2012

A Context-Aware Reflective Middleware Framework for Mobile Ad-hoc and Wireless Sensor Networks

Shengpu Liu
Lehigh University

A CONTEXT-AWARE REFLECTIVE MIDDLEWARE FRAMEWORK FOR MOBILE AD-HOC AND WIRELESS SENSOR NETWORKS

by

Shengpu Liu

A Dissertation
Presented to the Graduate and Research Committee
of Lehigh University
in Candidacy for the Degree of
Doctor of Philosophy
in
Computer Engineering

Lehigh University

January 2012

Approved and recommended for acceptance as a dissertation in partial fullfillment of the requirements for the degree of Doctor of Philosophy.

_____
Date

_____
Dissertation Director

_____
Accepted Date

Committee Members:

_____
Liang Cheng

_____
Edwin Kay

_____
Donald Hillman

_____
Tiffany Jing Li

This dissertation is dedicated to my wonderful family, particularly to my patient and understanding wife, Li, who is always encouraging me and taking care of my life. I must also thank my loving parents who have given me their fullest support.

Acknowledgments

I would like to thank all of those people who helped me make this dissertation possible.

I wish to thank my advisor, Dr. Liang Cheng for all his guidance, encouragement, support, and patience through out the time it took me to complete the research and write this dissertation. His sincere focus on "Original", "Significance", and "Scholarship" is my most important research guideline. The inspiration for doing this research came from the NSF project: Middleware for Adaptive Robust Collaborations across Heterogeneous Environments and Systems (MARCHES) that is headed by Dr. Cheng in the Laboratory Of Networking Group (LONGLAB) at Lehigh University. The project was one of the most important and formative experiences in my life.

My committee members Dr. Edwin Kay, Dr. Donald Hillman, and Dr. Tiffany Jing Li have given me very helpful insights, comments, and suggestions to improve my work. I thank them for their contribution and their good-natured support.

I am grateful to many persons who shared their knowledge and experiences, especially Dr. Qiang Wang and Dr. Qing Ye. They generously shared their meticulous research and insights that supported and expanded my own work. I also need to express my gratitude and deep appreciation to Lisa Frye, who has helped so much with proof-reading my academic papers and has given me a lot of

research suggestions.

I must acknowledge as well the many friends, colleagues, students, teachers, and other staffs who assisted, advised, and supported my research and writing efforts over the years. I need to thank especially Dr. Peng Yang, who was my classmate and is also my colleague now. He gave me a lot of suggestions on organizing and formatting this dissertation. My thanks must go also to Eric Xu Li, who has generously given his time to help me process my dissertation-related work.

Contents

List of Tables

List of Figures

Abstract

In smart environments, numerous devices need to be dynamically connected to form a Distributed Real-time and Embedded (DRE) system based on Mobile Ad-hoc NETworks (MANETs) or Wireless Sensor Networks (WSNs) and collaboratively react to changing contexts with dependable quality of service (QoS). Traditional middleware platforms, which have been designed as monolithic static systems, cannot effectively support the flexible and dynamic computing environments for emerging DRE applications. In consequence, there is an urgent need to provide a powerful adaptation approach for existing middleware.

Context-Aware Reflective Middleware (CARM), which supports dynamic reconfiguration and distributed behavior synchronization of component-based applications, has been an appealing technique for DRE systems in MANETs and WSNs. Existing CARM frameworks use single component-chain based architecture and synchronous synchronization protocols that are inefficient since they impose dependence restrictions and reconfiguration overhead. The achieved reconfiguration time is in a range of several seconds or even tens of seconds. We argue that they can not satisfy the efficiency requirements of some DRE applications in the dynamic environments, where reconfiguration is triggered every second or millisecond. Furthermore, there is no CARM framework implemented for extremely resource-limited wireless sensor nodes due to the complexity and overhead.

The key contribution of this dissertation research is the design and realization of a context-aware reflective middleware framework, called MassWare (Mobile Ad-hoc and Sensor Systems' Middleware), to meet the efficiency requirement of such adaptive DRE applications in MANETs and WSNs. Our thesis is that the reconfiguration efficiency can be improved by asynchronous synchronization support via a middleware framework. To prove this thesis, we propose a multiple component-chain based middleware architecture and an active-message oriented asynchronous synchronization protocol for the reconfiguration. The key idea behind the protocol is that each application-layer data packet takes an active message header that indexes the correct component-chain of the packet receiver to process the data payload. Therefore, the distributed behavior synchronization time is dramatically reduced by eliminating the operation suspension time and buffer clearance time. Based on the protocol, we have developed MassWare in MANETs and WSNs that helps the DRE applications adapt to changing contexts in an efficient and robust way according to user-defined adaptation rules.

In this dissertation, we describe the complete architecture design, model analysis, and implementation of MassWare, which addresses the major challenges of existing CARM frameworks: improving reconfiguration efficiency, realizing CARM in WSNs, and offering a unified development model for both MANETs and WSNs. MassWare and supported applications have been implemented on PDA platforms and Mica sensor nodes. The reconfiguration efficiency has also been analyzed and compared with those of peer CARM frameworks based on a novel theoretical model. Quantitative empirical results show that the reconfiguration time of MassWare for MANETs is reduced from seconds to hundreds of microseconds. Evaluations demonstrate that MassWare is robust, scalable and generates a small memory footprint.

Chapter 1

Introduction

1.1 Context-Aware Reflective Middleware and Applications

1.1.1 Context-Aware Reflective Middleware

Middleware [1][2] is a distributed software layer that sits above the network operation system and below the application layer and abstracts the heterogeneity of the underlying environment. Traditional communication middleware, like CORBA [3], Java RMI [4], and DCOM [5], has been a critical technology in the construction of distributed applications. Recently, there is a need to migrate the middleware platforms, which have been designed as monolithic static systems, to more flexible and dynamic computing environments due to the popularity of portable devices (e.g. laptops, PDAs, and sensor nodes) and advances in wireless communication techniques (e.g. Wi-Fi and ZigBee). The limited resource and dynamic resource availability requires applications to be adaptive and reconfigurable at runtime to improve performances in the Mobile Ad-hoc Networks (MANETs) and Wireless Sensor Networks (WSNs).

Adaptive and reflective middleware [6][7] has the ability to inspect its internal states by providing a representation of its internals through a process called reification, and allows the internals to be dynamically manipulated and runtime reconfigured through a process called absorption [8][9] to change its functional behaviors. The adaptive and reflective middleware uses component-based metamodel to build applications, in which an application consists of a set of interacting reflective components (e.g. a component chain). Therefore, the reconfiguration process is realized via the component interface-metamodel, which is able to dynamically discover and access the component interfaces to change its attributes and functions, and the application architecture-metamodel, which is able to access

and reconfigure the component graph (e.g. the component chain structure).

Context-aware reflective middleware (CARM) [10][11] can monitor real-time contextual information and adapt the application behaviors to the context changes. It provides a powerful reconfiguration approach to build Distributed Real-time and Embedded (DRE) systems [12] in mobile wireless environments because it can adapt the systems autonomously to changing contexts to ensure required quality of service (QoS) [13]. The reconfiguration process includes local behavior change and distributed behavior synchronization (e.g. changing or adding a compression component in local program may require a corresponding change or insertion of a decompression component in distributed peer programs).

1.1.2 Applications in Mobile Ad-hoc Networks

Distributed real-time and embedded (DRE) systems [12], such as aircraft mission planning systems in battlefield, rapid response systems, and vehicle safety systems in unmanned intelligent vehicles, provide an important approach to bridging the gap between the cyber world and the physical world. Generally, DRE systems are large-scale, integrated, and time-sensitive and operate in dynamic and resource limited environments [14]. This challenges system designers and developers when such DRE systems must be developed from scratch. Fortunately, CARM techniques may be used to address this challenge by reducing application development and maintenance costs, enabling component-based system integration, and supporting time-sensitive and resource-limited application.

To clarify the potential advantages behind the context-aware reflective middleware for DRE systems, we present an example of possible use case in vehicle safety applications (see Fig. 1.1). Suppose a road has two lanes in one direction, on which car 1 and car 5 are in lane 1 and car 2, 3, 4, and 6 are in lane 2. There are

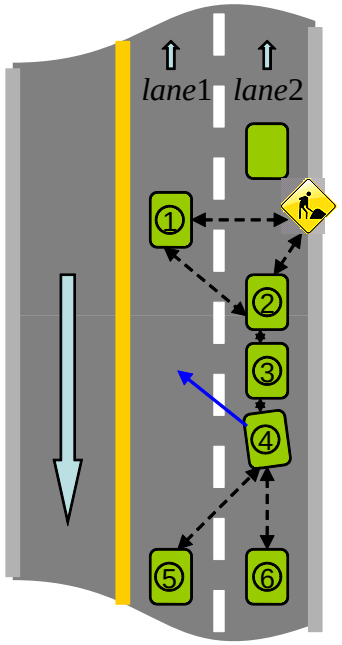


Figure 1.1: An example of vehicle application scenario.

two scenarios that a vehicle system may need to adapt its behaviors to real-time contexts.

The first scenario is for robust communication. Car 1 and car 2 share their visions by exchanging image data for action replan when they drive closely while both only have partial vision of the road condition. Each image frame is separated into tiles and transmitted in a sequence based on different priorities. The tiles closer to the interest point have higher priority and will be transmitted first with high image quality. However, the network condition, e.g. the bandwidth, between car 1 and car 2 is dynamic and volatile. The middleware can automatically measure the bandwidth and adaptively reconfigure the compression behaviors at runtime, e.g. using or not using compression component, or setting varied compression ratio, to satisfy the required QoS, like the specified transmission time, of the application while provide images as clear as possible.

The second scenario is for action replan. Car 4 finds that it is too congested to drive in lane 2 while there are fewer cars in lane 1 by communicating with nearby cars and roadside infrastructure. It then decides to switch to lane 1 to reduce traffic congestion. The middleware in this scenario will automatically collect the position and speed information of neighbor cars and the road conditions and then make the decision of switching to lane 2 by adjusting the direction and speed parameters of its software control components.

Another example is a dynamic collaborative mission planning DRE system that contains a command and control (C2) aircraft and a fighter aircraft [15]. The fighter aircraft and the C2 aircraft establish a collaboration to exchange virtual target folders (VTFs), consisting of image data to update the fighter's mission which is required to be completed in milliseconds for some critical avionics tasks. Context-aware reflective middleware can adapt the real-time collaboration task to the dynamic constraints of the embedded system. It breaks a request for a VTF image into tiles, monitors the progress of the tile acquisition, and changes the quality level of subsequent tiles to compensate for late or early downloading of an image.

1.1.3 Applications in Wireless Sensor Networks

Wireless Sensor Networks (WSNs) [16] can gather sensory data from the physical world and monitor environmental conditions. Therefore, they have also played an important role in the smart cyber-physical systems. A WSN consists of large numbers of low-cost networked sensor devices (also called nodes), which are capable of sensing, computation, and wireless communication. The nodes can sense and process environmental data and relay the sensor readings of other nodes to the base station through automatically constructed ad-hoc networks. Compared

with wired networks, WSNs can be deployed in highly dynamic and heterogeneous environments to perform distributed sensing and collaborative data processing.

WSN application areas include cyber-physical systems, habitat monitoring, intruder detection, infrastructure health monitoring, and in the future, possibly integrating all human-life applications in smart environments. The applications have benefited from advances in context-aware reflective middleware.

Data compression is an attractive in-network processing research topic in WSNs for reducing communication overhead since the amount of energy needed to send one bit of data is equivalent to the amount of energy consumed by executing thousands of instructions to produce the same data [17]. Based on its contextual information, a sensor node could select different compression algorithms to minimize the data redundancy. For example, when a node detects that it has multiple neighbor nodes (high density scenario), it needs to select a Distributed Source Coding (DSC) algorithm to reduce distributed redundancy (the redundancy among local sensed data and data sensed from neighbor nodes). However, when the neighbor nodes are in sleep mode or dead (low density scenario), the node needs to select an independent algorithm (e.g. Unary Coding) to only reduce local redundancy since there is no distributed redundancy.

Reprogramming WSNs [18] over the air is an appealing technique for the management and maintenance of WSNs because manually "burning" programs to sensor nodes is labor intensive or even impossible after the nodes have been deployed. In traditional reprogramming frameworks, a WSN program is compiled into a monolithic code image, in which application modules and TinyOS kernels are statically linked, and the entire code image needs to be updated even for minor changes. Context-aware reflective middleware supports dynamically loaded software components and provides a new reprogramming technique to update only specific components. Furthermore, context-aware reflective middleware supports

context measurement and dynamic reconfiguration of WSN applications by dynamically loading and unloading components according to the contextual information
.

1.2 Motivations and Objectives

1.2.1 Application Requirements

The advances in microelectronics and wireless communication techniques have benefited large-scale distributed, real-time, and embedded (DRE) systems [12], such as cyber-physical systems [19], rapid response systems [20], vehicle safety systems in unmanned intelligent vehicles [21][22][23][24][25], and possible all human-life applications in smart environments. Generally, the DRE systems are time-sensitive, heterogeneous, and integrated with MANETs and WSNs and operate in dynamic and resource limited environments, which challenges DRE system designers and developers.

In MANETs, being real-time is one of the most critical requirements of DRE systems. For the previous example, unmanned intelligent vehicles with DRE systems can reconfigure their behaviors (direction and speed) to adapt to situational contexts collected at runtime through temporally built ad-hoc and dynamic networks based on vehicle-to-vehicle and vehicle-to-roadside communications. However, the long reconfiguration time may result in critical accidents and loss of lives and property. In fact, two cars could hit each other in 1.5 seconds when they drive face to face based on the 3 second safe distance rule, which requires a vehicle safety system to respond in hundreds of milliseconds.

the reconfiguration time of the existing context-aware reflective frameworks

[26][8] is too long to be acceptable for time-critical DRE systems. The reconfiguration time is normally in the range of seconds or more according to the data reported in literature, but a DRE system requires the total processing time within 10ms for time-critical missions [27]. The reconfiguration process of a DRE application consists of two steps: local behavior change, which modifies the structure of the local functional path (or component chain), and distributed behavior synchronization, which coordinates distributed behaviors after the local behavior is changed. For example, in a distributed mobile video transmission application, changing or adding a compression component in a sender program (a local behavior) requires a corresponding change or insertion of a decompression component in the receiver programs (a distributed behavior). The long reconfiguration time of existing CARM techniques is caused by the inefficiency of their synchronization protocols, which are synchronous and require the synchronization participants to be blocked until the reconfiguration process is completed.

In WSNs, to the best of the author's knowledge, there is no existing context-aware reflective middleware. Due to the tight integration with the physical world and limited resources in early-stage sensor platforms, applications are usually constructed as monolithic programs that include the underlying embedded operating system and are tightly coupled with hardware components of sensor nodes. The monolithic application structure has two disadvantages. First, it hampers reusability. WSN applications often need to be developed from scratch, which increases the developer workload and the development cost. Second, it makes the sensor reprogramming process energy-intensive and error-prone because the whole application has to be updated even for a minor change of the application.

Another requirement of existing WSN applications is the flexibility and adaptability in mobile environments [28]. Because sensor nodes are often randomly deployed in heterogeneous environments, each individual node needs to deal with

different situations and even change its behavior at runtime. However, all sensor nodes are traditionally programmed with the same code and difficult to be changed once they are deployed, which hampers their adaptation capability.

1.2.2 Middleware Requirements

Middleware has succeeded because it masks the heterogeneity of underlying environment and simplifies the task of programming and managing applications. Traditional middleware focuses on integrating distributed computing systems to serve as a unified resource to reduce the application development cost. However, there are some challenges when migrating the traditional middleware to the flexible and dynamic mobile devices and sensor nodes due to their limited resource and dynamic resource availability. First, the hardware and operating systems deployed in these platforms may be significantly different. Mobile devices, like PDAs and S-mart Phones, may host Giga-Hertz processer and hundreds of Mega-Bytes memory and support general operating systems (e.g. Window Mobile OS) and Wi-Fi communication, while sensor nodes only have Mega-Hertz processor and Kilo-Bytes memory and support device-specific operating system (e.g. tinyOS) and ZigBee communication. Second, the application development model and programming techniques are also different in mobile devices and sensor nodes. Mobile devices normally support generic high-level programming languages (e.g. C# and .NET in Windows mobile OS) and multiple threads, while sensor nodes only support device-specific programming languages (e.g. NesC) and a single thread.

These hardware and software differences make the middleware techniques in MANETs and WSNs distinct and require the application developers of DRE systems to have expertise in both areas. Therefore, it is desirable to provide a unified programming interface for developing both MANET and WSN applications.

The middleware should be able to handle the low-level hardware and software heterogeneity and provide high-level services to the application developers. The developers can then efficiently construct an application based on existing software components, choose application interested contexts for measurement, and define the policies how the application adapts to the measured contexts.

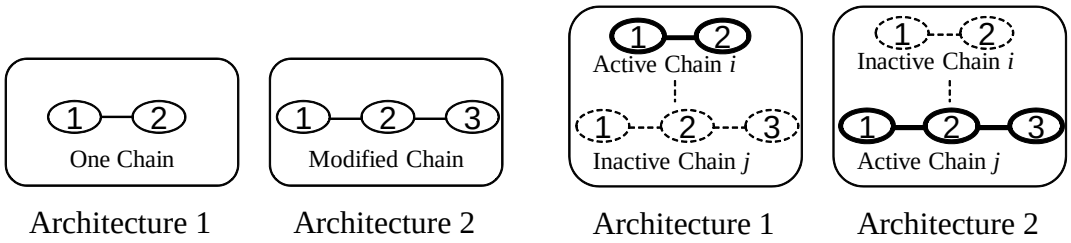
1.2.3 Objectives

According to the requirements and motivations mentioned above, the objectives of this dissertation include:

- Improve the reconfiguration efficiency of traditional context-aware reflective middleware to satisfy the real-time requirement of DRE systems in MANETs.
- Propose context-aware reflective middleware for wireless sensor nodes so that each individual node can adapt its behavior to the dynamic environments in WSNs. The middleware can also benefit sensor reprogramming techniques to update only required software components.
- Provide a unified middleware framework for developing context-aware reflective applications in both MANETs and WSNs. The framework should be simple to use while flexible enough to develop generic applications.

1.3 Contributions and Significance

This dissertation focuses on the designs of context-aware reflective middleware for MANETs and WSNs. The significant contributions of the dissertation are as follows:



(a) Single-component chain architecture in existing middleware

(b) Multiple-component chain architecture in MassWare

Figure 1.2: Dynamic reconfiguration architecture

- MassWare solves the critical issue of the long reconfiguration time of context-aware reflective middleware. Compared to the traditional middleware that supports single component-chain based application architecture (Fig. 1.2a), MassWare-MANET maintains multiple component chains (Fig. 1.2b). Therefore, there is a new method proposed for the local behavior change that switches active and inactive chains, which replaces the traditional method of modifying the single-chain structure to reduce the local behavior change time.

  Further, based on the multi-component chain architecture, an efficient active-message based synchronization protocol is designed to asynchronously coordinate the behaviors of distributed programs and the distributed behavior synchronization time is dramatically reduced by eliminating the operation suspension time and buffer clearance time required by existing middleware techniques.

  We have proposed a generic analytical model for compariing the reconfiguration efficiency of various CARM frameworks. According to the analysis and empirical measurement results, we conclude that the reconfiguration time in existing adaptive and reflective middleware has been reduced from second-s to milli-seconds. the magnitude reduction of application reconfiguration

time enabled a richer set of DRE systems for cyber-physical interactions to be designed and implemented.

- MassWare also offers, to the best of our knowledge, the first context-aware reflective middleware framework that has been implemented in a single sensor node. MassWare is a component-based middleware built on top of SOS [29] (a module-based dynamic operating system for WSNs). A MassWare component provides a set of interfaces through which it can change its states at runtime and communicate with other components. MassWare has the ability to dynamically update these software components, reconfigure the connections between them, and synchronize the reconfigured behavior of a sensor node with the base station. Moreover, MassWare can measure environmental contexts of sensor nodes and then adapt sensor application behaviors to the changing contexts at runtime based on user-defined policies. The MassWare framework and supported applications have been implemented and evaluated in MicaZ nodes. Experimental results show that MassWare is energy efficient with small memory footprint.

- We have design and implement a unified context-aware reflective middleware, called MassWare (Mobile Ad-hoc and Sensor Systems middeWare), for both MANETs and WSNs. To develop a context-aware reflective application based on MassWare, developers only need to provide a script file in XML syntax to describe the application-required functional components, measurement tool components, and adaptation policies. The middleware then constructs the application, measures application contextual information, adapts the application behavior to the contexts according to the defined adaptation policies, and synchronizes with peer middleware agents or the base station. MassWare includes two separate middleware frameworks: MassWare-MANET for

mobile ad-hoc networks and MassWare-WSN for wireless sensor networks.

1.4 Terminologies

The following terminology will be used in this dissertation:

- Synchronization is the process of coordinating the behaviors of collaborative programs in a DRE system. When the behavior of a local program is reconfigured to adapt to changing contexts, it requires its peer programs to change their behaviors correspondingly for system consistency.

- Asynchronous synchronization means that the synchronization is realized through an asynchronous method, in which the local program can resume its operation right after its own behavior is changed for adaptation and other synchronization participants reactively change their behaviors only when they communicate with this local program.

- Detector is the hierarchical context event sensor that can organize and evaluate specified contexts at runtime and notify subscribed actuators for adaptation.

- Actuator is a reflective component that contains a set of functional components and a meta-interface. The functional components form a functional path or component chain, which process application-layer data. The meta-interface can represent its internal states and reconfigure the actuator behaviors at runtime though component parameter tuning and chain structure reconfiguration.

- Active actuator means that the actuator status is active. There is one and only one actuator active at any time and only the component chain in the

active actuator processes application-layer data. Various actuators can be activated or deactivated to adapt to changing contexts according to user-defined policies.

- Proactive actuators are the actuators constructed at the system initialization phase to process local data. They can proactively change their behaviors to adapt to changing contexts at runtime according to user-defined adaptation policies (rules).

- Reactive actuators are the actuators constructed at the system synchronization phase to process received data from peer programs. They reactively change their behaviors according to the active message header of the received data packet.

1.5 Organization of the Dissertation

The rest of this dissertation is organized as follows. Chapter 2 covers the existing research results of related context-aware reflective middleware. Chapter 3 presents the MassWare-MANET reflection model and system architecture and Chapter 4 presents the MassWare-WSN design and implementation. In Chapter 5, we theoretically analyze the reconfiguration time of MassWare and compare it with peer research, followed by the system implementation and experiment validation. In Chapter 6, some MassWare-supported applications are designed and implemented. The dissertation concludes with Chapter 7.

Chapter 2

Related Work

2.1 Middleware for Mobile Ad-hoc Networks

Middleware has been a critical technology for developing DRE systems because it can mask the heterogeneity of the underlying environment and simplify the task of programming and managing applications. It can be categorized into multiple layers (Fig. 2) based on the various functions provided for DRE systems.

2.1.1 Communication Middleware

Communication middleware focuses on integrating distributed computing systems to serve as a unified resource to reduce the application development cost. Early stage middleware, like CORBA [3], Java RMI [4], and DCOM [5], is built on Remote Procedure Call (RPC) to abstract the low-level TCP/IP communication details and replace the communication interface with a local procedure call or function invocation.

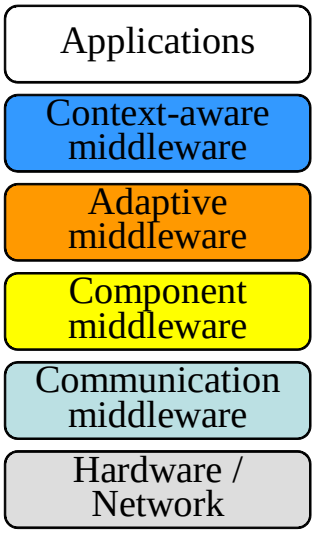


Figure 2.1: Middleware layers.

2.1.2 Component Middleware

Component middleware, normally based on a component model (e.g. CORBA Component Model [30]), enables reusable service components to be organized, configured, and deployed for developing applications efficiently and robustly. Component middleware provides standards for object implementations and interactions so that it can support generic service components and then reduce the complexity of software upgrades and increase the reusability and flexibility of distributed applications. Existing component middleware contains both reusable common services,

e.g. optimization of resource consumption (OSA+ [31], ACE [1]), configurability (TAO [32], Zen [33]), reusability (nORB [34]) etc., and domain-specific services, e.g. OSEK/VDX [35] for vehicle applications and ARINC 653 for avionics.

2.1.3 Adaptive and Reflective Middleware

Adaptive and reflective middleware [36][37][38] has the ability to inspect its internal states by providing a representation of its internals through a process called reification. It also allows the internals to be dynamically manipulated and reconfigured through a process called absorption, which changes its non-functional and functional behaviors. The non-functional behavior reconfiguration is realized by dynamically replacing or changing the non-functional components of the middleware, like security check and concurrency control, etc. The functional behavior reconfiguration is realized by reconfiguring the functional components of the application at runtime. Open ORB [6] provides both structural reflection for functional component reconfiguration and behavioral reflection for nonfunctional component reconfiguration. Dynamic TAO [7] is a reflective ORB based on a set of component configurators. The TAOConfigurator can inspect and dynamically change its nonfunctional behaviors.

2.1.4 Context-aware Reflective Middleware

Context-aware reflective middleware can measure applications' situational contexts and adapt application behaviors to them at runtime. It may be further divided into QoS-enabled middleware [39][40][41] and user-defined context-aware middleware [42][43]. QoS-enabled middleware can dynamically measure application-specific QoS and provide QoS reservation or adaptation to guarantee the required QoS, e.g.

MUSIC [10], CIAO [44], Qoskets [45][46], and QuO [47][48]. User-defined context-aware middleware supports not only application QoS, but also any other user-defined contexts. MARCHES [49], MADAM [11], MobiPADS [26], and CARISMA [8] are some example systems that belong to this category.

QuO (Quality Objects) [50][51][52] is a distributed object computing framework based on the CORBA model. It provides a QoS monitor and composes dynamic QoS provisioning capacity into DRE systems. QuO separates the QoS provisioning functionality from the application functionality; however, it still relies on ACE and TAO as it must use ORB based communication interfaces (e.g. TAO A/V streaming) and QoS tools (e.g GQoS and IntServ). MUSIC separates the self-adaptation concern from the business logic concern and delegates the complexity related to self-adaptation to generic middleware. It offers an adaptation-planning framework to evaluate the utility of alternative configurations in response to context changes, select a feasible one (e.g., the one with the highest utility) for the current context, and adapt the application accordingly. MADAM is a type of client/server based CARM for adaptive mobile applications. A master node (client) negotiates with slave nodes (servers) for an adaptation decision. It provides both reactive and proactive negotiation mechanisms for distributed adaptation decision. None of these frameworks provides any synchronization functionality; they assume that the adaptation has been constrained in safe conditions in advance. For example, the reconfiguration in QuO must be carefully studied so that the received data can still be understood by the receiver after reconfiguration.

MobiPADS [26] is a policy- (or rule-) based CARM framework for mobile applications. It supports both middleware-layer and application-layer adaptations according to user-defined policies. A client middleware agent uses a communication channel to synchronize the application behaviors with a server middleware agent in a synchronous way whenever the architecture is reconfigured. The reconfiguration

process includes operation suspension, buffer clearance, and chain-structure modifications. Because the initiator of the synchronization must be suspended until the system architecture of its own and other participants is reconfigured and the buffered data for previous architecture is processed, the reconfiguration time is in a range of seconds or even more according to the published experimental results. CARISMA [8] employs a novel micro-economic approach that relies on a particular type of sealed-bid auction to handle the adaptation conflicts between distributed policies. The processing time of the conflict resolution algorithm includes communication time among peer agents for message exchanges and local computation time for context evaluation, bidding calculation, and solution set computation. This reconfiguration process is still synchronous and the conflict resolution algorithm must be invoked whenever a context is changed. Similar to these frameworks, MassWare is also a policy-based CARM framework and focuses on the reconfiguration of stateless applications. MassWare is different from existing work because it maintains multiple component chains and leverages the active messages to realize the synchronization in an asynchronous way. According to analysis and evaluations, MassWare can significantly reduce the reconfiguration time and satisfy the responsiveness requirement of DRE systems. The preliminary results of MassWare were published in [49] and a substantial extension of the system and thorough evaluation of its performance based on a proposed analytical model and experiments is presented in [53].

2.2 Middleware for Wireless Sensor Networks

2.2.1 WSN Middleware Frameworks

Wireless Sensor Networks (WSNs) consist of large numbers of low-cost, small-scale sensor nodes, which can sense and process environmental data and transmit the sensor readings or processed results to the base station through automatically constructed wireless ad-hoc networks. WSNs have shown many benefits in the application areas of environment monitoring, event detecting, and object tracking. And they will be an attractive means to bridge the gap between the physical world and virtual cyber world in future smart environments. On the other hand, developing sensor applications is a very challenging task due to WSN characteristics. First, sensor nodes are very limited in the hardware resource and energy. Second, node mobility, node failures, and environmental obstructions make WSNs highly dynamic. Third, the large number of sensor nodes also makes the deployment of sensor applications difficult. Middleware is a novel approach for hiding low-level implementation details and providing standard high-level interfaces to facilitate the development of WSN applications.

Most WSN middleware frameworks focus on implementation of basic sensing and routing operations. COUGAR [54] views sensor networks as a virtual database and provides an SQL like language to query sensor data from the networks. Similar to COUGAR, TinyDB [55] is also a sensor database system, which uses a semantic tree routing protocol to accurately determine when queries should be propagated from a node to its children to save energy and extend battery life. TinyDB also supports event-based query and allow queries to be triggered by events generated by other queries or a sensor program. SINA [56] is a more comprehensive sensor database system. It not only supports SQL-like languages for sensor queries, but

also incorporates such low-level mechanisms as hierarchical clustering of sensors and efficient data aggregation. SINA views the sensor network as a logic datasheet composed of cells and each cell represents an attribute of a sensor node. It also provides a language called SQTL (Sensor Query and Tasking Language), which can be injected into the network at run-time, for sensor hardware access, communication, and event handling. The database-oriented approaches are only suitable for homogeneous networks because they require each sensor node to have identical data structure.

TinyLIME [57] supports efficient data query from local sensors (one-hop) based on a tuple space model with shared memory. TinyLIME applications create tuple templates whose formats are determined by sensor nodes, and subscribe them to the sensor nodes for their interested data. MiLAN [58] provides a standard API for applications to specify their sensing requirements, like required data type, data sets, and sensor Quality of Service (QoS) etc. The middleware can retrieve the current application state and efficiently configure the network, so that only required sensor nodes are organized to meet the application requirements.

The above middleware frameworks focus on the entire network and view nodes as basic sensing elements. Along with the rapid progress of hardware resource of WSNs, more and more data processing tasks have been migrated to individual sensor nodes to extend battery life as the energy cost of sending one single bit of data can consume the energy of executing thousands of instructions to produce the same data [17]. As applications become more complex, middleware is also required for these tiny sensor nodes. Mate [59] uses a virtual machine approach built on TinyOS to hide low-level operations and interpret received byte codes, which are broken into capsules. Mate programs can be easily replaced by injecting new capsules, which makes the network dynamic, flexible and reconfigurable. However, Mate is not suitable for complex sensor applications. First, the interpretation

overhead for large applications is wasteful; Second, Mate capsules, which contain 24 instructions at most, are not meaningful service components and difficult to maintain or upgrade; Third, The interaction between different capsules is not expressive. A capsule can call another subroutine capsule, but there is no message exchange between capsules; Fourth, Mate only supports bytecode; a higher-level language and a programming model for application development are needed. Magnet [60] and Impala [61] also design a virtual layer to mask the low-level hardware operation and heterogeneity for each node and provide a high-level interface to simplify application development and support application adaptation. However, they require complex software support (e.g. Magnet requires a Java virtual machine, and Impala only works on Linux systems) and not suitable for tiny sensor nodes.

2.2.2 WSN Reprogramming

Reprogramming WSNs [18] over the air is a desired technique for the management and maintenance of WSNs as manually "burning" programs to all sensor nodes is labor intensive or even impossible once they are deployed. On the other side, reprogramming is also a challenging task. In TinyOS, the current state of the art operating system for WSNs, a compiled program is a monolithic code image in which application modules and TinyOS kernels are statically compiled and globally optimized for execution efficiency. Therefore, the entire code image needs to be updated even for minor changes. In MNP [62] and Deluge [63], a program is divided into several segments (or pages), which are transferred in a pipeline fashion in networks. They use REQ packets as negative acknowledgement (NACK) to guarantee the integrity of the program. Incremental Network Programming [64] uses the Rsync algorithm to generate the difference between the two program

images and only transmit the incremental changes for the new program version. The powerful host machine (e.g. base station) keeps the histories of the program versions of all code receivers (e.g. sensor nodes) and calculates the differences locally to save communication energy. However, this algorithm is not suitable for major function changes or retasking in WSNs as the difference of compiled binary programs are rather large in these situations.

2.2.3 SOS: A Dynamic Sensor Operating System

SOS [29] is a new dynamic operating system for WSNs. Different from TinyOS, SOS consists of a statically-compiled kernel and dynamically-loaded modules. The kernel supports dynamic memory allocation, message scheduling, and loading and unloading modules. It also provides sensor APIs to help modules interact with sensor drivers. SOS modules are position-independent binaries that implement a specific task or function. The modules can communicate with each other and with the kernel through a direct function call or by passing asynchronous messages that are handled by the message scheduler. The dynamic memory is used to store module states and pass data address across various modules. SOS not only provides a reprogramming technique to update required functional modules, instead of the entire program, but also supports dynamic reconfiguration by dynamically linking and unlinking modules for WSN applications. MassWare is a context-aware reflective middleware framework built above SOS. Compared to SOS modules, MassWare components provide a set of interfaces that can be used to inspect and modify component states and communicate with each other. MassWare also supports context measurement, adaptation, and synchronization so that each sensor node can adapt its behavior to environmental contexts according to user-defined policies.

Chapter 3

MassWare for Mobile Ad-hoc Networks

3.1 System Architecture of MassWare

MassWare-MANET (also called MassWare in this chapter) uses a layered architecture to monitor contexts and adapt supported-applications to the contexts according to user-defined policies. It supports both component-level reflection for the accommodation of standard components and system-level reflection for the reconfiguration of component connections, it contains a hierarchical event notification model to efficiently evaluate comprehensive contexts, and it provides a lightweight XML-based script language to describe and manage adaptation policies.

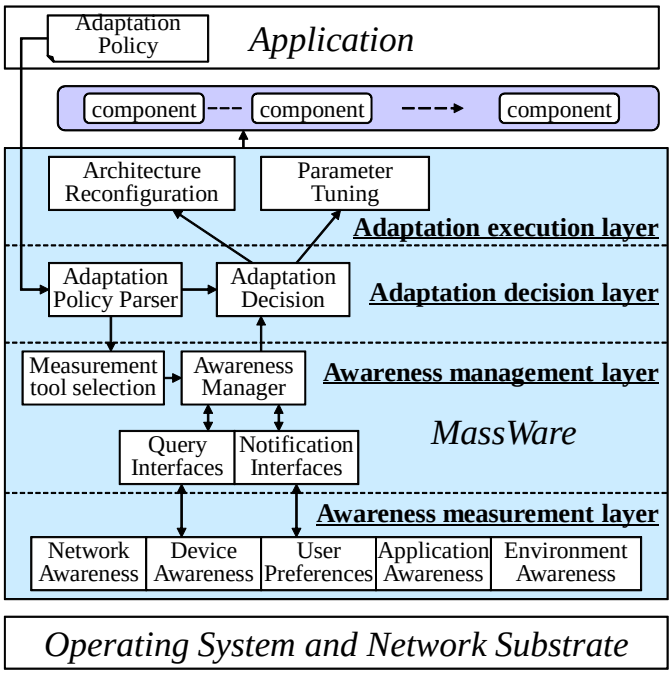


Figure 3.1: System architecture of MassWare-MANET.

MassWare is located between the upper application layer and the lower operating system and network layer to monitor contexts and support application adaptations. It is peer-to-peer middleware with one middleware agent per application in each host. MassWare consists of four major function layers as depicted

in Fig. 3.1:

- The awareness measurement layer consists of individual measurement tools, which may measure context-awareness information about networks, devices, end-user preferences, application internal states, and physical environments.
- The awareness management layer hosts an awareness manager that communicates with the measurement layer through notification and query interfaces. It organizes and evaluates measured contexts based on event trees (called detectors) built on a hierarchical event notification model.
- The adaptation decision layer has a script parser and a decision engine. The script parser parses the adaptation policy script file defined by application developers based on a declarative language in an XML format. The decision engine takes the adaptation policy file as input, creates the awareness manager and a reconfigurator in the adaptation execution layer, and subscribes the actuators in the reconfigurator to the detectors in the awareness manager according to the adaptation policies. This allows the actuator to be triggered by context changes for reconfiguration according to the policies.
- The adaptation execution layer contains a reconfigurator to execute the behavior changes of functional and nonfunctional components. In this dissertation, we focus on the functional reconfiguration for improving the performance of DRE systems, which includes the component chain reconfiguration and component parameter tuning. Between the middleware and application, there is another layer called the operation layer, in which various services are offered by software components. MassWare supports application-specific components and standard third-party components based on its reflection model.

Because MassWare-MANET aims at improving the reconfiguration efficiency of DRE systems, the thesis focuses on stateless applications and the reconfiguration of application-layer functional components. The proposed synchronization protocol can be combined with state-machine and model-based reconfiguration techniques to support the reconfiguration of stateful applications [65]. We also leave the reconfiguration of middleware-layer nonfunctional components, e.g. the concurrency, security, etc. for future work, which can potentially be supported by MassWare.

In summary, MassWare is responsible for monitoring situational contexts that trigger adaptations, deciding when, where, and how to adapt application behaviors, and for executing the adaptation policies specified by application developers at runtime.

3.2 MassWare Reflective Model

MassWare supports both component-level and system-level reflection. The component-level reflection deals with the content and behavior of a given component via an interface metamodel, which provides a way to discover and access the interfaces of a software component. Thus, reflective components can be supported by MassWare to incorporate new techniques and services and deal with the upgrade and extension of DRE systems. The system-level reflection deals with the structure of the component connections via an architecture metamodel, which enables the discovery and operation of the current active component chain. The system-level reflection allows MassWare to examine its internal states at runtime and dynamically reconfigure the application architecture to enhance its adaptability.

3.2.1 Components and Component-level Reflection

A MassWare component is a function-independent reflective element that provides an interface metaobject. This interface metaobject enables a component to read its own metadata, extract the metadata from the component (called reification), and use that metadata to either inform the component user or modify the component's behavior (called absorption). By using the interface metamodel and component-level reflection, MassWare can examine the types in a standard component, create new types at runtime, instantiate the types, and dynamically invoke properties and methods on the instantiated objects (called late binding).

```
<Masslets>
 <component cid="2002">
  <addr> D:\Masslets\JPEG.dll </addr>
  <name> Masslets.Compress.JPEG </name>
  <ctype> Masslet </ctype>
  <alias> COMPRESS </alias>
  <param pid="001">
   <name> SetCompressQuality </name>
   <vtype> Int32 </vtype>
   <value> 50 </value>
  </param>
  <interface iid="001">
   <name> PtrDataInput </name>
   <itype> Input </name>
   <Message> PDIBEventArgs </Message>
  </interface>
  <interface iid="002">
   <name> DataOutput </name>
   <itype> Output </name>
   <Message> JPEGEventArgs </Message>
  </interface>
 </component>
 ...
</Masslets>
```

```
<MassTools>
 <component cid="3001">
  <name> Awaretools.AvailableBW </name>
  <alias> AVI_BW </alias>
  <param pid="001">
   <name> packetSize </name>
   <vtype> Int32 </vtype>
   <value> 64 </value>
  </param>
 <param pid="002">
   <name> packetNum </name>
   <vtype> Int32 </vtype>
   <value> 2 </value>
  </param>
 <param pid="003">
   <name> Interval </name>
   <vtype> Int32 </vtype>
   <value> 300 </value>
  </param>
  <interface iid="001"> ... </interface>
  ...
</component>
 ...
</MassTools>
```

Figure 3.2: The component declaration in MassWare-MANET.

To incorporate a new software component in MassWare, users need to describe the types, interfaces, and other attributes of the component in a system script file using the defined IDL (Interface Description Language), as shown in Fig. 3.2. There are three methods to identify a MassWare component: 1) the exclusive

component name for a registered system component, 2) the complete address for a local component, or 3) the desired attributes for a registered component in the component manager. The component type is declared in the ctype part and the alias is the name of the component used in the adaptation policy part of the script. The component can be specified by setting its parameters, which can also be reconfigured at runtime according to adaptation rules. It also provides some interfaces. The input and output interfaces can be bound together through connectors if they support compatible event messages and their connections can also be reconfigured at runtime.

There are two types of MassWare components: reconfigurable functional components (namely masslets) and extensible context-awareness components (namely masstools).

Masslets are the basic functional units to construct DRE systems. Each masslet provides output and input interfaces for component assembly and communication based on the publish/subscribe model [66]. An output interface of a masslet can be subscribed by message-compatible input interfaces of other masslets and can publish messages to them through connectors.

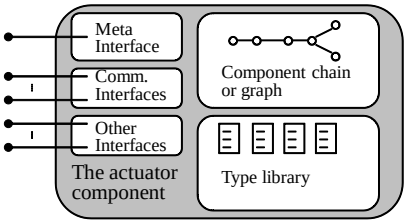
Masstools, which measure and predict real-time context changes in MassWare, are realized as reflective components to facilitate the reuse and extension of existing measurement tools. Masstools act as the lowest event sources that can be subscribed by higher level event nodes and organized in a hierarchical way to build detectors. There is a special type of masstool, called the function component, which supports user-defined functions to pre-process the results of measurement tools, e.g. getting the average value of the bandwidth in the last 5 minutes. A function component can subscribe to masstools and process their raw data as input parameters through interfaces.

To better maintain and update MassWare components, we have proposed a

distributed service module, called the component manager, which accepts component registration and provides the components runtime environments. A registered component can be identified by MassWare through its attribute name and value pairs. The major functions of the component manager include component evaluation, which is used to discover and utilize components not considered when the systems were designed [48], component migration, which is used to migrate required components from peer agents when the components are not available locally [67], and virtual connection, which is used to process high workload tasks in a resource-limited device by connecting to a physical component hosted in a powerful server [68]. However, since these functionalities are not related with the major contribution of this research, which is to improve the reconfiguration efficiency of CARM, their implementation details will not be discussed in the dissertation.

3.2.2 Reconfigurator and System-level Reflection

The MassWare reconfigurator contains multiple actuators and provides interfaces to manipulate the actuators so that the application behaviors can be reconfigured. The actuators are designed as reflective components to support MassWare system-level reflection. Each actuator (see Fig. 3.3a) contains a component chain for processing application data, a type library for browsing the component types, and a meta-interface exemplified by Fig. 3.3b in C#. The meta-interface provides the access to its underlying meta-information and internal states (reification), such as the structure of component connections, the actuator status (active/inactive), etc. By accessing the meta-interface, the reconfigurator can change the actuator's meta-information that leads to a change of the actuator implementation (absorption), including the structure modification of component connections and component parameter modifications.



(a) Actuator architecture

```
interface IMetaActuator
{
  componentList get_components();
  connectionList get_connections();
  bool set_components(componentList compList);
  bool set_connections(connectionList connList);

  bool add_component(CMarchletObj marchlet);
  bool remove_component(string marchlet);

  bool connect_all_components();
  bool disconnect_all_components();

  bool connect_components(string senderObj,  string senderInterf,
    string receiverObj,  string receiverInterf);

  bool disconnect_components(string senderObj,  string senderInterf,
    string receiverObj,  string receiverInterf);

  Object get_component_parameter(string comp,  string param);
  bool set_component_parameter(string comp, string param, Object value);

  EnumActuatorStatus get_active_status();
  bool activate();
  bool deactivate();
  ......
}
```

(b) Actuator meta-interface

Figure 3.3: MassWare actuator architecture and meta-interface.

3.3 Awareness Measurement Layer

To support adaptation, DRE systems need to be aware of their running contexts. In this dissertation, awareness is defined as the contextual information of DRE systems. Most existing context-aware middleware frameworks have the functionality to detect a certain context. Our efforts in awareness measurement focus on integrating existing tools that are publicly available and providing mechanisms for application developers to specify and customize these tools in the XML format.

3.3.1 Measurement Tools

Measurement tools in MassWare are implemented as reflective components, which can be declared in the script file, and then loaded and instantiated by MassWare to

measure interested contexts. For example, the real-time QoS monitoring tool and the Mobile Service Testing and Measurement Tool (MOSET) [69] can be declared in the Masstools section of the script file, see Fig. 3.2, to measure application-related QoS. Masstools can also be reconfigured to realize feedback control.

For awareness data that are unavailable from local measurement tools or beyond the middleware knowledge, like the remote information, the measurement is separated into two steps based on an information manager (IM). Awareness providers, like remote measurement tools and applications, send the awareness results to the IM. Masstools then retrieve the data from the IM through pull or push methods. By pulling, Masstools explicitly query awareness data. By pushing, the IM pushes data to subscribed masstools when pre-defined conditions are satisfied.

3.3.2 Context-awareness Categorization

MassWare categorizes the context awareness data in five categories listed in Table 3.1. Among these five categories, network awareness has continuously stimulated the interest in research and industry communities to provide reliable network-awareness measurement tools. Device awareness data, such as the CPU power, display size, memory capacity, display refresh rate, and battery consumption, may be measured through system APIs. User awareness can be collected in an explicit or implicit technique. In an explicit approach, users can specify their preferences through graphical user interfaces. In an implicit approach, measurement tools identify users' preferences by using machine learning agents. Physical sensors measure awareness of the environment.

Table 3.1: Categories of MassWare context-awareness

Network-Awareness	Network characteristics and its measurements
Device-Awareness	Capacity measurements of a particular device
Application-Awareness	Internal states of an application or application required QoS
User-Awareness	User specified preferences for the quality of the service
Environment-Awareness	Environmental measurements by wireless sensor networks

3.4 Awareness Management Layer

The awareness manager in the management layer aims to organize and evaluate the contexts measured from the awareness measurement layer. In DRE systems, data from multiple awareness categories may be needed for evaluating contexts. For example, a DRE system involving video transmissions may rely on the information about both local hardware resource and network bandwidth to select a proper compression strategy. The first difficulty of managing awareness is that the communication network among masstools cannot be fixed in advance since it is impossible to specify the masstools that are used by applications at middleware design-time. Fortunately, this difficulty can be solved by the component-level reflection introduced in Section 3.2. With the reflection model, users only need to specify the interfaces and parameters of masstools in a script file and the awareness manager will set up the communication network at run-time based on the subscribe/notification model.

The second difficulty is that the awareness manager should get sufficient information for accurate adaptation with as few messages as possible to fit the limited resource of DRE systems. To solve this difficulty, a binary tree based hierarchical

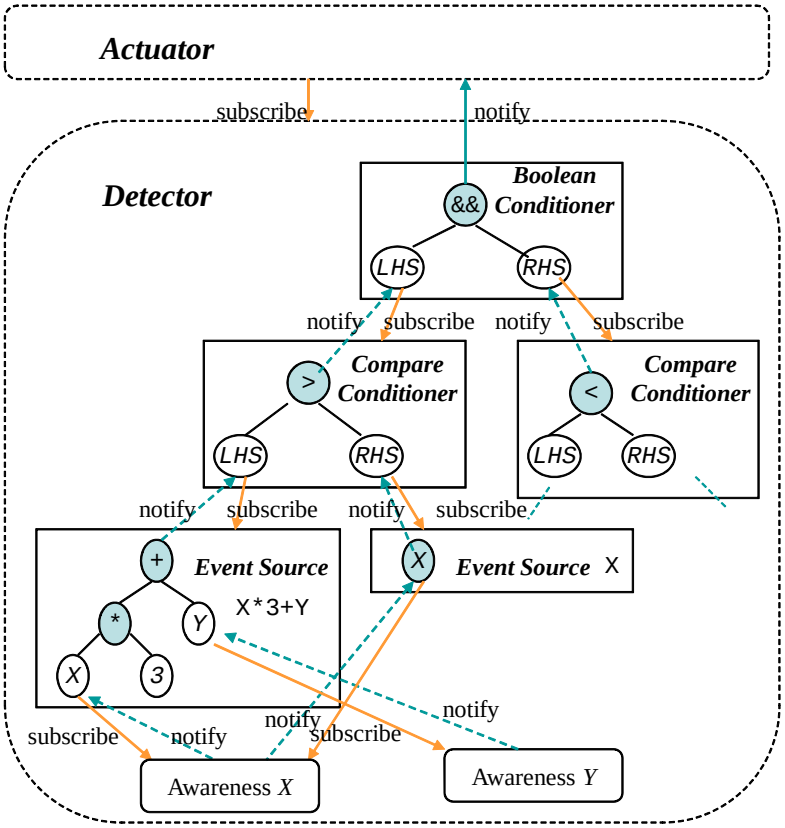


Figure 3.4: The event notification model.

event notification model (see Fig. 3.4) is proposed for conditional subscriptions. This allows context events to be organized and integrated in a tree structure to construct a detector that only monitors and evaluates required contexts and triggers reconfigurations at runtime when its conditions are satisfied.

Each node in the event tree contains a *conditioner*, a left hand side (*LHS*), and a right hand side (*RHS*). There are two types of conditioners: the *compare conditioner* and the *Boolean conditioner* perform comparison and Boolean operations on the LHS and the RHS. The LHS and the RHS can subscribe to the conditioner of a lower-layer event node or an event source. The event source can be a constant value, single context awareness, or an awareness expression. The expression is also

built on a binary tree structure, in which each node has an operator, a LHS, and a RHS. Therefore, all the contexts are organized in a hierarchical way to form a detector. An upper-layer event node or an actuator can subscribe to a lower-layer node as a listener, and only be notified when the conditions of the lower-layer node are satisfied. This structure minimizes the message exchanges in complex detectors.

To improve the efficiency of detectors, the hierarchical event tree is constructed based on the Modified Directed Acyclic Graph (MDAG). That is, before creating a new event node, it checks whether an identical node or an inverse node already exists. Event node a is defined as the inverse node of b if a and b have the same event source and comparison value, but inverse comparison operators. For example, the inverse event of $\min(AVI\_CPU, 10) < 1.0$ is $\min(AVI\_CPU, 10) \geq 1.0$.

To use the event model to identify interested contexts, DRE system developers or end users declare corresponding detectors in a script file. The example shown in Fig. 3.5a means when the average bandwidth during the last 5 seconds is greater than $10Mbps$ and less than $20Mbps$, the detector notifies its subscribed actuators. To facilitate the configuration of the detector script, the MassWare Generator, a tool with Graphic User Interface (GUI), has been developed to transfer a detector defined in the advanced language (Fig. 3.5b) to a XML script (Fig. 3.5a) according to the operator mapping (Fig. 3.5c). More details about the MassWare Generator will be discussed in the Section 3.5.

3.5 Adaptation Decision Layer

The adaptation decision layer contains a decision engine and a script parser. The decision engine takes the script file as input, creates the awareness manager in the awareness management layer and the reconfigurator in the adaptation execution

```
<detector>
 <event>
  <otype> And </otype>
  <lhs>
   <event>
    <otype> GT </otype>
    <lhs>
     <expr> Ave(AVI_BW, 5) </expr>
    </lhs>
    <rhs>
     <expr> 10 </expr>
    </rhs>
   </event>
  </lhs>
  <rhs>
   <event>
    <otype> LT </otype>
    <lhs>
     <expr> Ave(AVI_BW, 5) </expr>
    </lhs>
    <rhs>
     <expr> 20 </expr>
    </rhs>
   </event>
  </rhs>
 </event>
<detector>
```

(a) The detector declaration

Script Operator	Development Operator
GT	>
GE	>=
LT	<
LE	<=
NE	<>
EQ	==
And	&&
Or	||

(b) The mapping table

```
Ave(AVI_BW, 5)>10 && Ave(AVI_BW, 5)<20
```

(c) The detector usage in user development tool

Figure 3.5: A detector example.

layer, and subscribes the actuators in the reconfigurator to the detectors in the awareness manager according to the adaptation policies. This allows the actuator to be triggered by changing contexts for reconfiguration according to user-defined policies.

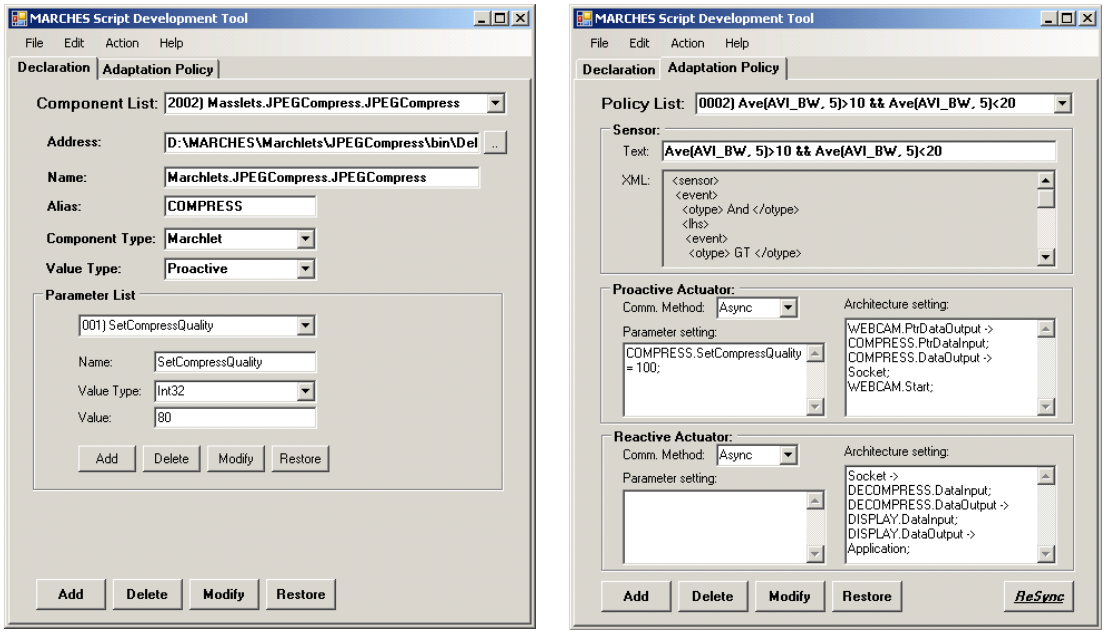
The script parser parses the application script file, which customizes the application configuration and adaptation policies based on a declarative language in the XML format. In particular, the script file can be divided into a declaration part and an adaptation-rule part (as shown in Fig. 3.6). The declaration part declares all components (masslets and masstools as shown in Fig. 3.2) used in

```
<Marchlets>  ...  </Marchlets>
<MarchTools>  ... </MarchTools>

<Rules>
 <rule>
  <detector>   ...  <detector>

  <Actuator type="proactive" sync="Async">
   <SetParam>
    COMPRESS.CompressQuality = 70;
   </SetParam>
   <SetArch>
    GRAB.PtrOutput -> COMPRESS.PtrInput;
    COMPRESS.StreamOutput -> SEND;
    Grab.Start;
   </SetArch>
  </Actuator>

  <Actuator type="reactive" sync="Async">
   <SetArch>
    RECEIVE -> DECOMPRESS.StreamInput;
    DECOMPRESS.StreamOutput -> DISPLAY.Input;
   </SetArch>
  </Actuator>
 </rule>
 ...
</Rules>
```

Figure 3.6: An XML script file example.

a local program and the middleware agent. According to the declaration, MassWare loads and instantiates the components through the reflection model. The adaptation-rule part contains adaptation policies and each policy can be further separated into a detector, a proactive actuator, and an optional reactive actuator. A detector section can be parsed by the event interpreter to build a detector (as shown in Fig. 3.5) that monitors contexts and accepts the subscription of the proactive actuator declared in the proactive actuator section. The proactive actuator contains the system architecture information that is used to update the

actuator internal states by the reconfigurator when it performs reconfiguration actions. Therefore, the system behaviors dynamically adapt to context changes through the system-level and component-level reflection (respectively, architecture reconfiguration and parameter tuning). The reactive actuator section describes the meta-information of an actuator in peer agents that processes the received data from the proactive actuator, so that the behaviors of the proactive and reactive actuators can be synchronized in distributed systems. The script example in Fig. 3.6 shows that the proactive actuator in the sender agent of a video transmission application contains three components: GRAB, COMPRESS, and SEND, which are connected in a sequence. The reactive actuator described in the same policy contains the meta-information of three components as well: RECEIVE, DECOMPRESS, and DISPLAY. The receiver agent constructs the reactive actuator based on the meta-information received through the synchronization process.



(a) Component declaration (b) Adaptation policies

Figure 3.7: The MassWare script file development tool.

The MassWare Generator can facilitate users' generation of script files. As shown in Fig. 3.7, the GUI tool enables users to manipulate both the component and policy configuration and runtime reconfiguration interactively. Furthermore, the tool supports the advanced language for describing event detectors and the re-sync function that can re-synchronize the local agent with peer agents when adaptation policies are modified at run-time.

3.6 Adaptation Execution Layer

The reconfiguration process of DRE systems consists of two steps: local behavior changes triggered by context changes and distributed behavior synchronization to synchronize local behaviors with the changed behaviors of other programs.

3.6.1 Local Behavior Reconfiguration

For traditional reflective middleware, there is only one component chain (or functional path) in each program. The reconfiguration process is to modify the chain structure. For the video transmission example, the original chain of the sender agent contains two components: GRAB and SEND, as shown in Fig. 1.2a. If the chain is reconfigured to contain three components: GRAB, COMPRESS, and SEND for adaptation, the reconfiguration process of the sender agent has the following steps:

- The sender agent stops its application workflow and stores component states into parameter lists;
- The sender agent clears buffered data that are not processed or transmitted to distributed peer agents;

- The sender agent disconnects the GRAB and SEND components and reconnects the GRAB, COMPRESS, and SEND components in sequence;
- The sender agent communicates with peer collaborative agents to synchronize the modified structure with them;
- Peer agents take the same steps: stop current workflow, clear buffered data received from the sender agent for old structure, and adjust component chains by disconnecting RECEIVE and DISPLAY components and reconnecting RECEIVE, DECOMPRESS, and DISPLAY components;
- The sender agent restores the states of the new component chain and restarts the application workflow.

The above reconfiguration process is synchronous and repetitive for each reconfiguration process. It is inefficient because the sender agent has to be suspended until all peer agents finish their corresponding reconfiguration, and all buffered data for previous structure are cleared.

By contrast, MassWare supports multiple component chains as shown in Fig. 1.2b. Each component chain is located in an actuator that is subscribed to an event detector (see Fig. 3.8). When contexts change and trigger a new detector, the detector will notify the decision engine for the reconfiguration by switching active and inactive actuators. There is one and only one active actuator that processes application data. For the above example, there are two chains in the sender agent: the active chain i contains two components: GRAB and SEND and the inactive chain j contains three components: GRAB, COMPRESS, and SEND. The reconfiguration of the sender agent has the following steps:

- The sender agent deactivates the current active actuator that contains chain i by suspending its workflow, storing run-time states, and disconnecting its

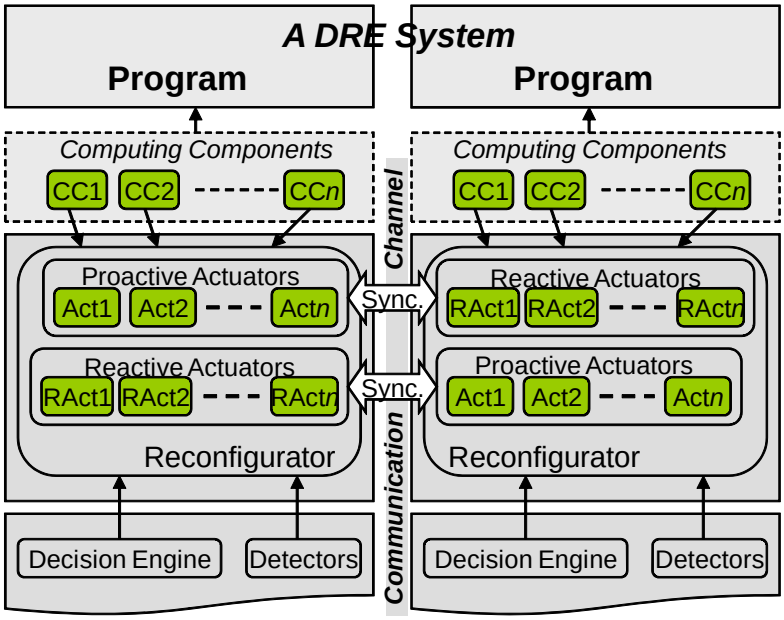


Figure 3.8: The MassWare reconfigurator.

components;

- It activates the target actuator containing chain j by connecting its components, restoring states, and resuming its workflow.

To reduce resource consumption, an actuator only maintains a chain of references, which point to masslet instances, and a customized parameter list for each reference to store component runtime states. The proposed reconfiguration process is asynchronous and efficient because it does not require peer agents to synchronously reconfigure their structure and no buffered data need to be cleared. The peer agents only synchronize their architecture on demand when their received data cannot be processed by existing reactive actuators.

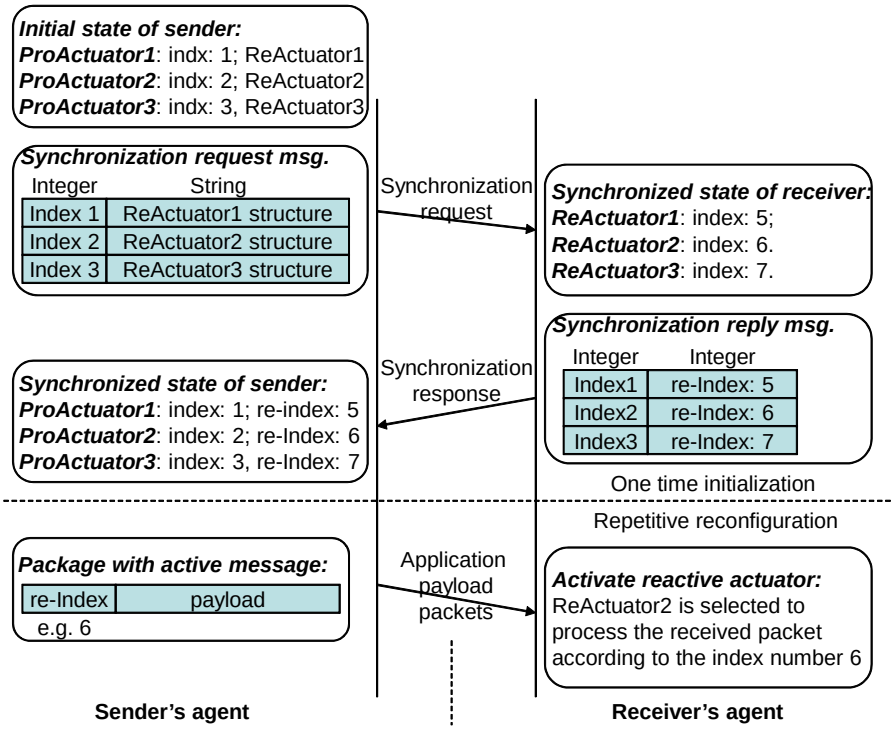


Figure 3.9: The synchronization process.

3.6.2 Distributed Behavior Synchronization

Based on the multiple-chain based architecture, an active-message based synchronization protocol is designed to coordinate reconfigured behaviors in an asynchronous way. The idea of the proposed asynchronous protocol is that each middleware agent constructs the reactive actuators for all peer agents when the middleware starts up, and activates one of them to process received application layer packets according to the active message header attached in the packets. This initialization has the following steps, as shown in Fig. 3.9.

- When the middleware starts up, proactive actuators of each agent are built based on the user-defined script file. Each proactive actuator is also associated with a middleware-assigned unique index and the meta-information of

an optional reactive actuator.

- The middleware agent sends a synchronization request packet to peer agents, which contains the indices of proactive actuators and the meta-information of reactive actuators.

- After receiving the synchronization request packet, the peer agent constructs the reactive actuators according to the meta-information, each of which is associated with a unique index. (The agent will notify the component manager for component migration or virtual connection if the required components can not be identified locally.)

- The receiver or the peer agent replies to the sender with a synchronization response packet that contains a set of index pairs, each of which contains an index of the proactive actuator and the index of the reactive actuator.

- The sender agent replaces the meta-information of each reactive actuator with its corresponding index received from the synchronization response packet.

The above-mentioned initialization is a one-time process. The middleware agent will then append the index of the reactive actuator, corresponding to the current active actuator, to the payload of each data packet as an active message header. The peer agent receiving the data packet activates the reactive actuator indexed by the received index to process the data packet correctly.

The active message based asynchronous synchronization protocol has four advantages: low overhead, short delay, high efficiency, and greater robustness. In general, only the index of the reactive actuator needs to be stored in the active message header for each data packet. By using the asynchronous method, the system does not need to pause in the synchronization process, which dramatically

reduces the reconfiguration time. Furthermore, based on the information in the active message header, a peer agent can always process the received packets by choosing the correct reactive actuator and then no suspension for buffered data is needed, which makes the reconfiguration by our middleware efficient. Moreover, once the reactive actuators are constructed in the system initialization phase, the local agent reconfiguration does not require the availability of other agents and thus it is not affected by the network condition or the capacity of other agents. Therefore the robustness of the application is improved and the communication overhead is reduced.

3.6.3 Correctness of MassWare Synchronization

In MassWare, every received packet needs to be processed correctly by the agent to choose the indexed reactive actuator, and the application workflow should not be affected or interrupted by middleware errors. To prove the correctness of the proposed synchronization protocol, we assume that:

- all errors are detected as errors;
- a synchronization process may fail due to network errors, but it succeeds with at least some probability $p > 0$; and
- both sender agents and peer receiver agents may fail during the data transmission.

We split the proof into two parts: safety and liveness. Safety is defined as the fact that a protocol never produces an incorrect result, which in this case means a received packet can always be properly processed. Liveness is defined as the fact that an algorithm can continue forever to produce results, which in this case means

the capability to continue forever to send new packets at sender agents and accept them at receiver agents.

MassWare uses re-transmission(s) for the synchronization process to handle network errors. If the number of synchronization failures from a sender agent to a peer agent is larger than a threshold, the peer agent is removed from the sender's receiver list. A new agent can request to join the sender's receiver list by sending a *node join* message to initialize the synchronization or leave the sender's receiver list by sending a *node leave* message to remove corresponding indices from the sender. After synchronization, every proactive actuator of the sender agent has an index (active message header) for each reactive actuator of every peer agent. To prove that a receiver agent can always process a received packet correctly, we consider the following three cases.

- The receiver agent can recognize the index in the received active message header of the packet and the index is correct. This case represents the normal situation. The peer agents can invoke the reactive actuator pointed by the index to process the packet payload.

- The receiver agent cannot recognize the index in the received active message header due to errors. This case happens when the receiver agent stops unexpectedly and does not notify the sender to remove it from its receiver list. The receiver can continue to receive packets from the sender after it is reloaded, but all the indices in the packets cannot be recognized as its local reactive actuators have been cleared. In this case, the receiver agent sends a *re-synchronization* message to the packet sender to initialize a synchronization process. Every received data packet can then be processed correctly without deletion.

- For the completeness of discussion we consider the third case that will not

happen: the receiver agent may recognize the index, but the index is incorrect and points to a wrong reactive actuator that cannot process the packet payload correctly. This case could be a concern if one would assume the following situation: both sender agent a and sender agent b communicated with the same receiver agent c and later c stopped unexpectedly while it might still be in the receiver lists of a and b; when c restarted and only synchronized with a, it constructed a reactive actuator for a, which might be used to process packets received from b if the reactive actuator for a shared the same index with the reactive actuator previously constructed for b. However, such assumed situation will not happen due to the way the index is created. In fact, the index of a reactive actuator is generated by the receiver agent based on the IP address of the synchronization requester and the hash value of the actuator meta-information. When both sender agents a and b are located at the same host and two different actuators from a and b have the same hash value, we will add an UID, which is unique for every application with respect to the host, in the active message header to ensure that each reactive actuator has a unique index. Therefore, a recognized index must point to a correct reactive actuator.

Based on the above analysis, we can conclude that a MassWare agent can always process received packets correctly, which proves the safety of MassWare.

Once the synchronization process is completed, a sender agent only needs local information for reconfigurations. It can then continue to process application data forever during its life time. Thus the liveness of MassWare is proved as we have concluded that a peer agent can always process received packets correctly.

3.6.4 Policy Modification at Runtime

The target users of MassWare are DRE system developers. However, end-user requirements must also be considered in customizing the middleware behavior as it is difficult to predict all desired adaptation policies in advance. The MassWare Generator provides end-users with the GUI (shown in Fig. 3.9), through which their preferences can be captured by MassWare to modify the policies at runtime. MassWare supports a re-synchronization method for the runtime policy modification, in which the agent suspends its operations, clears data buffer, re-synchronizes the modified policies with peer agents, and resumes its operations.

3.7 MassWare Application Development

MassWare offers an effective approach to build adaptive DRE systems. To develop a new system or migrate an existing system from another middleware framework to MassWare, developers need to provide required components, an XML-based script file, and an optional GUI program for UI systems. The first step is to create MassWare components or migrate existing components to the MassWare platform. MassWare component model currently supports COM components and .NET assemblies. To support other types of components, special component wrappers need to be developed, which will be part of the component manager functionalities in future work.

The second step is to develop a script file that declares all required components, including functional components (marchlets) and measurement tool components (marchtools), and adaptation rules using the XML language. The details of the script file structure are presented in Section 3.5 and a full example is decomposed into Fig. 3.2, Fig. 3.5, and Fig. 3.6 for component declaration, detector

declaration, and policy declaration separately.

There are two different methods to instantiate and invoke MassWare agents for constructing UI-based and service-based DRE applications. An UI application can create and invoke a MassWare instance in their UI program through MassWare interfaces. Some example UI applications using MassWare have been published on our website [70]. A service application without a UI program can be constructed implicitly through the script development tool. After a script file is composed, the application can then be started, paused or stopped through the "Action" menu of the script development tool.

MassWare is reflective middleware and provides a set of reification interfaces to present its internals to applications. The internals include component-chain structure, component states, reconfiguration events, and context information etc., which can be used for application debugging and validation. The applications also have the ability to reconfigure the internals through the absorption interfaces provided by MassWare so that application users can manually manipulate the application behaviors.

Chapter 4

MassWare for Wireless Sensor Networks

4.1 System Architecture of MassWare

MassWare-WSN (also called MassWare in this chapter) is located between the lower hardware/operating-system layer and the upper application layer to monitor contextual information and support application adaptation. It is client/server-based middleware: the sensor nodes run in the client mode and the base station runs in the server mode. To reduce communication overhead, there is one middleware agent in each sensor node. Because of limited resources of sensor nodes, MassWare has to be lightweight and efficient to be implemented in the senor nodes, while flexible and adaptable enough to support generic adaptive WSN applications. For these considerations, MassWare is designed in a layered architecture, as shown in Fig. 4.1.

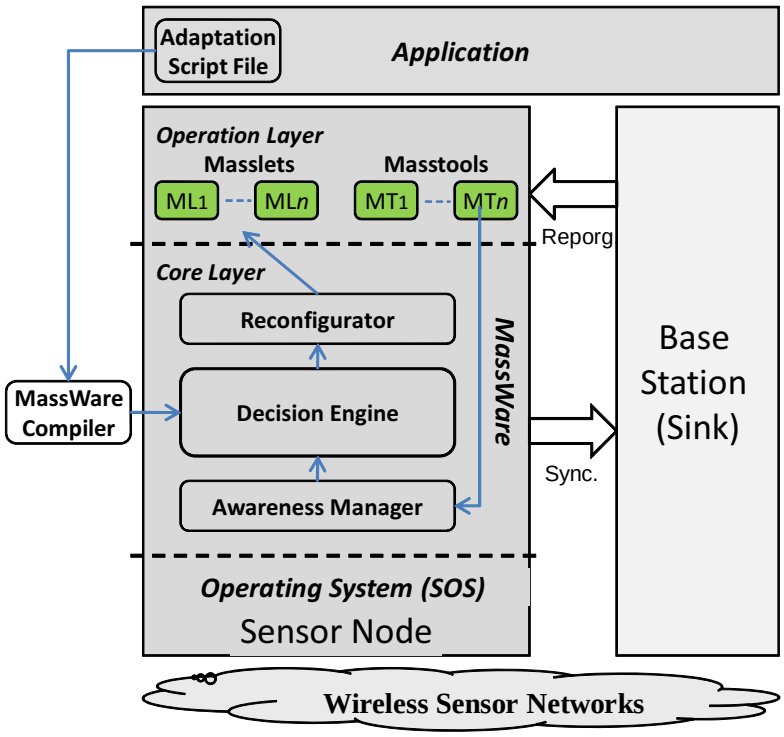


Figure 4.1: System architecture of MassWare-WSN.

A MassWare agent can be separated into a core layer and an operation layer. The operation layer contains all MassWare components that can be shared by different applications. There are two types of components, similar to MassWare-MANET, in the operation layer: functional components (called masslets) for data processing and communication, and measurement-tool components (called masstools) for measuring and evaluating situational contexts.

The core layer is the part that should be pre-installed to every sensor node. It consists of three function modules, which form the MassWare reflective framework for monitoring contexts and reconfiguring applications.

- **The awareness manager** contains a set of detectors that detect context-awareness information about networks, devices, and environments, e.g. node density, remaining battery, etc. A detector refers to a list of masstools provided in the operation layer. In a detector, use of its referred masstools is organized in a hierarchical event notification model. The details of the awareness manager are presented in Section 4.3.

- **The decision engine** is a special MassWare component that is automatically generated based on adaptation policies. An adaption policy is defined by application developers or end users in a script file in XML syntax. The decision engine can create detectors in the *awareness manager* and actuators in the *reconfigurator* and subscribe the actuators to the detectors. Therefore the actuators can be triggered by detectors to reconfigure application behavior according to the policies. More information about the decision engine is presented in Section 4.4.

- **The reconfigurator** contains a set of actuators that can perform reconfiguration actions. An actuator refers to a list of masslets that form a functional

path or component chain to process application-layer sensor data. There is one and only one actuator active at any time, and only the active actuator processes the data. Various actuators can be activated and deactivated by detectors to adapt to the context. The details of the reconfigurator are presented in Section 4.5.

To develop a sensor application based on MassWare, developers need to provide an XML-based adaptation script file besides required components (masslets and masstools). The script file uses exactly the same format with a MassWare-MANET script file that describes the required components and adaptation policies. Each policy specifies the meta-information of a detector, a proactive actuator, and a reactive actuator. The meta-information of the detector and the proactive actuator is used to construct a detector and an actuator. The meta-information of the reactive actuator will be sent to the base station for synchronization, so that the base station can correctly process the received data when the behavior of a remote sensor node is reconfigured. After the adaptation script file is developed, it will be compiled into a decision engine by the MassWare compiler. The MassWare compiler generates SOS-supported binary modules.

In summary, a MassWare application consists of three types of components: masslets, masstools, and the decision engine. When the decision engine component is loaded, it builds detectors and actuators based on a user-provided script file, subscribes the actuators to corresponding detectors that are described in the same policy, and sends the meta-information of reactive actuators to the base station for synchronization. When contexts trigger an event a detector monitors, the subscribed actuator will be activated to perform the reconfiguration action.

4.2 MassWare Reflective Model

MassWare provides both component-level and system-level reflections. The component-level reflection deals with the content and behavior of a given component via an interface metamodel. The interface metamodel provides discovery of and access to the set of provided and required interfaces of the component. Based on the component-level reflection, MassWare supports generic software components for sensor nodes in a cost-efficient manner. It is easily upgradeable to incorporate new components in its operation layer and meet the rapid progress of new algorithms and standards for WSN applications. The system-level reflection deals with the structure and graph of the component connections via an architecture metamodel. The architecture metamodel provides discovery and operation to the current active actuator. The system-level reflection enables MassWare to expose the internal states of sensor nodes to end users and allows them to change the adaptation policies at runtime by injecting a new decision engine component to the network.

4.2.1 MassWare Components

A MassWare component is a function-independent reflective element that provides some interfaces by which a component can communicate with others, retrieve its internal states (called reification), and changes the states at runtime (called absorption). By using the interfaces and the component-level reflection, MassWare can dynamically connect masslets and masstools to build actuators and detectors (called late binding) and change application architectures at runtime.

A MassWare component can be viewed as an SOS module with a set of MassWare interfaces. To develop a MassWare component, developers need to specify these interfaces in the source code. Two new keywords - *massware* and *interface* -

```
massware interface MSG_SET_FREQ
{
  int ID = MOD_MSG_START+102
  InterfType iType = Parameter
  ActionType aType = Set
  ValueType vType = uint8
}

massware interface MSG_OUTPUT
{
  int ID = MOD_MSG_START+116
  InterfType iType = Communication
  ActionType aType = Output
  MsgType mType = MassWareOutputMsg
}
```

(a) Component interfaces

```
<component cid="2001">
  <name> BLINK_RED_COMP </name>
  <comId> APP_MOD_MIN_PID + 44 </comId>
  <alias> LSWT </alias>
  <interface type="Parameter">
    <name> MSG_SET_FREQ </name>
    <interfId> MOD_MSG_START + 102 </interfId>
    <actionType> Set </actionType>
    <valueType> Integer </valueType>
  </interface>
  <interface type="Communication">
    <name> MSG_OUTPUT </name>
    <interfId> MOD_MSG_START + 116 </interfId>
    <actionType> Output </actionType>
    <msgType> MassWareOutputMsg </msgType>
  </interface>
  ...
</component>
```

(b) Component metafile

Figure 4.2: A MassWare-WSN component example.

have been created to define MassWare interfaces. An interface describes the interface name, the interface ID, the interface type, and the value type for a parameter interface or the message type for a communication interface. Fig. 4.2a presents the interfaces of the BLINK_RED_COMP component that controls the red LED of a sensor node. MassWare interfaces have two functions. First, the interfaces are used to reconfigure component behavior and connections at runtime. For example, the blink frequency of the BLINK_RED_COMP component can be changed at runtime through the MSG_SET_FREQ parameter interface and the component connections to other components can be reconfigured through the MSG_OUTPUT communication interface. Second, a MassWare component with interfaces will be compiled by a designed *Component Compiler* to create an SOS-supported binary module and a human-readable file (meta-file), which contains the component meta-information. For example, through the meta-file of the BLINK_RED_COMP as shown in Fig. 4.2b, we can get the name, ID, alias, and the parameter interfaces and communication interfaces of the component. With the meta-file, generic sensor services can be implemented as standard MassWare components, which can

be easily shared and reused by WSN applications. The component ID (comID) is a hash value calculated by the compiler based on the component checksum. Thus, different components can be identified by the decision engine according to their IDs.

```
<DecisionEngine xmlns:xsi=...>
<MassTools>
 <component cid="1001"> ... </component>
 ...
</MassTools>

<Masslets>
 <component cid="2001">
  <name> BLINK_RED_COMP </name>
  <comId> APP_MOD_MIN_PID + 44 </comId>
  <alias> BLINKR </alias>
  <interface type="Parameter">
   <name> MSG_SET_FREQ </name>
   <interfId>MOD_MSG_START + 102</interfId>
   <value> 3072 </value>
  </interface>
  <interface type="Communication">
   <name> MSG_START </name>
   <interfId>MOD_MSG_START + 100</interfId>
   <actiontype> Start </actiontype>
  </interface>
  <interface type="Communication">
   <name> MSG_OUTPUT </name>
   <interfId>MOD_MSG_START + 116</interfId>
   <actiontype> Output </actiontype>
  </interface>
 </component>
 ...
</Masslets>

<AdaptationPolicies>
 <policy pid="001">
  ...
 </policy>
 ...
</AdaptationPolicies>

</DecisionEngine>
```

```
<policy pid="001">
  <detector did="001">
   <event>
    <otype> AND </otype>
    <lhs> <event>
     <otype> GT </otype>
     <lhs><expr>NEIGHBER.NUMBER</expr></lhs>
     <rhs><expr> 1 </expr></rhs>
    </event> </lhs>
    <rhs> <event>
     <otype> LT </otype>
     <lhs><expr>POWER.REMAINING</expr></lhs>
     <rhs><expr> 5 </expr></rhs>
    </event> </rhs>
   </event>
  </detector>

  <Actuator aid="001">
   <atype> ProActive </atype>
   <SetParam>
    BLINKR.MSG_SET_FREQ = 5*1024;
   </SetParam>
   <SetArch>
    BLINKR.MSG_OUTPUT -> BLINKG.MSG_INPUT;
    BLINKG.MSG_OUTPUT -> BLINKY.MSG_INPUT;
    BLINKY.MSG_OUTPUT -> BASE_STATION
    BLINKR.MSG_START
   </SetArch>
  </Actuator>
  <Actuator aid="101">
   <atype> ReActive </atype>
   <SetArch>
    PACKET_RECEIVE -> BLINKY.MSG_INPUT;
    BLINKY.MSG_OUTPUT -> BLINKG.MSG_INPUT;
    BLINKG.MSG_OUTPUT -> BLINKR.MSG_INPUT;
   </SetArch>
  </Actuator>
 </policy>
```

Figure 4.3: A MassWare-WSN script file example.

To incorporate a new reflective component in MassWare, users need to describe the component meta-information, which is generated by the Component Compiler, in the XML Script file, as shown in the component section of Fig. 4.3. A MassWare component can be identified based on its component ID and specified by setting

its parameters, which can also be reconfigured at runtime according to adaptation policies. Components can be connected through input and output interfaces that support compatible message types to construct an application. The connections can also be reconfigured to change the application architecture at runtime through the late binding.

There are two types of MassWare components: reconfigurable functional components (*masslets*) and measurement tool components (*masstools*). Masslets are the basic functional units to construct programs in sensor nodes. MassWare supports the publish/subscribe model for communication - the output interface of a masslet can be subscribed by message-compatible input interfaces of other masslets and publish messages to them.

Masstools measure and predict real-time context awareness, like node density, remaining power, and connectivity etc., so that sensor applications can adapt to changing contexts. They are implemented as reflective components to facilitate the reuse and extension of existing measurement tools. Masstools are the lowest event sources to build hierarchical event detectors.

4.2.2 MassWare Actuator

MassWare actuators are located in the decision engine component and constructed based on the actuator section of each adaptation policy, as shown in the actuator section of Fig. 4.3. It consists of a set of masslets, a parameter lists, and a component graph. Every actuator is subscribed to a detector and will be activated when the conditions of the detector are satisfied by changing contexts. MassWare indexes all its actuators and synchronize with the base station in its initialization phase by sending the indices and the meta-information of the actuators. After synchronization, each packet processed by a local actuator takes the actuator index

as an active message header. Since only active actuator can process application data, users then know the current active actuator and its architecture of component connections base on the received active message header (called reification). The architecture can be changed for adaptation by actuators through the configuration of component connections (called absorption). The active message header is also used to identify a correct actuator in the base station to process the received data since the packets processed by different actuators need to be treated differently. For example, packets processed by a special compression algorithm in sensor nodes need to be decompressed by a corresponding decompression algorithm in the base station. By examining the sensor node status, users can also change the adaptation policies at runtime by injecting a new decision engine component to the network.

4.3 MassWare Awareness Management

To support adaptation, MassWare needs to be aware of its running contexts. In this dissertation, awareness is defined as the contextual information of WSN applications [53]. By using masstools, developers can integrate existing measurement tools that are publicly available and customize these tools in the script file. The awareness management aims to organize and evaluate the contexts measured from masstools by creating detectors. The difficulty is to measure information with little overhead and not stretch the limited resources of sensor nodes. In complex WSN applications, contexts from multiple masstools may be needed for context evaluations. For example, a sensor node may rely on the information of both node density and its remaining power to decide its sampling rate. To solve this difficulty, we have designed two approaches. The first approach uses a flat structure, in which all masstools are directly subscribed by the decision engine. The decision engine gets notified when any measurement result of the masstools is changed. The

detectors described in the XML script file are translated into expressions in the decision engine, which takes responsibility for evaluating the adaptation rules. In this way, the number of required messages is minimized, as all measurement results are directly sent to the decision engine. On the other hand, the decision engine is frequently interrupted and needs to perform an evaluation for every notification. Detectors described in the script file are also limited to the operators supported by the C language because in SOS they are translated into C language expressions.

The second approach utilizes a binary-tree based hierarchical structure for conditional subscriptions [53]. Context events are organized in a tree structure to construct detectors that monitor only required contexts with minimal evaluations, and actuators can be subscribed directly to the detectors. In this way, the decision engine does not need to evaluate rules and is only triggered to activate an actuator when an adaptation rule is satisfied. Because every node in the event tree is a masstool, this approach supports user-defined operations implemented in specified components.

Each node in the event tree is a MassWare component that contains a *conditioner*, a left hand side (*LHS*), and a right hand side (*RHS*) (see Fig. 3.4). There are two types of conditioners: the *compare conditioner* and the *Boolean conditioner* perform comparison and Boolean operations on the LHS and the RHS. The LHS and the RHS can subscribe to the conditioner of a lower-layer event node or an event source. The event source can be a constant value, single context awareness, or an awareness expression. The expression is also built on a binary tree structure, in which each node has an operator, a LHS, and a RHS. Therefore, all the contexts are organized in a hierarchical way to form a detector. An upper-layer event node or an actuator can subscribe to a lower-layer node as a listener, and only be notified when the conditions of the lower-layer node are satisfied. The detector in Fig. 3.4 means $((X \times 3 + Y) > X)$&&.... This structure minimizes the message

exchanges in detectors.

To use the event model to identify interested contexts, developers or end users declare corresponding detectors in a script file. The example shown in the detector section of Fig. 4.3 means when the number of neighbors is larger than 1 and remaining energy is less than $5mJ$, the red LED blink rate is set to $5s$. The detector script is compiled by the MassWare compiler to create extra masstools for intermediate operational nodes in the event tree. For the example in Fig. 4.3, there are three operational masstools that are created for $neighbor > 1$, $power < 5$, and $LHS\&\&RHS$. To improve the efficiency of sensors, the event tree is constructed based on a Directed Acyclic Graph (DAG). That is, before creating a new event node, it checks whether an identical node already exists. The tree structure is compiled into the decision engine, which will connect all the masstools to build detectors when it is loaded.

4.4 MassWare Compiler and Decision Engine

A major advantage of MassWare is that it facilitates the development of adaptive WSN applications, which are comprised of masslets, masstools, and the decision engine. Because masslets and masstools can be shared components from existing or third-party applications based on MassWare component-level reflection, users only need to provide an XML-based script file that is compiled by the MassWare compiler to generate the decision engine component and probably some operational masstools. Therefore, developers and users are not burdened by the details of coding sensor programs and complex adaptation logic using NesC or C languages. The script file is also easily customized by users to satisfy their preference and application specifications.

MassWare supports the same script files as the ones in MassWare-Manet. A

script file can be divided into a declaration part and an adaptation-rule part. The declaration part declares all masslets and masstools - components used in a local sensor program and the decision engine. Based on the declaration, the decision engine can identify the masslets and masstools located in a sensor node and initialize the components with provided parameters. The adaptation-rule part contains adaptation policies and each policy can be further separated into three sections: a detector, a proactive actuator, and an optional reactive actuator. The detector section is parsed by the compiler to create detectors: the result is either a context evaluation expression in the flat structure or operational masstools in the hierarchical structure. The proactive actuator section describes the meta-information of local actuators, which adapt application behavior to the changing context through the system-level and component-level reflections (respectively, architecture reconfiguration and parameter tuning). The reactive actuator section describes the meta-information of an actuator that can process the data from the proactive actuator of the same policy. The meta-information will be sent to the base station in the initialization phase of the decision engine, and the reactive actuators are constructed in the base station according to the meta-information to process received data from sensor nodes, so that the behaviors of the sensor nodes and the base station can be synchronized in distributed WSN applications. For example, the first policy shown in Fig. 4.3 means when a sensor node has one or more neighbors and its remaining energy is more than 5mj, the three components, BlinkR, BlinkG, and BlinkY, are connected in a chain and the BlinkR starts to blink the red LED with a frequency of 5s when the application is booted, BlinkG and BlinkY blink green and yellow LEDs each time they receive a message from their input components. The base station then connects BlinkY, BlinkG, and BlinkR and blinks the LEDs in a sequence when it receives a packet from the sensor node.

The decision engine, created by the MassWare compiler based on the script file,

is the major component of MassWare and is the last component loaded into sensor nodes. Once the decision engine is loaded, it executes the following setup steps:

- The decision engine sends a *synchronization request* packet, including the meta-information and a unique index for every reactive actuator, to the base station for synchronization in the callback function of the MSG_INIT event, which is the first event the decision engine triggers. It then sets a timer to wait for the base station to receive and process the packet.

- When the timeout event of the timer is triggered, the decision engine connects all masstools to construct detectors, subscribes its actuators to the detectors, and initializes all the components using their parameter interfaces. It then starts the components that have the "START" interface to process application data.

- When the decision engine receives a message from a masstool (for flat structure) or a detector (for tree structure), it will either evaluate all adaptation policies to activate the actuator with satisfied conditions or directly activate the actuator subscribed to the detector. The new active actuator reconfigures masslet connections and/or masslet status according to the policy

- If the base station receives a packet with an unknown active message header after synchronization, it sends back a re-synchronization message to the packet sender, which will resend the synchronization request packet to the base station.

4.5 MassWare Efficient Reconfiguration

The reconfiguration process of a MassWare-WSN application is similar to that of a MassWare-MANET application, which consists of two steps: the local behavior change of sensor nodes that is triggered actively by the changing context and the distributed behavior synchronization of the base station that is triggered reactively to process received packets. MassWare-WSN also adopts the asynchronous method introduced in [53] to improve reconfiguration efficiency.

4.5.1 Local Behavior Reconfiguration

MassWare supports multiple component chains, each of which is located in an actuator. There is one and only one active actuator that processes application data. For the above example, there are two chains in the sender agent: the active chain i contains two components: SAMPLE and SEND and the inactive chain j contains three components: SAMPLE, COMPRESS, and SEND. The reconfiguration process of the sender agent then has the following steps (see Fig. 1.2b):

- The sender agent deactivates the current active actuator that contains chain i by suspending its workflow, storing run-time states, and disconnecting its components;
- It activates the target actuator containing chain j by connecting its components, restoring states, and resuming its workflow.

The proposed reconfiguration process is asynchronous and efficient since it does not require the base station to synchronously reconfigure its structure and no buffered data need to be cleared. The base station chooses different reactive actuators to process data packets based on their active message header. The details of the synchronization protocol will be detailed in the next section.

4.5.2 Distributed Behavior Synchronization

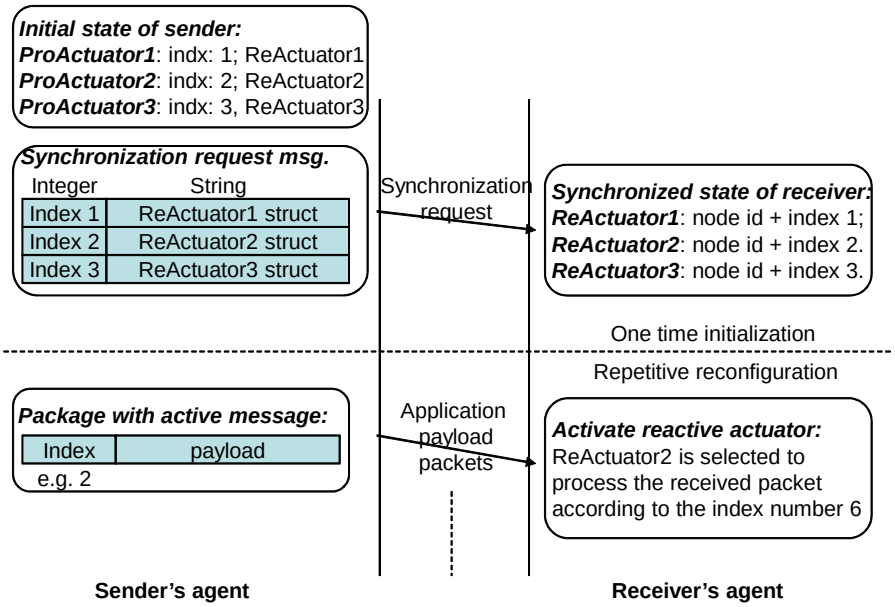


Figure 4.4: The synchronization process in MassWare-WSN.

Based on the multiple-chain architecture, an active-message based synchronization protocol is designed to coordinate reconfigured behaviors in an asynchronous way. The idea of the proposed asynchronous protocol is that the base station constructs the reactive actuators for all sensor nodes when the nodes starts up, and selects one of them to process received packets according to the active message header attached in the packets. Since communication is energy-consuming in WSNs, the MassWare-WSN synchronization process is even more efficient than that in MassWare-MANET and reduces half hand-shaking communication process by using sender assigned active message header. The initialization has the following steps, as shown in Fig. 4.4.

- When the decision engine starts up, actuators are built based on the meta-information of the proactive actuator section in the script file. Each proactive

actuator is also associated with a unique index calculated based on the actuator hash value.

- The middleware agent sends a *synchronization request* packet to the base station, which contains the index of the proactive actuators and the meta-information of corresponding reactive actuator for every policy.
- After receiving the *synchronization request* packet, the base station constructs the reactive actuators according to the meta-information of the packet, each of which is associated with the IP address of the packet sender and the received index.

The above-mentioned initialization is a one-time process. The decision engine will then append the index of the current active actuator as active message header to the payload of each data packet. The base station activates the reactive actuator indexed by the received index to process the data packet correctly. The initialization phase is one-way communication, which is energy-efficient. However, a sensor node cannot get any feedback if the synchronization fails. In the situation, the base station sends a "*re-synchronization*" message to the packet sender (node) for re-synchronization if it cannot identify the active message header of the received packets.

The active message based asynchronous synchronization protocol has four advantages: low overhead, short delay, high efficiency, and better robustness. In general, only the index of the reactive actuator needs to be stored in the active message header for each data packet. By using the asynchronous method, the system does not need to pause in the synchronization process, which dramatically reduces the reconfiguration time. Furthermore, based on the information in the active message header, the base station can always process the received packets by choosing the correct reactive actuator, and then no suspension for buffered data

is needed, which makes the reconfiguration by MassWare efficient. Moreover, once the reactive actuators are constructed in the system initialization phase, the local node reconfiguration does not require the availability of the base station, and thus it is not affected by the network condition or the capacity of other agents. Therefore the application robustness is improved and the communication overhead is reduced.

4.6 MassWare Application Development

MassWare offers an effective approach to build adaptive WSN applications. To develop a new WSN application or migrate an existing application to MassWare, developers need to provide required components and an XML-based script file. The first step is to create MassWare components (masslets and masstools) or use existing components shared by other MassWare applications. MassWare components are developed using the C language, like SOS modules, with MassWare interfaces. The developed source files are compiled by the component compiler to create SOS-supported modules and associated meta-files.

The second step is to develop a script file that declares all required components, including functional components (marchlets) and measurement tool components (marchtools), and adaptation rules using the XML language (The details of the script file structure are presented in Section 4.3 and a full example is shown in Fig. 4.3). The script file is compiled by the MassWare compiler to create the decision engine component and probably some operational masstools if detectors are built on the hierarchical structure.

The third step is to load all compiled components to sensor nodes with SOS, which provides a method to distribute modules through wireless links. The static SOS core burned into the nodes allocates dynamic memory for components and

boots them up by triggering a "MSG_INIT" event. Masslets and masstools need to be loaded before the decision engine component. After the decision engine is loaded, it will configure the components to start the application.

The target users of MassWare are WSN application developers. However, end-user requirements must also be considered in customizing the application behavior as it is difficult to predict all desired adaptation policies in advance. By examining a received active message header, users can check the current status of each sensor node and easily change the adaptation policies by injecting a new decision engine component created on a modified script file. Based on the MassWare reflection model, other components (masslets and masstools) can also be efficiently re-programmed via wireless links.

Chapter 5

Performance Analysis and Experiments

5.1 MassWare-MANET Evaluation by Analytical Models

In this section, we theoretically analyze the performance of MassWare-MANET in terms of the one-time initialization time in the middleware startup phase and the repetitive reconfiguration time whenever the reconfiguration process is triggered by context changes. To verify the time efficiency of the proposed multi-chain structure and active message oriented synchronization protocol, we compare the reconfiguration time of MassWare with that of MobiPADS and CARISMA. These context-aware reflective middleware frameworks can be fairly compared because:

- All these three frameworks are policy based with predefined adaptation policies specified in a script file.
- They all target stateless applications, which do not require the guarantee of application states or packet delivery sequence in the reconfiguration process.
- They all consider the behavior synchronization problem for distributed applications. However, the synchronization protocols they employ are different. MobiPADS uses a communication channel for synchronization and suspends application operations in the reconfiguration process. CARISMA uses a micro-economic approach to handle the adaptation conflicts between distributed policies. MassWare uses an active-message-oriented asynchronous method for synchronization to solve the behavior inconsistency.

In the comparison, we ignore the component and code migration among different middleware agents, which will impose the same overhead for each framework. Since the theoretical analysis is system and implementation independent, we can then fairly compare their performance [71].

Table 5.1: Parameter notation of reconfiguration time

Notation	Parameter
RTT	The minimum round trip time excluding the transmission delay
t_tcp	TCP socket establishing time
t_pend	Operation suspension time for component deletion
t_init	Initialization time for a component addition
n_add	The number of components to be added in a reconfiguration process
s_chain	The average size of a meta-chain
t_reso	The total local computation time of the conflict resolution algorithm in CARISMA
n_policy	The number of policies in an application
t_react	Reactive actuator construction time in MassWare
t_conn	Connection time of two components
t_rest	Restoration time of a component
n	The number of components in a component chain
B	The average available bandwidth

5.1.1 Analytical Model

To compare the reconfiguration efficiency of MassWare with that of MobiPADS and CRISMA, we use a unified model to formulate the reconfiguration time as the sum of the communication time T_{comm} among distributed middleware agents and the local computation time T_{comp}.

$$T = T_{comm} + T_{comp} \tag{5.1}$$

To simplify of the model and for fair comparison, we ignore the component migration time required by all systems, the transmission delay of control messages, which is much smaller than their propagation delay as the control message size is negligible, and other overhead, like socket buffering time, thread switch time, and internal message exchange time, which may be affected by different operating

systems and programming languages. All the middleware agents use TCP three-way handshakes for each reconfiguration message exchange, which takes 1.5RTT for the connection establishment. Some of the notation are defined in Table 5.1.

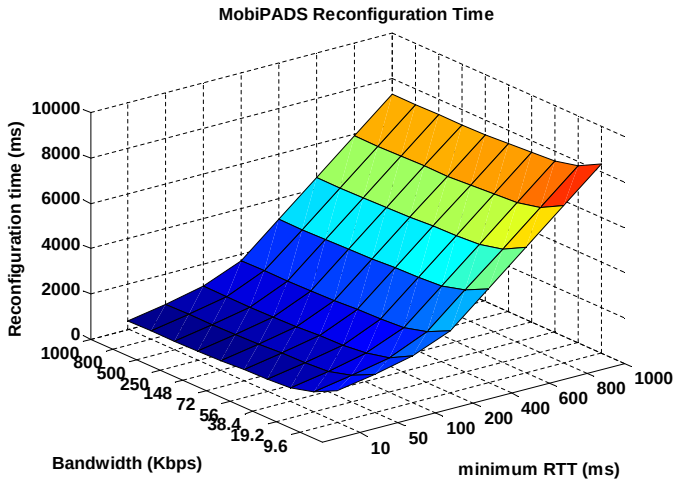


Figure 5.1: MobiPADS reconfiguration time.

In MobiPADS, there is only one component chain, and the reconfiguration process involves three steps: (i) initializing reconfiguration, (ii) deleting components, and (iii) adding components. The reconfiguration time expressed by Eq. (4) in [26] is shown as:

$$T_{MobiPADS} = (\beta + \gamma + \delta)/B + 2kn + 2m + 5.5RTT + C \tag{5.2}$$

where β is the meta-chain size; γ and δ are the component request message size and component size for component migration; $2kn$ is the component initialization time; m is the deletion time; and C is other overhead. we further separate m into 1.5 RTT for message exchange and an operation suspension time [26], and ignore the component migration time, the component initialization time, which is very small (i.e. few milliseconds as shown in Fig. 5.7), and other overhead to follow our model. we then rewrite Eq. 5.2 using the notations shown in Table 5.1 as Eq.

5.3 to express the reconfiguration time of MobiPADS for comparisons with those of MassWare and CARISMA in a unified model.

$$T_{MobiPADS} = T_{comm} + T_{comp} = (7RTT + s\_chain/B) + t\_pend \tag{5.3}$$

t_pend is affected by the number of buffered data and their processing time, and its value may vary for different applications. we set its default value as $300ms$ to match the experiment results shown in Fig. 13 of [26] for numerical evaluations. The meta-chain size s_chain is analyzed in Section 5.2.3 and we set its default value as $10Kbits$ for a chain with 10 components. The reconfiguration time of MobiPADS is depicted in Fig. 5.1.

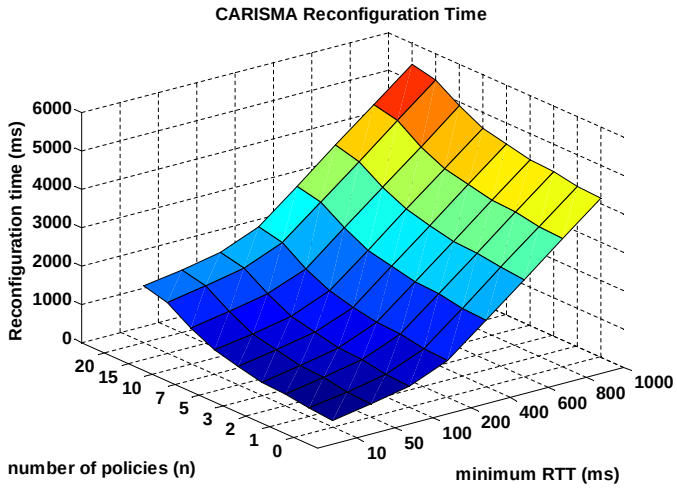


Figure 5.2: CARISMA reconfiguration time.

In CARISMA, the reconfiguration conflict resolution process consists of the following steps: 1) service request, 2) local context evaluation and enabled policy selection, 3) the enabled policy exchange, 4) solution set computation and conflict detection, 5) bidding request and reply, 6) winning policy calculation, and 7) the winning policy broadcast. Steps 1, 3, 5, and 7 involve communications for message exchanges, and steps 2, 4, and 6 involve local computations for conflict resolution,

which is related to the number of policies, contexts, resources, and conflicts. To compare CARISMA with MassWare in terms of reconfiguration time, we use the simplest case of CARISMA with minimum overhead, which is that each policy contains only one context and one resource and there is no conflict. The total reconfiguration time of CARISMA can then be expressed as:

$$T_{CARISMA} = T_{comm} + T_{comp} = 4RTT + t\_reso = 4RTT + f(n\_policy) \qquad (5.4)$$

We use the values of the conflict resolution time t_reso that are directly obtained from the Fig. 15 in [8]. The reconfiguration time of CARISMA is shown in Fig. 5.2.

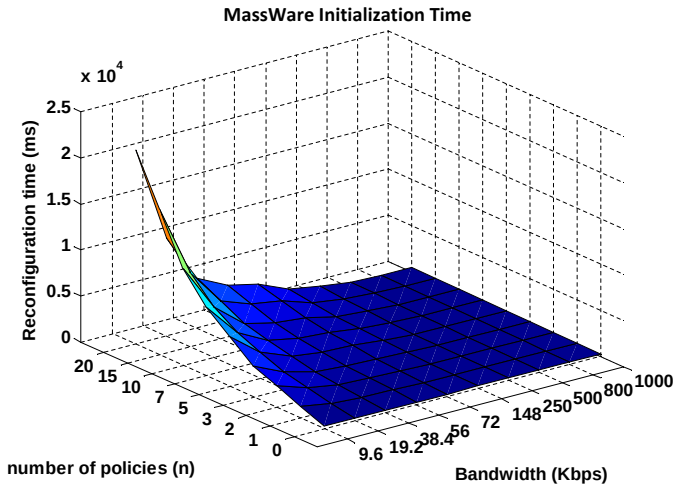


Figure 5.3: MassWare initialization time.

In MassWare, the one-time initialization time in the startup phase includes $2.5RTT$ communication time, where $1.5RTT$ is for the TCP connection and $1RTT$ is for synchronization message exchanges and the transmission delay of the meta-data for multiple component-chains stored in the synchronization request message. The initialization time is represented as:

$$T_{MassWare\_INIT} = T_{comm} + T_{comp} = 2.5RTT + n\_policy \times s\_chain/B \quad (5.5)$$

The initialization time of MassWare reconfiguration is related to the RTT, the number of policies, and bandwidth. we set the RTT as $100ms$ (referring [26]) to compute the initialization time on the number of policies and the bandwidth, as shown in Fig. 5.3.

In the repetitive reconfigurations after the one-time initialization, the reconfiguration time is the sum of the component assembly and restoration time that is related to the number of components in the component chain. The reconfiguration time is expressed as:

$$T_{MassWare} = T_{comp} = 2n(t\_conn + t\_rest) \quad (5.6)$$

The coefficient of 2 is needed because the reconfiguration process is carried out at both the proactive actuator of the sender and the reactive actuator of the receiver. Fig. 5.6 shows the reconfiguration time obtained by the benchmark experiments, see Section 5.2.2.

From the above analysis, we can see that the repetitive reconfiguration time of each middleware framework is dependent on different parameters. To compare their performance directly and prove the efficiency of MassWare, we list the effect of each parameter to the repetitive reconfiguration time in Table 5.2.

The minimum RTT has a major impact to the reconfiguration time of MobiPADS since there are a lot of control message exchanges in the synchronization process of the MobiPADS reconfiguration. The minimum RTT is set as $100ms$ in [26], which is in the same level as our experiment results. The performance of the

Table 5.2: The configuration time affected by various parameters

Parameter	**MobiPADS**	**CARISMA**	**MassWare**
RTT	Related (major factor)	Related	Not related
B	Related	Related (negligible)	Not related
n_policy	Not related	Related (major factor)	Not related
n_component	Related	Not related	Related
t_conn/t_rest	Related (negligible)	Related (negligible)	Related (major factor)

CARISMA conflict resolution mechanism significantly depends on the number of policies. For example, it takes about $900ms$ to determine which policy to apply out of ten [8]. Both MobiPADS and CARISMA use synchronous synchronization protocols and their performance depends on the network conditions. On the contrary, MassWare uses an asynchronous synchronization protocol and there is no communication involved in the reconfiguration process. Its reconfiguration time only depends on the component restoration time and connection time, which is in the range of hundreds of micro seconds in our experiments. The component restoration time and connection time are also required by MobiPADS and CARISMA, but negligible compared with their communication time.

In summary, the reconfiguration time of MobiPADS and CARISMA is typically in the range of seconds and related to the network condition and system complexity. For example, MobiPADS reconfiguration takes about $2s$ for $20Kbps$ bandwidth and $1.4s$ for $1Mbps$ bandwidth according to our theoretical analysis, which achieve the same results as those reported in [26]. CARISMA conflict resolution time is about $1.2s$ for 10 policies and $1.7s$ for 20 policies, and the time grows exponentially with the number of contexts and conflicts by our theoretical analysis, which conforms to the empirical results in [8]. Furthermore, their robustness is affected by the

reconfiguration because it requires the availability of all the related peer agents and its failure would cause the crash of succeeding data packets.

Although the initialization time of the proposed asynchronous method in MassWare is similar to the reconfiguration time of the synchronous methods in MobiPADS and CARISMA, the MassWare initialization is a one-time process while the reconfiguration of MobiPADS and CARISMA is a repetitive process. The repetitive reconfiguration time of MassWare after initialization is significantly shorter than that of MobiPADS or CARISMA. Moreover, the local agent reconfiguration is not affected by the network condition or the capacity of other agents, and the received data packets can be processed asynchronously based on their active message headers. Thus the robustness of the system is improved.

5.2 MassWare-MANET Evaluation by Experimental Measurements

In this section, we evaluation MassWare performance in MANETs using benchmark applications, which consist of a set of simple components, each of which has a parameter interface, an input interface, and an output interface so that two components can be connected. Each adaptation policy has the same components in a single test and each reconfiguration process will disconnect and reconnect all the components and load one parameter for each component. The number of the policies and the number of components in each policy are varied in the experiments to simulate different applications. All the data are calculated based on the average of 10 test samples.

One of the goals of MassWare is to reduce the reconfiguration time for DRE systems. However, the reconfiguration process introduces some performance cost,

such as the active message header and extra resource consumptions for maintaining the multiple chains. Therefore it is important to check the feasibility and efficiency of using MassWare. We evaluate its performance benefit and cost in this section in terms of the reconfiguration time, memory footprint, and scalability through benchmark applications on both PCs and PDAs.

5.2.1 Test Bed

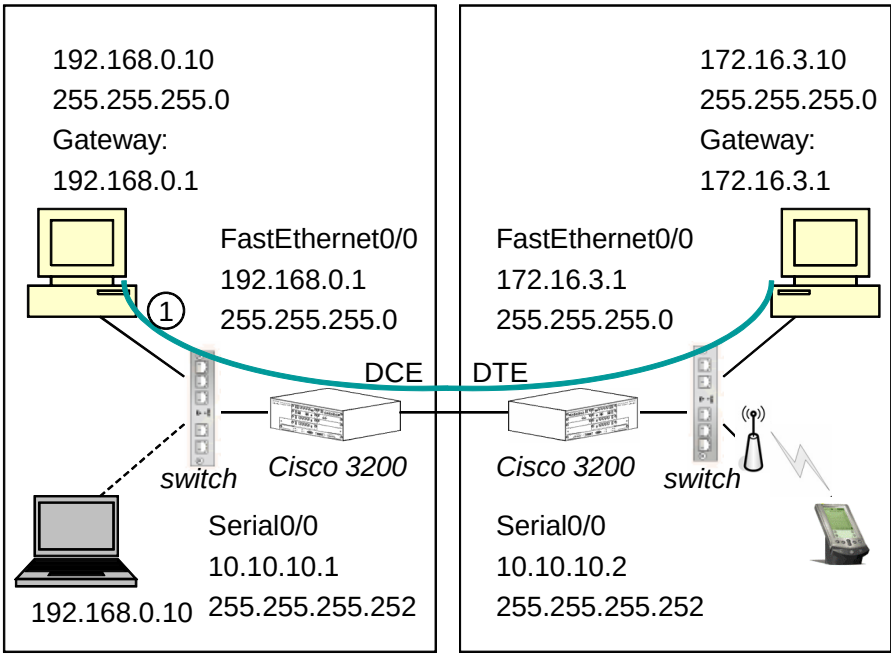


Figure 5.4: Experimental test bed.

We have implemented MassWare in C# for both Windows XP (WXP) and Windows Mobile 5 (WM5) systems using visual studio 2005, and we have encoded the script file using XML. The testbed consists of two PCs (Thinkpad-X60: Intel T2300 1.66GHz, 512MB, and WXP), two PDAs (Dell x51v: Intel XScale 624MHz, 64MB, and WM5), two Cisco routers (Cisco 3200), and two switches (Cisco 2900XL) as shown in Fig. 5.4 . The routers are connected back-to-back

through serial ports so that the network bandwidth can be controlled through the HyperTerminal tool.

5.2.2 Time Efficiency

One-time reconfiguration initialization time

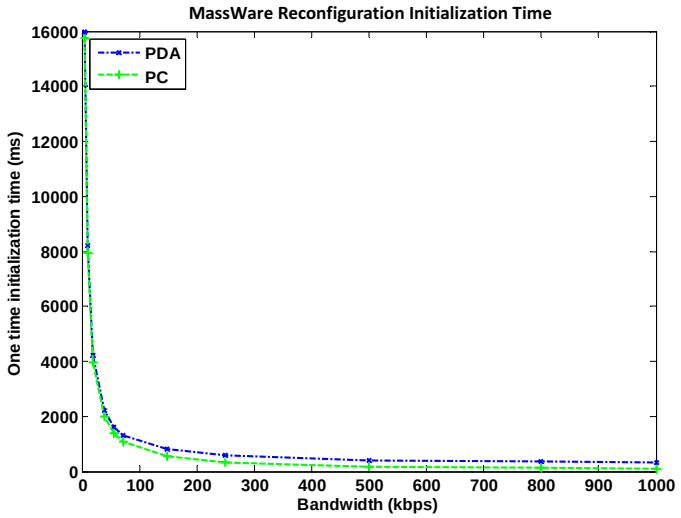


Figure 5.5: MassWare initialization time.

MassWare reconfiguration is separated into two phases: one-time initialization phase occurring once when the system starts up and repetitive reconfiguration phase occurring every time the reconfiguration is triggered. The one-time initialization time of MassWare has been analyzed theoretically and the analysis results are shown in Fig. 5.3. In this section, we evaluate the one-time initialization time using experimental results. According to Eq. 5.5, the initialization time of MassWare reconfiguration is related to the RTT, the number of policies, and bandwidth. To simplify the experiments, we set the RTT between the sender and receiver as $100ms$ and the number of policies as 10. The size of each policy, which contains an actuator meta-chain, is $10Kbit$. We then control the bandwidth by changing the

clock rate of the router serial ports to measure the variation of the initialization time to bandwidth. The experiment results are shown in Fig. 5.5, in which X-axis is the network bandwidth and Y-axis is the measured initialization time. In low bandwidth conditions, PDA and PC have similar performance since the communication time is the major part of the initialization time, which is only related to the network conditions. In high bandwidth conditions, local processing overhead also contributes to the initialization time and PDA has larger initialization time than PC due to its limited hardware resource. The experimental result also match well with the theoretical analysis with just slightly larger values because the unified model has ignored some processing overhead and control message transmission delay.

Repetitive reconfiguration time

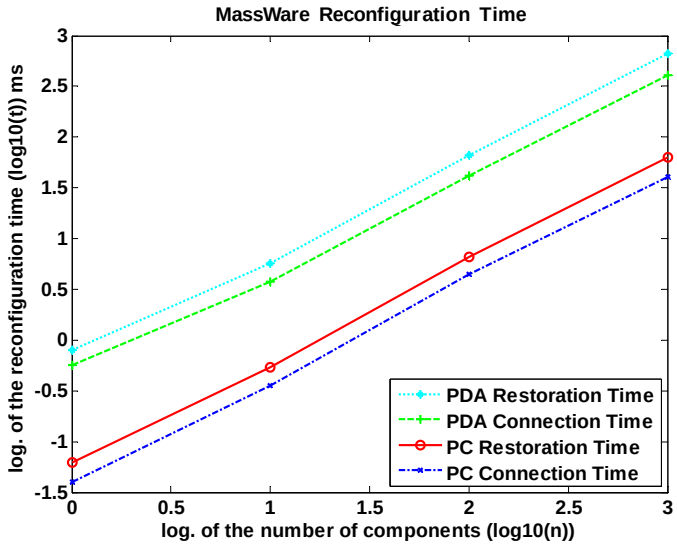


Figure 5.6: MassWare reconfiguration time.

The repetitive reconfiguration the most important part of MassWare reconfiguration since it occurs every time the reconfiguration is triggered. According to Eq.

5.6, it contains the connection time and status restoration time of the components in the active actuator, which are all local processing time and only related with the number of components. We then measure the component connection time and status restoration time separately in the experiments by changing the number of components. The experiment results are shown in Fig. 5.6. From the results, we can see that the change of both component connection time and status restoration time are linear of the number of components. PDA has larger reconfiguration time than PC due to its limited hardware resource. Because all the operations are executed in the same memory space and CPU process, it is in the range of several hundred microseconds to a few milliseconds. Moreover, the reconfiguration time is only determined by local hardware resources so that the time is very stable for every test.

Component initialization time

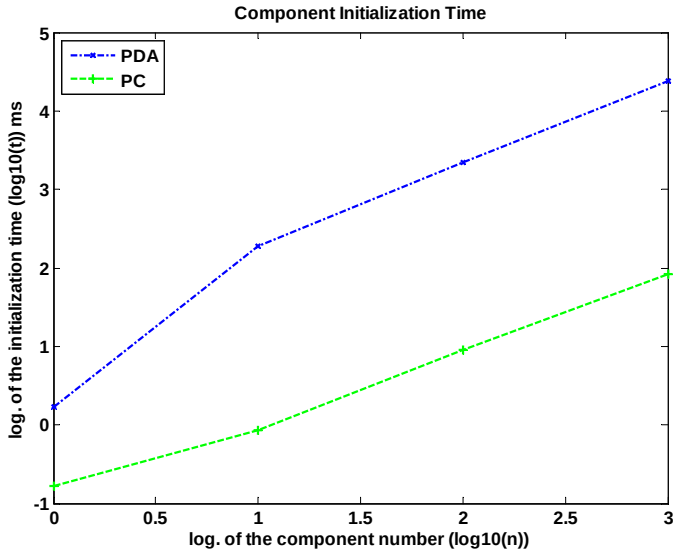


Figure 5.7: Component initialization time.

MassWare supports standard software components. After a MassWare agent is

loaded in the system initialization phase, it initializes all the components declared in the application script file. The component initialization time then represents the efficiency of the MassWare component model. It is defined as the time needed to load a component, check its types, and instantiate the component based on the encoded parameters. Fig. 5.7 shows the initialization time of the simple MassWare components based on the number of components. The initialization time is only related with local hardware resource and the change is linear of the number of component. The time is in the micro- to milli- second range.

We have also tested the event notification time of MassWare sensors, which is another important metric to evaluate the responsiveness of MassWare. Results show that it is in the microsecond level and much smaller than the repetitive reconfiguration time.

5.2.3 Memory Footprint and Scalability

In this experiment, we evaluate the local storage size and the run-time memory consumptions of MassWare framework, components and actuators. We utilize the C# serialization function to serialize MassWare and system objects and measure their run-time memory usage. Serialization means that objects are marshaled by value, that is, all their various member data are written out to the stream as a series of bytes. Therefore, we can use the length of the stream as the metric for memory consumption.

The local file size and run-time memory usage of MassWare middleware and components are shown in Table 5.3 for both Windows XP (WXP) and Windows Mobile 5 (WM5) respectively. The run-time memory usage of the middleware is measured after initializing the system and before loading and instantiating any components. An empty masslet is a reflective component containing no

Table 5.3: Resource consumption by MassWare

Components	**Windows XP system**		**Windows Mobile 5**	
	Local file size	**Run-time memory**	**Local file size**	**Run-time memory**
Middleware	$56KB$	$896KB$	$46KB$	$123KB$
Empty masslet	$4KB$	$139Byte$	$4KB$	$74Byte$
Simple masslet	$16KB$	$356Byte$	$5KB$	$147Byte$
Simple masstool	$16KB$	$279Byte$	$4KB$	$94Byte$

application-specific method or variable. A simple masslet contains one input interface, one output interface, and 5 double-type parameters. Although we use very similar source code for both WXP and WM5, the run-time memory consumption of WM5 is much less than that of WXP due to the code optimization in the mobile system.

Because MassWare contains multiple actuators, it is important to analyze the overhead of the actuators. The memory consumption R is then expressed as:

$$R = \sum_{i}(\sum_{j}(\sum_{k} p_{ijk} + l_{ij}) + a_i) \tag{5.7}$$

where p_{ijk} $(10Byptes)$ is the size of parameter k for masslet j in actuator i; l_{ij} $(12B)$ is the name and reference size of masslet j in actuator i; and a_i $(8B)$ is the index size of actuator i. For a MassWare agent that contains 5 actuators, 10 masslets for each actuator, and 10 parameters for each masslet, the resource consumption is $5640bytes$ $(\approx 5.5KB)$. In MassWare, a middleware agent maintains not only local proactive actuators, but also reactive actuators built for remote peer agents through the synchronization. Thus, the memory consumption is closely related to the application scale. According to Eq. 5.7, the memory consumption R for a DRE system is then modified as:

$$R = \sum_{t} (R_t) \tag{5.8}$$

where t is the index of peer middleware agents.

For a DRE system that has 10 distributed programs and each program has a middleware agent described above, the memory consumption of the program is $5640 bytes \times 10$ ($\approx 55KB$), which is still small compared to the capacity of most embedded devices.

5.2.4 Demo Applications and Releases

We have developed some adaptive DRE systems based on MassWare, like a first responder system in PDA platforms and a distance education system in heterogeneous platforms (PCs, Laptops, and PDAs). The implementation of real applications demonstrate that MassWare are easy to use, achieving fast responsiveness in reconfigurations, and supporting generic DRE systems. All the software we have developed and documents have been released on our website [70].

5.3 MassWare-WSN Evaluation

The MassWare-WSN system and supported applications have been developed in Ubuntu 6.10 Linux OS using the C language and tested on MicaZ nodes. We evaluated MassWare performance costs and benefits in terms of the memory footprint and component loading time. More comprehensive evaluations, including reconfiguration time and runtime memory consumption etc., will be part of our future work.

5.3.1 Memory Footprint

The memory footprint is the total static file size of the compiled MassWare components that are burned to sensor nodes. It is calculated by the component compiler. Because the decision engine is the extra part of MassWare in sensor nodes, we define the middleware memory overhead O_m as the ratio of the decision engine size to the application component size:

$$O_m = S_m/S_a = S_{dc}/(\sum S_{masslet} + \sum S_{masstool}) \tag{5.9}$$

where S_m is the middleware's memory-footprint size, which equals the decision engine's memory footprint (S_{dc}); S_a is the application's memory-footprint size, which equals to the total memory consumed by all masslets ($S_{masslet}$) and masstools ($S_{masstool}$).

Blink application

The blink application consists of three masslets: BlinkR (562$bytes$), BlinkG (532$bytes$), and BlinkY (478$bytes$), which control red, green, and yellow LEDs of a sensor node, one masstool: NeighborNum (586$bytes$), which measures the number of neighbor nodes, and the decision engine component (1178$bytes$). There are two adaptation policies for the Blink application:

- When there is no neighbor, *BlinkY* is connected to *BlinkR* and the frequency of *BlinkR* is one blink per 2 seconds;
- When there are one or more neighbors, *BlinkR*, *BlinkG*, and *BlinkY* are connected in sequence, and the frequency of *BlinkR* is one blink per 3 seconds.

The memory overhead of MassWare in the Blink application is 55%. MassWare is not memory efficient for simple applications since the application size is small.

Data compression application

The data compression application is developed based on a proposed sensor data compression algorithm [72] that aims to reduce distributed data redundancy. It consists of five masslets: *Sensing* (822*bytes*), *LSWT* (6628*bytes*), *Quantz* (8254*bytes*), *DSC* (1858*bytes*), and *Unary* (1614*bytes*), one masstool: *NeighborNum* (586*bytes*), and the *decision engine* component (1354*bytes*). *Sensing* has a start interface to start sampling data and sending them to output interfaces. *LSWT*, *Quantz*, *DSC*, and *Unary* perform Lifting Scheme Wavelet Transfer, Scalar Quantization, Distributed Source Coding, and Modified Unary Coding respectively. There are two adaptation policies:

- When there is no neighbor, which means there is no distributed redundancy, Sensing, LSWT, Quantz, and Unary are connected in sequence.
- When there are one or more neighbors, the Unary component is replaced by the DSC component, and Sensing, LSWT, Quantz, and DSC are connected in sequence.

The memory overhead of the data compression application is 6.9%, which is much lower than that of the blink application. This is because the decision engine size is only slightly larger for complex applications.

Benchmark experiment

Because the decision engine is generated based on a script file, its size is related to the number of adaption policies (N_p) and the number of masslets in each policy ($N_{c/p}$). The memory consumption can be expressed as:

$$M = A \times N_p + B \times N_p(N_{c/p} - 1) + C(N_{c/p} - 1) + D \times N_{c/p} + E \times N_p(N_{c/p} - 1) + F \quad (5.10)$$

where A, B, C, D, E, and F are coefficients. AN_p is the code memory for calculating the size of the synchronization request packet; $BN_p(N_{c/p}-1)$ is the code memory for generating the synchronization request packet; $C(N_{c/p}-1)$ is the memory for constructing the default actuator; $DN_{c/p}$ is the memory for initializing the default actuator; $EN_p(N_{c/p}-1)$ is the memory for building all actuators and subscribing them to corresponding detectors. F is other code memory consumption.

It is a special case when there is only one component in each actuator: since there is no component connection in this case, there is no code required to build actuators. When there is more than one component in each actuator, Eq. 5.10 can be simplified as:

$$M = A_1 \times N_p \times N_{c/p} + A_2 \times N_p + A_3 \times N_{c/p} + A_4 \tag{5.11}$$

where A_1, A_2, A_3, and A_4 are coefficients. To measure these coefficients, we have developed some benchmark experiments based on the number of policies (N_p) and the number of masslets in each policy ($N_{c/p}$). Every benchmark masslet has an input interface, an output interface, and a parameter interface. There is only one benchmark masstool used by all policies. In each policy, there is only one masstool in the detector; there are $N_{c/p}$ masslets connected in a sequence in the proactive actuator and the reactive actuator. The memory consumption of the decision is listed in Table 5.4.

Based on the measurement result, we get the coefficients $[A_1, A_2, A_3, A_4] = [7.82, 117.16, 90.88, 613.44]$. The standard deviation between the measurement results and calculation results using the coefficients are 36.53 (about 2% of the mean value), which also proves the correctness of the Eq. 5.11.

From the results, there are two observations:

Table 5.4: Benchmarking decision engine's memory size (byte)

$N_{c/p}$ \ N_p	1	2	3	4	5	6
1	792	818	838	858	878	898
2	910	1030	1214	1322	1428	1534
3	1014	1130	1320	1494	1626	1754
4	1118	1236	1478	1616	1752	1846
5	1208	1340	1594	1734	1842	1978
6	1296	1434	1678	1820	1962	2106

- the decision engine size is approximately linear to the number of policies and the number of masslets in each policy, which follows Eq. 5.11;
- even for a complex application (6×6), the decision engine size is still quite small (about $2KB$), compared to the size of most on-board programmable flash memory of sensor nodes (e.g. $128KB$ in MicaZ).

5.3.2 Time Efficiency

The decision engine loading time includes the time for creating and sending the *Synchronization Request* packet, constructing actuators and detectors, and starting the application, which can be expressed as:

$$M = A \times N_p + B \times N_p(N_{c/p} - 1) + C(N_{c/p} - 1) + D \times N_{c/p} + F \tag{5.12}$$

Compared to Eq. 5.10, the decision engine loading time does not contain $EN_p(N_{c/p}-1)$ since all other actuators are dynamically constructed except the default one. Eq. 5.12 can also be simplified as Eq. 5.11. Based on the benchmark results listed in Table 5.5, we get its coefficients: $[A_1, A_2, A_3, A_4] = [35.85, -4.5, 1.83, 273.02]$. The standard deviation between the measurement results and calculation

Table 5.5: Benchmarking decision engine's loading time (in CPU cycles)

$N_{c/p}$ \ N_p	1	2	3	4	5	6
1	335	362	363	390	417	445
2	358	408	485	536	600	664
3	381	494	581	695	781	934
4	417	554	691	840	990	1099
5	454	626	800	972	1145	N/A
6	490	699	936	1118	N/A	N/A

results using the coefficients are 12.19 (about 1.5% of the mean value).

From the results, there are two observations:

- the CPU loading time is also approximately linear to the number of policies and components in each policy;
- MassWare is time efficient for most applications (less than 1000 CPU clock cycles). (The last three readings (N/A) are caused by the failure to generate the *Synchronization Request* packet due to the limited memory of MicaZ nodes. This issue could be solved in new-generation sensor nodes with advances of micro-electronics, e.g. Imote2 hosts $256kB$ SRAM and $32MB$ SDRAM.)

5.3.3 Energy Consumption

MassWare consumes extra energy for application reconfiguration, which includes application local behavior change and distributed behavior synchronization. Since the energy is primarily consumed by wireless communications in WSNs [73], MassWare is energy efficient compared with other middleware frameworks since it does not require extra communication in the repetitive reconfiguration process

by using the active-message-based synchronization protocol. The only communication energy consumption for the reconfiguration is the one-time transmission of the "Synchronization Request" packet at the system initialization phase, which is negligible compared to the energy consumption of the life time sensor data transmission.

On the other hand, MassWare can reduce the energy consumption when reprogramming sensor applications. MassWare is component-based adaptive middleware and it supports partial updates of the sensor application by only replacing the required software components instead of the whole application.

Chapter 6

Applications and Implementation

To justify the functionality of MassWare in real applications, we have implemented MassWare to support two popular applications: Ad-hoc routing in MANETs and data compression in WSNs. Beyond the implementation, we have designed two new algorithms in these application areas to solve some existing problems. Evaluations demonstrate the significant performance improvement of MassWare-supported applications by using the designed algorithms in the specific contexts.

6.1 MassWare-Supported Routing Application in MANETs

In Mobile Ad-hoc NETworks (MANETs), how to select an optimal route from the source to the destination is a critical issue. Due to node mobility and link/channel dynamics, a link that exists between two nodes now may not exist in the future. Therefore, many routing protocols [74][75][76] have been proposed for various scenarios of MANETs. Due to the heterogeneity of the networks or the mobile devices, a single routing algorithm may not be suitable for the whole network. MassWare-supported applications have the ability to select different routing algorithms based on contextual information. Therefore, it can significantly improve the application performance by adopting the optimal routing algorithm in the current scenario.

Traditional ad-hoc routing can be divided into three categories: on-demand routing, table-driven routing, and hybrid routing. In on-demand routing, nodes only maintain route information when they need to send or relay packets. However, on-demand routing has longer response time than table-driven routing, and it does not scale well because of flooding of routing requests. In table-driven routing, each node always maintains up-to-date information about routing to any other nodes.

It would induce a heavy overhead for maintaining routing information in highly mobile scenarios. The hybrid routing is designed to achieve a tradeoff between the characteristics of on-demand and table-driven routing, mostly with a cost of high algorithm complexity. However, how to improve the routing scalability as well as reducing the routing overhead in MANETs is still an open problem.

Geometric routing is a special type of routing approach designed for MANETs where data packets are routed based on position information when node positions are known. In general, geometric routing is simple and efficient, and it improves the routing scalability because each node only needs to keep its neighbors' position information. There are common assumptions when designing geometric routing protocols. First, every node knows its own position. This information can be collected by using GPS devices or other means. Second, every node knows its neighbors' positions that can be obtained by one hop beacon messages. Third, the source node knows the destination position. This function can be provided by some location service mechanisms [77]. As the development of positioning devices such as GPS, geometric routing is becoming more and more practical.

Most existing geometric routing protocols are based on the greedy algorithm where every forwarder chooses the neighbor that is the closest to the destination as the next hop. Although the greedy algorithm is simple and efficient, it fails when a node cannot find a neighbor close to the destination for forwarding a packet (a "void area"). To guarantee the packet delivery, some geometric routing algorithms, such as GPSR (Greedy Parameter Stateless Routing) [77] and GOAFR (Greedy Other Adaptive Face Routing) [78], use face routing to bypass void areas. Face routing only works in planar graphs where there is no cross link. However, failures of face routing based on planarization have been observed in test-bed experiments [79][80] due to inconsistent radio ranges and asymmetric links.

We have designed a new geometric routing protocol for MANETs called LTGR

(Local Tree based Greedy Routing), which uses a local tree, instead of face routing, to recover routing bypassing void areas. LTGR uses the same assumptions as existing geometric routing protocols, but it does not require assumptions of uniform radio ranges and bi-directional links that are hard to be satisfied in real implementations. LTGR has the following features. It is simple and stateless so that it is suitable for highly dynamic MANETs. LTGR does not rely on planarization. Thus it keeps cross links in the network topology to achieve good hop stretch performance. It augments the greedy algorithm with routing history information to make informed decisions in routing.

In this section, we first present the design details of the LTGR. After that, we provide the implementation method and an example of a MassWare application that uses the LTGR algorithm when geometric information is available and switches to DSR (a traditional routing algorithm) when geometric information is unavailable. Last, we evaluate the performance improvement of the MassWare application by using LTGR in terms of the packet delivery ratio, routing overhead, and hop stretch.

6.1.1 Related Work

There exists ongoing research on geometric (position based) routing protocols [77][78][79][81]. The simplest one is based on the greedy algorithm by which each node, when forwarding traffic as a forwarder, chooses its neighboring node closest to the destination as the next hop. However, the greedy algorithm can not pass any void area where a forwarder can not find a neighbor that is closer than itself to the destination.

In order to recover packet routing from void areas and improve the packet delivery ratio, Karp et al. propose GPSR [77] to switch from the greedy mode to

a perimeter mode if a node cannot find the next hop using the greedy algorithm. In the perimeter mode, face routing [77][82] combined with a right-hand rule is utilized to traverse the perimeter of the void area. The basic idea of the face routing is to travel along the perimeter of the faces, which are intersected by the virtual line between the source and the destination. GPSR only uses the right hand rule to choose the next face for traversal. GPSR is not efficient if it can not find the correct face quickly; and in the worst case, it traverses all the bad faces and finds the correct one last.

To improve the performance of face routing, Kuhn et al. propose a Greedy Other Adaptive Face Routing (GOAFR) protocol [78]. GOAFR uses an adaptive face routing (AFR) mode if the greedy mode encounters a void area. The basic idea of AFR is to adjust the boundary of a traverse ellipse area around the face and choose an optimal value to reach the destination. The boundary is decided by the Boundary Face Routing (BFR) that uses the same rule of face routing except that the exploration around a face will walk back when it reaches the boundary. If a packet can not reach the destination via BFR, it will be routed back to the source node [78]. The boundary is then doubled, and the process is repeated again until the destination is reached.

Both the GPSR and GOAFR protocols planarize network topology to support face routing and the planarization (GG or RNG [77]) algorithms assume that the connectivity between nodes can be described by unit graphs, i.e. a node is always connected to all neighbors within its fixed transmission range while not connected to nodes outside this range. In an experimental deployment of GPSR protocol in wireless sensor networks, Kim et al. [79][80] observe that permanent routing failure occurs because the unit-graph assumption cannot be satisfied in practical scenarios. To solve this problem, Kim et al. [79] propose a distributed Cross-Link Detection Protocol (CLDP) to planarize the network. However, CLDP is complex

and costly because it induces new routing overhead caused by "probe" packets used for planarization.

In [81], Leong et al. present a new geometric routing protocol without using network planarization, i.e. Greedy Distributed Spanning Tree Routing (GDSTR). The GDSTR protocol generates spanning tree(s) and every node maintains a convex hull based on its children in the spanning tree and their convex hulls. When a node can not forward a packet using the greedy algorithm, it switches to the recovery mode and checks if the destination is contained in its convex hull and decides whether to forward the packet to a proper child or just send it to its parent, which has a bigger convex hull. GDSTR can achieve better hop stretch and path stretch than CLDP with less overhead. However, GDSTR is proposed for static sensor networks and not stateless ones. Therefore it is not suitable for MANETs because the convex hull maintenance is costly in dynamic scenarios.

Similar to GDSTR, our proposed protocol, LTGR, does not use planarization either. However, LTGR differs from GDSTR in that it is stateless and does not need any global information or extra message exchange to recover packet routing from void areas. Like GOAFR, LTGR keeps the adaptability of routing exploration in the recovery process, i.e. LTGR adaptively selects a sub-tree for packet forwarding based on the position information of the leaf nodes in each sub-tree when it routes packets. And because the selection utilizes position information, instead of a constant boundary value used in GOAFR, LTGR can make better routing decisions than GOAFR.

6.1.2 Local Tree Based Geometric Routing (LTGR)

Overview of LTGR

Like existing geometric routing protocols, LTGR takes advantage of a greedy algorithm to route packets whenever possible. A packet can be either routed in the greedy mode if the greedy algorithm works or in the recovery mode if the packet routing reaches a void area. A flag in the packet header marks the routing mode of a packet. LTGR uses a local tree based routing algorithm to route packets in the recovery mode to bypass void areas. Whenever a node receives a recovery-mode packet, it checks whether it is closer to the destination than the originator of the recovery process, and if positive, it switches the packet back to the greedy mode and uses the greedy algorithm to forward the packet. This process is repeated until the destination is reached, or all the possible paths are tried once and still no route is found to reach the destination, i.e. the network is partitioned.

If a packet is routed in the greedy mode, it would not encounter any routing loop. If the packet is routed in the recovery mode, the local tree information used by the LTGR to bypass the void area will be embedded in the packet header, thus any node can get the history of the tree to avoid forming routing loops. The local tree could expand to a spanning tree covering all nodes in the network. Therefore, LTGR can guarantee the packet delivery if there exists a path between the source and the destination.

Search Algorithms

LTGR uses local tree based search algorithms to find paths bypassing void areas so that packets in the recovery mode can be routed.

In this dissertation, we first study the uniform cost search where the source node knows nothing about the whole topology and the destination's position. For

example, the breadth first search, in which all nodes at level d are expanded before any nodes at level $d+1$, finds the shallowest goal state, i.e. the shortest path. If we define the function DEPTH(n) as the depth of the node n , then the node with the lowest DEPTH(n) value is always expanded first. If the route cost is a function of the depth of the solution, e.g. the hop count of the route, the breadth first search can achieve the best solution, i.e. the optimal path.

Although the uniform cost search can find the optimal route provided that there is no negative cost, it would traverse most of the possible routes, which could induce a large amount of overhead. Assume B stands for the average branching factor, and D is the depth of the solution, the complexity of the uniform cost search is $O(B^D)$. This overhead is prohibitive in large scale MANETs.

To address the overhead issue, we consider the greedy search algorithm. A node using the greedy search algorithm always finds a neighbor node as the next hop that is the closest to the destination among all neighbor nodes. Thus the greedy algorithm can select the next hop exclusively, therefore eliminating the overhead of traversing other possible route in the uniform cost search algorithm. Moreover, it is faster than the uniform cost search algorithm on average.

However, the greedy search algorithm is efficient but incomplete, which can not guarantee finding an existing path to the destination because no history information is recorded. The uniform cost search is complete because it records its history; and it can find the optimal path but is not as efficient as the greedy search. Thus it is desirable to combine them for path searching.

In this dissertation, we augment the greedy search algorithm with its history and propose the local tree based search algorithm.

A tree consists of a root, branch nodes and leaf nodes. The first node N0 that routes a packet reaching a void area marks the packet to be in the recovery mode and initializes the recovery process and the local tree: it sets itself as the root of the

local tree and its neighbors as the leaf nodes. After the tree is constructed, the tree information is stored in the packet header and forwarded along with the payload to the next-hop node N_1 that is a leaf node of the tree closest to the destination according to the greedy search algorithm. When a leaf node, say N_1, receives the packet, it first retrieves the local tree information from the packet header and checks if it is closer than the root to the destination. If so, the routing mode of the packet will be switched back to the greedy mode and the tree information will be removed from the packet header. Otherwise, the node N_1 expands the local tree by adding all its neighbors as its children; and thus it is changed from a leaf node to a branch node. The description of the expanded local tree is stored in the packet header and the packet is forwarded to a leaf node of the updated tree, say N_2, which is the closest one to the destination among all leaf nodes, based on the greedy search algorithm. Note that N_2 may not be the neighbor of N_1 but N_1 can find a path to N_2 based on the local tree information.

The local tree based search algorithm is complete since it can guarantee the packet delivery if there exists a path to the destination.

LTGR Protocol

Based on the LTGR protocol, a node in MANETs routes a packet by the greedy algorithm if the packet does not encounter a void area. Otherwise the packet will be routed in the recovery mode of the LTGR protocol.

Because LTGR uses a tree structure, there is no loop in the path. And in the worst case, the local tree will expand to a spanning tree that can reach every node in the network. The challenge of embedding local tree information in the packet header is that the header overhead may be very large in dense networks. To address this challenge, we propose two techniques used in LTGR.

First, each node divides the network space into four quadrants when it receives a recovery mode packet. The division is based on the axis x, which is the line connecting itself with the destination, and axis y, which is the line perpendicular to axis x and passing this node. After that, the node adds only three neighbors to the local tree, which are distributed in the three quadrants (except the quadrant that contains the previous hop node) and are closest to the destination among the neighbors in each quadrant. This means that each branch node in the local tree only has a maximum of three children no matter how dense the network is, and only the root has maximum four children from all four quadrants. The above-mentioned process may need to be repeated to guarantee the packet delivery when the network is not partitioned if a packet could not be routed to the destination node in the first round.

Second, we propose a new compression technique to compress the local tree information stored in the packet header. For the tree structure, we only use 2 bit memory to save the structure for each branch node because it only has 3 children. For the node information, because only the to-destination-distance values of the root and the leaf nodes are useful in the recovery mode, we do not need to include the values of branch nodes in the packet header. Compared to face routing algorithm that keeps the position information for all the nodes along a face, LTGR has much less overhead. Suppose that there are n nodes in total and m leaf nodes in a local tree, and the node id and the to-destination-distance value are represented by $kbits$ and $tbits$ respectively, the total bits that are needed to store the tree information is:

$$T(n, m, k, t) = 2(n - m) + k \times n + t(m + 1) \tag{6.1}$$

An example that uses LTGR to recover from the void area is illustrated in

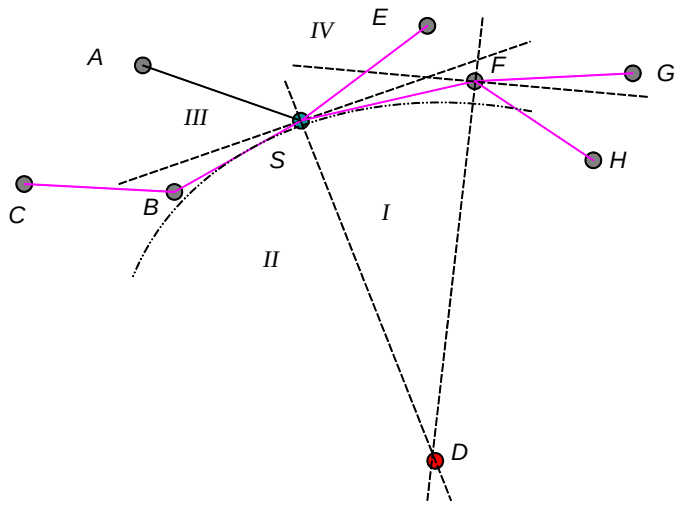


Figure 6.1: A local tree based routing example.

Fig. 6.1. A is the sender and D is the destination. First, A sends a packet to S using the greedy algorithm because S is closer than A to D. When S receives the packet, it initializes the recovery process because it has no neighbor closer to D than itself. It constructs a local tree by setting itself (S) as the root and adding three neighbors F, B, and E as leaf nodes, which are closest to D in the quadrants I, II, IV respectively. Then the packet is forwarded to B because B is the closest to D among all the leaf nodes. When B receives the packet, it adds C to the local tree. However, B finds that the leaf node with the shortest distance to D is F, instead of C or E. And it forwards the packet to F through the path $B \rightarrow S \rightarrow F$. After that, when F receives this packet, it adds its neighbors H and G to the local tree, and forwards the packet to H that is the closest to D among all the leaf nodes. At last, H receives the packet and finds it is closer than the root S to the D. It changes the packet to the greedy mode and forwards the packet using the greedy algorithm.

6.1.3 LTGR and MassWare Application Implementation

To develop a MassWare-supported MANET application, developers need to provide application required software components (masslets and masstools) and an adaptation-policy script file. In this example, masslets include the LTGR algorithm and other routing algorithms that are implemented as independent software components to be used by MassWare; masstools include the measurement tool (called GPS) that detects whether geometric information is available for a current mobile device.

Component Implementation

```
public class LTGR : Masslet, DataInput
{
   public LTGR() { /*...*/ }

   #region Interface functions
   public override bool Start() { /* Start LTGR */ }
   public override bool Stop() { /* Stop LTGR */ }
   public void SetRouteCalcFreq(int frequency) { /*...*/ }
   public void SetBeaconInterval(int interval) { /*...*/ }
   #endregion

   #region Masslet functions
   public event DataReadyHandler DataReadyEvent;
   public AppDataReadyHandler DataRequestHandler;
   void DataOutput.AddSubcriber(DataReadyHandler subs,
                                bool bAdd)
   {
      if (bAdd) DataReadyEvent += subs;
      else DataReadyEvent -= subs;
   }
   AppDataReadyHandler DataInput.GetSubscriber()
   {
      if (DataRequestHandler == null)
         DataRequestHandler = new
            AppDataReadyHandler (DataReceived);
      return DataRequestHandler;
   }
   void DataReceived (object sender, AppDataEventArgs e)
   {
      if (DataReadyEvent != null)  SendData(e.mData);
   }
   #endregion Masslet functions

   void SendData(IntPtr data) { /* Send data using LTGR */ }
   //...
}
```

(a) LTGR masslet implementation

```
<Masslets>
 <component cid="2002">
  <addr> D:\Masslets\LTGR.dll </addr>
  <name> Masslets.Routing.LTGR </name>
  <ctype> Masslet </ctype>
  <alias> LTGR </alias>
  <param pid="001">
   <name> SetRouteCalcFreq </name>
   <vtype> Int32 </vtype>
   <value> 5000 </value>
  </param>
  <param pid="001">
   <name> SetBeaconInterval </name>
   <vtype> Int32 </vtype>
   <value> 1000 </value>
  </param>
  <interface iid="001">
   <name> DataInput </name>
   <itype> Input </name>
   <Message> AppDataEventArgs </Message>
  </interface>
  <interface iid="002">
   <name> Start </name>
   <itype> Start </name>
  </interface>
 </component>
 ...
</Masslets>
```

(b) LTGR masslet declaration

Figure 6.2: Dynamic reconfiguration architecture

The first step is to implement specific algorithms or protocols into masslets. Since masslets are function-independent software components, they can be easily shared by various applications to reduce development cost. In this section, we present an implementation example of LTGR masslet as shown in Fig. 6.2.

Fig. 6.2a shows the C# code of the LTGR masslet that is developed for Window Mobile systems with .Net Compact framework 2.0. The interface functions implement the component interfaces that can be declared in the script file to set parameters (parameter interfaces) or communicate with other masslets (communication interfaces). The masslet functions connects the user-defined interfaces with real functionality implementation (the SendData function in this example). Therefore, any masslet can implement its own interfaces independently. To communicate with each other, an output interface must support the same message type with the input interface (the AppDataEventArgs in this this example).

By reading the metadata of the LTGR masslet, developers can create a script file to declare the component based on implemented interfaces. The masslet declaration script, as shown in Fig. 6.2b, is part of the application adaptation-policy script file (shown in Fig. 6.3) that is used to build the MassWare application.

Similar to LTGR masslets, developers need to create the other masslets and masstools if they are not shared components for the application.

Script file Implementation

After all components are prepared, the next step is to create a script file that declares the required components and adaptation policies as discussed in Chapter 3. A full script file example of the LTGR-based dynamic routing application is depicted in Fig. 6.3. The simplified application contains 3 masslets: LTGR and DSR, which implement LTGR and DSR routing protocols separately, and Sender,

```
<Marchlets>
 <component cid="2002">
  <addr> D:\Masslets\LTGR.dll </addr>
  <name> Masslets.Routing.LTGR </name>
  <ctype> Masslet </ctype>
  <alias> LTGR </alias>
  …
 </component>
 <component cid="2002">
  <addr> D:\Masslets\DSR.dll </addr>
  <name> Masslets.Routing.DSR </name>
  <ctype> Masslet </ctype>
  <alias> DSR </alias>
  …
 </component>
 <component>
 <addr> D:\Masslets\Sender.dll </addr>
  <name> Masslets.AppTest.Sender </name>
  <ctype> Masslet </ctype>
  <alias> Sender </alias>
  …
 </component>
 …
</Marchlets>

<MarchTools>
 <component cid="3001">
  <name> Masstools.GPS </name>
  <alias> GPS </alias>
  <interface iid="001">
   <name> isAvailable </name>
   <itype> Output </name>
  </interface>
 </component>
</MarchTools>
```

```
<Rules>
 <rule>
  <sensor>
   <event>
    <otype> EQ </otype>
    <lhs><expr>GPS.isAvailable</expr></lhs>
    <rhs><expr> 1 </expr> </rhs>
   </event>
  <sensor>

  <Actuator type="proactive" sync="Async">
   <SetParam>
    LTGR.SetRouteCalcFreq = 10000;
    LTGR.SetBeaconInterval = 2000
   </SetParam>
   <SetArch>
    Sender.DataOutput -> LTGR.DataInput;
    DSR.Stop;
    LTGR.Start;
   </SetArch>
  </Actuator>
 </rule>

 <rule>
  <sensor><-- GPS.isAvailable == 0></sensor>
  <Actuator type="proactive" sync="Async">
  <SetArch>
    Sender.DataOutput -> DSR.DataInput;
    LTGR.Stop;
    DSR.Start;
   </SetArch>
  </Actuator>
 </rule>
 …
</Rules>
```

Figure 6.3: The full MassWare application example using LTGR

which sends test data to another device in the same MANET. It also contains one masstool: GPS that detects whether geometric information is available. There are two adaptation policies used by the be application: LTGR is selected to route application data when GPS information is available since LTGR achieves better performance than DSR, and DSR is used when GPS information is unavailable to guarantee that the data can be delivered successfully.

6.1.4 Simulation and Analysis of Results

MassWare supports context-aware reflective applications, which can dynamically select optimal software components based on contextual information to improve the application performance. To validate the performance improvement of MassWare-supported routing application comparing to static applications, we compare LTGR with peer protocols. We implemented LTGR in ns-2(.28) and simulated the protocol using various mobile ad-hoc network topologies. We have also tested this protocol with different traffic patterns. LTGR is compared with GPSR instead of DSR since GPSR is also a geometric routing protocol and has been proved to outperform DSR in [77]. The GPSR source code was downloaded from its authors' website [83]. We do not compare LTGR with CLDP and GDSTR since they are designed for static sensor networks and not suitable for highly dynamic MANETs. Although there are numerous metrics to evaluate a routing protocol design for MANETs, we focus mainly on the packet delivery ratio, routing overhead, and average hop stretch achieved by the routing protocols.

Simulation Setup

We use the same parameters that are listed in Table 6.1 for both LTGR and GPSR simulations.

Table 6.1: Simulation parameters

Parameter	**Value**
Beacon interval	1s
Transmission range	250m
Position variable size	12 bits
Network size	1000m × 1000m
Simulation time	900s

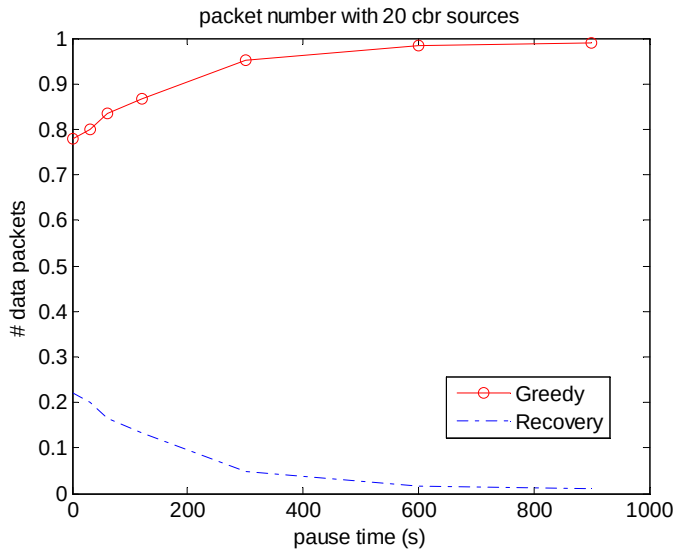


Figure 6.4: Percentage of packets in recovery mode vs. pause time.

- Movement model: Nodes move according to the "random waypoint" model [74]. We observe that the probability that a packet would be routed in the recovery mode is inversely proportional to the pause time (Fig. 6.4). Because both protocols use the greedy algorithm whenever possible, we set the pause time as 0 to compare their performance in the recovery process. We use CMU scene generator to generate 80 different pattern files based on 8 different numbers of nodes: 20, 30, 40, 50, 60, 70, 80 and 90 (10 files for each number respectively). The moving speed of nodes is distributed uniformly between 1 and 20 m/s. Both protocols are simulated in all the scenarios and the average values are calculated.

- Traffic pattern: we choose UDP as our transport layer protocol. Randomly selected 14 nodes generate 20 traffic flows. The transmission rate of every flow is $1Kbps$: one $512bytes$ packet is generated every 4 seconds. The starting time instances of the traffic flows are uniformly distributed between 0 and 180 seconds.

Both LTGR and GPSR are simulated based on 80 various scenarios combining the above-mentioned traffic pattern and movement models.

Packet Delivery Ratio

Packet delivery ratio is the number of packets received by the destinations divided by the number of packets originated from the sources in the application layer. It describes the loss rate of networks and characterizes both the completeness and the correctness of a routing protocol [74].

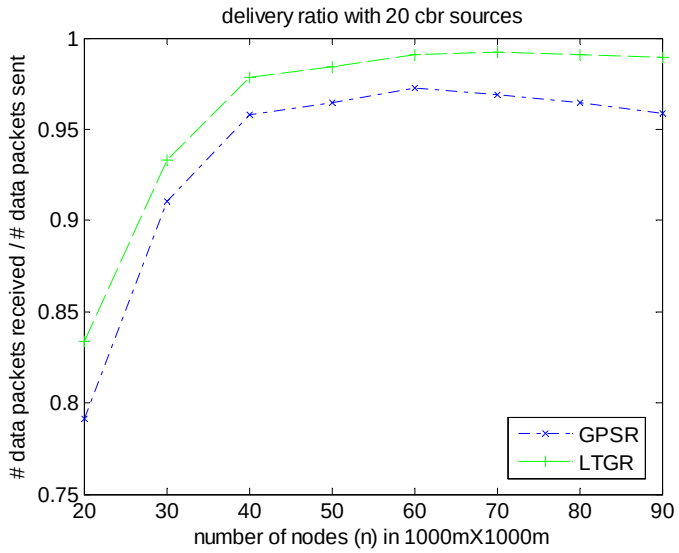


Figure 6.5: Packet delivery ratio vs. the number of nodes.

Both protocols can achieve good delivery ratio and there is only a slight difference between them as shown in Fig. 6.5. This is because both protocols can guarantee the packet delivery if there is a path between the source and the destination. LTGR performs slightly better than GPSR because they use different recovery algorithms. In fact, some packets are dropped by LTGR because of ARP errors that are caused by neighbors' mobility, while more packets are dropped by GPSR due to TTL (set to be 128 in both protocols) errors in that the face routing

may cause infinite routing loops in the dynamic scenarios. Because both LTGR and GPSR use the greedy algorithm when it works, we can conclude that the local tree based routing can achieve a higher success ratio than face routing when bypassing void areas.

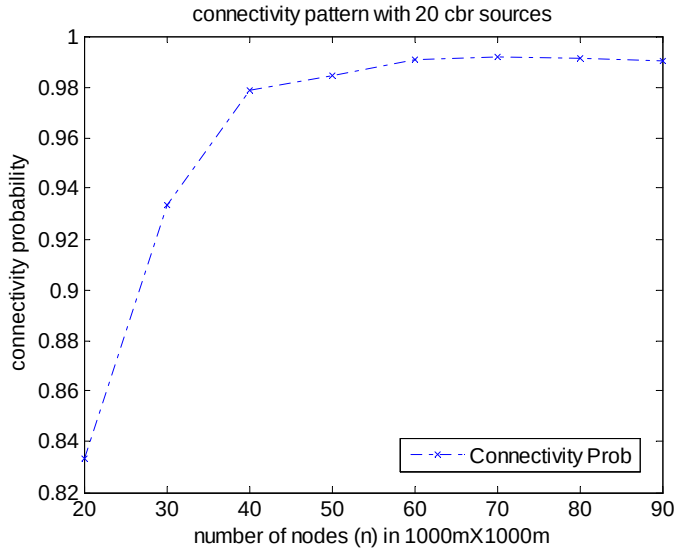


Figure 6.6: Source-destination connectivity probability vs. the number of nodes.

The delivery ratio increases along with the increase of the node density because the connectivity probabilities of the source nodes and the destination nodes increase when the node density increases, as shown in Fig. 6.6.

Average Hop Stretch

The average route hop stretch stands for the route optimization degree of a routing protocol. By establishing a shorter route, a routing protocol can take advantage of the network resources more efficiently. In this dissertation, we define hop stretch as the ratio of the real hop count that a packet passes from the source to the destination to the hop count of the optimal path. Because both protocols use the greedy algorithm whenever possible and most packets can reach the destination

by only using the greedy algorithm as the shown in Fig. 6.4, it is undesirable to compare the average hop stretch of all transferred packets. In our simulation, we mark every packet that has been delivered in the recovery mode (via the local tree algorithm or the face routing algorithm), and only compare the average hop stretch of these packets.

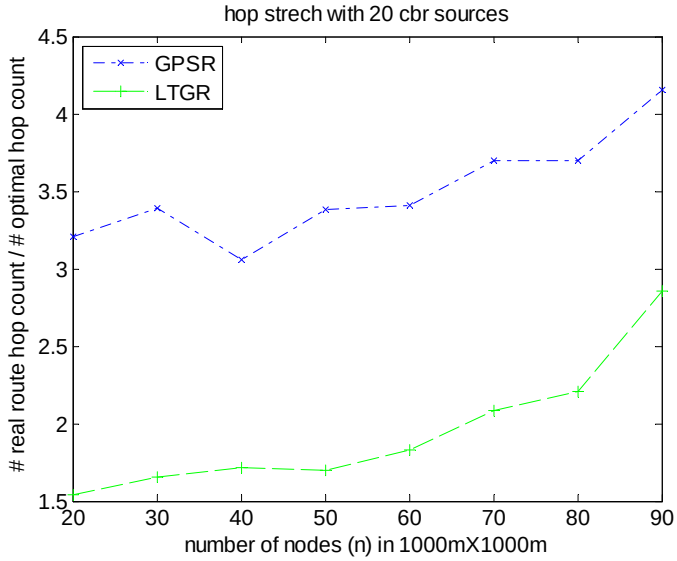


Figure 6.7: Average hop stretch vs. the number of nodes.

The average hop stretch of LTGR is much better than that of GPSR as shown in Fig. 6.7. GPSR use the face routing algorithm to recover from the void area. However, face routing is not efficient because of the following reasons. First, it uses the right hand rule to choose a face blindly. Second, it has to complete the face if it chooses a wrong face before changing to the next face. Third, in the planarization process, some shorter paths would be deleted, e.g. the diagonals of a full connected rectangle would be deleted to planarize the graph. LTGR is efficient because it still utilizes the position information to decide the next hop, i.e. the greedy search uses the to-destination-distance values of each leaf node. Because LTGR does not use planarization, it also keeps the shortest path to the destination.

The average hop stretches of both protocols roughly increase as the node density increases. This is because the number of possible paths to the destination increases as the node density increases. Thus the probability of selecting a right but not optimal path increases too.

Protocol Overhead

Routing overhead is another important metric for comparing the routing protocols because it implies the efficiency of a protocol in terms of bandwidth consumption and battery power usage. Large routing overhead induces congestion in a low-bandwidth environment and harms the scalability of the network. Here we define the routing overhead as the number of bytes sent by all nodes divided by the number of payload data bytes received by the destinations. The overhead does not include IEEE 802.11 RTS/CTS packets or ARP packets. However, it does include the overhead of the IP header because we modify the IP header in LTGR to store the position information of the destination as what is done in the GPSR. Thus the IP header in LTGR implementation could be larger than those in other ad-hoc routing algorithms, such as AODV. The routing overheads of GPSR and LTGR are shown in Fig. 6.8.

The overheads of both protocols generally decrease as the node density increases. This is because the connectivity probability of the network increases as the node density increases (as shown in Fig. 6.6). If the network is partitioned, both algorithms would try all the possible paths before dropping the packets, which generates more overhead.

There are two features of LTGR that contribute to its better overhead performance than that of GPSR. One is that the delivery ratio achieved by LTGR is higher than that of GPSR. Because GPSR drops extra packets due to TTL errors,

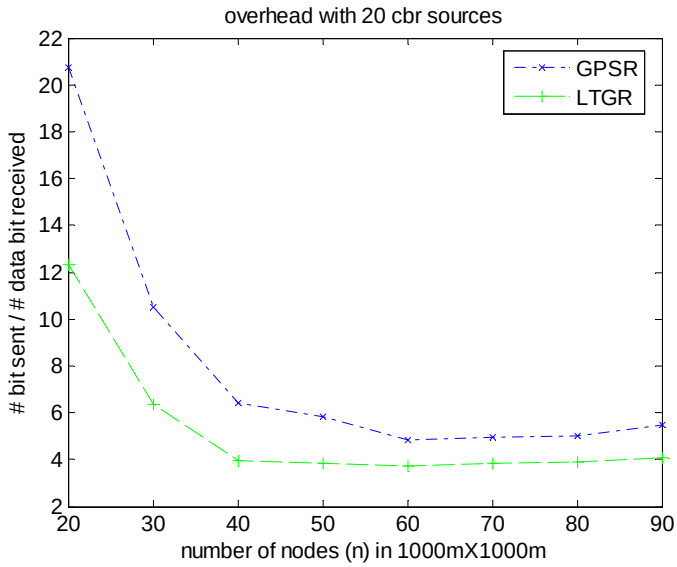


Figure 6.8: Protocol overhead vs. the number of nodes.

these packets generate extra overhead. The other is that LTGR is more efficient than GPSR in terms of hop stretch, which means that the local tree based routing algorithm in LTGR can bypass the void area quicker than the face routing algorithm in GPSR. A longer search process for a correct path requires the packets to be delivered to more invalid path candidates that cause more overhead.

6.1.5 LTGR Summary

In this section, we have developed a MassWare-supported routing application in MANETs and proposed a stateless geometric routing protocol LTGR to overcome the shortcomings of face-routing-based protocols. The MassWare-supported routing application outperforms static routing applications since it is able to dynamically select an optimal routing protocol based on the availability of geometric information. As validated in our simulations, LTGR is more efficient than GPSR, which outperforms traditional routing protocols, in terms of routing overhead and

hop stretch shown by extensive simulation results, e.g. LTGR can reduce the routing overhead by 25 $\sim$ 40% and hop stretch by 30 $\sim$ 50% comparing to GPSR in our simulation scenarios.

6.2 MassWare-Supported Data Compression Applications in WSNs

Data compression has been an important technique to reduce the redundancy of the raw data in WSNs. Large volumes of sensor data generated will make the data transmission between sensor nodes and a remote data acquisition center a very challenging task, especially given the limited power and bandwidth of currently available wireless sensors [84]. Data compression facilitates the power conservation of WSNs since the energy of a sensor node is consumed primarily by wireless communications [73]. In a dense sensor network, redundancy exists in both data collected at individual sensor nodes, which is called local redundancy, and data obtained from correlated sensor nodes, called distributed redundancy (assuming the sensor network is synchronized [85][86]). Classical data compression techniques [87], which involve transformation, quantization, and encoding, can reduce the local redundancy. Due to the nature of distributed sensor deployment in WSNs, an appealing technology for reducing distributed redundancy is DSC [88][89]. DSC refers to the compression of multiple correlated sensor outputs without communication among the sensor nodes: sensor data are encoded locally according to a predefined correlation and decoded at the remote sink based on the side information.

MassWare can support the data compression applications to dynamically select optimal compression algorithms based on contextual information (e.g. the existence of the data correlation). When there is no distributed redundancy (no data correlation), the applications have to choose local data compression algorithms to ensure the data quality. And when there is distributed redundancy, the applications can choose distributed data compression algorithms to improve the

compression ratio. Since sensor nodes are usually randomly deployed in heterogenous environments, it is difficult or impossible to predict the data correlation of distributed sensor nodes. MassWare then provides a power tool to measure the data correlation and switch to the optimal algorithm based on the correlation at runtime.

For the implementation of DSC in WSNs, one of the requirements is that the correlation is well known by each sensor node and sink. Most existing DSC algorithms take the correlation in time or space domain that constraints DSC's implementation only to process smooth (low frequency) signals. For high frequency signals, the correlation is hard to be found duo to high frequency noise pollution. In structural health monitoring applications, the collected raw vibration sample data consists of both a high frequency component, which is induced by high sample frequency ($50Hz$) and noise, and a low frequency component, which is our primary interest in because it contains the critical information of structure health menace [90].

We have designed a constructive algorithmic framework that supports DSC for high- and low-frequency signal compression in WSNs. To separate the low frequency component from the high frequency component, while keep the time-domain correlation among distributed sensor data, LSWT is used to preprocess the original data for signal decomposition and noise reduction. Since LSWT is more efficient than FFT or DCT and its transformed data keep time domain information, it can be naturally combined with DSC to reduce both local and distributed redundancy. To the best of the author's knowledge, it is the first time that the LSWT and DSC have been integrated for vibration data compression.

In this section, we first introduce the proposed LSWT-DSC algorithm. Then, we present the implementation details of a MassWare-supported data compression algorithm that uses the LSWT-DSC algorithm. Last, we evaluate the performance

improvement of the MassWare application by using the new algorithm, which achieves a higher compression ratio than the classical compression technique [90] while obtaining the same data quality.

6.2.1 Related Work

The problem of distributed data compression and data aggregation in sensor networks has led to new research challenges in networking, information theory and algorithm [91][92][88]. In [93], Slepian and Wolf have theoretically shown that separate encoding (with increased complexity at the joint decoder) is as efficient as joint encoding for lossless compression. Similar results were obtained by Wyner and Ziv with regard to lossy coding of joint Gaussian sources [94]. Currently, DSC is an active research area - more than 30 years after Slepian and Wolf laid the theoretical foundation [95]. S.S. Pradhan et al. [88] provide a constructive practical framework based on algebraic trellis codes dubbed as Distributed Source Coding Using Syndromes (DISCUS). They address the problem of compressing correlated distributed sources. They also discuss the rate loss from the DISCUS which is separated into source coding loss and channel coding loss.

Although DSC has been implemented successfully in some sensor network scenarios [96][97][98], most of them are based on time or space domain correlation. These algorithms work well only for low frequency signal. In the application, the sample frequency is 50Hz, making it difficult to decide the correlation of the sensor data as traditional DSC compression algorithms due to noise pollution. In this dissertation, we apply DSC in the frequency domain, and the proposed algorithms are suitable for both high- and low- frequency sensor data. In our work, the original data are decomposed into the low frequency component and the high frequency

component by LSWT. Scalar quantization is then utilized to treat each input symbol separately in producing the output to reduce the noise and strengthen the correlation. This algorithm can achieve a much higher compression ratio than another vibration data compression algorithm [90] with favorable signal-restoration quality.

There have been some implementations of LSWT in the structure health monitoring system based on WSNs. Both [99] and [90] utilize the LSWT to compress vibration data. Although their methods successfully reduced the high frequency information and achieved a good compression performance, e.g. a compression ratio of 1:14 in [90], they only compressed the individual data in every single sensor node instead of reducing the redundancy of distributed source data. In the proposed algorithms of this section, we not only compress the data generated by the individual source, but also consider the correlation of data from neighbor nodes. Experimental results indicate that the proposed algorithms can achieve a higher compression ratio while attaining the same signal to noise ratio as theirs because DSC is a lossless algorithm.

6.2.2 Distributed Source Coding and Lifting Scheme Wavelet Transform

Distributed Source Coding

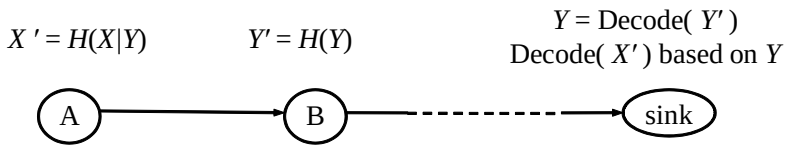


Figure 6.9: Basic structure of distributed source coding.

When DSC is implemented in WSNs, the correlated sensor nodes send their

encoded data to the base station (sink) for joint decoding. Assume $\{X_i\}$ and $\{Yi\}$ are the sequences of sample values collected in the sensor node A and B respectively (see Fig. 6.9). In the individual source coding, the entropy $H(X)$ and $H(Y)$ must be sent respectively to the base station. However, according to Shannon's theory, if they are correlated discrete random variables of independent and identical distribution (i.i.d), only joint entropy $H(X,Y)$ is needed for lossless compression if they are encoded together. Slepian-Wolf extends Shannon's theorem further to that even if X and Y must be separately encoded, a rate $H(X,Y)$ can also be achieved if decoding of X and Y is done jointly. In WSNs, that means every individual node can compress its data and reduce the distributed redundancy only based on its own information. DSC is very suitable for WSNs because it excludes the data exchange, which would be very expensive for the extremely limited power and bandwidth, among the correlated neighbor nodes in WSNs. Slepian-Wolf source coding is lossless. While in practice, Slepian-Wolf coding is often combined with quantization to provide an approach to address lossy DSC problems.

An Example of Slepian-Wolf Coding [88]

For binary sample value $Xi, Yi \in \{000, 001, \ldots, 111\}$, each of them needs to be encoded by 3 bits/sample. However, if the correlation is known that the hamming distance between X_i and Y_i is $d_H \leq 1$, the value space can be divided into four cosets: $Z_{00} = \{000, 111\}$, $Z_{01} = \{001, 110\}$, $Z_{10} = \{010, 101\}$ and $Z_{11} = \{100, 011\}$. In every coset, the hamming distance between any two values is larger than or equal to 3. Y is encoded into $H(Y) = 3$ bits/sample and this original data is sent to the base station. Additionally, X is encoded as the index of the cosets $H(X \mid Y) = 2$ bits/sample and this compressed data is also sent to the base station. In the base station, Y is first decoded, and then X can be decoded depending on the side

information Y. For a given Y, there are only four possible choices which belong to four separate cosets under the condition $d_H \leq 1$. For example, when $Y = 000$, $X \in \{000, 001, 010, 100\}$. Assume encoded value $X' = 01$ is the index of the coset, the only answer $X = 001$ can be decided because the hamming distance between $Y = 000$ and 110 (the other value in the coset) is 2 which is larger than $\max(d_H)$. Thus the Slepian-Wolf limit of $H(X, Y) = H(Y) + H(X \mid Y) = 3 + 2 = 5$ bits is indeed achieved in this example with lossless decoding.

From the above example, we can generalize the Slepian-Wolf coding to the case when X and Y are equiprobable $2n$ bit binary sources. Here $n \geq 3$ is a positive integer. The correlation model between X and Y is again characterized by $d_H(X, Y) \leq 2k - 1$. Let $m = k + 1$, then $H(X) = H(Y) = n$ bits per sample, $H(X \mid Y) = m$ bits per sample, and $H(X, Y) = n + m$ bits per pair of samples for joint encoding.

For the implementation of DSC in WSNs, the choice of parameters n (the source codebook size) and k (the correlation) is an interesting topic. It could be automatically adjusted based on the history information. In this dissertation, these values are generated by statistic analysis.

Wavelet and Lifting Scheme Wavelet Transform

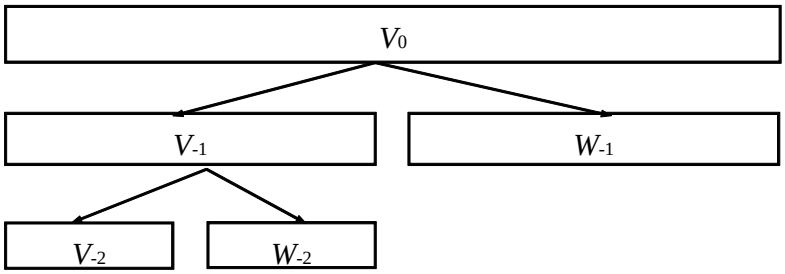


Figure 6.10: The wavelet decomposition tree with a scale level $n = 2$.

Because the interesting information of the structure monitoring application lies

in the low frequency component, it is important to decompose it from the high frequency component. Wavelet Transform (WT) can analyze the signals in a frequency domain and decompose signals into the low frequency and high frequency components. DSC works well for the low frequency component, and the high frequency component is quantized to a small value later. WT outperforms traditional frequency transforms, i.e. FFT and DCT, because it does not need to know the global time domain information and can detect both the low frequency and the high frequency information automatically. Another important advantage of WT is that it can analyze the signal in multi-scales - the low frequency component can be decomposed again (Fig. 6.10), which provides us the potential to achieve a tradeoff between the data quality and compression ratio. The computational complexity of Mallat WT is $O(n)$ and lower than that of FFT and DCT, which is $O(n \lg n)$. Therefore, Mallat WT is also called fast wavelet transform. All of these characteristics indicate that WT is suitable for data compression and DSC naturally in WSNs.

LSWT is the second generation WT which is extended from Mallat algorithm. LSWT replaces the *translating* and *dilating* operations of conventional WT with *splitting*, *prediction* (dual lifting) and *updating* (primal lifting) operations. Compared to the first generation WT, LSWT has three main advantages when implemented in sensor networks [100][101]. First, it is faster than the Mallat algorithm (although the computational complexity is still $O(n)$). Second, unlike the first generation WT, its inverse transform is easy to find and implement. Last, LSWT provides integer to integer mapping which is favorable in WSNs because the sensed data is a 10 bit integer.

In the simulation, we have tested two kinds of popular wavelets: CDF(1,1) (Haar wavelet) and Daubechies D4 wavelet [102]. In the version of Daubechies D4 transform, LSWT consists of splitting, two updates, prediction, and normalization.

Splitting refers to splitting the original data set λ_{j+1} into the even part λ_j and the odd part γ_j. The first update is to use the odd part to update the even part. After that, the even part is used to predict the odd part, followed by the update process again. The last step is normalization. The sequence of the steps is listed as equations (6.2) $\sim$ (6.5):

Update 1:

$$\lambda_j = \lambda_j + \sqrt{3}\gamma_j \tag{6.2}$$

Predict:

$$\gamma_j = \gamma_j - \frac{\sqrt{3}}{4}\lambda_j + \frac{\sqrt{3}-2}{4}\lambda_{j-1} \tag{6.3}$$

Update 2:

$$\lambda_j = \lambda_j - \gamma_{j+1} \tag{6.4}$$

Normalize:

$$\lambda_j = \frac{\sqrt{3}-1}{\sqrt{2}}\lambda_j \text{ and } \gamma_j = \frac{\sqrt{3}+1}{\sqrt{2}}\gamma_j \tag{6.5}$$

Assume that the length of data set λ_{j+1} is $2n$. To handle the edge problem, λ_{-1} and γ_n is replaced by λ_{n-1} and γ_0 respectively.

6.2.3 System Design

Simulation System Structure

In the simulation, we analyze the performance of our proposed algorithms and compare them with the algorithms in [90] based on the same sample data. The data are collected using Micaz nodes with $128K$ program flash memory and $10bit$ analog to digital converter by a civil engineering research group. The sensor notes are fixed in a five layer civil infrastructure model [90] and the distance between each layer is $15cm$. The first layer is attached to a motherboard which is driven by

a vibration exciter. All the upper layers oscillate along with the lower layers. One sensor node is put in each layer and acceleration data is collected at a sample frequency $50Hz$. Related research [99] has evaluated the accuracy of the ADXL202E onboard accelerometer for structure health monitoring and its modifications have been proposed. All the data collected are saved in the RAM. After collecting 4096 samples, the data will be compressed in the sensor node and sent to the base station which is connected to a PC. The data compression process is described in details in the next section.

Compression Process

To take advantage of DSC, the node in the first layer of the structure sends the original (self compressed) data as side information to the base station, and each other node sends the DSC compressed data.

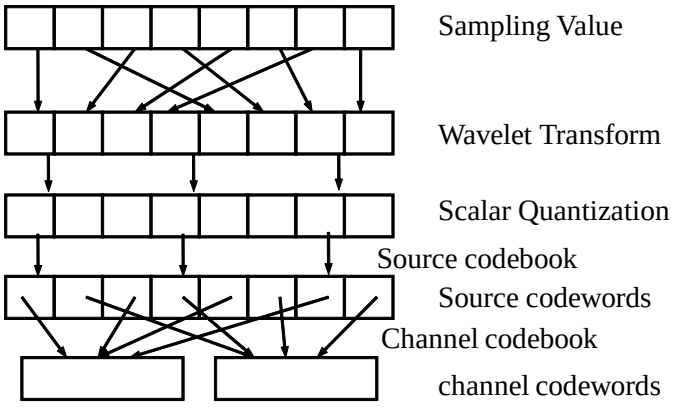


Figure 6.11: The compression process in sensor nodes.

The compress process is illustrated in Figure 6.11.

- The first step is LSWT. LSWT can be repeated by iteration on the λ_j, creating a multi-level or multi-resolution decomposition.

- The second step is quantization. Scalar quantization is used in the application to reduce the individual redundancy since it has been widely studied and successfully implemented in data compression combined with WT. After the quantization, most of the high frequency data value are set to zero. A modified unary coding algorithm is used to encode the high frequency data set. That is: only the nonzero data set $\{x_i\}$ are encoded. If $x_i > 0$, it is encoded with $2 \times x_i bits$ 1 and $10 bits$ relative position information. If $x_i < 0$, it is encoded with $2x_i - 1 bits$ 1 and $10 bits$ relative position information. For the low frequency component, it is encoded by DSC described in the following steps.

- The third step is to map the coefficient to the source codebook. The codebook area is from 0 to $2^n - 1$. n is decided by the vibration character and statistical analysis. After the mapping, every data value is encoded into n bits per sample which represents $H(Y)$. We rename the encoded data as base data. If the data are collected in the first layer sensor node, the compression process will be ended here and the base data will be sent to the base station, otherwise it continues to the next step.

- The fourth step is DSC. The data set is partitioned into different cosets as the channel codebook, and the original data are replaced with the channel codeword, which is the coset index and represents $H(X \mid Y)$. We rename these kinds of data as fully compressed data.

After the base station receives all the collected data, it will decompress all the data step by step. The decompression process includes:

- First, the base station decompresses base data received from first layer sensor node. Because these data represent $H(Y)$, they can be decompressed without

any side information.

- Then, the second layer data are decompressed based on the decompressed first layer data as side information.
- After that, the decompressed data of the second layer node can be utilized to estimate and decompress the data from the third layer. This process is repeated till all the data are decompressed.

The proposed compression algorithms are lossy ones. The distortion includes quantization error conduced by scalar quantization and estimation error conduced by the channel coding. To achieve tolerable distortion ratios, we can adjust the quantization parameter and correlation parameter k.

Data Format and System Topology Extension

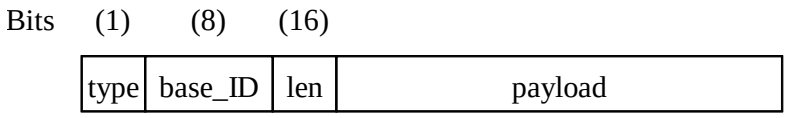


Figure 6.12: The compressed data format.

The encoded data structure in the application is illustrated in Fig. 6.12:

Type field distinguishes the base data (0) from fully compressed data (1). If *type* field is 0, *base_ID* field is its own node ID. Otherwise *base_ID* is the ID of its correlated node whose data is used as side information. *Len* field is the length of payload data.

For the implementation of the proposed algorithms to large scenarios of WSNs, we should also consider the topology management for DSC. In [103], four DSC encoding schemes are provided, with the compression rate and loss factor discussed separately. The cluster head, which sends the original data, could be selected

dynamically to balance the power consumption. The topology-controlled data compression algorithms will also be part of our future research topics.

6.2.4 MassWare-Supported Data Compression Application

We have briefly discussed the implementation of the MassWare-supported data compression application in 5. In this section, we will discuss more details about the application implementation. A sensor data compression algorithm normally contains three computing components (masslets): Transformation, Quantization, and Coding. In this example, we have implemented LSWT as the transformation masslet, scalar quantization as the quantization masslet, and two coding masslets: DSC and Modified Unary coding to reduce distributed and local redundancy separately. There is only one masstool in the application: Neighboring, which measures the number of neighbor nodes and maps the number to node density.

Component Implementation

As discussed in Chapter 5, there are five masslets: Sensing, LSWT, Quantz, DSC, and Unary; and one masstool: Neighboring in the data compression application. We only present the DSC masslet implementation as an example. The rest of component implementation can be found on the MARCHES website [70].

To develop a MassWare component, developers need to specify required interfaces in the source code using the keywords *massware* and *interface*, as shown in Fig. 6.13a. the source code is then be compiled with the *Component Compiler* to create a SOS-supported binary module and a human-readable file (meta-file), which contains the component meta-information as shown in Fig. 6.13b. Through the meta-file, users can get the component name, ID, alias, and its parameter interfaces and communication interfaces. Therefore, generic sensor services can

```
massware interface MSG_SET_CORRELATION
{
  int ID = MOD_MSG_START+102
  InterfType iType = Parameter
  ActionType aType = Set
  ValueType vType = uint8
}
massware interface MSG_FORWARD_INPUT
{
  int ID = MOD_MSG_START+100
  InterfType iType = Communication
  ActionType aType = Input
  MsgType mType = MassWareInputMsg
}
massware interface MSG_FORWARD_OUTPUT
{
  int ID = MOD_MSG_START+101
  InterfType iType = Communication
  ActionType aType = Output
  MsgType mType = MassWareOutputMsg
}
massware interface MSG_INVERSE_INPUT
{
  int ID = MOD_MSG_START+103
  InterfType iType = Communication
  ActionType aType = Input
  MsgType mType = MassWareInputMsg
}
massware interface MSG_INVERSE_OUTPUT
{
  int ID = MOD_MSG_START+104
  InterfType iType = Communication
  ActionType aType = Output
  MsgType mType = MassWareOutputMsg
}
```

(a) DSC interfaces

```
<component cid="2003">
 <name> DSC_MOD_ID </name>
 <comId> APP_MOD_MIN_PID + 46 </comId>
 <alias> DSC </alias>
 <interface type="Paramter">
  <name> MSG_SET_CORRELATION </name>
  <intfId> MOD_MSG_START + 102 </paramId>
  <actionType> Set </actionType>
  <valueType> Integer </valueType>
 </interface>
 <interface type="Communication">
  <name> MSG_FORWARD_INPUT </name>
  <intfId> MOD_MSG_START + 100 </intfId>
  <actionType> Input </actionType>
  <msgType> MassWareInputMsg </msgType>
 </interface>
 <interface type="Communication">
  <name> MSG_FORWARD_OUTPUT </name>
  <intfId> MOD_MSG_START + 101 </intfId>
  <actionType> Output </actionType>
  <msgType> MassWareOutputMsg </msgType>
 </interface>
 <interface type="Communication">
  <name> MSG_INVERSE_INPUT </name>
  <intfId> MOD_MSG_START + 103 </intfId>
  <actionType> Input </actionType>
  <msgType> MassWareInputMsg </msgType>
 </interface>
 <interface type="Communication">
  <name> MSG_INVERSE_OUTPUT </name>
  <intfId> MOD_MSG_START + 104 </intfId>
  <actionType> Output </actionType>
  <msgType> MassWareOutputMsg </msgType>
 </interface>
</component>
```

(b) DSC metafile

Figure 6.13: A DSC masslet example.

be implemented as standard MassWare components and easily shared by different WSN applications. Different components can be identified by the decision engine according to their IDs.

Script file Implementation

After all components are prepared, the second step is to develop a script file that declares the required components and adaptation rules using the XML language. In this example, one possible adaptation policy is: when a node has five or more

```
<DecisionEngine xmlns:xsi=...>
<MassTools> … </MassTools>
<Masslets>
 <component cid="2003">
  <name> DSC_MOD_ID </name>
  <comId> APP_MOD_MIN_PID + 46 </comId>
  <alias> DSC </alias>
  <interface type="Parameter">
   <name> MSG_SET_CORRELATION </name>
   <interfId>MOD_MSG_START + 102</interfId>
   <value> 1 </value>
  </interface>
  <interface type="Communication">
   <name> MSG_FORWARD_INPUT </name>
   <interfId>MOD_MSG_START + 100</interfId>
   <actiontype> Input </actiontype>
  </interface>
  <interface type="Communication">
   <name> MSG_FORWARD_OUTPUT </name>
   <interfId>MOD_MSG_START + 101</interfId>
   <actiontype> Output </actiontype>
  </interface>
  <interface type="Communication">
   <name> MSG_INVERSE_INPUT </name>
   <interfId>MOD_MSG_START + 100</interfId>
   <actiontype> Input </actiontype>
  </interface>
  <interface type="Communication">
   <name> MSG_INVERSE_OUTPUT </name>
   <interfId>MOD_MSG_START + 101</interfId>
   <actiontype> Output </actiontype>
  </interface>
 </component>
 ...
</Masslets>

<AdaptationPolicies>
 <policy pid="001">
  <detector did="001">
   <event>
    <otype> GE </otype>
    <lhs><expr>NEIGHBER.NUMBER</expr></lhs>
    <rhs><expr> 5 </expr></rhs>
   </event>
  </detector>

  <Actuator aid="001">
   <atype> ProActive </atype>
   <SetParam>
    DSC.MSG_SET_CORRELATION = 1;
   </SetParam>
   <SetArch>
    SENSOR.MSG_OUTPUT_DATA->LSWT.MSG_FORWARD_INPUT;
    LSWT.MSG_FORWARD_OUTPUT->QUAN.MSG_FORWARD_INPUT;
    QUAN.MSG_FORWARD_OUTPUT->DSC.MSG_FORWARD_INPUT;
    DSC.MSG_FORWARD_OUTPUT->MASSWARE.OUT(BCAST_ADDR);
   </SetArch>
  </Actuator>
  <Actuator aid="101">
   <atype> ReActive </atype>
   <SetArch>
    MASSWARE.IN->DSC.MSG_INVERSE_INPUT;
    DSC.MSG_INVERSE_OUTPUT->QUAN.MSG_INVERSE_INPUT;
    QUAN.MSG_INVERSE_OUTPUT->LSWT.MSG_INVERSE_INPUT;
   </SetArch>
  </Actuator>
 </policy>
 ...
</AdaptationPolicies>

</DecisionEngine>
```

Figure 6.14: The data compression application script file example.

neighbors, which means there exists distributed redundancy, DSC is selected to reduce distributed redundancy and the correlation value is set 1. According to the application requirement, users can change or add more policies. For example, when a node has more neighbors, the correlation value can be increased; on the other hand, if the number of neighbors is less than 5, unary coding instead of DSC should be selected to reduce local data redundancy only. The script file is compiled by the MassWare compiler to create the decision engine component.

The third step is to load all compiled components to sensor nodes with SOS. Masslets and masstools need to be loaded before the decision engine component. After the decision engine is loaded, it will configure the components to start the

application.

6.2.5 Experiments and Simulations

MassWare can optimize application performance by dynamically choosing suitable software components or adjusting component parameters in the current context. The performance of a sensor data compression application stands for its compression ratio, restored data performance (distortion ratio) and computational complexity. In this section, we will analyze these characters of the proposed algorithms and compare them with other peer algorithms in various scenarios so that application developers can take advantage of their merits to meet the application requirements in different scenarios when designing adaptation policies.

The system structure of the experiment is depicted in the previous section. The vibration exciter generates the vibration and drives the motherboard which connects the five-layered structure. The basic methodology used to measure the properties of the proposed compression algorithms is to change the vibration frequency of the exciter. However, because the highest sample frequency of the accelerometer (ADXL202E) in the sensor board is 60Hz, we can't detect higher frequency information. To justify the proposed WT-DSC based algorithms, Different White Gaussian Noises, varying in noise degree, are added to the collected sample value. We have compared the results of the proposed algorithms with other existing compression algorithms [90]. We name the proposed algorithms Haar-DSC (Haar wavelet based DSC) and Daub-DSC (Daubechies4 wavelet based DSC) and rename the algorithms in [90] as Haar-MUC and Daub-MUC.

The experiments also compare the compression properties based on different wavelets and DSC parameters. The results are analyzed for each experiment. In

order to enable a direct, fair comparison between different algorithms, we have implemented each selected algorithm on 20 sets of raw data sampled from 20 scenarios with disparate vibration frequencies. Because these algorithms are challenged in the same identical condition, their performance can be compared directly.

In the experiments, we only measure the performance and results of the compression algorithms, rather than simulate the wireless sensor network topology. We have implemented the experimental code [102] using MassWare structure.

Compression Ratio

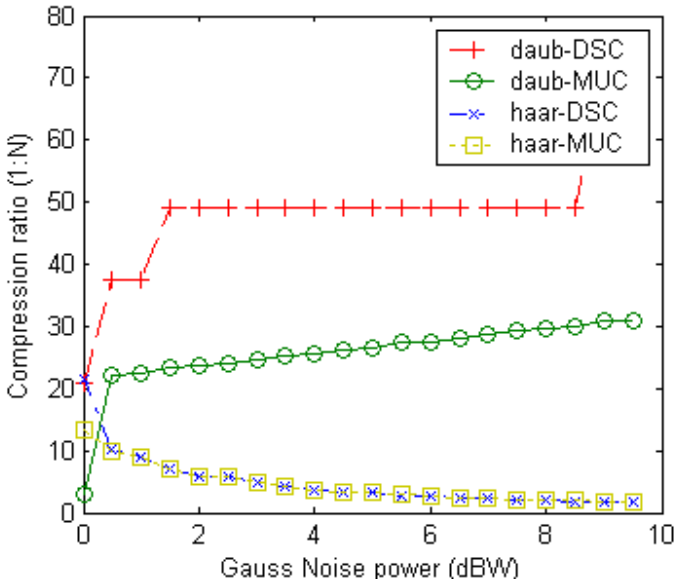


Figure 6.15: Compression ratio vs. noise degree.

Compression ratio is one of most important criterion for a compression algorithm. Fig. 6.15 highlights the relative compression ratio of the three compression algorithms as the noise degree increases. From the results, we can see that for the collected original signal without high frequency noise, Haar-DSC can achieve

a higher compression ratio than that of Daub-DSC. However, as the noise degree increases, the compression ratio of Daub-DSC increases rapidly while that of Haar-DSC decreases. Haar WT performs an average and difference on each pair of neighbor values. For the original signal without noise, the high frequency component that stands for the difference consists of mostly zeros, and the low frequency component that stands for the average is smoother than that of Daubechies wavelet transformed signal, which picks up some neighbor nodes for high pass and low pass filters, because there is an overlap between iterations in the transform step, and the overlap makes the transformed data not as smooth as that of Haar WT. The smoother the signal, the higher the correlation, allowing Haar-DSC to achieve a better compression ratio than Daub-DSC. However, as the noise power increase, the high frequency component in the Haar-DSC conserved more high frequency information which can not be filtered and it makes Modified Unary Coding inefficient.

DSC based compression algorithms, which reduce both the local and distributed redundancy, always outperform MUC based algorithms. Haar-DSC achieves almost the same compression ratio with the Haar-MUC when the noise ratio is larger than $0.5dBW$. The reason is that the high frequency component contains large values under this condition, and most of the compressed data bits come from this part which is encoded by MUC in both algorithms.

Compression Performance

The proposed compression algorithms in this dissertation are lossy algorithms, and the information is lost for quantization and estimation error. The compression performance is evaluated using three means: Peak Signal to Noise Ratio (PSNR), time domain analysis and low frequency domain analysis. PSNR of a reconstructed

signal x_i^* compared to the original signal x_i is defined as:

$$PSNR = 20 \log_{10} \left(\frac{x_{peak}}{RMSE} \right) dB \tag{6.6}$$

where $x_{peak} = \max_i \mid x_i \mid$ and the Root Mean Square Error:

$$RMSE = \sqrt{\frac{\sum_{i=0}^{n} (x_i - x_i^*)^2}{n}}$$

where n is the length of the sample data set.

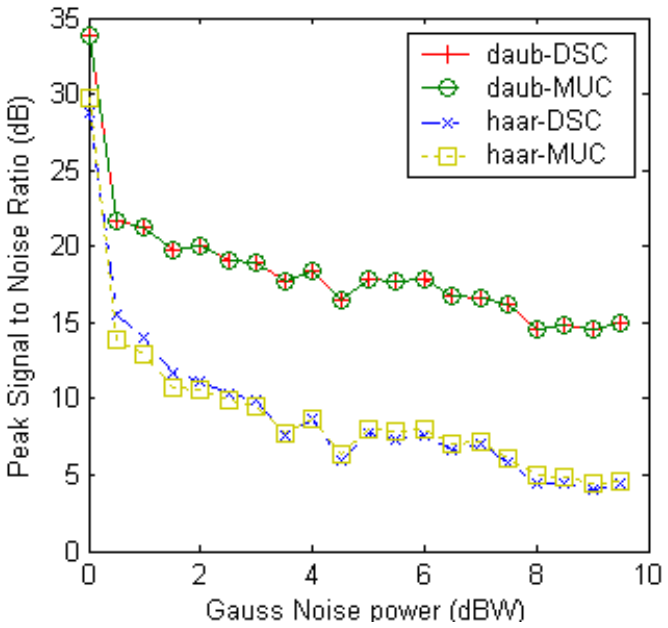
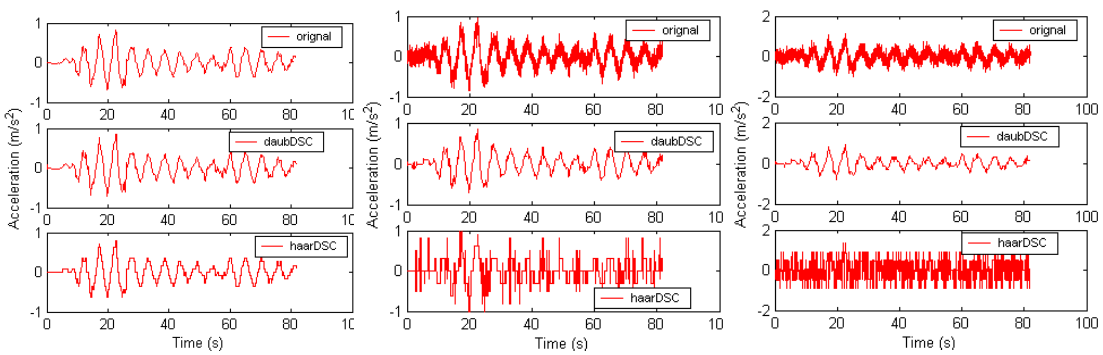


Figure 6.16: The peak signal to noise ratio vs. noise degree.

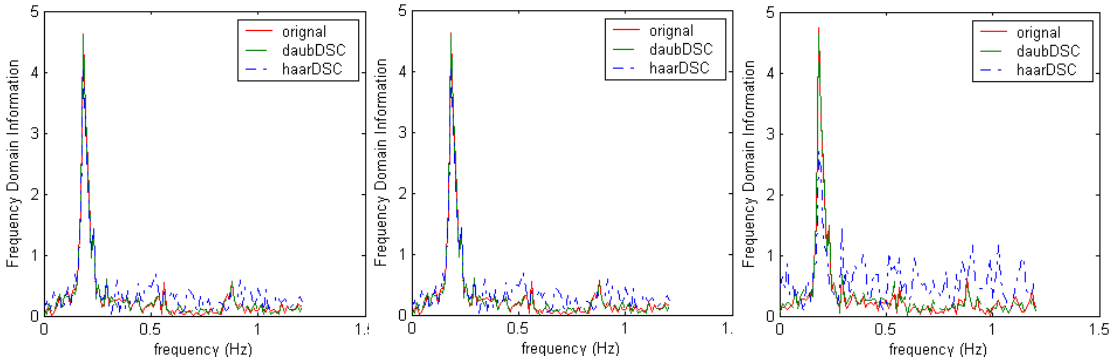
PSNR is related to the properties of (bi)orthogonal wavelets [100]: neglecting the wavelet coefficients with the smallest magnitudes is a good compression approach if one wants to keep a high PSNR. Fig. 6.16 shows that the proposed DSC based compression algorithms can always get the same compression quality as the

MUC based algorithms. Because MUC is a lossless entropy coding algorithm, it also justifies the estimation error is negligible in the proposed algorithms.



(a) Noise: 0dBW (b) Noise: 0.5dBW (c) Noise: 5dBW

Figure 6.17: Comparisons between the original and the restored signals.



(a) Noise: 0dBW (b) Noise: 0.5dBW (c) Noise: 5dBW

Figure 6.18: The frequency domain analysis.

To better illustrate the performance of LSWT based algorithms and compare them with other algorithms, the original and reconstructed signals in time and frequency domains are shown in Fig. 6.17 and Fig. 6.18. Both the Haar-DSC algorithm and Daub-DSC algorithm can achieve favorable performance in the low noise situation. However, when the noise increases slightly, the effect of Haar-DSC is weakened quickly, while Daub-DSC can still achieve fairish effect for the favorable

characters of Daubechies wavelet. In structure health monitoring applications, low frequency information of the sample signal is the important part. As the noise increases, the Haar-DSC algorithm can't restore the original low frequency signal, while Daub-DSC algorithm can still restore the original signal when the noise power increases to $5dBW$. The results validate the feasibility of the proposed algorithms in structure health monitoring applications. It is suitable for vibration and other high frequency correlated data compression.

Computational Complexity

Computational complexity is another important criterion when evaluating the compression algorithm, especially in the WSNs with limited resources. The compression process consists of three steps in the proposed algorithms: LSWT, scalar quantization and DSC or MUC. The computational complexity can be expressed as:

$$C(n) = C_{LSWT} + C_{quan} + p \times C_{DSC} + (1-p) \times C_{MUC} \tag{6.7}$$

where $p = 1/2^n$ and n is the scale level of LSWT.

As analyzed in Section 6.2.2, the complexity of LSWT (C_{LSWT}) is $O(n)$. However, the complexity of Haar LSWT is less than Daubechies D4 LSWT because Haar LSWT only has two steps and counts two filter coefficients, while Daubechies D4 has four steps and counts four filter coefficients.

The computation of the scalar quantization matrix is nontrivial. However, based on the experiments, we found that sample data from the same layer observe the same curve model in each experiment. To improve the efficiency of the algorithms, the quantization matrix is only calculated once, and the same matrix

Table 6.2: Computation time (s) for compressing 4096 sample values

Algorithm	**Wavelet transform**	**Quantization**	**Source coding**	**Total Time**
Haar-DSC	0.0102	0.0070	0.0629	0.0801
Haar-MUC	0.0102	0.0071	0.0611	0.0783
Daub-DSC	0.0450	0.0070	0.1196	0.1717
Daub-MUC	0.0451	0.0070	0.0447	0.0968

is used in all the later quantization processes, so that the quantization complexity (C_{quan}) is reduced to $O(n)$. Results show it does not affect the compression performance under this condition.

The computational complexity of both DSC (C_{DSC}) and MUC (C_{MUC}) is $O(n)$, while DSC is still faster than MUC because the coding complexity of DSC for every symbol is 1, compared to the complexity $2\times \mid x_i \mid$ of MUC for symbol x_i. Overall, the total computational complexity of the proposed algorithms is still $O(n)$.

From the above analysis, we can get the total complexity:

$$C(n) \in O(n) \tag{6.8}$$

The running time of all the algorithms is listed in table 6.2. All the data are measured in Micaz platform associated with an ADXL202E onboard accelerometer.

As the experiments and analysis result demonstrated previously, we can see that the proposed LSWT and DSC based algorithms outperform their peer algorithms on compression ratio with the same data quality and similar computation overhead. Comparing Haar-Wavelet and Daubechies-Wavelet, Haar-Wavelet is more suitable for low-noise conditions because it is simpler and can also achieve good performance. In high-noise conditions, Daubechies-Wavelet, which outperforms Haar-Wavelet significantly, is the better choice.

6.2.6 LSWT-DSC Summary

In this section, we have designed and implemented a MassWare-supported sensor data compression application and proposed a new data compression algorithm that integrates LSWT and DSC for civil infrastructure health monitoring. The MassWare-supported application outperforms static data compression applications since it is able to dynamically select optimal data compression algorithms based on distributed data correlation. To help developers effectively design good adaptation policies, we have analyzed and compared the characters of a set of masslets and their combinations based on compression ratio, data quality, and computational complexity. Therefore, the application can choose suitable software components or change component parameters based on real-time contexts. Experiments also demonstrate that the proposed algorithms can achieve 1:27 to 1:80 compression ratios without weakening the data quality when data redundancy exists in dense WSNs.

Chapter 7

Conclusions

In this dissertation, we have designed and implemented a context-aware reflective middleware (CARM) framework, called MassWare (Mobile Ad-hoc and Sensor Systems middeWare), to improve the reconfiguration efficiency of existing CARM frameworks in MANETs and WSNs. MassWare has two separate middleware frameworks: MassWare-MANET for mobile ad-hoc networks and MassWare-WSN for wireless sensor networks.

MassWare-MANET solves the critical issue of the long reconfiguration time of context-aware reflective middleware to satisfy the stringent real-time requirement of DRE systems. MassWare offers an original structure of multiple component chains to reduce local behavior change time and a novel synchronization protocol using active messages to reduce distributed behavior synchronization time. The key idea behind the protocol is that each application-layer data packet takes an active message header that indexes the correct component-chain of the packet receiver to process the data payload. Therefore, the distributed behavior synchronization time is dramatically reduced by eliminating the operation suspension time and buffer clearance time. To effectively support the new structure and protocol, MassWare-MANET is designed with a layered architecture and provides both component-level and system-level reflection to incorporate standard components, a hierarchical event notification model to evaluate contexts, and a lightweight XML-based script language to describe and manage adaptation policies.

We have established a generic analytical model for fair comparisons of the reconfiguration efficiency of MassWare and peer CARM frameworks: MobiPADS and CARISMA. Besides a theoretical analysis, the system performance of MassWare has been evaluated using benchmark applications. The complete implementation of MassWare and the benchmark applications allows us to test the feasibility and efficiency of MassWare and gain insights into the DRE system design supported by it. The theoretical and experimental results demonstrate that

- the reconfiguration time in traditional adaptive and reflective middleware is reduced by several magnitudes from seconds to hundreds of microseconds,
- the extra costs introduced by the multi-actuator architecture in MassWare are extremely low, and
- the robustness and scalability are improved as well in MassWare when compared with traditional middleware.

Based on MassWare, mission-critical DRE systems, like intelligent vehicle systems and unmanned aircraft systems, will be able to take advantage of future advances in CARM software in a dependable, timely, and cost effective manner.

MassWare-WSN is the first context-aware reflective middleware framework ,to the best of the author's knowledge, which has been implemented in single sensor nodes to support adaptive WSN applications. The overall objective of MassWare-WSN is to improve the reusability and flexibility of WSN applications, which has been achieved by a new component model and a reflective middleware framework. To be lightweight to fit resource-limited sensor nodes while flexible enough to support generic adaptive WSN applications, MassWare-WSN supports software components for efficient reconfiguration and utilizes the active-message-based synchronization protocol to synchronize the behaviors among the base station and reconfigured sensor nodes. It also uses the hierarchical event model to monitor application-interested contexts. MassWare-WSN offers other benefits, including:

- simplifying the task of developing and managing WSN applications;
- facilitating energy-efficient WSN reprogramming;
- providing an efficient synchronization protocol; and
- monitoring node status at runtime.

MassWare-WSN has been implemented in MicaZ sensor nodes and evaluated based on benchmark applications. Results and analysis demonstrate that

- the extra costs introduced by the middleware, like memory consumption and configuration time, are very low with respect to the hardware resources of sensor nodes; and
- the middleware is stable for complex WSN applications.

MassWare-WSN can significantly improve the reusability and reduce the reprogramming cost of WSN applications since an application can be separated into function-independent software components. It also improves the flexibility and adaptability of WSN applications in mobile environments.

MassWare offers an unified, hardware-independent application development model for both MANETs and WSNs to efficiently develop context-aware reflective applications. Application developers only need to provide a standard script file in XML syntax to declare application-required functional components, measurement tool components, and adaptation policies. Massware then uses the script file to construct the application, measure application contextual information, adapt the application behavior to the contexts according to the adaptation policies, and synchronize with peer middleware agents or the base station.

To prove the usability and justify the performance of the proposed middleware framework in real applications, we have implemented MassWare-supported applications in MANET and WSN environments and designed two new algorithms: Local Tree based Geometric Routing (LTGR) and Lifting Scheme Wavelet Transfer and Distributed Source Coding (LSWT-DSC) data compression.

When geometric information is available, a MassWare-supported MANET application can dynamically switch to use LTGR protocol for better performance. LTGR uses a local tree based search algorithm to overcome the shortcomings of

the face routing based protocols. Compared to GPSR, LTGR is more efficient in terms of routing overhead and hop stretch shown by extensive simulation results, e.g. LTGR can reduce the routing overhead by 25 $\sim$ 40% and hop stretch by 30 $\sim$ 50% comparing to GPSR in our simulation scenarios.

In WSNs, a MassWare-supported sensor data compression application can dynamically switch to use LSWT-DSC algorithm for improved compression ratio when the distributed redundancy exists in dense areas (detected by measuring the node density via measurement tools). The LSWT-DSC algorithm integrates L-SWT and DSC and reduces both local and distributed sensor data redundancy, which can achieve 1:27 to 1:80 compression ratios without weakening data quality. The nodes deployed in sparse areas can still use traditional data compression algorithm to reduce local data redundancy only to ensure the data quality.

Although the experimental results are encouraging, there are unexplored issues of MassWare for the future work.

- MassWare extension for stateful applications: The proposed synchronization protocol can be combined with state-machine and model based reconfiguration techniques to improve the reconfiguration efficiency of a state application, like the GSM-Oriented coding application [65].

- MassWare component model: MassWare supports COM components and .NET assemblies so far. It may be useful to extend the component manager to support more component models and components like CORBA components and JAVA Beans etc.

- MassWare deployment in a wide range of sensor platforms: MassWare-WSN is built on top of SOS, which is supported by limited platforms. Migrating the SOS core to other platforms is desired.

- Comprehensive evaluation of MassWare-WSN: Since sensor nodes are power constrained, energy consumption is an important metric of the software based on sensor networks. Thus future experiments about energy efficiency of MassWare are expected.

Bibliography

[1] D.C. Schmidt and S. Huston. *C++ Network Programming: Resolving Complexity with ACE and Patterns*. Addison-Wesley, Reading, MA, 2001. ISBN: 0201604647.

[2] Q.H. Mahmoud. *Middleware for Communications*. Wiley, 2004. ISBN: 9780470862063.

[3] R. Ben-Natan. *CORBA: A Guide to Common Object Request Broker Architecture*. Mcgraw-Hill Inc., 1995. ISBN: 0070054274.

[4] T. Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, 1998. ISBN: 0764580434.

[5] T.L. Thai. *Learning DCOM*. O'Reilly, 1999. ISBN: 1565925815.

[6] S.B. Gordon, C. Geoff, and A. Anders. The design and implementation of open orb 2. *IEEE Distributed Systems Online*, 2(6), 2001.

[7] K. Fabio, R. Manuel, and L. Ping. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'00)*, Aarbus, Denmark, 2000.

[8] L. Capra, W. Emmerich, and C. Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Transaction on Software Engineering*, 29(10):929–945, 2003.

[9] E.P. Kasten and P.K. McKinley. Adaptive java: Refractive and transmutative support for adaptive software. *Technical Report MSU-CSE-01-30*, 2001.

[10] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz. Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments. *Software Engineering for Self-Adaptive Systems*, LNCS 5525:164–182, 2009.

[11] MADAM Consortium. Specification of the madam core architecture and middleware services. *Final report within the 6th Framework Programme*, 2006. Priority 2.3.2.3.

[12] T. Abdelzaher, C.D. Gill, R. Rajkumar, and J.A. Stankovic. Distributed real-time and embedded systems research in the context of geni. *NSF Workshop on Distributed Real-time and Embedded Systems*, 2006.

[13] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli. Context-aware middleware for resource management in the wireless internet. *IEEE Transaction ON Software Engineering*, 23(12), December 2003.

[14] D.C. Schmidt. Adaptive and reflective middleware for distributed real-time and embedded systems. *Lecture Notes in Computer Science*, 2491:282–293, 2002.

[15] J.P. Loyall. Emerging trends in adaptive middleware and its application to distributed real-time embedded systems. *Lecture Notes in Computer Science*, 2855:20–34, 2003.

[16] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, March 2002.

[17] S. Hadim and N. Mohamed. Middleware for wireless sensor networks: A survey. In *Proceedings of The First International Conference on Communication System Software and Middleware (COMSWARE 2006)*, New Delhi, India, January 2006.

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

[19] L. Parolini, N. Tolia, B. Sinopoli, and B.H. Krogh. A cyber-physical systems approach to energy management in data centers. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS '10)*, New York, NY, 2010.

[20] K. Lorincz, D.J. Malan, T.R.F. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, M. Welsh, and S. Moulton. Sensor networks for emergency response: Challenges and opportunities. *IEEE Pervasive Computing*, 3(4):16–23, October-December 2004. doi:10.1109/MPRV.2004.18.

[21] R. Ray. Vehicle infrastructure integration program status. *http://www-nrd.nhtsa.dot.gov/pdf/nrd-01/NRDmtgs/2005Honda/Resendes_VII.pdf*, Accessed on November 15 2008.

[22] M. Gene. Cicas program overview. In *TRB VII / CICAS Workshop*, January 2008.

[23] K. Jiro. Advanced cruise-assist highway system (ahs) technology: System design and proving test facility design. In *Proceedings of the 6th AHS Research Seminar*, June 2002.

[24] A. Yasuyulu. Driving safety support system (dsss) in the aging society. In *Proceedings of Intelligent Transportation Systems*, Tokyo, Japan, 1999.

[25] CAR 2 CAR Communication Consortium. Online document. *http://www.car-2-car.org/fileadmin/downloads/C2C-CC_manifesto_v1.1.pdf*, Accessed on November 15 2008.

[26] A. T.C. Chan and S.N. Chuang. Mobipads: A reflective middleware for context-aware mobile computing. *IEEE Transaction on Software Engineering*, 29(12), 2003.

[27] J. Hu and S. Gorappa. A lightweight component middleware framework for composing distributed, real-time, embedded systems with real-time java. In *Proceedings ACM/IFIP/ USENIX 8th International Middleware Conference (Middleware'07)*, volume 4834, pages 41–59, Newport Beach, CA, November 2007.

[28] D.C. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, and L. DiPalma. Towards adaptive and reflective middleware for network centric combat systems. *CrossTalk C The Journal of Defense Software Engineering*, November 2001.

[29] C.C. Han, R. Kumar, R. Shea, E. Kohler, and M.B. Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys'05)*, pages 163–176, Seattle, WA, June 2005.

[30] D.S. Ruiz. Corba and corba component model. *http://ditec.um.es/ dsevilla/ccm/*, June 2008.

[31] B. Uwe, B. Aurelie, P. Florentin, and S. Etienne. Distributed real-time computing for microcontrollers - the osa+ approach. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 169, Crystal City, VA, April/May 2002.

[32] D.C. Schmidt, D.L. Levine, and S. Mungee. The design of the tao real-time object request broker. *Computer Communications*, 21(4):294–324, 1998.

[33] K. Raymond, D.C. Schmidt, and O. Carlos. Towards highly configurable real-time object request brokers. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'02)*, pages 437–447, Crystal City, VA, April/May 2002.

[34] V. Subramonian, G. Xing, C. Gill, and R. Cytron. The design and performance of special purpose middleware: A sensor networks case study. *Technical Report WUCSE-2003-6*, 2003.

[35] OSEK Comitee. Corba and corba component model. *http://www.osek-vdx.org/*, Accessed on August 2010.

[36] B. Garbinato, R. Guerraoui, and K.R. Mazouni. Distributed programming in garf. In *Proceedings of the ECOOP Workshop on Object-Based Distributed Programming*, pages 225–239, Kaiserslautern, Germany, 1995.

[37] J. McAffer. Meta-level programming with coda. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Aarbus, Denmark, 1993.

[38] G.S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, R. Moreira L. Johnston, N. Parlavantzas, and K. Saikoski. The design and implementation of open orb 2. *IEEE Distributed Systems Online*, 2(6), 2001.

[39] M.A. Miguel. Qos-aware component frameworks. In *the 10th International Workshop on Quality of Service (IWQoS2002)*, Miami Beach, FL, 2002.

[40] F. Kon, F. Costa, G. Blair, and R.H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, 2002.

[41] N. Wang, D.C. Schmidt, M. Kircher, and K. Parameswaran. Towards a reflective middleware framework for qos-enabled corba component model applications. *IEEE Distributed Systems Online*, 2(5), 2001. http://dsonline.computer.org/0105/features/wan0105_print.htm.

[42] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, Jyvaskyla, Finland, 1997.

[43] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An overview of aspectj. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[44] N. Wang, M. Kircher, and D.C. Schmidt. Towards an adaptive and reflective middleware framework for qos-enabled corba component model applications. *IEEE Distributed System Online, Special Issue on Reflective Middleware*, 2003.

[45] R. Schantz, J. Loyall, M. Atighetchi, and P. Pal. Packaging quality of service control behaviors for reuse. In *Processdings of the 5th IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, pages 375–385, Crystal City, VA, 2002.

[46] P.K. Sharma, J.P. Loyall, G.T. Heineman, R.E. Schantz, R. Shapiro, and G. Duzan. Component-based dynamic qos adaptations in distributed real-time and embedded systems. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA '04)*, volume 3291, pages 1208–1224, Larnaca, Cyprus, October 2004.

[47] J.A. Zinky, D.E. Bakken, and R. Schantz. Architectural support for quality of service for corba objects. *Theory and Practice of Object Systems*, 3(1):1–20, 1997.

[48] J.M. Paluska, H. Pham, U. Saif, G. Chau, C. Terman, and S. Ward. Structured decomposition of adaptive applications. In *Proceedings of the 6th IEEE International Conference on Pervasive Computing and Communication (PerCom)*, Hong Kong, China, March 2008.

[49] S. Liu and L. Cheng. Active message oriented adaptation middleware for collaborative applications in heterogeneous environments. In *Proceedings of the 2008 IEEE International Conf. on Communications (ICC 2008)*, pages 1866–1870, Beijing, China, 2008.

[50] J.P. Loyall, D.E. Bakken, R.E. Schantz, J.A. Zinky, D. Karr, R. Vanegas, and K.R. Anderson. Qos aspect languages and their runtime integration. In *Proceedings of the 4th Workshop on Languages, Compilers and Runtime Systems for Scalable Components (LCR98)*, pages 28–30, Pittsburgh, PA, 1998.

[51] R. Vanegas, J.A. Zinky, J.P. Loyall, D. Karr, R.E. Schantz, and D. Bakken. Quo's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Confereence on Distributed Systems Platform and Open Distributed Processing (Middleware 98)*, The Lake District, England, September 1998.

[52] R.E. Schantz, J.A. Zinky, D.A. Karr, D.E. Bakken, J. Megquier, and J.P. Loyall. An object-level gateway supporting integrated-property quality of service. In *Proceedings of the 2nd IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC 99)*, Saint Malo, France, May 1999.

[53] S. Liu and L. Cheng. A context-aware reflective middleware framework for distributed real-time and embedded systems. *Journal of Systems and Software*, 84(2):205–218, 2011.

[54] P. Bonnet, J.E. Gehrke, and P. Seshadri. Towards sensor database systems. In *2nd International Conference on Mobile Data Management (MDM)*, pages 3–14, 2001.

[55] S.R. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122–173, 2005.

[56] C.C. Shen, C. Srisathapornphat, and C. Jaikaeo. Sensor information networking architecture and applications. *IEEE Personal Communications*, 8(4):52–59, August 2001.

[57] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A.L. Murphy, and G.P. Picco. Tinylime: Bridging mobile and sensor networks through middleware. In *the*

3rd IEEE International Conference on Pervasive Computing and Communications (PerCom), pages 61–72, March 2005.

[58] W.B. Heinzelman, A.L. Murphy, H.S. Carvalho, and M. A. Perillo. Middleware to support sensor network applications. *IEEE Network*, 18(1):6–14, 2004.

[59] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, October 2002.

[60] R. Barr, J.C. Bicket, D.S Dantas, B. Du, T.W. Kim, B. Zhou, and E.G. Sirer. On the need for system-level support for ad hoc and sensor networks. *Operating Systems Review, ACM*, 36(2):1–5, April 2002.

[61] T. Liu and M. Martonosi. Impala: A middleware system for managing autonomic. In *Parallel Sensor Systems (PPoPP03)*, San Diego, CA, June 2003.

[62] S.S. Kulkarni and L. Wang. Mnp: Multihop network reprogramming service for sensor networks. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS 2005)*, pages 7–16, Columbus, OH, June 2005.

[63] J.W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, pages 81–94, Baltimore, MD, November 2004.

[64] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *Proceedings of the 1st Annual IEEE ComSoc. Conference on Sensor and Ad Hoc Communication and Networks (SECON)*, pages 25–33, 2004.

[65] J. Zhang and B. H.C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th international conference on Software engineering (ICSE 2006)*, Shanghai, China, May 2006.

[66] A. Ranganathan and R.H. Campbell. An infrastructure for context-awareness based on first order logic. *Journal of Personal and Ubiquitous Computing*, 7(6):353–364, December 2003.

[67] S. Zachariadis, C. Mascolo, and W. Emmerich. The satin component system-a metamodel for engineering adaptable mobile system. *IEEE Transaction on Software Engineering*, 32(10), October 2006.

[68] R. Litiu and A. Prakash. Dacia: A mobile component framework for building adaptive distributed applications. *Technical Report CSE-TR-416-99*, 1999.

[69] M. Palola, M. Jurvansuu, and J. Korva. Breaking down the mobile service response time. In *Proceedings of IEEE International Conference on Networks (ICON 04)*, volume 1, pages 31–34, Singapore, November 2004.

[70] S. Liu and Q. Wang. Marches homepage. *http://marches.cse.lehigh.edu/*, Accessed on May 2011.

[71] N. Shankarany, D.C. Schmidty, X.D. Koutsoukosy, Y. Chenz, and Chenyang Lu. Design and performance evaluation of configurable component middleware for end-to-end adaptation of distributed real-time embedded systems. In *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, pages 291–298, Santorini Island, Greece, May 2007.

[72] S. Liu and L. Cheng. Efficient data compression in wireless sensor networks for civil infrastructure health monitoring. In *Proceedings of the 2006 International Workshop on Wireless Ad-hoc and Sensor Networks*, pages 823–829, New York, NY, 2006.

[73] L. Doherty, B.A. Warnake, B. Baser, and K.S.J. Pister. Energy and performance considerations for smart dust. *International Journal of Parallel and Distributed Systems and Networks*, 4(3), 2001.

[74] J. Broch, D.A. Maltz, D.B. Johnson, Y.C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proceedings of ACM/IEEE Mobile Computing and Network*, pages 85–97, Dallas, TX, October 1998.

[75] E.M. Royer and C.K. Toh. A review of current routing protocols for ad-hoc mobile wireless networks. *IEEE Personal Communications Magazine*, 6(2):46–55, April 1999.

[76] V. Ramasubramanian, Z.J. Haas, and E.G. Sirer. Sharp: A hybrid adaptive routing protocol for mobile ad hoc networks. In *Proceedings of the ACM Symposium on Mobile Ad Hoc Networking and Computing (Mobihoc)*, pages 303–314, Annapolis, MD, June 2003.

[77] B. Karp and H.T. Kung. Gpsr: Greedy perimeter stateless routing for wireless networks. In *Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2000)*, pages 243–254, Boston, MA, August 2000.

[78] F. Kuhn, R. Wattenhofer, and A. Zollinger. Worst-case optimal and average-case efficient geometric ad-hoc routing. In *Proceedings of the Fourth ACM*

International Symposium on Mobile and Ad hoc Networking and Computing (MobiHoc '03), pages 267–278, Annapolis, MD, June 2003.

[79] Y.J. Kim, R. Govindan, B. Karp, and S. Shenker. Geographical routing made practical. In *Proceedings of the 2nd Annual Symposium on Networked Systems Design and Implementation*, pages 217–230, Boston, MA, May 2005.

[80] Y.J. Kim, R. Govindan, B. Karp, and S. Shenker. On the pitfalls of geographic face routing. In *Proceedings of the 2005 Joint Workshop on Foundations of Mobile Computing (DIALM-POMC '05)*, pages 34–43, Cologne, Germany, September 2005.

[81] B. Leong, B. Liskov, and R. Morris. Geographic routing without planarization. In *Proceedings of the 3rd Symposium on Network Systems Design and Implementation (NSDI 2006)*, San Jose, CA, May 2006.

[82] E. Kranakis, H. Singh, and J. Urrutia. Compass routing on geometric networks. In *Proceedings of the 11th Canadian Conference on Computational Geometry*, pages 51–54, Vancouver, Canada, August 1999.

[83] B. Karp. Gpsr website. *http://www.cs.cmu.edu/ bkarp/gpsr/gpsr.html*, Accessed on September 2006.

[84] W. Feng, E. Kaiser, W.C. Feng, and M.L. Baillif. Panoptes: scalable low-power video sensor networking technologies. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 1(2), May 2005.

[85] Q. Ye and L. Cheng. *Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless, and Peer-to-Peer Networks*, chapter Time Synchronization in Wireless Sensor Networks. CRC Press, 2005. ISBN: 0849328322.

[86] S. Ganeriwal, R. Kumar, and M.B. Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of the First ACM International Conference on Embedded Networked Sensor Systems (Sensys'03)*, Los Angeles, CA, November 2003.

[87] D. Salomon. *Data Compression: The Complete Reference, 3rd Edition.* Springer, 2004. ISBN: 0387406972.

[88] S.S. Pradhan, J. Kusuma, and K. Ramchandran. Distributed compression in a dense microsensor network. *IEEE Signal Processing Magazine*, 19:51–60, March 2002.

[89] Z. Xiong, A. Liveris, and S. Cheng. Distributed source coding for sensor networks. *IEEE Signal Processing Magazine*, 21:80–94, September 2004.

[90] Y. Zhang and L. Cheng. Issues in applying wireless sensor networks to health monitoring of large scale civil infrastructure systems. In *ASCE Structure Congress 2005*, New York, NY, April 2005.

[91] T. Cover and J. Thomas. *Elements of Information Theory.* Wiley, 1991. ISBN: 0471062596.

[92] S.S. Pradhan and K Ramchandran. Distributed source coding using syndromes: Designand construction. In *Proceedings of the Data Comression Conference (DCC'99)*, Snowbird, UT, March 1999.

[93] D. Slepian and J.K. Wolf. Noiseless coding of correlated information sources. *IEEE Transaction on Information Theory*, 19(4):471–480, July 1973.

[94] A. Wyner and J. Ziv. The rate-distortion function for source coding with side information at the decoder. *IEEE Transaction on Information Theory*, 22:1–10, January 1976.

[95] T. Berger. *The Information Theory Approach to Communications*, chapter Multiterminal source coding. Springer-Verlag, 1977. ISBN: 3211814841.

[96] S.S. Pradhan and K Ramchandran. Distributed source coding: Symmetric rates and applications to sensor networks. In *Proceedings of the Data Comression Conference (DCC'00)*, Snowbird, UT, March 2000.

[97] J. Chou, D. Petrovic, and K. Ramchandran. A distributed and adaptive signal processing approach to reducing energy consumption in sensor networks. In *Proceedings of the 22nd IEEE International Conference on Computer Communications (Infocom 03)*, San Francisco, CA, March 2003.

[98] R. Puri and K. Ramchandran. Prism: A new 'reversed' multimedia coding paradigm. In *Proceedings of the IEEE International Conference on Image Processing (ICIP 03)*, Barcelona, Spain, September 2003.

[99] N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems*, Baltimore, MD, November 2004.

[100] G. Uytterhoeven, D. Roose, and A. Bultheel. Wavelet transforms using the lifting scheme. *ITA-Wavelets Report WP 1.1*, 1996.

[101] G. Uytterhoeven, F. Van Wulpen, M. Jansen, D. Roose, and A. Bultheel. Waili: Wavelets with integer lifting. *TW Report 262*, 1997.

[102] I. Kaplan. Basic lifting scheme wavelets. *http://www.bearcave.com/misl/misl_tech/wavelets/lifting/basiclift.html*, Accessed on April 2005.

[103] D. Marco and D.L. Neuhoff. Reliability vs. efficiency in distributed source coding for field-gathering sensor networks. In *Information Processing in Sensor Networks (IPSN 04)*, Berkeley, CA, April 2004.

VITA

Shengpu Liu was born in Yueyang, China on April 24, 1979. After completing high school at Yueyang First High School, Yueyang, China in 1997, he attended the University of Science and Technology of China in Hefei, China from 1997-2004. There, he received a Bachelor of Engineering degree and a Master of Engineering degree in the Automation department in 2001 and 2004 respectively. From 2004-2009, he attended Lehigh University in Bethlehem, Pennsylvania to pursue his PH.D. degree under the guidance of Professor Liang Cheng in the Laboratory Of Networking Group (LONGLAB). After finishing his major research work, he changed to be a part-time student in 2009 to get more industry experience. From 2009 till now, he has been a software engineer in the Epic Systems Corporation.