much simpler than Crash Fault. For one, a separate detection mechanism is not needed. Detection occurs upon receiving a value and passing it through the algorithm. It is also simpler because there is no need to synchronize states among motes, since the analysis is centralized.

The algorithm uses only the SyncBefore and Proceed hooks. Upon receiving a data item, the fault tolerant task will send it to the FT Protocol, which will execute the SyncBefore hook. The SyncBefore hook is then plugged to a data validation module that will compare the current value to a configurable number of previous values in the time series and decide if it is valid.

If the data item is valid, the computed task is called in the Proceed hook. In case the data is invalid, it can be disregarded, or a network administrator can be contacted and informed.

In our approach, a simple algorithm to identify outliers in the time series was employed: to calculate the moving average of the last N readings and consider any value outside of $x * std\_deviation$ as an outlier. X can be defined as a parameter, but has a default value of 3, which is approximately 97% of the values in the normal distribution.

Figure 5.3 shows an example of how this approach works. In the figure, the moving average is shown in a black line at the center. With a discrete increase in the absolute values of readings, the first values are considered errors. Then, the moving average adjusts and they fall within the variance window again.

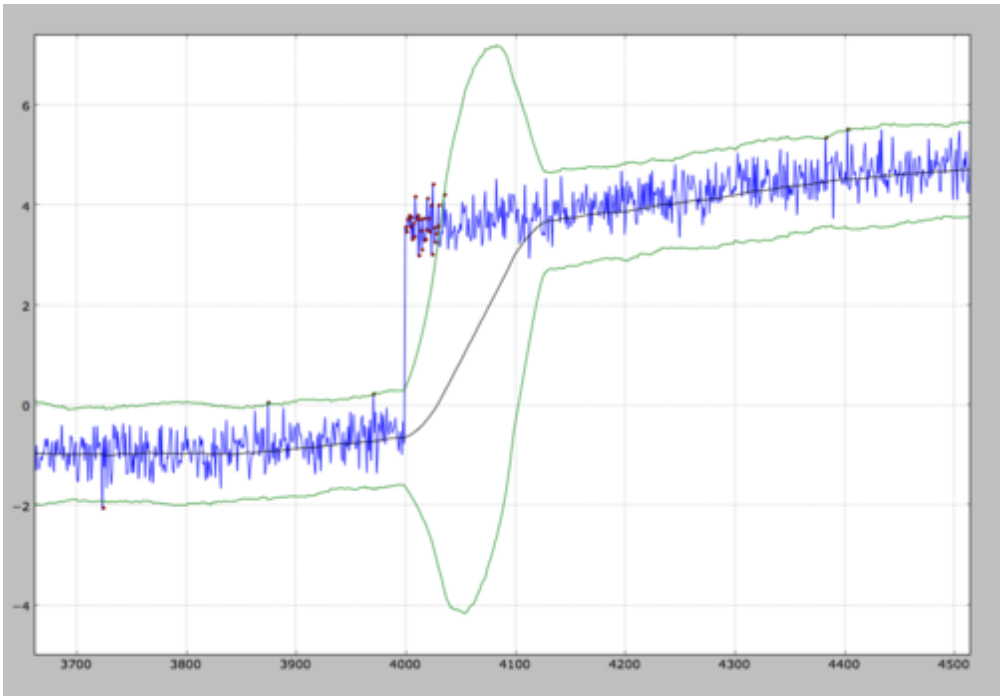

Figure 5.3: *Moving average filter with variance window*

5.4 Code Generation

We have added simple modifications to the Srijan compiler, enabling automatic generation of tasks supporting LFR and PBR fault tolerant replication mechanism, as well as using the moving average method for data fault detection. Adding support for LFR and PBR re-

quired significantly larger change, whereas data fault detection only involved the template processing.

Code generation consists of two main tasks: node selection and template processing. Node selection allocates tasks to specific nodes, generating a task mapping. In these initial steps, we randomly assigned fault tolerant tasks. For each available region, the algorithm presented in Listing 3 will cycle through the available nodes and select a master and a slave node, if available. The node type information is stored in the nodes' attribute map.

After the node mapping is created, each task's templates is processed for the assigned nodes, using the node's attributes when necessary. For example, to process the composite file for PBR master node, the slave node's IP address needs to be wired to the SyncAfter call, as shown in line 7 of Listing 5.

```
1  private void assignFaultTolerantTask(
2              HashMap<Integer, Set<Integer>> partialAssignment,
3              ATaGTaskDeclaration at, ArrayList<NodeInfo> targetNodes) {
4          // fault tolerant task requires j2se node
5          Iterator<NodeInfo> it = targetNodes.iterator();
6          while (it.hasNext()){
7              NodeInfo node = it.next();
8              if (!node.getAttributeByName("type").equals("j2se")){
9                  it.remove();
10             }
11         }
12
13         if (targetNodes.size() == 0){
14             // TODO Oh! Snap! No j2se nodes!
15         } else if (targetNodes.size() == 1) {
16             // only room for a master node, and this is the one!
17             NodeInfo ni = targetNodes.get(0);
18             partialAssignment.get(ni.getMyId()).add(at.getID());
19             at.assign(ni.getMyId());
20             ni.addNodeAttribute(new IntegerAttribute("ft-node-type", 0));
21         } else {
22             // master and slave!
23             Random rand = new Random();
24             int master = rand.nextInt(targetNodes.size());
25             int slave = rand.nextInt(targetNodes.size());
26             while (master == slave) slave = rand.nextInt(targetNodes.size());
27             NodeInfo masterNode = targetNodes.get(master);
28             NodeInfo slaveNode = targetNodes.get(slave);
29             String masterIP = (String) masterNode.getAttributeByName("physicaladdress");
30             String slaveIP = (String) slaveNode.getAttributeByName("physicaladdress");
31
32             masterNode.addNodeAttribute(new IntegerAttribute("ft-node-type",1));
33             masterNode.addNodeAttribute(new StringAttribute("ft-pair-ip", slaveIP));
34
35             slaveNode.addNodeAttribute(new IntegerAttribute("ft-node-type",2));
36             slaveNode.addNodeAttribute(new StringAttribute("ft-pair-ip", masterIP));
37
38             partialAssignment.get(masterNode.getMyId()).add(at.getID());
39             partialAssignment.get(slaveNode.getMyId()).add(at.getID());
40             at.assign(masterNode.getMyId());
41             at.assign(slaveNode.getMyId());
42         }
43     }
```

Listing 3: *Fault tolerant task: node assignment using random selection*

The template files are created using the Apache Velocity template language[2]. Each template is merged with a context, which contains the necessary variables. Listing 4 shows an example of how a template can be merged. In that snippet, three variables are put in the context ("taskName", "isGUI", and "taskID") and merged with the *abstract_ class.vm* template file.

[2]http://velocity.apache.org/

We created and modified template files for composite files, abstract task classes, concrete classes and the ATag manager.

```
public class AbstractClassGenerator extends BaseGenerator {
    String PATH = "./velocity/templates/java/abstract_class.vm";

    public void generate(ATaGTaskDeclaration t, boolean isGUI, String dartRoot) {
        VelocityContext context = new VelocityContext();
        context.put("isGUI", isGUI);
        context.put("taskName", t.getName());
        context.put("taskID", "T_" + t.getName().toUpperCase());

        Template template = null;
        template = Velocity.getTemplate(PATH);
        if (template != null) {
            StringWriter writer = new StringWriter();
            template.merge(context, writer);
            writeFile(writer, t.getName(), faultTolerant, dartRoot);
        }
    }
}
```

Listing 4: *Merging a velocity template file*

```
    <component name="syncAfter">
        <implementation.java class="fr.inria.arles.srijan.ftm.pbr.SyncAfter" />
        <service name="execute">
            <interface.java interface="fr.inria.arles.srijan.ftm.api.TriggerSyncAfterService" />
        </service>
        <reference name="synchronizeService">
            <frascati:binding.rest uri="http://${pair_ip}:8069/syncAfter" />
        </reference>
    </component>
```

Listing 5: *PBR composite template: SyncAfter snippet*

5.5 Discussion

In this Chapter we presented our implementation of fault tolerance features into Srijan, therefore creating our Srijan-FT extension. We presented the chosen techniques for dealing with crash faults, and data anomalies. We presented the internal runtime system modifications, as well as code generation done by the tool.

This represents our core contributions to improving macroprogramming for Wireless Sensor Networks, and in the next chapters, we present related works, the evaluation of Srijan-FT, our conclusion and future work on the platform.

Chapter 6

Related Work

In this section, we present related work in high abstraction platforms. As such, we divide them into two main categories.

In the first category, we discuss WSN middleware and operating systems, which are the basis for creating WSN applications as well as the first step towards making it easier to define and administrate WSNs.

The second category describes macroprogramming languages, focusing on their programming characteristics, and how they deal with faults. This category provides works in high-level abstractions for Wireless Sensor Network programming, and is more closely related to Srijan and our enhancements.

We used the following queries and online databases to find related work, and systematically selected works based on abstracts and full text:

> **query:** ("Wireless Sensor* Network*" OR WSN) AND (middleware OR platform OR macroprogramming OR high-level)
>
> **fields:** title AND abstract
>
> **sources:** Google Scholar, IEEE Xplore, ACM Digital Library

6.1 Middleware and OS

Although Middleware and OS are on a lower level of abstraction than macroprogramming, the majority of wireless sensor networks applications are written with them. For this reason, it is important to describe not only their advances in fault-tolerance, but also how applications can be programmed using this low-level approach.

The scarcity of computing and communication resources on wireless motes makes traditional middleware and operating system architectures costly. Current approaches to WSN application development favor architectures where the stack is highly application-specific, or even deployment-specific, rather than application-agnostic [MP12].

TinyOS [LMP+05], for example, is one of the most frequently used operating system for WSNs, and its system is component-based during compilation time only. The compiler assembles together the different components required by the application, and generates a static runtime monolithic system image. MagnetOS [BBD+02], like TinyOS, is monolithic, yet based on a virtual machine. Other systems, such as Contiki [DGV04], are modular operating systems, but still provide a very tight system-application integration. Contiki uses components at runtime in the form of local services: everything, from communication

to device drivers and sensor reading, is implemented as a system service, for which the application only knows the public interface.

Middleware are implemented on top of an operating system, or several, extending or encapsulating it inside a virtual machine. TinyOS is most often used as a base for building middleware, and several middleware, such as Matè [LGC04], Agilla [FRL05], Autosec [HV01], Mires [SGaV+04], and EnviroTrack [ABC+04], are implemented on top of it. Middleware are also often component-based, with the further requirement of being lightweight. Such component-based middleware include: RUNES (Reconfigurable, Ubiquous, Networked Embedded Systems) [CCG+07], which allows network runtime reconfiguration; and the loosely coupled Component Infrastructure (LooCI) [HTH+09], which is implemented in JavaME, supports a wide array of sensor devices, and favors heterogeneity.

WSN middleware are built to facilitate application development by focusing on one aspect of it, such as security, database-like querying, and so on. However, fault tolerance has been over-looked. Mottola and Picco [MP11] write that " Upon failure, current middleware lets the WSN break down in unpredictable ways, as the run-time support provides no guarantees in these situations. Transient faults (e.g., incorrect sensors readings) are usually not considered. Software errors are often fatal, yielding an erratic node behavior. To make things worse, faults at given nodes often affect others, causing a "domino" effect that ultimately renders the WSN unusable. These issues will become more and more important as WSNs become part of safety-critical systems. WSN middleware should provide known failure modes, along with tools and abstractions helping developers to understand the system behavior in these exceptional circumstances."

This scenario corroborates the need for easily available fault tolerant components for WSN application development. As such, we have identified one recent work regarding fault-tolerant middleware for WSN: the FlexFT framework [BUdAC13].

The FlexFT framework is a generic component-based framework for the construction of adaptive fault-tolerant systems. It relies on the "Sensor Web" paradigm of the Open Geospatial Consortium (OGC) to provide a standardized and interoperable interface for sensor observation. The authors implemented a Java prototype of the framework, and tested it against two example scenarios. Significantly, although similar to the work presented here, whereas Beder et al. focus on the component-based fault-tolerance aspect, we focus on providing fault tolerance features in macroprogramming. A more detailed comparison with FlexFT appears in Section 7.2.2.

Programming in TinyOS

TinyOS is heavily used, both as base for middleware and as a developers' programming platform. For this reason, we provide an example of TinyOS programming.

In TinyOS, each component is an independent computational entity exposing one or more interfaces, and built upon three abstractions: commands, events, and tasks. A task is used for intra-component concurrency, a command is a request to perform some action, and an event signals a command's completion.

Multithreading support was not available in TinyOS' earlier versions, and application development strictly followed the event-driven paradigm. However, since version 2.1, TinyOS provides support for multithreading using TOS Threads [KLP+09]. The authors point out that, given the resource constraints of the motes, event-based systems enables greater concurrency, but preemptive threads offer an intuitive paradigm.

TinyOS' scheduler uses a non-preemptive FIFO algorithm, and therefore is not suitable for real-time application. However, because of this, TinyOS's latency is much smaller than

others, since task creation simply means assigning a task's function pointer to a ready queue, and it does not need memory to be allocated.

Programming applications for TinyOS are made in nesC, a dialect of the C language. Code written in nesC can be run on actual motes or simulated using TOSSIM [LLWC03]. TOSSIM simulates the network stack at bit level, which allows low-level experimentation, but also provides a GUI tool, which can visualize and interact with running simulations.

An example application that periodically samples a node's default sensor and displays the bottom parts of the readings on the leds of the same node is shown in Listings 6 and 7. Listing 6 is the header file, specifying which components the application uses and defining the components' wiring. For example, on line 6, the MainC component, which represents the entry point of the application, is wired to SenseC.Boot.

```
1  configuration SenseAppC { }
2  implementation {
3
4    components SenseC, MainC, LedsC, new TimerMilliC(), new DemoSensorC() as Sensor;
5
6    SenseC.Boot -> MainC;
7    SenseC.Leds -> LedsC;
8    SenseC.Timer -> TimerMilliC;
9    SenseC.Read -> Sensor;
10 }
```

Listing 6: *TinyOS sensing application: header file*

```
#include "Timer.h"

module SenseC {
  uses {
    interface Boot;
    interface Leds;
    interface Timer<TMilli>;
    interface Read<uint16_t>;
  }
}

#define SAMPLING_FREQUENCY 100
implementation {

  event void Boot.booted() {
    call Timer.startPeriodic(SAMPLING_FREQUENCY);
  }

  event void Timer.fired() {
    call Read.read();
  }

  event void Read.readDone(error_t result, uint16_t data) {
    if (result == SUCCESS){
      if (data & 0x0004)
        call Leds.led2On();
      else
        call Leds.led2Off();
      if (data & 0x0002)
        call Leds.led1On();
      else
        call Leds.led1Off();
      if (data & 0x0001)
        call Leds.led0On();
      else
        call Leds.led0Off();
    }
  }

}
```

Listing 7: *TinyOS sensing application: implementation*

6.2 Macroprogramming languages/tools

Macroprogramming, a programming paradigm, simplifies the development of distributed applications comprising wireless networked nodes by abstracting away many of the complexities involved in orchestrating communication among them [GGG05, Kar10]. The main idea behind macroprogramming is to provide programming constructs that holistically address network behavior. As a result, the compiler gains program execution freedom, which enables optimizations that are out of reach for low-level languages. However, it significantly adds complexity to the compiler [Kar10]. Most of the macroprogramming languages limit developer's expressiveness in comparison to low-level languages, and instead encapsulate in a simpler programming interface details such as network communication and thread/task synchronization.

In this section, we describe macroprogramming languages, acknowledging their authors, their characteristics, and their different approaches to WSN application programming. We also classify each according to the following aspects:

Target Application Type: Does the language target *sense-only* or *sense-and-react* (sense-compute-actuate) applications?

Spatial Visibility: Are the nodes grouped into regional clusters or abstractions, or do they act as a single global network?

Language Paradigm: Is the framework programming language imperative, declarative, or does it follow a hybrid approach?

Fault-tolerance Features: Does the macroprogramming framework provide any guarantees against faults?

Active Development: Is it still under active development? Where can it be found?

6.2.1 Kairos

Kairos [GGG05] is a macroprogramming framework with a programming model that uses a centralized approach to specify the distributed sensor computation's global behavior. In other words, it provides an abstraction of the network as a collection of nodes that together execute the same task. It was developed in 2005 at the Embedded Networks Laboratory [1] of the University of Southern California in 2005. Kairos and the ideas behind it relate to shared-memory-based parallel programming models implemented over message-passing infrastructures.

Kairos provides the programmer with three constructs: reading and writing variables at nodes, iterating through one-hop node neighbors, and addressing arbitrary nodes. These three simple constructs can be implemented on top of any existing native language, as the framework's code generation is implemented as a language preprocessor add-on to a native language compiler. The authors argue that these constructs are natural for expressing computation in sensor networks, stating that, intuitively, sensor network algorithms process data generated at individual nodes, often by moving such data to other nodes. They also state that by enabling the programmer to express computation by manipulating variables at nodes, the use of "textbook" algorithms is almost direct.

Significantly, Kairos achieves very low overhead by using eventual consistency among nodes. They argue that individual intermediate node states are not guaranteed to be consistent, but, in the absence of failure, the computation eventually converges. Because of that, Kairos' runtime loosely synchronizes state across nodes: a read call to a remote object blocks only until the referenced object is initialized and available at the remote node. After the object is initialized, further read calls are not blocked. This allows nodes to synchronize changed variables in a lazy manner, thereby reducing communication overhead. When eventual consistency is inadequate, though, Kairos provides a tighter consistency model.

Listing 8 contains a snippet written in Kairos to compute the shortest-path routing tree. In the code we see the node and *node_list* data types being used. *get_available_nodes()* is used to return the complete set of nodes in the network, whereas *get_neighbors()* is used within a node to obtain a list of its one-hop neighbors. Finally, the @ operator is used to retrieve the *dist_from_root* variable at the *iter2* node.

[1] http://enl.usc.edu/projects/kairos/

```
1  void buildtree(node root)
2    node parent, self;
3    unsigned short dist_from_root;
4    node_list neighboring_nodes, full_node_set;
5    unsigned int sleep_interval = 1000;
6
7    full_node_set = get_available_nodes();
8    for (node temp = get_first(full_node_set); temp != NULL; temp = get_next(full_node_set))
9      self = get_local_node_id()
10     if (temp == root)
11       dist_from_root = 0
12       parent = self
13     else
14       dist_from_root = INF
15     neighbors = create_node_list(get_neighbors(temp))
16   full_node_set = get_available_nodes()
17   for (node iter1 = get_first(full_node_set); iter1 != NULL; iter1 = get_next(full_node_set))
18     for(;;) //Event Loop
19       sleep(sleep_interval);
20       for (node iter2 = get_first(neighbors); iter2!=NULL; iter2=get_next(neighbors))
21         if (dist_from_root@iter2+1 < dist_from_root)
22           dist_from_root = dist_from_root@iter2+1;
23           parent = iter2;
```

Listing 8: *Shortes-path routing tree using Kairos*

Classification

Target Application Type: Sense-only.

Spatial Visibility: Global.

Language Paradigm: Imperative. The language constructs are implemented as extensions to the Python programming language.

Fault-tolerance Features: Yes. A declarative checkpoint restoring API.

Active Development: No. It has been dropped in favor of Pleiades.

6.2.2 Pleiades

Pleiades is Kairos' successor, which was also implemented as language extensions to an existing programming language. Whereas Kairos is built using Python extensions, Pleiades extends the C language with constructs to address the nodes in the network and their local state. The C program is compiled by Pleiades into node-level nesC programs that can be directly linked with standard TinyOS components and the Pleiades runtime system.

By default, a Pleiades program has a single sequential thread of control, which provides a simple semantics for programmers to understand and reason about. This means only one node in the system is executing any Pleiades instruction at any time. However, Pleiades includes a novel language construct for parallel iteration called cfor, which can be used, for example, to iterate concurrently over all network nodes or all one-hop neighbors of a particular node.

In the example below we can see these constructs used in practice to build a street parking application. The function *reserve* is used to reserve a parking spot near a certain destination by looping through neighboring nodes and looking for an open spot.

The *nodelocal* modifier (line 3) indicates that the variable is present in each node of the application, and can then later be accessed using the *@node* notation, as done in line 14. The *node* and *nodeset* types are defined to represent a node, and a collection of nodes.

Helper functions come together with these types, such as: *add_node*, *remove_node*, and *get_neighbors*. The *cfor* is like a normal for, except that the execution of the block is concurrent for the nodes in a nodeset, which is required in the example to guarantee that only one free node is reserved in an iteration. Access to *loose* variables, such as nToExamine and nExamined, are synchronized among all nodes inside the loop.

```
#include "pleiades.h"

boolean nodelocal isfree=TRUE;
nodeset nodelocal neighbors;
node nodelocal neighborIter;

void reserve(pos dst) {
  boolean reserved = FALSE;
  node nodeIter,reservedNode = NULL;
  node n = closest_node(dst);
  nodeset loose nToExamine = add_node(n, empty_nodeset());
  nodeset loose nExamined = empty_nodeset();

  if (isfree@n) {
    reserved = TRUE;
    reservedNode = n;
    isfree@n = FALSE;
    return;
  }

  while(!reserved && !empty(nToExamine)){
    cfor(nodeIter=get_first(nToExamine);nodeIter!=NULL;
        nodeIter = get_next(nToExamine)){
      neighbors@nodeIter=get_neighbors(nodeIter);
      for(neighborIter@nodeIter=get_first(neighbors@nodeIter);
          neighborIter@nodeIter!=NULL;
          neighborIter@nodeIter=get_next(neighbors@nodeIter)){
        if(!member(neighborIter@nodeIter,nExamined))
          add_node(neighborIter@nodeIter,nToExamine);
      }
      if(isfree@nodeIter){
        if(!reserved){
          reserved=TRUE; reservedNode=nodeIter;
          isfree@nodeIter=FALSE;
          break;
        }
      }
      remove_node(nodeIter,nToExamine);
      add_node(nodeIter,nExamined);
    }
  }
}
```

Listing 9: *Pleiades street-parking application*

Classification

Target Application Type: Sense-only.

Spatial Visibility: Global.

Language Paradigm: Imperative.

Fault-tolerance Features: No.

Active Development: No.

6.2.3 Regiment

Regiment [NMW07] is a system based on a functional programming language and designed for applications with spacial locality, such as object tracking or intrusion detection. Regiment exposes the sensor network to the developer as a set of data streams, called signals, which represent individual states of nodes (readings, or the result of a local computation), or an aggregated value obtained by processing multiple input signals. The programmer can manipulate sets of these signals, which may be defined by topological, functional, or geographical relationships between nodes.

The authors cite the benefits of a functional approach to sensor network programming as:

- Parallelism can be extracted in a straightforward manner;
- Computation can be migrated or replicated without effecting evaluation; and
- Functional programming hides state manipulations from the developer, which allows the compiler to decide how state should be stored.

Regiment is an unrestricted functional language, since it enables function composition in which functions are arguments. Functional programs written in Regiment may compute and derive events from abstractions of the sensor network's elements, space, and time. Each sensor is represented as a Node data object that exposes state, a collection of Nodes sharing a common characteristic is a Space, and a stream of Spaces is called Region. Regiment provides constructs to manipulate these data types, including aggregation, mapping, filtering, and conditional event handlers.

The Regiment system takes multiple compilation steps to generate the final node-level executable. The program is first translated into an intermediate language called Rquery, and, subsequently, the region streams are translated into local streams. The output of the compiler is an event-driven code written in an intermediate language called Token Machine Language [NTW05]. This language does not assume a threaded concurrency model, and is therefore suited for implementation on top of event-driven WSN operating systems, such as TinyOS. Regarding communication, nodes in a given region exchange data using a spanning tree of the network Graph that is created and maintained on every node by the Regiment runtime support.

Listing 10 shows a simple program that detects plumes by ensuring that the overall sum of readings around the phenomena exceeds a pre-specified threshold. The program first starts by defining: *abovethreshold*, a boolean function for filtering sensed data; *read*, a function to read values from the sensors; and *sum*, which uses *rfold* to aggregate values in a region *r*. In line 5, it reads values from all sensors, mapping the read function to the *world* region, comprised of all the nodes in the system. Then, it builds a region of detected nodes using the *rfilter* function, which is expanded into the hoods region in line 8, which is itself generated by taking the one-hop neighborhood of all nodes in the detected region. It next sums the values in the expanded detected region and sends a notification to the base station if any sum is exceeds the threshold.

```
fun abovethreshold(t) { t > CHEMICAL_THRESHOLD }
fun read(n) { sense("concentration", n) }
fun sum(r) { rfold( (+), 0, r) }

readings = rmap(read, world);
detects = rfilter(abovethreshold, readings);

hoods = rmap( fun(t, nd){ khood(1, nd) }, detects);
sums = rmap(sum, hoods);

base <- rfilter( fun(t){ t > CLUSTER_THRESHOLD }, sums);
```

Listing 10: *Plume detection using Regiment*

Classification

Target Application Type: Sense-only.

Spatial Visibility: Regional.

Language Paradigm: Functional.

Fault-tolerance Features: None.

Active Development: No.

6.2.4 SOSNA

SOSNA [KC08, Kar10], like Regiment and Proto, is a functional macroprogramming language focused on the stream programming paradigm for wireless sensor and actuator networks. It was developed by Marcin Karpiński during his doctoral studies at Trinity College Dublin.

The author claims that wireless sensor and actuators networks (WSANs) are not just an extension to WSNs with actuators, because the addition of actuators changes the focus from sensing to control, and thus poses new requirements. In this view, WSAN is a kind of control system in which sensing, actuation, and decision-making are distributed, whereas resource-constrained components communicate over unreliable channels. Also, because WSAN is a kind of control system, real-time functionality is operationally important and often requires actuator synchronization.

Based on these requirements, SOSNA is proposed with the following properties:

- Network topology, node heterogeneity and node mobility are abstracted while making it possible to realize different sensor-sensor, sensor-actuator, and actuator-actuator coordination strategies
- Program execution time is deterministic and proceeds in rounds. Together with the provision of time synchronization in the network, it imposes real-time bounds on actuator decision-making and enables actuator synchronization
- Distributed state can be maintained by accessing the previous round's stream values.
- Static program semantic enables the compiler to generate communication protocols that use radio duty-cycling[2] for energy conservation.

[2]A duty cycle is the percent of time that an entity spends in an active state as a fraction of the total time under consideration. For example, in an electrical device, a 60% duty cycle means the power is on 60% of the time and off 40% of the time

There are two kinds of spatial values used to form the streams SOSNA programs operate on: *fields* and *clusters*. Local values are data items stored in an individual node, and fields are collections of these local values of a given type, present at a subset of the nodes in a given execution round. Each node can contribute to at most one local value to a field, and fields do not have to be contiguous. Nodes that contribute a local value to a field are called *members* of that field. SOSNA field streams have the same semantics as regions in Regiment.

Clusters, on the other hand, are sparse fields, whose local values reside on network nodes that may be separated from each other by nodes that hold without values. Contrary to fields, separated nodes form an integral part of clusters and are called *cluster members*. Clusters are collections of local values that possess certain spatial aspects.

SOSNA programs are compiled into nesC code to be compiled and run by TinyOS.

Classification

Target Application Type: Sense-and-react.

Spatial Visibility: Global.

Language Paradigm: Functional.

Fault-tolerance Features: None.

Active Development: No.

6.2.5 COSMOS

COSMOS [AJG07] is a macroprogramming framework targeting heterogeneous sense-and-respond networks and based on stream processing. It consists of a programming language, called mPL, and the accompanying mOS portable operating system capable of running mPL programs on different hardware platforms. It offers a static dataflow programming model in which applications are composed as static graphs of stream processing components, called functional components.

Each functional component declaration specifies functional requirements, e.g. CPU speed or sensor type, for the node that will run a functional component instance. Additionally, the developer can specify constraints for specific instances. An instance of the functional component will be instantiated onto every node satisfying these conditions. These conditions, however, must be explicitly specified by the developer, and cannot be inferred using the overall non-functional application requirements.

The functional components in COSMOS are programmed in a subset of C, and the mPL language is used for defining components wiring and partitioning. The framework targets a tiered system architecture in which devices with different capabilities are organized in a hierarchical tree network topology. The language enables data to flow through the tree in all directions, making feedback signals possible, which are useful for control systems.

Classification

Target Application Type: Sense-only.

Spatial Visibility: Regional.

Language Paradigm: Declarative.

Fault-tolerance Features: Yes. Each component declaration specifies functional requirements (such as CPU speed, memory, etc.) for the node. The developer can also define constraints. These constraints, however, are solely based on node capabilities.

Active Development: No.

6.2.6 Chronus

Chronus [WBS10] is a macroprogramming language based on a new programming paradigm called spatiotemporal macroprogramming, which is designed to aid in WSN spatiotemporal event detection and data collection. Both the paradigm and the language were created by PhD. Hiroshi Wada et al. at the National ICT Australia (NICTA) laboratory.

The language treats space and time as first-class programming primitives, combined as a spacetime continuum. A spacetime is a three-dimensional object consisting of two spatial dimensions and a time dimension. This notion of spacetime provides an abstraction to seamlessly express event detection and data collection. Perhaps the most noticeable characteristic of using this abstraction is its ability to detect complex events consisting of multiple anomalies.

Although the language was designed to operate in a variety of dynamic spatiotemporal environments, it currently targets oil spillage detection and monitoring. Broken coastal oil station equipment and illegal dumping can cause oil spills, among other causes. Once spilled, oil can spread, change direction of movement, and split into multiple chunks. Some chunks may burn, and other may evaporate. These diverse characteristics make detecting oil spillage hard, as it involves complex events occurring over time.

The language is designed as an extension to Ruby, an object-oriented language that supports dynamic typing. Ruby accepts embedded domain-specific languages (DSLs), which extend Ruby's constructs. Chronus then reuses the language's syntax and semantics while introducing new keywords and primitives specific to spatiotemporal event detection and data gathering. It allows developers to define three types of complex events: sequence, any, and all. Each complex event is defined with a set of events. A sequence event fires when a set of events occur in chronological order. An any and all event fires when one or all of the defined events occur, respectively.

In addition to textual macroprogramming, Chronus also provides a visual environment. It leverages Google Maps[3] to show sensor node locations as icons, and allows application developers to graphically specify a space where they observe them.

[3] http://maps.google.com/

```
sp = Spacetime.new( GLOBALSPACE , Period.new( NOW , INF ) )
spaces = sp.get_spaces_every ( Min 10, Sec 30, 100 )

event spaces {
    sequence {
        not get_data('f-spectrum', MAX) > 290);
        get_data ('f-spectrum', MAX) > 290;
        get_data ('droplet - concentration ', MAX) > 10;
        within MIN 30;
    }
    any {
        get_data ('f-spectrum', MAX) > 320;
        window ( get_data('d- concentration ', MAX), AVG , HOUR -1) > 15;
    }
    all {
        get_data ('f-spectrum', AVG) > 300;
        get_data ('d- concentration ', AVG) > 20;
    }
}
execute{ | event_space , event_time |
    # query for the past
    sp1 = Spacetime.new( event_space , event_time , Min -30)
    past_spaces = sp1.get_spaces_every (Min 6, Sec 20, 50)
    num_of_nodes = past_spaces.get_nodes. select { |node|
        # @CWS_ROUTING
        node. get_data ('f-spectrum', Min 3) > 280}.size

    # query for the future
    s2 = Circle.new( event_area.centroid , event_area.radius * 2 )
    sp2 = Spacetime.new( s2 , event_time , Hr 1 )
    future_spaces = sp2.get_spaces_every ( Min 3, Sec 10, 80 )
    future_spaces.get_data ( 'f-spectrum', MAX , Min 1 ) {
        | data_type , value , space , time |
        # data handler }
    }
}
```

Listing 11: *Event-based data query using Chronus*

Classification

Target Application Type: Sense-only

Spatial Visibility: Global

Language Paradigm: Imperative, implemented as a DSL on top of Ruby.

Fault-tolerance Features: None.

Active Development: Yes.

6.2.7 Flask

Flask [MMW08] is a functional programming language targeting resource-constrained stream processing sensor applications. It was developed by Geoffrey Mainland during his work at Harvard Sensor Network Laboratory. It is embedded in Haskell, a general purpose functional language, and programs are translated into executable nesC code by a dedicated pre-processor.

The programming model is based on data-flow, and wiring operators and functions in an acyclic graph data specifies the flow. Each operator is a computational unit that takes multiple inputs and produces a single output, and the control flow moves operators in a depth-first manner. Different operators can be located on different nodes, and connected through a publisher-subscriber infrastructure.

Flask is designed such that higher-level abstractions can be built in terms of data-flow operators. As an example, the authors implement a version of TinyDB [MFHH05] in Flask.

Classification

Target Application Type: Sense-only.

Spatial Visibility: Global.

Language Paradigm: Functional, on top of Haskell.

Fault-tolerance Features: None.

Active Development: No.

6.2.8 Summarization

Framework	Target Application Type	Spatial Visibility	Language Paradigm	Fault-tolerance Features	Active Development
Kairos	Sense-only	Global	Imperative	Yes	No
Pleiades	Sense-only	Global	Imperative	No	No
Regiment	Sense-only	Regional	Functional	No	No
SOSNA	Sense-and-react	Global	Functional	No	No
COSMOS	Sense-only	Regional	Declarative	Yes	No
Chronus	Sense-only	Global	Imperative	No	Yes
Flask	Sense-only	Global	Functional	No	No
Srijan-FT	Sense-and-react	Regional	Declarative/ Imperative	Yes	Academic-only

Table 6.1: *Summary of Macroprogramming frameworks*

6.3 Discussion

In this chapter, we presented a heavily used wireless sensor network operating system and its accompanying programming language, as well as several macroprogramming languages and fault tolerance methods applied to this domain. These three areas form the basis of this work.

We described operating systems and their provided languages, as they are still the standard for WSN application development today. We showed how programming closely to the

operating systems can still be a hard task, requiring developers to think and write code in very low-level abstractions. It represents the complexity we aim to avoid.

A large number of people have studied macroprogramming languages for WSN application development. Here we describe seven we consider relevant. Although they vary in types of applications supported, language paradigm, and visibility, only two of the seven, Kairos and COSMOS, have some sort of fault tolerance mechanism. During our research, using sources such as Google Scholar, IEEE Xplore, and ACM DigitalLibrary, we did not find any other macroprogramming language that described explicit fault tolerance mechanisms.

The fact that almost no macroprogramming language explicitly deals with fault tolerance, and that developers have to implement it in low-level code, underscores the need for developing a macroprogramming tool with fault tolerant features, which is the goal of this work.

Chapter 7

Evaluation

Our goal is to investigate a way to make it easier for developers to integrate fault-tolerance into WSN applications. A qualitative approach is thus suitable for evaluating this work, yet we also conducted a complementary quantitative evaluation. The following presents both.

7.1 Quantitative Evaluation

With quantitative analysis, our purpose is to internally assess the proposed solution's efficiency. The tests below regard the framework's performance during generation time. What we aim to show is the framework's performance during development; in other words, the time it takes to generate an application with fault tolerant features.

The tests were run on a DELL XPS15-L502X notebook with a quad-core 2GHz Core i7-2630QM CPU and 8GB of RAM.

7.1.1 Code template generation time

We first aim to analyze whether adding fault tolerance features to an application built with Srijan increases the time it takes to generate it. In other words, we want to measure the impact in the code template generation phase according to how many fault tolerant tasks exist in the application.

The code template generation phase (Section 4.2) occurs when the initial graphical description of the application is translated to code templates that developers must fill in. In this analysis, we measured the time taken by the modified ATaG compiler to generate task templates according to the application's number of fault tolerant tasks. For each number of fault tolerant tasks, we took five readings and determined the median and standard deviation.

The assumption was that time would linearly scale, since each new FT task's generation requires roughly the same amount of code. We observed this behaviour in our tests, as shown in Figure 7.1. The time taken to generate the templates for each task is approximately 50 milliseconds.

7.1.2 Task mapping time

We also considered the time it took to generate all task mappings according to the number of fault tolerant tasks and nodes in the system. Task mapping is the process, determined by the completed templates and a network description, of generating the code that will be

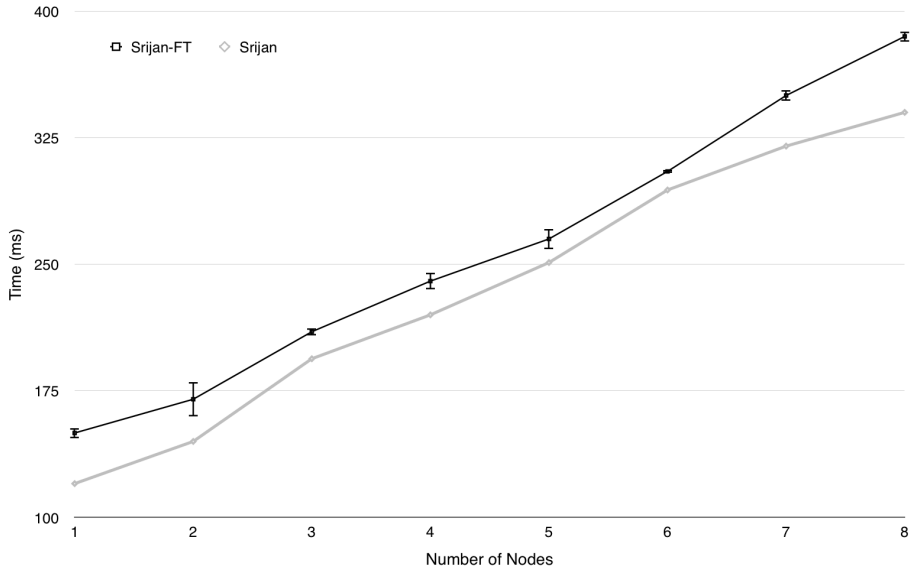

Figure 7.1: *Template Generation time x Number of FT tasks*

deployed to the nodes. Thus, we measured how long it took to generate the code for all the nodes, given a fixed number of fault tolerant tasks that needed to be deployed on each.

Figure 7.2 displays the time it took to generate code for applications containing one to three fault tolerant tasks per node (blue, red, and black lines). The time for the task mapping process also scales linearly, and the time taken to generate the code for a given node is approximately 500 milliseconds.

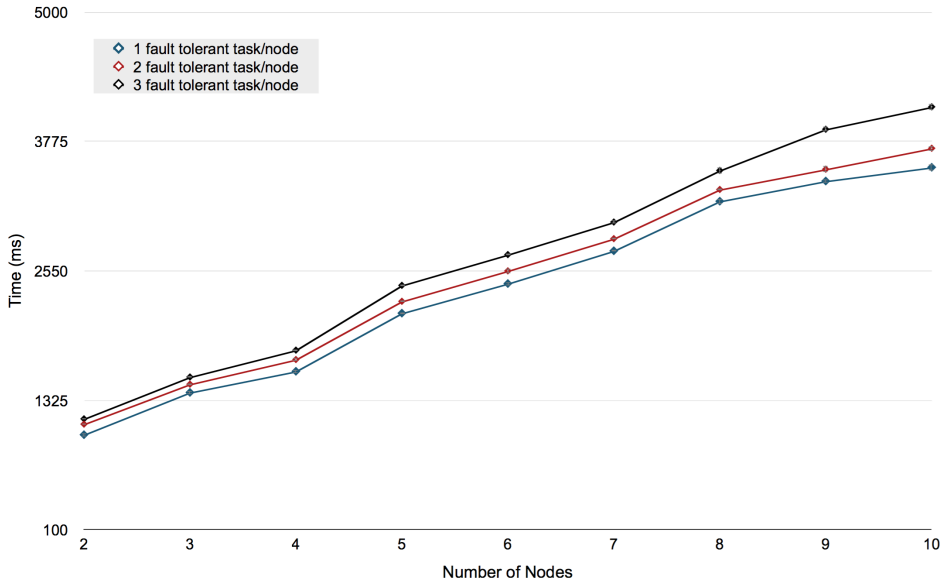

Figure 7.2: *Task Mapping time x Number of Motes x Number of FT tasks*

7.1.3 Quantitative analysis conclusion

The two analyses show that the time needed to generate application code scales linearly regarding the number of fault tolerant tasks and nodes in the network. The linearity was expected, since the number of generated files per node differs very little. Since writing files to the disk is what takes the longest time during the generation phase, it follows that the amount of time to generate all files for a node should be similar.

Given that both are linear, it could be argued that this approach is not suitable for larger deployments of thousands of nodes. However, one can only assess such a claim by conducting a proper study, with developers, which could compare the time and effort taken to write and deploy applications using our framework and an alternative platform, such as TinyOS or Contiki. Such assessment is left to future work.

7.2 Qualitative Assessments

The purpose of the qualitative assessments is to investigate whether the work fulfils the goal of providing easier fault tolerance features to WSN application development.

7.2.1 Fault tolerance validation

The first step in validating the proposed modifications is verifying whether they work as proposed, which means they generate fault tolerant mechanisms based on annotations in a high-level language.

For analysing the crash fault tolerance, we created an application composed of a producer, a consumer, and a display task. We generated the code and deployed it in two machines. Machine A had the producer and the display, while machine B had the consumer. The consumer gathered data from the producer and generated an output for the display. In this test, the consumer task was the one annotated to tolerate crash faults.

We manually "crashed" machine B, which was the master node with the consumer task, by disconnecting the network cable from the machine. Consequently, the system started a replica of the consumer task in machine A. As soon as we reconnected the previous master, it came online again and assumed the slave role. Finally, after we disconnected the current master, the previous machine became master again.

The output of this process can be observed in Figure 7.3.

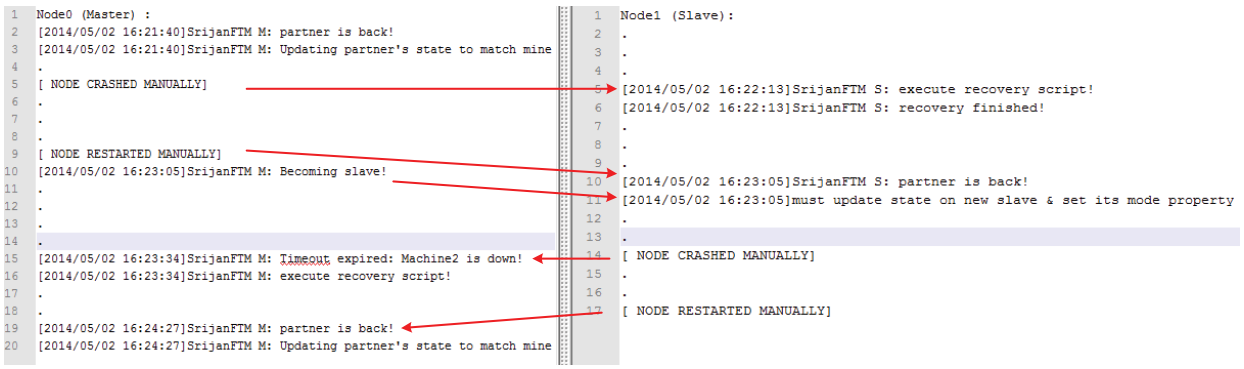

Figure 7.3: *Master/Slave replication tolerating a crash fault*

7.2.2 Comparison with a Similar Platform

During our investigation, we found only one platform similar to what we proposed: FlexFT [BUdAC13]. As the authors describe it, FlexFT is "a generic framework for constructing reliable systems that can deal with both hardware and software heterogeneity." It consists of a microkernel where fault tolerance policies are incremented as demanded, and provides a standardized and interoperable interface for sensor observations relying on the Sensor Web paradigm. These policies are deployed in the form of component plugins, which are destroyed when no longer required and can be reconfigured at runtime.

The component plugins are implemented using the OpenCOMJ api and the OpenCOM implementation in Java. The fault tolerance components' rationale is redundancy, using design diversity both for error detection and error recovery. This means the framework provides base interfaces for creating recovery blocks and n-version components.

Although the framework could be used with other sensors, the authors provide all the evaluations using SunSPOT devices, as we do with Srijan.

In the following subsections, we compare important characteristics of Srijan and FlexFT.

Framework type

We regard FlexFT as a middleware for building component-based fault-tolerant applications that use sensor platforms. As a macroprogramming framework, Srijan, on the other hand, takes a high-level application design and generates applications for different platforms.

Whereas FlexFT provides a microkernel to load components, Srijan relies on a pre-defined application architecture, which is implemented using the underlying platform and leaves only key implementation logic for the developer to fill in.

Programming paradigm

FlexFT relies on a component-based paradigm in which each component implements a common interface. The internal implementation is purely object-oriented, with no abstractions regarding data and the interaction model.

Srijan uses a data-driven paradigm, wherein the application is broken into processing units, called tasks, and data units. This paradigm simplifies the interface and implementation aspects. Once a task's code is generated, the internal implementation only needs to know what data type it expects and what data type to produce.

Fault tolerance aspects

Both our work and FlexFT are based on the premise that wireless sensor applications have to deal with faults, and that platforms that make it easy to integrate fault tolerance features are needed. Both platforms aim to provide ways for developers to better integrate fault tolerance into the development process.

Our approaches differ in implementation. FlexFT focuses on design diversity and provides very generic components for a developer to implement different algorithm variants. Our approach focuses on ready-to-use pluggable fault tolerance mechanisms that can be added to the application via simple declarative properties requiring minimal configuration.

FlexFT vs Srijan-FT

We believe that Srijan-FT presents a higher level of abstraction than FlexFT, which makes the two solutions complementary. Srijan's data-driven paradigm could be implemented

using FlexFT; therefore, Srijan could be used to design an application at a higher level of abstraction, and then generate FlexFT code with tasks implementing the component framework's interfaces.

Our views on fault tolerance are also similar. The N-version technique, for example, could be added in the declarative Srijan language, thereby offering a simple way to tell the compiler to generate "n" versions of a task.

The differing level of abstraction is exemplified by the LightController application, shown in Chapter 4. In this application, there are three tasks: a light sampling, an aggregator that calculates the medium light, and a controller or display that consumes the medium light produced by the aggregator.

In Srijan, the developer would have to specify the application in high-level, as is shown in Figure 7.4, then generate the templates and fill them with the implementation. All data transition is handled by the framework runtime. The full templates are shown in Appendix A, and we display the collector template in Listing 12.

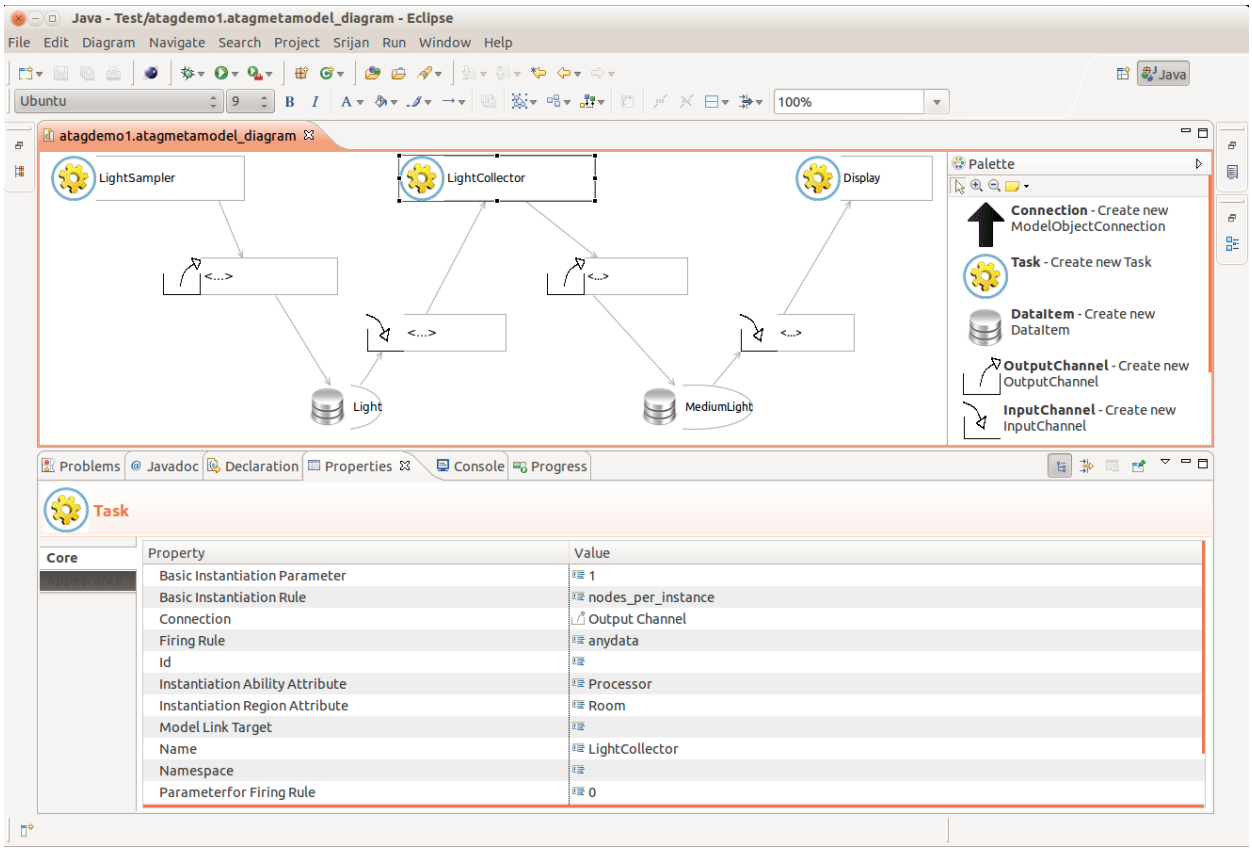

Figure 7.4: *LightCollector application in Srijan (high-level)*

```
package atag.apps.mainApp.tasks;

import atag.runtime.DataPool;
import atag.runtime.NodeInfo;
import atag.apps.mainApp.gen.*;

public class LightCollector extends AbstractLightCollector{

   public LightCollector(DataPool dp, NodeInfo myconfig) {
       super(dp,myconfig);
   }

   /* This method is called when Light is produced */
   protected void handleLightProduced(Light r_Light) {
       debugPrint("[LightCollector]Got a Data Item: Light");
       //TODO Add code to respond to production of data item Light
   }

   /*
       // Sample code for producing MediumLight

       debugPrint("[LightCollector]Producing a Data Item: MediumLight");
       MediumLight m_MediumLight= new MediumLight();
       //TODO Fill in the parameters of MediumLight
       //Use your IDE's autocomplete features or browse the API of the Autogenerated code
       this.produceMediumLight(m_MediumLight);
   */

}
```

Listing 12: *LightCollector template*

With FlexFT, the runtime system does not assume a data-driven application. Thus, the developer has to program all data flow within the application. This could be accomplished with a main loop that gathers data from sensors and sends it to the aggregator. Listing 13 displays a pseudo-code exemplifying how this could be implemented in FlexFT.

The pseudo-code describes the flow code needed for data wiring . The developer also must implement the aggregator and display. Overall, FlexFT provides the necessary components to produce fault tolerance via redundancy in the application, but the developer still has to link all the components together via low-level programming.

```
public LightCollectorApp {

    public run() {
        List<ISensor> lSensors = buildSensorList();

        while(true) {
            for(ISensor s : lSensors) {
                x = s.readMetric();
                this.mediumValue = aggregate(x);
            }

            updateDisplay(this.mediumValue);
            sleep(FIVE_SECONDS);
            exitIfNecessary();
        }
    }

    SensorListObject buildSensorList() {
        // load openCOM runtime
        // get interfaces of the Sensors
        // wire classNames with Sensors
        // wire referencesList to impl object
        return SensorListObject representing all Sensors
    }

    MediumLight aggregate(Value x) {
        // N variant implementations of the aggregator implement FlexFT NVariant interface
        // Each variant is executed, and the agreeing value is returned
        MediumLight value = executeAggregatorVariant(x);
    }


    void updateDisplay(MediumLight value) {
        // Executes display
        Display.display(value);
    }

}
```

Listing 13: *FlexFT LightCollector pseudo-code*

7.3 Discussion

In this chapter, we presented an analysis of our work. Although very preliminary, we believe this analysis provides evidence that our approach works and can be used as intended.

Our quantitative analysis indicates the algorithms' linearity in regard to the number of fault tolerance mechanisms, which should not impact small to medium sized applications.

We also provided a comparison to the most similar work we found, FlexFT. It is our view that they focus on two different aspects of fault tolerance for Wireless Sensor Networks. As such, Srijan could be extended to support code generation for the FlexFT framework.

Chapter 8

Conclusion

In this work, we implemented fault tolerance features in a macroprogramming framework for Wireless Sensor Network by integrating an adaptive fault tolerance protocol. We also presented the updated workflow for writing WSN applications, including the modified runtime and compilation process.

Through the analysis of related work, we showed that although failure detection and recovery in the area has widely been studied, proper support for fault tolerance was lacking in current high-level abstraction tools. This lack of fault tolerance support makes it harder to program applications for WSN. Although fault tolerance can be achieved by implementing detection and recovery algorithms at a low level for each application, it makes the process of creating applications harder and more prone to error.

Although in a small setup and with limited functionality, we demonstrated in this dissertation that fault tolerant features can be added to macroprogramming frameworks and impact the application development cycle for better, diminishing the amount of code needed for achieving fault tolerant tasks in a data-driven application.

This outcome is significant as a first effort in bringing fault tolerance to macroprogramming, and for its extensibility, which makes it possible to evolve the platform and use it as a basis for other work.

8.1 Future Work

We believe this is a promising beginning for fault tolerance becoming a first-class concept in macroprogramming tools for Wireless Sensor Networks. A great deal of future work can be done, including:

Extending the platform's Code Generation tool support to other more wildly adopted WSN platforms
We used the ATaG language in Srijan to model data-driven applications at a high level, using a GUI or XML. Currently, Srijan only generates code for the SunSPOT platform, but this application description model can be used to generate applications for other platforms, such as TinyOS and Contiki, that target a different range of devices.

Assessment of the platform using real developers
As said in Chapter 7, to better analyze the impact of this work, we need to conduct a study with developers, comparing the time and effort taken to write and deploy applications using our framework and an alternative platform, such as TinyOS or Contiki.

Modeling and integrating other fault tolerance techniques to the protocol
As a proof of concept, for this thesis we implemented only a small handful of detection and recovery techniques. An important evolution of the work will be to implement and evaluate more techniques (Chapter 3) following the high level protocol.

Using a lightweight component-based middleware for the fault tolerance protocol
The current version uses the FraSCAti framework in the FT protocol, which, although a comprehensible service component architecture framework, is not suitable for more lightweight and resource-less platforms. Therefore, it prevents the protocol from running in every node in a decentralized fashion.

Optimizing the task assignment process
The task assignment process could take into account properties of the node, such as location, to analyze the cost of performing a master-slave synchronization among them and choosing the best pair for each task/section.

Changing to a DSL
Recent work has been conducted on altering the Srijan platform to provide a DSL for writing Wireless Sensor Network applications. The fault tolerance aspects presented in this work could be added to the DSL. This could also lead to changes in the language itself, or its extensibility.

8.2 Publications

During this project we submitted a paper to two Wireless Sensor Networks conferences: the International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP) and IEEE International Conference on Mobile Ad hoc and Sensor Systems.

Both times our paper was rejected because reviewers focused on the fault tolerance mechanisms and not on the macroprogramming extension for supporting these mechanisms.

We are currently working on a different version of the paper to deal with this issue.

8.2.1 Other publications

While working at this project, the author also collaborated with other students at IME-USP, which culminated in the following publication:

Leonardo A. Leite, Gustavo Ansaldi Oliva, Guilherme M. Nogueira, Marco Aurélio Gerosa, Fabio Kon, and Dejan S. Milojicic. 2013. *"A systematic literature review of service choreography adaptation"*. Service Oriented Computing and Applications. 7, 3 (September 2013), 199-216.

Appendix A

Srijan Code Generation Example

Constants

The constants file is generated only as a helper for naming tasks across the application.

```
1  package atag.apps.mainApp.gen;
2
3  public class IDConstants {
4
5      public static final int T_LIGHTSAMPLER = 0;
6      public static final int T_LIGHTCOLLECTOR = 1;
7      public static final int T_DISPLAY = 2;
8
9
10     public static final int D_LIGHT = 0;
11     public static final int D_MEDIUMLIGHT = 1;
12 }
```

Listing 14: *Generated IDConstants*

DataItems

Data items are generate as Java Beans with *getters* and *setters* for all fields. We show here the `Light` data item generated for the light controller application. DataItems also have serialization and deserialization methods that write and read from the communication channel.

```java
package atag.apps.mainApp.gen;

import atag.j2mespecific.*;
import java.io.IOException;
import java.io.DataOutput;
import java.io.DataInput;

public class Light implements Serializable {

    private double val;
    private double max;
    private String unit;

    public Light(){
       //NOTE needs a default constructor for deserialization
    }

    public double getVal() {
        return val;
    }
    public void setVal(double val) {
        this.val = val;
    }

    public double getMax() {
        return max;
    }
    public void setMax(double max) {
        this.max = max;
    }

    public String getUnit() {
        return unit;
    }
    public void setUnit(String unit) {
        this.unit = unit;
    }

    public void serialize(DataOutput dg) throws IOException {
        dg.writeDouble(val);
        dg.writeDouble(max);
        dg.writeUTF(unit);
    }

    public void deserialize(DataInput dg) throws IOException,
        ClassNotFoundException, IllegalAccessException, InstantiationException {
        val = dg.readDouble();
        max = dg.readDouble();
        unit = dg.readUTF();
    }

}
```

Listing 15: *Generated Light data item*

Tasks

```
package atag.apps.mainApp.tasks;

import atag.runtime.DataPool;
import atag.runtime.NodeInfo;
import atag.apps.mainApp.gen.*;

public class LightSampler extends AbstractLightSampler{

   public LightSampler(DataPool dp, NodeInfo myconfig) {
       super(dp,myconfig);
   }

   /* This method is called periodically */
   protected void handleExpiryOfTimer() {
       debugPrint("[LightSampler]Periodic timer expired");
       //TODO Add code to perform the desired periodic action
   }

   /*
       // Sample code for producing Light

       debugPrint("[LightSampler]Producing a Data Item: Light");
       Light m_Light= new Light();
       //TODO Fill in the parameters of Light
       //Use your IDE's autocomplete features or browse the API of the Autogenerated code
       this.produceLight(m_Light);
   */

}
```

Listing 16: *LightSampler template*

```
package atag.apps.mainApp.tasks;

import atag.runtime.DataPool;
import atag.runtime.NodeInfo;
import atag.apps.mainApp.gen.*;

public class LightCollector extends AbstractLightCollector{

   public LightCollector(DataPool dp, NodeInfo myconfig) {
       super(dp,myconfig);
   }

   /* This method is called when Light is produced */
   protected void handleLightProduced(Light r_Light) {
       debugPrint("[LightCollector]Got a Data Item: Light");
       //TODO Add code to respond to production of data item Light
   }

   /*
       // Sample code for producing MediumLight

       debugPrint("[LightCollector]Producing a Data Item: MediumLight");
       MediumLight m_MediumLight= new MediumLight();
       //TODO Fill in the parameters of MediumLight
       //Use your IDE's autocomplete features or browse the API of the Autogenerated code
       this.produceMediumLight(m_MediumLight);
   */

}
```

Listing 17: *LightCollector template*

```
package atag.apps.mainApp.tasks;

import atag.runtime.DataPool;
import atag.runtime.NodeInfo;
import atag.apps.mainApp.gen.*;

public class LightController extends AbstractLightController{


   public LightController(DataPool dp, NodeInfo myconfig) {
       super(dp,myconfig);
   }

   /* This method is called when MediumLight is produced */
   protected void handleMediumLightProduced(MediumLight r_MediumLight) {
       debugPrint("[Display]Got a Data Item: MediumLight");
       //TODO Add code to respond to production of data item MediumLight
   }

}
```

Listing 18: *LightController template*

Abstract Tasks

Abstract Tasks are responsible for the boilerplate code regarding each task. It provides the methods for receiving data items meant for the task, and for sending data items produced by the task to the data pool. They extend ATaGAppTask, which implements Runnable and provides methods for retrieving the region the task is operating on.

```
package atag.apps.mainApp.gen;

import atag.common.ATaGAppTask;
import atag.runtime.DataItem;
import atag.runtime.DataPool;
import atag.runtime.NodeInfo;

public abstract class AbstractLightSampler extends ATaGAppTask {

  private DataPool m_dataPool;

  private int _requesting_nodeID;

  public AbstractLightSampler(DataPool dp, NodeInfo myconfig) {
    super(myconfig);
    this.m_dataPool=dp;
  }

  public synchronized void run() {
    try {
      while (true) {
        Thread.sleep(1000);
        handleExpiryOfTimer();
      }
    } catch (InterruptedException e) {
      return;
    }
  }

  /*
   * This method is called periodically
   */
  protected abstract void handleExpiryOfTimer() ;

  /**
    * This method produces Light
    */
  public final void produceLight(Light r_Light) {
    DataItem m_dataitem = new DataItem(IDConstants.D_LIGHT,
      IDConstants.T_LIGHTSAMPLER, r_Light);
    m_dataPool.putData(m_dataitem);
  }

}
```

Listing 19: *Generated AbstractLightSampler*

```
package atag.apps.mainApp.gen;

import atag.common.ATaGAppTask;
import atag.runtime.DataItem;
import atag.runtime.DataPool;
import atag.runtime.NodeInfo;

public abstract class AbstractLightCollector extends ATaGAppTask {

  private DataPool m_dataPool;

  private int _requesting_nodeID;

  public AbstractLightCollector(DataPool dp, NodeInfo myconfig) {
     super(myconfig);
     this.m_dataPool=dp;
  }

  public void run() {
     DataItem t_dataItem = m_dataPool.getData(IDConstants.T_LIGHTCOLLECTOR,IDConstants.D_LIGHT);
     if(t_dataItem != null) {
        Light recvdLight = (Light) t_dataItem.core();
        this.handleLightProduced(recvdLight);
     }
  }

  /*
    * This method is called when Light is produced
    */
  protected abstract void handleLightProduced(Light r_Light);

  /**
    * This method produces MediumLight
    */
  public final void produceMediumLight(MediumLight r_MediumLight) {
     DataItem m_dataitem = new DataItem(IDConstants.D_MEDIUMLIGHT,
        IDConstants.T_LIGHTCOLLECTOR, r_MediumLight);
     m_dataPool.putData(m_dataitem);
  }

}
```

Listing 20: *Generated AbstractLightCollector*

```
package atag.apps.mainApp.gen;

import atag.common.ATaGAppTask;
import atag.runtime.DataItem;
import atag.runtime.DataPool;
import atag.runtime.NodeInfo;

public abstract class AbstractLightController extends ATaGAppTask {

  private DataPool m_dataPool;

  private int _requesting_nodeID;

  public AbstractLightController(DataPool dp, NodeInfo myconfig) {
    super(myconfig);
    this.m_dataPool=dp;
  }

  public void run() {
    DataItem t_dataItem = m_dataPool.getData(IDConstants.T_DISPLAY,IDConstants.D_MEDIUMLIGHT);
    if(t_dataItem != null) {
      MediumLight recvdMediumLight = (MediumLight) t_dataItem.core();
      this.handleMediumLightProduced(recvdMediumLight);
    }
  }

  /**
  * This method is called when MediumLight is produced
  */
  protected abstract void handleMediumLightProduced(MediumLight r_MediumLight);

}
```

Listing 21: *Generated AbstractLightController*

Manager

Currently, the generated ATagManager only sets up the neighborhoods for each node, and populates the task firing table. It extends the PreBuiltAtagManager, which starts the necessary tasks based on the firing table and verifies all data items generated by tasks running at the node.

```
package atag.apps.mainApp.gen;

import atag.common.*;
import atag.runtime.*;
import atag.runtime.ln.*;
import atag.runtime.ln.neighborhoodDefs.*;
import atag.apps.mainApp.tasks.*;

public class AtagManager extends PrebuiltAtagManager {

    public AtagManager() { }

    public void setUp() {
        taskDecls.addElement(
            new RunnableTask((ATaGTaskDeclaration) this.m_program.getTaskList().elementAt(0),
                new LightSampler(this.m_dataPool, this.m_config),
                Thread.MAX_PRIORITY - 0));
        taskDecls.addElement(
            new RunnableTask((ATaGTaskDeclaration) this.m_program.getTaskList().elementAt(1),
                new LightCollector(this.m_dataPool, this.m_config),
                Thread.MAX_PRIORITY - 0));
        taskDecls.addElement(
            new RunnableTask((ATaGTaskDeclaration) this.m_program.getTaskList().elementAt(2),
                new Display(this.m_dataPool, this.m_config),
                Thread.MAX_PRIORITY - 0));
    }

    public Neighborhood[] getLNScopeForData(int taskID, int dataID) {
        switch (dataID) {
            case IDConstants.D_LIGHT:{
                Predicate[] tempPred0 = new Predicate[3];
                int count0 = 0;

                tempPred0[count0++] = new StringSetMembershipPredicate(
                        String.valueOf(IDConstants.T_LIGHTCOLLECTOR),
                        StringSetMembershipPredicate.IS_IN,
                        NodeInfo.ASSIGNED_TASK_ATTR_NAME);

                tempPred0[count0++] = new IntegerRangePredicate("Room",
                        ((Integer) m_config.getAttributeByName("Room")).intValue() - 0,
                        ((Integer) m_config.getAttributeByName("Room")).intValue() + 0);
                Predicate[] finalPred0 = new Predicate[count0];
                for(int i = 0; i < count0; i++){
                    finalPred0[i] = tempPred0[i];
                }
                Neighborhood scopeOfChannel0 = new ConjunctiveNeighborhood(finalPred0);
                return new Neighborhood[] { scopeOfChannel0    };
            }
            case IDConstants.D_MEDIUMLIGHT:{
                Predicate[] tempPred0 = new Predicate[3];
                int count0 = 0;

                tempPred0[count0++] = new StringSetMembershipPredicate(
                        String.valueOf(IDConstants.T_DISPLAY),
                        StringSetMembershipPredicate.IS_IN,
                        NodeInfo.ASSIGNED_TASK_ATTR_NAME);

                tempPred0[count0++] = new IntegerRangePredicate("Room",
                        ((Integer) m_config.getAttributeByName("Room")).intValue() - 0,
                        ((Integer) m_config.getAttributeByName("Room")).intValue() + 0);
                Predicate[] finalPred0 = new Predicate[count0];
                for(int i = 0; i < count0; i++){
                    finalPred0[i] = tempPred0[i];
                }
                Neighborhood scopeOfChannel0 = new ConjunctiveNeighborhood(finalPred0);
                return new Neighborhood[] { scopeOfChannel0    };
            }
            default:
                return new Neighborhood[] {};
        }/*end switch*/
    }

}
```

Listing 22: *Generated AtagManager*

Appendix B

Prototype code

Listing 23 shows the methods in ATagManager used to instantiate the fault tolerance protocol. We use the FraSCAti SCA framework [SMR+12] to load the composite files having the proper wirings for each method, in lines 11 to 23. Depending on which node type is loading the protocol, different actions are taken: if it is a master node, the one that is active, the protocol is returned. If it is a slave node, one that is inactive in the beginning of the execution, we configure the ATagManager as a task starter service in the fault tolerance protocol. If the master fails, the protocol will call a method asking the manager to start the task in the slave replica. In the *setFTTask* method, we set a reference to the task in a controller, if the node is a slave replica.

```
    private FraSCAti frascati;
    private Component composite;

    private FTMProtocolAPI getFTMProtocol(Integer ftNodeType) {
        if (ftNodeType == 0) return null;

        FTMProtocolAPI ft = null;
        TriggerTaskStarter ts = null;

        try {
            System.out.println("------------------ >>> Loading FraSCAti");
            frascati = org.ow2.frascati.FraSCAti.newFraSCAti();

            System.out.println("------------------ >>> Getting composite");
            composite = frascati.getComposite("srijan-node.composite");

            if (ftNodeType == 1) { // master
                System.out.println("------------------ >>> Getting FT Protocol");
                ft = frascati.getService(composite, "r", FTMProtocolAPI.class);
            } else if (ftNodeType == 2){ // slave
                System.out.println("------------------ >>> Getting TaskStarterService");
                ts = frascati.getService(composite, "taskStarter", TriggerTaskStarter.class);
                ts.setTaskStarter(this);
            }
        } catch (FrascatiException e) {
            e.printStackTrace();
        }
        return ft;
    }

    /**
     * Sets the fault tolerant task reference on the Task Controller
     * component of the fault tolerance protocol.
     *
     * @param ftNodeType the type of node, i.e. master, slave, etc
     * @param ft
     */
    private void setFTTask(Integer ftNodeType, FTTask ft){
        if (ftNodeType != 2) return;
        try {
            FTTaskController ftc;
            ftc = frascati.getService(composite, "taskController", FTTaskController.class);
            ftc.setFTTask(ft);
        } catch (FrascatiException e) {
            e.printStackTrace();
        }
    }

    /**
     * Creates the necessary fault tolerance protocol for a task that
     * is marked to be fault tolerant.
     *
     * @return the fault tolerant task
     */
    private Runnable getFTTask(){ we
        // get node type parameter from NodeInfo
        Integer ftNodeType = (Integer) m_config.getAttributeByName("ft-node-type");

        // load fault tolerance protocol
        FTMProtocolAPI ftProtocol = getFTMProtocol(ftNodeType);

        // create task
        LightCollector task = new LightCollector(this.m_dataPool, this.m_config, ftProtocol);

        // set task reference on task controller
        setFTTask(ftNodeType, task);

        return task;
    }
```

Listing 23: *Loading fault tolerance protocol in ATagManager*

FTTaskController is an interface which the SlaveServer (Listing 24) implements. The

server receives inputs from the corresponding methods in the master protocol, and dispatches it to the fault tolerant task to process. This all comes together with the abstract task class implementing the FTTask interface and providing methods to handle these generic calls. SyncAfter is used to deserialize internal state into annotated fields of the concrete class (Listing 25), while SyncBefore is used to forward inputs to the task without generating output (Listing 26).

```
@Scope("COMPOSITE")
@Service(interfaces={})
public class SlaveServer implements
    SyncBeforeService,
    SyncAfterService,
    FTTaskController
{
    private FTTask ftTask;

    public void setFTTask(FTTask ftTask) {
        this.ftTask = ftTask;
    }

    @Init
    public void init() {
        System.out.println("SLAVE SERVER Init");
    }

    public void executeAfter(byte[] syncMessage) {
        System.out.println("---->> receiving SyncAfter");
        this.ftTask.handleSyncAfter(syncMessage);
    }


    public void executeBefore(byte[] syncMessage) {
        System.out.println("--->> receiving SyncBefore");
        this.ftTask.handleSyncBefore(syncMessage);
    }

}
```

Listing 24: *Slave server composite implementation*

```
public void handleSyncAfter(byte[] input) {
    setState(input);
}

public void setState(byte[] state) {
    try {
        ByteArrayInputStream bin = new ByteArrayInputStream(state);
        while(bin.available() != 0) {
            Object[] objArray = (Object[]) SerializationUtils.deserialize(bin);
            if (objArray != null) {
                String fieldName = (String) objArray[0];
                Field f = this.getClass().getDeclaredField(fieldName);
                System.out.println("--> Deserializing: " + fieldName);
                f.setAccessible(true);
                f.set(this, objArray[1]);
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Listing 25: *Deserializing internal state when receiving SyncAfter*

```
public void handleSyncBefore(byte[] input) {
    DataItem dataItem = new DataItem();
    ByteArrayInputStream bin= new ByteArrayInputStream(input);
    DataInputStream din = new DataInputStream(bin);
    try {
        dataItem.deserialize(din);
        receiveData(dataItem);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

public void receiveData(DataItem t_dataItem){
    if (t_dataItem != null) {
        if (ftProtocol != null) {
            ftProtocol.executeBefore(SerializationUtils.dataItemToBytes(t_dataItem));
        }

        Light recvdLight = (Light) t_dataItem.core();
        this.handleLightProduced(recvdLight);
    }
}
```

Listing 26: *Handling forwarded inputs on SyncBefore*

On the master replica, the FT protocol is called with the result of state serializaton. Serialization takes place by looping through all fields of the concrete class and checking for the *@FTState* annotation. The state is then sent to the slave via the syncAfter call, as shown in Listing 27.

```
    public final void produceMediumLight(MediumLight r_MediumLight) {
        DataItem m_dataitem = new DataItem(IDConstants.D_MEDIUMLIGHT,
            IDConstants.T_LIGHTCOLLECTOR, r_MediumLight);
        m_dataPool.putData(m_dataitem);

        if (ftProtocol != null){
            ftProtocol.executeAfter(getState());
        }
    }

    public byte[] getState(){
        ByteArrayOutputStream result = new ByteArrayOutputStream();

        try {
            for(Field f: this.getClass().getDeclaredFields()){
                FTMStateVar annotation = f.getAnnotation(FTMStateVar.class);
                if (annotation != null){
                    System.out.println("--> Serializing: " + f.getName());
                    f.setAccessible(true);
                    result.write(SerializationUtils.serialize(
                            new Object[]{f.getName(), f.get(this)})
                        );
                    f.setAccessible(false);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        return result.toByteArray();
    }
```

Listing 27: *Master node call to syncAfter*

Bibliography

[ABC+04] Tarek F. Abdelzaher, Brian M. Blum, Qing Cao, David Evans, Jemin George, Selvin George, Tian He, Liqian Luo, Sang H. Son, Radu Stoleru, John Stankovic, and Anthony D. Wood. Envirotrack: towards an environmental computing paradigm for distributed sensor networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 582 – 589, 2004. 28

[AJG07] Asad Awan, Suresh Jagannathan, and Ananth Grama. Macroprogramming heterogeneous sensor networks using COSMOS. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 159–172, New York, NY, USA, 2007. ACM. 2, 36

[AKK04] Jamal N. Al-Karaki and Ahmed E. Kamal. Routing techniques in wireless sensor networks: a survey. *IEEE Wireless Communications*, 11(6):6–28, 2004. 10, 12

[Ame11] Wouter Amerijckx. Language support for programming interactions among wireless sensor network nodes. Master's thesis, Vrije Universiteit Brussel, Faculty of Science and Bio-Engineering Sciences, Department of Computer Science, June 2011. 13

[ASSC02] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–114, aug 2002. 1

[AY05] Kemal Akkaya and Mohamed Younis. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks*, 3(3):325 – 349, 2005. 10

[BBD+02] Rimon Barr, John C. Bicket, Daniel S. Dantas, Bowei Du, T. W. Danny Kim, Bing Zhou, and Emin Gün Sirer. On the need for system-level support for ad hoc and sensor networks. *SIGOPS Operating System Review*, 36(2):1–5, April 2002. 27

[BDU+12] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. Organizing the aggregate: Languages for spatial computing. *CoRR*, abs/1202.5509, 2012. 13

[BHG+13] Emmanuel Baccelli, Oliver Hahm, Mesut Günes, Matthias Wählisch, and Thomas Schmidt. RIOT OS: Towards an OS for the Internet of Things. In

The 32nd IEEE International Conference on Computer Communications (INFOCOM), Turin, Italy, April 2013. 7

[Bis08] Urs Bischoff. *Engineering Distributed Sensor-Actuator Network Applications*. PhD thesis, Lancaster University, Computing Department, October 2008. 13

[BK07] Urs Bischoff and Gerd Kortuem. A compiler for the smart space. In Bernt Schiele, AnindK. Dey, Hans Gellersen, Boris Ruyter, Manfred Tscheligi, Reiner Wichert, Emile Aarts, and Alejandro Buchmann, editors, *Ambient Intelligence*, volume 4794 of *Lecture Notes in Computer Science*, pages 230–247. Springer Berlin Heidelberg, 2007. 13

[BPRL05] Amol Bakshi, Viktor K. Prasanna, Jim Reich, and Daniel Larner. The abstract task graph: a methodology for architecture-independent programming of networked sensor systems. In *Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services*, EESR '05, pages 19–24, Berkeley, CA, USA, 2005. USENIX Association. 13, 14

[BUdAC13] Delano Medeiros Beder, Jó Ueyama, João Porto de Albuquerque, and Marcos Lordello Chaim. FlexFT: A generic framework for developing fault-tolerant applications in the sensor web. *International Journal of Distributed Sensor Networks*, 2013, 2013. 28, 44

[CB11] Alexandra Caracas and Alexander Bernauer. Compiling business process models for sensor networks. In *Proceedings of the 7th International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS)*, pages 1–8, June 2011. 13

[CCG+07] Paolo Costa, Geoff Coulson, Richard Gold, Manish Lad, Cecilia Mascolo, Luca Mottola, Gian Pietro Picco, Thirunavukkarasu Sivaharan, Nirmal Weerasinghe, and Stefanos Zachariadis. The runes middleware for networked embedded systems and its application in a disaster management scenario. In *Proceedings of the 5th Annual IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 69–78, 2007. 28

[CKS06] Jinran Chen, Shubha Kher, and Arun Somani. Distributed fault detection of wireless sensor networks. In *Proceedings of the 2006 workshop on Dependability issues in wireless ad hoc networks and sensor networks*, DIWANS '06, pages 65–72, New York, NY, USA, 2006. ACM. 11

[CTB09] Marcus Chang, Andreas Terzis, and Philippe Bonnet. Mote-based online anomaly detection using echo state networks. In Bhaskar Krishnamachari, Subhash Suri, Wendi Heinzelman, and Urbashi Mitra, editors, *Distributed Computing in Sensor Systems*, volume 5516 of *Lecture Notes in Computer Science*, pages 72–86. Springer Berlin Heidelberg, 2009. 11

[dBSdR+07] Talles M. G. de Barbosa, Iwens G. Sene, Adson F. da Rocha, Francisco A. de O. Nascimento, Joao L. A. Carvalho, and Hervaldo S. Carvalho. A new model for programming software in body sensor networks. In *Proceedings of the 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 1515 –1518, August 2007. 13

[DGV04] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Tampa, Florida, USA, November 2004. 27

[dSVB07] Luciana Moreira Sá de Souza, Harald Vogt, and Michael Beigl. A survey on fault tolerance in wireless sensor networks. Technical report, Fakultät für Informatic, Karlsruhe Institut für Technologie, 2007. 1, 9, 10, 12

[FRL05] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Mobile agent middleware for sensor networks: an application case study. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press. 28

[GGAH14] Manish Gupta, Jing Gao, Charu C. Aggarwal, and Jiawei Han. Outlier detection for temporal data: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 26(9):2250–2267, 2014. 21

[GGG05] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macroprogramming wireless sensor networks using kairos. In Viktor K. Prasanna, Sitharama S. Iyengar, Paul G. Spirakis, and Matt Welsh, editors, *Distributed Computing in Sensor Systems*, volume 3560 of *Lecture Notes in Computer Science*, pages 126–140. Springer Berlin Heidelberg, 2005. 2, 7, 30, 31

[GGSE01] Deepak Ganesan, Ramesh Govindan, Scott Shenker, and Deborah Estrin. Highly-resilient, energy-efficient multipath routing in wireless sensor networks. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(4):11–25, October 2001. 12

[GKMG07] Ramakrishna Gummadi, Nupur Kothari, Todd Millstein, and Ramesh Govindan. Declarative failure recovery for sensor networks. In *Proceedings of the 6th international conference on Aspect-oriented software development*, AOSD '07, pages 173–184. ACM, 2007. 9

[GY03] Gaurav Gupta and Mohamed Younis. Fault-tolerant clustering of wireless sensor networks. In *Proceedings of the 2003 IEEE Wireless Communications and Networking Conference*, volume 3, pages 1579–1584 vol.3, 2003. 10

[HMLU10] Danny Hughes, Ka Lok Man, Kevin Lee, and Jo Ueyama. A wireless sensor network based green marketplace for electrical appliances. In *Proceedings of the 2nd International Conference on Future Networks (ICFN)*, pages 299–303, 2010. 1

[HT05] Chi-Fu Huang and Yu-Chee Tseng. The coverage problem in a wireless sensor network. *Mobile Networks and Applications*, 10(4):519–528, August 2005. 9

[HTH+09] Danny Hughes, Klaas Thoelen, Wouter Horré, Nelson Matthys, Javier Del Cid, Sam Michiels, Christophe Huygens, and Wouter Joosen. Looci: a loosely-coupled component infrastructure for networked embedded systems. In *Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia*, MoMM '09, pages 195–203, New York, NY, USA, 2009. ACM. 28

[HUM+11] Danny Hughes, Jo Ueyama, Eduardo Mendiondo, Nelson Matthys, Wouter Horré, Sam Michiels, Christophe Huygens, Wouter Joosen, KaLok Man, and Sheng-Uei Guan. A middleware platform to support river monitoring using wireless sensor networks. *Journal of the Brazilian Computer Society*, 17(2):85–102, 2011. 1

[HV01] Qi Han and Nalini Venkatasubramanian. Autosec: An integrated middleware framework for dynamic service brokering. *IEEE Distributed Systems Online*, 2(7), 2001. 28

[JRBB11] Raja Jurdak, Antonio G. Ruzzelli, Alessio Barbirato, and Samuel Boivineau. Octopus: monitoring, visualization, and control of sensor networks. *Wireless Communications and Mobile Computing*, 11(8):1073–1091, 2011. 11

[JWOV11] Raja Jurdak, X. Rosalind Wang, Oliver Obst, and Philip Valencia. *Wireless Sensor Network Anomalies: Diagnosis and Detection Strategies*, volume 10 of *Intelligent Systems Reference Library*, chapter 12, pages 309–325. Springer, Berlin, Heidelberg, 2011. 9, 10, 20

[Kar10] Marcin Karpiński. *Synchronous Macro-Programming of Energy-Efficient Wireless Sensor-Actuator Networks*. PhD thesis, Trinity College Dublin, June 2010. 2, 7, 30, 35

[KC08] Marcin Karpiński and Vinny Cahill. Stream-based macro-programming of wireless sensor, actuator network applications with sosna. In *Proceedings of the 5th workshop on Data management for sensor networks*, DMSN '08, pages 49–55, New York, NY, USA, 2008. ACM. 35

[KI04] Bhaskar Krishnamachari and Sitharama Iyengar. Distributed bayesian algorithms for fault-tolerant event region detection in wireless sensor networks. *IEEE Transactions on Computers*, 53(3):241–250, March 2004. 11

[KLP+09] Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Răzvan Musăloiu-E, Philip Levis, Andreas Terzis, and Ramesh Govindan. Tosthreads: thread-safe and non-invasive preemption in tinyos. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 127–140. ACM, 2009. 28

[LBV06] Koen Langendoen, Aline Baggio, and Otto Visser. Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, april 2006. 1, 10

[LDH06] Xuanwen Luo, Ming Dong, and Yinlun Huang. On distributed fault-tolerant detection in wireless sensor networks. *IEEE Transactions on Computers*, 55(1):58–70, 2006. 11

[Lev12] Philip Levis. Experiences from a decade of tinyos development. In *Proceedings of 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 207–220, Hollywood, CA, 2012. USENIX. 7

[LGC04] Philip Levis, David Gay, and David Culler. Bridging the gap: Programming sensor networks with application specific virtual machines. Technical Report UCB/CSD-04-1343, EECS Department, University of California, Berkeley, 2004. 28

[LKK10] Woojin Lee, Juil Kim, and JangMook Kang. Automated construction of node software using attributes in a ubiquitous sensor network environment. *MDPI Sensors*, 10(9):8663–8682, 2010. 13

[LLWC03] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137. ACM, 2003. 29

[LMP+05] Philip Levis, Samuel Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. Tinyos: An operating system for sensor networks. In Werner Weber, JanM. Rabaey, and Emile Aarts, editors, *Ambient Intelligence*, pages 115–148. Springer Berlin Heidelberg, 2005. 27

[LRST07] Clemens Lombriser, Daniel Roggen, Mathias Stäger, and Gerhard Tröster. Titan: A tiny task network for dynamically reconfigurable heterogeneous sensor networks. In Torsten Braun, Georg Carle, and Burkhard Stiller, editors, *Kommunikation in Verteilten Systemen (KiVS)*, Informatik aktuell, pages 127–138. Springer Berlin Heidelberg, 2007. 13

[MFHH05] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, March 2005. 39

[MLZ+06] Cintia Borges Margi, Xiaoye Lu, G. Zhang, G. Stanek, Roberto Manduchi, and K. Obraczka. Meerkats: A power-aware, self-managing wireless camera network for wide area monitoring. In *Proceedings of the 2006 Workshop on Distributed Smart Cameras (DSC)*, Boulder (CA), 2006. 1

[MMW08] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: staged functional programming for sensor networks. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 335–346, New York, NY, USA, 2008. ACM. 38

[MOH04] Kirk Martinez, Royan Ong, and Jane Hart. Glacsweb: a sensor network for hostile environments. In *Proceedings of the 1st IEEE International Conference on Sensing, Communication, and Networking (IEEE SECON)*, pages 81–87, October 2004. 10

[MP11] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys*, 43:19:1–19:51, April 2011. 1, 2, 28

[MP12] Luca Mottola and Gian Pietro Picco. Middleware for wireless sensor networks: an outlook. *Journal of Internet Services and Applications*, 3:31–39, 2012. 1, 27

[NMW07] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 489–498, New York, NY, USA, 2007. ACM. 2, 34

[NTW05] Ryan Newton, Arvind Thiagarajan, and Matt Welsh. Building up to macroprogramming: an intermediate language for sensor networks. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, pages 37–44, april 2005. 34

[Obs09] Oliver Obst. Poster abstract: Distributed fault detection using a recurrent neural network. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks (IPSN)*, IPSN '09, pages 373–374, Washington, DC, USA, 2009. IEEE Computer Society. 11

[Oce08] Michael James Ocean. *The Sensor Network Workbench: Towards Functional Specification, Verification And Deployment Of Constrained Distributed Systems*. PhD thesis, Boston University, CS Department, September 2008. 13

[Pat08] Animesh Pathak. *Compilation of data-driven macroprograms for a class of networked sensing applications*. PhD thesis, University of Southern California, August 2008. 2, 13

[PH07] Lilia Paradis and Qi Han. A survey of fault management in wireless sensor networks. *Journal of Network and Systems Management*, 15:171–190, June 2007. 1, 9, 10, 20

[PMB+07] Animesh Pathak, Luca Mottola, Amol Bakshi, Viktor K. Prasanna, and Gian Pietro Picco. Expressing sensor network interaction patterns using data-driven macroprogramming. In *Proceeding of the 5th Annual IEEE International Conference on Pervasive Computing and Communications Workshops.*, pages 255 –260, march 2007. 15

[PP10] Animesh Pathak and Viktor K. Prasanna. Energy-efficient task mapping for data-driven sensor network macroprogramming. *IEEE Transactions on Computers*, 59(7):955–968, 2010. 19

[RB06] Stanislav Rost and Hari Balakrishnan. Memento: A health monitoring system for wireless sensor networks. In *Proceedings of the 3rd Annual IEEE Communications Society on Sensor and Ad Hoc Communications and Networks (SECON)*, volume 2, pages 575–584, 2006. 11

[RCK+05] Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. Sympathy for the sensor network debugger. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, SenSys '05, pages 255–267, New York, NY, USA, 2005. ACM. 11

[RLPB06] Sutharshan Rajasegarar, Christopher Leckie, Marimuthu Palaniswami, and James C. Bezdek. Distributed anomaly detection in wireless sensor networks. In *Proceedings of the 10th IEEE Singapore International Conference on Communication systems (ICCS)*, pages 1–5, October 2006. 11

[RNEV08] Ram Rajagopal, XuanLong Nguyen, Sinem Coleri Ergen, and Pravin Varaiya. Distributed online simultaneous fault detection for multiple sensors. In *Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN)*, pages 133–144, 2008. 11

[SFR12] Miruna Stoicescu, Jean-Charles Fabre, and Matthieu Roy. From design for adaptation to component-based resilient computing. In *Proceedings of the 18th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 1–10, November 2012. 2, 23

[SGaV+04] Eduardo Souto, Germano Guimarães, Glauco Vasconcelos, Mardoqueu Vieira, Nelson Rosa, and Carlos Ferraz. A message-oriented middleware for sensor networks. In *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, MPAC '04, pages 127–134, New York, NY, USA, 2004. ACM. 28

[SMP+04] Robert Szewczyk, Alan Mainwaring, Joseph Polastre, John Anderson, and David Culler. An analysis of a large scale habitat monitoring application. In *Proceedings of the 2nd international Conference on Embedded networked sensor systems*, SenSys '04, pages 214–226. ACM, 2004. 10

[SMR+12] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience*, 42(5):559–583, May 2012. 23, 58

[SMZ07] Kazem Sohraby, Daniel Minoli, and Taieb Znati. *Wireless Sensor Networks: Technology, Protocols, and Applications*. Wiley, 2007. 4

[SPMC04] Robert Szewczyk, Joseph Polastre, Alan Mainwaring, and David Culler. Lessons from a sensor network expedition. In Holger Karl, Adam Wolisz, and Andreas Willig, editors, *Wireless Sensor Networks*, volume 2920 of *Lecture Notes in Computer Science*, pages 307–322. Springer Berlin Heidelberg, 2004. 10

[WBS10] Hiroshi Wada, Pruet Boonma, and Junichi Suzuki. Chronus: A spatiotemporal macroprogramming language for autonomic wireless sensor networks. In *Autonomic Network Management Principles: From Concepts to Applications*. Academic Press, Elsevier, November 2010. 37

[WHVC05] Tsang-Yi Wang, Yunghsiang S. Han, Pramod K. Varshney, and Po-Ning Chen. Distributed fault-tolerant classification in wireless sensor networks. *IEEE Journal on Selected Areas in Communications*, 23(4):724–734, 2005. 12

[WLO+08] X.Rosalind Wang, JosephT. Lizier, Oliver Obst, Mikhail Prokopenko, and Peter Wang. Spatiotemporal anomaly detection in gas monitoring sensor networks. In Roberto Verdone, editor, *Wireless Sensor Networks*, volume 4913 of *Lecture Notes in Computer Science*, pages 90–105. Springer Berlin Heidelberg, 2008. 11

[YZ08] Ting Yuan and Shiyong Zhang. Secure fault tolerance in wireless sensor networks. In *Proceedings of the IEEE 8th International Conference on Computer and Information Technology Workshops*, pages 477–482, July 2008. 11

[ZS09] Xiao Zheng and Behcet Sarikaya. Task dissemination with multicast deluge in sensor networks. *IEEE Transactions on Wireless Communications*, 8(5):2726 –2734, May 2009. 13